Difference Matching*

David Basin
Max-Planck-Institut für Informatik
Im Stadtwald, Saarbrücken, Germany

Toby Walsh
Department of AI, Edinburgh University
Edinburgh, Scotland

Abstract

Difference matching is a generalization of first-order matching where terms are made identical both by variable instantiation and by structure hiding. After matching, the hidden structure may be removed by a type of controlled rewriting, called rippling, that leaves the rest of the term unaltered. Rippling has proved highly successful in inductive theorem proving. Difference matching allows us to use rippling in other contexts, e.g., equational, inequational, and propositional reasoning. We present a difference matching algorithm, its properties, several applications, and suggest extensions.

1 Introduction

A central problem in theorem proving is showing that one equality follows from others or more generally that one formula is entailed by others. Many techniques have been developed for this purpose, for example, the use of canonical sets of rewrite rules and resolution theorem proving. Within inductive theorem proving, this problem arises in the proof that the induction conclusion follows from the induction hypothesis. Boyer and Moore, in their theorem prover NQTHM [4, 5], approach this problem essentially by normalizing both the induction hypothesis and conclusion using the same set of rewrite rules. Their approach works remarkably well; however, their normalization procedure (actually, a combination of procedures) is complex, contains a large amount of heuristic information, and is difficult to detach from their theorem prover itself. Motivated by their success, Bundy suggested in [7] an alternative approach based not on exhaustive heuristic based normalization but instead upon the application of structure preserving rewrite rules; he called this rewriting *rippling.*[1]

The ideas behind rippling are fairly straightforward. 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 first author was supported, while at Edinburgh, by SERC grant GR/F/71799, the second by a SERC PostDoctoral Fellowship. We would like to thank the other members of the Edinbrugh Mathematical Reasoning Group for their feedback on this project.

[1]The name rippling comes from *rippling-out* a term coined by Aubin [1], 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 [10]) recursive functions in the induction conclusion, preserving the structure of the induction hypothesis while unfolding.

We call these function symbols *wave-fronts* (terminology and notation will be formally defined in Section 2). The rest of the induction conclusion, which is an exact image of the induction hypothesis is called the *skeleton*. For example, if we wish to prove that a proposition $P(x)$ is true for all natural numbers, we assume it is true for n and attempt to show it is true for $s(n)$. That is, we show $P(s(n))$ follows from the hypothesis $P(n)$. The hypothesis and the conclusion are identical except for the successor function applied to the induction variable n. We mark this wave-front by placing a box around it, and underlying 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:

$$\alpha(\boxed{\beta(\underline{\gamma})}) \quad \Rightarrow \quad \boxed{\rho(\underline{\alpha(\gamma)})}$$

By design, the skeleton $\alpha(\gamma)$ remains unaltered by their application. We eventually hope to rewrite the conclusion, $P(\boxed{s(\underline{n})})$ using such wave-rules into some function of $P(n)$; that is, into $\boxed{f(\underline{P(n)})}$ (f may be the identity) and then call upon the induction hypothesis.

Rippling, and its extensions (e.g., rippling in other directions than out, and more general forms of wave-rules) are presented in [8] and are employed in the Oyster-Clam system. Many of the same ideas were developed independently by Hutter [14], from ideas in [7], and employed in the INKA system. Both systems have enjoyed a high degree of success. Clam, for example, has been tested on 68 examples from the Boyer-Moore corpus and proved all but 3 of them.[2] This success seems to stem from three properties of rippling (a more detailed analysis of these properties can be found in [9]):

1. Rippling involves little search. In rippling, the wave-fronts in the pattern must correspond to wave-fronts in the instance. This leads to a very controlled application of rewrite rules that gives very low branching rates.

2. Rippling terminates. Rippling always makes progress moving wave-fronts in a desired direction; hence termination is guaranteed, even when applying rewrite rules that would normally, without wave-front annotation, lead to loops.

3. Rippling applies only "good" rewrites. As wave-rules are structure preserving, if rippling terminates successfully, the hypothesis can be used to prove or simplify the conclusion. This use of induction hypotheses is often called *fertilization*.

From these properties, we can see that rippling is an attractive kind of rewriting. Inductive theorem proving is well suited for rippling because, in the induction step, there is a natural annotation of the conclusion so that its skeleton is identical to the induction hypothesis.[3] However, there is no reason why rippling cannot be used as a

[2]The failures resulted from essentially trivial features of the object-level logic, Martin-Löf type theory, not from limitations of rippling itself.

[3]This is for constructor style induction schemas; for destructor schemas one could ripple on hypotheses or move the wave-fronts into the conclusion. See [9].

general rewrite procedure when one term has the same skeleton as another; its only precondition is that it is applied to a term with wave-front annotations.

In this report we introduce difference matching, a procedure that supports the general application of rippling in theorem proving and term simplification. A difference matcher takes as inputs two terms (or formulas) s and t. It returns s annotated with wave-fronts, and a set of substitutions such that the skeleton of the annotated term equals t under substitution. Although difference matching generalizes first-order matching, it is much more than matching. It is an attempt to make two terms identical not just by variable instantiation, but also by structure hiding; the hidden structure is the part of the term within the wave-front that serves to direct rippling.

A brief example will illustrate the spirit of how difference matching and rippling can be combined. The example is simple enough to describe quickly (more complex examples are given in the experience section) and has the virtue of illustrating an interesting special case of difference matching where there are no match variables to be instantiated. Unlike first-order matching, which degenerates to syntactic identity, difference matching remains non trivial and interesting in the absence of match variables. We will, however, give examples of difference matches involving variable instantiation in Section 4.

Suppose we wish to demonstrate that

$$s(y) + x < s(y) * s(y) \tag{1}$$

follows from

$$x < y * y. \tag{2}$$

Here x and y are skolem constants and s is the successor function. An informal proof might first add $s(y)$ to each side of Equation 2 and then y to the lefthand side.

$$\begin{array}{rcl} s(y) + x & < & s(y) + y * y \\ & < & s(y) + y * y + y \\ & = & s(y) * s(y) \end{array}$$

Hence Equation 1 follows. Automating such a proof would seem to require search to select which terms to add. It would also require reasoning about the properties of addition, multiplication, and basic monotonicity and inequality reasoning.

An alternative is to use difference matching and rippling. That is, to find an annotation of Equation 1 so that its skeleton is the same as Equation 2; afterwards, rippling will hopefully reduce the equation to its skeleton which is our given hypothesis.

We shall need some basic wave-rules that relate plus, times, and less-than.

$$\boxed{Z + \underline{X}} < \boxed{Z + \underline{Y}} \quad \Rightarrow \quad X < Y \tag{W_1}$$

$$X < \boxed{\underline{Y} + Z} \quad \Rightarrow \quad X < Y \tag{W_2}$$

$$\boxed{s(\underline{X})} * Z \quad \Rightarrow \quad \boxed{\underline{X * Z} + Z} \tag{W_3}$$

$$Z * \boxed{s(\underline{X})} \quad \Rightarrow \quad \boxed{Z + \underline{Z * X}} \tag{W_4}$$

Note that $\Rightarrow$ stands for rewriting as opposed to implication. Since we will reason backwards, a wave-rule $\phi \Rightarrow \psi$ is justified by an implication $\psi \rightarrow \phi$; that is, if we can prove ψ, then we can also prove ϕ. For example, W_2 is a valid rule only in the direction shown. These wave-rules could be supplied as ordinary rewrite rules and have wave-front annotation automatically added (as in Edinburgh Clam system).[4] Wave-rules can also be automatically synthesized from function definitions (as in the INKA system and also in Clam).

Difference matching Equation 1 with 2 will annotate the former as

$$\boxed{s(y) + \underline{x}} < \boxed{s(\underline{y})} * \boxed{s(\underline{y})}$$

and rippling will apply wave-rules W_4, W_1, W_3, and W_2 to the above which results in the following inequalities.

$$\begin{aligned} \boxed{s(y) + \underline{x}} &< \boxed{s(y) + \underline{\boxed{s(\underline{y})} * y}} \\ x &< \boxed{s(\underline{y})} * y \\ x &< \boxed{\underline{y * y} + y} \\ x &< y * y \end{aligned}$$

Since the final inequality is Equation 2, our hypothesis, the proof is completed. Note that explosive search was not required as rippling provided the key ideas needed to transform the conclusion into its skeleton, the hypothesis. More complex examples of this kind of rewriting are found in Section 4.

The idea behind the combination of difference matching and rippling is rather general. The combination captures a basic problem solving strategy of finding differences or mismatches between terms and working to eliminate those differences. Difference identification and reduction are also central themes in the research of [3, 11, 16]. In their work, a partial unification results in a special kind of resolution step (E and RUE-resolution) where the failure to completely unify gives rise to new inequalities that represent the differences between the two terms. This leads to a controlled application of equality reasoning where paramodulation is only used when needed. This is analogous to our use of difference matching and rippling as a means of controlling the use of general rewrite rules. A different strategy for removing differences between two terms is to seek a generalization for the two; there may also be connections between difference matching (and its extensions) to anti-unification[18], although we have not explored this direction.

The remainder of our paper is organized as follows. In Section 2, we present definitions and necessary background. In Section 3, we present the formal properties required of a difference match and give an algorithm which returns exactly the matches with these properties. We analyze some other properties of our algorithm, and formally prove its correctness. In Section 4, we report on experience using

[4]Automatically annotating rewrite rules as wave-rules can be seen as a special case of difference unification (see Section 5) between the right and left hand sides of the rewrite rule with universally quantified variables treated as constants. Clam implements such "wave-rule parsing" by simply enumerating annotations and comparing resulting skeletons.

difference matching in finding the sums of series. These results are interesting not only as exercises in equality reasoning, but also in demonstrating the use of difference matching and rippling in solving non-inductive problems. In the final section, we sketch directions for extensions and draw conclusions.

2 Terms and Annotation

Terms

The algorithm we present acts on both terms and formulas. To facilitate this we work in a simple first-order sorted framework (as in [13]) in which sorts can be used to distinguish between such syntactic categories. This is important so that we can guarantee the well-formedness of skeletons.

A *sort* is the name of a set of syntactic entities called *constants*. For $\mathcal{S}$ a set of sorts, *types* are specified by finite sequences of members of $\mathcal{S}$ which we shall write as $S_1 \times .... \times S_n \to S$ where the S_i range over $\mathcal{S}$ (notational conventions are given at end of this section). A *signature* Σ *over* $\mathcal{S}$ consists of a set Σ of operators and a typing function, *Ty*, from operators to types. For the remainder of this paper we shall assume that we are given a set of sorts $\mathcal{S}$ and a signature Σ over $\mathcal{S}$.

Object level variables of sort s are members of V_S, where V_S is one of a S-indexed family of sets. We let V denote the union of these sets. *Object level terms of sort s* are members of the set $\mathcal{T}_S(\Sigma, \mathcal{V})$ which is the smallest set of terms that includes V_S and for all $f \in \Sigma$, if $Ty(f) = S_1 \times ... \times S_n \to S$ then $f(t_1, ..., t_n)$ is in $\mathcal{T}_S(\Sigma, \mathcal{V})$ whenever $t_i \in \mathcal{T}_{S_i}(\Sigma, \mathcal{V})$. The *terms over* Σ are members of $\mathcal{T}(\Sigma, \mathcal{V})$, which is the S-indexed union of the $\mathcal{T}_S(\Sigma, \mathcal{V})$. Whenever possible, we will omit sorts and types when these are immaterial or context makes our meaning clear.

Let $\mathcal{M}_S$ be a denumerable set of meta-variables of sort S and $\mathcal{M}$ the union of these sets. These variables may be thought of (and are sometimes referred to in the literature) as *match-variables* or *logic-variables.* A *meta-level term (of sort S)* is a member of $\mathcal{T}(\Sigma, \mathcal{V} \cup \mathcal{M})$ (respectively $\mathcal{T}_S(\Sigma, \mathcal{V} \cup \mathcal{M})$. We shall call both object and meta-level terms simply *terms* when there is no danger of confusion. A *ground meta-level term* is one containing no meta-variables, although it may contain members of $\mathcal{V}$, and is ground from the standpoint of the matching algorithm that we will define. Indeed, in defining this difference matching and analyzing its correctness, we will simplify matters by treating members of $\mathcal{V}_S$ as if they were constants of sort S. Given a meta-level term t, $Vars(t)$ will denote the members of $\mathcal{M}$ that occur in t.

Annotation

An *object-level W-term* is a member of $\mathcal{T}(\Sigma, \mathcal{V})$ that may contain contain *wave-fronts.* A wave-front is a term t with a proper sub-term t' deleted. The deleted subterm may itself contain wave-fronts. We represent it by an annotation of t which encloses t in a box and underlines the sub-term t'. We let $\mathcal{WT}(\Sigma, \mathcal{V})$ represent the set of such W-terms. For example, we can annotate $f(a, b)$ as $\boxed{f(\underline{a}, b)}$ when $Ty(f) = S_1 \times S_2 \to S_1$. A meta-level W-term is defined analogously to object-level W-terms except over $\mathcal{T}(\Sigma, \mathcal{V} \cup \mathcal{M})$.

There are various alternative ways W-terms may be represented on paper or in a computer. We shall display them using the new Edinburgh "box-and-hole" notation[9]; however, the older Edinburgh "box" notation[8] and Hutter's representation of his C-terms[14] are other possible options. All that we require of any implementation is that we can tell which parts of the term are "deleted" by a wave-front and hence do not contribute to the skeleton. To this end, if t is a subterm of a W-term, let the predicate $\mathit{InWave}(t)$ hold when the entire subterm is within a wave-front and $\mathit{HdInWave}(t)$ hold if t is a function application and the leading function symbol is within a wave-front. For example, if t is $\boxed{f(\underline{a}, b)}$ then $\mathit{HdInWave}(t)$ and $\mathit{InWave}(b)$ both hold and neither $\mathit{HdInWave}(a)$ nor $\mathit{InWave}(a)$ hold. In the remainder of this paper we will not distinguish between annotations that appear different but are indistinguishable to the predicates *InWave* and *HdInWave*. E.g., $\boxed{s(s(\underline{0}))}$ and $\boxed{s(\underline{\boxed{s(\underline{0})}})}$. This is sensible as in systems like Clam and INKA it is possible to represent internally both annotated terms and wave-rules in a normal form where each wave-front has an immediate subterm deleted (e.g., the second term in the above example) and there is no loss of generality in rippling with such a representation. Our algorithm, given in the following section returns terms annotated this way.

We now define two functions from W-terms into terms. The first, *Skel*, returns the unannotated (not boxed) part of an annotated term. It is defined recursively by cases. Since the definition of W-terms disallows an annotated constant or variable to itself be a W-term (since there are no proper subterms to be "deleted") we must have in the base case that $\mathit{Skel}(X) = X$ when X a constant, variable, or meta-variable. In the step case, if the leading function is not in a wave-front (that is, if $\neg \mathit{HdInWave}(f(t_1, \ldots, t_n)))$ we have

$$\mathit{Skel}(f(t_1, \ldots, t_n)) = f(\mathit{Skel}(t_1), \ldots, \mathit{Skel}(t_n));$$

alternatively if $\mathit{HdInWave}(f(t_1, \ldots, t_n))$, it must be the case that for some $j \in \{1..n\}$ $\neg \mathit{InWave}(t_j)$, and then

$$\mathit{Skel}(f(t_1, \ldots, t_n)) = \mathit{Skel}(t_j).$$

For example,

$$\mathit{Skel}(f(\boxed{f(\underline{a}, b)}, b)) = \mathit{Skel}(\boxed{f(\underline{f(a, b)}, b)}) = f(a, b)$$

Skeleton preserving rewrite rules are called *wave-rules* by Bundy and *C-equations* by Hutter.

The second function we define, *Erase*, simply removes wave-front annotation but leaves the term otherwise unchanged. E.g., $\mathit{Erase}(f(\boxed{f(\underline{a}, b)}, b)) = f(f(a, b), b)$. For a W-term t, we call the term computed by $\mathit{Erase}(t)$ the *body* of t and the part returned by $\mathit{Skel}(t)$ its *skeleton*. Note that, since wavefronts are deleted, the skeleton may not well-typed. Additionally, even if a skeleton can be typed, its type may be different to that of the body. In the rest of the paper, we shall restrict ourselves to W-terms t whose skeletons are well-typed and in which $\mathit{Ty}(\mathit{Skel}(t)) = \mathit{Ty}(\mathit{Erase}(t))$.

A substitution is a sort respecting partial function of type $\mathcal{M} \to \mathcal{T}(\Sigma, \mathcal{V})$ that takes values on only finitely many members of $\mathcal{M}$. Substitutions are extended to terms in the standard way. If σ is a substitution, let $Dom(\sigma)$ be those elements of $\mathcal{M}$ for which σ takes a value and $\sigma^\circ(\alpha)$ the value of σ on α. We shall represent a substitution σ by a finite set of pairs $\{\langle \alpha_1, t_1 \rangle, \ldots \langle \alpha_n, t_n \rangle\}$ where $t_i \in \mathcal{T}_{S_i}(\Sigma, \mathcal{V})$, $\alpha_i \in \mathcal{M}_{S_i}$. We combine substitutions σ_1 and σ_2 using the operator $\cup$ which takes the set-theoretic union of the sets representing the two substitutions. This returns a well-defined substitution provided that

$$\forall \alpha \in (Dom(\sigma_1) \cap Dom(\sigma_2)).\, \sigma_1^\circ(\alpha) = \sigma_2^\circ(\alpha)$$

For brevity, we will write this condition as $compatible(\sigma_1, \sigma_2)$. We shall also write Sub to represent the set of substitutions.

Notation

We will use the following notational conventions. We shall let S range over sorts; τ will range over types; σ will range over substitutions; $a, b, c, \ldots$ will range over (possibly annotated) constants; $f, g, h, \ldots$ will range over (possibly annotated) members of Σ; $x, y, z, \ldots$ will range over (possibly annotated) members of Var; Greek letters such as $\alpha, \beta, \gamma, \ldots$ will range over members of $\mathcal{M}$; $r, s, t, \ldots$ will range over both (possibly annotated) terms and meta-level terms. These variables may be sub or super-scripted. The empty set symbol, $\emptyset$, shall denote both the empty set and the substitution with empty domain.

3 Difference Matching

The Algorithm

The input to a difference matcher is a term s, possibly containing meta-variables, called the *pattern*, and a term t called the *instance*. The difference matcher returns annotated terms r, and substitutions σ such that r is an annotation of s and the skeleton of r matches t under substitution σ. As the output to a difference matcher is not always unique, we shall give an an algorithm for difference matching in a logic programming like language that computes terms satisfying the relation of type $\mathcal{T}_S(\Sigma, \mathcal{V} \cup \mathcal{M}) \times \mathcal{T}_S(\Sigma, \mathcal{V}) \times \mathcal{WT}_S(\Sigma, \mathcal{V}) \times Sub$. An implementation of this relation,

$$dm(s, t, r, \sigma)$$

should satisfy the property P_1,

$$Erase(r) = s,$$

the property P_2,

$$\sigma(Skel(r))) = t,$$

and the property P_3,

$$Dom(\sigma) = Vars(Skel(r)).$$

The first property insists that r is simply an annotation of s. The second insists that the skeleton of r equals t under substitution σ. The last property enforces a kind of minimality on substitutions and allows us to ignore those with "extraneous" assignments. As shorthand, we will let $\mathit{MatchRel}(s,t,r,\sigma)$ represent the conjunction of these three properties, $P_1 \wedge P_2 \wedge P_3$.

The following is a specification of a relation dm that has these properties.

```
% Difference Matching
% Clause 1, Meta-Variable:
$dm(\alpha, t, \alpha, \{\langle \alpha, t\rangle\}) \Leftarrow Ty(\alpha) = Ty(t).$

% Clause 2, Constant (or member of $\mathcal{V}$):
$dm(a, a, a, \emptyset).$

% Clause 3, equal outer functors ($j > 0$):
$dm(f(s_1, \ldots, s_j), f(t_1, \ldots, t_j), f(r_1, \ldots, r_j), \sigma) \Leftarrow$
  $\forall i \in \{1..j\}.\, dm(s_i, t_i, r_i, \sigma_i),$
  $\mathit{if}\ \forall i, i' \in \{1..j\}.\, \mathit{compatible}(\sigma_i, \sigma_{i'})\ \ \mathit{then}\ \sigma = \cup_{i=1}^{j} \sigma_i.$

% Clause 4, (possibly) different outer functors ($j > 0$)
$dm(f_s(s_1, \ldots, s_j), f_t(t_1, \ldots, t_k), \boxed{f_s(s_1, \ldots, s_{i-1}, \underline{r_i}, s_{i+1} \ldots s_j)}, \sigma) \Leftarrow$
  $\exists i \in \{1..j\}.\, dm(s_i, f_t(t_1, \ldots, t_k), r_i, \sigma).$
```

The program notation we use is similar to that of Prolog. We have used some syntactic sugar such as ellipsis and bounded universal and existential quantification which have the obvious "intended meaning" and can be coded in Prolog in a straightforward way. When translated into a Prolog program (wich we have done), the algorithm may be executed in in mode(+,+,?,?). We shall call the first two arguments, s and t, the *inputs* and those pairs of r and σ that satisfy dm the *outputs*.

As an example of difference matching, the following table contains the outputs which difference match with the pattern $x + 1 < (\alpha + 1) * (\alpha + 1)$ and the instance $x < (y + 1) * (y + 1)$.

Annotations	*Substitutions*
$\boxed{\underline{x}+1} < (\alpha+1)*(\alpha+1)$	$\{\langle \alpha, y\rangle\}$
$\boxed{\underline{x}+1} < \boxed{(\underline{\alpha}+1)} * \boxed{(\underline{\alpha}+1)}$	$\{\langle \alpha, y+1\rangle\}$
$\boxed{\underline{x}+1} < \boxed{(\underline{\alpha}+1)*(\alpha+1)}$	$\{\langle \alpha, (y+1)*(y+1)\rangle\}$
$\boxed{\underline{x}+1} < \boxed{(\alpha+1)*(\underline{\alpha}+1)}$	$\{\langle \alpha, (y+1)*(y+1)\rangle\}$

Properties

Several properties of dm are immediately apparent. First, the dm program can be decomposed into definitions of several logical relations. Clauses 2 and 3 define the relationship of syntactic identity. Adding Clause 1 defines the relationship of (sorted) first-order matching. That is, if inputs s and t first-order match, i.e. there is a substitution σ such that $\sigma(s) = t$, then there is a unique r and σ that satisfies the relation defined by the first three clauses. Adding Clause 4 instead of Clause 1, yields an algorithm for ground difference matching, illustrated in the introduction.

The union of the four clauses defines a relationship more general than first-order matching. For example, when the pattern s contains meta-variables of the same sort as s, a difference match is always possible (and one for each such meta-variable). For instance, if $Ty(f) = Ty(g) = S_1 \to S_1$ and $Ty(\alpha) = Ty(b) = S_1$ then

$$dm(f(\alpha), g(b), \boxed{f(\underline{\alpha})}, \{\langle \alpha, g(b)\rangle\}).$$

Also unlike standard matching, for a given input, there may be, in the worst case, exponentially many (in the size of the pattern) outputs that satisfy the difference matching relation. However, such bad behavior is unusual; in practice there are only a few successful matches. But because there are exponentially many possible ways of annotating a term, difference matching appears more difficult than first order matching. The complexity of the algorithm given, in mode(+,+,– ,–) is exponential because of the j-ary branching in Clause 4. Note that it is linear in mode(+,+,+,+), hence given a pair of inputs, finding a pair of outputs which satisfy the difference matching relation is in NP since we can check if a guessed annotation satisfies *MatchRel* in time linear in the size of the pattern. Of course, there may be faster implementations of difference matching than the one we have given; we have not analyzed the problem's exact complexity.

What is less obvious is that *dm* returns all and only all those matches that satisfy P_1, P_2, and P_3.

Lemma 1 *Soundness for dm:* $dm(s,t,r,\sigma) \Rightarrow \mathrm{MatchRel}(s,t,r,\sigma)$.

Proof: By structural induction on the pattern s. In the base case, s is either a meta-variable or a constant and *MatchRel* is obviously satisfied by the only applicable clauses, 1 and 2 respectively.

In the step case we have $s = f_s(s_1, \ldots, s_j)$. Suppose Clause 3 applies. Then $t = f_s(t_1, \ldots, t_j)$ and $r = f_s(r_1, \ldots r_j)$. Moreover for $i \in \{1..j\}$, $dm(s_i, t_i, r_i, \sigma_i)$ and by the induction hypothesis *MatchRel* holds of these smaller instances. So

$$Erase(r) = Erase(f_s(r_1, \ldots, r_j)) = f_s(Erase(r_1), \ldots, Erase(r_j)) = f_s(s_1, \ldots, s_j) = s.$$

Also, by the induction hypothesis $\sigma_i(Skel(r_i)) = t_i$, and, by Clause 3, $\sigma = \cup_i \sigma_i$. Hence

$$\sigma(Skel(r)) = \sigma(Skel(f_s(r_1, \ldots, r_j)) = f_s(\sigma_1(Skel(r_1)), \ldots \sigma_j(Skel(r_j))) = t.$$

Finally since the $Vars(Skel(r)) = \cup_i Vars(Skel(r_i))$, then $Dom(\sigma) = Vars(Skel(r))$.

Alternatively, suppose that Clause 4 applies, so $t = f_t(t_1, \ldots, t_k)$. Now there is some $i \in \{1..j\}$ and some r' where $dm(s_i, f_t(t_1, \ldots, t_k), r', \sigma)$. Now by the induction hypothesis, $Erase(r') = s_i$, so we have

$$\begin{aligned} Erase(r) &= Erase(\boxed{f_s(s_1, \ldots, s_{i-1}, \underline{r'}, s_{i+1} \ldots s_j)}) \\ &= Erase(\boxed{f_s(s_1, \ldots, \underline{s_i}, \ldots, s_j)}) \\ &= f_s(s_1, \ldots, s_i, \ldots, s_j) = s \end{aligned}$$

Similarly, P_2 holds because by the induction hypothesis $\sigma(\mathit{Skel}(r')) = f_t(t_1, \ldots, t_k)$, so

$$\sigma(\mathit{Skel}(r)) = \sigma(\mathit{Skel}(\boxed{f_s(s_1, \ldots, s_{i-1}, \underline{r'}, s_{i+1} \ldots, s_j)})) = \sigma(\mathit{Skel}(r')) = t.$$

Moreover, as $\mathit{Skel}(r') = \mathit{Skel}(r)$ and $\mathit{Dom}(\sigma) = \mathit{Vars}(\mathit{Skel}(r')))$, then $\mathit{Dom}(\sigma) = \mathit{Vars}(\mathit{Skel}(r))$. □

To prove the converse, we use a few facts which may be easily verified by the diligent reader.

Fact 1 *If* $s = f_s(s_1, \ldots, s_j)$, $t = f_t(t_1, \ldots, t_k)$, $r = f_r(r_1, \ldots, r_l)$, $\mathrm{MatchRel}(s, t, r, \sigma)$, *and* $\neg\mathrm{HdInWave}(r)$, *then* $f_s = f_t = f_r$, $j = k = l$, $\forall i \in \{1..j\}. \mathrm{MatchRel}(s_i, t_i, r_i, \sigma_i)$, $\sigma = \cup_{i=1}^{j} \sigma_i$ *and* $\mathrm{Dom}(\sigma_i) = \mathrm{Vars}(\mathrm{Skel}(r_i))$.

Fact 2 *If* $s = f_s(s_1, \ldots, s_j)$, $t = f_t(t_1, \ldots, t_k)$, $r = f_r(r_1, \ldots, r_l)$, $\mathrm{MatchRel}(s, t, r, \sigma)$, *and* $\mathrm{HdInWave}(r)$, *then* $f_s = f_r$, $j = l$, *and* $\exists i \in \{1..j\}. \mathrm{MatchRel}(s_i, t, r_i, \sigma) \wedge \forall k \in \{1..j\}. k \neq i \Rightarrow \mathrm{InWave}(r_k)$.

Lemma 2 *Completeness for dm:* $\mathrm{MatchRel}(s, t, r, \sigma) \Rightarrow dm(s, t, r, \sigma)$.

Proof: Proof by structural induction on s. There are two base-cases. In the first, s is a meta-variable α of the same sort as t. So by P_1, r must also be α and, by P_2 and P_3, σ must be the substitution $\{\langle\alpha, t\rangle\}$. But then we have Clause 1 of dm satisfied. In the second base-case, s is a constant a or the same sort as t. By P_1, $t = a$ and, by P_3, since $Var(t) = \emptyset$, we have $\sigma = \emptyset$. So Clause 2 of dm is satisfied.

In the step case we must have $s = f_s(s_1, \ldots, s_j)$, $t = f_t(t_1, \ldots, t_k)$ and $r = f_r(r_1, \ldots, r_l)$. But by P_1 we must have that the bodies of s and r are the same so f_r equals f_s and $j = l$. Now f_r may or may not be within a wave-front and we split on these two cases. In the first case, $\neg \mathit{HdInWave}(r)$, so we may apply Fact 1 and conclude that $\forall i \in \{1, ..j\}. \mathit{MatchRel}(s_i, t_i, r_i, \sigma_i)$. From the induction hypothesis it follows that $dm(s_i, t_i, r_i, \sigma_i)$ and since, by Fact 1, $\sigma = \cup\sigma_i$ we conclude $dm(s, t, r, \sigma)$ using Clause 3. In the second case, $\mathit{HdInWave}(r)$, we have by P_1 that the body of r and s are identical and, by Fact 2, $\exists i \in \{1..j\}. \forall k \in \{1..j\}. k \neq i \Rightarrow \mathit{InWave}(r_k)$. So r must look like

$$r = \boxed{f_s(s_1, \ldots, s_{i-1}, \underline{r_i}, s_{i+1} \ldots s_j)}.$$

Moreover, $\mathit{MatchRel}(s_i, t, r_i, \sigma)$ holds, so by the induction hypothesis $dm(s_i, t, r_i, \sigma)$. Hence $dm(s, t, r, \sigma)$ by Clause 4. □

4 Experience

We have explored the use of difference matching and rippling in several domains: equational reasoning arising in hardware verification, summing series, and calculating products, derivatives and integrals. This section will focus on examples in summing series.

Although an inductive proof can be used to show that a sum (or a product) has a certain closed form, we have been investigating *non-inductive* methods for *discovering* the sum. For example, one method for summing a series is to manipulate the sum into some function of known standard results. This is analogous to the situation in inductive theorem proving where we try to manipulate the induction conclusion into some function of the induction hypothesis. Consider:

$$s_n = \sum_{i=0}^{n} a * b^{i+1} \tag{3}$$

One slight complication is that to represent equations involving summation requires an extension of term syntax as summation is actually a functional, one of whose arguments is a function of the summation index. We shall not go into details here (see [**?**]), but representing patterns involving sums uses higher-order match variables in an essentially trivial way[5] and we can make direct use of our difference matcher to find substitution instances for such expressions.

In summing series, we will call upon various standard results like:

$$\sum_{I=0}^{N} C = (N+1) * C \tag{4}$$

$$\sum_{I=0}^{N} I = \frac{N(N-1)}{2} \tag{5}$$

$$\begin{array}{rcll} \displaystyle\sum_{I=0}^{N} C^I & = & \dfrac{C^{N+1}-1}{C-1} & C \neq 1 \\ & = & N+1 & C = 1 \end{array} \tag{6}$$

where C is a constant. Additionally, we will need various wave-rules for manipulating series and algebraic expressions. Some of these rules include:

$$\sum_{I=A}^{B} \boxed{(\underline{U}+V)} \Rightarrow \boxed{\underline{\sum_{I=A}^{B} U} + \sum_{I=A}^{B} V}$$

$$\sum_{I=A}^{B} \boxed{C * \underline{U}} \Rightarrow \boxed{C * \underline{\sum_{I=A}^{B} U}}$$

$$X^{\boxed{\underline{Y}+1}} \Rightarrow \boxed{X * \underline{X^Y}}$$

Again where C is a constant. We begin by difference matching our goal (3) against known standard results. In this case, we can successfully difference match and ripple against all three of the standard results given above. However, there only exist wave-rules for moving the wave-front annotations returned by difference matching against

[5]Even more trivial than Miller and Nipkow's higher-order patterns[15, 17] as each match variable within the body of the sum is a function precisely of the single binding variable representing the summation index.

(6):

$$\sum_{i=0}^{n} \boxed{a * \underline{b^{\boxed{\underline{i}+1}}}}$$

Of course, we are not always so lucky; sometimes, we can successfully difference match against several different standard results. This introduces an element of search (although in practice the search space is rather small).

Rippling can now be used to move these wave-fronts out of the way:

$$\begin{aligned}
s_n &= \sum_{i=0}^{n} \boxed{a * \underline{b^{\boxed{\underline{i}+1}}}} \\
&= \sum_{i=0}^{n} \boxed{a * b * \underline{b^i}} \\
&= \boxed{a * \underline{\sum_{i=0}^{n} \boxed{b * \underline{b^i}}}} \\
&= \boxed{a * b * \underline{\sum_{i=0}^{n} b^i}}
\end{aligned}$$

Finally, we fertilize with the closed form solution (6):

$$s_n = \begin{cases} a * b * \frac{b^{n+1}-1}{b-1} & \textit{if } b \neq 1 \\ a * b * (n+1) & \textit{if } b = 1 \end{cases}$$

Difference matching and rippling play important roles in several other methods we have investigated for summing series. For example, another method using difference matching and rippling *perturbates* the sum by one term. This is closely related to induction. Consider, for example, trying to sum the geometric progression of Equation 6 from first principles.

$$s_n = \sum_{i=0}^{n} b^i$$

We begin by perturbing this sum by one term:

$$s_{n+1} = s_n + b^{n+1}$$

Now, we can also strip off not the last term but the first term:

$$s_{n+1} = b^0 + \sum_{i=0}^{n} b^{i+1}$$

Combining the last two equations, we get:

$$s_n + b^{n+1} = b^0 + \sum_{i=0}^{n} b^{i+1}$$

This is nearly an equation in s_n. If we difference match the sum in the righthand side of the equation against s_n, we get the wave-fronts:

$$\sum_{i=0}^{n} b^{\boxed{\underline{i}+1}}$$

If we ripple these wave-fronts out of the way, we will have an equation just in s_n which we can algebraically solve:

$$\begin{aligned}
s_n + b^{n+1} &= b^0 + \sum_{i=0}^{n} b^{\boxed{\underline{i}+1}} \\
&= b^0 + \sum_{i=0}^{n} \boxed{b * \underline{b^i}} \\
&= b^0 + \boxed{b * \underline{\sum_{i=0}^{n} b^i}} \\
&= b^0 + \boxed{b * \underline{s_n}}
\end{aligned}$$

Thus,

$$s_n + b^{n+1} = b^0 + b * s_n$$

Solving this last equation (using the collect and isolate methods developed in the algebraic problem solver PRESS [6]) gives:

$$s_n = \frac{b^{n+1} - 1}{b - 1} \qquad b \neq 1$$

The case for $b = 1$ is trivial.

These and other methods based on difference matching for summing series have been implemented in the Oyster-Clam system [19]. They successfully sum a large number of different series. For example, they can tackle most of those problems give in the introductory chapters of a freshman text like *Concrete Mathematics*[12] (*e.g.* $\sum i^m$, $\sum(i+1) * a^i$, $\sum \sin(i * \theta)$). We can also solve all those sums reported by Hutter in [14]. Note that, unlike Hutter, we are discovering the closed form for the sums as opposed to verifying the answer using induction.

5 Conclusion and Future Work

This paper presents a difference matching algorithm and proves various properties it possesses (like soundness and completeness). Difference matching appears to be a promising means of marking differences between terms such that they can be made similar through rippling or some other means of selective simplification. It appears to be a key idea enabling the use of rippling as a controlled means of rewriting in a variety of inductive and non-inductive domains.

There are a number of directions for extensions that we have just begun to consider. Below we list several of the more promising directions.

Equational Matching: In the previous section we required that matching respect bound variables but other obvious extensions include full second and higher-order matching as well as first-order equational matching.

Unification: Matching can be extended by allowing match variables in both the pattern and instance. Another interesting "unification" extension is to return annotations of both the pattern and the instance. This would enable rippling to simplify both hypothesis and conclusion. As previously indicated, a difference unification algorithm can also serve as a wave-rule parser.

Annotated Substitutions: Difference matching could also be extended to return annotated substitutions. This will generate a much larger set of matches (when the inputs have at least one solution, they will have infinitely many) so there is a control problem of picking or efficiently enumerating useful matches. However, this extension is simple to implement and involves only an extension to the base-case of the difference matcher.

General Wave Annotations: Currently we allow no more than one sub-term to be deleted within a wave-front; however, returning "multi-wave annotations" (see [8]) requires this restriction to be weakened.

References

[1] 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.

[2] David A. Basin and Toby Walsh. β_0-difference matching. In Preparation.

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

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

[5] Robert S. Boyer and J. Strother Moore. *A Computational Logic Handbook*. Academic Press, 1988. Perspectives in Computing, Vol 23.

[6] Alan Bundy. *The Computer Modelling of Mathematical Reasoning*. Academic Press, 1983.

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

[8] Alan Bundy, Frank van Harmelen, Alan Smaill, and Andrew Ireland. Extensions to the rippling-out tactic for guiding inductive proofs. In M.E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 132–146. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449.

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

[10] 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.

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

[12] R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete Mathematics.* Addison-Wesley, 1989.

[13] Gérard Huet and D.C. Oppen. Equations and rewrite rules: a survey. In R. Book, editor, *Formal Languages: Perspectives and Open Problems.* Academic Press, 1980.

[14] 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.

[15] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. Technical Report ECS-LFCS-01-159, University of Edinburgh, LFCS, May 1991.

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

[17] Tobias Nipkow. Higher-order critical pairs. In *Symposium on Logic in Computer Science*, 1991.

[18] Gordon D. Plotkin. A note on inductive generalization. *Machine Intelligence,* 5:153-163, 1970.

[19] Toby Walsh, Alex Nunes, and Alan Bundy. The use of proof plans to sum series. In D. Kapur, editor, *11th International Conference on Automated Deduction.* Springer-Verlag, 1992.