MAX-PLANCK-INSTITUT FÜR INFORMATIK

Difference Unification

David Basin
Toby Walsh

MPI–I–92–247 November 1992

Im Stadtwald
W 6600 Saarbrücken
Germany

Authors' Addresses

David Basin,
Max-Planck-Institut für Informatik,
Im Stadtwald,
Saarbrücken, Germany.
basin@mpi-sb.mpg.de

Toby Walsh
Department of AI,
University of Edinburgh,
80 South Bridge,
Edinburgh, EH1 1HN, Scotland
tw@aisb.ed.ac.uk

Publication Notes

An abridged version of this report has been submitted to IJCAI-93.

Acknowledgements

David Basin was supported by the German Ministry for Research and Technology (BMFT) under grant ITS 9102. Responsibility for the contents of this publication lies with the authors. We thank the Edinburgh Mathematical Reasoning Group for their encouragement and criticism. In particular, we thank Alan Bundy, Andrew Ireland, Séan Matthews, and Andreas Tönne.

Abstract

We extend previous work on difference identification and reduction as a technique for automated reasoning. We generalize unification so that terms are made equal not only by finding substitutions for variables but also by hiding term structure. This annotation of structural differences serves to direct rippling, a kind of rewriting designed to remove structural differences in a controlled way. On the technical side, we give a rule-based algorithm for difference unification, and analyze its correctness, completeness, and complexity. On the practical side, we present a novel search strategy (called left-first search) for applying these rules in an efficient way. Finally, we show how this algorithm can be used in new ways to direct rippling and how it can play an important role in theorem proving and other kinds of automated reasoning.

1 Introduction

Heuristics for judging similarity between terms and subsequently reducing differences have been applied to automated deduction since the 1950s when Newell, Shaw, and Simon built their "logic machine"[NSS63] for a propositional calculus. Their intent was to simulate the behavior of a human on the same task. More recently, in resolution theorem proving, a similar theme of difference identification and reduction appears in [BS88, Dig85, Mor69]. In this work a partial unification results in a special kind of resolution step (E or RUE-resolution) where the failure to unify completely produces new inequalities that represent the differences between the two terms. This leads to a controlled application of equality reasoning where paramodulation is used only when needed. The intention was not to design a human oriented problem solving strategy, but rather, to use difference identification and reduction as a means of reordering a potentially infinite search space.

Here we report on research sharing both these cognitive and pragmatic aims. We have developed a general procedure called *difference unification* for identifying differences between two terms or formulas. Difference unification extends unification in that it decides if terms are syntactically equal not only by giving assignments for variables but also by computing what incompatible term structure must be removed. This incompatible term structure, called *wave-fronts*, is marked by sets of *annotations* which are used to direct a special kind of rewriting called *rippling*[1]; rippling seeks to reduce the differences between the terms by moving the wave-fronts "out of the way" while not disturbing the unannotated parts of the terms.

This research is the outgrowth of previous work at Edinburgh in inductive theorem proving. There Bundy [Bun88, BSvH$^+$92] suggested that in proofs by mathematical induction, the induction conclusion could be proven from the induction hypothesis by rippling on the induction conclusion. Rippling has been employed in the OYSTER/CLAM prover. A similar kind of rewriting was developed independently by Hutter [Hut90], from ideas in [Bun88], and employed in the INKA system. Both systems have enjoyed a high degree of success stemming from several desirable properties of rippling. These include (see [BSvH$^+$92]) that rippling involves very little search and rippling always terminates since wave-fronts are only moved in some desired (well-founded) way — usually to the top of the term.

Motivated by a desire to apply rippling outside of inductive theorem proving, in [BW92] we introduced *difference matching* which extends matching to annotate the matched term so it can be rewritten using rippling. We list there, as well as in [WNB92] several applications of this idea. In this report we take another step forward. Our contributions are several fold. First we extend difference matching to difference unification whereby substitutions and annotations are returned for both terms. The rule based algorithm we give uses conventional unification in a transparent way whereby other additions to unification, such as equations or higher order patterns, can be easily made. We prove the algorithm given is both sound and complete with respect to its specification. Second, unlike difference matching, difference unification can return a large number of matches which we are not interested in. For example, there may be exponentially many ways to annotate two identical terms. Hence, we demarcate two restricted classes of useful answers (which we call *strongly* and *weakly* minimal). Further, we give a novel search strategy (a meta-interpreter) that finds answers in these classes with minimal search. We have written such an interpreter and report on this as well. Third, we give a thorough analysis of the complexity of difference unification and subproblems. Finally, we provide examples of how difference unification can be used. In doing so, we present a new paradigm for theorem proving/problem solving whereby proof proceeds by alternating between annotating differences and reducing them. This combination is different from previous work combining rippling and difference matching since here successful rippling does not guarantee successful rewriting of one term with another; rather, it must be seen as one step, in possibly many, of difference reduction. This, along with differences between traditional rewrite based theorem proving, is developed further in the next section.

[1] The name rippling comes from *rippling-out* a term coined by Aubin [Aub75], a student of Boyer and Moore's, during his study of generalization in inductive theorem proving. It is based on an observation that one can iteratively unfold (as in [BD77]) recursive functions in the induction conclusion, preserving the structure of the induction hypothesis while unfolding.

2 Applications

2.1 Normalization

We begin with a simple example that both introduces notation and illustrates how difference unification can be used to apply rippling in a new way: as an iterative difference reduction technique. In rippling's original role in inductive theorem proving, successfully rippling the goal always allows use of the induction hypothesis. More particularly, in an inductive proof the induction conclusion is an image of the induction hypotheses except for the appearance of certain function symbols applied to the induction variable in the conclusion. The rest of the induction conclusion, which is an exact image of the induction hypothesis, is called the *skeleton*. The function symbols that must be moved are the wave-fronts. For example, if we wish to prove $p(x)$ for all natural numbers, we assume $p(n)$ and attempt to show $p(s(n))$. The hypothesis and the conclusion are identical except for the successor function $s(.)$ applied to the induction variable n. We mark this wave-front by placing a box around it and underlining the subterm contained in the skeleton, $p(\boxed{s(\underline{n})})$. Rippling then applies just those rewrite rules, called *wave-rules*, which move the difference out of the way leaving behind the skeleton. In their simplest form, wave-rules are rewrite rules of the form $\alpha(\boxed{\beta(\underline{\gamma})}) \Rightarrow \boxed{\rho(\underline{\alpha(\gamma)})}$. By design, the skeleton $\alpha(\gamma)$ remains unaltered by their application. If rippling succeeds then the conclusion $p(\boxed{s(\underline{n})})$ is rewritten using wave-rules into some function of $p(n)$; that is, into $\boxed{f(\underline{p(n)})}$ (f may be the identity). At this point we can call upon the induction hypothesis.

An analogous situation occurs in difference matching. If we can match two terms, annotating one with wave-fronts, then successful rippling allows rewriting one to the other. However, this fails with difference unification as both terms are annotated. For example, consider the associative (infix) function symbol $+$. The following are wave-rules.[2]

$$\boxed{(X + \underline{Y})} + Z \rightarrow \boxed{X + \underline{(Y + Z)}} \tag{1}$$

$$X + \boxed{(\underline{Y} + Z)} \rightarrow \boxed{\underline{(X + Y)} + Z} \tag{2}$$

As previously noted, rippling terminates because wave-fronts in the rewrite rules must match those in the rewritten term and these are only moved in some well-founded direction. We may therefore rewrite with the associativity of $+$ in both directions. Consider proving

$$((a + b) + c) + d = a + (b + (c + d)).$$

If we difference unify the left hand side of this equation with the right, there are 10 annotated answers corresponding to the 6 ways of selecting any 2 constants from the 2 terms and 4 ways of selecting any one. In general, we prefer only those with minimal amounts of annotation. Furthermore, as wave-rules only exist to ripple these minimal annotations, rippling would not find proofs for the others. Picking minimal annotations (formally defined in §3) narrows the choice to 2:

$$\boxed{\underline{((a + b)} + c) + d} = a + \boxed{(\underline{b} + (c + d))} \tag{3}$$

$$\boxed{(a + b) + \underline{c)} + d} = \boxed{a + (b + \underline{(c + d)})} \tag{4}$$

Both of these will lead to proofs by rippling (the first giving a left associative normal form, the second giving a right). In what follows we concentrate on the first. The left hand side of this equation is completely

[2] In what follows capital letters represent variables and lower case letters constants and bound variables.

rippled-out: no more wave-rules need (or can) be applied since the wave-fronts are already outermost. The right hand side ripples with (2) yielding

$$\boxed{(\underline{(a+b)}+c)+d} = \boxed{\underline{(a+b)}+(c+d)}$$

and now both terms are rippled-out. Though rippling is done, we have not succeeded in proving the terms equal since the wave-fronts themselves differ.

One might conclude that rippling has not accomplished anything but that would be false. It has reduced the "inner difference" between these terms: each now contain a copy of the previous skeleton $a+b$ intact. Difference unifying $((a+b)+c)+d$ against $(a+b)+(c+d)$ reveals this. There are 12 annotations in total, but only 3 are minimal, and only one of these can be rippled:

$$\boxed{\underline{((a+b)+c)}+d} = (a+b)+\boxed{(\underline{c}+d)}.$$

We have made progress since these terms have the larger skeleton $(a+b)+c$. Again the right hand side is rippled-out; rippling on the left with (2) yields the right hand side, so we are done. This example illustrates a general phenomenon: iterating difference unification and rippling successively decreases the difference between two terms.

This combination can be very effective. In associative reasoning each iteration of difference unification and rippling increases the skeleton and hence terminates successfully. Of course, exhaustive application of one of the associativity rules would also suffice, but there are advantages in using difference unification and rippling. To begin with, one needn't completely normalize terms, rippling proceeds only as far as is required to reduce the difference. Moreover, as both left and right associativity may be used, fewer rewrite steps may be required. More significantly, there are theories where we need both (See §2.3 for an example); here normalization would loop. The combination of difference unification and rippling is often an effective heuristic in theories where rewrite based procedures do not exist; the next two examples, aside from being more general, illustrate this.

2.2 Series

Difference unification and rippling have proved also very useful in summing series. Consider, for example, the problem of finding a closed form sum for

$$\sum_{j=0}^{m}\sum_{k=0}^{n} k \times \frac{1}{s(j) \times s(s(j))}$$

using the standard result (such results are computed automatically in [WNB92])

$$\sum_{i=0}^{N} \frac{1}{s(i)} - \frac{1}{s(s(i))} \quad = \quad 1 - \frac{1}{s(s(N))}. \tag{5}$$

We encode the problem of finding a closed form sum as the task of proving a theorem of the form,

$$\exists S \;.\; \sum_{j=0}^{m}\sum_{k=0}^{n} k \times \frac{1}{s(j) \times s(s(j))} \quad = \quad S$$

where the existential witness S is restricted to be in closed form. To prove this theorem, we first eliminate the existential quantifier. The *standard form* method [WNB92] then difference unifies the dequantified goal

with (5) giving the minimal annotations

$$\sum_{i=0}^{N} \boxed{\frac{1}{\underline{s(i)}} - \frac{1}{s(s(i))}} = 1 - \frac{1}{s(s(N))} \quad \vdash \quad \sum_{j=0}^{m} \boxed{\sum_{k=0}^{n} k \times \frac{1}{\boxed{\underline{s(j)} \times s(s(j))}}} = S.$$

To ripple these differences away we use the wave-rules:

$$\sum_{j=A}^{B} \boxed{\sum_{k=C}^{D} \underline{U}} \quad \rightarrow \quad \boxed{\sum_{k=C}^{D} \sum_{\underline{j=A}}^{B} U} \tag{6}$$

$$\sum_{j=A}^{B} \boxed{C \times \underline{U}} \quad \rightarrow \quad \boxed{C \times \sum_{\underline{j=A}}^{B} U} \tag{7}$$

$$\frac{1}{\boxed{\underline{s(U)} \times s(s(U))}} \quad \rightarrow \quad \boxed{\frac{1}{\underline{s(U)}} - \frac{1}{s(s(U))}} \tag{8}$$

where C and D are constant with respect to j. Note that (6) could not be used in a procedure based on exhaustive rewriting since, like associativity when used in both directions, it would loop.

The standard form method first applies wave-rule (6) to the goal dividing its wave-front into two,

$$\sum_{i=0}^{N} \boxed{\frac{1}{\underline{s(i)}} - \frac{1}{s(s(i))}} = 1 - \frac{1}{s(s(N))} \quad \vdash \quad \boxed{\sum_{k=0}^{n} \sum_{\underline{j=0}}^{m} \boxed{k \times \frac{1}{\boxed{\underline{s(j)} \times s(s(j))}}}} = S$$

then wave-rule (7),

$$\sum_{i=0}^{N} \boxed{\frac{1}{\underline{s(i)}} - \frac{1}{s(s(i))}} = 1 - \frac{1}{s(s(N))} \quad \vdash \quad \boxed{\sum_{k=0}^{n} k \times \sum_{\underline{j=0}}^{m} \frac{1}{\boxed{\underline{s(j)} \times s(s(j))}}} = S$$

and finally (8), after which rippling no longer applies,

$$\sum_{i=0}^{N} \boxed{\frac{1}{\underline{s(i)}} - \frac{1}{s(s(i))}} = 1 - \frac{1}{s(s(N))} \quad \vdash \quad \boxed{\sum_{k=0}^{n} k \times \sum_{\underline{j=0}}^{m} \boxed{\frac{1}{\underline{s(j)}} - \frac{1}{s(s(j))}}} = S.$$

We therefore re-difference unify to give, as with the associativity example, a larger skeleton,

$$\sum_{i=0}^{N} \frac{1}{s(i)} - \frac{1}{s(s(i))} = 1 - \frac{1}{s(s(N))} \quad \vdash \quad \boxed{\sum_{k=0}^{n} k \times \sum_{\underline{j=0}}^{m} \frac{1}{s(j)} - \frac{1}{s(s(j))}} = S.$$

Rippling, though unable to move the differences up completely, has reduced the inner difference. Indeed, the difference has been so reduced that we can now substitute with the standard result,

$$\sum_{i=0}^{N} \frac{1}{s(i)} - \frac{1}{s(s(i))} = 1 - \frac{1}{s(s(N))} \quad \vdash \quad \sum_{k=0}^{n} k \times (1 - \frac{1}{s(s(m))}) = S.$$

The *standard form* method now difference unifies against the standard result for the sum of the first n integers, and ripples with (7) to complete the proof.

2.3 Proof by Consistency

Our final example addresses the integration with so called *proof by consistency* techniques, e.g. [Bac88, Fri86].[3] Proofs are sets of equations and are transformed by applications of rules which may be roughly classified as deduction and simplification. Deduction rules generate new equations to be proven via superposition (critical pair formation) and simplification rewrites equations using a set of (ground) confluent rewrite rules $\mathcal{R}$, lemmas $\mathcal{L}$, and the original conjectured equations. In the presentation of such proof procedures, proof search strategies are usually not given. In implementations, e.g., Unicom [Gra90] considerable provision must be made for human guidance, especially when lemmas are used to simplify equations.

Here we show that a strategy based on difference unification and rippling can often guide such proofs. This is based on the fact that recursive equations and many lemmas can be parsed as wave-rules. As an example of the kind of control problems, consider a proof of:

$$(y + z) \times x = y \times x + z \times x \tag{9}$$

from the rewrite rules

$$R = \{0 + X \to X,\ s(X) + Y \to s(X + Y),\ 0 \times X \to 0,\ s(X) \times Y \to Y + X \times Y\}.$$

Superimposing the base and step case of addition against the goal yields the following critical pairs.

$$\langle z \times x, 0 \times x + z \times x \rangle \tag{10}$$

$$\langle s(y + z) \times x, s(y) \times x + z \times x \rangle \tag{11}$$

The first pair conflates after rewriting with the base-case of $\times$ and $+$. The second simplifies on the left to $x + (y + z) \times x$ which can be further reduced using (9) to $x + (y \times x + z \times x)$. On the right we can simplify to $(x + y \times x) + z \times x$ which is irreducible. Since the two sides are not identical, the proof has failed.

Since proof by consistency formalisms allow application of lemmas this failed proof can be completed using the associativity of $+$. But how do we control such rewrite rules? Unlike the application of rewrite rules from $\mathcal{R}$, application of lemmas won't necessarily terminate.

Here we suggest that difference unification and rippling provide a suitable strategy for controlling simplification based on the following observations. First, definitions in $\mathcal{R}$ can usually be parsed as wave-rules using difference unification where variables in equations are treated as constants during the difference unification.[4] Primitive recursive definitions can always be parsed as wave-rules. For example, if we have a primitive recursive definition

$$f(s(X), Y) \to h(X, Y, f(X, Y))$$

we can difference unify the two sides yielding the minimal annotation

$$f(\boxed{s(\underline{X})}, Y) \to \boxed{h(X, Y, \underline{f(X, Y)})}.$$

There is one other minimal annotation with skeleton Y but this is not a wave-rule since it does not move wave-fronts up. In our example, the recursive definitions in the rewrite set $\mathcal{R}$ are parsed as the wave rules $\boxed{s(\underline{X})} + Y \to \boxed{s(\underline{X + Y})}$ and $\boxed{s(\underline{X})} \times Y \to \boxed{Y + \underline{X \times Y}}$. Second, new equations generated by deduction steps may be annotated by difference unifying them with the original goal that was superimposed upon.

[3] We assume familiarity with such techniques since a detailed presentation is beyond the scope of this paper. [BBH92] contains a more in depth discussion of these techniques and their relationship with rippling.

[4] For proving theorems using explicit induction, CLAM has an algorithm for wave-rule parsing. Unfortunately, we have recently discovered that this is both incomplete and unsound. As well as failing to return all wave-rules, it returns some annotations which are ill-formed. We are currently re-implementing CLAM's wave-rule parser with an algorithm based on *ground* difference unification and termination analysis for orienting the wave-rules.

When recursive definitions are superimposed, difference unification succeeds if the definitions form wave-rules since rewriting with wave-rules is skeleton preserving. In our example, difference unifying (11) with (9) annotates the critical pair as

$$\langle \boxed{s(\underline{y+z})} \times x, \boxed{s(\underline{y})} \times x + z \times x \rangle.$$

Finally, lemmas in $\mathcal{L}$ can often be parsed as wave-rules via difference unification, e.g., the associativity of $+$ in our example.

To complete our example, we use the annotated rules in $\mathcal{R}$ and lemmas in $\mathcal{L}$ to ripple the annotated critical pair. Rippling with the recursive definition of $\times$ on both sides yields

$$\boxed{x + \underline{(y+z) \times x}} = \boxed{(x + \underline{y \times x})} + z \times x$$

Now we ripple with (1) on the right hand side,

$$\boxed{x + \underline{(y+z) \times x}} = \boxed{x + \underline{(y \times x + z \times x)}},$$

and both sides of the equality are fully rippled. We can either complete the proof by rewriting the left hand side with (9) (analogous to what Boyer and Moore call "cross fertilization" in explicit induction) or by removing wave-fronts all together with the wave-rule $\boxed{X + \underline{Y}} = \boxed{X + \underline{Z}} \rightarrow Y = Z$ which yields (9) (analogous to "fertilization" in explicit induction).

We have used [BBH92] as a source of theorems to test the above ideas. The 20 theorems there contain two types of operators: propositional connectives (like $\Rightarrow$) and recursively defined functions. Of these, 6 use propositional connectives which do not form wave-rules (e.g., $(false \Rightarrow X) = true$) and hence cannot be proven using rippling alone. This is not surprising for rippling is a syntactic heuristic; theorem provers based on rippling (like Clam and INKA) prove propositional theorems by other means. The remaining 14 theorems only use functions defined by primitive recursion (on the first suitable argument) whose definitions are parsed by our difference unifier as wave-rules. For these 14 theorems, we generated critical pairs by considering superpositions at all complete sets of positions (even when this generated unnecessary critical pairs). All 14 are then proved using rippling on the rule set augmented in some cases with additional lemmas that form wave-rules. The following table contains a representative set of examples.

Theorem	Lemmas (if any) used as wave-rules
$rev(rev(X)) = X$	none
$eq(double(halff(X), X) = even(X)$	none
$halff(double(X)) = X$	none
$rev(app(X,Y)) = app(rev(Y), rev(X))$	$\boxed{app(\underline{X}, [])} = X, app(\boxed{app(X, \underline{Y})}, Z) = \boxed{app(X, \underline{app(Y, Z)})}$
$len(rev(X)) = len(X)$	$len(\boxed{app(\underline{X}, [Y])}) = \boxed{s(\underline{len(X)})}$

3 Specification

To specify difference unification we must be more precise about the representation of annotations. As in [BW92] annotations are represented in a normal form in which every wavefront has an immediate subterm deleted (i.e. all wavefronts are one functor thick). In addition, rather than superimposing a particular representation on terms (like the "box-and-hole" notation used earlier in this paper), annotations will be abstracted out and represented separately; this makes it much easier to specify and describe a difference unification algorithm (although we will continue to use the "box-and-hole" for aiding visualisation of annotation sets). Annotations will therefore be represented by the set of positions of the wave-holes; as the

wavefronts are always one functor thick, the position of the wave-hole uniquely determines the wavefront. Positions are defined recursively as follows: the set of positions in the term t is $Pos(t)$ where,

$$Pos(f(s_1, ..., s_n)) \quad = \quad \{\Lambda\} \cup \{i.p \mid 1 \leq i \leq n \ \wedge \ p \in Pos(s_i)\}$$

The subterm of a term t at position p is t/p where:

$$\begin{aligned} t/\Lambda &= t \\ f(s_1, ..., s_n)/i.p &= s_i/p \end{aligned}$$

For example, annotations for $f(g(\boxed{f(\underline{a}, b)}), \boxed{g(\underline{b})})$ are given by the set $\{1.1.1.\Lambda, 2.1.\Lambda\}$. In what follows we shall only work with sets of annotations that are *well-formed* with respect to given terms. That is the addresses refer only to positions inside the expression tree, and no two addresses differ only in the final address position (which would correspond to a wave-front with two wave-holes).

A few remaining auxiliary definitions are needed. By recursion on terms it is simple to define a function $skeleton(t, A_t)$ which takes a term t and a set of annotations for that term A_t, and returns the unannotated part of the term. For example, the skeleton of $f(g(\boxed{f(\underline{a}, b)}), \boxed{g(\underline{b})})$ is $f(g(a), b)$. In defining difference unification we use a position consing function $p@i$ that adds addresses to the *end* of a position.

$$\begin{aligned} \Lambda@i &= i.\Lambda \\ (p.q)@i &= p.(q@i) \end{aligned}$$

We say that q *extends* p iff $q = p@i$, or q extends some r and r extends p.

Difference unification is a relation written $du(s, t, A_s, A_t, \sigma)$ that satisfies the property

$$\sigma(skeleton(s, A_s)) \quad = \quad \sigma(skeleton(t, A_t)),$$

where σ is a most general unifier. Note that this is rather different from the much harder homomorphic embedding problem [NS87] where the substitution is applied before deleting function symbols possibly including those introduced by the substitution.

As in the examples, we often demand a minimality condition on the annotations. Annotations are minimal if they are the least amount of annotation necessary to make terms unifiable (just as a most general unifier is the least amount of substitution needed to make the terms identical). There is a choice though concerning whether annotations are minimal with respect to a given substitution, or with respect to all possible substitutions. This choice has important consequences both for applications of difference unification, and as we will later demonstrate, for the algorithm's search properties.

Definition 1 (weak minimality) *A_s and A_t are weakly minimal annotations of s and t and σ iff $\neg\exists A'_s \subset A_s$, $A'_t \subset A_t$ with $\sigma(skeleton(s, A'_s)) = \sigma(skeleton(t, A'_t))$*

Definition 2 (strong minimality) *A_s and A_t are strongly minimal annotations of s and t iff $\neg\exists A'_s, A'_t$ with $(A'_s \subset A_s, A'_t \subseteq A_t)$ or $(A'_s \subseteq A_s, A'_t \subset A_t)$ and $skeleton(s, A'_s)$ unifiable with $skeleton(t, A'_t)$*

For example, $\langle \boxed{f(\underline{X})}, f(a) \rangle$ is weakly minimal with substitution $X \mapsto f(a)$ but not strongly minimal, whilst $\langle \boxed{f(\underline{X})}, \boxed{f(\underline{Y})} \rangle$ is neither weakly minimal nor strongly minimal (the only strongly minimal difference unification is no annotation). A simple consequence of the definitions is that strongly minimal solutions are also weakly minimal and in the ground case (e.g., wave-rule parsing) they coincide. Note that all difference matches (variables and annotations only on one of the two terms) are weakly minimal. As we illustrated in the applications, we can often avoid many useless difference unifiers by restricting our attention just to minimal difference unifiers.

4 Algorithm

As is common practice in the unification community (e.g., [JK91]), we give an algorithm for difference unification by means of transformation rules and (in the next section) a search strategy for applying these rules. To difference unify s with t, we reduce the quadruple

$$\langle \{s = t \,/\, \Lambda, \Lambda\}, \{\}, \{\}, \{\} \rangle$$

to

$$\langle \{\}, \sigma, A_s, A_t \rangle$$

where σ is a set of substitutions, and A_s and A_t are the annotations of s and t. The notation "/" marks positions of terms within s and t; these are used to record annotation addresses.

The rules for difference unification are given below. The predicates $valid_s(p)$ and $valid_t(p)$ are defined relative to the input terms s and t and are defined as $p \in Pos(s)$ and $p \in Pos(t)$ respectively.

DELETE	$\langle S \cup \{s = s \,/\, p, q\}, \sigma, A_s, A_t \rangle$	$\Rightarrow$	$\langle S, \sigma, A_s, A_t \rangle$
DECOMPOSE	$\langle S \cup \{f(s_1, ..., s_n) = f(t_1, ..., t_n) \,/\, p, q\}, \sigma, A_s, A_t \rangle$	$\Rightarrow$	$\langle S \cup \{s_i = t_i \,/\, p@i, q@i \mid 1 \leq i \leq n\}, \sigma, A_s, A_t \rangle$
$\text{ELIMINATE}_\text{L}$	$\langle S \cup \{X = t \,/\, p, q\}, \sigma, A_s, A_t \rangle$	$\Rightarrow$	$\langle S[X \mapsto t], \sigma \circ t/X, A_s, A_t \rangle$ if $\neg occurs(X, t)$
$\text{ELIMINATE}_\text{R}$	$\langle S \cup \{s = X \,/\, p, q\}, \sigma, A_s, A_t \rangle$	$\Rightarrow$	$\langle S[X \mapsto s], \sigma \circ s/X, A_s, A_t \rangle$ if $\neg occurs(X, s)$
IMITATE_L	$\langle S \cup \{X = f(t_1, ..., t_n) \,/\, p, q\}, \sigma, A_s, A_t \rangle$	$\Rightarrow$	$\langle S[X \mapsto f(X_1, ..., X_n)] \cup \{X_i = t_i \,/\, p@i, q@i \mid 1 \leq i \leq n\}, \sigma \circ f(X_1, ..., X_n)/X, A_s, A_t, \rangle$ if $\forall i \in [1, n]\ valid_t(q@i)$
IMITATE_R	$\langle S \cup \{f(s_1, ..., s_n) = X \,/\, p, q\}, \sigma, A_s, A_t \rangle$	$\Rightarrow$	$\langle S[X \mapsto f(X_1, ..., X_n)] \cup \{s_i = X_i \,/\, p@i, q@i \mid 1 \leq i \leq n\}, \sigma \circ f(X_1, ..., X_n)/X, A_s, A_t, \rangle$ if $\forall i \in [1, n]\ valid_s(p@i)$
HIDE_L	$\langle S \cup \{f(s_1, ..., s_n) = t \,/\, p, q\}, \sigma, A_s, A_t \rangle$	$\Rightarrow$	$\langle S \cup \{s_i = t \,/\, p@i, q\}, \sigma, A_s \cup \{p@i\}, A_t \rangle$ if $valid_s(p@i)$
HIDE_R	$\langle S \cup \{s = f(t_1, ..., t_n) \,/\, p, q\}, \sigma, A_s, A_t \rangle$	$\Rightarrow$	$\langle S \cup \{s = t_i \,/\, p, q@i\}, \sigma, A_s, A_t \cup \{q@i\} \rangle$ if $valid_t(q@i)$

DUNIFY : transformation rules for difference unifying s and t

---

These rules need to be applied non-deterministically. For example, in difference unifying $f(x, a)$ with $f(g(a), x)$ if we apply DECOMPOSE and $\text{ELIMINATE}_\text{L}$ committing to $x \mapsto g(a)$, we fail to get a solution. The rules are closely related to rules for matching, difference matching and unification. Matching is implemented by DELETE, DECOMPOSE, $\text{ELIMINATE}_\text{L}$; difference matching is implemented by DELETE, DECOMPOSE, $\text{ELIMINATE}_\text{L}$, HIDE_L; and unification is implemented by DELETE, DECOMPOSE, $\text{ELIMINATE}_\text{L}$, $\text{ELIMINATE}_\text{R}$. Note that unification can also use rules, often called CONFLICT and CHECK, which cause unification to fail immediately when the outermost functions disagree or an occurs check error happens. In difference unification we cannot use such rules and must fail only when the search space has been exhaustively traversed. The IMITATE_L and IMITATE_R rules can also be used in unification, although $\text{ELIMINATE}_\text{L}$ and $\text{ELIMINATE}_\text{R}$ will always work (more efficiently) in their place without losing the completeness of unification. Difference unification does, however, need IMITATE_L and IMITATE_R for completeness: consider difference unifying X and $f(g(a))$, returning the substitution $X \mapsto f(a)$ and the annotations $f(\boxed{g(\underline{a})})$. The DUNIFY rules can be seen as the merging of the rules for unification and the rules DELETE, DECOMPOSE, HIDE_L, and HIDE_R which add an arbitrary annotation; since the rules for recursing through the term structure are common to both these rule sets, their merging is more efficient than a naive "generate annotations" and "unify skeletons".

These rules have been implemented in Prolog and the following is a table of the results of difference unifying $(X + Y) + Z$ with $X + (Y + Z)$ and a trace of the execution of the first result.

No.	s	t	σ
1	$\underline{(X+Y)+Z}$	X + $\boxed{(\underline{Y}+Z)}$	{}
2	$\boxed{(X+\underline{Y})}$ + Z	$X+\underline{(Y+Z)}$	{}
3	$\boxed{(\underline{X}+Y)}$ + Z	X + $\boxed{(Y+\underline{Z})}$	{}
4	$\boxed{(\underline{X}+Y)+Z}$	$\boxed{\underline{X}+(Y+Z)}$	{}
5	$\boxed{(X+\underline{Y})+Z}$	$\boxed{X+(\underline{Y}+Z)}$	{}
6	$\boxed{(X+Y)+\underline{Z}}$	$\boxed{X+(Y+\underline{Z})}$	{}
7	$\boxed{\boxed{(\underline{X}+Y)+Z}}$	$\boxed{X+(\underline{Y}+Z)}$	{Y+Z/X}
8	$\boxed{\boxed{(\underline{X}+\underline{Y})+Z}}$	$\boxed{X+(Y+\underline{Z})}$	{X+Y/Z}

$$\langle\{(X+Y)+Z = X+(Y+Z)\,/\,\Lambda,\Lambda\},\{\},\{\},\{\}\rangle$$
$$\Downarrow \text{HIDE}_{\text{L}}$$
$$\langle\{X+Y = X+(Y+Z)\,/\,1.\Lambda,\Lambda\},\{1.\Lambda\},\{\},\{\}\rangle$$
$$\Downarrow \text{DECOMPOSE}$$
$$\langle\{X = X)\,/\,1.1.\Lambda,1.\Lambda,Y = Y+Z\,/\,1.2.\Lambda,2.\Lambda\},\{1.\Lambda\},\{\},\{\}\rangle$$
$$\Downarrow \text{DELETE}$$
$$\langle\{Y = Y+Z\,/\,1.2.\Lambda,2.\Lambda\},\{1.\Lambda\},\{\},\{\}\rangle$$
$$\Downarrow \text{HIDE}_{\text{R}}$$
$$\langle\{Y = Y\,/\,1.2.\Lambda,2.1.\Lambda\},\{1.\Lambda\},\{2.1.\Lambda\},\{\}\rangle$$
$$\Downarrow \text{DELETE}$$
$$\langle\{\},\{1.\Lambda\},\{2.1.\Lambda\},\{\}\rangle$$

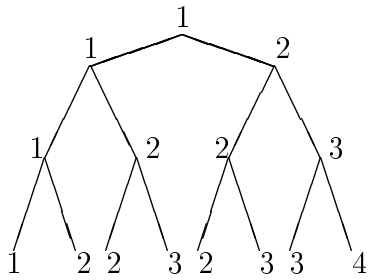
The first three annotations are strongly minimal and give the only wave-rules (oriented appropriately). The fourth, fifth and sixth annotations are neither weakly nor strongly minimal. The last two annotations are weakly but not strongly minimal. This once again demonstrates that difference unification is not unitary, even when restricted to strong or weak minimality.

5 Left-first Search

The transformation rules, when exhaustively and non-deterministically applied, generate all possible difference unifications, not just those that are weakly or strongly minimal. We have therefore implemented a search algorithm (i.e., a meta-interpreter) for traversing the space defined by these rules so that we are guaranteed to encounter just the weakly or strongly minimal difference unifications. In the strongly minimal case, potentially an exponential amount of search is saved.

We first describe the structure of the search space. Nodes correspond to the quadruples giving the current state. Arcs to the left result from applying one of the unification rules: DELETE, DECOMPOSE, IMITATE$_{\text{L}}$, IMITATE$_{\text{R}}$, ELIMINATE$_{\text{L}}$, and ELIMINATE$_{\text{R}}$. Arcs to the right result from applying a hiding rule: HIDE$_{\text{L}}$ and HIDE$_{\text{R}}$. The key to returning minimal difference unifications is observing that a non-minimal difference unification uses more applications of the hiding rules than a minimal one, though it may use a greater, lesser or equal number of unification rules. Thus, in searching the tree we want to minimize right arcs since each adds more annotation. We call a search algorithm which does this *left-first* search. At the $n+1$-th ply of the search we explore all those nodes whose path back to the root includes n right arcs. This search strategy returns minimal cost solutions where hiding rules (right-rules) have (unit) cost and other rules (left-rules) having no cost. We have implemented a meta-interpreter that performs this search as follows. Given a set of nodes N, *left*(N)* returns the set of nodes reachable from the nodes in N by taking any number of left arcs. The function $right(N)$ returns the set of nodes reachable from the nodes in N by taking one right arc. Finally $solutions(N)$ returns any answers in the set of nodes N. The figure accompanying the following algorithm illustrates the order in which nodes in a binary tree are explored under left-first search.

function *left-first(Start);*
  Nodes := left({Start});*
  loop :
    Ans := solutions(Nodes);
    if *Ans* $\neq$ {} **then return** *Ans;*
    Node := left(right(Nodes));*
    goto loop;



For strongly minimal difference unifications, this algorithm returns the first set of answers and stops. For weakly minimal difference unifications, we must save the answers generated, and continues to search comparing new answers for weak minimality against previous ones. Unfortunately, to return all the weakly minimal difference unifications we must search the whole tree. The advantage of left-first search is that we can immediately tell whether an answer is weakly minimal.

6 Properties

Let us introduce some notation that will be used to prove properties of difference unification and the DUNIFY rule set. We write

$$\langle E, \sigma, A_s, A_t \rangle \stackrel{D}{\Longrightarrow} \langle E', \sigma', A'_s, A'_t \rangle$$

to indicate that there is a derivation D (that is, a possible empty sequence of DUNIFY rule applications) which transforms $\langle E, \sigma, A_s, A_t \rangle$ into $\langle E', \sigma', A'_s, A'_t \rangle$. We call annotation sets A_s and A_t *proper* with respect to an equation set E iff for all $s = t \,/\, p, q \in E$, no annotation in A_s extends p and no annotation in A_t extends q. Furthermore, we call a substitution σ *disjoint* with respect to E iff the domain of σ is disjoint from the set of variables in the equations in E. Finally, to relate solutions of subproblems to solutions of superproblems we define a function *strip* which "adjusts" an annotation set by returning suffixes of a given address prefix.

$$\begin{aligned} strip(A, \Lambda) &= A \\ strip(A, p.\Lambda) &= \{q \mid p.q \in A\} \end{aligned}$$

To prove soundness we first prove something stronger.

Theorem 1 (Soundness on Subequations) *If* $\langle s = t \,/\, \Lambda, \Lambda, \{\}, \{\}, \{\} \rangle \stackrel{D}{\Longrightarrow} \langle E, \sigma, A_s, A_t \rangle \stackrel{D'}{\Longrightarrow} \langle \{\}, \sigma', A'_s, A'_t \rangle$ *then for all* $s' = t' \,/\, p, q \in E$, $\sigma'(skeleton(s', strip(A'_s, p))) = \sigma'(skeleton(t', strip(A'_t, q)))$

Proof: The proof uses induction on the length of D'. The base case, a 0 length derivation, is trivial since $E = \{\}$ and so there are no $s = t \,/\, p, q \in E$.

In the step case, assume the theorem for derivations of length m and consider a length $m+1$ derivation. Consider the first rule application. Suppose it is a unification rule. By the induction hypothesis we know that σ is a unifier for the skeleton of the subproblems produced by this rule. But the skeleton is not changed by unification rule and since they are sound for unification, then σ is a unifier for E itself. Hence, given that the term addresses are properly computed by each unification (easily checked), that annotations in A_s or A_t don't refer to term addresses in E (properness) and that σ does not already contain bindings for terms in E (disjointness) the theorem follows. Properness and disjointness hold since they hold for $\langle E, \sigma, A_s, A_t \rangle$ and are invariant under applications of difference unification rules.

Alternatively, the first rule is a hiding rule. Without loss of generality, we assume it is an application of HIDE_Lwhich replaces the equation, $f(s_1, ...s_n) = t \,/\, p, q$ by $s_i = t \,/\, p@i, q$ and adds $p@i$ to A_s. As $f(s_1, ...s_n) = t \,/\, p, q$ is the only equation in E which changes, this is the only equation for which we need to verify that the induction conclusion still holds. By definition, $skeleton(f(s_1, ..., s_n), strip(A'_s, p)) = s_i$. And by the induction hypothesis, $\sigma'(skeleton(s_i, strip(A'_s, p@i))) = \sigma'(skeleton(t, strip(A'_t, q)))$ Thus, $\sigma'(skeleton(f(s_1, ..., s_n), strip(A'_s, p))) = \sigma'(skeleton(t, strip(A'_t, q)))$ □

As a consequnce we have soundness of the rule set.

Theorem 2 (Soundness) *If* $\langle s = t \,/\, \Lambda, \Lambda\}, \{\}, \{\}, \{\} \rangle \stackrel{D}{\Longrightarrow} \langle \{\}, \sigma, A_s, A_t \rangle$ *then* $du(s, t, A_s, A_t, \sigma)$.

Proof: The above theorem gives us a unifier and annotations satisfying the *du* relation, we only need show the unifier is most general. To do this, we show it is the mgu of $skeleton(s, A_s)$ and $skeleton(t, A_t)$.

To do this we construct a derivation parallel to D. It begins with

$$\{\langle skeleton(s, A_s) = skeleton(t, At) \,/\, \Lambda, \Lambda\}, \{\}, \{\}, \{\}\rangle$$

and for each unification rule (ignoring hiding rules) in D applies the same rule to this new sequence of equations. It is easy to see that each such rule is applicable and at the end the same unifier σ is built. As the underlying unification algorithm construts most general unifiers, it follows that σ is such a unifier. □

The converse is proven by a dual kind of "parallel construction".

Theorem 3 (Completeness) *$du(s, t, A_s, A_t, \sigma)$ then $\langle\{s = t \,/\, \Lambda, \Lambda\}, \{\}, \{\}, \{\}\rangle \stackrel{D}{\Longrightarrow} \langle\{\}, \sigma, A_s, A_t\rangle$.*

Proof: The unification rules of DUNIFY are complete for standard unification. Indeed, they are complete (but less efficient) with several restrictions which we now make. If we can show completeness with these restrictions, clearly the unrestricted rules are also complete. First, we restrict DELETE to only delete equations between atoms since we can strip equal terms down to atoms using DECOMPOSE. Second, we restrict $\text{ELIMINATE}_\text{L}$ and $\text{ELIMINATE}_\text{R}$ to cases when the non-variable term is atomic, since we can use the imitate rules to incrementally strip off and assign outermost function symbols.

Now, consider an arbitrary s, t, A_s, A_t, and σ as above. Since $\sigma(skeleton(s, A_s)) = \sigma(skeleton(t, A_t))$ then, as our restricted DUNIFY is complete it will find this mgu σ using only unification rules on input of these skeletons. Now let $E_0, ...E_n$ be the $n+1$ sets of equations generated by n rule applications $R_1, ..., R_n$ of DUNIFY as it computes σ. We claim that we can construct a "parallel" sequence of rule applications $R'_1, ..., R'_m$ that difference unifies s and t where the DUNIFY rules R_i occur in the same order and have interleaved hiding rules which hide exactly the addresses in A_s and A_t. If this claim is true then this theorem follows, as it yields DUNIFY execution sequence which returns σ, A_s, and A_t.

We construct this sequence as follows. In the 0th step begin with the equation set $E_0 = \{s = t \,/\, \Lambda, \Lambda\}$. In the i+1st step we look at the equations in our set E_i and if we have an equation

$$f(s_1, ..., s_n) = t' \,/\, p, q$$

and we also have $p@p_{k+1} \in A_s$ (and there is at most one such address given a well-formed annotation set) then we set R'_i to be the HIDE_L rule (hiding p_{k+1}) and replace this equation with

$$s_i = t' \,/\, p@p_{k+1}, q$$

to form E_{i+1} We do the analogous check with A_t to add HIDE_R rules. If neither of these apply, we instead pick the next rule from the sequence for R'_i. Now as the original sequence of rules is finite, and the annotation sets are finite, this process must terminate. Furthermore, recall our restrictions on the deletion and elimination rule; they guarantee that every term position in $Pos(s)$ and $Pos(t)$ eventually appears in equations of some E_i. Since elements of A_s are a subset of $Pos(s)$ corresponding to addresses of function arguments in s, and likewise for A_t, there is some point at which a corresponding hiding rule is applied. □

Note that these soundness and completeness arguments only rely on the underlying unification algorithm being sound, complete, and "decompositional" in the sense that every term position in the original terms eventually appears associated with an equation in some E_i. Hence, if we replace the underlying unification algorithm with something stronger, e.g., incorporating equations that preserves these properties for some equational theory, then again we will have a sound and complete algorithm with respect to that theory. We suspect there are many natural applications of equational difference unification. Hutter has recently reported an example based on using associative commutative difference unification and rippling to solve SAMs lemma in the INKA system.[Hut92]

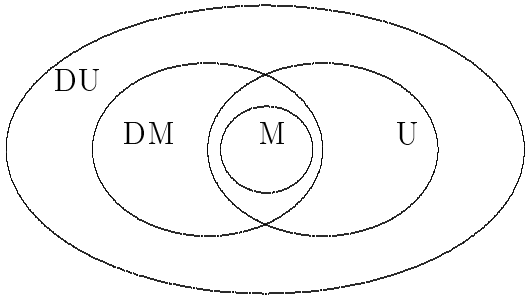
Theorem 4 (Termination) *The* DUNIFY *rule set always terminates.*

Proof: Let $E_0, E_1, ...E_n$ be a sequence of equations resulting from applying DUNIFY rules. Associate with each such sequence a triple of numbers $\langle I, N, F\rangle$ were I is $|Pos(s)|+|Pos(t)|-I'$, I' is the number of imitation steps performed in the sequence, V is the number of distinct variables in E_n, and F is the number of function symbols in E_n (with constants considered as nullary functions). Each of these is well ordered by the $<$ relation on the natural numbers, hence so is their lexicographic ordering. Now, examining the rules shows that I is decreased by the imitation rules and is otherwise unchanged. With the exception of these I decreasing rules, V is decreased by the elimination rules and is otherwise unchanged. And finally, with the exception of these I and V decreasing rules, all other rules decrease F. We therefore conclude that a sequence cannot be infinitely extended since each rule application is order decreasing. □

Theorem 5 (Subsumption) *Matching and unification are a special case of difference unification.*

Proof: For matching, $\sigma(s) = t$ if and only if $du(s, t, \{\}, \{\}, \sigma)$ and for unification, s and t have most general unifier σ if and only if $du(s, t, \{\}, \{\}, \sigma)$. This follows easily from definitions of difference unification. □

These results (and more) are summarized in the following picture where U stands for unification, M for matching, and D for difference.



7 Complexity

DUNIFY has been given as a set of rules. If they are applied non-deterministically, it is easy to see that it can take exponential time to find a solution to a problem as we may, using the hide and imitate rules, consider all the ways of hiding function symbols.[5] A term of size n (n function symbols) has $O(n)$ interior (neither constants or variables) function symbols that can be hidden in $O(2^n)$ different ways; hence, naive execution can be exponential. It is natural to ask whether this the best that we can do, and which are the tractible cases. In asking such questions we must distinguish between the problem of generating all solutions and that of generating a solution or knowing if one exists. The first problem is easily seen to require exponential size and space even in the very restricted of case of ground difference matching.

Theorem 6 *There are difference matching problems requiring exponential time and space.*

Proof: Consider difference matching $\langle s, t\rangle$ where $s = f^n(a)$ and $t = f^{2n}(a)$ (f^m(a) is the m-fold application of f to a). The solutions correspond to choosing n out of $2n$ occurrence of f in t to hide. That is there are $\left(\begin{array}{c} 2n \\ n \end{array}\right) = (2n!)/((n!)^2)$ which is $O(2^n)$. Note that all of these are strongly minimal. □

Problems generating exponential numbers of solutions are exceptional as they involve unusual amount of repeated structure. In general, there are far fewer matches and unifiers; so it is interesting to investigate the complexity of returning a single solution, or determining if one exists. Below we show that, in the

[5] Also note that unification algorithms which explicitly represent substitutions are not efficient. This can, however, be avoided by using a rule-based approach such as [JK91] at the cost of rules with rather more complicated side-conditions.

ground case, determining the existence of a solution to difference matching or difference unification is polynomial time decidable. The algorithms given are based on dynamic programming and can be easily modified to return solutions as well as indicate their existence or non-existence. After, we show that when variables are admitted, the problem becomes NP complete.

7.1 Ground difference matching and unification

Theorem 7 *Given terms s and t we can determine if s difference matches t (s may be annotated with skeleton t) or s difference unifies with t in polynomial time.*

Proof: For difference unification, consider the following rewrite rules on pairs of terms.

$$
\begin{array}{lcl}
\langle t, t\rangle & \rightarrow & TRUE \\
\langle a, b\rangle & \rightarrow & FALSE \text{ (for } a \neq b \text{ and } a,b \text{ atoms)} \\
\langle h(s_1, ..., s_k), h(t_1, ..., t_k)\rangle & \rightarrow & \langle s_1, h(t_1, ..., t_k)\rangle \vee ... \vee \langle s_k, h(t_1, ..., t_k)\rangle \vee \\
 & & \langle h(s_1, ..., s_k), t_1\rangle \vee ... \vee \langle h(s_1, ..., s_k), t_k\rangle \vee \\
 & & (\langle s_1, t_1\rangle \wedge ... \wedge \langle s_k, t_k\rangle) \\
\langle h(s_1, ..., s_k), t\rangle & \rightarrow & \langle s_1, t\rangle \vee ... \vee \langle s_k, t\rangle \\
\langle s, h(t_1, ..., t_k)\rangle & \rightarrow & \langle s, t_1\rangle \vee ... \vee \langle s, t_k\rangle
\end{array}
$$

If we apply these rewrite rules deterministically, given priority in the order listed above, it is not difficult to see see that there is a rewriting of $\langle s, t\rangle$ to $TRUE$ iff s difference unifies with t.

Whilst naive application of these rewrite rules results in an exponential amount of computation we can do better. In particular, if we assume some fixed maximal arity for the functions in the signature, each reduction can be computed in constant time[6] given truth values for the subproblems. However, notice that there are only $|s| * |t|$ subproblems, corresponding to the cartesian product of subterms from s and t. If we use dynamic programming to compute these in a sensible order, or alternatively use a memo function with constant lookup time, then the overall complexity is $O(|s| * |t|)$.

The rules are easily modified for the ground difference matching problem: change b to a term t in the second rule, delete the $\langle h(s_1, ..., s_n), t_i\rangle$ disjuncts in the third rule, and delete the final rule entirely. Note that this ground DM algorithm can be used to solve the homeomorphic embedding problem of one ground term into another in polynomial time and is similar to the algorithm of [NS87] □

As a side note, observe that while the above ground difference unification algorithm can be easily modified to yield minimal answers, there is a trivial linear time algorithm for determining difference unifiability although it does not give minimal answers. That is, s and t will difference unify iff they share at least one constant (of arity 0). In the non-ground case, s and t are difference unifiable iff they share one constant or if either contains a variable. In this respect, difference unification is, perhaps surprisingly, easier than difference matching.

7.2 Difference unification with variables

Difference unification and all its subproblems are trivially in NP since we can guess annotations and then unify or match resulting skeletons in polynomial time. To show NP completeness, we prove that when variables are added determining the existence of a solution is NP hard.

Theorem 8 *Difference unifying s and t, with annotation on only one side is NP hard.*

Proof: Assume we only allow annotation on the second term t (i.e. delete one of the two hiding rules). We reduce 3SAT to this restricted difference unification problem by the following construction which has similarities to one used in [KN86]. Let $C = \{c_1, c_2, ..., c_m\}$ be an instance of 3SAT over the boolean

[6] Without this assumption of fixed arity, the bound becomes linear in the size of the two terms.

variables $x_1, ..., x_n$. We construct two terms s and t to difference unify where s represents the clauses and t the satisfying assignments.

The term signature is as follows. Corresponding to each clause c_i is the distinct ternary function g_i. We also employ the m-ary function h and the 7ary f and the constants 0 and 1. Further, let V, the set of variables be $\{x_1, ..., x_n\}$. We intend that the truth or falsity of a boolean variable x_i is simulated by $x_i = 1$ and $x_i = 0$ respectively. For each clause c_j, do the following: let x_1, x_2, x_3 be the variables in c_j. There are exactly 7 sets of truth-value-assignments that make clause c_j true. Define 7 distinct terms $q_{j1}, ..., q_{j7}$ as follows: $q_{ji} = g_j(b_1, b_2, b_3)$ where $b_i \in \{0, 1\}$ and the assignment (b_1, b_2, b_3) satisfies c_j. Now let $s_j = g_j(x_1, x_2, x_3)$ and $t_j = f(q_{j1}, ..., q_{j7})$. Note that if s_j difference unifies with t_j than a solution yields a substitution that is a satisfying assignment for clause c_j.

Now form terms s and t as follows: $s = h(s_1, ..., s_m)$ and $t = h(t_1, ..., t_m)$. Observe that the solution to difference matching must pick, for each j, exactly one q_{ji} to unify with the variables in clause c_j and the combination of these is a satisfying assignment. Hence if there exists a solution, the clauses are satisfiable.

Since the reduction is trivially polynomial time computable we have shown NP-hardness. □

8 Conclusions

In [Rob89], J.A. Robinson presented a simple account of unification in terms of difference reduction. He observed

> "Unifiers remove differences ... We repeatedly reduce the difference between the two given expressions by applying to them an arbitrary reduction of the difference and accumulate the product of these reductions. This process eventually halts when the difference is no longer negotiable [reducible via an assignment], at which point the outcome depends on whether the difference is empty or nonempty."

In this light, our research can be seen as a direct extension of Robinson's notion of difference reduction: we reduce differences not just by variable assignment, but also by term structure annotation. What makes our extended notion of unification tenable, indeed attractive, is that this annotation is precisely what is required for rippling to remove this difference.

References

[Aub75] R. Aubin. Some generalization heuristics in proofs by induction. In G. Huet and G. Kahn, editors, *Actes du Colloque Construction: Amelioration et verification de Programmes*. Institut de recherche d'informatique et d'automatique, 1975.

[Bac88] Leo Bachmair. Proof by consistency in equational theories. In *Symposium on Logic in Computer Science*, 1988.

[BBH92] Richard Barnett, David Basin, and Jane Hesketh. A recursion planning analysis of inductive completion. Technical Report MPI-I-92-230, Max-Planck-Institut für Informatik, 1992.

[BD77] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, 1977.

[BS88] Karl Hans Bläsius and Jörg H. Siekmann. Partial unification for graph based equational reasoning. In *9th International Conference On Automated Deduction*, pages 397 – 414, Argonne, Illinois, 1988. Springer-Verlag.

[BSvH+92] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. newblock Rippling: A heuristic for guiding inductive proofs. Research Paper 567, Dept. of Artificial Intelligence, Edinburgh, 1991.

[Bun88] Alan Bundy. The use of explicit plans to guide inductive proofs. In *9th International Conference On Automated Deduction*, pages 111–120, Argonne, Illinois, 1988.

[BW92] David Basin and Toby Walsh. Difference matching. In *Proc. of 11th International Conference On Automated Deduction (CADE-11)*, pages 295–309, Saratoga Springs, New York, June 1992. Springer-Verlag.

[BW92a] David Basin and Toby Walsh. Difference unification. Technical Report MPI-I-92-247, Max-Planck-Institute für Informatik, 1992.

[Dig85] Vincent J. Digricoli. The management of heuristic search in boolean experiments with RUE resolution. In *9th IJCAI*, 1985.

[DK92] Vincent J. Digricoli and Eugene Kochendorfer. LIM+ challenge problems by RUE hyper-resolution. In D. Kapur, editor, *11th Conference on Automated Deduction*, pages 239–252. Springer Verlag, 1992.

[Fri86] L. Fribourg. A strong restriction of the inductive completion procedure. In *ICALP 13*. Springer-Verlag LNCS 226, 1986.

[Gra90] B. Gramlich. UNICOM: a refined completion based inductive theorem prover. In *10th CADE*, 1990. Lecture Notes in Artificial Intelligence No. 449.

[Hut90] Dieter Hutter. Guiding inductive proofs. In M.E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 147–161. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449.

[Hut92] Dieter Hutter. An application of rippling to SAMs lemma. Talk given in seminar on Kontrolle von Problemlösungsverfahren, GWAI-92, Bonn, Germany, September 1992.

[JK91] Jean-Pierre Jouannaud and Claude Kirchner. Solving Equations in Abstract Algebras: A Rule Based Survey of Unification. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic: essays in honour of Alan Robinson*. MIT Press, 1991.

[KN86] Deepak Kapur and Paliath Narendran. NP-completeness of the set unification and matching problems. In *8th International Conference On Automated Deduction*, pages 489–495, Oxford, UK, 1986.

[Mor69] J. Morris. E-resolution: an extension of resolution to include the equality relation. In *Proceedings of the IJCAI-69*, 1969.

[NS87] Paliath Narendran and Jonathan Stillman. In *Fifth International Conference on Applied Algebra and Error Correcting Codes*, Menorca, Spain, 1987.

[NSS63] A. Newell, J.C. Shaw, and H.A. Simon. The logic theory machine. In Feigenbaum and Feldman, editors, *Computers and Thought*, pages 61–79. McGraw-Hill, 1963.

[Rob89] J.A. Robinson. Notes on resolution. In F.L.Bauer, editor, *Logic, Algebra, and Computation*, pages 109–151. Springer Verlag, 1989.

[WNB92] T. Walsh, A. Nunes, and A. Bundy. The use of proof plans to sum series. In D. Kapur, editor, *11th Conference on Automated Deduction*, pages 325–339. Springer Verlag, 1992. Lecture Notes in Computer Science No. 607. Also available from Edinburgh as DAI Research Paper 563.