Difference Unification*

David A. Basin
Max-Planck-Institut fur Informatik
Im Stadtwald, Saarbriicken, Germany
(basin@mpi-sb.mpg.de)

Toby Walsh
Dept. of AI, University of Edinburgh,
80 S. Bridge, Edinburgh, Scotland
(tw@aisb.ed.ac.uk)

Abstract

We extend work on difference identification and reduction as a technique for automated reasoning. We generalise 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 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 for efficiently applying these rules. 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

Motivation and Context

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. 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 annotations which are used to direct a special kind of rewriting called rippling; 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

Related Work

This research is the outgrowth of previous work at Edinburgh in inductive theorem proving. There Bundy [Bun88; BS+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 [BS+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.

Research Contributions

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,

*This research was partially funded by the German Ministry for Research and Technology (BMFT) under grant ITS 9102 and a SERC postdoctoral fellowship. The 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 Alan Bundy and Andrew Ireland.

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; 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. 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 from traditional rewrite based theorem proving, is developed in the next section.

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 a{y) 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(capital letters represent variables and lower case letters constants and bound variables).

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

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

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:

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

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

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 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 a larger skeleton. As before the left hand side is rippled-out; rippling on the right with (2) yields the left 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 and where normalization would therefore loop. The combination of difference unification and rippling is often an effective heuristic in theories where rewrite based procedures do not exist; the next example, 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])

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

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

$$\sum_{i=0}^{N} \boxed{\frac{1}{\underline{s(i)}} - \frac{1}{s(s(i))}} = 1 - \frac{1}{s(s(N))} \quad \vdash$$

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

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

$$\sum_{j=A}^{B} \boxed{\sum_{k=C}^{D} \underline{U}} \rightarrow \boxed{\sum_{k=C}^{D} \underline{\sum_{j=A}^{B}} U} \qquad (6)$$

$$\sum_{j=A}^{B} \boxed{C \times \underline{U}} \rightarrow \boxed{C \times \underline{\sum_{j=A}^{B} U}} \qquad (7)$$

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

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,

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

then wave-rule (7),

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

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

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

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

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

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

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 the standard result into the goal,

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

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 Other applications

We have explored a number of other applications of difference unification that, for lack of space, we cannot develop here. For example, in [BW92a; BBH93] we show how difference unification can be used to guide rewriting in so called *proof by consistency* techniques. Other researchers have also explored applications of these ideas. Hutter has recently reported on applying associative commutative difference unification and rippling to solve SAMs lemma in the INKA system [CH92J.

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); this is without loss of generality as "thicker" wavefronts can be represented by nested wavefronts. In addition, rather than superimposing a particular representation on terms, 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 $Pos(f(s_1, ..., s_n))$ equals $\{\Lambda\} \cup \{i.p \mid 1 \leq i \leq n \ \wedge \ p \in Pos(s_i)\}$. So Λ represents the empty sequence and "." concatenation. The subterm of a term t at position p is t/p where $t/\Lambda = t$ and $f(s_1, ..., s_n)/i.p = s_i/p$. For example, annotations for $f(g(\boxed{f(\underline{a}, b)}), \boxed{g(\underline{b})})$ are given by the set $\{1.1.1, 2.1\}$. In what follows we only work with sets of annotations that are *well-formed* with respect to given terms. That is 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 $skel(t, A_t)$ 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 $f(g(a), b)$. We say that q *extends* p iff $q = p.i$, or q extends some r and r extends p.

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

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

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 $\sigma(skel(s, A'_s)) = \sigma(skel(t, A'_t))$*

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 $skel(s, A'_s)$ unifiable with $skel(t, A'_t)$*

For example, $\langle \boxed{f(\underline{X})}, f(a)\rangle$ is weakly minimal with substitution $\{f(a)/X\}$ 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 record annotation addresses.

The rules for difference unification are given in Figure 1. 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.

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 ELIMINATE$_L$ committing to $\{g(a)/x\}$, 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, ELIMINATE$_L$; difference matching is implemented by DELETE, DECOMPOSE, ELIMINATE$_L$, HIDE$_L$; and unification is implemented by DELETE, DECOMPOSE, ELIMINATE$_L$, ELIMINATE$_R$. Therefore matching and unification are special cases of difference unification. 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 ELIMINATE$_L$ and ELIMINATE$_R$ 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 $\{f(a)/X\}$ 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".

$$
\begin{array}{llll}
\text{DELETE} & \langle S \cup \{s = s \,/\, p,q\}, \sigma, A_s, A_t \rangle & \Rightarrow & \langle S, \sigma, A_s, A_t \rangle \\
\text{DECOMPOSE} & \langle S \cup \{f(s_1,\ldots,s_n) = f(t_1,\ldots,t_n) \,/\, p,q\}, \sigma, A_s, A_t \rangle & \Rightarrow & \\
 & & & \langle S \cup \{s_i = t_i \,/\, p.i, q.i \mid 1 \le i \le n\}, \sigma, A_s, A_t \rangle \\
\text{ELIMINATE}_L & \langle S \cup \{X = t \,/\, p,q\}, \sigma, A_s, A_t \rangle & \Rightarrow & \langle S[X \mapsto t], \sigma \circ \{t/X\}, A_s, A_t \rangle \ \text{ if } \neg occurs(X,t) \\
\text{ELIMINATE}_R & \langle S \cup \{s = X \,/\, p,q\}, \sigma, A_s, A_t \rangle & \Rightarrow & \langle S[X \mapsto s], \sigma \circ \{s/X\}, A_s, A_t \rangle \ \text{ if } \neg occurs(X,s) \\
\text{IMITATE}_L & \langle S \cup \{X = f(t_1,\ldots,t_n) \,/\, p,q\}, \sigma, A_s, A_t \rangle & \Rightarrow & \langle S[X \mapsto f(X_1,\ldots,X_n)] \cup \{X_i = t_i \,/\, p.i, q.i \mid 1 \le i \le n\}, \\
 & & & \sigma \circ \{f(X_1,\ldots,X_n)/X\}, A_s, A_t, \rangle \ \text{ if } \forall i \in [1,n]\ valid_t(q.i) \\
\text{IMITATE}_R & \langle S \cup \{f(s_1,\ldots,s_n) = X \,/\, p,q\}, \sigma, A_s, A_t \rangle & \Rightarrow & \langle S[X \mapsto f(X_1,\ldots,X_n)] \cup \{s_i = X_i \,/\, p.i, q.i \mid 1 \le i \le n\}, \\
 & & & \sigma \circ \{f(X_1,\ldots,X_n)/X\}, A_s, A_t, \rangle \ \text{ if } \forall i \in [1,n]\ valid_s(p.i) \\
\text{HIDE}_L & \langle S \cup \{f(s_1,\ldots,s_n) = t \,/\, p,q\}, \sigma, A_s, A_t \rangle & \Rightarrow & \langle S \cup \{s_i = t \,/\, p.i, q\}, \sigma, A_s \cup \{p.i\}, A_t \rangle \ \text{ if } valid_s(p.i) \\
\text{HIDE}_R & \langle S \cup \{s = f(t_1,\ldots,t_n) \,/\, p,q\}, \sigma, A_s, A_t \rangle & \Rightarrow & \langle S \cup \{s = t_i \,/\, p, q.i\}, \sigma, A_s, A_t \cup \{q.i\} \rangle \ \text{ if } valid_t(q.i)
\end{array}
$$

Figure 1: DUNIFY : transformation rules for difference unifying s and *t*

These rules have been implemented in Prolog. The following table gives 8 out of the 24 results of difference unifying (X + Y) + Z with X + (Y + Z). Note that for readability we have merged adjacent wavefronts in the "box and hole" presentation.

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

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.

The following is a sample execution trace of the first result. It results from applying the rules: HIDEL, DECOMPOSE, DELETE, HIDER, and DELETE.

$$
\begin{array}{l}
\langle \{(X+Y)+Z = X+(Y+Z) \,/\, \Lambda, \Lambda\}, \{\}, \{\}, \{\} \rangle \\
\langle \{X+Y = X+(Y+Z) \,/\, 1.\Lambda, \Lambda\}, \{1\}, \{\}, \{\} \rangle \\
\langle \{X = X\} \,/\, 1.1, 1, Y = Y+Z \,/\, 1.2, 2\}, \{1\}, \{\}, \{\} \rangle \\
\langle \{Y = Y+Z \,/\, 1.2, 2\}, \{1\}, \{\}, \{\} \rangle \\
\langle \{Y = Y \,/\, 1.2, 2.1\}, \{1\}, \{2.1\}, \{\} \rangle \\
\langle \{\}, \{1\}, \{2.1\}, \{\} \rangle
\end{array}
$$

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. This is both time consuming and almost always unnecessary. 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$_L$, IMITATER, EL1MINATE$_L$, and ELIMINATER. Arcs to the right result from applying a hiding rule: HIDEL and HIDER. 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 TV, *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*. Figure 2 gives the algorithm and illustrates the order in which nodes in a binary tree are explored under left-first search.

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 possibly empty sequence of DUNIFY rule applications) which

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

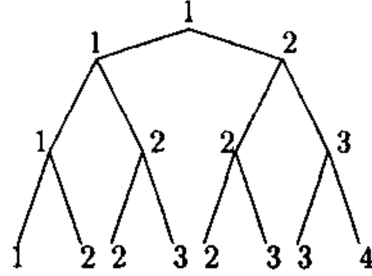

Figure 2: Algorithm and Search Tree

transforms $\langle E, \sigma, A_s, A_t\rangle$ into $\langle E', \sigma', A'_s, A'_t\rangle$.

We now sketch proofs of soundness and completeness; that is, proofs that the rules return all and only the difference unifiers. Full proofs are given in [BW92a].

Theorem 1 (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 (sketch): We first establish a lemma: 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 $s' = t'\ /\ p, q \in E$,

$$\sigma'(skel(s', strip(A'_s, p))) = \sigma'(skel(t', strip(A'_t, q))),$$

where *strip* is a function that "adjusts" an annotation set by deleting prefixes from addresses. That is, $strip(A, \Lambda) = A$ and $strip(A, p) = \{q \mid p.q \in A\}$. Soundness follows from the lemma provided the unifier σ is a most general unifier. This holds since we can construct a derivation which unifies $skel(s, A_s)$ and $skel(t, A_t)$ by deleting every hide rule from D and applying the remaining unification rules. This builds the same unifier σ and, as the underlying unification algorithm returns mgus, it follows that σ is a mgu.

The lemma follows by induction on the length of D''. The base case is trivial. In the step case, unification rules are easily seen to preserve sound solutions. The HIDE$_L$ rule replaces an equation like $f(s_1, \ldots s_n) = t\ /\ p, q$ by $s_i = t\ /\ p.i, q$ and adds $p.i$ to A_s. By definition, $skel(f(s_1, \ldots, s_n), strip(A'_s, p)) = s_i$, and by hypothesis, $\sigma'(skel(s_i, strip(A'_s, p.i))) = \sigma'(skel(t, strip(A'_t, q)))$. Conclude that $\sigma'(skel(f(s_1, \ldots, s_n), strip(A'_s, p))) = \sigma'(skel(t, strip(A'_t, q)))$ and the theorem follows. HIDE$_R$ is analogous. □

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

Proof (sketch): It suffices to consider the DUNIFY rules with DELETE restricted to delete only equations between atoms and ELIMINATE$_L$ and ELIMINATE$_R$ restricted to cases when the non-variable term is atomic. These restricted rules are complete for unification and so (without the hiding rules) return σ as a mgu for $skel(s, A_s)$ and $skel(t, A_t)$. The theorem follows by showing that the unification rules applied in computing this unifier can be applied (in the same order) to compute the same unifier for s and t while being interleaved with hiding rules which hide the addresses in A_s and A_t. This in turn follows from our restrictions on the deletion and elimination rule; they guarantee that every term position in *Pos(B)* and *Pos(t)* eventually appears in some equation in the derivation *D.* Since elements of *A,* are a subset of *Pos(s)* corresponding to addresses of function arguments in *s,* and likewise for A_t, appropriate hiding rule can always be applied. □

The soundness and completeness arguments only rely on the underlying unification algorithm being sound, complete, and "decompositional" in the sense that every position in the original terms eventually appears in the course of the derivation. Hence, if we replace the underlying unification algorithm with something stronger, e.g., incorporating equations that preserve these properties for some equational theory, then again we will have a sound and complete algorithm with respect to that theory. We suspect that there are many natural applications of equational difference unification, e.g., the previously mentioned work of Hutter.

Theorem 3 (Termination) *The* DUNIFY *rule set always terminates.*

Proof (sketch): Given input *s* and i, use a lexicographic ordering on the triple (I, *V, F)* were */* is |Pos(s)| + |Pos()tl — /', *V* is the number of imitation steps performed in a sequence of rule applications, *V* is the number of distinct variables in the (current) equation set, and *F* is the number of function symbols (including constants) in the (current) equation set. □

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.[1] A term of size n (n function symbols) has *O(n)* interior (neither constants or variables) function symbols that can be hidden in $0(2^n)$ 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

[1]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 [JK9l] at the cost of rules with rather more complicated side-conditions.

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 even in the very restricted of case of ground difference matching.

Theorem 4 There are difference matching problems requiring exponential time.

Proof: Consider difference matching (s,t) 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 $(2n!)/((n!)^2)$ 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 amounts 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. The first problem is polynomial time solvable for ground terms.

Theorem 5 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 (sketch): In [NS87] an algorithm based on dynamic programming is given for solving the homomorphic embedding problem of one ground term into another in polynomial time. This problem is the same problem as ground difference matching. It is easy to modify this idea to provide an algorithm for ground difference unification. Furthermore, these algorithms can be easily modified to return sets of answers as well as indicating if answers exist [BW92a].

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.

In general, 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. In the nonground case, when variables are added determining the existence of a solution is NP hard.

Theorem 6 Difference unifying s and t, with annotation on only one side is NP hard.

The proof is given in [BW92a] and uses a reduction from 3SAT similar to that used in the proof of the NP hardness of set-matching given in [KN86].

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 ts no longer negotiable [via an assignment]f 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

[BBH93] R. Barnett, D. Basin and J. Hesketh. A recursion planning analysis of inductive completion. Annals of Maths, and AI, 8(3-4), 1993.

[BS88] K. Hans Blasius and J. Siekmann. Partial unification for graph based equational reasoning. In 9th CADE, 397-414, 1988.

[BS+92] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland and A. Smaill. Rippling: A heuristic for guiding inductive proofs. To appear in Artificial Intelligence, 1993.

[Bun88] A. Bundy. The use of explicit plans to guide inductive proofs.In 9th CADE, 111 120, 1988

[BW92] D. Basin and T. Walsh. Difference matching. In 11th CADE, 295-309, 1992.

[BW92a] D. Basin and T. Walsh. Difference unification. Technical Report MPI-I-92-247, Max-Planck-Institute fur Informatik, 1992.

[CH92] J. Cleve and D. Hutter. A new methodology for equational reasoning. Univerity Saarbriicken Technical Report, 1993.

[Dig85] V. Digricoli. The management of heuristic search in boolean experiments with RUE resolution. In 9th IJCAI, 1154-1161, 1985.

[Hut90] D. Hutter. Guiding inductive proofs. In 10th CADE, 147-161, 1990.

[JK91] J.-P. Jouannaud and C. Kirchner. Solving Equations in Abstract Algebras: A Rule Based Survey of Unification. In J.-L. Lassez and G. Plotkin, eds., Computational Logic. MIT Press, 1991.

[KN86] D. Kapur and P. Narendran. NP-completeness of the set unification and matching problems. In 8th CADE, 489-495, 1986.

[Mor69] J. Morris. E-resolution: an extension of resolution to include the equality relation. In Proceedings of the 1JCA1-69, 1969.

[NS87] P. Narendran and J. Stillman. In Fifth Int. Conf 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. 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 11th CADE, 325-339, 1992.