Constrained completion: Theory, implementation, and results

Daniel Patrick Murphy

THESIS
T6409

CONSTRAINED COMPLETION:
THEORY, IMPLEMENTATION, AND RESULTS

by

DANIEL PATRICK MURPHY, 1966-

A DISSERTATION

Presented to the Faculty of the Graduate School of the

UNIVERSITY OF MISSOURI-ROLLA

In Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

1992

T6409
Copy 1
142 pages

[illegible]
Ralph Wilkerson, Advisor

C.Y. Ho
C.Y. Ho

Tom Sager
Thomas Sager

[illegible]
Richard Smith

[illegible]
George Zobrist

Leon M. Hall
Leon Hall

ABSTRACT

The Knuth-Bendix completion procedure produces complete sets of reductions but can not handle certain rewrite rules such as commutativity. In order to handle such theories, completion procedure were created to find complete sets of reductions modulo an equational theory. The major problem with this method is that it requires a specialized unification algorithm for the equational theory. Although this method works well when such an algorithm exists, these algorithms are not always available and thus alternative methods are needed to attack problems. A way of doing this is to use a completion procedure which finds complete sets of constrained reductions. This type of completion procedure neither requires specialized unification algorithms nor will it fail due to un-orientable identities.

We present a look at complete sets of reductions with constraints, developed by Gerald Peterson, and the implementation of such a completion procedure for use with HIPER - a fast completion system. The completion procedure code is given and shown correct along with the various support procedures which are needed by the constrained system. These support procedures include a procedure to find constraints using the lexicographic path ordering and a normal form procedure for constraints.

The procedure has been implemented for use under the fast HIPER system, developed by Jim Christian, and thus is quick. We apply this new system, HIPER-extension, to attack a variety of word problems. Implementation alternatives are discussed, developed, and compared with each other as well as with the HIPER system.

Finally, we look at the problem of finding a complete set of reductions for a ternary boolean algebra. Given are alternatives to attacking this problem and the already known solution along with its run in the HIPER-extension system.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Ralph Wilkerson for his guidance and patience with me throughout my years at UMR. You have taught me a lot both in and out of the classroom. I would also like to thank the rest of my committee: Dr. Leon Hall, Dr. C.Y. Ho, Dr. Thomas Sager, Dr. Richard Smith, and Dr. George Zobrist for their teaching and willingness to answer my questions.

Thank you to Dr. Gerald Peterson of the McDonnell-Douglas Corporation for answering my many questions about his constrained reductions. Also, thank you to Dr. DeKock and the computer science and math departments for giving me the chance to learn so much and to the Generic Mineral Technology Center for Pyrometallurgy, the Intelligent Systems Center, and Dr. Gary Leininger for supporting me while I did.

Thank you to Doug Meyer, Chris Merz, Steve Kern and many other graduate students here for your friendship. Thank you Gary Larson and Bill Watterson for the humor given to me during my stressful graduate years.

Thanks to my family and friends for all the good times. Finally thank you Mom and Dad for the constant love and support that I needed - without it I could not have accomplished half as much as I have.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

I. INTRODUCTION

A. STRUCTURE

Section II will present the definitions and notation which we will use for our discussion.

We will review previous work done in the field of unification theory and complete sets of reductions in Section III. We include the Knuth-Bendix completion procedure and the extensions of it modulo an equational theory as given by Peterson & Stickel and Jouannand & Kirchner.

Various orderings used to prove termination, both with or without an equational theory, are described in Section IV.

Section V gives additional extensions to the Knuth-Bendix procedure. This is given so that we might see when our constrained completion procedure might be more applicable.

In Section VI we describe an algorithm which produces the constraints we will need for our completion procedure. The ordering relative to which we produce these constraints is the lexicographic path ordering.

Constrained rewriting and the entire constrained completion procedure is given in Section VII. We present the algorithms to make constrained reductions from identities, prove joinability, and to inter-reduce constrained rewrite rules.

A very fast completion procedure called HIPER is described in Section VIII along with our modified version of this procedure, HIPER-extension, which is a constrained completion procedure. We discuss various implementation details and compare these

We present attempts at finding a complete set of reductions for a ternary boolean algebra in Section IX. In it we describe a ternary associative-commutative unification algorithm for use with HIPER to find the complete set and give a complete set of constrained reductions using HIPER-extension.

Section X gives conclusions from this paper and directions for future work.

Finally, Appendix A contains runs in the HIPER-extension system while Appendix B gives proofs mentioned but not presented in Section IX.

B. MOTIVATION

The problem of deciding whether two first order terms are equal with respect to an equational theory is in general an undecidable process. If however, a complete set of reductions exists for the equational theory then the problem is decidable. Completion procedures not only determine if a set of reductions is complete but also tries to add reductions in an attempt to make the resulting set complete. The process of proving completeness has been well-studied and many variations have been developed. The most important variation, in which completeness is proved modulo an equational theory, requires the existence of a special unification algorithm. This is restrictive since, in general, developing these algorithms is difficult. The variations, unfailing completion and conditional completion do not have this restriction but are somewhat limited in their applicability.

Constrained completion is more general and will, thus, work on a larger set of problems without the need for a specialized unification algorithm. This procedure is "unfailing" since we can always orient our equations. Thus the result is either successful or the procedure runs forever without finding a complete set of reductions. This paper presents this procedure, speedup considerations, and an implementation.

Complete sets of reductions have many applications including use in symbolic math systems such as MACSYMA, abstract data type specifications, compiler optimization, software validation, and automated theorem proving. For references and other uses see [De89] and [JK86].

II. DEFINITIONS AND NOTATION

A. TERMS

We will define *terms* in the usual fashion. We use an infinite set of variables $\mathcal{V}$ and a finite set of function symbols $\mathcal{F}$ such that $\mathcal{V} \cap \mathcal{F} = \emptyset$. Each function symbol has a fixed arity ≥ 0. The set of all terms over $\mathcal{F}$ and $\mathcal{V}$, denoted $\mathcal{T}(\mathcal{F},\mathcal{V})$, is defined below. We will simply use $\mathcal{T}$ when no ambiguity can arise.

(1) If $v \in \mathcal{V}$, then v is a term.

(2) If $c \in \mathcal{F}$ and arity(c)=0 (i.e. c is a constant), then c is a term.

(3) If $t_1, \ldots, t_n$ are terms, $f \in \mathcal{F}$ and arity(f) = n > 0, then $f(t_1, \ldots, t_n)$ is a term.

(4) $\mathcal{T}$ is the intersection of all sets satisfying 1, 2, and 3.

The domain of a term t, written *dom(t)*, is the set of all subterm labels of t. (e.g. if t = f(g(a),x,h(y)) then dom(t) = {(), 1, 1.1, 2, 3, 3.1} where () is the root term, 1 is the first subterm, g(a), 1.1 is the first subterm of the first subterm, a, etc.) The strict domain of a term t, written *sdom(t)*, is the set of all subterm labels of t which are not variables. For example, if $f,g,h,a \in \mathcal{F}$ and $x,y \in \mathcal{V}$ then sdom(f(g(a),x,h(y))) is {(), 1, 1.1, 3} which corresponds to {f(g(a),x,f(y)), g(a), a, h(y)}.

We write t/i to refer to a subterm i of a term t where $i \in$ dom(t). We write $t[i \leftarrow s]$ to denote the term t with its subterm i replaced by s. For example, if t = f(g(f(a,y)),z,g(g(z))), t/i = f(a,y), and s = g(b) then $t[i \leftarrow s]$ = f(g(g(b)),z,g(g(z))). Subterms t/i and t/j are *disjoint* if neither is a subterm of the other.

Var(t) is the set of all variables in a term t. A term t is *ground* if Var(t) = $\emptyset$. Two terms t_1 and t_2 are *variable disjoint* if $\mathrm{Var}(t_1) \cap \mathrm{Var}(t_2) = \emptyset$.

B. SUBSTITUTIONS AND EQUATIONS

A *substitution* σ is a set of ordered pairs (v/t) with $v \in \mathcal{V}$ and $t \in \mathcal{T}$ such that for all (v_i/t_i), $(v_j/t_j) \in \sigma$ if $i \neq j$ then $v_i \neq v_j$. Given a substitution σ and a term t we apply σ to t, written $t\sigma$, as follows:

(1) If t is a constant, then $t\sigma = t$.

(2) If t is a variable not appearing in the left-hand side of a pair in σ, then $t\sigma = t$.

(3) If t is a variable and there exists a t' such that $(t/t') \in \sigma$, then $t\sigma = t'$.

(4) If $t = f(t_1, \ldots, t_n)$, then $t\sigma = f(t_1\sigma, \ldots, t_n\sigma)$.

For example, if $\sigma = \{(x/f(a,z)), (y/g(b))\}$ and $t=h(x,x,y,z)$ then $t\sigma = h(f(a,z),f(a,z),g(b),z)$.

If $\sigma_1 = \{(x_1/t_1), \ldots, (x_n/t_n)\}$ and $\sigma_2 = \{(y_1/s_1), \ldots, (y_m/s_m)\}$ are substitutions, then the *composition* of the substitutions, written $\sigma_1\sigma_2$, is $\{(x_1/t_1\sigma_2), \ldots, (x_n/t_n\sigma_2)\} \cup \{(y/s) \mid (y/s) \in \sigma_2$ and for all $t \in \mathcal{T}, (y/t) \notin \sigma_1\}$. Composition is defined this way so that the result of applying $\sigma_1\sigma_2$ to t is the same as applying σ_2 to $t\sigma_1$. That is, $t(\sigma_1\sigma_2) = (t\sigma_1)\sigma_2$.

Substitutions σ_1 and σ_2 are equal if for all $t \in \mathcal{T}, t\sigma_1 = t\sigma_2$. If $\sigma_1 = \sigma_2\delta$, then σ_2 is an instance of σ_1. If no variables in the ordered pairs of σ_1 appear in σ_2 and vice versa (i.e. σ_1 and σ_2 are variable disjoint), then $\sigma_1\sigma_2 = \sigma_2\sigma_1$.

A substitution σ is a *matcher* for terms t_1 and t_2 if $t_1 = t_2\sigma$. A substitution θ is a two-way matcher or *unifier* of terms t_1 and t_2 if $t_1\theta = t_2\theta$. A unifier is *most general* if for all other unifiers δ there exists a substitution σ such that $\delta=\sigma\theta$. Thus, all other unifiers can be produced by further instantiating variables in θ.

A *unification algorithm* is an algorithm which produces a most general unifier for given terms or replies that no such unifier exists.

An *equational theory* is a set of equations. We write $t_1 =^1_{\mathcal{E}} t_2$ if and only if $t_1/u = \lambda\sigma$ for some equation $\lambda == \rho$ in the equational theory E and $t_2 = t_1[u \leftarrow \rho\sigma]$. We let $=_{\mathcal{E}}$ be the reflexive-transitive closure of $=^1_{\mathcal{E}}$. If $t_1 =_{\mathcal{E}} t_2$, then t_1 and t_2 are said to be *E-equal.* A *congruence class* $[t]_{\mathcal{E}}$ is the set of all terms which are E-equal to t.

A substitution σ is an *E-matcher* of t_1 and t_2 if $t_1 =_{\mathcal{E}} t_2\sigma$. A substitution σ is an *E-unifier* of t_1 and t_2 if $t_1\sigma =_{\mathcal{E}} t_2\sigma$. For substitutions σ_1 and σ_2, $\sigma_1 =_{\mathcal{E}} \sigma_2$ if and only if for all $v \in \mathcal{V}$, $\sigma_1(v) =_{\mathcal{E}} \sigma_2(v)$.

E-unification algorithms produce sets of E-unifiers. A set of E-unifiers Γ for terms s and t is *complete* if

(1) If $\sigma \in \Gamma$, then $s\sigma =_{\mathcal{E}} t\sigma$. (correctness)

(2) If $s\theta =_{\mathcal{E}} t\theta$, then there exists a $\sigma \in \Gamma$ and a substitution δ such that $\theta = \sigma\delta$. (completeness)

A complete set of unifiers Γ is *minimal* if and only if

(3) For all pairs $\sigma_1, \sigma_2 \in \Gamma$, if there exists a substitution δ such that $\sigma_2 =_E \sigma_1\delta$, then $\sigma_2 = \sigma_1$. (minimality).

That is, no two E-unifiers in Γ are instances of the other.

C. REDUCTIONS

A *reduction* or *rewrite rule* is an ordered pair of terms $\lambda \rightarrow \rho$ (or $\rho \leftarrow \lambda$) such that $\lambda = \rho$ is an identity and ρ is in some sense simpler than λ. When we make a rewrite rule from an equation we are *orienting* the equation from complex to simple. We apply a reduction $\lambda \rightarrow \rho$ to a term t if $t/i = \lambda\sigma$ giving the term $t' = t[i \leftarrow \rho\sigma]$. We write this as $t \rightarrow t'$ and say that t rewrites or reduces to t′. We write $t \rightarrow_{\mathcal{R}} t'$ if t rewrites to t′ by application of some reduction in a set of reductions $\mathcal{R}$. The relations $\rightarrow^{+}_{\mathcal{R}}$ and $\rightarrow^{*}_{\mathcal{R}}$ are, respectively, the transitive and reflexive-transitive closure of the rewriting relation $\rightarrow_{\mathcal{R}}$.

A term t is *irreducible* by $\mathcal{R}$ if no reduction in $\mathcal{R}$ can reduce t. If $t \rightarrow^{*}_{\mathcal{R}} t'$ and t' is irreducible by $\mathcal{R}$ then t' is the *$\mathcal{R}$-normal* form of t, written $t\downarrow_{\mathcal{R}}$. '$\mathcal{R}$' will be dropped from the notation when the context is clear.

A reduction $\lambda \rightarrow \rho \in \mathcal{R}$ *E-rewrites* s to t, written $s \rightarrow_{\mathcal{R},\mathcal{E}} t$, if $s/i =_{\mathcal{E}} \lambda\sigma$ for some substitution σ and $t = s[i \leftarrow \rho\sigma]$. The relations $\rightarrow^{+}_{\mathcal{R},\mathcal{E}}$ and $\rightarrow^{*}_{\mathcal{R},\mathcal{E}}$ are the transitive and reflexive-transitive closure of $\rightarrow_{\mathcal{R},\mathcal{E}}$, respectively. A term t is *E-irreducible* if no reduction in $\mathcal{R}$ can E-reduce it.

For a given term t a *rewrite tree* is the set of terms which t can rewrite to through any number of reductions. Its name comes form the familiar tree-like structure which can represent it with each branch representing a rewrite. See Figure 1.

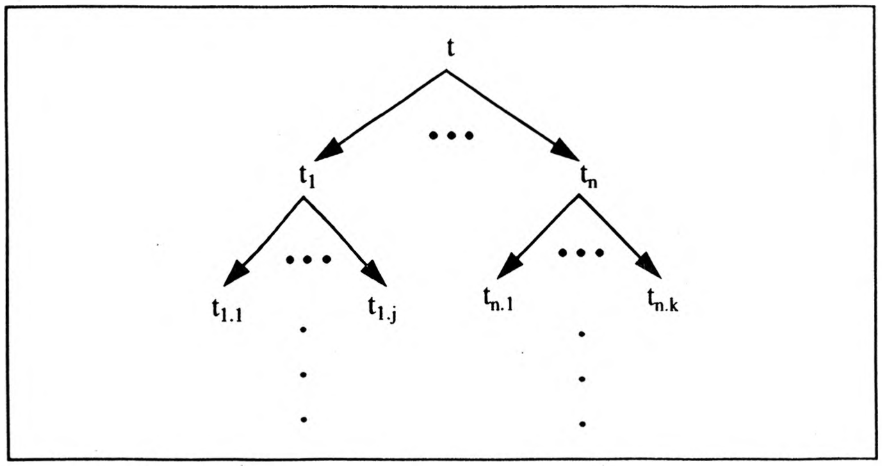

Figure 1. Rewrite tree.

III. LITERATURE REVIEW

A. UNIFICATION

Unification has been around since Post in the 1920s and Herbrand in the 1930s. J.A. Robinson gave the first practical algorithm to generate a most general unifier in his 1965 paper [Ro65]. Remembering from the previous section that unification, simply put, is the replacement of variables by terms to make terms equal we can look at the unification

```
**R-Unify**($t_1$,$t_2$)
$\sigma = \varnothing$
if $t_1 = t_2$ then return ($\sigma$)
if variable($t_1$) then
  if occurs($t_1$,$t_2$) then
    return (FALSE)
  else
    return ($\{t_1/t_2\}$)

if variable($t_2$) then
  if occurs($t_2$,$t_1$) then
    return (FALSE)
  else
    return ($\{t_2/t_1\}$)

if top-level-symbol($t_1$)$\neq$top-level-symbol($t_2$) then return (FALSE)
For i = 1 to arity (top-level-symbol($t_1$))
  $\delta$ = R-Unify (subterm($t_1$,i), subterm($t_2$,i))
  if $\delta$ = FALSE then return (FALSE)
  $\sigma = \delta\sigma$
  $t_1 = t_1\sigma$
  $t_2 = t_2\sigma$
end for
return($\sigma$)
```

algorithm given in Figure 2. An almost linear time and space algorithm was given in [MM76] and the first linear time and space algorithm was given in [PW78].

Robinson (R-)unification has since been modified to find E-unifiers for a given equational theory. Equational unification was first done by Plotkin [Pl73], where he developed an associative unification algorithm. Siekmann created one for commutativity [Si79] and this was extended to n-ary complete commutativity in [Mu92]. Associative-commutative (AC) unification has been described in [St81] and [CL88]. These are some important unification algorithms but many others exist.

E-unification problems fall into the following four classes:

(1) unitary - a single mgu is present (if it exists).

(2) finitary - a finite, minimal, complete set of E-unifiers exists.

(3) infinitary - an infinite, minimal, complete set of E-unifiers exists.

(4) nullary - no minimal, complete set of E-unifiers exists.

For example, when $E = \varnothing$ unification is unitary, when E = commutativity unification is finitary, and when E = associativity unification is infinitary.

The combination of unification algorithms does not always work as expected. For example, AC-unification - a combination of an infinitary (A) to a finitary (C) theory - is finitary; AI-unification - a combination of an infinitary (A) to a finitary (I) theory - is nullary[Sc86]. See Table 3.1 for more theories. For further information on the cardinality of E-unification problems see [JK90] or [Si89].

Table I. Classification of some unification problems.

Theory	Type	Theory	Type
∅	unitary	CI	finitary
A	infinitary	ACI	finitary
C	finitary	Dl	unitary
I	finitary	Dr	unitary
AC	finitary	D	infinitary
AI	nullary		

A=Associativity; C=Commutativity; I=Idempotenecy; Dr=right distributivity; Dl= left distributivity; D=Dr+Dl

We can also divide up the equational theories into different classes. For example, an equational theory is *regular* if equal terms have the same variables. A theory is *collapse-free* if non-variable terms are not equal to a variable (e.g. if $E = \{f(x) = x\}$ then it is not collapse-free). A theory is *permutative* if all equal terms have the same symbols. Other classes exist. For an excellent look at how these classes relate to the E-unification problem see [BHS89].

Yelick [Ye85] presented a method for combining different E-unification algorithms but is restricted to collapse-free, regular theories. Claude Kirchner [Ki87] developed tools for automatically generating some unification algorithms which is modified and improved in [Ch89].

Although R-unification has been shown to have a linear time solution, most E-unification algorithms are NP-hard (see [KN86]). For two excellent survey articles on unification theory and its applications see [Kn89] and [Si89].

B. WORD PROBLEMS AND COMPLETE SETS OF REDUCTIONS

The *word problem* is the problem of deciding whether two terms are equal under a given equational theory. It is well known that the word problem is in general undecidable. However, if we have a complete set of reductions for the equational theory then the corresponding word problem is a finite, decidable process. A set of reductions is a *complete set of reductions* if every term has one and only one irreducible form and terms are equal under the equational theory if and only if their irreducible forms are identical.

In 1970, Knuth and Bendix [KB70] presented a process which took an equational theory as input, transformed the equations into reductions, and determined whether the set of reductions was complete. Also, if the given set of reductions was not complete new reductions were added in an attempt to complete the set. This type of procedure is called a *completion procedure*. Three possible outcomes are possible with such a procedure: success while returning the complete set of reductions; failure due to a non-orientable equation; or the procedure continues forever, neither completing the set nor finding a non-orientable equation.

Theorem 3.1 (KB70) A set of reductions is complete if the following two properties hold:

(1) The finite termination property.

(2) The Church-Rosser property.

A set of reductions has the *finite termination property* if there exists no infinite chain of rewrites $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \ldots$ This implies that every term has an irreducible form.

To show that this property holds an ordering over all terms is established. Knuth and Bendix developed a weighting function to establish a partial ordering on the terms.

Taking from [KB70], let w_i be the weight of function symbol i. All weights must be greater than 0 with the possible exception of a single unary operator which may have a weight of 0. Let n(x,t) be the number of occurrences of the symbol x in the term t. The weight of a ground term t, $W(t) = \sum_{j \geq 1} w_j n(f_j, t)$. Let w_0 be the minimum weight of a ground term. Letting v_i be a variable and f_i be a function symbol, the weight for an arbitrary term t, $W(t) = w_0 \sum_{j \geq 1} n(v_j, t) + \sum_{j \geq 1} w_j n(f_j, t)$.

Two terms are ordered, $s >_{KB} t$, by the *Knuth-Bendix ordering* if and only if one of the following holds

(1) $W(s) > W(t)$ and $n(v_i, s) \geq n(v_i, t)$ for all i; or

(2) $W(s) = W(t)$ and $n(v_i,s) = n(v_i,t)$ for all i

and either t is a variable and $s = f(\ldots t \ldots)$, or

$s = f_i(s_1, \ldots, s_n)$, $t = f_j(t_1, \ldots, t_m)$, and either

(2a) $i > j$; or

(2b) $i = j$ and $s_1 = t_1, \ldots, s_{k-1} = t_{k-1}, s_k > t_k$ for some k.

Thus terms can either be identical, $s >_{KB} t$, $t >_{KB} s$, or s and t are *incomparable.* If the last case happens we write s # t. As we shall see, when we are given incomparable terms the Knuth-Bendix procedure stops with failure since we cannot orient them into a reduction.

Knuth and Bendix proved that the above ordering has the finite termination property but it is not the only ordering which has this property. Others are presented in the section on termination, but all must have the following properties:

(1) There exists no infinite series of terms such that $t_1 > t_2 > t_3 > \ldots$.

(2) If $s > t$, then $s\sigma > t\sigma$ for any substitution σ (i.e. substitution preserves the ordering).

(3) If $s > t$, then $f(\ldots s \ldots) > f(\ldots t \ldots)$ - this assures that if a subterm is reduced then the entire term is reduced.

Relations which exhibit property 1 are *well-founded*, relations which exhibit property 2 are *subterm preserving*, and relations which exhibit property 3 are *monotonic*. To show why we want these properties, recall that when a reduction $\lambda \rightarrow \rho$ is applied to a term t at position i with substitution σ then the resulting term $t' = t[i \leftarrow \rho\sigma]$. Thus by property 2 we know that $\lambda\sigma > \rho\sigma$ and by monotonicity we thus have $t > t'$. Now since all reductions $(\lambda \rightarrow \rho) \in \mathcal{R}$ are ordered such that $\lambda > \rho$ we know that any application of a rewrite rule will result in a simpler term. Thus if well-foundedness holds for the relation we see that the finite termination property holds.

It is interesting to note that the finite termination property is enough to solve the related word problem. You need only search the rewrite trees of the two terms for a common (irreducible) term - a guaranteed finite process. This is likely to be a very expensive task, though, so we would like to have our set of reductions be Church-Rosser also.

A set of reductions is *Church-Rosser* if all terms which are equivalent with respect to the reductions have a common rewriting. In [Ne42], Newman, building on [CR36], showed that a set of reductions is Church-Rosser if and only if it is confluent.

A set of reductions is *confluent* if and only if for all terms if $t_1 \rightarrow^* t_2$ and $t_1 \rightarrow^* t_3$ then there exists a term t_4 such that $t_2 \rightarrow^* t_4$ and $t_3 \rightarrow^* t_4$. See Figure 3.

$$\begin{tikzcd}
& t_1 \arrow[dl, "*"'] \arrow[dr, "*"] & \\
t_2 \arrow[dr, "*"'] & & t_3 \arrow[dl, "*"] \\
& t_4 &
\end{tikzcd}$$

Figure 3. Confluence.

This means that to show confluence we must pair-wise check that all terms in a term's rewrite tree have a common rewriting. If the branching factor and depth of the rewrite tree are great this can easily become a very time consuming task. Luckily, Newman showed that proving local confluence is enough to show that the Church-Rosser property holds.

A set of reductions is *locally confluent* if and only if for all terms t_1, if $t_1 \rightarrow t_2$ and $t_1 \rightarrow t_3$ then there exists a term t_4 such that $t_2 \rightarrow^* t_4$ and $t_3 \rightarrow^* t_4$. See Figure 4.

$$\begin{tikzcd}
& t_1 \arrow[dl] \arrow[dr] & \\
t_2 \arrow[dr, "*"'] & & t_3 \arrow[dl, "*"] \\
& t_4 &
\end{tikzcd}$$

Figure 4. Local confluence.

Theorem 3.2 (KB70) The following statements are true about finitely terminating sets of reductions.

(1) The set is complete.

(2) The set has the Church-Rosser property.

(3) The set is confluent.

(4) The set is locally confluent.

The only difference between confluence and local confluence is that local confluence pair-wise checks for a common rewriting only between terms which are rewritten once from t while confluence pair-wise checks all rewritten terms. This will, in most cases, greatly reduce the number of checks which must be performed and might make the process computationally possible except that we must prove that local confluence holds for the infinite set of all terms. With our next discussion we will remove this severe problem.

Let us look at how we prove local confluence for any term t. Let $r_1 = (\lambda_1 \rightarrow \rho_1)$ and $r_2 = (\lambda_2 \rightarrow \rho_2)$ be two reductions which can be applied to t at positions i and j, respectively. If i and j are disjoint subterms then confluence is trivially shown since $t \rightarrow_{r_1} t[i \leftarrow \rho_1\sigma_1] \rightarrow_{r_2} t[i \leftarrow \rho_1\sigma_1; j \leftarrow \rho_2\sigma_2] {}_{r_1}\!\leftarrow t[j \leftarrow \rho_2\sigma_2] {}_{r_2}\!\leftarrow t$.

If, however, i and j are not disjoint let us assume, without loss of generality that j is a subterm of i and that r_1 and r_2 are variable disjoint. Thus for some position k of i we have $t/i = \lambda_1\sigma_1 = t/j = t/i.k = \lambda_2\sigma_2$. It can be shown that there exists a position k' such that $(\lambda_1\sigma_1)/k = (\lambda_1/k')\sigma_1$. Now since r_1 and r_2 are variable disjoint $(\lambda_1/k')\sigma_1 = (\lambda_1/k')\sigma_2\sigma_1$ and $\lambda_2\sigma_2 = \lambda_2\sigma_2\sigma_1$. Thus $(\lambda_1/k')\sigma_2\sigma_1 = \lambda_2\sigma_2\sigma_1$ which makes $\sigma_2\sigma_1$ a unifier for λ_1/k' and λ_2. Let θ be the most general unifier for the two terms. Thus the terms $t_1 = t[i \leftarrow \rho_1\sigma_1]$ and $t_2 = t[i \leftarrow (\lambda_1[k' \leftarrow \rho_2\sigma_2])\sigma_1]$ can be written as:

$$t_1 = t[i \leftarrow \rho_1\theta]$$

$$t_2 = t[i \leftarrow (\lambda_1[k' \leftarrow \rho_2)\theta].$$

We can see now that the only position which the two rewritten terms differ at is i. Thus the terms t_1 and t_2 conflate (reduce to a common term) if and only if t_1/i and t_2/i conflate. We prove conflation by putting the terms into their normal forms and seeing if they are identical.

Thus, a term t is locally confluent if and only if for all pairs of reductions $r_1 = \lambda_1 \rightarrow \rho_1$ and $r_2 = \lambda_2 \rightarrow \rho_2$

(1) the resulting terms trivially have a common rewriting, or

(2) the pair $<\rho_1\theta, (\lambda_1[k' \leftarrow \rho_2])\theta>$ conflates.

These pairs are called *critical pairs* and the process of forming and conflating critical pairs is called the *superposition process*.

Note that neither term in the critical pair depend on the initial term t. Instead they are dependent only on the two reductions. Thus if we can show local confluence for one term we show it for all terms and our process in computationally possible since we now need only consider the left-hand sides of reductions for superposition.

The superposition process is the heart of the Knuth-Bendix completion procedure. We need only generate these critical pairs and show conflation. If the terms do not conflate then we try to add a new rewrite rule if the normal forms of the two terms can be oriented and see if the resulting set of reductions is complete. If they cannot be oriented we stop with failure.

A complete proof of correctness of the Knuth-Bendix algorithm is given in [Hu81].

Forgaard and Guttag [FG84] made the Knuth-Bendix completion procedure more failure resistant through a simple improvement. When a pair cannot be oriented it is 'shelved' in case a new rewrite rule is later found which can then allow the shelved pair to be oriented.

Knuth and Bendix provided many examples of their completion procedure in use. For example, given the following rewrite rules from group theory

(1) $e \bullet x \rightarrow x$

(2) $x^{-} \bullet x \rightarrow e$

(3) $(x \bullet y) \bullet z \rightarrow x \bullet (y \bullet z)$

the following reductions were added to make the set complete

(4) $x^{-} \bullet (x \bullet y) \rightarrow y$

(5) $x \bullet e \rightarrow x$

(6) $e^{-} \rightarrow e$

(7) $x^{--} \rightarrow x$

(8) $x \bullet x^{-} \rightarrow e$

(9) $x \bullet (x^{-} \bullet y) \rightarrow y$

(10) $(x \bullet y)^{-} \rightarrow y^{-} \bullet x^{-}$.

The Knuth Bendix completion procedure can be seen in Figure 5. The function *inter-reduce* puts both sides of reductions into normal form. If this makes a reduction an identity we can delete it. Else we re-orient it if necessary. If this is not possible we abort the entire completion procedure with failure. See Figure 6.

C. COMPLETE SETS OF REDUCTIONS MODULO EQUATIONAL THEORIES

1. Complete Sets of Reductions Modulo Associativity and Commutativity

Although the Knuth-Bendix procedure is a generic procedure the authors realized that it could not easily handle certain identities. Commutativity cannot be oriented with their ordering and associativity is not handled in a completely generic way [PS81]. To overcome these problems Lankford and Ballantyne developed algorithms to handle commutative axioms [LB77a], permutative axioms [LB77b], and associative-commutative axioms [LB77c]. Their method had the drawback that it was a semi-decision procedure since it was not guaranteed to terminate after finding a complete set of reductions.

Huet developed a decision procedure using methods like Lankford and Ballantyne's

```
**KB-Completion** ($\mathcal{E}$)
/* $\mathcal{E}$ is a set of equations of the form $\lambda == \rho$ */
 $\mathcal{R}$= set of reductions formed from $\mathcal{E}$
 repeat
   Complete=TRUE
   for all reductions $\lambda_1 \rightarrow \rho_1 \in \mathcal{R}$
     for all reductions $\lambda_2 \rightarrow \rho_2 \in \mathcal{R}$
       for all subterms $\lambda_1/i$ which are not variables
         $\theta$ = mgu of $\lambda_1/i$ and $\lambda_2$
         if $\theta$ exists
           $t_1 = \rho_1\theta\downarrow_{\mathcal{R}}$
           $t_2 = \lambda_1[i \leftarrow \rho_2]\theta\downarrow_{\mathcal{R}}$
           if $t_1 = t_2$
             $t_1$ and $t_2$ conflate; test next critical pair
           else if $t_1 > t_2$
             $\mathcal{R} = \mathcal{R} \cup (t_1 \rightarrow t_2)$
             inter-reduce $\mathcal{R}$
             Complete = REPEAT
             goto until statement
           else if $t_2 > t_1$
             $\mathcal{R} = \mathcal{R} \cup (t_2 \rightarrow t_1)$
             inter-reduce $\mathcal{R}$
             Complete = REPEAT
             goto until statement
           else /* $t_1$ and $t_2$ are incomparable */
             Complete = FALSE
             goto until statement
           end if
         end if
       end for
     end for
   end for
until (Complete = TRUE or Complete = FALSE)
return(Complete, $\mathcal{R}$)
```

Figure 5. Knuth-Bendix completion procedure.

```
**Inter-reduce** ($\mathcal{R}$)
    for all $r_1$ ($\equiv \lambda_1 \rightarrow \rho_1$) $\in \mathcal{R}$
        $t_1 = \lambda_1 \downarrow_{\mathcal{R}-\{r_1\}}$
        $t_2 = \rho_1 \downarrow_{\mathcal{R}\{r_1\}}$
        if $t_1 = t_2$
            /* do not add a reduction to $\mathcal{R}_{new}$*/
            goto for statement
        else if $t_1 < t_2$
            $\mathcal{R}_{new} = \mathcal{R}_{new} \cup \{t_2 \rightarrow t_1\}$
        else if $t_2 < t_1$
            $\mathcal{R}_{new}=\mathcal{R}_{new} \cup \{t_1 \rightarrow t_2\}$
        else /* $t_1$ # $t_2$ */
            Complete=FAILURE
            exit Inter-reduce
        end if
    end for
    $\mathcal{R}= \mathcal{R}_{new}$
end Inter-reduce
```

Figure 6. Inter-reduce algorithm.

but required that the reductions be left linear (the left-hand side of the reduction must contain no variable more than once) [Hu80].

In 1981, Peterson and Stickel developed a Knuth-Bendix type algorithm which separates associative and commutative identities from the others and then attempts to find a complete set of reductions modulo the associative and commutative laws [PS81].

Actually, their formal treatment only assumes that a finite, complete unification algorithm exists for the given equational theory (which does exist for associativity and commutativity [St81], [CL88]). A set of reductions is then attempted to be made complete using the unification algorithm. A set of reductions $\mathcal{R}$ is *E-complete*, where $\mathcal{E}$ is an equational theory for which a finite, complete unification algorithm exists, if for all terms s and t if $s =_{\mathcal{R}\cup\mathcal{E}} t$, $s \rightarrow^*_{\mathcal{R},\mathcal{E}} s'$, and $t \rightarrow^*_{\mathcal{R},\mathcal{E}} t'$, where s' and t' are irreducible, then $s' =_{\mathcal{E}} t'$. See Figure 7.

$$\begin{tikzcd}
s \arrow[r, phantom, "=_{\mathcal{R}\cup\mathcal{E}}"] \arrow[d, "\mathcal{R},\mathcal{E}", "*"'] & t \arrow[d, "\mathcal{R},\mathcal{E}"', "*"] \\
s' \arrow[r, phantom, "=_{\mathcal{E}}"] & t'
\end{tikzcd}$$

Figure 7. E-completeness.

To prove completeness Peterson and Stickel required that the set of reductions be E-compatible. A set of reductions $\mathcal{R}$ is *E-compatible* if whenever $t \rightarrow_{\mathcal{R},\mathcal{E}} s$, there exists a node m, substitution σ, and a reduction $\lambda \rightarrow \rho \in R$ such that $t/i =_{\mathcal{E}} \lambda\sigma$, $s =_{\mathcal{E}} s'$, $s' \rightarrow^*_{\mathcal{R}} t'$, and $t' =_{\mathcal{E}} t[i \leftarrow \rho\sigma]$ for some terms s' and t'. See Figure 8.

$$\begin{tikzcd}
t \rightarrow_{\mathcal{R},\mathcal{E}} s & & \Rightarrow \\
t/i & =_{\mathcal{E}} \lambda\sigma & \text{and} \\
s & =_{\mathcal{E}} & s' \arrow[d, "\mathcal{R}"', "*"] \\
t[i \leftarrow \rho\sigma] & =_{\mathcal{E}} & t'
\end{tikzcd}$$

Figure 8. E-compatibility.

Critical pairs for R and E are defined similar to that in [KB70]. Let $\lambda_1 \rightarrow \rho_1$ and $\lambda_2 \rightarrow \rho_2$ be reductions in $\mathcal{R}$. If some subterm k of λ_1 E-unifies with λ_2 using substitution σ, then $<\lambda_1[k \leftarrow \rho_2]\sigma, \rho_1\sigma>$ is a critical pair.

A set of reductions is *E-terminating* if there exists no infinite sequence of E-rewrites. This leads us to the theorem used to prove completeness.

Theorem 3.3 (PS81) Let $\mathcal{R}$ be an E-compatible set of reductions. If $\mathcal{R}$ is E-terminating, then $\mathcal{R}$ is E-complete if and only if for every critical pair <s,t> there exists terms s′ and t′ such that $s \rightarrow_{\mathcal{R}} s'$, $t \rightarrow_{\mathcal{R}} t'$, and $s' =_{\mathcal{E}} t'$.

Peterson and Stickel went on to show necessary conditions for E-compatibility and more importantly to show that when $\mathcal{E}$ is an associative-commutative theory then E-compatibility holds for all sets of reductions $\mathcal{R}$ when certain extensions are added to $\mathcal{R}$.

Their algorithm for proving completeness is given in the paper and varies somewhat from the description given in the above theorem (e.g. only overlapping critical pairs need be checked). With this algorithm they produced complete sets of reductions for free commutative groups, free commutative rings with a unit element, distributive lattices, etc. Attempts were also made using incomplete associative unification algorithms with some success.

While success was gained by Peterson and Stickel, their method was not the most general treatment of the area. Indeed, their title said it was for "some equational theories". In particular to show E-compatibility the axioms in $\mathcal{E}$ had to have identical unique variables (i.e. permutative axioms with no repeating variables on a given side). Thus axioms such as $x \bullet x = x$, $x + 0 = x$, and $h(x,x,y) = h(y,x,x)$ could not be in $\mathcal{E}$. Jouannand and Kirchner removed these restrictions in their paper [JK86].

2. Complete Sets of Reductions Modulo an Equational Theory

Jouannand and Kirchner developed a general method for finding a complete set of reductions modulo an equational theory. Like with Peterson and Stickel's method the identities are split up into two sets. The first set $\mathcal{R}$ is a set of reductions and the second set $\mathcal{E}$ is an equational theory for which a finite, complete unification algorithm exists.

Let us now build up their results starting with some definitions. A *complete set of critical pairs* is the set of all critical pairs using the complete set of E-unifiers. Note that these pairs are formed from $\mathcal{R} \cup \mathcal{E}$.

$\mathcal{R}$ is *$\mathcal{E}$-confluent* if and only if for all terms t, t_1, and t_2 if $t \rightarrow^*_{\mathcal{R},\mathcal{E}} t_1$ and $t \rightarrow^*_{\mathcal{R},\mathcal{E}} t_2$ then there exist terms s_1 and s_2 such that $t_1 \rightarrow^*_{\mathcal{R},\mathcal{E}} s_1$, $t_2 \rightarrow^*_{\mathcal{R},\mathcal{E}} s_2$, and $s_1 =_{\mathcal{E}} s_2$. See Figure 9.

$$\begin{tikzcd}
& t \arrow[dl, "*"', "\mathcal{R},\mathcal{E}"] \arrow[dr, "*", "\mathcal{R},\mathcal{E}"'] & \\
t_1 \arrow[d, "*"', "\mathcal{R},\mathcal{E}"] & & t_2 \arrow[d, "\mathcal{R},\mathcal{E}"', "*"] \\
s_1 & =_{\mathcal{E}} & s_2
\end{tikzcd}$$

Figure 9. E-Confluence.

$\mathcal{R}$ is *locally $\mathcal{E}$-confluent* if and only if for all terms t, t_1, and t_2 if $t \rightarrow_{\mathcal{R},\mathcal{E}} t_1$ and $t \rightarrow_{\mathcal{R},\mathcal{E}} t_2$ then there exist terms s_1 and s_2 such that $t_1 \rightarrow^*_{\mathcal{R},\mathcal{E}} s_1$, $t_2 \rightarrow^*_{\mathcal{R},\mathcal{E}} s_2$, and $s_1 =_{\mathcal{E}} s_2$. See Figure 10.

$$\begin{tikzcd}
& t \arrow[dl, "\mathcal{R},\mathcal{E}"] \arrow[dr, "\mathcal{R},\mathcal{E}"'] & \\
t_1 \arrow[d, "*"', "\mathcal{R},\mathcal{E}"] & & t_2 \arrow[d, "\mathcal{R},\mathcal{E}"', "*"] \\
s_1 & =_{\mathcal{E}} & s_2
\end{tikzcd}$$

Figure 10. Local E-confluence.

Note the similarity between local E-confluence and Theorem 3.3. A comparison of the two will be discussed later.

Let $\mathcal{T}$ be the equational theory associated with $\mathcal{R}$. $\mathcal{R}$ is *$\mathcal{E}$-Church-Rosser* if and only if for all terms t_1 and t_2, if $t_1 =_{\mathcal{T}\cup\mathcal{E}} t_2$ then there exist terms s_1 and s_2 such that $t_1 \rightarrow^*_{\mathcal{R},\mathcal{E}} s_1$, $t_2 \rightarrow^*_{\mathcal{R},\mathcal{E}} s_2$, and $s_1 =_{\mathcal{E}} s_2$. See Figure 11.

$$\begin{tikzcd}
t_1 \arrow[r, phantom, "=_{\mathcal{T}\cup\mathcal{E}}"] \arrow[d, "*"', "\mathcal{R},\mathcal{E}"] & t_2 \arrow[d, "\mathcal{R},\mathcal{E}"', "*"] \\
s_1 \arrow[r, phantom, "=_{\mathcal{E}}"] & s_2
\end{tikzcd}$$

Figure 11. E-Church-Rosser.

From the previous discussion we might expect that E-termination and (local) E-confluence to imply E-Church-Rosser but in fact another property is needed to show this. This property is (local) coherence.

The reason why E-confluence is insufficient is that it only checks critical pairs formed between two reductions in $\mathcal{R}$. What we need to add are critical pairs between reductions in $\mathcal{R}$ and equations in $\mathcal{E}$ - which coherence will do for us.

$\mathcal{R}$ is *coherent modulo* $\mathcal{E}$ if and only if for all terms t, t_1, and t_2 if $t \to^{+}_{\mathcal{R},\mathcal{E}} t_1$ and $t =_{\mathcal{E}} t_2$ then there exists terms s_1 and s_2 such that $t_1 \to^{*}_{\mathcal{R},\mathcal{E}} s_1$, $t_2 \to^{+}_{\mathcal{R},\mathcal{E}} s_2$, and $s_1 =_{\mathcal{E}} s_2$. See Figure 12.

$$\begin{tikzcd}
t \arrow[r, phantom, "=_{\mathcal{E}}"] \arrow[d, "+"', "\mathcal{R,E}"] & t_2 \arrow[dd, "\mathcal{R,E}"', "+"] \\
t_1 \arrow[d, "*"', "\mathcal{R,E}"] & \\
s_1 \arrow[r, phantom, "=_{\mathcal{E}}"] & s_2
\end{tikzcd}$$

Figure 12. Coherence modulo E.

$\mathcal{R}$ is *locally coherent modulo* $\mathcal{E}$ if and only if for all terms t, t_1, and t_2 if $t \rightarrow_{\mathcal{R,E}} t_1$ and $t =^1_{\mathcal{E}} t_2$ then there exists terms s_1 and s_2 such that $t_1 \rightarrow^*_{\mathcal{R,E}} s_1$, $t_2 \rightarrow^+_{\mathcal{R,E}} s_2$, and $s_1 =_{\mathcal{E}} s_2$. See Figure 13.

$$\begin{tikzcd}
t \arrow[r, phantom, "=^1_{\mathcal{E}}"] \arrow[d, "\mathcal{R,E}"] & t_2 \arrow[dd, "\mathcal{R,E}"', "+"] \\
t_1 \arrow[d, "*"', "\mathcal{R,E}"] & \\
s_1 \arrow[r, phantom, "=_{\mathcal{E}}"] & s_2
\end{tikzcd}$$

Figure 13. Local coherence modulo E.

Theorem 3.4 (JK86) The following properties are equivalent for an E-terminating set of reductions $\mathcal{R}$.

(1) $\mathcal{R}$ is $\mathcal{E}$-Church-Rosser.

(2) $\mathcal{R}$ is $\mathcal{E}$-confluent and coherent modulo $\mathcal{E}$.

(3) $\mathcal{R}$ is locally $\mathcal{E}$-confluent and locally coherent modulo $\mathcal{E}$.

(4) All $\mathcal{E}$-critical and coherence pairs of $\mathcal{R}$ and $\mathcal{E}$ reduce under $\mathcal{R}$ to $\mathcal{E}$-identities.

The coherence property in the above equivalences can actually be removed when the equations in E are permutative. Christian [Ch89] showed this to be true in his dissertation by showing that E-confluence implied coherence for permutative equational theories and sped up his HIPER implementation by restricting it to permutative theories, thus not having to check for coherence. We will look at Christian's HIPER system in more detail in Section VII since our system, HIPER-extension, is a modified version of this fast completion system.

With the above, the relationship of Theorem 3.3 to the E-confluence property can be appreciated. Since Peterson and Stickel worked with associative and commutative theories and these are permutative we would expect the results to coincide (that is, the coherence property to be removed).

Jouannand and Kirchner developed a completion procedure, seen in Figures 14 and 15, which can find complete sets of reductions modulo an equational theory for more equational theories than that of [PS81] but it too has the problem that a finite, complete unification algorithm has to be provided for the equational theory.

The question of how we can prove completion modulo an equational theory which generates infinite congruence classes has been addressed by Bachmair and Dershowitz [BD89] but Baird [Ba88] noted, of the earlier version [BP87], that the finite termination

```
E-Completion ($\mathcal{R},\mathcal{E}$)
/* $\mathcal{E}$ is the equational theory for which a finite, complete unification algorithm exists;
   $\mathcal{R}$ is a set of reductions */
for all reductions $r = \lambda \rightarrow \rho \in \mathcal{R}$
        Pairs = Compute-critical-pairs(r, $\mathcal{R}$, $\mathcal{E}$)
Start: for all $<t_1,t_2> \in$ Pairs
            $t_1 = t_1\downarrow_{\mathcal{R},\mathcal{E}}$
            $t_2 = t_2\downarrow_{\mathcal{R},\mathcal{E}}$
            if $t_1 = t_2$
                $t_1$ and $t_2$ conflate; goto Start
            else if $t_1 > t_2$
                $\mathcal{R} = \mathcal{R} \cup (t_1 \rightarrow t_2)$
                inter-reduce $\mathcal{R}$
                Pairs = Pairs $\cup$ Compute-critical-pairs(r,$\{t_1 \rightarrow t_2\}$, $\emptyset$)
            else if $t_2 > t_1$
                $\mathcal{R} = \mathcal{R} \cup (t_2 \rightarrow t_1)$
                inter-reduce $\mathcal{R}$
                Pairs = Pairs $\cup$ Compute-critical-pairs(r,$\{t_2 \rightarrow t_1\}$, $\emptyset$)
            else /* $t_1$ and $t_2$ are incomparable */
                Stop with FAILURE
            end if
        end for
end for
Stop with SUCCESS; Return $\mathcal{R}$
```

Figure 14. E-completion procedure.

```
**Compute-critical-pairs** ($r \equiv \lambda \rightarrow \rho, \mathcal{R}, \mathcal{E}$)
Pairs = $\varnothing$
for all $r' = \lambda' \rightarrow \rho' \in \mathcal{R}$
    for all $i \in$ sdom($\lambda$)
        if not ($r' = r$ and $i = ()$ (meaning the top-level term) ) then
            for all $\delta$ which are E-unifiers of $\lambda/i$ and $\lambda'$
                Pairs = Pairs $\cup$ {<$\delta(\rho)$, $\delta(\lambda[i \leftarrow \rho'])$>}
            end for
        end if
    end for
end for

for all $i \in$ sdom($\lambda$)
    for all $\lambda' == \rho' \in \mathcal{E}$
        for all $\delta$ which are E-unifiers of $\lambda/i$ and $\lambda'$
            Pairs = Pairs $\cup$ {<$\delta(\rho)$, $\delta(\lambda[i \leftarrow \rho'])$>}
        end for
        for all $\delta$ which are E-unifiers of $\lambda/i$ and $\rho'$
            Pairs = Pairs $\cup$ {<$\delta(\rho)$, $\delta(\lambda[i \leftarrow \lambda'])$>}
        end for
    end for
end for

Return (Pairs)
```

Figure 15. E-critical pair procedure.

property needed in their model is usually lost, though, in their treatment. Baird then looked at the associative-commutative-identity equational theory. This theory generates infinite congruence classes, but Baird built an ACI-completion procedure which is finitely terminating.

For a list of known complete sets of reductions see [Hul81].

IV. ORDERINGS AND TERMINATION

In Section III the problem of proving termination was not directly addressed. This section will consider E-termination both when $E = \varnothing$ (in which case we will talk simply of 'termination') and $E \neq \varnothing$. First we will look at the case when $E = \varnothing$.

A. TERMINATION

In Section III we gave one ordering which had the finite termination property - the Knuth-Bendix ordering - but there are many other orderings which could have been used instead. The existence of other orderings is quite useful since a set of reductions could be found not to be complete with one ordering but found to be complete with another ordering. Thus different efficiently implemented orderings which have the finite termination property are needed when searching for a complete set of reductions.

Termination is normally proved by using the following theorem [MN70]:

Theorem 4.1 (MN70) A set of reductions $\mathcal{R}$ is terminating, if there exists a well-founded ordering > over all terms such that

$t > s$ implies $f(\ldots t \ldots) > f(\ldots s \ldots)$ (monotonic)

for terms t, s, $f(\ldots t \ldots)$, $f(\ldots s \ldots)$.

and if $\lambda \to \rho \in \mathcal{R}$ then $\lambda\sigma > \rho\sigma$ for all substitutions σ (substitution preserving).

Well-foundedness was defined in Section III.

Dershowitz [De79] presented an important class of orderings called *simplification orderings*. An ordering is a simplification ordering if it is a reduction ordering which has the subterm property (i.e. $f(\ldots t \ldots) > t$, for all terms t). An ordering is a *reduction ordering* if it is monotonic. The Knuth-Bendix ordering is a simplification ordering.

A property of a simplification ordering is that it contains the *embedding relation.* Let $\mathcal{V}(t)$ be the set of all variables in t. If s and t are terms then s is embedded in t, written $s \Rightarrow t$, if and only if

(1) $s \in \mathcal{V}(t)$, or

(2) $s = f(\ldots s_i \ldots)$ and $t = f(\ldots t_i \ldots)$ and $s_i \Rightarrow t_i$ for any i, or

(3) $s \Rightarrow t_i$ for t_i a subterm of t.

Example:

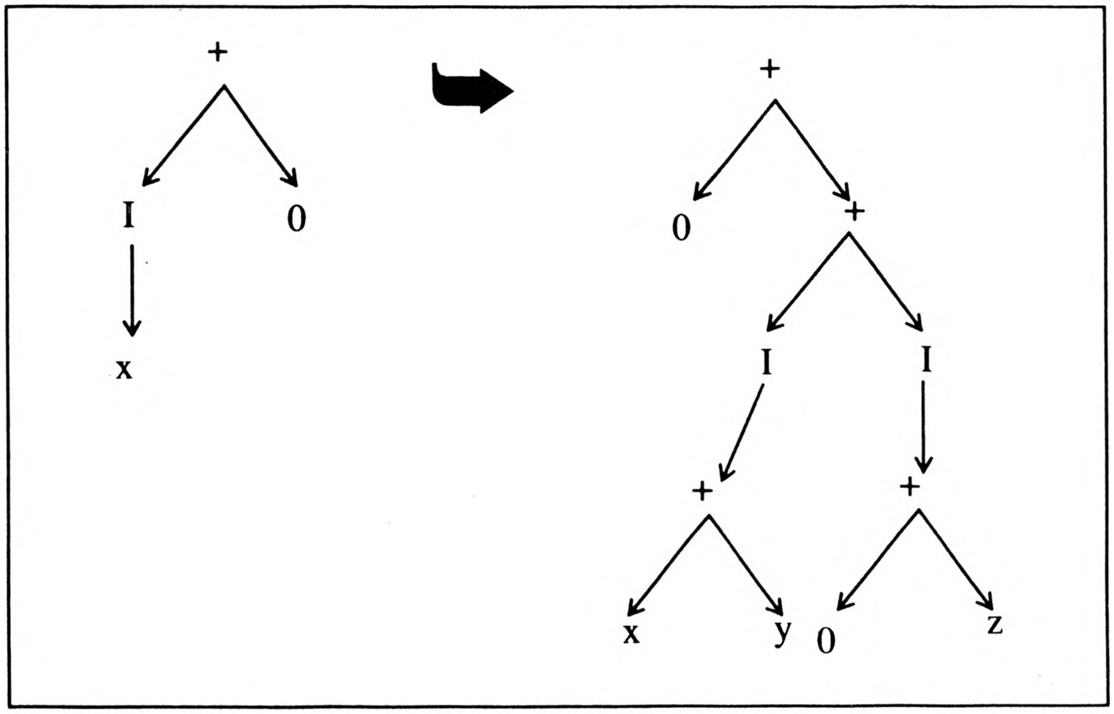

Figure 16. Embedding.

Note that a simplification ordering > does not require > to be well-founded. Despite this Dershowitz proved the following theorem:

Theorem 4.2 (De79) A set of reductions $\mathcal{R}$ terminates if there exists a simplification ordering > such that

for all $\lambda \rightarrow \rho \in \mathcal{R}$ then $\lambda\sigma > \rho\sigma$ for all substitutions σ.

To help in proving termination for other orderings, Dershowitz also showed the following [De79]:

Theorem 4.3 (De79) A set of reductions R terminates if there exists a quasi-simplification ordering > such that

for all $\lambda \to \rho \in \mathcal{R}$ then $\lambda\sigma > \rho\sigma$ for all substitutions σ.

An ordering is a *quasi-simplification ordering* if it is

(1) transitive,

(2) reflexive,

(3) subterm preserving, and

(4) $f(\ldots t \ldots) > t$, for all terms t.

Dershowitz presented a class of simplification orderings in his paper called recursive path orderings. The *recursive path ordering*, $>_{rpo}$, is defined as follows: If > is a partial ordering of the function symbols then $s = f(s_1, \ldots, s_m) >_{rpo} t = g(t_1, \ldots, t_n)$ if and only if

(1) $f = g$ and $\{s_1, \ldots, s_m\} \gg_{rpo} \{t_1, \ldots t_n\}$, or

(2) $f > g$ and $s >_{rpo} t_i$, for all i in $\{1, \ldots, n\}$, or

(3) $f \not\geq g$ and $s_i \geq_{rpo} t$ for some i in $\{1, \ldots, m\}$.

The relation $\gg_{rpo}$ is a multiset ordering defined as $M \gg_{rpo} N$ if and only if $M \neq N$ and for all $y \in N - M$ there exists an $x \in M - N$ such that $x >_{rpo} y$ is a partial ordering.

The recursive path ordering can sometimes orient equations that the KB-ordering cannot. For example, if '-' has a weight of 0 then $(-x) + (-y) = y + x$ cannot be ordered by the KB-ordering but can be ordered left to right by the recursive path ordering.

Another simplification ordering which we are concerned with is the *lexicographic path ordering* (lpo) [De87]. This ordering will be discussed in more detail in the next section so we will only briefly state its definition here.

Let $s = f(s_1, \ldots, s_m)$ and $t = g(t_1, \ldots, t_n)$, $s >_{lpo} t$ if and only if

(1) $s_i \geq_{lpo} t$ for some $i \in \{1, \ldots, m\}$.

(2) $f > g$ and $s >_{lpo} t_j$ for all $j \in \{1, \ldots, n\}$.

(3) $f = g$ and for some $k \in \{1, \ldots, n\}$, $s_i = t_i$ for all $i < k$, $s_k >_{lpo} t_k$, and $s >_{lpo} t_i$ when $k < i \leq n$.

Other orderings include the recursive decomposition ordering [Le82], and the path of subterms ordering [Pl78]. For surveys on orderings see [De87] and [DJ90].

B. E-TERMINATION

Lets now look at $\mathcal{E}$-termination where $\mathcal{E} \neq \emptyset$. A definition of $\mathcal{E}$-termination is: $\mathcal{R}$ is *E-terminating* if and only if there exist no infinite sequence of terms $t_1, t_2, t_3, \ldots$ such that $t_1 =_{\mathcal{E}} t_1' \to_{\mathcal{R}} t_2 =_{\mathcal{E}} t_2' \to_{\mathcal{R}} t_3 =_{\mathcal{E}} t_3' \ldots$. Note that $\to_{\mathcal{R},\mathcal{E}} \subset =_{\mathcal{E}} \circ \to_{\mathcal{R}}$ since in $\to_{\mathcal{R},\mathcal{E}}$ equations can only be applied on a term at or below the subterm $\mathcal{R}$ is applied while equations can be applied anywhere in $=_{\mathcal{E}} \circ \to_{\mathcal{R}}$.

Example:

Let $\mathcal{R} = \{x+e \to x\}$ and $\mathcal{E} = \{x+(y+z) = (x+y)+z\}$ then $y+(e+x)$ is in normal form for $\to_{\mathcal{R},\mathcal{E}}$ but is equal to $y+x$ using $=_{\mathcal{E}} \circ \to_{\mathcal{R}}$.

$\mathcal{E}$-termination has been shown using polynomial orderings [HO80] and associative path orderings [BP85] but are limited to associative, commutative, and associative-commutative theories. Christian [Ch89] also gives two new orderings for simple, linear, permutative theories. A more general investigation of the problem of $\mathcal{E}$-termination was given in [JM84]. Though no orderings were given in the paper, general criteria were given for $\mathcal{E}$-termination to hold for a relation.

Let us look at the results of [JM84]. First, restrictions for $\mathcal{E}$-termination to hold were noted.

(1) For equations $\lambda == \rho$, $\mathcal{V}(\lambda) = \mathcal{V}(\rho)$. This prevents the following from happening. Given $f(x,y) = g(x)$ and $I(x) \to e$ then $f(x,y) \to f(x,I(x)) = g(x) = f(x,y) \ldots$

(2) If a term s has more than one occurrence in t then equations cannot be of the form s $==$ t. This prevents the following from happening. Given $f(x) = g(f(x),f(x))$ and $f(x) \to e$ then $f(x) = g(f(x),f(x)) \to g(e,f(x)) = g(e,g(f(x),f(x)) \ldots$

For the rest of this discussion let $\to$ be a rewriting relation such that $\to_{\mathcal{R}} \subseteq \to \subseteq =_{\mathcal{E}} \circ \to_{\mathcal{R}} \circ =_{\mathcal{E}}$. The relation $\to$ is *E-commuting* if and only if for all terms t, t_2, and s_1 if $t_2 =_{\mathcal{E}} t$ and $t \to^{+} s_1$ then there exists a term s_2 such that $t_2 \to^{+} s_2$ and $s_2 =_{\mathcal{E}} s_1$. See Figure 17.

$$\begin{tikzcd}
t \arrow[r, phantom, "=_{\mathcal{E}}"] \arrow[d, "+"'] & t_2 \arrow[d, "+"] \\
s_1 \arrow[r, phantom, "=_{\mathcal{E}}"] & s_2
\end{tikzcd}$$

Figure 17. E-Commutation.

The relation $\to$ is *locally E-commuting* if and only if for all terms t, t_2, and s_1 if $t_2 =^{1}_{\mathcal{E}} t$ and $t \to s_1$ then there exists a term s_2 such that $t_2 \to^{+} s_2$ and $s_2 =_{\mathcal{E}} s_1$. See Figure 18.

$$\begin{tikzcd}
t \arrow[r, phantom, "=^1_{\mathcal{E}}"] \arrow[d] & t_2 \arrow[d, "+"] \\
s_1 \arrow[r, phantom, "=_{\mathcal{E}}"] & s_2
\end{tikzcd}$$

Figure 18. Local E-Commutation.

Theorem 4.4 (JM84) For *R*, a set of reductions, and *E*, a set of equations, *R* is *E*-terminating if $\rightarrow$ is terminating and (locally) *E*-commuting.

Note that commutation is stronger than coherence and thus:

Theorem 4.5 (JM84) The relation $\rightarrow$ is *E*-Church-Rosser if and only if it is terminating, (locally) *E*-confluent, and (locally) *E*-commuting.

If $\rightarrow$ is not E-commuting then E-termination can still be shown if we can find another relation > which is *E-commuting with* $\rightarrow$. A relation > is *E-commuting with* $\rightarrow$ if and only if > contains the embedding relation and for all t, t_2, and s_1 if $t_2 =_{\mathcal{E}} t$ and $t \rightarrow s_1$ then there exists a term s_2 such that $t_2 > s_2$ and $s_2 =_{\mathcal{E}} s_1$. If such a relation exists then $\rightarrow_{\mathcal{R}}$ is *E*-terminating. See Figure 19.

$$\begin{tikzcd}
t_2 \arrow[r, phantom, ">"] \arrow[d, phantom, "\parallel_{\mathcal{E}}"] & s_2 \arrow[d, phantom, "\parallel_{\mathcal{E}}"] \\
t \arrow[r] & s_1
\end{tikzcd}$$

Figure 19. > is E-Commuting with $\rightarrow$.

For another method of proving termination of rewriting systems see [DM79], in which Dershowitz and Manna use multiset orderings for its termination proofs and also present other types of orderings

V. SOME EXTENSIONS TO COMPLETION

A. UNFAILING COMPLETION

An alternative to standard KB-completion is the notion of unfailing completion. In this method un-orientable equations can be handled in such a way as to guarantee the existence of a decision procedure for the associated word problem, if one exists.

The KB-completion procedure will not always find a complete set of reductions when one exists since the sequence of attempted completion can affect its outcome. An example taken from [De89] in which $k > m$, $k > n$, $m \# n$, $m > c$, and $n > c$ (where $m \# n$ means that m and n are incomparable through the ordering $>$) has a sequence which forms a complete set of reductions with

$(\{k = m, k = n, f(k) = c\}, \{f(m) \to m\}) \Rightarrow$

$(\{k = n, f(k) = c\}, \{k \to m, f(m) \to m\}) \Rightarrow^+$

$(\{m = n, m = c\}, \{k \to m, f(m) \to m\}) \Rightarrow^+$

$(\{m = n\}, \{m \to c, k \to m, f(m) \to m\}) \Rightarrow^+$

$(\{f(c) = c, c = n\}, \{m \to c, k \to c\}) \Rightarrow^+$

$(\varnothing, \{f(c) \to c, n \to c, m \to c, k \to c\})$

and one which does not form a complete set of reductions

$(\{k = m, k = n, f(k) = c\}, \{f(m) \to m\}) \Rightarrow$

$(\{k = m, f(k) = c\}, \{k \to n, f(m) \to m\}) \Rightarrow^+$

$(\{n = m\}, \{f(n) \to c, k \to n, f(m) \to m)$.

Thus a backtracking algorithm might be useful but sometimes the initial problem fails so backtracking over sequences will not help. Function symbol introduction (see

[KB70]) sometimes helps but will not work on many types of permutative theories such as commutativity. If we can totally order the ground terms with a reduction ordering, which we always can, we can handle un-orientable equations by carrying them along in an unfailing completion procedure.

Hsiang and Rusinowitch presented such a completion procedure which does not fail in their 1987 paper [HR87]. Their results build on Huet's [Hu81] who produced semi-decision procedures for the word problem for non-confluent rewriting systems. His idea was that given an equational theory E and an equation $s = t$ then letting $s' \neq t'$ be the skolemized inequality of the negation of $s = t$ then $s = t$ is a result of E if and only if the KB-completion procedure eventually reduces $s' \neq t'$ to some $r \neq r$. Skolemization of not(s=t) results in s' and t' being ground terms since variables are replaced by skolem constants in s' and t'.

Hsiang and Rusinowitch required that a ground-linear simplification ordering exist on the terms $\mathcal{T}$ for their unfailing completion procedure. An ordering is a *ground-linear simplification ordering* > if it is an ordering such that

(1) it has the subterm property ($f(\ldots t\ldots) > t$),

(2) it is monotonic (if $s > t$ then $f(\ldots s\ldots) > f(\ldots t\ldots)$),

(3) it is substitution preserving (if $s > t$ then $s\sigma > t\sigma$), and

(4) > is total on the set of ground terms.

For their method the notions of critical pairs and rewriting are slightly changed. Given two equations $s = t$ and $l = r$ if $i \in$ sdom(s) and σ is a mgu of s/i and l and

(1) $r\sigma \not\geq l\sigma$, and

(2) $t\sigma \not\geq s\sigma$,

then $<t\sigma, s[i \leftarrow r\sigma]>$ is an *extended critical pair.*

The new *extended superposition process* now must generate these extended critical pairs. We should note that if the equations $s = t$ and $l = r$ are orientable then (1) becomes $l\sigma > r\sigma$ and (2) becomes $s\sigma > t\sigma$ and the process is the same as that for KB-completion.

Reductions are modified so that $s \rightarrow t$ if there exists an equation $l = r$ such that $s/i = l\sigma$, $l\sigma > r\sigma$, and $t = s[i \leftarrow r\sigma]$ for some $i \in \text{sdom}(s)$ and some substitution σ. Note that we do not require $l > r$, only $l\sigma > r\sigma$.

There are three basic steps in the unfailing completion procedure.

(1) Generate extended critical pairs and put into normal form - using existing equations. If the pair does not conflate, add it as a rewrite rule if possible.

(2) Reduce $s' \neq t'$ using the new equation, if possible.

(3) If $r \neq r$ is generated, then use $r \neq r$ and $x = x$ to produce a contradiction.

Assuming that the above procedure (UKB) is fair then the following theorem holds:

Theorem 5.1 (HR87) Given E and $s = t$, $s = t$ is a result of E if and only if UKB, applied to $E \cup (s' \neq t')$, produces a contradiction.

This says that we can prove $s =_E t$ by proving a contradiction in UKB for $E \cup (s' \neq t')$.

To work on word problems with the above we need to know the following

Theorem 5.2 (HR87) If all equations are orientable and the system is confluent then the set of reductions is complete.

But if we have non-orientable equations we can use this

Theorem 5.3 (HR87) If the system is confluent then the set of equations is complete on ground terms.

Confluence in the above two theorems being modified for our new ideas on reductions and critical pairs.

If Theorem 5.2 holds the word problem can be solved as with the KB-completion procedure, but what about if only Theorem 5.3 holds - i.e. only ground completeness holds. For this case we see that unique normal forms do not always exist but that a linear decision procedure exists to solve the word problem even for non-ground terms. The procedure is listed below in Figure 20.

```
Solve-Word-Problem (E, s = t)
    /* E is our ground complete set of equations; s = t is our problem */
    (s′ ≠ t′) := Skolemize( not(s = t))
    /* s′ and t′ have unique normal forms since they are ground and E is ground
       confluent */
    s′ := s′↓
    t′ := t′↓
    if s′ = t′ then
        return(TRUE)
    else
        return(FALSE)
    end if
end Solve-Word-Problem
```

Figure 20. Solving word problems with ground complete systems.

Other treatments of unfailing completion appear in [BDP89] and [MN90].

B. COMPLETION WITH CONDITIONAL REWRITE RULES

Conditional rewrite systems have been investigated for application in the area of abstract data type specification. We will look at it now to note how it relates to the constrained systems described in Section VII. A *(positive-)conditional equation* is of the form:

$s = t$ if $s_1 = t_1 \wedge s_2 = t_2 \wedge \ldots \wedge s_n = t_n$

where $n \geq 0$ and where $s = t$ can be thought of as the result and $s_1 = t_1 \wedge s_2 = t_2 \wedge \ldots \wedge s_n = t_n$ the conditions. Given $s = t$ if $s_1 = t_1 \wedge s_2 = t_2 \wedge \ldots \wedge s_n = t_n$, $u \leftrightarrow u[j \leftarrow t2]$, if $u/j = s\sigma$ for some substitution σ and $s_i\sigma \leftrightarrow^* t_i\sigma$ for all i.

A *conditional rule* is a conditional equation in which the right-hand side is oriented (e.g. $s \rightarrow t$ if $s_1 = t_1 \wedge s_2 = t_2 \wedge \ldots \wedge s_n = t_n$). A set of conditional rules is called a *conditional rewrite system*. We define conditional rewriting as follows: $u \rightarrow u[j \leftarrow r\sigma]$, if $u/j = l\sigma$, $l \rightarrow r$ if c is a conditional rule, and $c\sigma$ meets the test for the conditions. This test can be one of several and the different types will be listed below.

If s and t are *joinable*, written $s \downarrow t$, then $s \rightarrow^* v$ and $t \rightarrow^* v$ for some term v. The notions of irreducibility, termination, and (local) confluence are the same as before except that they are defined relative to the new relation $\rightarrow$.

Seven different types of tests for conditions are listed in [DOS88]. In the below σ is the substitution mentioned in the rewriting relation. The different types are:

(1) Semi-equational systems

- Conditions $s_1 = t_1 \wedge s_2 = t_2 \wedge \ldots \wedge s_n = t_n$ are checked for truth by seeing if $s_i\sigma \leftrightarrow t_i\sigma$ for all i.

(2) Join systems

- Conditions $s_1 = t_1 \wedge s_2 = t_2 \wedge \ldots \wedge s_n = t_n$ are checked for truth by seeing if $s_i\sigma \downarrow t_i\sigma$ for all i.

(3) Normal-join systems

- Conditions $s_1 = t_1 \wedge s_2 = t_2 \wedge \ldots \wedge s_n = t_n$ are checked for truth by seeing if $s_i\sigma \downarrow^! t_i\sigma$ for all i. We write $s_i \downarrow^! t_i$ is the same as $s_i \downarrow t_i$ except there must also exist a term v which is irreducible such that both s_i and t_i both rewrite to v.

(4) Normal systems

- Conditions $s_1 = t_1 \wedge s_2 = t_2 \wedge \ldots \wedge s_n = t_n$ are checked for truth by seeing if $s_i\sigma \rightarrow^! t_i\sigma$ for all i. We let $s_i \rightarrow^! t_i$, if $s_i \rightarrow^* t_i$ and t_i is an irreducible ground term.

(5) Inner-join systems

- Conditions $s_1 = t_1 \wedge s_2 = t_2 \wedge \ldots \wedge s_n = t_n$ are checked for truth by seeing if $s_i\sigma \downarrow^{in} t_i\sigma$ for all i. We write $s_i \downarrow^{in} t_i$ if s_i and t_i are joinable by innermost rewriting. *Innermost rewriting* requires that rewriting can occur only if all subterms are irreducible.

(6) Outer-join systems

- Conditions $s_1 = t_1 \wedge s_2 = t_2 \wedge \ldots \wedge s_n = t_n$ are checked for truth by seeing if $s_i\sigma \downarrow^{out} t_i\sigma$ for all i. We write $s_i \downarrow^{out} t_i$ if s_i and t_i are joinable by outermost rewriting. *Outermost rewriting* requires that rewriting can occur only if no superterm can be reduced.

(7) Meta-conditional systems

- Other types of conditions are allowed in the condition such as $x \in X$, $s > t$, $s \rightarrow^! t$, etc.

Join systems are the types of systems most commonly used.

Critical pairs between conditional rewrite rules $s_1 = t_1$ if c_1 and $s_2 = t_2$ if c_2 are formed as we would expect: $<t_1\sigma = s_1\sigma[i \leftarrow t_2\sigma]$ if $(c_1 \wedge c_2)\sigma>$ is a *critical pair* if $s_1/i = s_2\sigma$. The critical pair is an *overlay* if i = (). A critical pair s = t if c is *feasible* if $c\sigma \rightarrow^*$ true for some substitution σ. A critical pair s = t if c is joinable if for all feasible solutions $s\sigma \downarrow t\sigma$.

For unconditional rewrite systems we have, from Section III, that the system is confluent if and only if every critical pair is joinable. To prove confluence for the various types of conditional systems Dershowitz, et. al. [DOS88] showed the following are sufficient conditions:

(1) Semi-equational systems

- Termination and all critical pairs are joinable.

(2) Join systems

- Termination and all critical pairs are overlays and joinable.

(3) Inner systems

- Termination and all critical pairs are joinable.

Let E be an equational system and R be a rewrite system. E $\vdash$ s = t if and only if $s \leftrightarrow^* t$ is provable from E. R $\vdash$ $s \downarrow t$ means that s and t are joinable using R. E and R have the same logical strength if E $\vdash$ s = t if and only if R $\vdash$ $s \downarrow t$. If one system is stronger than the other then if the weaker system is complete then so is the stronger system. Dershowitz, et. al. [DOS88] gave the relationships between the various systems and we depict this in Figure 21. In the figure, A $\rightarrow$ B if A is stronger that B, in general.

Confluency conditions for some of the other systems are also presented in the paper. For a look at conditional rewriting when restricted to left-linear systems without any critical pairs see [BK86].

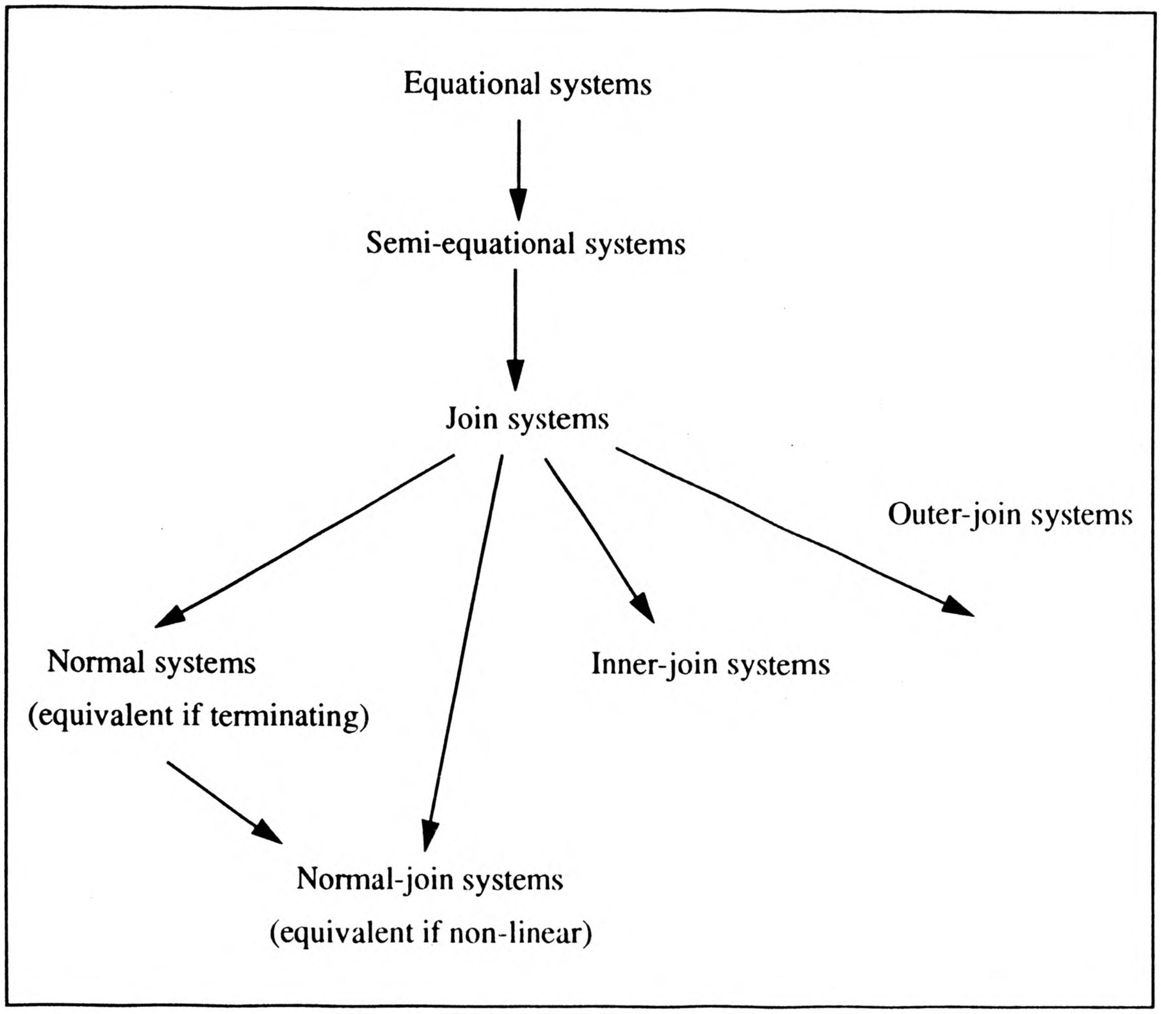

Figure 21. Logical strengths of various conditional systems.

For another look at conditional rewrite systems see [ZR85].

VI. SOLVING INEQUALITIES USING LEXICOGRAPHIC PATH ORDERINGS

For the next section on constrained completion we need to have method for deriving constraints for a given inequality problem. That is, we want to derive a constraint using an ordering which tells us when we can apply a reduction (e.g for a commutativity axiom, $f(x,y) > f(y,x)$, the constraint using the lexicographic path ordering is $x > y$).

To do this we first need to state some definitions. If s and t are terms, a *term inequality* is a relation of the form $s > t$; *term equalities* are relations of the form $s = t$. A *term problem* consists of term equalities, term inequalities, and the logical operators not ($\neg$), and ($\wedge$), and or ($\vee$). In the discussion which follows > will be interpreted as the lexicographic path ordering.

Comon [Co90] studied this problem of term inequalities to apply to unfailing completion by showing the existence of a ground substitution for a term inequality is a decidable problem. This is useful in the generation of critical pairs. Comon's treatment of the subject is well-presented but we will look at Peterson's work [Pe90a] since his is oriented towards a machine implementation and is the work which our implementation is based upon.

Some definitions from [Pe90a] are: A *simple term (in)equality* is an (in)equality of the form $x = t$ ($x > t$) where $x \in \mathcal{V}$ and $x \notin Var(t)$. A *simple component* is **T** (true) or a logically independent conjunction of simple term inequalities. A *component* is a pair $<\theta,\alpha>$ where θ is a substitution, α is a simple component, and if $(x/t) \in \theta$ then $x \notin \bigcup_{c} Var(c)$ where c is an inequality in α. A *partition* is a disjunction of components in which the disjuncts are pair-wise disjoint (i.e. $c_1 \wedge c_2 \neq$ **F** (false)).

Peterson [Pe91b] gives thirty-four different rules to produce a constraint for a given term inequality based on the lexicographic path ordering (see Section IV for its definition).

```
**Solve-Term-Problem**(P,C,p,E)
/* we will abbreviate this by STP(P,C,p,E) */
(1)  $STP(\mathbf{T}, C, p, E) \rightarrow \mathbf{T}$
(2)  $STP(P, \mathbf{T}, \mathbf{T}, \mathbf{T}) \rightarrow \mathbf{T}$
(3a) $STP(P, \mathbf{F}, p, E) \rightarrow P$
(3b) $STP(P, C, \mathbf{F}, E) \rightarrow P$
(3c) $STP(P, C, p, \mathbf{F}) \rightarrow P$
(4)  $STP(P, C, \mathbf{T}, \mathbf{T}) \rightarrow P \vee C$
(5)  $STP(P, C, \mathbf{T}, E) \rightarrow STP(P, C, E, \mathbf{T})$
(6)  $STP(P, C, p \wedge q, \mathbf{T}) \rightarrow STP(P, C, p, q)$
(7)  $STP(P, C, p \wedge q, E) \rightarrow STP(P, C, p, q \wedge E)$
(8)  $STP(P, C, p \vee q, E) \rightarrow STP(\, STP(P, C, p, E), C, q, E \wedge (\neg p))$
(9)  $STP(P, C, \neg\mathbf{F}, E) \rightarrow STP(P, C, E, \mathbf{T})$
(10) $STP(P, C, \neg\mathbf{T}, E) \rightarrow P$
(11) $STP(P, C, \neg\neg p, E) \rightarrow STP(P, C, p, E)$
(12) $STP(P, C, \neg(p \wedge q), E) \rightarrow STP(P, C, \neg p \vee \neg q, E)$
(13) $STP(P, C, \neg(p \vee q), E) \rightarrow STP(P, C, \neg p, \neg q \wedge E)$
(14) $STP(P, C, \neg(p = q), E) \rightarrow STP(\, STP(P, C, (p > q), E), C, q > p, E)$
(15) $STP(P, C, \neg(p > q), E) \rightarrow STP(\, STP(P, C, (p = q), E), C, q > p, E)$
For rules 16-18, let $\theta$ be the mgu of p and q
(16) if $\theta$ does not exist, then $STP(P, C, p = q, E) \rightarrow P$
(17) $STP(P, \mathbf{T}, p = q, E) \rightarrow STP(P, (\theta,\mathbf{T}), \mathbf{T}, E\theta)$
(18) $STP(P, (\phi,\alpha), p = q, E) \rightarrow STP(P, (\theta\phi,\mathbf{T}), \alpha\theta, E\theta)$
(19) if $u \in Var(v)$, then $STP(P, C, u > v, E) \rightarrow P$
(20) if $v \in Var(u)$, then $STP(P, C, u > v, E) \rightarrow STP(P, C, E, \mathbf{T})$
(21) if u or v is a variable and $C \Rightarrow u > v$, then $STP(P, C, u > v, E) \rightarrow STP(P, C, E, \mathbf{T})$
(22) if u or v is a variable and $C \Rightarrow v > u$, then $STP(P, C, u > v, E) \rightarrow P$
(23) if u or v is a variable, then $STP(P,C,u > v,E) \rightarrow STP(P,AndTogether(u > v,C),E,\mathbf{T})$
```

Figure 22. a.) Procedure for solving term problems (continued).

For rules 24-26 let $u = f(u_1, \ldots, u_n)$ and $v = g(v_1, \ldots, v_n)$

(24) if $f > g$, then $STP(P, C, u > v, E) \rightarrow STP(P, C, (\text{all } u\ (v_1, \ldots, v_m)), E)$

(25) if $g > f$, then $STP(P, C, u > v, E) \rightarrow STP(P, C, (\text{some } (u_1, \ldots, u_n)\ v), E)$

(26) if $f = g$, then $STP(P, C, u > v, E) \rightarrow$

$STP(\ STP(P, C, (\text{lex } u\ (u_1, \ldots, u_n)\ (v_1, \ldots, v_m)), E),$
$\quad C, (\text{lexsome } v\ (v_1)\ (u_1)\ (v_2, \ldots, v_n)\ (u_2, \ldots, u_n)), E)$

(27) $STP(P, C, (\text{all } u \text{ nil}), E) \rightarrow STP(P, C, E, \mathbf{T})$

(28) $STP(P, C, (\text{all } u\ (v_1, \ldots, v_m)), E) \rightarrow STP(P, C, (u > v_1) \wedge (\text{all } u\ (v_2, \ldots, v_m)), E)$

(29) $STP(P, C, (\text{some nil } v), E) \rightarrow P$

(30) $STP(P, C, (\text{some } (u_1, \ldots, u_n)\ v), E) \rightarrow$

$STP(\ STP(\ STP(P, C, (u_1 = v), E),$
$\qquad C, (u_1 > v), E),$
$\quad C, (\text{some } (u_2, \ldots, u_n)\ v)), E \wedge (v > u_1))$

(31) $STP(P, C, (\text{lex } u \text{ nil nil}), E) \rightarrow P$

(32) $STP(P, C, (\text{lex } u\ (u_1, \ldots, u_n)\ (v_1, \ldots, v_n)), E) \rightarrow$

$STP(\ STP(P, C, (u_1 > v_1) \wedge (\text{all } u\ (v_2, \ldots, v_n)), E),$
$\quad C, (u_1 = v_1) \wedge (\text{lex } u\ (u_2, \ldots, u_n)\ (v_2, \ldots, v_n), E)$

(33) $STP(P, C, (\text{lexsome } v\ (v_1, \ldots, v_n)\ (u_1, \ldots, u_n) \text{ nil nil}), E) \rightarrow P$

(34) $STP(P, C, (\text{lexsome } v\ (v_1, \ldots, v_{k-1})\ (u_1, \ldots, u_{k-1})\ (v_k, \ldots, v_n)\ (u_k, \ldots, u_n)), E) \rightarrow$

$STP(\ STP(\ STP(P, C, u_k = v, E \wedge (\text{lex } v\ (v_1, \ldots, v_{k-1})\ (u_1, \ldots, u_{k-1}))),$
$\qquad C, (u_k > v), E \wedge (\text{lex } v\ (v_1, \ldots, v_{k-1})\ (u_1, \ldots, u_{k-1}))),$
$\quad C, (\text{lexsome } v\ (v_1, \ldots, v_k)\ (u_1, \ldots u_k)\ (v_{k+1}, \ldots, v_n)\ (u_{k+1}, \ldots, u_n)), E)$

Figure 22 (continued). b.) Procedure for solving term problems.

These are listed in Figure 22. The initial input is STP(**F**,**T**,p,**T**) where p is the term problem. In general, STP(P,C,p,E) has parts P, for the accumulating partition; C, for simple equality and inequality components; p, for the current part of the problem; and E, for the rest of the problem. In rule 23, *AndTogether*($u > v$, C) is $(u > v) \wedge C$ reduced to make it logically independent. Peterson also gives a method to determine if $\alpha \Rightarrow s > t$ in his paper [Pe91b].

Examples using STP (in the below $\bullet > +$):

- For the term problem $x \bullet (y + z) > x + (z \bullet y)$, the constraint is $(x = z) \vee (x > z)$.
- For the term problem $x + (y \bullet z) > y + (z + x)$, the constraint is $(y > x) \vee (x = y) \vee (x > y \wedge z > x) \vee (x = z \wedge z > y) \vee (x > y \wedge x > z)$.
- For the term problem, $x \bullet (y \bullet (x \bullet z)) > x \bullet (x \bullet (y \bullet z))$ the constraint is $y > x$.

Peterson also wished to keep the constraints in a reduced form so he performed two operations on constraints gotten from the STP procedure. First, he 'factored' the constraints. A *factor* of a component C is a simple (in)equality E such that $C \Rightarrow E$. The process of factoring E out of C gives a component D such that $E \wedge D = C$ and D is *logically independent relative to* E. This process is shown in Figure 23. The actual code for this is written in LISP (see [St84]) for use with many of Christian's HIPER functions [Ch89].

To factor a component P we take a factor G and get $P_1 \vee (G \wedge P_2)$ where P_1 is made up of components which G does not factor while P_2 has G as a factor. If P_2 reduces to true (i.e. it is of the form $(s > t) \vee (s = t) \vee (t > s)$) then we are left with $P_1 \vee G$ and we try to find more factors. Else, we try to factor P_2 further with factor G_2 giving $P_1 \vee (G \wedge P_{21}) \vee (G \wedge G_2 \wedge P_{22})$ where P_{21} is not factored by G_2 and P_{22} is. Factoring in the presence of G is slightly different, it means that we find a G_2 such that $G \wedge C \Rightarrow G_2$ and P_{22} is made up of components such that $G \wedge G_2 \wedge P_{22} \Rightarrow C$ and P_{22} is logically independent relative to

```
factor-out(E,D)
/* This is initially called with factor(E,C) */
found = false
for all d such that D=d ∧ D′ for some D′
    if D′ ∧ E ⇒ d then
        found = true
        break from loop
    end if
end for
if found = true then
    return factor-out(E,D′)
else
    return (D)
end if
```

Figure 23. Factor-out procedure.

$G \wedge G_2$. The term P_{21} is the conjunction of the components of P_2 which are not factored by G_2 in the presence of G.

The benefit of this factoring can be seen in the below examples which significantly reduces the complexity of constraints.

Examples using the factor procedure (in the below $\bullet > +$):

- For the term problem $x + (y \bullet z) > y + (z + x)$ the constraint from STP is $(y > x) \vee (x = y) \vee (x > y \wedge z > x) \vee (x = z \wedge z > y) \vee (x > y \wedge x > z)$. After factoring this constraint is simply **T** (meaning that the constraint is unconditionally applied).
- For the term problem $x + (y + z) \bullet x > z + (y + x \bullet z)$ the constraint from STP is $(z > x) \vee (x = z) \vee (x > z \wedge y > x) \vee (x = y \wedge y > z) \vee (x = y + z) \vee (x > y \wedge x > z \wedge y + z > x)$. After factoring this constraint is $(x = y + z) \vee (y + z > x)$.

Peterson also presents a normal form for constraints in [Pe90b] to further simplify the constraints and the process is similar to that in [Co90]. The following steps will be referred to as the *normalform* function in the next section. First, NOTs are eliminated using the usual equalities from logic

(1) $\neg\mathbf{T} \rightarrow \mathbf{F}$.

(2) $\neg\mathbf{F} \rightarrow \mathbf{T}$.

(3) $\neg\neg x \rightarrow x$.

(4) $\neg(x > y) \rightarrow (x = y) \vee (y > x)$.

(5) $\neg(x \wedge y) \rightarrow \neg x \vee \neg y$.

(6) $\neg(x \vee y) \rightarrow \neg x \wedge \neg y$.

Then the constraints are put in disjunctive normal form using

(1) $x \wedge (y \vee z) \rightarrow (x \wedge y) \vee (x \wedge z)$, and

(2) $(x \vee y) \wedge z \rightarrow (x \wedge z) \vee (y \wedge z)$.

Next, an equation is made for each disjunct (i.e. $s = t$ if $c_1 \vee c_2 \vee \ldots \vee c_n \Rightarrow s = t$ if c_1, $s = t$ if c_2, …, $s = t$ if c_n). The reason for this will be explained in the next section. The constants **T** and **F** are removed now with

(1) $x \wedge \mathbf{T} \rightarrow x$, and

(2) $\mathbf{T} \wedge x \rightarrow x$, and

(3) $\mathbf{F} \wedge x \rightarrow \mathbf{F}$, and

(4) $x \wedge \mathbf{F} \rightarrow \mathbf{F}$.

Equalities are removed next with

(1) $(s = t) \wedge C \rightarrow \mathbf{F}$ if s and t do not unify, and

(2) $(s = t) \wedge C \rightarrow C\sigma$ if s and t have mgu σ.

Again the justification of this will be given in the next section. Notice that now the constraint is either **T** or a conjunction of inequalities. We will now eliminate redundancies ($x > y \Rightarrow x\bullet 1 > y$), tautologies ($x\bullet 1 > x \Rightarrow \mathbf{T}$), inconsistencies ($x > y \wedge y > x \Rightarrow \mathbf{F}$), and add transitive implications ($x > y \wedge y > z \Rightarrow x > z$). This function is shown in Figure 24. The initial input is *greater-than-normal*($\varnothing$,C).

```
greater-than-normal($c_1$,$c_2$)
if $c_2 = \varnothing$ then return $c_1$
/* else $c_2 = s > t \wedge C$ */
while ($(s > t) = u[i \leftarrow l] > u[i \leftarrow r]$ AND $i \neq ()$)
    $s > t = l > r$
if $t/i = s$ then return F
if $s/i = t$ then return greater-than-normal($c_1$,c)
if $u/i = t$ for some $s > u \in c_1$ then return greater-than-normal($c_1$,c)
if $s/i = u$ for some $u > t \in c_1$ then return greater-than-normal($c_1$,c)
if $s > u \in c_1$ AND $t/i = u$ then $c_1 = c_1 - (s > u)$
if $u > t \in c_1$ AND $u/i = s$ then $c_1 = c_1 - (u > t)$
if $u > v \in c_1$ AND $v/i = s$ then $c_2 = c_2 \cup \{u > v[i \leftarrow t]\}$
if $u > v \in c_1$ AND $t/i = u$ then $c_2 = c_2 \cup \{s > t[i \leftarrow v]\}$
return greater-than-normal($c_1 \wedge (s > t)$, c)
```

Figure 24. Normal form procedure.

Peterson mentions that he has no proof that this function always terminates but that it has always done so for him. Our experiments concur with his and agree that it will likely always do so. A proof of this though would be desirable. More interestingly, though, the above methods for reducing constraints does not guarantee that all inconsistencies are found. In particular if $a > b$ and a and b are adjacent then $(x < a) \wedge (x > b)$ should reduce to false but this will not be found with the above method. Comon gives a method to solve this oversight [Co90] but his solution makes the problem NP-hard. Since Peterson's completion procedure (and thus ours) does not seriously degrade by not checking for this it is correct not to search for these inconsistencies in the implementation.

VII. COMPLETE SETS OF REDUCTIONS WITH CONSTRAINTS

The biggest problem with completion procedures modulo an equational theory is the requirement that a finite, complete unification algorithm exist for the equational theory. These unification algorithms must normally be created as they are needed and thus much time must be spent creating and optimizing these algorithms (along with matching algorithms) before completion can even be attempted. This can be a daunting task when it is not known whether or not we will find a complete set of reductions and especially if the algorithms are so specialized that it is unlikely that they can be used on other problems too. Although specialized unification algorithms will normally work better when they exist, we will eliminate the need for specialized unification algorithms by putting constraints on the rewrite rules.

A constraint will restrict the applicability of a rewrite rule but not its truth. For example, the constraint for commutativity, $f(x,y) \rightarrow f(y,x)$, is $x > y$. By writing our rewrite rules as $f(x,y) \rightarrow f(y,x)$ if $x > y$ we can restrict the application of it as long as we can determine if the constraint is true. We avoid rewriting $f(a,b) \rightarrow f(b,a) \rightarrow f(a,b) \rightarrow \ldots$ since either $a > b$ or $b > a$ and thus only one of these rewrites can be performed. Thus we can orient identities which could not be oriented by the KB-ordering (and all of the other orderings we have studied). We are then not forced to use a specialized unification algorithms ever since, from Section VI, we showed that every identity is orientable using the lexicographic path ordering (the ordering we will use).

Note that constraints are not the same as conditions since constraints only restrict when we can apply the reduction while conditions tell us when the identity is true. Using our commutativity example, we know that $f(x,y) = f(y,x)$ at all times while the constraint on when we apply it is if $x > y$. But if we have the equation $x + y = x$ (under normal algebraic rules), we see that the condition on this rule is that $y = 0$. When we define constraints formally we will see that their forms also differ since conditions are only disjunctions of equations while constraints allow a wider selection of symbols. Thus conditional and constrained completion are only loosely related.

Unfailing completion handled rules like commutativity by carrying them along through the procedure but, as Kirchner, et. al. noted in [KKR90], constrained completion often finds a complete set of reductions when unfailing completion will not.

Peterson introduced constrained completion in his 1990 paper [Pe90b] and said that his inspiration for this came from working on the ACI-completion procedure ([Ba88] and [BPW89]) in which constraints were needed in some places. He wanted to back up and apply these constraints to a completion procedure in general.

We will now formally define what we mean by a constraint. A *constraint* is a formula made up of terms, logical connectives ($\wedge$, $\vee$, $\neg$), $>$, $=$, **T**, and **F**. For example $(x > y) \vee \neg(y = x \bullet z)$ is a constraint. Thus a *constrained equation* is of the form ($\lambda = \rho$ if c) where $\lambda = \rho$ is an identity and c is a constraint. We will require that $\mathrm{Var}(\rho) \cup \mathrm{Var}(c) \subseteq \mathrm{Var}(\lambda)$, where Var(c) is the union of Var applied to each term in c. The constraints for the identities which will be given to our completion procedure will be generated using the *STP* procedure from Section VI.

A *ground instance* of a constraint $c\theta$ is one in which $\mathrm{Var}(c\theta) = \varnothing$. Ground instances of equations ($\lambda\theta = \rho\theta$ if $c\theta$) are *true ground instances* if $c\theta \equiv \mathbf{T}$. Two constraints are *equivalent*, $c_1 \equiv c_2$, if for all θ such that $c_1\theta$ and $c_2\theta$ are ground instances, then $c_1\theta \equiv c_2\theta$. We write $c_1 \Rightarrow c_2$ if for all θ such that $c_1\theta$ and $c_2\theta$ are ground instances, then $c_1\theta \Rightarrow c_2\theta$. We will determine equivalence and implications using our *normalform* function of Section VI.

Constrained reductions are constrained equations which are ordered, written

$\lambda \rightarrow \rho$ if c, and all true ground instances of which must have $\lambda\theta > \rho\theta$. We apply a constrained reduction $\lambda \rightarrow \rho$ if c to a term $t \rightarrow t[i \leftarrow \rho\theta]$ if $t/i = \lambda\theta$ for some substitution θ, and $c\theta \equiv \mathbf{T}$.

Other definitions such as $\leftrightarrow$ and $\downarrow$ have the same meanings but is now defined relative to this new type of rewriting. The relation $=_R$ is defined as before without regard to the constraints in R. Note that this means that $=_R$ is not equivalent to $\leftrightarrow^*$, as before, because we must consider the constraints in $\leftrightarrow^*$ but not in $=_R$.

We again have the notions of confluence and complete set of reductions in our constrained form. A set of reductions is *complete* if $t =_R s$ if and only if $t\downarrow = s\downarrow$. Since constraints can only be guaranteed to have truth values when they are ground we will restrict ourselves to ground confluence. If for any two terms, s and t, $s \leftrightarrow^* t$ implies
$s \rightarrow^* u \ {}^*\!\!\leftarrow t$ for some term u, then the set of reductions is *ground confluent*. The condition $s \rightarrow^* u \ {}^*\!\!\leftarrow t$ is called the *joinability* condition and the process of showing joinability for all terms is how we will prove completeness. In [Pe90b], Peterson showed that, when showing completeness, working over ground terms is equivalent to working over all terms provided that we have 'enough' constants to work with.

Our constrained completion procedure will be equivalent to the unconstrained procedure when the constraints on the rewrite rules are always **T**. The notion of critical pairs (or critical equations [Pe90b]) is therefore similar. A *critical pair of the hard type* is formed between reductions $\lambda_1 \rightarrow \rho_1$ if c_1 and $\lambda_2 \rightarrow \rho_2$ if c_2 if $(\lambda_1/i)\sigma = \lambda_2\sigma$ for some substitution σ and $i \in \mathrm{sdom}(\lambda_1)$ and gives the critical pair:

$$\rho_1\sigma = \lambda_1[i \leftarrow \rho_2]\sigma \text{ if } c_1\sigma \wedge c_2\sigma.$$

Critical pairs of the easy type are formed from equations $\lambda \rightarrow \rho$ if c in R and are of the form:

$\lambda \rightarrow \rho$ if $\neg c$

An equation is *joinable* if all true ground instances are joinable. An equation $s = t$ is *connected below* u if $s \leftrightarrow t_1 \leftrightarrow \dots t_n \leftrightarrow t$ and $u > s,t,t_i$ for all i. A constrained equation, $s = t$ if c, is *subconnected* if for every true ground instance $s\theta = t\theta$, if $s\theta \ {}^{+}\!\!\leftarrow u \rightarrow^{+} t\theta$ then $s\theta = t\theta$ is connected below u.

Theorem 7.1 (Pe90b) A set of constrained reductions is complete if

(1) every critical pair (both hard and easy types) is joinable.

(2) every critical pair (both hard and easy types) is subconnected.

We will concern ourselves with the joinability test to prove completeness. Peterson included the subconnected test in hopes that we could find a method of excluding some critical pairs from consideration akin to that which can be done for KB-completion [ZK89] and [WB83].

An equation $s = t$ if c is joinable if one of the following conditions hold

(1) s and t are identical.

(2) $c =$ **F**.

(3) $s = t$ if c can be reduced into equations all of which are joinable.

To prove joinability through (3) we need to first state some theorems given in [Pe90b].

Theorem 7.2 (Pe90b) If $c \equiv c'$ then $s = t$ if c is joinable if and only if $s = t$ if c' is joinable.

Theorem 7.3 (Pe90b) The constrained equation $s = t$ if $c_1 \vee c_2$ is joinable if and only if $s = t$ if c_1 and $s = t$ if c_2 are joinable.

Theorem 7.4 (Pe90b) The constrained equation $s = t$ if $(t_1 = t_2) \wedge c$ is joinable if and only if t_1 and t_2 are not unifiable or $s\theta = t\theta$ if $c\theta$ is joinable where θ is the mgu of s and t.

The above theorems give justifications to some of the actions taken in the *normalform* procedure of Section VI.

Now we will state how we prove joinability through (3) (Theorem 5.5 in [Pe90b]). Given an equation $s = t$ if c then we rewrite it if there exists a reduction $\lambda \rightarrow \rho$ if c' which reduces $s = t$ if c to

(e1) $(s = t)[i \leftarrow \rho\sigma]$ if $c \wedge c'\sigma$, and

(e2) $s = t$ if $c \wedge \neg c'\sigma$

where $(s = t)/i = \lambda\sigma$ for some substitution σ, $i \in \mathrm{sdom}(s) \cup \mathrm{sdom}(t)$, and $c \wedge c'\sigma \not\equiv \mathbf{F}$. Note that if $c \Rightarrow c'\sigma$ then (e2) is trivially joinable since $c \wedge \neg c'\sigma \equiv \mathbf{F}$, also the constraint in (e1) becomes simply c.

Theorem 7.5 (Pe90b) If an equation e reduces to e_1 and e_2, then e is joinable if and only if e_1 and e_2 are joinable.

Peterson stated in his paper that we show completeness in the above way and that the constrained completion procedure was similar to KB-completion but he did not give the constrained completion procedure. He said that "some theory necessary to actually automatically prove completeness" was not present in his paper [Pe90b] and that he would later present this in a paper. He has since stated that he did not see himself doing this any time soon [Pe91a] although he has actually implemented the procedure. We hope to fill in this gap with the following presentation.

Presented in Figures 25 through 28 is the constrained completion procedure. The function

```
constr-complete (E)
/* E is the set of identities (with no constraints) */

for all $\lambda = \rho \in E$
    add-reduction( $\lambda = \rho$)
end for

for all $\lambda_1 \rightarrow \rho_1$ if $c_1 \in R$
    for all $\lambda_2 \rightarrow \rho_2$ if $c_2 \in R$
        for all $i \in$ sdom($\lambda_1$)
            if $\lambda_1/i$ and $\lambda_2$ unify with substitution $\sigma$ then
                joinable($\rho_2\sigma = \lambda_1[i \leftarrow \rho_2\sigma]$ if $c_1\sigma \wedge c_2\sigma$)
            end if
        end for
    end for
end for
end constr-complete
```

Figure 25. Constrained completion procedure.

add-reduction takes an equation and attempts to find the constraint needed to rewrite an equation $\lambda = \rho$ both ways. The constraint c_1 is the result of orienting $\lambda \rightarrow \rho$ and c_2 from orienting $\rho \rightarrow \lambda$. We must attempt to orient both ways since we may be given $x = x + 0$ and orienting $x \rightarrow x + 0$ yields a result of **F** and thus no rewrite rule would be added which means R will not accurately reflect the equational theory given in E. If we try $x + 0 \rightarrow x$ we get a constraint of **T** and R is correct. The constraints c_1 and c_2 are, in general, not merely the negation of one another. For example, the commutative law $f(x,y) = f(y,x)$ will give $c_1 = x > y$ and $c_2 = y > x$ but $\neg c_1 = (x = y) \vee (y > x)$ and $\neg c_2 = (x = y) \vee (x > y)$. Note,

```
add-reduction($\lambda = \rho$)

$c_1$ = normalform(factor(STP(F, T, $\lambda > \rho$, T)))
$c_2$ = normalform(factor(STP(F, T, $\rho > \lambda$, T)))

if $c_1 \not\equiv$ F then
    R = R $\cup$ {$\lambda \rightarrow \rho$ if $c_1$}
    Pairs = Pairs $\cup$ {$\lambda \rightarrow \rho$ if $\neg c_1$}
end if

if $c_2 \not\equiv$ F then
    R = R $\cup$ {$\rho \rightarrow \lambda$ if $c_2$}
    Pairs = Pairs $\cup$ {$\rho \rightarrow \lambda$ if $\neg c_2$}
end if
end add-reduction
```

Figure 26. Procedure to add a reduction.

however, for the commutative law we do not want to add both $f(x,y) \rightarrow f(y,x)$ if $x > y$ and $f(y,x) \rightarrow f(x,y)$ if $y > x$ since they are merely instances of one another. For the same reason, we would not want to add $f(x,0) \rightarrow f(0,x)$ if $x > 0$ in the presence of a commutative law. Thus we assume that when we add to R (and Pairs) we do not add equations which can be subsumed in this manner. An equation $\lambda_1 \rightarrow \rho_1$ if c_1 is *subsumed* by ($\lambda_2 \rightarrow \rho_2$ if c_2) if $(\lambda_1 \rightarrow \rho_1)\theta = \lambda_2 \rightarrow \rho_2$ for some substitution θ. Easy critical pairs are also added to Pairs in the procedure but if the original constraint is **T** then we need not add the critical pair to Pairs. Notice that this formulation of *add-reductions* will coincide with KB-completion since the constraints will be **T** and **F**, and the easy critical pairs (having a constraint of $\neg$**T**)

```
joinable(e = (s = t if c) )
if s and t are identical then
    return()

c = normalform(c)

if c = F then
    return()

If e reduces to $e_1$ and $e_2$ then
    /* the form of $e_1$ and $e_2$ are stated in ($e_1$) and ($e_2$) on page 57 */
    joinable($e_1$)
    if $c_2 \not\equiv$ F then
        joinable($e_2$)
    end if
else
    /* e is in normal form so we must add a new constrained reduction to R */
    add-reduction( $\lambda = \rho$ )
    constr-inter-reduce(R)
end if

end joinable
```

Figure 27.   Joinable procedure.

```
constr-inter-reduce(R)
/* R is a set of constrained reductions */
for e = $\lambda \rightarrow \rho$ if c $\in$ R
    for $\lambda_1 \rightarrow \rho_1$ if $c_1 \in$ R - {$\lambda \rightarrow \rho$ if c}
        for all i $\in$ sdom($\lambda$) $\cup$ sdom($\rho$)
            if ($\lambda \rightarrow \rho$)/i = $\lambda_1\sigma$, for some substitution $\sigma$ then
                if c $\Rightarrow c_1\sigma$ then
                    ($\lambda \rightarrow \rho$)[i $\leftarrow \rho\sigma$] if c
                end if
            end if
        end for
    end for
    if e was reduced then
        R = R - {$\lambda \rightarrow \rho$ if c}
        Pairs = Pairs - {p | p is a pair formed with e}
        if $\lambda' \neq \rho'$ then
            add-reduction($\lambda' = \rho'$)
        end if
    end if
end for
end constr-inter-reduce
```

Figure 28. Constrained inter-reduce procedure.

will trivially be found joinable.

The procedure *joinable* takes an equation and uses our constrained reduction rules to reduce it to an identity or an equation with a constraint of **F**. If this cannot be done then we must add the equation as a reduction using *add-reduction*. This procedure, when restricted to equations whose constraints are **T**, is equivalent to the process of putting the left- and right-hand sides into normal form, seeing if they are identical, and adding a reduction if not. This, again, shows us that when reduced to only equations which have true constraints our procedure is equivalent to the KB-completion procedure.

Our *constr-inter-reduce* procedure keeps the constrained reductions in a type of normal form. Peterson only defined normal forms for ground terms but we will define a *normal form for constrained reductions* as follows:

Given a reduction $r = \lambda \rightarrow \rho$ if c we let $r \twoheadrightarrow r'$, using $\lambda_1 \rightarrow \rho_1$ if c_1, if there exists an $i \in \text{sdom}(\lambda) \cup \text{sdom}(\rho)$ such that $(\lambda \rightarrow \rho)/i = \lambda_1\sigma$, for some substitution σ and if $c \Rightarrow c_1\sigma$, then $r' = (\lambda \rightarrow \rho)[i \leftarrow \rho\sigma]$ if c. A reduction r is in *normal form relative to R* if $\twoheadrightarrow$ cannot be applied to r by any reduction in R.

The procedure *constr-inter-reduce* produces normal forms for each r relative to R - {r} and re-orients the reduction r if necessary. To see why we generate a new constraint rather than keeping old constraints consider the following:

Given $r_1 = x + y \rightarrow x$ if **T**:

$(0 \bullet y) + (x \bullet 1) \rightarrow x$ if **T** $\twoheadrightarrow r_1$

$0 \bullet y \rightarrow x$ if **T**

but using *STP* we get $0 \bullet y \rightarrow x$ if $0 \bullet y > x$

Given $r_1 = 0 + x \rightarrow x$ if **T**:

$x + y \rightarrow 0 + y$ if $x > 0 \twoheadrightarrow r_1$

$x + y \rightarrow y$ if $x > 0$

but using *STP* we get $x + y \rightarrow y$ if **T**

Again, our inter-reduce procedure corresponds to KB-completion when only true constraints are generated. This is because our *constr-inter-reduce* procedure puts both sides into normal form and if changes were made we delete the old rule and insert the new. In *constr-inter-reduce* we are actually a little over-aggressive since we try to re-orient when only the right-hand side is rewritten in addition to when the left-hand side is rewritten which is not necessary in KB-completion.

Finally, we see that the overall procedure *constr-complete* performs the necessary checking of joinability for all critical pairs and we can now compare it to the KB-completion given in Figure 5. Remember that it is not enough that the <u>input</u> reductions have constraints **T** for the two procedures to give the same results; we must also have any reductions which are generated by our procedure result in constraints which are **T**.

We will check for $c_2 = c \wedge \neg c_1\sigma \not\equiv$ **F** when generating critical pairs, by seeing if *greater-than-normal*($c, c_1\sigma$) (shown in Figure 23) returns **F**. This way of checking for equivalence to **F** is not complete. We mentioned in Section VI why this is so but the consequence of this is simply that we will reduce e to e_1 and e_2 and it will still hold that e is reducible if and only if e_1 and e_2 are joinable but if we cannot show this we may end up not showing completeness when it really exists. We will check for $c \Rightarrow c'\sigma$, to see if we can eliminate e_2, by putting $c'\sigma$ into normal form and seeing if for every $s > t \in c'\sigma$, $c \wedge (s > t) \equiv c$. This procedure is shown in Figure 29.

```
**redundant** (c, $c_1$)
/* returns **T** if $c \Longrightarrow c_1$ and **F** otherwise */
$c_1$-conjuncts = {set of conjuncts in $c_1$}
while $c_1$-conjuncts $\neq \emptyset$
    $c_1$-conjuncts = $c_1$-conjuncts - current
    /* current = $s > t$ */
    for all $s_2 > t_2$ in c
        if $x = s_2$ AND t is a subterm of $t_2$ OR
          $t = t_2$ AND $s_2$ is a subterm of s then
          return(**T**)
        end if
    end for
end while
return(**F**)
end redundant
```

Figure 29. Redundant procedure.

We now present a correctness proof.

Theorem 7.6 Whenever the *constr-complete* procedure stops with success the final set of reductions R is complete.

Proof This is not a particularly deep result. Peterson showed that if all critical pairs are joinable then the set is complete (Theorem 7.1). We generate the critical pairs in the correct manner. Also, *joinable* procedure is correct by the following reasoning: First, the *normalform* procedure was shown to be valid using Theorems 7.2 through 7.4. Theorem 7.5 tells us that we prove joinability of e by showing joinability of e_1 and e_2 and therefore joinable and the entire procedure is correct. ■

The implementation of the *joinable* procedure, as we shall see in the next section, is important. Obviously we want to avoid proving joinability on as many equations as possible. Thus if we have two rewrite rules which can rewrite our equation and the first produces an e_1 and e_2 while the second produces an only e_1 then the second rewrite rule should be chosen. However, we do not want to check for a rewrite rule which only produces an e_1 for so long that the time saved, by avoiding a joinability proof for e_2, is lost. Thus we want to order the reductions in R so that we find the second type of rewrite rules quickly. These and other implementation details will be discussed in the next section.

For a look at using constraints in other logic programming environments and in completion see [KKR90].

VIII. HIPER AND HIPER-EXTENSION

This section describes our implementation of constrained completion. We wished to make our implementation as fast as possible and thus decided to try to modify what is probably the fastest completion procedure in existence, when applicable - a system called HIPER. HIPER stands for HIgh PERformance permutative completion and was developed by Jim Christian at the University of Texas-Austin [Ch89] (HIPER is available through ftp at rascal.ics.utexas.edu (128.83.144.1) in "pub/jimc"). We decided to use this system's many quick procedures and to add more functionality (the constraints) to produce our *HIPER-extension* system.

Since we built our system on top of HIPER we will first describe it and why it is so fast.

A. HIPER

HIPER is a completion procedure designed to find complete sets of reductions modulo an equational theory. HIPER gains some easy speed up by restricting itself to simple, linear permutative equational theories (described below) because Christian showed that when dealing with these theories confluence implies coherence (the properties which along with termination prove completeness). By restricting the theories like this Christian did not have to generate coherence pairs or show that they reduced to identities. The fact that HIPER does not generate coherence pairs does not hurt our implementation since we are not concerned with them either - indeed we have no separate equational theory to speak of.

Let #(p,t) be the number of occurrences of p in t, where $p \in \mathcal{V} \cup \mathcal{F}$. A term is *simple* if all non-root subterms are variables. A term t is *linear* if $\#(x,t) \leq 1$ for all $x \in \mathcal{V}$. A term s is a *permutation* of t if #(p,s) = #(p,t) for all $p \in \mathcal{V} \cup \mathcal{F}$. An equational theory E is a *simple, linear permutative theory* if for all s == t ∈ E, s is a simple, linear permutation of t. Note that although commutativity is a simple, linear permutation, associativity is not.

Christian also improved his speed by using 'flatterms' to store terms. A flatterm is a data structure with fields *symbol*, *next*, *prev*, and *end* so that each record contains a function symbol or variable name, pointers to the next and previous symbols, and a pointer to the end of the subterm which this symbol starts. Constants and variables have their end pointers pointing to themselves. Thus flatterms are doubly-linked lists with end pointers added. A flatterm is visualized in Figure 30.

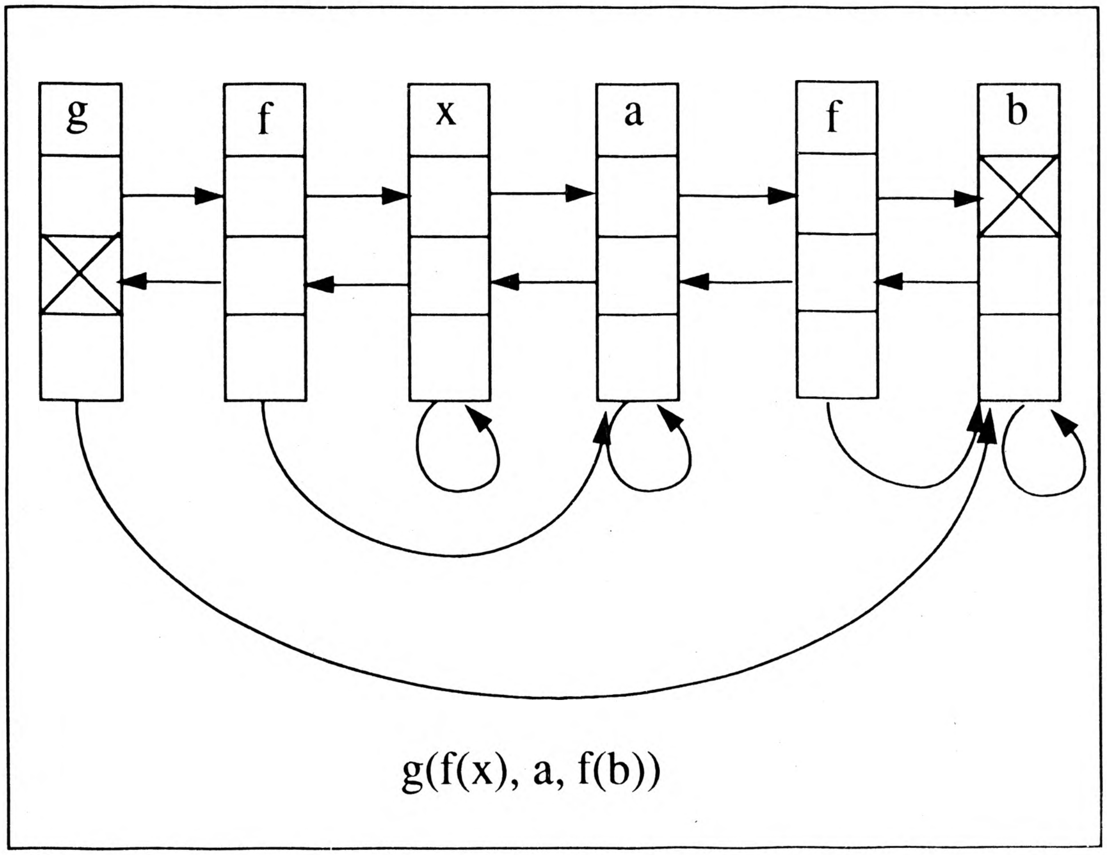

Figure 30. Flatterm representation.

The traditional a tree-like structures used to represent terms would not be as efficient in traversing terms in HIPER. Traversing is important since traversing forward is used when generating critical pairs while traversing backward is used in rewriting and copying.

Christian [Ch89] claims that the use of these flatterms causes a 25 to 30 percent speedup, although unification and matching procedures do not change their run times much.

The most time spent in completion procedures is, however, taken up in trying to find a rule which will reduce a given term (to get our normal forms) or in finding rules which are reduced by a given rule. To do these operations quickly Christian used discrimination nets to hold the left-hand sides of rules. These discrimination nets are tree-like structures which have, at each level, all function symbols listed plus a wild card spot, *, to stand for any variable and term structure can be determined by following a path through the net. See Figure 31.

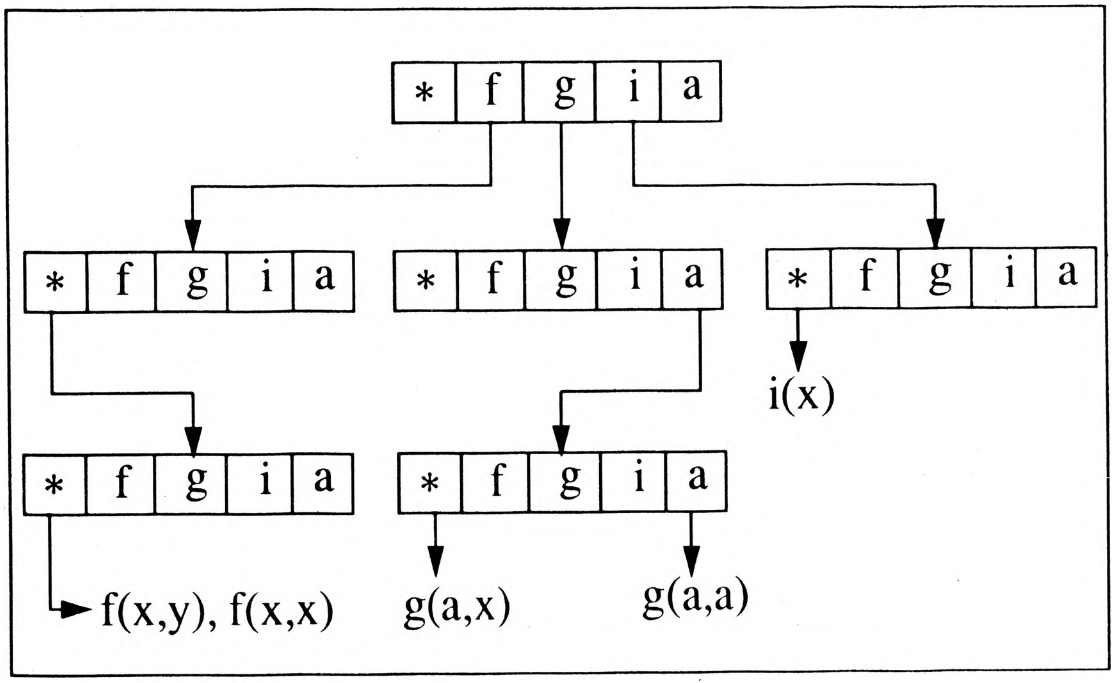

Figure 31. A Discrimination net.

By putting terms in this data structure we can check for structure compatibility quickly. We use wild cards for the variables because matching rarely fails due to variable binding inconsistency but, rather, fail often due to structure incompatibility. When variables are encountered at a level we keep ‘choicepoints’ so that if variable binding does fail we can go back to this choicepoint to find a different rule which matches the given subterm.

This process of searching for rules to reduce a given subterm takes about 20 percent of the overall time [Ch89] so it is important that we find this type of fast procedure.

Another procedure, which uses 10 to 20 percent of the total time [Ch89], is the searching for terms which can be reduced by a rule which has just been added. This is needed so that we can inter-reduce our reductions. To do this we need to add the right-hand side of terms into a net also since we want to keep both sides of the reductions reduced. This procedure generally takes more time since more choicepoints are needed because a new reduction’s variables will be able to make many matches.

Subsumption of rewrite rules is also performed using these discrimination nets.

Christian also tries to avoid garbage collecting by the LISP interpreter in his system. *Garbage collecting* is a process in LISP which periodically checks for values which are no longer referenced and collects them to have their memory re-used by the system later. He, in effect, does much of the garbage collecting himself and thus does not need to allocate memory for new structures very often.

Since LISP does not offer pointers or ‘C’-like arrays Christian also takes advantage of his knowledge of Austin Kyoto Common LISP (AKCL) to implement these data structures and thus speed up his implementation. When using other versions of Common LISP, though, these enhancements are left out.

Christian [Ch89] claims that discrimination nets are the key to the speed up he achieved. Using them has resulted in an increase of speed of 20 to 30 times.

B. HIPER-extension

We will now look at our implementation of constrained completion using HIPER. We first need a structure to represent our constraints in HIPER. To do this we modified Christian's existing 'eqn' data structure which took a left- and right-hand side (called 'lhs' and 'rhs' respectively) of an equation each of which had flatterms stored in them. To this we added an operator field in which we could put in **T**, **F**, =, >, $\wedge$, $\vee$, and $\neg$ to make up our constraints. We let our constraints and our reductions share the same data structure with the reductions using the added 'constraint' field

We also let the operator field take on values of 'lex', 'all', 'some', and 'lexsome' for use in the *STP* procedure. Since 'lexsome' takes five arguments we added 'term3', 'term4', and 'term5' to the structure and store the first and second arguments in 'lhs' and 'rhs', respectively.

We have also modified HIPER so that we can garbage collect our new equations. This must be done carefully since they may be deeply nested and end in flatterms which we also want to garbage collect. Also, code was deleted from HIPER which pertained to unfailing completion, different orderings, dynamic permutative unification, function symbol introduction, etc.

We gave the basic algorithms for the constrained completion procedure in Section VI and these were implemented in HIPER-extension. We said that it should correspond to KB-completion under certain conditions and thus we might wonder if the functionality we added when implementing this procedure caused the system to slow down when they should produce the same results. The table below shows that it degrades somewhat. We suspect this is because we did not always use the discrimination nets in our implementation and that time was spent while checking and generating constraints.

Table II. Comparison of HIPER and HIPER-extension.

Problem	HIPER-extension (seconds)	HIPER (seconds)
p1	0.2333	0.0667
lesc	0.1667	0.1667
kbinv	0.05	0.01667
kbgc1	0.1667	0.0333
kbcancel	0.08333	0.05

The above runs were made on a Sun SparcStation I running AKCL and are listed fully in Appendix A.

For a sample run of HIPER we will look at run on a associativity-commutativity problem:

```
> (constr-complete "actest2")
Declaring symbol */2 with weight 2


Input equation (* X0 X1) = (* X1 X0)                (commutativity)
2: (* X0 X1) --> (* X1 X0)                          IF X0 > X1
Input equation (* (* X0 X1) X2) = (* X0 (* X1 X2))  (associativity)
1: (* (* X0 X1) X2) --> (* X0 (* X1 X2))            IF TRUE


Pair
= (* X0 (* X1 X2)) (* X2 (* X0 X1))                 IF (* X0 X1) > X2 from 1 on 2
Pair
= (* (* X0 X1) X2) (* X1 (* X0 X2))                 IF X1 > X0 from 2 on 1
Pair
= (* (* X0 (* X1 X2)) X3) (* (* X0 X1) (* X2 X3))   IF TRUE from 1 on 1
```

```
Pair
= (* X0 (* X1 X2)) (* (* X0 X2) X1)                  IF X1 > (* X0 X2) from 3 on 2
Pair
= (* X0 (* X1 X2)) (* X2 (* X0 X1))
IF X0 > X1 AND X2 > X1 AND X0 > X2 from 2 on 3
Pair
= (* X0 (* (* X1 X2) X3)) (* X1 (* X2 (* X0 X3)))  IF (* X1 X2) > X0 from 3 on 1
Pair
= (* (* X0 (* X1 X2)) X3) (* X1 (* (* X0 X2) X3))  IF X1 > X0 from 3 on 1
Pair
= (* X0 (* X1 (* X2 X3))) (* (* X1 X2) (* X0 X3))  IF X0 > (* X1 X2) from 1 on 3
Pair
= (* X0 (* X1 (* X2 X3))) (* X2 (* X0 (* X1 X3)))
IF X0 > X1 AND X2 > X1 AND X0 > X2 from 3 on 3


 *** Completion terminated ***


Rules/Failures:


2: (* X0 X1) --> (* X1 X0)                           IF X0 > X1
1: (* (* X0 X1) X2) --> (* X0 (* X1 X2))             IF TRUE
3: (* X0 (* X2 X1)) --> (* X2 (* X0 X1))             IF X0 > X2


Run time: 4.5 seconds
 3 Equations retained
 9 Pairs generated
 17 Equations processed
```

For runs on other problems see Appendix A.

As was mentioned in Section VII, we found that our implementation of the *joinable* procedure was important in achieving a quick completion procedure. We want to avoid partially reducing an equation e into e_1 and e_2, instead we would like to reduce e into e_1. Remember that if $e = (s = t \text{ if } c)$ and $r = \lambda \rightarrow \rho$ if c_1 then e reduces to

$e_1 = (s = t)[i \leftarrow \rho\sigma]$ if c

if $(s = t)/i = \lambda\sigma$ and $c \Rightarrow c_1\sigma$

and partially reduces to

$e_1 = (s = t)[i \leftarrow \rho\sigma]$ if $c \wedge c_1\sigma$

$e_2 = s = t$ if $c \wedge \neg c_1\sigma$

if $(s = t)/i = \lambda\sigma$ and $c \wedge c_1\sigma \not\equiv$ **F**.

If we can find a reduction r_1 which non-partially reduces e then not only do we avoid proving joinability for two equations, we also keep the constraint in a simpler form. This is important since our checking for $c_1 \Rightarrow c_2$ and our putting new terms into normal form takes more time when the constraints are larger.

The most obvious way to check for reducibility is to simply check the reductions one at a time and reduce as soon as we can, whether the reduction is partial or not. If we do not order the reductions so that it is more likely to non-partially reduce than partially reduce then we can waste much time proving joinability. As an extreme example, a run which can be done in HIPER-extension in ~32 seconds with an ordering on the reductions to try small constraints (fewest conjuncts) first was tried with the inverse of this ordering - trying reductions with constraints with the largest number of conjuncts first. This run was aborted after ~15 minutes after generating 93 of the 124 critical pairs needed and seemed to be having trouble proving joinability at this point since no new pairs were generated for ~5 minutes when we aborted. This example is the ternary boolean algebra example and will be listed in Section IX.

The most obvious ordering to apply is to check reductions whose constraints are **T** first. This way we know that when these reductions are applied that $c \Rightarrow c_1\sigma$ (= $\mathbf{T}\sigma = \mathbf{T}$) and thus we do not partially reduce. Reductions without true constraints could be ordered so that the reduction with the least number of conjuncts is attempted first. This will help keep the size of the constraint small in most cases. We will call this strategy for proving joinability the Ordered First Find (OFF) strategy. See Figure 32 for its pseudo-code.

Although the OFF strategy will keep constraint size down in most cases it will not always produce a non-partial reduction. If, for example, the constraint c_1 has fewer conjuncts than c_2 but only c_2 is implied by the original conjunct we would use the reduction with constraint c_1, call it r_1, before the one with c_2, call it r_2, even though r_1 partially reduces while r_2 non-partially reduces. To avoid this we could try to check for reducibility and when we find a non-partial reduction we reduce immediately while if we find a partial reduction we 'shelve' it and try to find a non-partial reduction with the other rules. If we cannot find a non-partial reduction then we use the best partial reduction (least number of conjuncts) which partially reduces our equation e. If we implement this way we are assured of non-partially reducing when it is possible. We will call this strategy the Best Find (BF) strategy.

The OFF strategy cannot use the discrimination nets since we cannot order the matching done in them. This is disappointing since Christian said that the discrimination nets were important to his speed up. We will, however, use them in the BF method since it is not necessary to keep them ordered when we check the reductions for a non-partial reduction. Our hope in doing this is that use of the discrimination nets in BF will offset the

```
OFF-strategy(R, s = t if c)
/* The reductions in R are assumed to be ordered form smallest constraint to largest
   constraint so that when we loop over R we attempt the reductions in this order. */
c = normalform(c)

if s and t are identical then
    return ()

if c = F then
    return ()

for all $\lambda \rightarrow \rho$ if $c_1 \in$ R
    if $\lambda \rightarrow \rho$ if $c_1$ reduces s = t if c on the left-hand side then
        if $c \Rightarrow c_1\sigma$ then
            OFF(R, $e_1 = (s = t)[i \leftarrow \rho\sigma]$ if c)
            exit (OFF-strategy)
        else
            OFF(R, $e_1 = (s = t)[i \leftarrow \rho\sigma]$ if $c \wedge c'\sigma$)
            OFF(R, $e_2 = s = t$ if $c \wedge \neg c'\sigma$)
        end if
    else if $\lambda \rightarrow \rho$ if $c_1$ reduces s = t if c on the right-hand side then
        if $c \Rightarrow c_1\sigma$ then
            OFF(R, $e_1 = (s = t)[i \leftarrow \rho\sigma]$ if c)
            exit (OFF-strategy)
        else
            OFF(R, $e_1 = (s = t)[i \leftarrow \rho\sigma]$ if $c \wedge c'\sigma$)
            OFF(R, $e_2 = s = t$ if $c \wedge \neg c'\sigma$)
        end if
    end if
end for

add-reduction(s = t)
constr-inter-reduce(R)

end OFF-strategy
```

Figure 32. Ordered first find strategy.

non-ordering of reductions.

Something we must consider is: Given an equation e do we try each reduction first on the left-hand side of e and then on the right-hand side of e or do we try to reduce the left-hand side of e by all reductions and then the right-hand side of e by all reductions. Obviously if the reductions were ordered we would want to try each reduction on the left- and right-hand sides before trying the next reduction. We do this in OFF but not in BF. In BF we use the other method since the reductions are not ordered and the matching routines are not designed to handle this method.

We present the BF strategy in Figure 33.

C. RESULTS

1. OFF vs. BF

We will now compare the results of our two implementations and suggest improvements to the strategies. We first show a comparison of the OFF and BF strategies using a Sun SparcStation I running AKCL in the below table.

Table III. Comparison of OFF and BF strategies.

Problem	OFF (seconds)	BF (seconds)
p1	0.36	0.2333
lesc	0.1667	0.1667
kbinv	0.05	0.08333
kbgc1	0.1667	0.18333
kbcancel	0.08333	0.0667
ac	4.5	6.2333

```
BF-strategy(R, s = t if c)
/* The reductions in R are not in any particular order */
/* We will keep the best partial reduction in 'best-partial' */

c = normalform(c)
if s and t are identical then
    return ()
end if
if c = F then
    return ()
end if
for all $\lambda \rightarrow \rho$ if $c_1 \in$ R
    /* let the reduction we are working with be called 'current' */
    if $\lambda \rightarrow \rho$ if $c_1$ reduces s = t if c on the left-hand side then
        if $c \Rightarrow c_1\sigma$ then
            BF(R, $e_1 = (s = t)[i \leftarrow \rho\sigma]$ if c)
            exit (BF-strategy)
        else
            if current is a 'better' partial reduction than best-partial OR
              best-partial does not exist then
                best-partial = current
            end if
        end if
    end if
end for
```

Figure 33. a.) Best find strategy (continued).

```
for all $\lambda \rightarrow \rho$ if $c_1 \in R$

    /* let the reduction we are working with be called 'current' */

    if $\lambda \rightarrow \rho$ if $c_1$ reduces $s = t$ if $c$ on the right-hand side then

        if $c \Rightarrow c_1\sigma$ then

            BF(R, $e_1 = (s = t)[i \leftarrow \rho\sigma]$ if c)
            exit (BF-strategy)

        else

            if current is a 'better' partial reduction than best-partial OR
              best-partial does not exist then

                best-partial = current

            end if

        end if

    end if

end for


if best-partial exists then

    /* prove joinability using this partial reduction */

    BF(R, $e_1 = (s = t)[i \leftarrow \rho\sigma]$ if $c \wedge c'\sigma$)
    BF(R, $e_2 = s = t$ if $c \wedge \neg c'\sigma$)
    exit(BF-strategy)

end if

/* The equation was not joinable */

add-reduction($s = t$)

constr-inter-reduce(R)

end BF-strategy
```

Figure 33 (continued). b.) Best find strategy.

The above problems are fully presented in Appendix A.

As we can see the results are actually disappointing for our BF method. We expected it to outperform OFF since it does not partially reduce as much and because we were using discrimination nets. We suspect that the lack of ordering on the terms presented the greatest degradation since it causes us to attempt too many reductions than in OFF. Also, the fact that we try all reductions on the left-hand side before trying any on the right-hand side also contributes to the wasted time.

Two modifications can be thought of which could make the completion procedure run faster. First, we can try our BF strategy without the use of discrimination nets. This way we can order our reductions when trying to find non-partial reductions so that we find the non-partial reductions quicker.

The other modification we can make is to have some sort of ordering when using discrimination nets. To do this though we will need multiple nets with each net only containing the left-hand side rewrite rules whose constraints have a given length. we probably need to only have four such nets: a **T** constraint net and nets for constraints with 1, 2 and ≥ 3 conjuncts (not many naturally occurring rewrite rules have constraints with more than three conjuncts).

This implementation must be carefully done since it will take up more space and since we do not want to search multiple nets often in order to find a reducing rule. The implementation of this in HIPER will thus take some time since we would be modifying the heart of the system and do not want to introduce inefficiencies.

2. Constrained reductions vs. other methods

Earlier, we gave a comparison of HIPER and HIPER-extension when the procedures produced the same results but we will now compare the methods when HIPER-extension can use its constraints to its advantage and HIPER can use its dynamic unification algorithms and function symbol introduction.

Before we do this let us look at the original set of axioms which prompted Peterson to look at constrained completion - ACI completion. Here is the run from HIPER-extension used to find a complete set of reductions for this theory.

```
> (constr-complete "aci")
Declaring symbol */2 with weight 2
Declaring symbol ZERO/0 with weight 1


Input equation
(* X0 X1) > (* X1 X0)


Input equation
(* (* X0 X1) X2) > (* X0 (* X1 X2))


Input equation
(* (ZERO) X0) > X0


1: (* (ZERO) X0) --> X0                         IF TRUE
2: (* (* X0 X1) X2) --> (* X0 (* X1 X2))        IF TRUE
3: (* X0 X1) --> (* X1 X0)                      IF > X0 X1
.
.
.
 *** Completion terminated ***
```

```
Rules/Failures:

4: (* X0 (ZERO)) --> X0                         IF TRUE
1: (* (ZERO) X0) --> X0                         IF TRUE
2: (* (* X0 X1) X2) --> (* X0 (* X1 X2))        IF TRUE
5: (* X0 (* X2 X1)) --> (* X2 (* X0 X1))        IF > X0 X2
3: (* X0 X1) --> (* X1 X0)                      IF > X0 X1

Run time: 7.35 seconds
 5 Equations retained
 18 Pairs generated
 28 Equations processed
```

We have thus provided a complete set of reductions for this theory without even producing the unification algorithm which the original completion procedure required.

Let us now compare our completion procedure to HIPER when the two do not produce the same results: in cases such as the "if" problem (listed in Appendix A) we produce a much smaller set of reductions, ten compared to thirty-five, and do not require function symbol introduction to be used. In other problems such as "ct" and "aci" (again, listed in Appendix A) we produce roughly the same number of reductions but HIPER requires a unification algorithm to be generated for some of the axioms. HIPER was consistently faster in these runs. Our method, however, is more general since we can never fail due to an un-orientable equation.

IX. TERNARY BOOLEAN ALGEBRA - UNIFICATION AND COMPLETION

Part of the author's initial motivation for looking at constrained completion was to find a complete set of reductions for ternary boolean algebras. This problem was presented in [Wo88] as Test problem 14. Although its applicability in the real world might be questionable, Wos suspected that it was a very difficult problem to solve and figured that the attempt of solving this problem would help advance the field of automated reasoning.

Ternary boolean algebra was first presented by Grau in [Gr47]. A ternary boolean algebra is a non-empty set satisfying the following five axioms:

(1) $h(u,v,h(x,y,z)) = h(h(u,v,x),y,h(u,v,z))$.

(2) $h(y,x,x) = x$

(3) $h(x,y,i(y)) = x$

(4) $h(x,x,y) = x$

(5) $h(i(y),y,x) = x$

It has been shown that axioms 4 and 5 are dependent on the other axioms while each of axioms 1, 2, and 3 are independent of the rest [WW82]. Thus to find a complete set of reductions for these axioms we need only give the first three as input.

Christian [Ch89] attacked this problem with his HIPER system but was not able to come up with a complete set of reductions. HIPER ran for a long time on this problem and Christian noted that many permuters were found in his runs and hypothesized that a unification algorithm encompassing the following equational theory was needed to find the complete set of reductions:

(1) $h(x,y,z) = h(x,z,y)$

(2) $h(x,y,z) = h(y,x,z)$

(3) h(x,y,z) = h(y,z,x)

(4) h(x,y,z) = h(z,x,y)

(5) h(x,y,z) = h(z,y,x)

(6) h(x,y,h(z,y,w)) = h(h(x,y,z),y,w)

Identities 1 through 5 tells us that h is completely commutative and identity 6 is an associative law. These identities are actually proved in [Gr47].

Such a 'ternary boolean algebra' or 'ternary associative-commutative' unification algorithm was created in [WMJ90] and will now be presented.

A. TERNARY BOOLEAN ALGEBRA UNIFICATION

Formally stated the ternary boolean algebra unification algorithm needs to produce a complete set of unifiers Γ for the unification problem $<s,t>_{A+CC}$ such that if $\sigma \in \Gamma$ then $\sigma(s) =_{A+CC} \sigma(t)$ where A+CC is the equational theory for ternary associativity (A) and ternary complete commutativity (CC). To solve this problem we will produce the two sets $\{s' \mid s' =_{A'} s\}$ and $\{t' \mid t' =_{A+CC} t\}$ and find the set of all Robinson unifiers from the cross product of these sets. The theory A′ is the theory of associative laws which allows the bridge element ('y' in identity 6 above) to be in any term - not just the middle term since h is completely commutative. This would give us 36 (3!3!) different associative laws. We will collectively call these associative laws the *cc-associative laws*.

To produce this set of unifiers we must first produce the set $\{s' \mid s' =_{A'} s\}$ and $\{t' \mid t' =_{A+CC} t\}$. Producing these sets is more difficult than it might seem at first glance.

To apply the associative law to a term h(x,y,h(z,y,w)) we need to have the *bridge element* y to produce the *combinators* h(z,y,h(x,y,w)) and h(w,y,h(z,y,x)). It becomes more difficult to produce these combinators when many bridges occur such as in h(x,h(x,y,z),h(x,w,h(x,y,z))) since the bridges are nested and cycling can occur. This term produces sixty-three combinators while h(h(x,y, h(z,w,y)),x,h(h(x,y,h(z,w,y)),z,w)) produces 1062 combinators!

These combinators will produce our set $\{t' \mid t' =_{A'} t\}$.

To produce the combinators we sort our terms so that the subterms are ordered with the deeper nested subterms put on the right. This means that the bridge element always occurs to the left of the subterm in which it appears.

We need two functions to produce the combinators. First, the function *depth* will return the nesting depths of a term (e.g. *depth*(h(x,y,z)) = 0 and *depth*(h(x,h(x,y,z),h(u,v,w))) = 1). Second, the function *swap-at-depth* finds all combinators at a given depth. The function to produce the combinators, *tercomb*, can now be shown in Figure 34.

```
Tercomb(t)
[1]   toprocess := {t};
[2]   processed := ∅;
[3]   while toprocess ≠ ∅ do
[4]     term := Pop(toprocess);
[5]     temp := ∅;
[6]     if Depth(term) ≠ 0 then
[7]       for i := 1 to Depth(term) do
[8]         temp := temp ∪cc Swap-at-depth(i,term);
[9]     processed := processed ∪cc {term};
[10]    toprocess := toprocess ∪cc {temp -cc processed};
[11]  return (processed);
```

Figure 34. Algorithm to find combinators.

In the *tercomb* algorithm, $\cup_{CC}$ means that we do not add terms which are $=_{CC}$ to already found combinators. The function $-_{CC}$ takes out terms which are $=_{CC}$ to the stated term. This algorithm's correctness is given in Appendix B.

Now to produce our set $\{t' \mid t' =_{A+CC} t\}$ we produce it as $\{t' \mid t' =_{CC} t''$ where $t'' \in$ *tercomb*(t)$\}$. Producing $\{t' \mid t' =_{CC} t\}$ is relatively simple: we simply recursively permute all subterms in s.

Let $| t |_h$ denote the number of occurrences of h in t. Let *allperms*(t) produce the set $\{t' \mid t' =_{CC} t\}$ and R-unify produce Robinson mgus. Now we give the ternary A+CC-unification algorithm in Figure 35. Remember that the method is to find all mgus from $\{s' \mid s' =_{CC} s\}$ and $\{t' \mid t' =_{A+CC} t\}$.

```
TBA-unify(s,t)
if $| s |_h > | t |_h$ then swap(s,t)
$list_1$ = nil
for all x in tercomb(s)
    $list_1 = list_1 \cup$ allperms(s)
$list_2$ = tercomb(t)
unifiers = nil
for all x in $list_2$
    for all y in $list_1$
        unifiers = unifiers $\cup$ R-unify(x,y)
return (unifiers)
```

Figure 35. Ternary A+CC unification algorithm.

This algorithm is certainly not minimal but we believe it is complete, yet no proof of this exists. In Appendix B we present work towards this completeness proof. Namely, we show the completeness and correctness of an algorithm to produce a set of unifiers under complete commutativity.

B. TERNARY BOOLEAN ALGEBRA COMPLETION

This ternary A+CC unification algorithm was implemented to work both inside and outside of HIPER but our results were disappointing since no complete set of reductions was found using it. These attempts have not met with success due to the large computing time necessary to produce the combinators. An algorithm which produces a closer to minimal set of unifiers is needed to make another attempt. Also, the attempts in HIPER were early attempts and a better understanding of the system may make another attempt worthwhile.

Since we could not find a complete set of reductions using HIPER we might wonder if we can do so in HIPER-extension (i.e. using constraints). Indeed we can, but not in a straight-forward way.

Peterson presented the first such set (that we know of) in his paper [Pe90b]. To create the set however he first produced a complete set of reductions for boolean algebras which is listed below with '$\wedge$' as *and*, and '$\oplus$' as *exclusive or*.

$(x \wedge y) \wedge y \rightarrow x \wedge (y \wedge z)$

$x \wedge y \rightarrow y \wedge x$ if $x > y$

$x \wedge (y \wedge z) \rightarrow y \wedge (x \wedge z)$ if $x > y$

$x \wedge x \rightarrow x$

$x \wedge (x \wedge y) \rightarrow x \wedge y$

$(x \oplus y) \oplus z \rightarrow x \oplus (y \oplus z)$

$x \oplus y \rightarrow y \oplus x$ if $x > y$

$x \oplus (y \oplus z) \rightarrow y \oplus (x \oplus z)$ if $x > y$

$x \oplus x \rightarrow \mathbf{F}$

$x \oplus (x \oplus y) \rightarrow y$

$x \wedge \mathbf{T} \rightarrow x$

$\mathbf{T} \wedge x \rightarrow x$

$x \oplus \mathbf{F} \rightarrow x$

$\mathbf{F} \oplus x \rightarrow x$

$x \wedge \mathbf{F} \rightarrow \mathbf{F}$

$\mathbf{F} \wedge x \rightarrow \mathbf{F}$

$x \wedge (y \oplus z) \rightarrow (x \wedge y) \oplus (x \wedge z)$

$(x \oplus y) \wedge z \rightarrow (x \wedge z) \oplus (y \wedge z)$

Peterson then added the following two axioms:

$h(x,y,z) \rightarrow (x \wedge y) \oplus (y \wedge z) \oplus (z \wedge y)$

$i(x) \rightarrow \mathbf{T} \oplus x$

These twenty axioms form a complete set of reductions and since the last two axioms present the only realization of a ternary boolean algebra in a boolean algebra (Theorem III in [Gr47]) it forms a complete set of reductions for a ternary boolean algebra.

Below is a run in HIPER-extension showing that this set is complete.

```
> (constr-complete "tba")
Declaring symbol */2 with weight 3
Declaring symbol +/2 with weight 2
Declaring symbol ZERO/0 with weight 0
Declaring symbol ONE/0 with weight 1
Declaring symbol -/1 with weight 4
Declaring symbol F/3 with weight 5
```

11: (+ X0 (+ X0 X1)) --> X1	IF TRUE
16: (* X0 (* X0 X1)) --> (* X0 X1)	IF TRUE
17: (* X0 X0) --> X0	IF TRUE
12: (+ X0 X0) --> (ZERO)	IF TRUE
10: (* X0 (ONE)) --> X0	IF TRUE
9: (* (ONE) X0) --> X0	IF TRUE
8: (+ X0 (ZERO)) --> X0	IF TRUE
7: (+ (ZERO) X0) --> X0	IF TRUE
6: (* X0 (ZERO)) --> (ZERO)	IF TRUE
5: (* (ZERO) X0) --> (ZERO)	IF TRUE
20: (* (* X0 X1) X2) --> (* X0 (* X1 X2))	IF TRUE
15: (+ (+ X0 X1) X2) --> (+ X0 (+ X1 X2))	IF TRUE
4: (* X0 (+ X1 X2)) --> (+ (* X0 X1) (* X0 X2))	IF TRUE
3: (* (+ X0 X1) X2) --> (+ (* X0 X2) (* X1 X2))	IF TRUE
1: (- X0) --> (+ (ONE) X0)	IF TRUE
2: (F X0 X1 X2) --> (+ (* X0 X1) (+ (* X1 X2) (* X2 X0)))	IF TRUE
18: (* X0 (* X1 X2)) --> (* X1 (* X0 X2))	IF X0 > X1
13: (+ X0 (+ X1 X2)) --> (+ X1 (+ X0 X2))	IF X0 > X1
19: (* X0 X1) --> (* X1 X0)	IF X0 > X1
14: (+ X0 X1) --> (+ X1 X0)	IF X0 > X1

Pair

= (+ X0 X1) (+ X0 X1)　　　“IF “ TRUE from 11 on 11

Pair

= (* X0 (* X0 X1)) (* X0 (* X0 X1))　　　“IF “ TRUE from 16 on 16

```
Pair
= (* X0 X0) (* X0 X0)                    "IF " TRUE from 17 on 16
Pair
= (+ X0 (ZERO)) XO                       "IF " TRUE from 12 on 11
.
.
.
Pair
= (+ (+ X0 X1) X2) (+ X0 (+ X2 X1))      "IF " X2 > (+ X0 X1) from 14 on 13


Pair
= (+ X0 (+ X1 X2)) (+ X2 (+ X0 X1))
"IF " X0 > X1 AND X2 > X1 AND X0 > X2 from 14 on 13


 *** Completion terminated ***


Rules/Failures:


11: (+ X0 (+ X0 X1)) --> X1                         IF TRUE
16: (* X0 (* X0 X1)) --> (* X0 X1)                  IF TRUE
5: (* (ZERO) X0) --> (ZERO)                         IF TRUE
6: (* X0 (ZERO)) --> (ZERO)                         IF TRUE
7: (+ (ZERO) X0) --> X0                             IF TRUE
8: (+ X0 (ZERO)) --> X0                             IF TRUE
9: (* (ONE) X0) --> X0                              IF TRUE
10: (* X0 (ONE)) --> X0                             IF TRUE
12: (+ X0 X0) --> (ZERO)                            IF TRUE
```

```
17: (* X0 X0) --> X0 IF TRUE
15: (+ (+ X0 X1) X2) --> (+ X0 (+ X1 X2))                    IF TRUE
20: (* (* X0 X1) X2) --> (* X0 (* X1 X2))                    IF TRUE
3: (* (+ X0 X1) X2) --> (+ (* X0 X2) (* X1 X2))              IF TRUE
4: (* X0 (+ X1 X2)) --> (+ (* X0 X1) (* X0 X2))              IF TRUE
1: (- X0) --> (+ (ONE) X0)                                   IF TRUE
2: (F X0 X1 X2) --> (+ (* X0 X1) (+ (* X1 X2) (* X2 X0)))    IF TRUE
13: (+ X0 (+ X1 X2)) --> (+ X1 (+ X0 X2))                    IF > X0 X1
18: (* X0 (* X1 X2)) --> (* X1 (* X0 X2))                    IF > X0 X1
14: (+ X0 X1) --> (+ X1 X0)                                  IF > X0 X1
19: (* X0 X1) --> (* X1 X0)                                  IF > X0 X1

Run time: 33.78333 seconds
 20 Equations retained
 124 Pairs generated
 159 Equations processed
```

Although we have thus found a complete set of reductions for ternary boolean algebra this does not mean that finding the set using HIPER and the unification algorithm is a useless cause. Again, the research and not the result is what is more important. Thus, an interesting research problem still remains: find a complete set of reductions using the three axioms for ternary boolean algebras rather than Peterson's roundabout method.

X. CONCLUSIONS

This paper has accomplished several goals along the way to developing an efficient constrained completion system.

First, we presented standard completion methods and have seen their inadequacies when dealing with certain types of problems which has lead us to constrained completion.

In this paper we present the constrained completion procedure which was previously left out of the literature. In so doing we reviewed how we produce constraints for the lexicographic path ordering and discuss how we should keep our constrained reductions in their simplest forms. Also, we discussed the joinable procedure and its importance for efficiency considerations. Alternatives to implementing the procedure were discussed and two were implemented.

We looked at the HIPER system and our modification, HIPER-extension, which implements constrained completion. Results are shown comparing it to HIPER and comparing it against itself with the two main strategies we have discussed. Some complete sets of constrained reductions are given in the text and the appendices.

Finally, we presented our attack at finding a complete set of reductions for ternary boolean algebras - including a specialized unification algorithm. A complete set is given but is not derived directly from the ternary boolean algebra axioms.

Several interesting problems still exist:

- We need to implement a quicker joinable procedure so that we can attack the harder problems better.

- Also a criterion for eliminating unnecessary critical pairs would help speed up our completion proofs. This has already been done for KB-completion and needs to be extended for constrained completion.

- Currently we only have one algorithm to generate our constraints and it is based on the lexicographic path ordering. We need to try to implement other orderings so that we can find a greater number of complete sets.

- A theorem prover using the constrained completion procedure should prove interesting and should be considered.

- A complete set of reductions using only Grau's original axioms for a ternary boolean algebra still does not exist. The attempt at finding one should increase our knowledge about completion procedures.

APPENDIX A

SOME RUNS IN HIPER-EXTENSION

```
*************************************************************
Problem "p1.comp" given with the HIPER system.
*************************************************************

    > (constr-complete "p1")
    Declaring symbol E/0 with weight 1
    Declaring symbol H/2 with weight 3
    Declaring symbol G/2 with weight 2
    Declaring symbol F/2 with weight 4


    Input equation
    > (F X0 (G X0 X1)) X1
    Input equation
    > (F (H X0 X1) X1) X0
    Input equation
    > (G X0 (F X0 X1)) X1
    Input equation
    > (H (F X0 X1) X1) X0
    Input equation
    > (F X0 (E)) X0
    Input equation
    > (F (E) X0) X0


    6: (F X0 (G X0 X1)) --> X1 IF TRUE
    5: (F (H X0 X1) X1) --> X0 IF TRUE
    4: (G X0 (F X0 X1)) --> X1 IF TRUE
    3: (H (F X0 X1) X1) --> X0 IF TRUE
    2: (F X0 (E)) --> X0 IF TRUE
```

```
1: (F (E) X0) --> X0 IF TRUE


Pair

= (G (H X0 X1) X0) X1 "IF " TRUE from 5 on 4

Pair

= (F X0 X1) (F X0 X1) "IF " TRUE from 4 on 6

Pair

= (G X0 X1) (G X0 X1) "IF " TRUE from 6 on 4

Pair

= (F (E) X0) X0 "IF " TRUE from 11 on 5

Pair

= (E) (E) "IF " TRUE from 11 on 9


 *** Completion terminated ***


Rules/Failures:


8: (H X0 (G X1 X0)) --> X1 IF TRUE

3: (H (F X0 X1) X1) --> X0 IF TRUE

7: (G (H X0 X1) X0) --> X1 IF TRUE

4: (G X0 (F X0 X1)) --> X1 IF TRUE

5: (F (H X0 X1) X1) --> X0 IF TRUE

6: (F X0 (G X0 X1)) --> X1 IF TRUE

11: (H X0 X0) --> (E) IF TRUE

12: (G (E) X0) --> X0 IF TRUE

1: (F (E) X0) --> X0 IF TRUE

9: (H X0 (E)) --> X0 IF TRUE
```

```
10: (G X0 X0) --> (E) IF TRUE

2: (F X0 (E)) --> X0 IF TRUE


Unprocessed pairs:


Run time: 0.3666667 seconds

 12 Equations retained

 31 Pairs generated

 45 Equations processed
```

****

Easy problem given with the HIPER system ("lesc.comp")

****

```
> (constr-complete "lesc")

Declaring symbol F/1 with weight 1

Declaring symbol H/1 with weight 2

Declaring symbol G/1 with weight 3


Input equation

> (F (G (H X0))) (F (H X0))

Input equation

> (G (H (H X0))) (G (G (H X0)))


2: (F (G (H X0))) --> (F (H X0)) IF TRUE

1: (G (G (H X0))) --> (G (H (H X0))) IF TRUE


 *** Completion terminated ***
```

```
Rules/Failures:


2: (F (G (H X0))) --> (F (H X0)) IF TRUE

1: (G (G (H X0))) --> (G (H (H X0))) IF TRUE


Unprocessed pairs:


Run time: 0.016667 seconds

 2 Equations retained

 0 Pairs generated

 3 Equations processed
```

****

Boyer-Moore axiomatization of if-then-else

- This system needs function symbol introduction to complete the set in HIPER.

****

```
> (constr-complete "if")

Declaring symbol T/0 with weight 1

Declaring symbol F/0 with weight 2

Declaring symbol IF/3 with weight 3


Input equation

> (IF X0 X0 X1) (IF X0 (T) X1)

Input equation

> (IF X0 X1 X0) (IF X0 X1 (F))

Input equation

> (IF X0 X0 (F)) (IF X0 (T) X0)
```

```
Input equation

> (IF X0 X0 X0) (IF X0 (T) (F))

Input equation

> (IF (T) X0 X1) X0

Input equation

> (IF (F) X0 X1) X1

Input equation

> (IF X0 (IF X0 X1 X2) X3) (IF X0 X1 X3)

Input equation

> (IF X0 X1 (IF X0 X2 X3)) (IF X0 X1 X3)


2: (IF X0 (IF X0 X1 X2) X3) --> (IF X0 X1 X3) IF TRUE

1: (IF X0 X1 (IF X0 X2 X3)) --> (IF X0 X1 X3) IF TRUE

4: (IF (T) X0 X1) --> X0 IF TRUE

3: (IF (F) X0 X1) --> X1 IF TRUE

10: (IF X0 (T) X1) --> (IF X0 X0 X1) IF > (T) X0

9: (IF X0 X0 X1) --> (IF X0 (T) X1) IF > X0 (T)

8: (IF X0 X1 (F)) --> (IF X0 X1 X0) IF > (F) X0

7: (IF X0 X1 X0) --> (IF X0 X1 (F)) IF > X0 (F)

6: (IF X0 (T) X0) --> (IF X0 X0 (F)) IF > (T) X0

5: (IF X0 X0 X0) --> (IF X0 (T) (F)) IF > X0 (T)


Pair

= (IF X0 (IF X0 X1 X2) X3) (IF X0 (IF X0 X1 X4) X3) "IF " TRUE from 2 on 2

Pair

= (IF X0 X1 (IF X0 X2 X3)) (IF X0 X1 (IF X0 X4 X3)) "IF " TRUE from 1 on 1
```

```
Pair
= (IF X0 (IF X0 X1 X2) X3) (IF X0 X1 (IF X0 X4 X3)) "IF " TRUE from 1 on 2
Pair
= (IF X0 (IF X0 X1 X2) X3) (IF X0 X1 X3) "IF " TRUE from 1 on 2
.
.
.
Pair
= (IF X0 (T) (F)) (IF X0 (T) X0) "IF " > X0 (T) from 5 on 9
Pair
= (IF (T) (T) (F)) (IF (T) (T) (T)) "IF " FALSE from 5 on 10

 *** Completion terminated ***

Rules/Failures:

1: (IF X0 X1 (IF X0 X2 X3)) --> (IF X0 X1 X3) IF TRUE
2: (IF X0 (IF X0 X1 X2) X3) --> (IF X0 X1 X3) IF TRUE
3: (IF (F) X0 X1) --> X1 IF TRUE
4: (IF (T) X0 X1) --> X0 IF TRUE
5: (IF X0 X0 X0) --> (IF X0 (T) (F)) IF > X0 (T)
6: (IF X0 (T) X0) --> (IF X0 X0 (F)) IF > (T) X0
7: (IF X0 X1 X0) --> (IF X0 X1 (F)) IF > X0 (F)
8: (IF X0 X1 (F)) --> (IF X0 X1 X0) IF > (F) X0
9: (IF X0 X0 X1) --> (IF X0 (T) X1) IF > X0 (T)
10: (IF X0 (T) X1) --> (IF X0 X0 X1) IF > (T) X0
```

```
Run time: 3.35 seconds
 10 Equations retained
 56 Pairs generated
 86 Equations processed
```

****

Commutativity-Transitivity example

****

```
> (constr-complete "ct")
Declaring symbol R/2 with weight 2
Declaring symbol */2 with weight 1


Input equation
> (* (R X0 X1) (R X1 X2)) (* (R X0 X1) (R X0 X2))
Input equation
> (R X0 X1) (R X1 X0)
Input equation
> (* (R X0 X1) (R X0 X2)) (* (R X0 X1) (R X2 X1))
Input equation
> (* (R X0 X1) (R X1 X2)) (* (R X0 X1) (R X2 X1))


5: (* (R X0 X1) (R X0 X2)) --> (* (R X0 X1) (R X1 X2)) IF > X0 X1
4: (* (R X0 X1) (R X1 X2)) --> (* (R X0 X1) (R X0 X2)) IF > X1 X0
2: (* (R X0 X1) (R X2 X1)) --> (* (R X0 X1) (R X1 X2)) IF > X2 X1
1: (* (R X0 X1) (R X1 X2)) --> (* (R X0 X1) (R X2 X1)) IF > X1 X2
3: (R X0 X1) --> (R X1 X0) IF > X0 X1
```

```
Pair
= (* (R X0 (R X1 X0)) (R X1 X0)) (* (R X0 (R X1 X0)) (R X1 X0)) "IF " TRUE
from 7 on 6
Pair
= (* (R X0 X0) (R X1 X0)) (* (R X0 X0) (R X1 X0)) "IF " FALSE from 7 on 8
Pair
= (* (R X0 (R X0 X0)) (R (R X0 X0) (R X0 X0))) (* (R X0 (R X0 X0)) (R X0 X0))
"IF " FALSE from 5 on 6
.
.
.
Rules/Failures:

6: (* (R X0 (R X2 X0)) (R X2 (R X2 X0))) --> (* (R X0 (R X2 X0)) (R X2 X0)) IF
TRUE
1: (* (R X0 X1) (R X1 X2)) --> (* (R X0 X1) (R X2 X1)) IF > X1 X2
2: (* (R X0 X1) (R X2 X1)) --> (* (R X0 X1) (R X1 X2)) IF > X2 X1
4: (* (R X0 X1) (R X1 X2)) --> (* (R X0 X1) (R X0 X2)) IF > X1 X0
5: (* (R X0 X1) (R X0 X2)) --> (* (R X0 X1) (R X1 X2)) IF > X0 X1
7: (* (R X0 X1) (R X2 X1)) --> (* (R X0 X1) (R X2 X0)) IF > X1 X0
8: (* (R X0 X1) (R X2 X0)) --> (* (R X0 X1) (R X2 X1)) IF > X0 X1
3: (R X0 X1) --> (R X1 X0) IF > X0 X1

Run time: 13.56667 seconds
 8 Equations retained
 34 Pairs generated
 76 Equations processed
```

****

Inverse property problem (Example 4) in [KB70].

****

```
>(constr-complete "kbinv")

Declaring symbol */2 with weight 2

Declaring symbol -/1 with weight 1


Input equation

> (* (- X0) (* X0 X1)) X1


1: (* (- X0) (* X0 X1)) --> X1 IF TRUE


Pair

= (* (- (- X0)) X1) (* X0 X1) "IF " TRUE from 1 on 1

Pair

= (* X0 (* (- X0) X1)) X1 "IF " TRUE from 2 on 1

Pair

= (* (- (- (- X0))) (* X0 X1)) X1 "IF " TRUE from 2 on 1

Pair

= (* X0 X1) (* (- (- X0)) X1) "IF " TRUE from 3 on 3

Pair

= (* (- X0) X1) (* (- X0) X1) "IF " TRUE from 3 on 1

Pair

= (* X0 X1) (* X0 X1) "IF " TRUE from 1 on 3

Pair

= X0 (* X1 (* (- (- (- X1))) X0)) "IF " TRUE from 3 on 2
```

```
Pair
= (* (- X0) (* X0 X1)) X1 “IF “ TRUE from 2 on 3


 *** Completion terminated ***


Rules/Failures:


3: (* X0 (* (- X0) X1)) --> X1 IF TRUE
1: (* (- X0) (* X0 X1)) --> X1 IF TRUE
2: (* (- (- X0)) X1) --> (* X0 X1) IF TRUE


Unprocessed pairs:


Run time: 0.05 seconds
 3 Equations retained
 8 Pairs generated
 11 Equations processed
```

****

Knuth-Bendix [KB70] commutative group problem.

****

```
> (constr-complete “kbcg1”)
Declaring symbol */2 with weight 2


Input equation
> (* (* X0 X1) (* X1 X2)) X1


1: (* (* X0 X1) (* X1 X2)) --> X1 IF TRUE
```

Pair

= (* X0 (* (* X0 X1) X2)) (* X0 X1) "IF " TRUE from 1 on 1

Pair

= (* (* X0 (* X1 X2)) X2) (* X1 X2) "IF " TRUE from 1 on 1

Pair

= (* (* X0 X1) (* (* (* X0 X1) X2) X3)) (* (* X0 X1) X2) "IF " TRUE from 3 on 1

Pair

= (* (* X0 X1) (* X1 X2)) X1 "IF " TRUE from 3 on 1

Pair

= (* X0 X1) (* X0 X1) "IF " TRUE from 1 on 3

Pair

= (* (* X0 X1) (* X1 X2)) (* (* X0 X1) (* X1 X3)) "IF " TRUE from 1 on 3

Pair

= (* X0 (* (* X0 X1) X2)) (* X0 X1) "IF " TRUE from 3 on 3

Pair

= (* X0 (* (* X0 X1) X2)) (* X0 (* (* X0 X1) X3)) "IF " TRUE from 3 on 3

Pair

= (* (* X0 X1) (* X1 X2)) X1 "IF " TRUE from 2 on 1

Pair

= (* (* X0 (* X1 (* X2 X3))) (* X2 X3)) (* X1 (* X2 X3)) "IF " TRUE from 2 on 1

Pair

= (* X0 X1) (* X0 X1) "IF " TRUE from 1 on 2

Pair

= (* (* X0 X1) (* X1 X2)) (* (* X3 X1) (* X1 X2)) "IF " TRUE from 1 on 2

Pair

= (* (* X0 (* X1 X2)) X2) (* X1 X2) "IF " TRUE from 2 on 2

```
Pair
= (* (* X0 (* X1 X2)) X2) (* (* X3 (* X1 X2)) X2) "IF " TRUE from 2 on 2
Pair
= (* X0 (* X1 X2)) (* X0 (* X1 X2)) "IF " TRUE from 2 on 3
Pair
= (* (* X0 (* X1 X2)) (* (* X1 X2) X3)) (* (* X0 (* X1 X2)) X2) "IF " TRUE from
2 on 3
Pair
= (* (* X0 X1) X2) (* (* X0 X1) X2) "IF " TRUE from 3 on 2
Pair
= (* (* X0 (* X1 X2)) (* (* X1 X2) X3)) (* X1 (* (* X1 X2) X3)) "IF " TRUE from
3 on 2

*** Completion terminated ***

Rules/Failures:

1: (* (* X0 X1) (* X1 X2)) --> X1 IF TRUE
2: (* (* X0 (* X1 X2)) X2) --> (* X1 X2) IF TRUE
3: (* X0 (* (* X0 X1) X2)) --> (* X0 X1) IF TRUE

Run time: 0.1667 seconds
 3 Equations retained
 18 Pairs generated
 21 Equations processed
```

```
****************************************************************
The below system is the cancellation law (Example 9) from [KB70]
****************************************************************

> (constr-complete "kbcancel")
Declaring symbol */2 with weight 2
Declaring symbol E/0 with weight 1
Declaring symbol F/2 with weight 3
Declaring symbol G/2 with weight 4

Input equation
> (F X0 (* X0 X1)) X1
Input equation
> (G (* X0 X1) X1) X0
Input equation
> (* (E) X0) X0
Input equation
> (* X0 (E)) X0

4: (F X0 (* X0 X1)) --> X1 IF TRUE
3: (G (* X0 X1) X1) --> X0 IF TRUE
2: (* (E) X0) --> X0 IF TRUE
1: (* X0 (E)) --> X0 IF TRUE

Pair
= (G X0 X0) (E) "IF " TRUE from 2 on 3
Pair
= (F (E) X0) X0 "IF " TRUE from 2 on 4
```

```
Pair
= (* (E) X0) X0 "IF " TRUE from 6 on 4


Pair
= (G X0 (E)) X0 "IF " TRUE from 1 on 3
Pair
= (F X0 X0) (E) "IF " TRUE from 1 on 4
Pair
= (E) (E) "IF " TRUE from 1 on 2
Pair
= (E) (E) "IF " TRUE from 8 on 6
Pair
= (* X0 (E)) X0 "IF " TRUE from 7 on 3
Pair
= (E) (E) "IF " TRUE from 7 on 5


 *** Completion terminated ***


Rules/Failures:
3: (G (* X0 X1) X1) --> X0 IF TRUE
4: (F X0 (* X0 X1)) --> X1 IF TRUE
7: (G X0 (E)) --> X0 IF TRUE
8: (F X0 X0) --> (E) IF TRUE
1: (* X0 (E)) --> X0 IF TRUE
5: (G X0 X0) --> (E) IF TRUE
6: (F (E) X0) --> X0 IF TRUE
2: (* (E) X0) --> X0 IF TRUE
```

```
        Run time: 0.08333 seconds
        8 Equations retained
        9 Pairs generated
        19 Equations processed
**********************************************************************
Axioms for an associative-commutative-identity function
**********************************************************************
        > (constr-complete "aci")
        Declaring symbol */2 with weight 2
        Declaring symbol ZERO/0 with weight 1


        Input equation
        (* X0 X1) > (* X1 X0)
        Input equation
        (* (* X0 X1) X2) > (* X0 (* X1 X2))
        Input equation
        (* (ZERO) X0) > X0


        1: (* (ZERO) X0) --> X0 IF TRUE
        2: (* (* X0 X1) X2) --> (* X0 (* X1 X2)) IF TRUE
        3: (* X0 X1) --> (* X1 X0) IF > X0 X1


        Pair
        = (* X0 X1) (* (ZERO) (* X0 X1)) IF TRUE from 1 on 2
        Pair
        = (* (* X0 (* X1 X2)) X3) (* (* X0 X1) (* X2 X3)) IF TRUE from 2 on 2
        .

        .

        .

        Pair
        = (* X0 (* X1 X2)) (* X2 (* X0 X1)) IF X0 > X1 AND X2 > X1 AND X0 > X2
from 3 on 5


         *** Completion terminated ***



        Rules/Failures:


        4: (* X0 (ZERO)) --> X0 IF TRUE
        1: (* (ZERO) X0) --> X0 IF TRUE
        2: (* (* X0 X1) X2) --> (* X0 (* X1 X2)) IF TRUE
        5: (* X0 (* X2 X1)) --> (* X2 (* X0 X1)) IF > X0 X2
        3: (* X0 X1) --> (* X1 X0) IF > X0 X1

        Run time: 7.35 seconds
         5 Equations retained
         18 Pairs generated
         28 Equations processed
```

APPENDIX B

PROOFS IN TERNARY BOOLEAN ALGEBRA

A. <u>PROOF OF CORRECTNESS OF THE FUNCTION TERCOMB</u>

It will now be shown that the function *Tercomb*(t) produces all terms equivalent to *t* by the cc-associative laws. Remembering from Section IX: this function will allow us to use complete commutativity when trying to find a bridge element but no terms which are $=_{cc}$ will be returned. The terms returned will be called *combinators*. Terms which are equal using complete commutativity will be added later.

Let t_1 and t_2 be terms which can be made equal by applying the cc-associative laws some number of times. We define a distance function *d*(t_1,t_2) as follows:

d(t_1,t_2) = <u>minimum</u> number of applications of the cc-associative laws (to t_1) that it takes to make the resulting term $=_{cc}$ to t_2 (thus, d(t_1,t_2) $\geq$ 0).

Again, remembering from Section IX *Tercomb* uses the functions *Depth* and *Swap-at-depth*. The *Depth* function returns the number of levels of nested subexpressions. For instance, *Depth*(h(x,y,z)) returns 0 while *Depth*(h(x,y,h(u,v,w))) and *Depth*(h(h(x,y,z),h(x,q,y),h(z,b,c))) both return 1. *Swap-at-depth* finds all possible combinators for a term at a specified depth. Note that the distance between the given term and the resulting terms from *Swap-at-Depth* is at most 1 (the resulting terms may actually be $=_{cc}$ to the original term and thus the distance would be 0).

It will first be shown that the statements 4-10 in *Tercomb* in Figure 34 find all combinators of *term* which have a distance of 1. Statements 4-10 do the above as follows: *term* is set and *temp* is initialized to the null set (it will contain all of *term*'s combinators when completed). First, clearly if the *term* is of the form h(x,y,z) (i.e. *Depth(term)* = 0) then we cannot use the associative law so *temp* remains null. Otherwise, we check at each depth for bridges and generate new terms with *Swap-at-depth*(i,term). Since all new terms generated by *Swap-at-depth* have distance $\leq$ 1 and since checking at each level for bridges insures that all bridges will be found (and the resulting combinators generated) then upon exiting statements 6-8 *temp* contains all terms equal to *term* with distance ≤ 1. Statement 10 insures that only terms with distance = 1 will be put on the *toprocess* list.

Now we have shown that we can find all combinators of *term* with distance = 1. This is not enough, we need to find all combinators of *term* with distance $\leq$ *max_distance* (the existence of such a bound will be established later). This is done with the rest of the algorithm by using a *processed* and *toprocess* list.

First, note that once a term has been processed by statements 4-10 in order to find combinators with distance = 2 we merely need to put the generated combinators through statements 4-10. Next, these newly generated combinators are used to find all combinators with distance = 3 and so on. Cycling is prevented by controlling a *toprocess* list which holds only newly generated combinators which are not $=_{cc}$ to existing ones

The rest of the statements do the above as follows. Initially, *toprocess* is set to {t} and *processed* is set to ∅, the null set. While there are still terms which have not gone through statements 4-10 (i.e. while *toprocess* $\neq \emptyset$), we take a combinator from *toprocess*, find its combinators of distance ≤ 1, put it on the *processed* list, and put the generated terms in *toprocess* if they are not already in *toprocess* or *processed* (since no new terms can be generated if the term is all ready in one of these lists). Thus on each pass of combinators through statements 4-10 only combinators of distance one greater than the current *term* are added to *toprocess*. This filtering prevents the algorithm from cycling. When *toprocess* is empty the *processed* list, which contains all combinators, is returned.

We can now say that *Tercomb* generates all combinators of any distance from the original term *t*. Also, since it only allows new combinators to be processed the algorithm will not cycle and therefore if a *max_distance* exists, then the algorithm is guaranteed to terminate.

To show that *max_distance* exists let us first look a the original term *t* with the ternary operators removed. The term *t* then has two types of symbols parentheses (right and left) and ternary elements such as x, y, and z. Let j equal the number of ternary elements in *t*, we can show that the number of pairs of parentheses in *t* must be $(j-3)/2 + 1$. Using this result we see that there are $(j + 2*((j-3)/2 + 1))!$ arrangements of ternary elements and parentheses and thus there are at most that many combinators. Actually there are far fewer than this since the terms must have matching parentheses, be a ternary expression, and actually be $=_{cc}$ to *t* after applying the cc-associative laws some number of times. Now to get our *max_distance* we note that as a worst case scenario all possible arrangements are valid ternary expressions, none of the terms are $=_{cc}$ to each other, and each can be generated by applying the associative law so that our bound, *max_distance*, in this worst case scenario is $(j + 2*((j-3)/2 + 1))!$. Again, this bound is much greater than what would actually occur but it does show that a bound exists. Since the bound exists we can conclude that the process is finite and thus is guaranteed to stop.

B. PROOF OF CORRECTNESS AND COMPLETENESS OF A COMPLETE COMMUTATIVITY UNIFICATION ALGORITHM

We present here a proof that an obvious solution to unification for completely commutative theories is both complete and correct. This proof may help us prove that the A+CC-unification algorithm is complete and correct. This proof is taken from [Mu92].

The cc-unification problem $<s,t>_{cc}$ is to find a complete set of unifiers Γ such that if $\sigma \in \Gamma$ then $s\sigma =_{cc} t\sigma$. We start by looking at $=_{cc}$.

Given two ternary terms t_1 and t_2, $t_1 =_{cc} t_2$ iff

i) $t_1 = t_2$

or

ii) $t_1 = h(x_1,x_2,x_3)$ and $t_2 = h(y_1,y_2,y_3)$ and one of the following is true:

a) $x_1 =_{cc} y_1$, $x_2 =_{cc} y_2$, and $x_3 =_{cc} y_3$

b) $x_1 =_{cc} y_1$, $x_2 =_{cc} y_3$, and $x_3 =_{cc} y_2$

c) $x_1 =_{cc} y_2$, $x_2 =_{cc} y_1$, and $x_3 =_{cc} y_3$

d) $x_1 =_{cc} y_2$, $x_2 =_{cc} y_3$, and $x_3 =_{cc} y_1$

e) $x_1 =_{cc} y_3$, $x_2 =_{cc} y_1$, and $x_3 =_{cc} y_2$

f) $x_1 =_{cc} y_3$, $x_2 =_{cc} y_2$, and $x_3 =_{cc} y_1$

Let $<t_1,t_2>_{cc}$ be a cc-unification problem. A substitution δ is a *cc-unifier* of t_1 and t_2 iff $\delta(t_1) =_{cc} \delta(t_2)$. Given a term t, let $\mathcal{T}(t) = \{t' \mid t' =_{cc} t\}$. Given substitutions δ_1 and δ_2, $\delta_1 =_{cc} \delta_2$ iff for all $v \in \mathcal{V}$, $\delta_1(v) =_{cc} \delta_2(v)$. Given a substitution δ, $\Delta(\delta) = \{\delta' \mid \delta' =_{cc} \delta\}$.

Now we will formally give the *obvious solution (OS)* to any cc-unification problem $<t_1,t_2>_{cc}$. The OS finds the set of all mgu's using R-unification for the problems $<t_1',t_2'>$ where $t_1' \in \mathcal{T}(t_1)$ and $t_2' \in \mathcal{T}(t_2)$. We will name this set $\Sigma<t_1,t_2>_{cc}$.

Theorem 1 (Correctness)

$\Sigma<t_1,t_2>_{cc} \subseteq \mathcal{U}<t_1,t_2>_{cc}$. (i.e. the obvious solution is correct).

Proof

Let $\sigma \in \Sigma<t_1,t_2>_{cc}$. Thus $\sigma(t_1') = \sigma(t_2')$ where $t_1' =_{cc} t_1$ and $t_2' =_{cc} t_2$. $\sigma(t_1) =_{cc} \sigma(t_1') = \sigma(t_2') =_{cc} \sigma(t_2)$ and therefore $\sigma(t_1) =_{cc} \sigma(t_2)$. Thus $\sigma \in \mathcal{U}<t_1,t_2>_{cc}$ and so $\Sigma<t_1,t_2>_{cc} \subseteq \mathcal{U}<t_1,t_2>_{cc}$. ■

To prove completeness of OS we will first need to label and manipulate our terms. We will label terms as follows:

i) the root node is 2.

ii) the left child of a node is p0, the middle child p1, and the right child p2 where p is the parent node label.

Example

h(x, h(u, v, w), y) =

$$\begin{tikzcd}
 & \begin{matrix}h\\2\end{matrix} \arrow[dl] \arrow[d] \arrow[dr] & \\
\begin{matrix}x\\20\end{matrix} & \begin{matrix}h\\21\end{matrix} \arrow[dl] \arrow[d] \arrow[dr] & \begin{matrix}y\\22\end{matrix} \\
\begin{matrix}u\\210\end{matrix} & \begin{matrix}v\\211\end{matrix} & \begin{matrix}w\\212\end{matrix}
\end{tikzcd}$$

Figure 36. Ternary term representation.

Definition:

Let k be an ordered pair$<\Gamma,\Psi>$. Let l = the number of ternary digits in m and p = the number of ternary digits in Γ. Let $a_i, b_i \in \{0,1,2\}$ and $\Psi \in \{012,021,102,120,201,210\}$. Also let Ψ_i be the number of the subterm containing i (e.g. if $\Psi = 120$, then $\Psi_0 = 2$, $\Psi_1 = 0$, and $\Psi_2 = 1$).

Let $m = \sum_{i=0}^{l} a_i \cdot 3^{l-i}$ and $k = <\sum_{i=0}^{p} b_i \cdot 3^{p-i}, \Psi>$

A *ternary node transformation*, k⊗m, is defined as follows:

k⊗m =

if $\Gamma \geq m$ then m

else if $b_0 = a_0$ and $b_1 = a_1$ and ... and $b_p = a_p$ then

$$\sum_{i=0}^{p} a_i \cdot 3^{l-i} + (\Psi_{a_{p+1}} \cdot 3^{l-(p+1)}) + \sum_{i=p+2}^{l} a_i \cdot 3^{l-i}$$

else m

The idea is that if Γ is equal to m for the first p digits (the length of Γ) then change the next position in m according to Ψ.

<u>Examples</u>

m = 20, k = <21,120>, k$\otimes$m = 20

m = 212, k = <21,120>, k$\otimes$m = 211

m = 211112, k = <211,201>, k$\otimes$m = 211212

<u>Definition:</u>

A *single (ternary) swap* ρ is a mapping $\rho:\mathcal{T}\rightarrow\mathcal{T}$, if $t \in \mathcal{T}$, $\rho = <\Gamma, \Psi>$ then $\rho(t)$ is the term obtained from t by replacing every node, n, of t by $\rho\otimes$n.

<u>Example</u>

If ρ = <21,021> and t = h(v,h(x,y,z),w), then

ρ(t) = h(v,h(x,z,y),w).

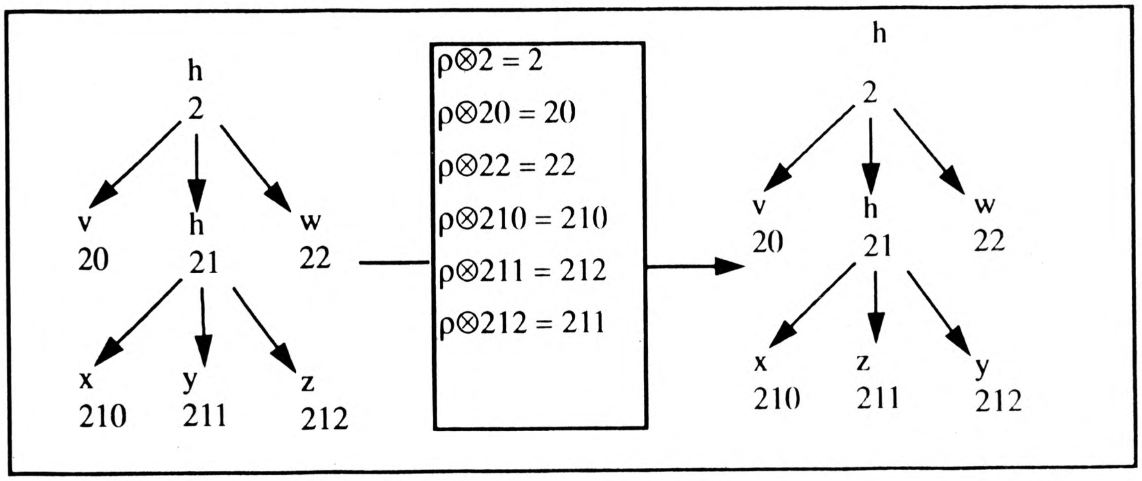

Figure 37. Ternary swap.

<u>Definitions:</u>

i) Let ρ_1 and ρ_2 be two single swaps, $\rho_1 \circ \rho_2(t) = \rho_1(\rho_2(t))$.

ii) A *(ternary) swap* ρ is a $\rho:\mathcal{T} \rightarrow \mathcal{T}$ denoted by the tuple $<\rho_1,\rho_2,\ldots,\rho_n>$ such that $\rho = \rho_1 \circ \rho_2 \circ \ldots \circ \rho_n$.

iii) $\rho_1 = \rho_2$ iff for all $t \in \mathcal{T}$, $\rho_1(t) = \rho_2(t)$.

iv) ρ_{id} is the identical swap (i.e. for all nodes n, $\rho_{id}(n) = n$).

(Note that any single swap of the form $\rho = <\Gamma,012>$ is an identical swap).

This lemma shows some interesting properties of swaps:

Lemma 1

i) $\exists$ (single)swaps ρ_1, ρ_2 such that $\rho_1 \circ \rho_2 \neq \rho_2 \circ \rho_1$

(e.g. $\rho_1 = <2,120>$, $\rho_2 = <2,210>$).

ii) $\exists$ a (single) swap such that $\rho \circ \rho \neq \rho_{id}$

(e.g. $\rho = <2,120>$).

iii) $\exists$ a (single) swap such that $\rho \circ \rho = \rho_{id}$

(e.g. $\rho = <2,012>$).

Definitions:

i) Let $\rho_1 = <\Gamma^1, \Psi^1>$, $\rho_2 = <\Gamma^2, \Psi^2>$ be single swaps. $\rho_1 < \rho_2$ iff $\Gamma^1 < \Gamma^2$ or ($\Gamma^1 = \Gamma^2$ and $\Psi^1 < \Psi^2$).

ii) A swap $\rho = <\rho_1,...,\rho_n>$ is in *normal form* iff $\rho_1 < \rho_2 < ... < \rho_n$.

iii) ρ is the *ordered form* of $\hat{\rho} = <\rho_1,\rho_2,...,\rho_n>$ iff $\rho = <\alpha_1,\alpha_2,...,\alpha_n>$ is in normal form and the sequence $\alpha_1, ..., \alpha_n$ is a permutation of the sequence $\rho_1, ..., \rho_n$. In other words, ρ is $\hat{\rho}$ with the members ordered from lowest (nearest to the root) to highest (farthest from the root). Note, ρ is not necessarily equal to $\hat{\rho}$.

Here is a useful property relating $=_{cc}$ and swaps that shows why we use swaps in our proof of completeness.

Lemma 2

Let $t_1 =_{cc} t_2$. Then there exists a swap ρ such that $\rho(t_1) = t_2$.

Proof

We will do a proof by construction. If $t_1 = t_2$ then let $\rho = \rho_{id}$ and we are done. Otherwise, $t_1 = h(x_1,x_2,x_3)$ and $t_2 = h(y_1,y_2,y_3)$ and (at least) one of the following is true:

1) $x_1 =_{cc} y_1$, $x_2 =_{cc} y_2$, and $x_3 =_{cc} y_3$

2) $x_1 =_{cc} y_1$, $x_2 =_{cc} y_3$, and $x_3 =_{cc} y_2$

3) $x_1 =_{cc} y_2$, $x_2 =_{cc} y_1$, and $x_3 =_{cc} y_3$

4) $x_1 =_{cc} y_2$, $x_2 =_{cc} y_3$, and $x_3 =_{cc} y_1$

5) $x_1 =_{cc} y_3$, $x_2 =_{cc} y_1$, and $x_3 =_{cc} y_2$

6) $x_1 =_{cc} y_3$, $x_2 =_{cc} y_2$, and $x_3 =_{cc} y_1$

Let ρ be the ordered form of $\rho' \circ \rho_{rest}$ where

$\rho' = \rho_{id}$ if 1) is true

$= \langle n,021 \rangle$ if 2) is true

$= \langle n,102 \rangle$ if 3) is true

$= \langle n,201 \rangle$ if 4) is true

$= \langle n,120 \rangle$ if 5) is true

$= \langle n,210 \rangle$ if 6) is true

where n is the node whose subterms we are trying to make equal (at the top level n=2).

ρ_{rest} is a swap and is the composition of the swaps ρ_0, ρ_1, and ρ_2 where ρ_i is the swap constructed from the ith subterm equality using the method above.

By inspection it can be seen that $\rho(t_1) = t_2$. ■

The following lemmas will help us produce equal swaps which are ordered.

Lemma 3

Let $\rho_1 = <\Gamma^1,\Psi^1>$, $\rho_2 = <\Gamma^2,\Psi^2>$ be single swaps. If $\Gamma^1 > \Gamma^2$ and $\Gamma^1 \neq \Gamma^2\alpha$ where $\alpha \in \{0,1,2\}^+$ then $\rho_1 \circ \rho_2 = \rho_2 \circ \rho_1$.

Proof

Let $\beta,\beta' \in \{0,1,2\}^+$ in the following. Suppose $\rho_1 \circ \rho_2 \neq \rho_2 \circ \rho_1$. Then there exists a $t \in \mathcal{T}$ such that $\rho_1(\rho_2(t)) \neq \rho_2(\rho_1(t))$. Let n be a node in t in which $\rho_1(\rho_2(n)) \neq \rho_2(\rho_1(n))$. Then at least one of ρ_1 or ρ_2 changes n's value. If ρ_2 changes n's value then $n = \Gamma^2\beta$ and $\rho_2(n) = \Gamma^2\beta'$. Now since $\Gamma^1 \neq \Gamma^2\alpha$, $\rho_1(\rho_2(n)) = \rho_2(n)$ and $\rho_1(n) = n$. Thus $\rho_2(\rho_1(n)) = \rho_2(n)$. Therefore $\rho_1(\rho_2(n)) = \rho_2(\rho_1(n))$ - a contradiction. Otherwise, if ρ_1 changes the value of n then $n = \Gamma^1\beta$ and $\rho_1(n) = \Gamma^1\beta'$. $\Gamma^1 \neq \Gamma^2\alpha$ and thus $\rho_2(\rho_1(n)) = \rho_1(n)$ and $\rho_2(n) = n$. Thus $\rho_1(\rho_2(n)) = \rho_1(n)$ and therefore $\rho_1(\rho_2(n)) = \rho_2(\rho_1(n))$ - a contradiction. Thus all nodes are the same. All terms are therefore also the same and we have our proof.■

Definition:

Let $k \oplus m$ be defined the same as $k \otimes m$ except that Ψ_i is defined as the number in the ith position of Ψ (e.g. if $\Psi = 120$, $\Psi_0 = 1$, $\Psi_1 = 2$, $\Psi_2 = 0$).

Lemma 4

Let $k_1 = <\Gamma^1,\Psi^1>$ and $k_2 = <\Gamma^2,\Psi^2>$ be single swaps. If $k_1 = k_2$ or ($k_2 < k_1$ and $\Gamma^1 \neq \Gamma^2$) then $<k_1> \circ <k_2> = k_2 \circ <k_2 \oplus \Gamma^1, \Psi^1>$.

Proof (by cases)

If $k_1 = k_2$ then $k_2 \oplus \Gamma^1 = \Gamma^1$ and thus $k_2 \circ <k_2 \oplus \Gamma^1,\Psi^1> = k_2 \circ <\Gamma^1,\Psi^1> = <k_2> \circ <k_1> = <k_1> \circ <k_2>$ (since k1 = k2).

Else if $k_2 < k_1$ and $\Gamma^1 \neq \Gamma^2$ then

$k_2 \oplus \Gamma^1 = \Gamma^1$ (if $\Gamma^1 \neq \Gamma^2\alpha$ with $\alpha \in \{0,1,2\}^+$), or

$= \Gamma^2\beta$ (if $\Gamma^1 = \Gamma^2\alpha$ with $\alpha,\beta \in \{0,1,2\}^+$)

(Note here, for Corollary 1, that $\Gamma^2 < k_2 \oplus \Gamma^1$).

If $k_2 \oplus \Gamma^1 = \Gamma^1$ then $k_2 \circ \langle k_2 \oplus \Gamma^1, \Psi^1 \rangle = k_2 \circ \langle \Gamma^1, \Psi^1 \rangle = \langle k_2 \rangle \circ \langle k_1 \rangle$. Now since $\Gamma^1 > \Gamma^2$ and $\Gamma^1 \neq \Gamma^2\alpha$, by Lemma 3 $\langle k_2 \rangle \circ \langle k_1 \rangle$

$(= k_2 \circ \langle k_2 \oplus \Gamma^1, \Psi^1 \rangle) = \langle k_1 \rangle \circ \langle k_2 \rangle$ and we are done.

Else $\Gamma^1 = \Gamma^2\alpha$. By inspecting $k_2 \oplus \Gamma^1$ we see that the process modifies Γ^1 to take into account that k_2 will not be applied before k_1 and thus the swaps are equal.■

Corollary 1

After using Lemma 4 on $\langle k_1 \rangle \circ \langle k_2 \rangle$, the result $\langle k_1', k_2' \rangle$ is in normal form or ($\Gamma^1 = \Gamma^2$ and $\Psi^2 < \Psi^1$).

Proof

First, note that if $k_1 < k_2$ or ($k_2 < k_1$ and $\Gamma^1 = \Gamma^2$) then $k_1' = k_1$ and $k_2' = k_2$ and we are done. Otherwise, $k_2 < k_1$, $\Gamma^1 \neq \Gamma^2$, $k_1' = k_2$ and $k_2' = \langle k_2 \oplus \Gamma^1, \Psi^1 \rangle$. Now by the parenthesized note in Lemma 4 we see that it must be that $k_1' < k_2'$ and we are done.■

Definitions:

Let $\rho = \langle \rho_1, \rho_2, \ldots, \rho_m \rangle$ and $\hat{\rho} = \langle \alpha_1, \alpha_2, \ldots, \alpha_n \rangle$ be two swaps.

i) $\hat{\rho} \subseteq \rho$ iff $\hat{\rho} = \rho$ and $m \leq n$.

ii) ρ is *maximal* iff for all $\hat{\rho}$ with $\hat{\rho} = \rho$: $\hat{\rho} \subseteq \rho$.

The following two lemmas help produce swaps which are maximal.

Lemma 5

If $k_1 = \langle \Gamma, 012 \rangle$ and k_2 is a swap then $k_1 \circ k_2 = k_2 \circ k_1 = k_2$.

Proof

Comes from the fact that $\langle \Gamma, 012 \rangle = p_{id}$.

Lemma 6

If $k_1 = <\Gamma,\Psi^1>$ and $k_2 = <\Gamma,\Psi^2>$ then there exist a Ψ such that $<k_1>\circ<k_2> = <\Gamma,\Psi>$.

Proof

The following table gives the necessary conversions.

Table IV. Ψ conversion table.

Ψ^1	Ψ^2	Ψ
012	Ψ^2	Ψ^2
Ψ^1	012	Ψ^1
021	021	012
021	102	201
021	120	210
021	201	102
021	210	120
102	021	120
102	102	012
102	120	021
102	201	210
102	210	201
120	021	102
120	102	210
120	120	201
120	201	012
120	210	021
201	021	210
201	102	021
201	120	012
201	201	120
201	210	102
210	021	201
210	102	120
210	120	102
210	201	021
210	210	012

The next two lemmas allow us to find maximal swaps in normal form.

Lemma 7

For every swap ρ there exists a swap $\hat{\rho}$ such that

i) $\hat{\rho}$ is in normal form

ii) $\hat{\rho} = \rho$

Proof

Let $\rho = <\rho_1,...,\rho_n>$. If we apply Lemma 4 to ρ until it cannot be applied anymore (giving us ρ') then, by Corollary 1, either ρ' is in normal form or there exist a ρ_i, ρ_{i+1} such that $\Gamma^i = \Gamma^{i+1}$ but $\Psi^{i+1} < \Psi^i$. (Note that single swaps with the same Γ must be consecutive). If such ρ_i and ρ_{i+1} exist we apply the Lemma 6 repeatedly until this no longer exists (giving $\hat{\rho}$). All Γ's in $\hat{\rho}$ are then unique and in order and thus $\hat{\rho} = \rho$ and is in normal form. ■

Lemma 8

To every swap $\hat{\rho}$ there exist a unique maximal swap ρ in normal form such that $\hat{\rho} = \rho$.

Proof

By the Lemma 7 there exist a ρ' in normal form such that $\rho' = \hat{\rho}$. Let ρ be a maximal swap with $\rho = \rho'$. Suppose, by way of contradiction, that ρ is not unique, then there exist a ρ'' such that $\rho'' = \rho'$, ρ'' is maximal, and ρ'' is not syntactically equal to ρ. $\rho'' = \rho$ by transitivity of equal swaps. $\rho'' \subseteq \rho$ and $\rho \subseteq \rho''$ and thus the number of single swaps is the same. Letting $\rho'' = <\rho_1,...,\rho_m>$, $\rho = <\alpha_1,...,\alpha_m>$, suppose $\rho_m \neq \alpha_m$ then assume, without loss of generality, $\rho_m < \alpha_m$. Since $\alpha_m > \alpha_i$ for $i < m$, $\alpha_i \neq \rho_m$ for all $1 \leq i \leq m$. Thus the single swap ρ_m has no equivalent in ρ (i.e. swaps performed higher up will not produce the same results). But this is contrary to $\rho = \rho''$, therefore $\rho_m = \alpha_m$. Clearly, now using $\rho'' = <\rho_1,...,\rho_{m-1}>\circ\rho_m$ and $\rho = <\alpha_1,...,\alpha_{m-1}>\circ\rho_m$ we can use the above arguments to show $\rho_{m-1} = \alpha_{m-1}$ and so on until $\rho_1 = \alpha_1$ and thus $\rho'' = <\rho_1,...,\rho_m> = \rho$ and thus ρ'' is syntactically equal to ρ - a contradiction- and thus ρ is unique. ■

Corollary 2

Let $t_1 =_{cc} t_2$. Then there exists a unique maximal swap ρ in normal form such that $\rho(t_1) = t_2$. ■

Proof

This follows directly from Lemma 2 and Lemma 8.

The following definition is tedious but will help us in our completeness proof.

Definition:

Let $\delta(t_1) =_{cc} \delta(t_2)$ and ρ_1 and ρ_2 be swaps such that $\rho_1(\delta(t_1)) = \rho_2(\delta(t_2))$ then ρ_1 and ρ_2 are *compatible with* δ iff for all $v \in \mathrm{Var}(t_1) \cup \mathrm{Var}(t_2)$,

$\mathrm{Sub}(v,t_1,\rho_1(\delta(t_1))) \cup \mathrm{Sub}(v,t_2,\rho_2(\delta(t_2)))$ is a singleton.

where $\mathrm{Sub}(v,t,\rho(\delta(t)))$ returns: if $v \in \mathrm{Var}(t)$ and $(v/t') \in \delta$ return what t' looks like at every location of $v \in t$ after ρ is applied to $\delta(t)$.

Example

Let $\delta = \{x/h(u,v,w)\}$, $\rho_1 = <20,021> \circ <21,210>$ then

$\mathrm{Sub}(x,h(u,v,w),\rho_1(\delta(h(x,x,y)))) =$

$\mathrm{Sub}(x,h(u,v,w),\rho_1(h(h(u,v,w),h(u,v,w),y))) =$

$\mathrm{Sub}(x,h(u,v,w),h(h(u,w,v),h(w,v,u),y)) =$

$\{h(u,w,v),h(w,v,u)\}$.

Lemma 9

Let $<t_1,t_2>_{cc}$ be a cc-unification problem with cc-unifier $\delta{:}\delta(t_1) =_{cc} \delta(t_2)$. Let ρ_1 and ρ_2 be swaps compatible with δ such that $\rho_1(\delta(t_1)) = \rho_2(\delta(t_2))$. Then there exists $\hat{\delta} \in \Delta(\delta)$ such that $\hat{\delta}$ is a R-unifier for $<\rho_1(t_1),\rho_2(t_2)>$: $\hat{\delta}(\rho_1(t_1)) = \hat{\delta}(\rho_2(t_2))$

Proof

Let $\hat{\delta} = \{v/t \mid v \in \mathrm{Var}(t_1) \cup \mathrm{Var}(t_2)$, t is the unique term described in the definition of compatibility$\}$. $\hat{\delta} \in \Delta(\delta)$ since, because ρ_1 and ρ_2 are compatible swaps, the new t's can only be commutative terms of the original t's in δ. $\hat{\delta}(\rho_1(t_1)) = \hat{\delta}(\rho_2(t_2))$ because

ρ_1 and ρ_2 are compatible and thus the terms will be made as alike as possible before R-unification is applied and variables can thus be replaced by a single term. ■

Example

$t_1 = h(x,x,x)$ and $t_2 = h(h(x',y,z),h(z,y,x'),x)$

$\delta = \{x/(h(z,x',y)\}$ and $\delta(t_1) =_{cc} \delta(t_2)$

$\rho_1 = <20,201>\circ<21,201>\circ<22,201>$ and $\rho_2 = <21,210>\circ<22,201>$

$\rho_1(\delta(t_1)) = h(h(x',y,z),h(x',y,z),h(x',y,z)) = \rho_2(\delta(t_2))$

$\rho_1(t_1) = h(x,x,x)$ and $\rho_2(t_2) = h(h(x',y,z),h(x',y,z),x)$

$\hat{\delta} = \{x/h(x',y,z)\} \in \Delta(\delta)$

$\hat{\delta}(\rho_1(t_1)) = \hat{\delta}(\rho_2(t_2))$

Note that if ρ_1 and ρ_2 were not compatible then we could not find an R-unifier of $\rho_1(t_1)$ and $\rho_2(t_2)$ since we would need both $(x/h(x',y,z))$ and $(x/h(z,y,x'))$ or some similarly cc-equal terms.

The correctness of $\Sigma<t_1,t_2>_{cc}$ was previously presented. The completeness of obvious solution follows.

Theorem2 (Completeness)

Let $t_1,t_2 \in \mathcal{T}$ and $\Sigma<t_1,t_2>_{cc}$ be the set of unifiers generated by OS. Then, for all δ with $\delta(t_1) =_{cc} \delta(t_2)$, there exists $\sigma \in \Sigma<t_1,t_2>_{cc}$ and a λ such that $\delta =_{cc} \lambda\circ\sigma$.

Proof

Let $\delta(t_1) =_{cc} \delta(t_2)$ and ρ_1 and ρ_2 be swaps compatible with δ such that $\rho_1(\delta(t_1)) = \rho_2(\delta(t_2))$. By Lemma 9 there exist $\hat{\delta} \in \Delta(\delta)$ such that $\hat{\delta}$ is a R-unifier for $<\rho_1(t_1),\rho_2(t_2)>$: $\hat{\delta}(\rho_1(t_1)) = \hat{\delta}(\rho_2(t_2))$. Let σ be the mgu of $<\rho_1(t_1),\rho_2(t_2)>$ and λ be the substitution such that $\hat{\delta} = \lambda\circ\sigma$, which exists by the definition of a mgu. Note that

$\rho_1(t_1) \in \mathcal{T}(t_1)$ and $\rho_2(t_2) \in \mathcal{T}(t_2)$ and thus $\sigma \in \Sigma\langle t_1,t_2\rangle_{cc}$. Since $\lambda\circ\sigma = \hat{\delta} \in \Delta(\delta)$, $\lambda\circ\sigma =_{cc} \delta$ and thus we have our proof. ■

This proof is an extension of the results of Siekmannn. Siekmann's paper [Si79] not only proved completeness and correctness for binary commutative unification but also presented an algorithm which did not produce so many dependent unifiers (i.e. a closer to minimal set of unifiers). This needs to be done for the presented solution also. Doing so will greatly speed up any implemented unification algorithm for complete commutativity.

REFERENCES

[Ba88] Baird, T., (1988). "Complete sets of reductions modulo a class of equational theories which generate infinite congruence classes." Ph.D. dissertation, University of Missouri-Rolla, Rolla, MO.

[BD89] Bachmair, L., and N. Dershowitz (1989). "Completion for rewriting modulo a congruence." *Theoretical Computer Science*, volume 67, pp. 173-201.

[BDP89] Bachmair, L., N. Dershowitz, and D. Plaisted (1989). "Completion without failure." *Resolution of Equations in Algebraic Structures, Volume 2: Rewriting Techniques*, H. Ait-Kaci and M. Nivat, eds., Academic Press, pp. 1-30.

[BHS89] Burckert, H.-J., A. Herold, and M. Schmidt-Schauss (1989). "On equational theories, unification, and (un)decidability." *Journal of Symbolic Computation*, volume 8, pp. 3-49.

[BK86] Bergstra, J., and J. Klop (1986). "Conditional rewrite rules: Confluence and termination." *Journal of Computer and Systems Sciences*, volume 32, pp.323-362.

[BP85] Bachmair, L., and D. Plaisted (1985). "Termination orderings for associative-commutative rewriting systems." *Journal of Symbolic Computation*, volume 1, pp. 329-349.

[BP87] Bachmair, L., and D. Plaisted (1987). "Completion for rewriting modulo a congruence." *Proceedings of the Conference on Rewriting Techniques and Applications*, P. Lescanne, ed., Lecture Notes in Computer Science 256, Springer-Verlag, pp. 192-203.

[BPW89] Baird, T., G. Peterson, and R. Wilkerson (1989). "Complete sets of reductions modulo associativity, commutativity and identity." *Proceedings of the Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science 355, Springer-Verlag, pp. 29-44.

[Ch89] Christian, J., (1989). "High performance permutative completion." Ph.D. dissertation, University of Texas-Austin, Austin, TX.

[CL88] Christian, J., and P. Lincoln (1988). "Adventures in associative-commutative unification." Technical Report, MCC, Number ACA-ST-275-87, Austin, Texas.

[Co90] Comon, H., (1990). "Solving inequations in term algebras." *Proceedings of the Fifth Symposium on Logic in Computer Science*, pp. 62-69.

[CR36] Church, A., and J. B. Rosser (1936). "Some properties of conversion." *Transactions of the American Mathematical Society*, volume 39, pp. 472-482.

[De79] Dershowitz, N., (1979). "Orderings for term-rewriting systems." *Proceedings of the 20th Symposium on Foundations of Computer Science*, pp. 123-131 (Later version in *Journal of Theoretical Computer Science*, volume 17, 1982, pp. 279-301).

[De87] Dershowitz, N., (1987). "Termination of rewriting." *Journal of Symbolic Computation*, volume 3, pp. 69-115.

[De89] Dershowitz, N., (1989). "Completion and its applications." *Resolution of Equations in Algebraic Structures, Volume 2: Rewriting Techniques*, H. Ait-Kaci and M. Nivat, eds., Academic Press, pp. 1-30.

[DJ90] Dershowitz, N., and J.-P. Jouannaud (1990). "Rewrite systems." *Handbook of Theoretical Computer Science*, Edited by J. von Leeuwen, Elsevier Science Publishers, pp. 269-278.

[DM79] Dershowitz, N., and Z. Manna (1979). "Proving termination with multiset orderings." *Communications of the ACM*, volume 22, pp. 465-467.

[DOS88] Dershowitz, N., M. Okada, and G. Sivakumar (1988). "Canonical conditional rewrite systems." *Proceedings of the 9th International Conference on Automated Deduction*, Lecture Notes in Computer Science 310, E. Lusk and R. Overbeek, eds., Springer-Verlag, pp. 538-549.

[FG84] Forgaard, R., and J. V. Guttag (1984). "A term rewriting system generator with failure-resistance Knuth-Bendix." Technical Report, MIT Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

[Gr47] Grau, A., (1947). "Ternary boolean algebra." *Bulletin of the American Mathematical Society*, volume 53, pp. 567-572.

[HO80] Huet, G., and D. Oppen (1980). "Equations and rewrite rules: A survey." *Formal Language Theory: Perspectives and Open Problems*, R. Book, ed., Academic Press.

[HR87] Hsiang, J., and M. Rusinowitch (1987). "On word problems in equational theories." *Proceedings of the 19th International Colloquium on Automata, Languages, and Programming*, Th. Ottomann, ed., Karlsruhe (Germany), Springer-Verlag, Lecture Notes in Computer Science 267, pp. 54-71.

[Hu80] Huet, G., (1980). "Confluent reductions: Abstract properties and applications to term rewriting systems." *Journal of the Association for Computing Machinery*, volume 27, pp. 797-821.

[Hu81] Huet, G., (1981). "A complete proof of correctness of the Knuth-Bendix completion algorithm." *Journal of Computer and System Sciences*, volume 23, pp. 11-21.

[Hul81] Hullot, J.-M., (1981). "A catalogue of canonical term rewriting systems." Technical Report CSL-113, SRI International.

[JK86] Jouannand, J.-P., and H. Kirchner (1986). "Completion of a set of rules modulo a set of equations." *SIAM Journal of Computing*, volume 15, pp. 1155-1194.

[JK90] Jouannand, J.-P., and C. Kirchner (1990). "Solving equations in abstract algebras: A rule-based survey of unification.", Research Report, CRIN 1990. To appear in *Festschrift for Robinson*, J.-L. Lassez and G. Plotkin, eds.

[JM84] Jouannand, J.-P., and M. Munoz (1984). "Termination of a set of rules modulo a set of equations." *Proceedings of the 7th International Conference on Automated Deduction*, R. Shostak, ed., Springer Lecture Notes on Computer Science, volume 170, pp. 175-193.

[KB70] Knuth, D., and P. Bendix (1970). "Simple word problems in universal algebras." *Computational Problems in Abstract Algebras*, J. Leech, ed., Pergammon Press, Oxford, England, pp. 263-297.

[Ki87] Kirchner, C., (1987). "Methods and tools for equational unification." *Proceedings of the Colloquium on the Resolution of Equations in Algebraic Structures*, Austin, TX.

[KKR90] Kirchner, C., H. Kirchner and M. Rusinowitch (1990). "Deduction with symbolic constraints." *Revue d'intelligence artificielle*, volume4, pp. 9-52 (Also available as Rapports de Recherche No. 1358, Institut National de Recherche en Informatique et en Automatique, France).

[KN86] Kapur, D., and P. Narendran (1986). "NP-completeness of the set unification and matching problems." *Proceedings of the 8th International Conference on Automated Deduction*, Lecture Notes in Computer Science 230, Springer-Verlag, pp. 489-495.

[Kn89] Knight, K., (1989). "Unification: A multidisciplinary survey." *ACM Computing Surveys*, volume 21, pp. 93-124.

[LB77a] Lankford, D. S., and A. M. Ballantyne (1977). "Decision procedures for simple equational theories with commutative axiom: Complete sets of commutative reductions." Technical Report, Mathematics Department, University of Texas-Austin, Austin, Texas.

[LB77b] Lankford, D. S., and A. M. Ballantyne (1977). "Decision procedures for simple equational theories with permutative axioms: Complete sets of permutative reductions." Technical Report, Mathematics Department, University of Texas-Austin, Austin, Texas.

[LB77c] Lankford, D. S., and A. M. Ballantyne (1977). "Decision procedures for simple equational theories with commutative-associative axioms: Complete sets of commutative-associative reductions." Technical Report, Mathematics Department, University of Texas-Austin, Austin, Texas.

[Le82] Lescanne, P., (1982). "Some properties of decomposition ordering: A simplification ordering to prove termination of rewriting systems." *RAIRO Theoretical Informatics*, volume 16, pp. 331-347.

[MM76] Martelli, A., and U. Montanari (1976). "Unification in linear time and space: A structured presentation." Internal Report B 76-16, Istituto di Elaborazione della Informazione, Pisa, Italy.

[MN70] Mana, Z., and S. Ness (1970). "On the termination of Markov algorithms." *Proceedings of the Third Hawaii International Conference on System Sciences*, Honolulu, HI, pp. 789-792.

[MN90] Martin, U., and T. Nipkow (1990). "Ordered rewriting and confluence." *Proceedings of the 10th International Conference on Automated Deduction*, Lecture Notes in Computer Science 449, M. Stickel, ed., Springer-Verlag, pp. 366-380.

[Mu92] Murphy, D., (1992). "Unification for completely commutative theories." *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing*, H. Berghel, E. Deaton, et. al., eds., ACM Press, pp. 547-553.

[Ne42] Newman, M. H. A., (1942). "On theories with a combinatorial definition of 'equivalence'." *Annals of Mathematics*, volume 43, pp. 223-243.

[Pe90a] Peterson, G., (1990). "Solving term inequalities." *Proceedings of the Eight National Conference on Artificial Intelligence*, AAAI Press, Menlo Park, pp. 258-263.

[Pe90b] Peterson, G., (1990). "Complete sets of reductions with constraints." *Proceedings of the 10th International Conference on Automated Deduction*, Lecture Notes in Computer Science 449, M. Stickel, ed., Springer-Verlag, pp. 381-395.

[Pe91a] Peterson, G., (1991). Personal communication.

[Pe91b] Peterson, G., (1991). "Term algebra." Unpublished manuscript.

[Pl73] Plotkin, G., (1973). "Building-in equational theories." *Machine Intelligence*, volume 7, B. Meltzer and D. Michie, eds., American Elsevier, New York, pp. 73-90.

[Pl78] Plaisted, D., (1978). "A recursively defined ordering for proving termination of term rewriting systems." Report R-78-943, Department of Computer Science, University of Illinois, Urbana, IL.

[PS81] Peterson, G., and M. Stickel (1981). "Complete sets of reductions for some equational theories." *Journal of the Association for Computing Machinery*, volume 28, pp. 233-264.

[PW78] Paterson, M., and M. Wegman (1978). "Linear unification.", *Journal of Computer and System Sciences*, volume 16, pp. 158-167.

[Ro65] Robinson, J. A., (1965). "A machine-oriented logic based on the resolution principle." *Journal of the Association for Computing Machinery*, volume 12, pp. 23-41.

[Sc86] Schmidt-Schauss, M., (1986). "Unification under associativity and idempotence is of type nullary." *Journal of Automated Reasoning*, volume 2, pp. 277-281.

[Si79] Siekmann, J., (1979). "Matching under commutativity." *Symbolic and Algebraic Computation*, Springer-Verlag, Berlin, West Germany, pp. 531-545.

[Si89] Siekmann, J., (1989). "Unification theory." *Journal of Symbolic Computation*, volume 7, pp. 207-247.

[St81] Stickel, M., (1981). "A unification algorithm for associative commutative functions." *Journal of the Association for Computing Machinery*, volume 28, pp. 423-434.

[St84] Steele, G., (1984). *COMMON LISP: the language*, Digital Press, Burlington, MA.

[WB83] Winkler, F., and B. Buchberger (1983). "A criterion for eliminating unnecessary reductions in the Knuth-Bendix algorithm." *Colloquia Mathematica Societatis Janos Bolyai 42, Algebra, Combinatorics and Logic in Computer Science*, pp. 849-869.

[WMJ90] Wilkerson, R., D. Murphy, J. Jenness, W. Pinet, and D. Brazeal (1990). "Ternary boolean algebra unification." *Proceedings of the Symposium on Applied Computing*, H. Berghel, J. Talburt, and D. Roach, eds., IEEE Computer Society Press, pp. 119-121.

[Wo88] Wos, L., (1988). *Automated Reasoning: 33 Basic Research Problems*, Prentice-Hall, Inc.

[WW82] Winker, S., and L. Wos (1982). "Automated generation of models and counterexamples and its application to open questions in ternary boolean algebra" *Proceedings of the 8th International Symposium on Multiple-Valued Logic*, IEEE Press, New York, pp. 251-256.

[Ye85] Yelick, K., (1985). "Combining unification algorithms for confined regular equational theories." *Proceedings of the Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science 202, J.-P. Jouannaud, ed., Springer-Verlag, pp. 365-380.

[ZK89] Zhang, H., and D. Kapur (1989). "Consider only general superpositions in completion procedures." *Proceedings of the Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science 355, pp. 513-527.

[ZR85] Zhang, H., and J.-L. Remy (1985). "Contextual rewriting." *Proceedings of the Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science 202, J.-P. Jouannaud, ed., Springer-Verlag, pp. 46-62.

VITA

Daniel Patrick Murphy was born on July 9, 1966 in St. Louis, Missouri. He graduated from Rosary High School in St. Louis, Missouri in May 1984.

He graduated Magna Cum Laude with his Bachelor of Science degree in Computer Science in August 1987 from the University of Missouri-St. Louis. During this time he tutored high school and college students through the university.

He received his Master of Science degree in Computer Science in May 1989 from the University of Missouri-Rolla. During this time and while pursuing his Ph.D. he worked as a programmer for the Generic Mineral Technology Center for Pyrometallurgy, a graduate research assistant for the Intelligent Systems Center, and as an independent consultant.