Symbolic Reasoning by Difference Reduction

David A. Basin
Max-Planck-Institut für Informatik
Im Stadtwald
66123 Saarbrücken, Germany
Tel: (+49) 681 302 5435
Fax: (+49) 681 302 5401
Email: basin@mpi-sb.mpg.de

Toby Walsh*
INRIA-Lorraine,
Villers-les-Nancy, France
Tel: (+33) 83 59 30 15
Fax: (+33) 83 27 83 19
Email: walsh@loria.fr

February 18, 1994

Correspondence to be sent to first author.

*Current address: IRST, Location Panté di Povo, I38100, Trento, Italy. Email: toby@irst.it

Abstract

We present a new approach to automated reasoning based on difference identification and reduction. Difference identification is based on a generalization of 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 type of rewriting designed to reduce and 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. Moreover we show how it can be efficiently implemented based on a novel search strategy for unifiers. On the practical side, we show how this algorithm can be used in new ways to support and extend the role of rippling in theorem proving and other kinds of automated reasoning.

1 Introduction

1.1 Motivation and Background

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 by reducing syntactical differences between the formula to be proved and the hypotheses. 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. As Bläsius and Siekmann reported on their work [BS88]:

> "Although paramodulation was a substantial improvement compared to the axiomatical formalization of the equality relation, it still leads to enormous search spaces, as this inference rule can be applied almost everywhere in the clause space. .... Whereas most research is based with remarkable success on simplification mechanisms, especially on demodulation and term rewriting, difference reduction methods have found less attention, although they are at least as important for an automated deduction system."

Here we report on research sharing these cognitive and pragmatic aims. We have developed a 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 finding 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.

A close analogy between this kind of difference reduction and that of RUE resolution can be drawn. In RUE a failed resolution step, for example resolving $\{P(h(a), b)\}$ with $\{\neg P(g(a), b)\}$ gives rise to mismatches that occurred during unification. Here $h(a)$ and $g(a)$. A new clause is generated, $\{\neg Eq(h(a), g(a))\}$, that expresses the difference. Afterwards, proving that $h(a)$ equals $g(a)$ suffices to complete the proof. This strategy allows equality reasoning to be applied in a demand driven way to reduce differences. In our work, difference unification identifies the differences between two terms or formulas. Unlike RUE, the differences are not terms, but parts of terms (wave-fronts). Whereas in RUE mismatches are made equal by a chain of equality reasoning, in our work, rippling provides the link to manipulate the marked differences. One noteworthy place where this analogy fails is that the combination of difference unification and rippling can provide a stronger restriction on the search space then RUE. In particular, rippling is guarnateed to terminate: differences are moved along well-founded paths and rippling will therefore always halt.

This research is the outgrowth of previous work at Edinburgh in inductive theorem proving [Bun88, BSvH$^+$93] in which the induction conclusion is proven from the induction hypothesis by rippling. In "constructor" inductions, the induction conclusion is a copy of the induction hypotheses except for the appearance of certain constructor 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* (see §2 for formal definition). The constructor 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. This difference forms a wave-front which we mark by placing a box around it and underlining the subterm contained in the skeleton, $P(\boxed{s(\underline{n})})$. Rippling[1] 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 induction conclusion $P(\boxed{s(\underline{n})})$ is rewritten using wave-rules into some function of the induction hypothesis, $P(n)$; that is, into $\boxed{f(\underline{P(n)})}$ (f may be the identity). At this point we can call upon the induction hypothesis.

Rippling has been implemented in the OYSTER/CLAM prover [BvHHS90]. A similar kind of rewriting was developed independently by Hutter [Hut90], from ideas in [Bun88], and employed in the INKA system. Both systems have

[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.

enjoyed a high degree of success stemming from several desirable properties of rippling. Because annotation in the left hand side of a wave-rule must match annotation in the term being rewritten, wave-rules are only applied if they move annotated differences. As a result, there is very little search during rewriting. Indeed rippling always terminates since wave-fronts are only moved in some desired, well-founded, way — usually to the top of the term. This allows us to use as rules like associativity (given below[2]) as rewrite rules in both directions even though unannotated this would cause looping.

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

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

In addition, since wave-rules leave the skeleton unchanged, if rippling succeeds then we are guaranteed to be able to use the induction hypothesis. These properties are detailed in [BSvH+93], whilst [BvHS91] reports on experiments using rippling for inductive theorem proving.

Rippling can be used in the step case of an inductive proof as there is a natural way to annotate the inductive conclusion so that it has the same skeleton as the induction hypothesis: namely wave-fronts are placed around the new constructor symbols. Motivated by the success of rippling in this domain, we have developed difference unification as a technique to enable the use of rippling as a general purpose rewriting procedure. Difference unification is designed to identify differences between terms in a way suitable for rippling to remove them. Difference unification thus provides precisely the missing ingredient (annotations) needed to apply rippling in general settings where we have candidate terms or equations which may be used in rewrite proofs. Difference unification and rippling together form a new strategy for controlling proof search based on difference reduction and subsequent removal.

1.2 A Simple Example

We give here a brief illustration of the idea of difference reduction based on difference unification and rippling. The example we give is artificially simple (see §7 and §8 for more substantial examples) but it illustrates the ideas behind the combination of difference unification and rippling. In particular, it shows how rippling can be used in a new role as an iterative difference reduction technique. This role is rather different than in inductive theorem proving as discussed above where successful rippling of the induction conclusion guarantees use of the induction hypothesis.

Consider, for example, reasoning about equality terms built from an associative operator, like plus, using the wave-rules (1) and (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 plus in both directions.

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

Consider proving

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

We proceed by applying the difference unification algorithm (given in §3) which finds annotations for the two sides of the equation so that they share a common skeleton. Here there are 10 possible annotated answers corresponding to the 6 ways of selecting any 2 constants from the terms and 4 ways of selecting any one. In general, we prefer only those with minimal (formally defined in §2) 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 narrows the choice to 2:

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

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

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 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 sides 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 of these terms that share the same skeleton, 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 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.[3] 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

[3]Note that although rippling always terminates, iterating rippling with difference unification does not have this guarantee. Namely, it is possible for the skeleton to increase without bound in cases where rippling leads to larger terms. This is cannot be the case here as the overall term size (e.g., number of function symbols) is never increased.

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. For example using difference unification we may annotate $a + ((b + c) + d) = ((a + b) + c) + d$ as

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

This is rewritten to an identity by rippling with (2) on the left and (1) on the right. Examples where the savings over normalization are more dramatic are easy to construct.

More significantly, there are theories where associativity in both directions is required (See §7.1 for an example); here normalization would loop. Alternatively, we may require other wave-rules to be integrated into such a normalization procedure. For example, we might require additional wave-rules for the recursive definition of plus, or relating plus to other operators for more complex examples. Indeed, the combination of difference unification and rippling is often an effective heuristic in theories where rewrite based procedures do not exist. The combination serves to restrict search in a possibly infinite search space to a finite subspace in which all rewrites steps preserve a desired invariant: the skeleton returned by difference unification. Because the skeleton often represents an axiom or lemma that must be used in the proof, or a desired intermediate state, this heuristic can be very effective. The examples given in later sections demonstrate this.

1.3 Contributions

Our contributions are several fold. First, we define difference unification as an extension of unification where terms are made equal not just by finding substitutions for variables but also by hiding structure via term annotation. We give a rule based algorithm which uses conventional unification in a transparent way so that other additions to unification, such as incorporating equations for "equational difference unification" can be easily made. We prove the algorithm given is both sound and complete with respect to its specification. 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 define two restricted classes of useful answers (which we call *strongly* and *weakly* minimal) and 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. We then 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 and problem solving where proof proceeds by alternating between identifying differences and reducing them. This combination is different from the way rippling is used in induction proofs where successful rippling guarantees use of the induction hypothesis. Here we may have to ripple on two separate terms and even though rippling may succeed on both, we may still be unable to use one to rewrite the other. Hence, rippling combined with difference unification must be seen as one step, in possibly many, of difference reduction. Examples of this are given in §8.1 and §8.2.

1.4 Outline

In this paper we give a complete account of difference unification, which supersedes earlier accounts in [BW93, BW92]. The first part of the paper, §2–5 present difference unification and its properties. In §2 we present notation and specify what properties we expect from a difference unification algorithm. In particular, analogous to the notion of minimality on substitutions, we define notions of minimality on annotation. In §3 we give the difference unification algorithm as a set of transformation rules and in §4 we provide a meta-interpreter that applies these rules in a way that returns minimal answers first. In §5 we prove that our algorithm correctly meets the specification, and terminates. In §6 we analyze its complexity as well as its relationship to other problems such as homeomorphic tree embedding.

In the second part of the paper, §7 and §8, we present applications. The first sections consider applications in inductive theorem proving. We consider both so called "implicit" or "inductionless" induction calculi and explicit induction.[4] Our applications include using difference unification to bring rippling into the implicit induction framework, divergence analysis, creation of wave-rules, and search control during proof. In §8 we present applications of difference unification to non-inductive proofs, in particular, finding closed forms for series, equational reasoning, and learning recursive programs. Finally, in §9 we draw conclusions.

2 Notation and Specification

To specify difference unification we must be precise about annotation and its representation. The following notation was developed in [BSvH+93]. A wave-front is a term with a proper subterm deleted. We represent this by placing a box around the term and underlying the subterm. For example, the term $f(g(\boxed{f(\underline{a}, b)}), \boxed{g(\underline{b})})$ contains two wave-fronts. In this term, the two wave-fronts have an immediate subterm deleted, but this needn't always be the case, e.g., $f(\boxed{s(s(\underline{x}))})$. However, such a wave-front can always be maximally split into wave-fronts that each have an immediate subterm deleted, e.g., $f(\boxed{s(\underline{\boxed{s(\underline{x})}})})$. With out loss of generality we shall consider wave-fronts as always being maximally split, although we will, for convenience, print them as maximally merged.[5]

[4] Although, the line between these paradigms is blurred (see, for example, [Red90]). By implicit induction we mean provers based on some kind of completion or superposition calculus (e.g., [Bac88, Fri86, Mus80]). By explicit induction we mean systems that build or use inductive schemata to generate inductive cases, such as NQTHM, CLAM, or INKA [BvHHS90, BHHW86, BM79].

[5] This is without loss of generality as implementations of rippling are not dependent on how wave-fronts are split or merged. The CLAM system, for example, stores wave-fronts in maximally merged form, but splits them on demand during rewriting; This is required

Note too that we are only speaking here about wave-fronts with exactly one proper subterm deleted; this does not include the so called *multi-wave fronts* of [BSvH+93] in which wave-fronts have multiple wave-holes.

By assuming that a wave-front always has an immediate subterm deleted, we may specify wave-fronts uniquely by the address of the deleted subterm. In our algorithm, we will keep track of annotations with sets of such addresses. Positions are strings and are defined recursively as follows: the set of positions in the term t is $Pos(t)$ where,

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

So Λ represents the empty string and "." is the string concatenation operator. 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 $\{1.1.1, 2.1\}$. In what follows we 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).

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. Namely, $skeleton(t, A) = aux(t, A, \Lambda)$ where aux is the auxiliary function defined as follows.

$$\begin{aligned} aux(c, A, p) &= c \ \text{ for } c \text{ a variable or constant} \\ aux(f(t_1, ..., t_k), A, p) &= f(aux(t_1, A, p.1), ..., aux(t_k, A, p.k)) \ \text{ if } \forall i \in \{1..k\}.p.i \notin A \\ &= aux(t_i, A, p.i) \ \text{ if } p.i \in A \end{aligned}$$

Using our example from above, $skeleton(f(g(f(a, b)), g(b)), \{1.1.1, 2.1\})$ is $f(g(a), b)$. One final piece of notation is that we will say q *extends* p iff $q = p.i$, or q extends some r and r extends p.

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

$$\sigma(skeleton(s, A_s)) = \sigma(skeleton(t, A_t)), \tag{5}$$

where σ is a most general unifier.

In practice 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

since the boxes and holes are explicitly represented as part of the term structure of the term being rewritten and they must match annotation on the left hand side of the rewrite rule being applied. The alternative, to store wave-fronts maximally merged, obviates the need for splitting or merging during rewriting but requires more storage space and annotated terms are cumbersome to display.

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 $\{f(a)/X\}$ 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 in §7.3) 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.

3 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.

Description of the rules for difference unification are given below. For simplicity, the schema,

$$s = t \,/\, p, q \quad \rightarrow \quad \langle T, \mu, a_s, a_t\rangle,$$

represents the rule,

$$\langle S \cup \{s = t \,/\, p, q\}, \sigma, A_s, A_t\rangle \quad \rightarrow \quad \langle \mu(S) \cup T, \sigma \circ \mu, A_s \cup \{a_s\}, A_t \cup \{a_t\}\rangle,$$

where $\circ$ is the standard composition of substitutions. 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. In addition, the X_i are new variables.

$$
\begin{array}{lll}
\text{DELETE} & s = s \,/\, p, q & \rightarrow \;\; \langle \{\}, \{\}, \{\}, \{\} \rangle \\
\text{DECOMPOSE} & f(s_1, ..., s_n) = f(t_1, ..., t_n) \,/\, p, q & \\
 & & \rightarrow \;\; \langle \{ s_i = t_i \,/\, p.i, q.i \mid 1 \leq i \leq n \}, \{\}, \{\}, \{\} \rangle \\
\text{ELIMINATE}_{\text{L}} & X = t \,/\, p, q & \rightarrow \;\; \langle \{\}, \{t/X\}, \{\}, \{\} \rangle \;\; \text{if } \neg occurs(X, t) \\
\text{ELIMINATE}_{\text{R}} & s = X \,/\, p, q & \rightarrow \;\; \langle \{\}, \{s/X\}, \{\}, \{\} \rangle \;\; \text{if } \neg occurs(X, s) \\
\text{IMITATE}_{\text{L}} & X = f(t_1, ..., t_n) \,/\, p, q & \rightarrow \;\; \langle \{ X_i = t_i \,/\, p.i, q.i \mid 1 \leq i \leq n \}, \\
 & & \quad\;\; \{ f(X_1, ..., X_n)/X \}, \{\}, \{\} \rangle \;\; \text{if } \forall i \in [1, n].\, valid_t(q.i) \\
\text{IMITATE}_{\text{R}} & f(s_1, ..., s_n) = X \,/\, p, q & \rightarrow \;\; \langle \{ s_i = X_i \,/\, p.i, q.i \mid 1 \leq i \leq n \}, \\
 & & \quad\;\; \{ f(X_1, ..., X_n)/X \}, \{\}, \{\} \rangle \;\; \text{if } \forall i \in [1, n].\, valid_s(p.i) \\
\text{HIDE}_{\text{L}} & f(s_1, ..., s_n) = t \,/\, p, q & \rightarrow \;\; \langle \{ s_i = t \,/\, p.i, q \}, \{\}, p.i, \{\} \rangle \text{ if } valid_s(p.i) \\
\text{HIDE}_{\text{R}} & s = f(t_1, ..., t_n) \,/\, p, q & \rightarrow \;\; \langle \{ s = t_i \,/\, p, q.i \}, \{\}, \{\}, q.i \rangle \text{ if } valid_t(q.i)
\end{array}
$$

The conditions on the $\text{ELIMINATE}_{\text{L}}$ and $\text{ELIMINATE}_{\text{R}}$ rules are simple occurs checks. The conditions on the $\text{IMITATE}_{\text{L}}$ and $\text{IMITATE}_{\text{R}}$ rules ensure that we do not needlessly imitate on term structure introduced by substitution. The conditions on the HIDE_{L} and HIDE_{R} rules allow annotation to be placed only on the original terms, and not term structure introduced by substitution. Note also that 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 $\{g(a)/x\}$, we fail to get a solution. To find a solution, we must instead apply DECOMPOSE then $\text{ELIMINATE}_{\text{R}}$ committing to $\{a/x\}$, and finally HIDE_{R} and DELETE.

The rules are closely related to rules for matching, and unification. Matching is implemented by the rules DELETE, DECOMPOSE, and $\text{ELIMINATE}_{\text{L}}$ whilst unification is implemented by DELETE, DECOMPOSE, $\text{ELIMINATE}_{\text{L}}$, and $\text{ELIMINATE}_{\text{R}}$. Moreover, there are two kinds of difference matching possible depending if we wish to annotate the side with or without variables. The first of these (with variables and annotations on the first-term) is implemented by the rules DELETE, DECOMPOSE, $\text{ELIMINATE}_{\text{L}}$, and HIDE_{L}. The second of these (with variables on left and annotations on the right) by the rules DELETE, DECOMPOSE, $\text{ELIMINATE}_{\text{L}}$, $\text{IMITATE}_{\text{L}}$, and HIDE_{R}.

In rule base unification, algorithms often use additional rules 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 $\text{IMITATE}_{\text{L}}$ and $\text{IMITATE}_{\text{R}}$ 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 $\text{IMITATE}_{\text{L}}$ and $\text{IMITATE}_{\text{R}}$ 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". The complexity of difference unification is further discussed in §6.

These rules have been implemented in Prolog and the following table gives 8 out of the 24 results of difference unifying $(X+Y)+Z$ with $X+(Y+Z)$. Recall 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}

Table 1: Sample Unifiers for $(X+Y)+Z = X+(Y+Z)$

The first three annotations are strongly minimal. 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 trace of the execution of difference unification that yields the first result.

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

4 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. For example, if two terms unify without annotation, our algorithm will find this unifer without ever trying any annotation.

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, $\mathrm{IMITATE_L}$, $\mathrm{IMITATE_R}$, $\mathrm{ELIMINATE_L}$, and $\mathrm{ELIMINATE_R}$. Arcs to the right result from applying a hiding rule: $\mathrm{HIDE_L}$ and $\mathrm{HIDE_R}$. 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. Figure 4 contains pseudo-code for this interpreter as well as an example to illustrate 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 ≠ {} then return Ans;
        Node := left*(right(Nodes));
        goto loop;
```

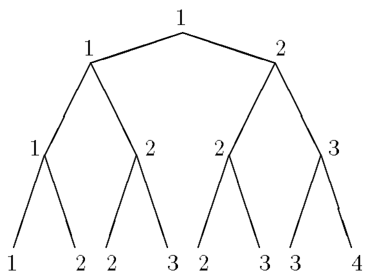

Figure 1: Left-First-Search Algorithm and an Example

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 continue 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.

5 Properties

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

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

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.

$$strip(A, p) \quad = \quad \{q \mid p.q \in A\}$$

To prove soundness we first prove something stronger.

Theorem 1 (Soundness on Subequations) *If* $\langle s = t \,/\, \Lambda, \Lambda, \{\}, \{\}, \{\}\rangle \overset{D}{\Longrightarrow} \langle E, \sigma, A_s, A_t\rangle \overset{D'}{\Longrightarrow} \langle \{\}, \sigma', A'_s, A'_t\rangle$ *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. However, the skeleton is not changed by the unification rules and these rules are sound for unification, thus σ is a unifier for E itself. Hence, given that the term addresses are properly computed by each unification rule (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 $\mathrm{HIDE_L}$ which 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,

$$\sigma'(skeleton(s_i, strip(A'_s, p.i))) = \sigma'(skeleton(t, strip(A'_t, q))),$$

and thus $\sigma'(skeleton(f(s_1, ..., s_n), strip(A'_s, p))) = \sigma'(skeleton(t, strip(A'_t, q)))$. $\Box$

As a consequence 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

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

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 constructs most general unifiers, it follows that σ is such a unifier. $\Box$

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; 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.

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 contains the rules for ordinary unification, there is a derivation finding this mgu σ using only the 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 $\mathrm{HIDE_L}$ 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 $\mathrm{HIDE_R}$ 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 DELETE and elimination rules; 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. □

Our unification algorithm carries term positions along with equations during derivations. Call such an algorithm *decompositional* if for any unifier there is a derivation where every term position in the original terms is associated with some intermediate equation during a derivation. The addition of the imitation rules in our algorithm gives us this property. An inspection of our soundness and completeness arguments will reveal that they follow from the soundness and completeness of the given unification algorithm combined with decompositionality which allows us to perform hiding at any required term position.

If we strengthen the underlying unification algorithm, for example adding in equations (in a sound and complete way) for a desired equational theory then provided we preserve decompositionality the resulting difference unification algorithm will be sound and complete for that theory. For example, adding a *mutate* rule which would decompose an equation $f(s_1, s_2) = f(t_1, t_2)$ to $s_1 = t_2$ and $s_2 = t_1$, for f commutative, extends our unification rules to rules for commutative difference unification. The addition to unification algorithms of rules for equational theories is discussed in [JK91]. As the rules added usually preserve decompositionality, they will usually lead to sound and complete equational difference unification algorithms. An application of equational difference matching is discussed in §8.2.

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 $\langle I, V, F\rangle$ 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. □

6 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.[6] 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 time even in the very restricted of case of ground difference matching.

Theorem 5 *There are difference matching problems requiring exponential time.*

Proof: Consider difference matching (with annotations on the first argument) $\langle s, t\rangle$ 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 $\binom{2n}{n} = (2n!)/((n!)^2)$ 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 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.

6.1 Ground difference matching and unification

Theorem 6 *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.*

[6] 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.

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, giving 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[7] 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. □

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.

6.2 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 linear time. To show NP completeness, we prove that when variables are added determining the existence of a solution is NP hard.

Theorem 7 *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

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

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 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 use the m-ary function h, the 7-ary function 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. □

6.3 Relationship to Other Problems

It is worth comparing difference unification and its subproblems to other known classes of problems. As has previously been observed, the rules for difference unification include variants of the rules for matching and unification; difference unification can be used to solve these problems directly. For matching, $\sigma(s) = t$ if and only if $du(s, t, \{\}, \{\}, \sigma)$ and for unification, $\sigma(s) = \sigma(t)$ where σ is a most general unifier if and only if $du(s, t, \{\}, \{\}, \sigma)$.

There is also a relationship between difference unification and tree embedding. *Tree homeomorphic embedding* (or simply embedding) is the reduction relation associated with the rewrite system comprised of all rules of the form

$$f(x_1, ..., x_n) \rightarrow x_i$$

for $i \in \{1, ..., n\}$ and each function f in the (relevant) signature. The embedding relation is a well ordering on terms, denoted by $s \trianglerighteq t$ if $s \xrightarrow{*} t$ using the above rules. The question of embedding is important in the study of ordering constraints and termination [NS87, BC93]. Difference unification has close connections to several problems related to embedding. The first is the ground embedding problem: given ground terms s and t, $s \trianglerighteq t$ if and only if there is some way of annotating s so that the skeleton is t. In the ground case, this question is identical to asking if s difference matches t where annotations are only allowed on s. Hence, Theorem 6 gives us a polynomial time algorithm for the ground embedding problem. This turns out to be very similar to the algorithm for this problem given in [NS87].

A second problem concerns embedding terms with variables. [BC93] have recently given a decision procedure for the positive existential fragment of tree embedding. That is, is there a substitution σ for a set of inequations with variables $s_i > t_i$ such that $\sigma(s_i) \trianglerighteq \sigma(t_i)$? Using our notation, this is equivalent to asking if there exists a substitution σ and an annotation set A such that

$$skeleton(\sigma(s), A) = \sigma(t).$$

The annotation set may be seen as specifying where the rewrite rules are applied to achieve this embedding. It is instructive to compare this to the specification for difference unification (Equation 5) restricted to annotation on one side which is

$$\sigma(skeleton(s, A_s)) = \sigma(t).$$

This is essentially a sub-problem as we need not consider annotations on terms substituted for variables. For example, $f(x, x, x) \trianglerighteq f(x, a, h(a))$ with the substitution $\{h(a)/x\}$ since $f(h(a), h(a), h(a)) \trianglerighteq f(h(a), a, h(a))$. However, these two terms do not difference unify with the same substitution. Although difference unification is less general, it appears to be a considerably easier problem (though the complexity of Boudet and Comon's algorithm and of embedding terms with variables have not yet been established).

We summarize the above relationships in the following picture. M stands for matching, U for Unification, D for difference, E for embedding, and G for ground. For example, GDM = GE is the class of problems solvable by ground difference matching which is the same as ground embedding.

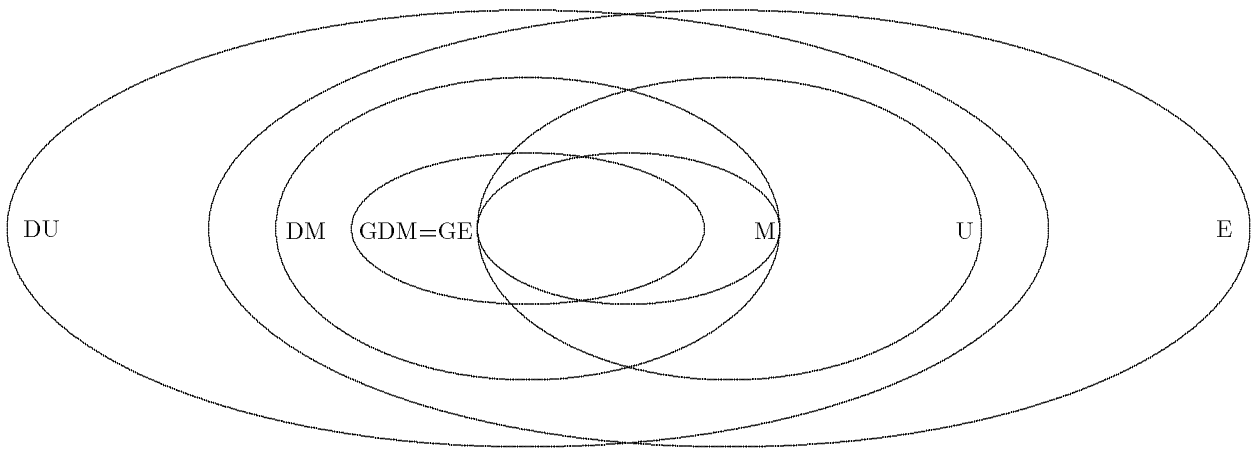

Figure 2: Relationship Between Unification/Embedding Problems

7 Difference Unification in Induction Proofs

7.1 Proof by Consistency

Our first example addresses the integration with so called *proof by consistency* techniques, e.g. [Bac88, Fri86].[8] 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] provision must, however, be made for 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. As an example of the kind of search problems, consider a proof of the distributivity of times over plus:

$$(y + z) \times x = y \times x + z \times x \tag{6}$$

using the rewrite rules

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

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

$$\langle z \times x, 0 \times x + z \times x\rangle \tag{7}$$

$$\langle s(y + z) \times x, s(y) \times x + z \times x\rangle \tag{8}$$

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 (6) 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. We propose 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 (see §7.3 for more on parsing). Primitive recursive definitions can always be parsed as wave-rules. For example, if we have a primitive recursive definition

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

[8] 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.

we can difference unify the two sides yielding the minimal annotation

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

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

$$\boxed{s(\underline{X})} + Y \rightarrow \boxed{s(\underline{X + Y})}$$

and

$$\boxed{s(\underline{X})} \times Y \rightarrow \boxed{Y + \underline{X \times Y}}.$$

Second, new equations generated by deduction steps may be annotated by difference unifying them with the original goal that was superimposed upon. 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 (8) with (6) 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 complete the proof either by rewriting the left hand side with (6) (analogous to what Boyer and Moore call "cross fertilization" in explicit induction) or by removing wave-fronts completely with the wave-rule $\boxed{X + \underline{Y}} = \boxed{X + \underline{Z}} \rightarrow Y = Z$ which yields (6) (analogous to "fertilization" in explicit induction).

To test these ideas, we have selected a set of twelve theorems based on recursively defined functions. Here rev, peq, $half$, len, and $member$ stand for the reverse, equality, halving, length, and membership functions. $<>$ is infix for append. These functions have been defined (in the obvious way) by recursion on the first suitable arguments. These definitions are parsed by difference unification as wave-rules. For example, addition and multiplication are defined as in (7.1) and parsed as the wave-rules given there.

For these theorems, we generated critical pairs by considering superpositions at all complete sets of positions (even when this generated unnecessary critical pairs). All twelve could then be proved using rippling on the rule set augmented

Name	Theorem
APPREV	$rev(X <> Y) = rev(Y) <> rev(X)$
ASSAPP	$(X <> Y) <> Z = X <> (Y <> Z)$
ASSP	$(X + Y) + Z = X + (Y + Z)$
DISTTWO	$(X + Y) \times Z = (X \times Z) + (Y \times Z)$
DOUBLEHALF	$peq(double(half(X)), X) = even(X)$
HALFDOUBLE	$half(double(X)) = X$
LENREV	$len(rev(X)) = len(X)$
LENSUM	$len(X <> Y) = len(X) + len(Y)$
MEMREVA	$member(X, rev(Y)) = member(X, Y)$
REVREV	$rev(rev(X)) = X$
TAILREV	$rev(X <> [Y]) = [Y \vert rev(X)]$
TAILREV3	$rev(rev(X) <> [Y]) = [Y \vert X]$

Table 2: Comparison Theorems

in two cases with additional lemmas that form wave-rules. More precisely, APPREV required the addition of the two wave-rules,

$$\boxed{\underline{X} <> []} = X, \quad \boxed{X <> \underline{Y}} <> Z = \boxed{X <> \underline{Y <> Z}}$$

LENREV required the addition of the wave-rule,

$$len(\boxed{\underline{X} <> [Y]}) = \boxed{s(\underline{len(X)})}.$$

These results suggest that rippling and difference unification can play as important a role in proof by explicit induction as rippling has in proof by explicit induction [BSvH+93]. In the next section, we describe a system which uses failed proof attempts and difference unification to automatically generate additional lemmas like these. For example, using the failed attempt to prove the LENREV theorem from the definitions alone, the system proposes the additional wave-rule needed here.

7.2 Divergence Analysis

A common problem with implicit induction provers is that the need for additional lemmas with which to perform simplification. Without additional lemmas, the interaction of the simplification and deduction rules will often cause divergence. In [Wal94] a critic is described for overcoming divergence in SPIKE, an implicit induction prover. The critic is a program which monitors the output of the theorem prover attempting to identify diverging proof attempts. The critic proposes lemmas which will hopefully allow the proof to go through without divergence. The critic uses difference unification to identify term structure introduced by the deduction rules which is accumulating in an equation and which is preventing simplification. To overcome divergence, the critic then speculates a lemma which moves this accumulating term structure out of the way.

To illustrate the essential ideas, consider again the LENREV theorem,

$$len(rev(X)) = len(X)$$

SPIKE's attempt to prove this equation diverges producing the following sequence of equations.

$$\begin{array}{rcl}
len(rev(X)) & = & len(X) \\
len(rev(X) <> [Y]) & = & s(len(X)) \\
len((rev(X) <> [Y]) <> [Z]) & = & s(s(len(X))) \\
& \vdots &
\end{array}$$

Difference unfiying successive equations identifies the term structure accumulating within len which cannot be simplified and is thus causing divergence.

$$\begin{array}{rcl}
len(rev(X)) & = & len(X) \\
len(\boxed{\underline{rev(X)} <> [Y]}) & = & \boxed{s(\underline{len(X)})} \\
len(\boxed{(\underline{rev(X) <> [Y]}) <> [Z]}) & = & \boxed{s(\underline{s(len(X))})} \\
& \vdots &
\end{array}$$

The critic therefore speculates a wave-rule for simplifying this accumulating term structure. In this case, a wave-rule is speculated which ripples the accumulating term structure to the top of the term where immediate cancellation can occur with the term structure accumulating on the other side of the equality,

$$len(\boxed{\underline{rev(X)} <> [Y]}) = \boxed{s(\underline{len(rev(X))})}.$$

Finally, the divergence critic generalizes the wave-rule using Aubin's primary term heuristic [Aub75]. This gives the wave-rule,

$$len(\boxed{\underline{X} <> [Y]}) = \boxed{s(\underline{len(X)})}.$$

This wave-rule is exactly the lemma needed by SPIKE to overcome divergence and prove the theorem. In addition, the lemma is itself sufficiently simple that it can be proved from the definitions alone without assistance (this is not true of the ungeneralized lemma which causes fresh divergence).

Difference unification is useful in identifying divergence in a wide variety of situations. Some examples are given in Table 3. The predicate $sorted(l)$ is true iff l is a sorted list, $isort$ is insertion sort, $insert$ adds an element to a list in order, $rot(n, l)$ rotates the list l by n elements, and $qrev$ is tail recursive reverse using an accumulator. In each example, SPIKE diverges when given

Theorem	Lemmas Speculated
$len(X <> Y) = len(X) + len(Y)$	$X + s(Y) = s(X + Y)$
$rot(len(X), X) = X$	$rot(len(X), X <> [Y]) = Y :: rot(len(X), X)$
$rev(rev(X)) = X$	$rev(X <> [Y]) = Y :: rev(X)$
$rev(qrev(X, [])) = X$	$rev(qrev(X, [Y])) = Y :: rev(qrev(X, []))$
$len(qrev(X, Y)) = len(X) + len(Y)$	$X + s(Y) = s(X + Y)$
$len(isort(X)) = len(X)$	$len(insert(Y, X)) = s(len(X))$
$sorted(isort(X))$	$sorted(insert(Y, X)) = sorted(X)$

Table 3: Some theorems proved by SPIKE using divergence analysis and lemma speculation.

just the definitions. However, using difference unification, the critic is able to identify the cause of divergence and speculate a which lemma allows proof to go through (and can itself be proved without assistance by SPIKE). In [Wal94] a comparison is made with other techniques for divergence analysis based on generalization [DP90, TJ89, TW3]. It is shown that divergence analysis based on difference unification is strictly more powerful since difference unification can identify nested accumulating term structure whilst that based on most specific generalization cannot.

7.3 Parsing

Rippling uses wave-rules, annotated and orientated rewrite rules. [BSvH+93, BW94] explain how these rules may be created (or "parsed") given (ordinary) unannotated rewrite rules. In particular, a wave-rule may be formally defined to be any annotated rewrite rule which is structure preserving and which leads to terminating rewriting. These properties may be established independently.

Structure preservation is defined so that if t ripples to t' then both share the same skeleton. A sufficient condition for this is that the left and right hand side of each wave-rule has the same skeleton when considered as ground terms (i.e., variables are considered as constants). Hence, to parse rewrite rules, we use ground difference unification. In practice, we also only need consider annotations returned from minimal matches. For example, there are three minimal matches of

$$(X + Y) + Z \rightarrow X + (Y + Z)$$

corresponding to the first three unifiers of table 3.

$$\boxed{\underline{(X + Y)} + Z} \quad \rightarrow \quad X + \boxed{(\underline{Y} + Z)} \tag{9}$$

$$\boxed{(X + \underline{Y})} + Z \quad \rightarrow \quad \boxed{X + \underline{(Y + Z)}} \tag{10}$$

$$\boxed{(\underline{X} + Y)} + Z \quad \rightarrow \quad X + \boxed{(Y + \underline{Z})} \tag{11}$$

The second criteria for wave-rules, namely termination, may be ensured by choosing an appropriate ordering $>$ on annotated terms and checking that for each annotated rewrite rule $l \to r$ returned by ground difference unification that $l > r$. [BW94] presents a number of simple orderings that measure position of wave-fronts relative to the term skeleton and are used to insure that wave-fronts make some well-founded notion of progress. For example, all of the wave-rules used in this paper make progress moving wave-fronts towards the root of the term (that is, upward in the skeleton). Under such an ordering, only (10) above would be admitted as a wave-rule [BW94] provides the full account of wave-rule parsing based on difference unification.[9] A parser based on ground difference unification is currently being incorporated into the CLAM system.

8 Difference Unification in Non-inductive Proofs

8.1 Series

We have had considerable success applying the combination of difference unification and rippling in non-inductive domains. One such domain is finding closed form sums for arithmetic series. That is, given a series, finding an equivalent expression built from simpler arithmetic functions not including summation. This domain typifies those in which we feel difference unification can make a significant contribution to automating reasoning. As in inductive theorem proving, the search space is usually infinite yet mathematicians can often solve problems easily. In induction, mathematicians are guided by the fact that the induction hypothesis should be used during proof; rippling restricts the search space to proof steps that reduce the difference between the induction hypothesis and the induction conclusion. The situation is analogous with summing series. Mathematicians have knowledge about or access to collections of standard results about series (e.g., [AS65]). Their intuition often suggests similarities between a problem to be solved and known results. A proof then consist of transforming the given problem into a form where a known result may be applied. We have used difference unification and rippling to mechanize this strategy: the problem is difference unified against known series and rippling is used to perform the subsequent transformation.

As a simple example, consider the problem of finding a closed form sum for

$$\sum_{j=0}^{m} \sum_{k=0}^{n} k \times j + c$$

using the standard results for the sum of integers and of a constant.

$$\sum_{i=0}^{N} i \quad = \quad \frac{N \times s(N)}{2} \tag{12}$$

[9] The full story is rather more complex. In particular, we describe how stable, monotone, well-founded orderings for rippling are defined, and how wave-fronts can be associated with directions relative to these orderings. For example, under an ordering where wave-fronts make progress moving downwards, (9) may be admitted as a wave-rule. And an ordering where wave-fronts may be moved both upwards and downwards (and have markings to indicate which direction they are moving) may be used to orient (11). However, ground difference unification is used to guarantee structure preservation in all cases.

$$\sum_{i=0}^{N} C \quad = \quad s(N) \times C \tag{13}$$

In [WNB92], we express the problem of finding a closed form sum as the task of proving a theorem of the form,

$$\exists S \;.\; \sum_{j=0}^{m} \sum_{k=0}^{n} k \times j + c \quad = \quad S$$

where the existential witness S is restricted to be in closed form.

As with induction, to prove such theorems we use a set of *methods*. A method describes part of a strategy for proving a theorem. A method is specified by a set of preconditions and a set of postconditions. The CLAM system puts methods together by matching these conditions using standard AI search techniques (e.g., depth-first, iterated deepening, breadth-first, ...). In the example given, CLAM first applies the *standard form* method [WNB92]. This difference unifies the dequantified goal with (12) giving the following minimal annotation on the goal.

$$\sum_{j=0}^{m} \boxed{\sum_{k=0}^{n} k \times \underline{j} + c} = S$$

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

$$\sum_{j=A}^{B} \boxed{\sum_{k=C}^{D} \underline{U}} \quad \rightarrow \quad \boxed{\sum_{k=C}^{D} \underline{\sum_{j=A}^{B} U}} \tag{14}$$

$$\sum_{j=A}^{B} \boxed{C \times \underline{U}} \quad \rightarrow \quad \boxed{C \times \underline{\sum_{j=A}^{B} U}} \tag{15}$$

$$\sum_{j=A}^{B} \boxed{\underline{U} \times C} \quad \rightarrow \quad \boxed{\underline{\sum_{j=A}^{B} U} \times C} \tag{16}$$

$$\sum_{j=A}^{B} \boxed{\underline{U} + V} \quad \rightarrow \quad \boxed{\underline{\sum_{j=A}^{B} U} + \sum_{j=A}^{B} V} \tag{17}$$

where C and D are constant with respect to j. Note that (14) 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 applies wave-rule (14) to the goal,

$$\boxed{\sum_{k=0}^{n} \underline{\sum_{j=0}^{m} \boxed{k \times \underline{j} + c}}} = S$$

followed by wave-rule (17),

$$\boxed{\sum_{k=0}^{n} (\underline{\sum_{j=0}^{m} \boxed{k \times \underline{j}} + \sum_{j=0}^{m} c})} = S$$

and finally (15), after which rippling no longer applies,

$$\boxed{\sum_{k=0}^{n}(k \times \underline{\sum_{j=0}^{m} j} + \sum_{j=0}^{m} c)} = S$$

The differences between the goal and the standard results have now been moved out of the way so we can substitute with both the standard results to give,

$$\sum_{k=0}^{n}(k \times \frac{m \times s(m)}{2} + s(m) \times c) = S$$

Since the summation sign has not yet been eliminated, the *standard form* method again difference unifies the goal with (12), giving the minimally annotated goal,

$$\sum_{k=0}^{n} \boxed{(\underline{k} \times \tfrac{m \times s(m)}{2} + s(m) \times c)} = S$$

Rippling with (17) gives,

$$\boxed{\underline{\sum_{k=0}^{n} \boxed{(\underline{k} \times \tfrac{m \times s(m)}{2})}} + \sum_{k=0}^{n} s(m) \times c} = S$$

And with (16),

$$\boxed{(\underline{\sum_{k=0}^{n} k}) \times \tfrac{m \times s(m)}{2} + \sum_{k=0}^{n} s(m) \times c} = S$$

We can now substitute with both the standard results to give the closed form,

$$\frac{n \times s(n)}{2} \times \frac{m \times s(m)}{2} + s(n) \times s(m) \times c = S$$

In [BNT92] several other methods for summing series are described. To our surprise, difference unification and rippling played a key role in all of the methods. Table 4 gives a sample of series summed using these methods. Most previous work on summation has been on verification (that a series sums to a given closed form) or on decision procedures (black boxes that return a closed form without an explicit proof). By comparison these methods solve the synthesis problem of finding a closed form. In addition, being heuristic, they can work on problems outside the range of state of the art decision procedures like Gosper's algorithm [Gos77].

8.2 Goal Directed Equational Reasoning

Hutter, motivated by [Bun88], has developed independently of [BSvH$^+$93] a calculus for manipulating annotated terms [Hut90, Hut]. His calculus shares

Series	Closed Form
$\sum i^2$	$\frac{2.n^3+3.n^2+n}{6}$
$\sum a^i$	$\frac{a^{s(n)}-1}{a-1}$
$\sum \frac{1}{i.(i+1)}$	$\frac{n}{s(n)}$
$\sum F_i$	$F_{n+2} - 1$
$\sum \begin{pmatrix} s(i) \\ s(m) \end{pmatrix}$	$\begin{pmatrix} s(n) \\ s(s(m)) \end{pmatrix} + \begin{pmatrix} s(n) \\ s(m) \end{pmatrix}$

All sums are from 0 *to* n*,* $a \neq 1$*,* F_i *is the ith Fibonacci number.*

Table 4: Some series summed by CLAM using difference unification and rippling.

strong similarities with the Edinburgh calculus. Hutter's calculus features annotated terms where the function symbols are colored to indicate if they are part of the skeleton or if they may be changed during rewriting. Analogous to wave-rules he has developed skeleton preserving rewrite rules and a calculus to apply these rules.

In [CH94], a paramodulation based resolution calculus is presented for equality reasoning based on ideas very similar to ours. In their work, the goal is difference matched against the available equational axioms. Each match yields an annotation of the goal, namely, one with a skeleton identical to an instance of the axiom matched. Rippling is then used to apply those equational axioms which are (structure preserving) wave-rules. Finally, the matched axiom is used as a rewrite rule, and the process may be repeated again if necessary. The matched axioms are called *bridge lemmas* and are used to direct the proof in a goal directed way. This methodology is essentially the same as the one we used for summing series.

[CH94] show how this methodology can be applied to solve equality reasoning problems where the search space is normally very large. The difference match and ripple methodology is applied to problems in arithmetic, in particular, to the challenge problem of SAM's lemma [McC88]. The proof given is direct, although an intersting twist is that as the function symbols involved are associative, commutative, and idempotent, a difference matcher is used that builds in these axioms. (See discussion of this in §5.)

8.3 Learning Recursive Programs

Our final example is rather different; it comes from the machine learning community. Lapoint and Matwin have invented a technique called *sub-unification*. This is used as the basis of a system for learning logic programs from examples [SM92]. Sub-unification is strictly weaker than difference unification. The input to sub-unification is a pair of terms t_1 and t_2. If some subterm s of t_2 unifies with t_1 with substitution σ, then the output is essentially t_2 in a wavefront with s underlined, and the unifier σ. For example $t_1 = f(a,b)$ and $t_2 = h(g(a, f(a,X)))$ subunify with substitution $\{b/X\}$ and t_2 annotated as $\boxed{h(g(a, \underline{f(a,X)}))}$. This is a restricted case of difference unification where only one term is annotated, and only at the root.

The idea behind their work is to synthesize recursive logic programs by comparing the structure of terms in a set of training examples. Certain classes of logic programs can be seen as simplifying complex terms until some base cases is reached. Lapoint and Matwin use sub-unification to determine the structural differences of two arguments, say t_1 and t_2, and use this difference to suggest a likely depth and structure of recursion necessary for a logic program clause to simplify t_2 to t_1. They are able to use this technique to synthesize members of a simple class of logic programs. Their results are comparable with many inductive logic programming systems presented in the literature.

As a simple example, consider learning the *append* relation from the two training examples, $append([], L, L)$ and $append(a :: [], b :: [], a :: b :: [])$ where :: is infix cons. Difference unifying these terms gives the minimal annotations,

$$append(\boxed{a :: \underline{[]}}, b :: [], \boxed{a :: \underline{b :: []}}),$$

and the substitution of $b :: []$ for L. These annotations indicate the term structure changed in a recursive call, whilst the skeleton indicates the term structure left invariant. Lapoint and Matwin's system uses this information and generalization heuristics to propose the recursive clause,

$$append(\boxed{H :: \underline{T}}, L, \boxed{H :: \underline{S}}) : -append(T, L, S).$$

This is checked for consistency with the examples. Combined with the base case, $append([], X, X)$, this defines the *append* relation completely. Their system is also able to learn other relations like $member/2$, $delete/3$, $extractN/3$ (extract Nth element from a list), $factorial/2$ and $reverse/2$.

Since difference unification is more general than sub-unification, it can be used to learn a larger class of logic programs. Consider for example, a relation for rotating a list by n elements. We are supplied with two training examples, $rotate(0, L, L)$, and $rotate(s(0), a :: b :: c :: [], b :: c :: a :: [])$. If we difference unify the two examples, we get the minimal annotations,

$$rotate(\boxed{s(\underline{0})}, \boxed{a :: \underline{b :: c :: []}}, b :: c :: \boxed{a :: \underline{[]}}),$$

and the substitution of $b :: c :: []$ for L. This identifies the invariant term structure (*e.g.* the *cdr* of the input list $a :: b :: c :: []$) and the term structure that needs to be changed (*e.g.* the *car* of the input list, which needs to be moved to the end of the list). An appropriate program for *rotate* could therefore be learnt with a recursive step case which has this invariant term structure and which achieves this change. It is still quite a challenge to determine how to achieve the desired change. In this case, one must invent a function for appending onto the end of a list. Difference unification has, however, at least identified the change that needs to be achieved. Note that the wavefront in the third argument position is not outer-most. Sub-unification is therefore of little use in identifying the term structure changed and that left invariant in a recursive call. At best, sub-unification will identify the wavefronts,

$$rotate(\boxed{s(\underline{0})}, \boxed{a :: b :: c :: \underline{[]}}, \boxed{b :: c :: a :: \underline{[]}})$$

These non-minimal annotations fail to indicate the required changes to the term structure in a single recursive call.

Observe that learning recursive programs is closely related to divergence analysis. In both we are presented a sequence of relations, and must identify invariant term structure, and speculate a wave-rule with this invariant. It is not therefore surprising that difference unification has an important role in both problems.

9 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.

Acknowledgements

The first author was funded by the German Ministry for Research and Technology (BMFT) under grant ITS 9102. The second author was supported by a SERC postdoctoral fellowship and a EU Human Capital and Mobility postdoctoral fellowship. The responsibility for the contents of this publication lies with the authors. We thank a number of our collegues for discussions, comments, and criticisms on this work. In particular, we thank Alan Bundy, Andrew Ireland, and Pierre Lescanne.

References

[AS65] M. Abramowitz and I. A. Stegun, editors. *Handbook of mathematical functions*. Dover, 1965.

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

[BBH92] R. Barnett, D. Basin, and J. Hesketh. A recursion planning analysis of inductive completion. Technical Report MPI-I-92-230, Max-Planck-Institut für Informatik, 1992. To appear in the Annals of Mathematics and Artificial Intelligence.

[BC93] A. Boudet and H. Comon. About the theory of tree embedding. In Marie-Claude Gaudel and Jean-Pierre Jouannaud, editors, *TAPSOFT '93: Theory and Practice of Software Development, 4th Int ernational Joint Conference CAAP/FASE*, LNCS 668, pages 376–390, Orsay, France, April 13–17, 1993. Springer-Verlag.

[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.

[BHHW86] S. Biundo, B. Hummel, D. Hutter, and C. Walther. The Karlsruhe induction theorem proving system. In *8th International Conference On Automated Deduction*, Oxford, UK, 1986.

[BM79] R.S. Boyer and J. Strother Moore. *A Computational Logic*. Academic Press, 1979.

[BNT92] A. Bundy, A. Nunes and T. Walsh. The Use of Proof Plans to Sum Series. In *Proc. of 11th International Conference On Automated Deduction (CADE-11)*, Saratoga Springs, New York, June 1992. Springer-Verlag.

[BS88] K. Hans Bläsius and J.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+93] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.

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

[BvHHS90] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M.E. Stickel, editor, *10th International Conference* *on Automated Deduction*, pages 647–648. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.

[BvHS91] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324, 1991.

[BW92] D. Basin and T. 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.

[BW93] D. Basin and T. Walsh. Difference unification. In *Proceedings of the 13th IJCAI.* International Joint Conference on Artificial Intelligence, 1993. Also available as Technical Report MPI-I-92-247, Max-Planck-Institute für Informatik.

[BW94] D. Basin and T. Walsh. Termination Orderings for Rippling. To appear in the *Proc. of 12th International Conference On Automated Deduction (CADE-12)*, 1994.

[CH94] Jürgen. Cleve and Dieter Hutter. A methodology for equational reasoning. In *HICSS-27.* IEEE, 1994. To appear.

[DP90] N. Dershowitz and E. Pinchover. Inductive Synthesis of Equational Programs. In *Proceedings of the 8th National Conference on AI*, pages 234–239. American Association for Artificial Intelligence, 1990.

[Dig85] V.J. Digricoli. The management of heuristic search in boolean experiments with RUE resolution. In *9th IJCAI*, pages 1154 – 1161, Los Angeles, California, 1985.

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

[Gos77] R.W. Gosper. Indefinite hypergeometric sums in MACSYMA. In *Proc. MACSYMA Users Conference*, pages 237–252, 1977.

[Gra90] B. Gramlich. UNICOM: a refined completion based inductive theoremprover. In M.E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 655–656. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449.

[Hut] D. Hutter. Colouring terms to control equational reasoning. An Expanded Version of PhD Thesis: Mustergesteuerte Strategien für Beweisen von Gleichheiten (Universität Karlsruhe, 1991), in preparation.

[Hut90] D. 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.

[JK91] J.-P. Jouannaud and C. 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*, pages 257 – 321. MIT Press, 1991.

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

[McC88] W. McCune. Challenge equality problems in lattice theory. In Ewing L. Lusk and Ross A. Overbeek, editors, *9th International Conference on Automated Deduction*, pages 704–709, 1988.

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

[Mus80] D. R. Musser. On proving inductive properties of abstract data types. In *ACM Symposium on Principles of Programming Languages*, pages 1154–162, Las Vegas, 1980.

[NS87] P. Narendran and J. 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.

[Red90] U.S. Reddy. Term rewriting induction. In *Proc. of 10th International Conference On Automated Deduction (CADE-10)*, pages 162–177, Kaiserslautern, Germany, July 1990. Springer-Verlag.

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

[SM92] L. Stéphane and S. Matwin. Sub-unification: A tool for efficient induction of recursive programs. In *Proceedings of the 9th International Workshop on ML*, pages 273–281. Morgan Kaufmann, 1992.

[TJ89] M. Thomas and K.P. Jantke. Inductive Inference for Solving Divergence in Knuth-Bendix Completion. In *Proceedings of International Workshop AII'89*, pages 288–303, 1989.

[TW3] M. Thomas and P. Watson. Solving divergence in Knuth-Bendix completion by enriching signatures. *Theoretical Computer Science*, 112:145–185, 1993.

[Wal94] T. Walsh. A Divergence Critic. To appear in the *Proc. of 12th International Conference On Automated Deduction (CADE-12)*, 1994.

[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.