Term-Rewriting Techniques for Logic Programming I: Completion

TR88-019

April 1988

*Michael P. Smith**
David A. Plaisted

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

**Department of Computer Science, Duke University, Durham, NC.*

Term-Rewriting Techniques for Logic Programming I: Completion*

Michael P. Smith
Department of Computer Science
Duke University
Durham, NC 27706
mps@cs.duke.edu

David A. Plaisted
Department of Computer Science
The University of North Carolina
Chapel Hill, NC 27514
plaisted@cs.unc.edu

April 19, 1988

*Research supported in part by National Science Foundation Grant DCR-8516243. Thanks to Donald Loveland, Gopalan Nadathur, Xumin Nie, and the participants in the Declarative Languages Seminar at UNC and the Theorem-Proving Seminar at Duke for valuable comments.

$$\begin{tikzcd}
 & & & & 0 \arrow[dlll] \arrow[drr] & & & \\
 & 1 \arrow[dl] \arrow[dr] & & & & & 2 \arrow[dl] \arrow[dr] & \\
11 & & 12 \arrow[dl] \arrow[dr] & & & 21 \arrow[dl] \arrow[dr] & & 22 \\
 & 121 & & 122 & 211 & & 212 &
\end{tikzcd}$$

Figure 1: Occurrences in a(a(b,a(b,w)),a(a(b,b),c))

1 Theoretical Background

Reasoning about equations is fundamental in computer science and elsewhere. Yet equality is difficult for many automated theorem provers to handle. The chief obstacle to automating equational reasoning is the lack of control inherent in the basic mode of equational inference: bidirectional matching and substitution. A natural answer to this problem is to add direction to equations, turning them into **rewrite rules**. We have simulated the Knuth-Bendix technique for converting a set of equations into a complete set of rewrite rules in a first-order theorem prover implemented in Prolog. Since unification plays a primary role in this technique, Knuth-Bendix completion provides an obvious route for dealing with equations in a logic programming context.

1.1 Equational and Rewrite Systems

Let T be a set of first-order terms constructed from some set of function symbols F and variables V. We shall informally use strings of integers, or *occurrences*, to identify subterm positions. We write $\tau[v]_p$ to denote a term τ with a subterm v at position p. For example, the occurrences of s = a(a(b,a(b,w)),a(a(b,b),c)), are mapped in tree form in Figure 1. Note

$$\mathrm{s}[\mathrm{a}(\mathrm{a}(\mathrm{b},\mathrm{a}(\mathrm{b},\mathrm{w})),\mathrm{a}(\mathrm{a}(\mathrm{b},\mathrm{b}),\mathrm{c}))]_0 = \mathrm{s}[\mathrm{a}(\mathrm{b},\mathrm{a}(\mathrm{b},\mathrm{w}))]_1 = \mathrm{s}[\mathrm{b}]_{11} = \mathrm{s}[\mathrm{a}(\mathrm{b},\mathrm{w})]_{12} =$$

$$\mathrm{s}[\mathrm{b}]_{121} = \mathrm{s}[\mathrm{w}]_{122} = \mathrm{s}[\mathrm{a}(\mathrm{a}(\mathrm{b},\mathrm{b}),\mathrm{c})]_2 = \mathrm{s}[\mathrm{a}(\mathrm{b},\mathrm{b})]_{21} = \mathrm{s}[\mathrm{b}]_{211} = \mathrm{s}[\mathrm{b}]_{212} = \mathrm{s}[\mathrm{c}]_{22}.$$

An **equational system** $\mathcal{E}$ over T has as axioms a set of equations of the form $\alpha = \beta$. When a term contains a subterm (not necessarily proper) matching either side of an equation, that subterm may be replaced by the other side of the equation. More formally, an equational system has the single rule of inference:

$$\frac{\Xi(\tau[u\sigma]_p)\,,\; u \doteq v}{\Xi(\tau[v\sigma]_p)}$$

where Ξ stands for any propositional context, $\alpha \doteq \beta$ represents ambiguously either $\alpha = \beta$ or $\beta = \alpha$, and σ is a substitution of terms for variables. Birkhoff [1] showed the soundness and completeness of equational systems.

The problem with equational systems from the computational standpoint is that their single rule of inference provides no strategy other than to exhaustively compare pairs of terms on either side of the axioms. In common practice, equations are efficiently used to simplify complex terms according to some ordering principle. For example, in verifying that

$$\sqrt{4! * 4 + 4} = \sum_{i=0}^{4} i$$

we first replace *definienda* by *definiens*, and then replace equals by equals in the direction of shorter terms until both sides are identical. This suggests that automated equational reasoning should use directed equations similarly to cut down search, as many systems do.

A **rewrite system** $\mathcal{R}$ over T is set of directed equations of the form $l \rightarrow r$, called **rewrite rules**. Directed equations are applied like undirected equations, but only left-hand sides are matched and replaced by right-hand sides only. More formally again, a rewrite system has this rule of inference:

$$\frac{\Xi(\tau[u\sigma]_p)\,,\; u \rightarrow v}{\Xi(\tau[v\sigma]_p)}$$

with Ξ and σ understood as before.

Not surprisingly, not just any rewrite system is complete in the sense that one can derive any valid consequence of the rules considered as undirected equations. There is, however, a family of rewrite systems that are not only sound and complete, but decidable, despite the fact that in general equational consequence is undecidable. To introduce them we need a bit of terminology first.

We write $s \rightarrow t$ for "s rewrites to t in a single step" and $s \overset{*}{\rightarrow} t$ for the reflexive, transitive closure of $\rightarrow$. $s \uparrow t$ is an abbreviation for $\exists u(u \rightarrow s \;\&\; u \rightarrow t)$: in other words, some term u diverges under rewriting. $s \downarrow t$ will likewise abbreviate convergence: $\exists u(s \rightarrow u \;\&\; t \rightarrow u)$. We shall indicate eventual convergence: $\exists u(s \overset{*}{\rightarrow} u \;\&\; t \overset{*}{\rightarrow} u)$ by $s \downarrow\!* t$, and eventual divergence:

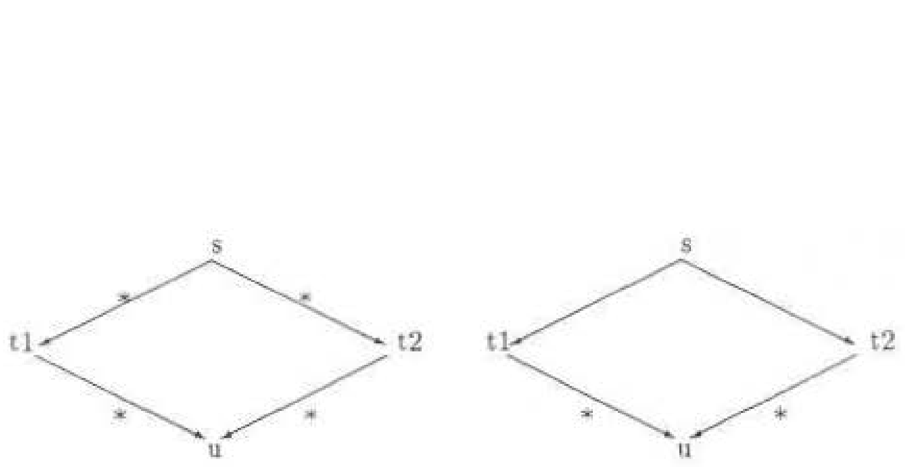

Figure 2: Confluence and Local Confluence

$\exists u(u \xrightarrow{*} s \,\&\, u \xrightarrow{*} t)$, by $s *\!\uparrow t$. We shall write $s \xrightarrow{!} t$ to indicate that t is a **normal form** of s , i.e., that $s \xrightarrow{*} t$ and $\neg \exists u(t \xrightarrow{*} u)$.

One way to decide whether an equation follows from a set of equations is to use that set to match and replace the terms in the target equation until either both sides of the target equation are the same, or else both sides are irreducibly distinct. If every term has a unique normal form, this procedure is complete. In a canonical rewrite system, every term has a unique normal form.

A **canonical** rewrite system $\mathcal{R}$ is one which is *finite*, *noetherian*, and *confluent*. A system of rewrite rules $\mathcal{R}$ is **noetherian** just in case every sequence of rewrites terminates; in other words, iff $\xrightarrow{*}$ is well-founded in $\mathcal{R}$. **Confluence**, sometimes called the *diamond* or *lattice* property, ensures that terms that diverge under rewriting eventually converge. In other words, $\mathcal{R}$ is confluent iff $\forall s, t\,(s *\!\uparrow t \supset s \downarrow\!* t)$. (See Figure 2.)

1.2 Noetherian Orderings

A binary relation $\succ$ on T is **monotonic** iff it has the *replacement property*, i.e.,

$\forall t, u \in T, \forall f \in F, t \succ u \supset f(\alpha_1, ..., \alpha_j, t, \alpha_k, ..., \alpha_n) \succ f(\alpha_1, ..., \alpha_j, u\, \alpha_k, ..., \alpha_n)$

It is **stable** (under substitution) iff

$$\forall t, u \in T,\ t \succ u \supset t\sigma \succ u\sigma$$

for any substitution σ of terms in T for variables. A monotonic partial ordering is a **simplification ordering** if it has in addition the *subterm property*:

$$\forall t \in T, \forall f \in F,\ f(...t...) \succ t$$

Dershowitz [2] proves the following theorem:

> A rewriting system $\{l_i \rightarrow r_i\}$ over a set of terms T is noetherian if there exists a stable simplification ordering $\succ$ over T such that $l_i \succ r_i$.

We shall call such a stable simplification ordering a **reduction** ordering.

For our proofs we used a recursive procedure which lexicographically orders terms in the following way. First a routine we'll call LEX attempts to order the terms on the following principles:

1. Variables are not ordered among themselves.

2. Compound terms[1] $\succ_{lex}$ atoms: e.g., $f(X,Y) \succ_{lex} a$.

3. Compound terms $\succ_{lex}$ subterms: e.g., $f(X, g(b,Y)) \succ_{lex} g(b,Y)$.

4. Two atoms are reverse lexicographically ordered by their names: e.g., $adam \succ_{lex} zoe$.[2]

5. Two compound terms are ordered by RANK.

6. Compound terms partially orderable by RANK are ordered by calling LEX on their subterms.

RANK orders terms in the following way. First unifiable pairs are rejected as non-orderable, since this would obviously lead to cycles. Non-unifiable terms are then ranked. $t1 \succ_{rank} t2$ iff:

> (i) $V_{t2} \subseteq V_{t1}$, where V_{t1}, V_{t2} are the multisets of variables in *t1* and *t2* respectively; and
> (ii) $w(t1) > w(t2)$, where $w(\tau)$ is a linear polynomial weighting of the functors in τ.

A pair of terms is **partially orderable** it meets (i) of RANK in one or both directions (i.e., the occurrences of variables on one side is a subset, not necessarily proper, of those on the other), but does not meet (ii) in either direction (i.e., the weight is the same on both sides). If the ordering reported by applying LEX to subterms conflicts with the direction of the partial ordering of RANK on the terms, the terms are not ordered by LEX.

[1] *viz.*, nonvariable terms that are neither numeric nor symbolic atoms, nor lists.

[2] Thus the user can give different weights to each identifier through careful naming.

Note that in the simplest case where all functors are weighted equally, this ranking orders from longer to shorter terms, according to the number of functors. In the ordering we use on the combinator problems reported here, all zero-place functors receive a weight of zero, and all other functors a weight of one. Thus

$$f(g(a,X,c),h(b,c,Y)) \succ g(a,b,c)$$

by a simple count of the functors, whereas

$$f(g(a,X,c),Y) \succ f(X,g(Y,b,c))$$

is determined by a recursive call to lexicographically order subterms.

$$f(X,g(a,Y,Y)),\ f(Y,g(b,X,X))$$

is incomparable since it fails condition (i) in both directions.

LEX/RANK is a reduction ordering since it fulfills the *subterm* condition by 3. of LEX and the *replacement* condition by ii of RANK.

1.3 Confluence and Local Confluence

A rewrite system $\mathcal{R}$ is **locally confluent** when terms that diverge in a single step eventually converge: $\forall s,t(s \uparrow t \supset s \downarrow\!* t)$. (See Figure 2.) Newman [3] proved that a noetherian relation is confluent iff it is locally confluent.
Theorem: If $\mathcal{R}$ is noetherian, then

$$\forall s,t\,[(s \uparrow t \supset s \downarrow\!* t) \text{ iff } (s{*}\!\uparrow t \supset s \downarrow\!* t)].$$

Proof ([4]): The 'if' direction is trivial. The proof of the 'only if' part uses a pair of inductions to complete the characteristic diamond shape of confluence (see Figure 3).

We assume that $\rightarrow$ is a noetherian, locally confluent relation. We show $\rightarrow$ to be confluent by noetherian induction. We assume it for everything less than x under $\rightarrow$; i.e., $\forall y(x \xrightarrow{+} y)$, where $\xrightarrow{+}$ is the transitive closure of $\rightarrow$, and show it for x.

If $x \rightarrow y$ or $x \rightarrow z$ then we are done by local confluence. Otherwise, split $x \xrightarrow{*} y$ into $x \rightarrow s$ and $s \xrightarrow{*} y$, and $x \xrightarrow{*} z$ into $x \rightarrow t$ and $t \xrightarrow{*} z$ as shown in the diagram. By local confluence, $\exists u(s \xrightarrow{*} u \;\&\; t \xrightarrow{*} u)$. Now two applications of the induction hypothesis do the trick: first to get us from y and u to v; and then from v and z to w. □

The necessity that the relation be noetherian is shown in Figure 4, which illustrates a locally confluent but non-noetherian relation which is obviously not confluent.

$$\begin{tikzcd}
 & & x \arrow[dl] \arrow[dr] & & \\
 & s \arrow[dl, "*"'] \arrow[dr, "*"] & \text{local conf.} & t \arrow[dl, "*"'] \arrow[dr, "*"] & \\
y \arrow[dr, "*"'] & \text{induction step} & u \arrow[dl, "*"] & \text{induction step} & z \arrow[ddll, "*"] \\
 & v \arrow[dr, "*"'] & & & \\
 & & w & &
\end{tikzcd}$$

Figure 3: Noetherian local confluence implies confluence

$$\begin{tikzcd}
u & s \arrow[l] \arrow[r, bend right] & t \arrow[l, bend right] \arrow[r] & v
\end{tikzcd}$$

Figure 4: Non-noetherian local confluence does not imply confluence.

1.4 Local Confluence and Critical Pairs

Donald Knuth and Peter Bendix [5] discovered a test for local confluence and a procedure for turning a set of equations into locally confluent (and ultimately canonical) rewrite rules, which they implemented and tried on several problems involving groups. Their basic discovery was that a noetherian rewrite system would be locally confluent iff every **critical pair** of terms reduced to the same normal form.

One way to ensure confluence in a rewrite system would be to equate the resulting terms each time a term had distinct rewrites. Given that the rewrite rules are congruence-preserving, such a procedure would clearly be sound. Unfortunately there are typically infinitely many such divergent pairs if there are any. Note however that this approach involves infinite duplication, since the set of divergent pairs includes infinite subsets of unifiable variants. The insight of Knuth and Bendix was to search for the source of divergence in the rewrite rules themselves, using most general unification to render the task finite.

A critical pair is generated when the left-hand sides of two rules overlap or **superpose**, so that one is unifiable with a non-variable subterm, not necessarily proper, of the other. The critical pair consists of the most general version of the terms that could be generated by such overlapping rules. For instance, the two rules

$$f(X, g(X, a)) \rightarrow h(X)$$

$$g(b, Y) \rightarrow i(Y)$$

superpose to yield the critical pair

$$< f(b, i(a)),\ h(b) >$$

More formally, suppose $\lambda_1 \rightarrow \rho_1,\ \lambda_2 \rightarrow \rho_2\ \in R$, and suppose further that $\lambda_1 = u[\lambda_2\sigma]_p$ for some most general unifier σ and some (non-variable) occurrence p. Then there is a critical pair $< u[\rho_2\sigma], \rho_1\sigma >$. For any pair of terms s, t such that $s \uparrow t$, there is a critical pair $< \alpha, \beta >$ and a unifier σ such that $s = \alpha\sigma$ and $t = \beta\sigma$ (or *vice versa*).

By equating critical pairs, we handle together unifiable classes of terms that would diverge, with great economy. Provided the original set of rules is finite, there will only finitely many critical pairs. Once all critical pairs are equated, local confluence is guaranteed.

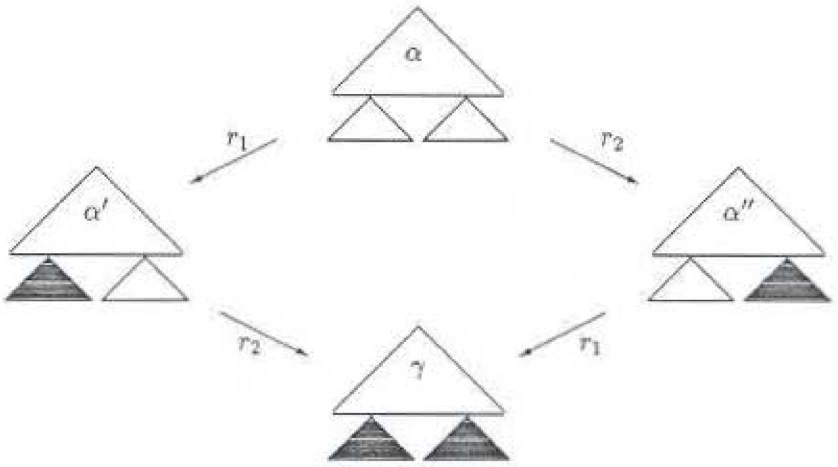

Figure 5: Disjoint Subterms Case

Theorem: A rewrite system $\mathcal{R}$ is locally confluent iff for every critical pair $< S, T >$, $S \downarrow * T$.

Proof ([5,6]):
Clearly $\mathcal{R}$ is not locally confluent if some critical pair does not converge. To show that critical pair convergence suffices for local confluence, we note that there are only three possible cases, depicted in the next three figures (adapted from [7]).

Suppose that a term α diverges under rewriting. Then there are subterms of α, say β_1 and β_2, so that $\alpha = u[\beta_1]_p = v[\beta_2]_q$for some contexts u and v, and positions p and q. Further there must be be rules r_1 and r_2 in $\mathcal{R}$ of the form $\lambda_1 \rightarrow \rho_1$, $\lambda_2 \rightarrow \rho_2$ and most general unifiers σ_1 and σ_2 such that $\beta_1 = \lambda_1\sigma_1$ and $\beta_2 = \lambda_2\sigma_2$. Applying r_1 or r_2 to α yields the divergent pair α', α'':

$$< u[\rho_1]_p \, , \, v[\rho_2]_q > .$$

Local confluence demands that $\alpha' \downarrow * \alpha''$. Call that necessary meeting place γ.

The simplest case is depicted in Figure 5. If β_1 and β_2 are disjoint subterms, then $r_1 r_2 \alpha = r_2 r_1 \alpha = w[\rho_1]_p[\rho_2]_q$. For example, if $r_1 = f(X, Y) \rightarrow X$, $r_2 = g(X, Y) \rightarrow Y$, and $\alpha = h(f(a, b), g(b, c))$, then $\alpha' = h(a, g(b, c))$ and $\alpha'' = h(f(a, b), c)$. So $\gamma = h(a, c)$.

If the two subterms are not disjoint, then we may suppose that β_1 is a subterm of β_2. There are two possibilities under this heading: either β_1 is contained in one of the terms substituted for variables in λ_2 by σ_2, or else it is unifiable with a non-variable subterm of λ_2.

The first possibility is depicted in Figure 6. Suppose both σ_1 and σ_2 replace some variable in α by τ. r_1 applied to $\alpha\sigma_1$ replaces an occurrence of

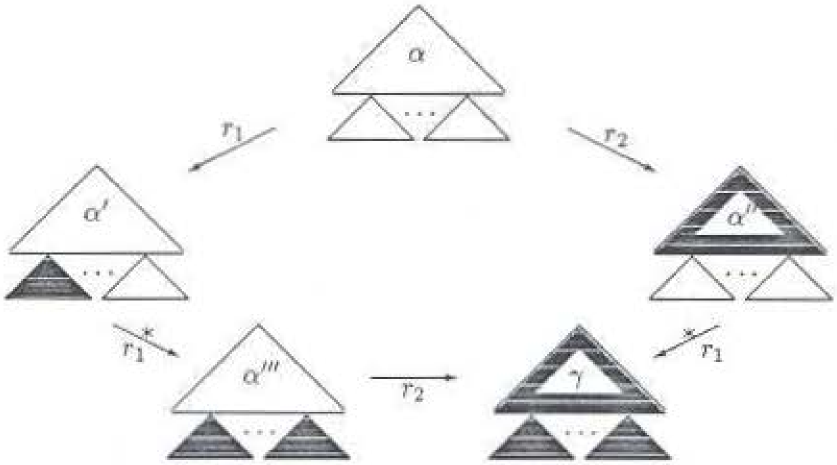

Figure 6: Variable Case

τ by τ'. Additional applications of r_1 will replace the other occurrences of τ in $\alpha_1\sigma_1$ by τ'. Call the expression obtained in this way α'''. Applying r_2 to α''' yields $(\alpha\sigma_2)_{\tau/\tau'}$, (i.e, $\alpha\sigma_2$ with τ' uniformly substituted for τ). This same term can be derived by first applying r_2 to $\alpha\sigma_2$, followed by as many applications of r_1 as are needed to replace all occurrences of τ by τ'.

For example, suppose $r_1 = n(X) \rightarrow -X$ and $r_2 = X * (Y + Z) \rightarrow (X * Y) + (X * Z)$. If $\alpha = 2 * (n(3) + n(4))$, then $\alpha' = 2 * (-3 + n(4))$ (or $2 * (n(3) + -4)$), and $\alpha'' = (2 * n(3)) + (2 * n(4))$. $\alpha''' = 2 * (-3 + -4)$, and $\gamma = (2 * -3) + (2 * -4)$.

The final possibility, depicted in Figure 7, is simply the critical pair case. Since we are given the convergence of critical pairs by hypothesis, we are done. As an example, let $r_1 = X + 0 \rightarrow X$, $X * (Y + Z) \rightarrow (X * Y) + (X * Z)$, and $\alpha = 2 * (1 + 0)$. Then $\alpha' = 2 * 1$ and $\alpha'' = (2 * 1) + (2 * 0)$. Given only these two rules, these two terms are irreducible.

1.5 Completion

The Knuth-Bendix completion procedure is simply to equate critical pairs and orient the resulting equations in accordance with some noetherian ordering. The enlarged set of rewrite rules is then checked for critical pairs once again, and the process iterates. There are three possible outcomes. If the procedure terminates, then the resulting rewrite system is locally conflu-

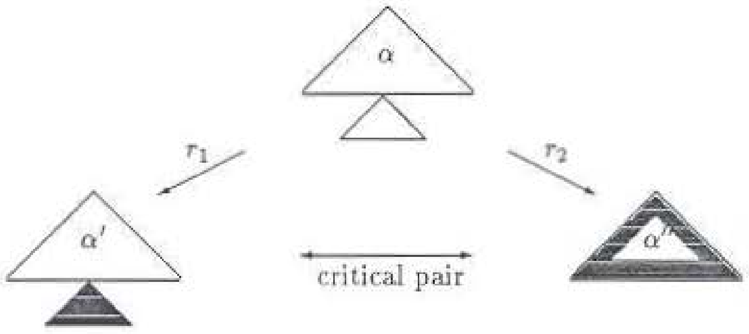

Figure 7: Critical Pair Case

ent, and in fact canonical, since the finiteness of the original set of equations and a noetherian rewrite relation are assumed by the procedure. The procedure may never terminate, with each addition to the set of rewrite rules generating another round of critical pairs. Or the procedure may abort, if a critical pair of terms is generated which is not orderable in either direction.

Thus the Knuth-Bendix completion procedure assumes a noetherian ordering on T. If the ordering is not total, the procedure may generate a non-orientable critical pair and so fail. This problem may be resolved by *ad hoc* extensions to the ordering to handle incommensurable terms as they appear, but this tactic risks rendering the ordering non-noetherian. In fact, even given an ordering total on ground terms, it is not always possible to order terms with variables. The approach we have followed is that of "unfailing completion" as described in [8]. Non-orientable equations are kept in the database and used bidirectionally for the generation of critical pairs in the manner of paramodulation.

Following [8], we present the completion algorithm abstractly as an inference system for a set of equations E and a set of rewrite rules R. Assume here that $\succ$ is a reduction ordering on T. First the basic Knuth-Bendix procedure is characterized by the following rules:

C1: Orienting an equation

$$\frac{E \cup \{s \doteq t\},\ R}{E, R \cup \{s \rightarrow t\}} \text{ if } s \succ t$$

C2: Equating a critical pair

$$\frac{E,\ R}{E \cup \{s = t\}, R} \text{ if } s \uparrow_R t$$

C3: Simplifying an equation

$$\frac{E \cup \{s \doteq t\},\ R}{E \cup \{u \doteq t\}, R} \text{ if } s \to_R u$$

C4: Deleting a trivial equation

$$\frac{E \cup \{s = s\},\ R}{E,\ R}$$

The next rules do not affect the final outcome of the Knuth-Bendix procedure, but are practically necessary for efficiency. They simplify rewrite rules so that right-hand sides are in normal form relative to all the rules derived so far, and the left-hand sides in normal form relative to all rules derived so far except, naturally, those in which they occur.

S1: Simplifying the right-hand side of a rewrite rule

$$\frac{E,\ R \cup \{s \to t\}}{E,\ R \cup \{s \to u\}} \text{ if } t \to_R u$$

S2: Simplifying the left-hand side of a rewrite rule

$$\frac{E,\ R \cup \{s \to t\}}{E \cup \{u = t\},\ R}$$

if $l \to r \in R, s = v[l\sigma]_p, u = v[r\sigma]_p$; or $l \to r \in R$ and $s \triangleright l$. $\alpha \triangleright \beta$ signifies α is a **proper instance** of β: $\exists \sigma\, (\alpha = \beta\sigma)\ \&\ \neg\exists\, \sigma\, (\beta = \alpha\sigma)$.

The intent of **S2** is to delete subsumed instances of rewrite rules. The restrictions are necessary to distinguish these cases from those falling under **C2**: non-subsumed overlapping rules should not be dropped from R!

Standard completion fails when an equation can neither be simplified by **C3** or **C4**, nor oriented by **C1**. Naturally this can be avoided if $\succ$ is total, but this is too much to expect in general. Commutativity is an often indispensible axiom that cannot be oriented by any reduction ordering. Various approaches for rewriting and unification in associative-commutative systems have been suggested: eg, [9,10,11]. None of these approaches completely excludes the possibility of failure due to incomparable terms, as does the approach sketched next.

We obtain unfailing completion by adding the inference rule:

C5: Equating a critical pair.

$$\frac{E,\ R}{E \cup \{s = t\},\ R} \text{ if } s \uparrow_{E^{=} \cup R} t$$

together with the following rules for simplification:
S3: Simplifying the right-hand side

$$\frac{E,\ R \cup \{s \to t\}}{E,\ R \cup \{s \to u\}} \text{ if } t \to_{\vec{E}} u$$

S4: Simplifying the left-hand side

$$\text{a)}\frac{E \cup \{s \doteq t\},\ R}{E \cup \{u \doteq t\},\ R}$$

$$\text{b)}\frac{E,\ R \cup \{s \to t\}}{E \cup \{u = t\},\ R}$$

if $l \to r \in \vec{E}, s = v[l\sigma]_p, u = v[r\sigma]_p$; or $l = r \in E$ and $s \triangleright l$.

Inference rule **C5** subsumes **C2** as a special case. $E^=$ signifies the rules generated by the symmetric closure of E. Unfailing completion superposes not only the left-hand sides of rewrite rules with one another to generate critical pairs, but also the left-hand sides of rewrite rules with either side of equations, and equations with equations. **S3** and **S4** extend **S1** and **S2** by allowing orientable instances of equations to be used as rewrite rules. $\alpha\sigma = \beta\sigma$ is an **orientable instance** of $\alpha = \beta$ if $\alpha\sigma \succ \beta\sigma$. $\vec{E}$ denotes the rewrite system consisting of all orientable instances of equations in E.

Essentially, when faced with a non-orientable equation, unfailing completion falls back on a paramodulation strategy for the determination of critical pairs. The completeness of this strategy depends ultimately on Birkhoff's result. While giving up nothing in completeness, however, non-oriented equations are expensive computationally and should be avoided if possible. The use of orientable instances in the simplification rules is an attempt to reap some of the advantages of direction from non-orientable equations. For example, the two equations $X * Y = Y * X$ and $(X * Y) * Z = (Y * X) * Z$ are not orientable. Nevertheless, using them we can rewrite ground terms, e.g., $((((a*b)*c)*d)*e) \xrightarrow{*} (e*(d*(c*(b*a))))$, using lexicographic reverse alphabetical ordering. See Figure 8.

The soundness and completeness of unfailing completion are proved in [8].

1.6 Two Examples of Completion

To see how standard completion works, consider the following definition of a group:

$$\begin{array}{ll}
((((a*b)*c)*d)*e) \rightarrow ((((b*a)*c)*d)*e) & [(X*Y)*Z=(Y*X)*Z] \\
((((b*a)*c)*d)*e) \rightarrow (((c*(b*a))*d)*e) & [(X*Y)*Z=(Y*X)*Z] \\
(((c*(b*a))*d)*e) \rightarrow ((d*(c*(b*a)))*e) & [(X*Y)*Z=(Y*X)*Z] \\
((d*(c*(b*a)))*e) \rightarrow (e*(d*(c*(b*a)))) & [X*Y=Y*X]
\end{array}$$

Figure 8: Rewriting instances of unorientable equations.

$$\begin{aligned}
1*x &= x \\
x*x^{-} &= 1 \\
(x*y)*z &= x*(y*z)
\end{aligned}$$

We shall trace in part the completion sequence dictated by the rules **C1** - **C4**, under the lexicographic ordering where $* \succ -$. The results appear in Figure 9.

We start by directing the three equations in the order listed. Note that the lexicographic ordering has the effect of moving parentheses to the right in the association axiom, since $(x*y) \succ x$. The first critical pair added comes from superposing the first two rewrite rules; the second from superposing the rule just generated with the second rule. In general, we have followed such a linear or depth-first strategy. The third critical pair arises from the overlap between rules 2 and 3, and the next two by superposing the rule just added with 2 again. The sixth critical pair is generated by superposing the previous rule with rule 5. The last critical pair results from superposing the rule before the immediately preceding rule with rule 3.

In a straightforward manner we have derived a nearly complete system of rewrite rules. Only one rule is missing, $(x*y)^{-} \rightarrow y^{-} * x^{-}$. This is a bit more complicated, requiring some intermediate rules. See [5] for one derivation.

We illustrate unfailing completion with the axioms for an associative-commutative ring:

$$\begin{aligned}
x+y &= y+x \\
x*y &= y*x \\
(x+y)+z &= x+(y+z) \\
(x*y)*z &= x*(y*z) \\
x+0 &= x \\
x+i(x) &= 0 \\
x*(y+z) &= (x*y)+(x*z) \\
(y+z)*x &= (y*x)+(z*x)
\end{aligned}$$

Note that no procedure will succeed in orienting the symmetry axiom.

i	E_i	R_i	rule
0	$1 * x = x$ $x * x^- = 1$ $(x * y) * z = x * (y * z)$		
1	$x * x^- = 1$ $(x * y) * z = x * (y * z)$	$R_0 +$ $1 * x \to x$	**C1**
2	$(x * y) * z = x * (y * z)$	$R_1 +$ $x * x^- \to 1$	**C1**
3		$R_2 +$ $(x * y) * z \to x * (y * z)$	**C1**
4	$1^- = 1$	R_3	**C2**
5		$R_3 +$ $1^- \to 1$	**C1**
6	$1 * 1 = 1$	R_5	**C2**
7	$1 = 1$	R_5	**C3**
8		R_5	**C4**
9	$1 * z = x * (x^- * z)$	R_5	**C2**
10	$z = x * (x^- * z)$	R_5	**C3**
11		$R_5 +$ $x * (x^- * z) \to z$	**C1**
12	$x * 1 = (x^-)^-$	R_{11}	**C2**
13	$x = (x^-)^-$	R_{11}	**C3**
14		$R_{11} +$ $(x^-)^- \to x$	**C1**
15	$x^- * x = 1$	R_{14}	**C2**
16		$R_{14} +$ $x^- * x \to 1$	**C1**
17	$x * 1 = x$	R_{16}	**C2**
18		$R_{16} +$ $x * 1 \to x$	**C1**
19	$1 * z = x^- * (x * z)$	R_{18}	**C2**
20	$z = x^- * (x * z)$	R_{18}	**C3**
21		$R_{18} +$ $x^- * (x * z) \to z$	**C1**

Figure 9: Partial completion of group axioms

In Figure 10, the beginning of a completion sequence is illustrated. We use the lexicographic ordering described earlier. The initial orientation steps are combined to save space. The first two critical pairs produced overlap the first non-orientable equation with the third and fourth rewrite rules. The last critical pair is simplified and oriented to give us the closing rewrite rule.

2 Application

2.1 Integrating Completion and Proving

We get the effect of unfailing completion by arranging the input to a prover implemented in Prolog by David Plaisted, based on his simplified problem reduction format, called sprfn [14]. The crux of the arrangement is to induce the addition of critical pairs to the equational database.

sprfn may be viewed as an extension to Prolog with true (sound) unification and negation, a complete search strategy (iterative deepening) and caching of intermediate results. Alternatively, it may be seen as a theorem-prover that takes advantage of the built-in unification and back-chaining of Prolog to achieve respectable results in a relatively short and comprehensible piece of code. sprfn inserts a limited forward-chaining phase every time the depth-bound is increased and the problem restarted. Solutions obtained in either phase are available in both. Given a set of rewrite rules, sprfn does automatic rewriting of terms at the end of each forward-chaining phase. Using the Prolog-like interface of sprfn, it is possible to represent equational problems in a way that causes the prover to simulate the Knuth-Bendix completion procedure. In this paper we compare these representations with the standard axiomatic representations of equality on several problems involving combinators.

A feature of our the strategy is that we do not first compile a complete system of rewrite rules and then attempt to prove a theorem, but rather interleave completion and proof steps. Once we have generated enough rewrite rules to prove the theorem we are done, or so we hope.

Most of the rules of unfailing completion are built into sprfn. **C1** is taken care of either interactively by the user or by an automated orienter. In the results to be described, we use the lexicographic orienter described earlier. **C2** is subsumed by **C5**, as we have said. **C3** is taken care of automatically during the rewrite phase. Simplification using orientable instances as in **S3** and **S4** is again provided by the rewriting part of sprfn. This leaves **C4** and **C5** to be taken care of in the input. **C4** is satisfied by adding a reflexivity

i	E_i	R_i	rule
	$x + y = y + x$		
	$x * y = y * x$		
	$(x + y) + z = x + (y + z)$		
0	$(x * y) * z = x * (y * z)$		
	$x + 0 = x$		
	$x + i(x) = 0$		
	$x * (y + z) = (x * y) + (x * z)$		
	$(y + z) * x = (y * x) + (z * x)$		
	$x + y = y + x$		
	$x * y = y * x$		
1,2	$x + 0 = x$	$R_0 +$	
	$x + i(x) = 0$		
	$x * (y + z) = (x * y) + (x * z)$	$(x + y) + z \rightarrow x + (y + z)$	C1
	$(y + z) * x = (y * x) + (z * x)$	$(x * y) * z \rightarrow x * (y * z)$	C1
	$x + y = y + x$		
3,4	$x * y = y * x$	$R_2 +$	
	$x * (y + z) = (x * y) + (x * z)$	$x + 0 \rightarrow x$	C1
	$(y + z) * x = (y * x) + (z * x)$	$x + i(x) \rightarrow 0$	C1
		$R_4 +$	
5,6	$x + y = y + x$	$(x * y) + (x * z) \rightarrow x * (y + z)$	C1
	$x * y = y * x$	$(y * x) + (z * x) \rightarrow (y + z) * x$	C1
7	$0 + x = x$	R_6	C5
		$R_6 +$	
8		$0 + x \rightarrow x$	C1
9	$i(x) + x = 0$	R_8	C5
		$R_9 +$	
10		$i(x) + x \rightarrow 0$	C1
11	$i(0) + 0 = 0$	R_{10}	C5
12	$i(0) = 0$	R_{10}	S4a
		$R_{10} +$	
13		$i(0) \rightarrow 0$	C1

Figure 10: Partial completion of associative-commutative ring axioms

axiom to our representations, which will subsume any instances of $X = X$. The principal job of the input representation is the simulation of **C5**, which adds the critical pairs to the database.

2.2 The Critical Pair Representation

In this representation, a rule with a critical-pair equation as its head is introduced for each occurrence of a non-variable subterm in the terms of the input. The Knuth-Bendix conditions for the generation of critical pairs appear in the body. Because this representation contains rules solely to introduce and equate critical pairs, we call it the **critical pair** representation.

For example, the b combinator, which given three functions returns the composition of the first and the second applied to the third, is defined as follows:

$$\mathsf{b}xyz = x(yz).$$

Represented in first-order terms, this becomes

$$\mathrm{a(a(a(b,X),Y),Z) = a(X,a(Y,Z))}$$

using a for functional application. We depict this equation in tree form in Figure 11.

In this forest, there are five distinct positions occupied by non-variable terms that may overlap with other terms. In practice, we ignore those positions occupied by combinators, since they will be irreducible on any reasonable ordering. Those positions rooted in an occurrence of a, the root position and postions 1, 2, and 11, are left as possible trouble spots. For each one, we construct a rule to introduce the appropriate critical pair into the equation database. See Figure 12.

In fact, there seems to be no reason to think that critical pair rules for each non-variable occurrence in the input would be sufficient, given that new and deeper terms will ordinarily be generated and superposed as the critical pair processes iterates. Since sprfn tends to be distracted by too many available inferences, we thought it would be best to start with a minimal number of critical pair rules, and add to them as needed. In practice, just the critical pair rules tailored to the input has proven sufficient for theorem-proving. (That is, anything unprovable with this basic set of critical pair rules is still unprovable with an augmented set.) More work needs to be done on the practical and theoretical sufficiency of this or any set of occurrence-specific critical pair rules.

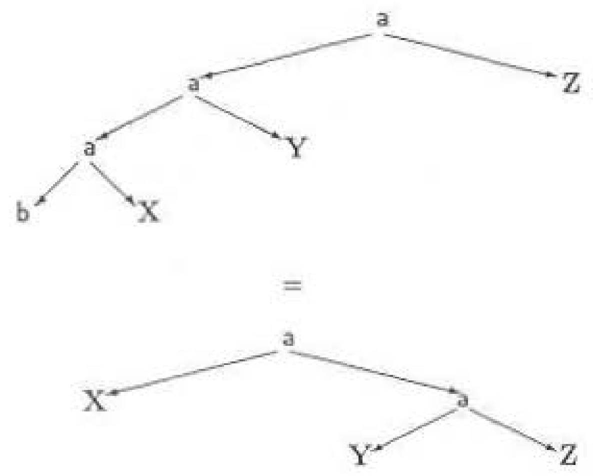

Figure 11: b$xyz = x(yz)$

These rules exhibit the Prolog-like interface to sprfn, including one misleading resemblance: '=' here stands for the relation of equality, not the operation of unification, as in Prolog. Note that the rules for introducing critical pairs search only the equations, because only terms in equations are accessible for unification. It is vital then that all rewrite rules be represented also by equations. In addition, equations in the input will not be oriented by sprfn unless they are independently derived. Thus it is important that the input file contain an equation for every rewrite rule, and a rewrite rule for every orientable equation. sprfn will preserve this parity, but the user is responsible for the input.

Calls to Prolog restrict these rules to the forward-chaining phase. The equation of critical pairs is not an appropriate goal for the prover. If the body of a critical pair rule is satisfied, then the equation will be generated; but no subgoals will be added by these rules.

2.3 The Modification Representation

In our first representation, declarations of definitions and axioms were separate from the procedures for generating and equating critical pairs. In our next representation, they are combined, and with this multiple cases

```
        % root position
V = U :-                        % critical pair to be equated
    prolog(f_chaining),         % keep completion in forward phase
    X = U,                      % equation in database contains a
    prolog(nonvar(X)),          % non-variable subterm which is
    X = V,                      % also a term in another equation
    prolog((X \== V)),          % not syntactically identical
    prolog((V \== U)).          % critical pair non-trivial

        % 1 position
a(V,Y) = U :-
    prolog(f_chaining),
    a(X,Y) = U,
    prolog(nonvar(X)),
    X = V,
    prolog((X \== V)),
    prolog((a(V,Y) \== U)).

        % 2 position
a(X,V) = U :-
    prolog(f_chaining),
    a(X,Y) = U,
    prolog(nonvar(Y)),
    Y = V,
    prolog((Y \== V)),
    prolog((a(X,V) \== U)).

        % 11 position
a(a(V,Y),Z) = U :-
    prolog(f_chaining),
    a(a(X,Y),Z) = U,
    prolog(nonvar(X)),
    X = V,
    prolog((X \== V)),
    prolog((a(a(V,Y),Z) \== U)).
```

Figure 12: Critical pair rules required for b combinator

```
B1 = B2 :- prolog(f_chaining),
        a(b,X) = U, a(U,Y) = V, a(V,Z) = B1,       % B1 = Bxyz
        a(Y,Z) = W, a(X,W) = B2.                   % B2 = x(yz)
```

Figure 13: Definition of b combinator in modification representation

of superposition are combined as well. Because our strategy for representing equality resembles that of [12] (*cf.* [13]), we call this the **modification** representation.

In this representation, each non-variable subterm in the term being defined is "pulled out" and equated with a variable, much like the usual Prolog representation of functional application. If this chain of equations exists in the database, ultimately equating the left-hand side of an equation to say, $B1$ and the right-hand side to $B2$, then the equation $B1 = B2$ is added to the database.

As an example, Figure 13 gives the definition of the b combinator. The various restrictions of the previous procedure have been dropped here: any of the variables may be bound to variables or to terms that are syntactically identical with other terms in the clause. These differences arise from the fact that this rule is intended to define the b combinator as well as express the conditions for the addition of critical pairs. Since the rules serve to define the combinators as well as to introduce critical pairs, it is less than obvious they should be restricted to the forward-chaining phase, but in fact the prover performed much better on our examples with these guards installed, as opposed to no guards, or back-chaining guards.

The pertinent difference between this representation and the last is that this representation telescopes multiple subterm replacements at different positions and adds the pair $s = t$ directly, rather than working at one position at a time, and adding first the equations $u = s$ and $u = t$, and only then $s = t$.

2.4 An Example Proof

As an example proof we choose the derivation a fixed-point combinator from three other combinators: a composition combinator: $\mathsf{b}xyz = x(yz)$; and two repeating combinators: $\mathsf{l}xy = x(yy)$, $\mathsf{m}x = xx$. This problem appears in colorful guise in [15], where the combinators are given the names of birds. In

Smullyan's vocabulary, our problem is to derive a *sage* bird from a *bluebird*, a *lark*, and a *mockingbird*. The essential insight is that it follows immediately from the definition of the LARK, $lxy = x(yy)$, that $lxlx = x(lxlx)$ for any x; i.e., that $lxlx$ is a fixed-point for any x.

2.4.1 No Rewrite Version

Our first representation uses equality axioms and no rewrite rules:

```
%%% Bluebird + Lark + Mockingbird ==> Sage (tmm, p. 91)

% definition of SAGE BIRD: Yx = xYx
false :- prolog(\+ f_chaining),
         eq(a(Y,f(Y)), B), eq(a(f(Y),B), B).

% definition of MOCKINGBIRD: Mx = xx
        eq(a(X,X), a(m,X)).

% definition of LARK: Lxy = x(yy)
        eq(a(X,a(Y,Y)), a(a(l,X),Y)).

% definition of BLUEBIRD: Bxyz = x(yz)
        eq(a(a(a(b,X),Y),Z), a(X,a(Y,Z))).

%%% equality

        % equivalence relation
eq(X, X).
eq(X, Y) :- prolog(f_chaining), eq(Y, X).
eq(X, Z) :- prolog(\+ f_chaining),
    eq(X,Y), prolog(X \== Y),
    eq(Y,Z), prolog(Y \== Z).

        % replacement rules
eq(U,V) :- prolog(\+ f_chaining),
           prolog((nonvar(U) ; nonvar(V))),
           replaceq(U,V).
replaceq(a(T1,T2),a(T3,T4)) :-
    prolog(\+ f_chaining), eq(T1,T3), eq(T2,T4).
replaceq(f(T1),f(T2)) :-
    prolog(\+ f_chaining), eq(T1,T2).
```

We use eq to represent equality here, since sprfn is designed to automatically rewrite using terms related by =. The existence of a fixed-point combinator is denied and this denial made the first subgoal for our top-level goal **false**. Then the three input combinators are defined. The axioms defining **eq/2** as an equivalence relation and the the rules allowing replacement of equal subterms while preserving equality are standard except for the introduction of various guards, via calls to the Prolog interpreter, attempting to control their use. Both the transitivity axiom and the replacement rules are restricted to the back-chaining phase. Syntactically identical terms would obviously not be useful subgoals for the the transitivity axiom to generate. Again, if both `U` and `V` are variables, then they should be shown equal through unification and the application of the reflexivity axiom, not through the replacement rules.

Given this input file, sprfn produces the following output:

```
solution_size_mult(0.1) is asserted
proof_size_mult(1) is asserted
clause_count_mult(0) is asserted
.
.
.
.
proof found
false:-
    prolog(\+ f_chaining)
    eq(a(a(a(b,m),1),f(a(a(b,m),1))),
       a(a(1,f(a(a(b,m),1))),a(1,f(a(a(b,m),1))))) :-
         prolog(\+f_chaining)
         input(eq(a(a(a(b,m),1),f(a(a(b,m),1))),
                  a(m,a(1,f(a(a(b,m),1))))))
         prolog(a(a(a(b,m),1),f(a(a(b,m),1)))
                \== a(m,a(1,f(a(a(b,m),1)))))
         eq(a(m,a(1,f(a(a(b,m),1)))),
            a(a(1,f(a(a(b,m),1))),a(1,f(a(a(b,m),1))))) :-
             prolog(f_chaining)
             input(eq(a(a(1,f(a(a(b,m),1))),a(1,f(a(a(b,m),1)))),
                      a(m,a(1,f(a(a(b,m),1))))))
         prolog(a(m,a(1,f(a(a(b,m),1)))) \==
                a(a(1,f(a(a(b,m),1))),a(1,f(a(a(b,m),1)))))
   input(eq(a(f(a(a(b,m),1)),a(a(1,f(a(a(b,m),1))),
            a(1,f(a(a(b,m),1))))),
           a(a(1,f(a(a(b,m),1))),a(1,f(a(a(b,m),1))))))
size of proof 11
clause count 6
72.0333 cpu seconds used
```

```
240 inferences done
```

The assertions reported are parameters for weighting the various factors that make up the cost of a particular proof subtree: length of the solution, depth of the tree, and the number of nodes. Varying these weights can help or hinder the prover in its search for a proof.

Each dot marks the restarting of sprfn with a greater depth bound. At the end we are told that this proof was found at depth 11, and that the proof tree has 6 nodes. (calls to Prolog are not counted as nodes in the proof tree.) As usual with UNIX statistics, cpu time is accurate only to within ± 10 % or so. The number of inferences represents each time an old solution or input clause was used.

The proof is simply a report of the subgoals generated from the top-level goal, `false`. This generates three subgoals: a call to Prolog, and two equations. The second equation turns out to be that instance of the definition of the LARK combinator showing $lxlx$ to be a fixed point. The first matches the head of the transitivity rule and generates five further subgoals. The first equation in these subgoals is an instance of the definition of the BLUEBIRD combinator, and the second an instance of the definition of the MOCKINGBIRD combinator, once it is turned around through an appeal to symmetry.

2.4.2 Critical Pair Representation

The critical pair representation of the theorem is as we have described it, with the addition of the equality axioms and a rewrite rule. Of those axioms defining equality as an equivalence relation, only reflexivity is demanded by the unfailing completion inference system. The symmetry axiom is added to get the effect of $\doteq$ in **C1** and **C3**, and also $E^{=}$ in **C5**: in other words, to allow the equations in the database to be used in either direction. Incidentally, equations are stored as directed left-to-right (but not as rewrite rules), so it is helpful if all equations in the input have the more complex term on the left. The `f_chaining` guard is an attempt to control the application of the symmetry axiom, by restricting it to the completion phase. Transitivity is not needed for completion, but is required for the completeness of the prover. Hence it is restricted to the back-chaining phase, outside of completion. Finally the rewrite rule is necessary to compensate for sprfn's quirk: it treats all input as "old solutions", and so will not attempt to convert input equations into rewrite rules.

```
% Bluebird + Lark + Mockingbird => Sage (tmm, p. 91)
% definition of SAGE BIRD: Yx = xYx
false :-
        prolog(\+ f_chaining),
        a(Y,f(Y)) = B, a(f(Y),B) = B.
% definition of MOCKINGBIRD
        a(X,X) = a(m,X).
% definition of LARK
        a(X,a(Y,Y)) = a(a(l,X),Y).
% definition of BLUEBIRD
        a(a(a(b,X),Y),Z) = a(X,a(Y,Z)).
%%% critical pairs
        % root position
V = U :-
        prolog(f_chaining),
        X = U,
        prolog(nonvar(X)),
        X = V,
        prolog((X \== V)),
        prolog((V \== U)).
        % 1 position
a(V,Y) = U :-
        prolog(f_chaining),
        a(X,Y) = U,
        prolog(nonvar(X)),
        X = V,
        prolog((X \== V)),
        prolog((a(V,Y) \== U)).
        % 2 position
a(X,V) = U :-
        prolog(f_chaining),
        a(X,Y) = U,
        prolog(nonvar(Y)),
        Y = V,
        prolog((Y \== V)),
        prolog((a(X,V) \== U)).
        % 11 position
a(a(V,Y),Z) = U :-
        prolog(f_chaining),
        a(a(X,Y),Z) = U,
        prolog(nonvar(X)),
        X = V,
        prolog((X \== V)),
        prolog((a(a(V,Y),Z) \== U)).
```

```
%%% equality
X = X.
X = Y :- prolog(f_chaining), Y = X.
X = Z :- prolog(\+ f_chaining), prolog(nonvar(X)),
         X = Y, prolog(nonvar(Y)), prolog(X \== Y),
         Y = Z, prolog(Y \== Z).
%%% orientable equations
rewrite(a(a(a(b,X),Y),Z), a(X,a(Y,Z))).
```

Note that the statement of the theorem and the definition of the combinators is the same as in the no-rewrite version, except that now '=' is used to trigger the automated rewriting capabilites of sprfn.

Given the preceding input, and using the lexicographic ordering in which functors are weighted alphabetically, sprfn returns the following:

```
solution_size_mult(0.1) is asserted.
proof_size_mult(1) is asserted.
orient is asserted.
.
a(a(l,a(X,X)),X)->a(m,a(X,X))
a(X,a(Y,a(a(b,X),Y)))->a(m,a(a(b,X),Y))
a(a(l,X),Y)->a(X,a(m,Y))
a(a(a(m,b),X),Y)->a(b,a(X,Y))
a(m,X)=a(X,X)
a(X,a(m,Y))=a(X,a(Y,Y))
a(a(X,X),a(m,X))->a(m,a(X,X))
a(X,a(Y,Y))=a(X,a(m,Y))
.
a(X,a(Y,a(Z,Z)))=a(X,a(Y,a(m,Z)))
a(a(X,X),a(X,X))->a(m,a(m,X))
a(X,a(Y,a(m,Z)))=a(X,a(Y,a(Z,Z)))
a(X,a(a(m,Y),Z))=a(X,a(a(Y,Y),Z))
a(a(m,a(b,X)),Y)=a(X,a(a(b,X),Y))
a(a(m,X),a(Y,Z))=a(a(X,X),a(Y,Z))
.
a(a(m,X),a(X,X))->a(m,a(m,X))
a(a(m,X),a(m,X))->a(m,a(X,X))
a(X,a(m,a(l,X)))->a(m,a(l,X))
a(a(m,l),X)->a(l,a(m,X))
a(X,a(a(Y,Y),Z))=a(X,a(a(m,Y),Z))
a(a(X,X),a(Y,Z))=a(a(m,X),a(Y,Z))
a(X,a(a(b,X),Y))=a(a(m,a(b,X)),Y)
.
```

```
a(b,a(X,a(a(m,b),X)))->a(m,a(a(m,b),X))
a(X,a(a(b,X),a(m,a(b,X))))->a(m,a(a(b,X),a(b,X)))
a(a(m,a(X,X)),a(m,a(X,X)))->a(m,a(m,a(m,X)))
a(a(m,a(m,b)),X)->a(b,a(a(m,b),X))
a(a(m,a(b,b)),X)->a(b,a(a(m,b),X))
a(a(m,X),a(m,Y))=a(a(X,X),a(Y,Y))
.
proof found
false:-
    prolog(\+f_chaining)
    input(a(a(a(b,m),l),f(a(a(b,m),l)))
          =a(m,a(l,f(a(a(b,m),l)))))
    a(f(a(a(b,m),l)),a(m,a(l,f(a(a(b,m),l)))))
      =a(m,a(l,f(a(a(b,m),l)))):-
        prolog(f_chaining)
        input(a(a(l,f(a(a(b,m),l))),a(l,f(a(a(b,m),l))))
              =a(m,a(l,f(a(a(b,m),l)))))
        prolog(nonvar(a(a(l,f(a(a(b,m),l))),
                      a(l,f(a(a(b,m),l))))))
        a(f(a(a(b,m),l)),a(m,a(l,f(a(a(b,m),l)))))
          =a(a(l,f(a(a(b,m),l))),a(l,f(a(a(b,m),l)))):-
            prolog(f_chaining)
            input(a(f(a(a(b,m),l)),a(a(l,f(a(a(b,m),l))),
                  a(l,f(a(a(b,m),l)))))
                  =a(a(l,f(a(a(b,m),l))),a(l,f(a(a(b,m),l)))))
            prolog(nonvar(a(a(l,f(a(a(b,m),l))),
                          a(l,f(a(a(b,m),l))))))
            input(a(a(l,f(a(a(b,m),l))),a(l,f(a(a(b,m),l))))
                  =a(m,a(l,f(a(a(b,m),l)))))
            prolog(a(a(l,f(a(a(b,m),l))),a(l,f(a(a(b,m),l))))
                  \==a(m,a(l,f(a(a(b,m),l)))))
            prolog(a(f(a(a(b,m),l)),a(m,a(l,f(a(a(b,m),l)))))
                  \==a(a(l,f(a(a(b,m),l))),a(l,f(a(a(b,m),l)))))
        prolog(a(a(l,f(a(a(b,m),l))),a(l,f(a(a(b,m),l))))
              \==a(f(a(a(b,m),l)),a(m,a(l,f(a(a(b,m),l))))))
        prolog(a(f(a(a(b,m),l)),a(m,a(l,f(a(a(b,m),l)))))
              \==a(m,a(l,f(a(a(b,m),l)))))
size of proof 14
clause count 7
210.533 cpu seconds used
633 inferences done
```

Each equation has been added by a critical pair rule, and oriented by the automated lexicographic orienter if possible. As before, the top-level goal `false` generates three subgoals. The first is an instance of the definition of the MOCKINGBIRD. The second matches the root position critical pair rule, and so generates 6 further subgoals of its own. The first necessary equation is again an instance of the MOCKINGBIRD definition, and the second again matches a critical pair rule, this time at position 2. (It helps to identify the `V` and `U` terms through the last two Prolog identity checks.) The first equation required by this rule represents sprfn's insight that $lxlx$ is a fixed-point for any x, and is accepted as an instance of the definition of LARK. The next is yet another instance of the MOCKINGBIRD. The remaining subgoals are simply to ensure that the terms are not identical.

2.4.3 The Modification Representation

The modification representation follows the previous the previous ones, except that now the definitions of the combinators have been "pulled-out." There are no occurrence-specific critical pair rules.

```
%%% Bluebird + Lark + Mockingbird ==> Sage
% Definition of SAGE bird: Yx = xYx
false :- prolog(\+ f_chaining),
         a(Y,f(Y)) = B, a(f(Y),B) = B.
% definition of BLUEBIRD : Bxyz = x(yz)
B1 = B2 :-
        prolog(f_chaining),
        a(b,X) = U, a(U,Y) = V, a(V,Z) = B1,
        a(Y,Z) = W, a(X,W) = B2.
% definition of MOCKINGBIRD : Mx = xx
B1 = B2 :-
        prolog(f_chaining),
        a(m,X) = B1, a(X,X) = B2.
% definition of LARK : Lxy = x(yy)
B1 = B2 :-
        prolog(f_chaining),
        n(l, X) = U, n(U, Y) = B1,
        n(Y, Y) = V, n(X, V) = B2.
%%% Equality
X = X.
X = Y :- prolog(f_chaining), Y = X.
X = Z :- prolog(\+ f_chaining), prolog(nonvar(X)),
         X = Y, prolog(nonvar(Y)), prolog(X \== Y),
         Y = Z, prolog(Y \== Z).
```

Given this file and the same orienter again, we get the following output. Note that sprfn first tries to orient the equations it constructs from the input definitions.

```
solution_size_mult(0.1) is asserted.
proof_size_mult(1) is asserted.
orient is asserted.
.
a(m,X)=a(X,X)
.
a(a(1,X),Y)=a(X,a(Y,Y))
a(X,X)=a(m,X)
.
a(a(a(b,X),Y),Z)->a(X,a(Y,Z))
a(X,a(Y,Y))=a(a(1,X),Y)
.
a(X,a(a(1,X),a(1,X)))->a(m,a(1,X))
a(a(X,X),a(X,X))->a(a(1,m),X)
.
a(X,a(m,Y))=a(X,a(Y,Y))
a(a(1,a(X,X)),X)->a(m,a(X,X))
a(a(m,1),X)=a(1,a(X,X))
a(a(1,X),Y)->a(X,a(m,Y))
a(a(X,X),a(m,X))->a(m,a(X,X))
a(X,a(m,Y))=a(X,a(Y,Y))
a(X,a(Y,Y))=a(X,a(m,Y))
a(X,a(X,a(m,a(1,X))))->a(m,a(1,X))
a(a(X,X),a(X,X))->a(m,a(m,X))
.
a(a(a(m,b),X),Y)->a(b,a(X,Y))
a(a(m,a(m,b)),a(b,b))->a(b,a(m,a(m,b)))
a(a(m,a(m,b)),X)->a(b,a(a(b,b),X))
a(a(a(m,X),a(m,X)),a(m,a(X,X)))->a(m,a(m,a(m,X)))
a(X,a(a(Y,Y),Z))=a(X,a(a(m,Y),Z))
a(a(m,a(b,X)),Y)=a(X,a(a(b,X),Y))
a(m,a(m,a(a(b,m),a(a(b,m),a(a(b,m),a(b,m))))))
  ->a(m,a(m,a(b,m)))
a(X,a(a(b,X),a(a(b,X),a(b,X))))->a(m,a(m,a(b,X)))
a(a(a(m,X),a(m,X)),a(a(m,X),a(X,X)))->a(m,a(m,a(m,X)))
a(X,a(Y,a(Z,Z)))=a(X,a(Y,a(m,Z)))
a(X,a(Y,a(a(b,X),Y)))->a(m,a(a(b,X),Y))
a(a(m,a(m,X)),a(m,a(X,X)))->a(m,a(m,a(X,X)))
a(X,a(Y,a(m,Z)))=a(X,a(Y,a(Z,Z)))
a(a(m,a(m,a(1,m))),a(m,a(m,a(1,m))))->a(m,a(1,m))
```

```
a(a(m,X),a(X,X))->a(m,a(X,X))
a(a(m,X),a(m,X))->a(m,a(X,X))
a(b,a(a(b,b),a(m,a(m,b))))->a(m,a(m,a(m,b)))
a(m,a(X,X))=a(m,a(m,X))
a(a(X,X),a(m,a(Y,Y)))=a(a(m,X),a(m,a(m,Y)))
a(a(X,X),a(m,Y))=a(a(m,X),a(Y,Y))
a(a(m,1),a(X,X))->a(1,a(m,a(m,X)))
a(1,a(m,a(1,1)))->a(m,a(m,1))
a(1,a(m,a(m,1)))->a(m,a(m,1))
a(m,a(m,a(X,X)))=a(m,a(m,a(m,X)))
a(X,a(m,a(Y,Y)))=a(X,a(m,a(m,Y)))
a(X,a(m,a(m,Y)))=a(X,a(m,a(Y,Y)))
a(X,a(m,a(m,Y)))=a(X,a(m,a(Y,Y)))
a(1,a(X,X))=a(a(m,1),X)
a(m,a(m,a(1,m)))->a(m,a(1,m))
a(m,a(a(b,X),a(b,X)))->a(m,a(m,a(b,X)))

a(a(m,1),f(a(m,1)))->a(1,a(m,f(a(m,1))))
a(a(m,a(m,X)),a(m,a(m,a(b,Y))))
  =a(a(m,a(X,X)),a(m,a(m,a(b,Y))))
a(a(m,a(m,X)),a(m,a(m,Y)))=a(a(m,a(X,X)),a(m,a(m,Y)))
a(a(m,a(m,X)),a(Y,Z))=a(a(m,a(X,X)),a(Y,Z))
a(a(m,X),a(m,a(m,a(b,Y))))=a(a(X,X),a(m,a(m,a(b,Y))))
a(a(m,X),a(m,a(m,Y)))=a(a(X,X),a(m,a(m,Y)))
a(a(m,X),a(Y,Z))=a(a(X,X),a(Y,Z))
a(a(m,X),a(a(b,a(X,X)),a(a(b,a(X,X)),a(b,a(X,X)))))
  ->a(m,a(m,a(b,a(X,X))))
a(a(X,X),a(m,a(m,a(b,Y))))=a(a(m,X),a(m,a(m,a(b,Y))))
a(a(X,X),a(m,a(m,Y)))=a(a(m,X),a(m,a(m,Y)))
a(a(X,X),a(Y,Z))=a(a(m,X),a(Y,Z))
a(b,a(a(m,b),a(m,a(b,b))))->a(m,a(m,a(m,b)))
a(a(m,a(m,a(m,X))),a(m,a(m,a(X,X))))->a(m,a(m,a(m,a(X,X))))
a(X,a(a(m,a(m,Y)),Z))=a(X,a(a(m,a(Y,Y)),Z))
a(b,a(a(b,b),a(m,a(m,X))))=a(b,a(a(b,b),a(m,a(X,X))))
a(b,a(a(b,b),a(m,a(X,X))))=a(b,a(a(b,b),a(m,a(m,X))))
a(a(a(X,X),Y),a(a(X,X),Y))->a(m,a(a(m,X),Y))
a(X,a(a(m,Y),Z))=a(X,a(a(Y,Y),Z))
a(b,a(a(m,b),a(m,a(m,b))))->a(m,a(m,a(b,b)))
a(a(a(m,X),Y),a(a(X,X),Y))->a(m,a(a(m,X),Y))
a(a(a(m,X),Y),a(a(m,X),Y))->a(m,a(a(X,X),Y))
a(a(m,a(b,b)),a(b,b))->a(b,a(m,a(m,b)))
a(a(m,a(b,m)),a(b,m))->a(m,a(m,a(b,m)))
a(a(m,a(m,a(m,X))),a(a(m,a(X,X)),a(m,a(m,X))))
  ->a(m,a(m,a(m,a(X,X))))
```

```
a(X,a(Y,a(m,a(m,Z))))=a(X,a(Y,a(m,a(Z,Z))))
a(X,a(a(b,X),a(m,a(b,X))))->a(m,a(m,a(b,X)))
a(a(X,a(Y,Y)),a(X,a(Y,Y)))->a(m,a(X,a(m,Y)))
a(a(X,a(m,Y)),a(X,a(Y,Y)))->a(m,a(X,a(m,Y)))
a(a(X,a(m,Y)),a(X,a(m,Y)))->a(m,a(X,a(Y,Y)))
a(a(X,a(a(b,m),X)),a(X,a(a(b,m),X)))->a(m,a(a(b,m),X))
a(a(m,a(m,a(X,X))),a(m,a(m,a(m,X))))->a(m,a(m,a(m,a(m,X))))
a(X,a(Y,a(m,a(Z,Z))))=a(X,a(Y,a(m,a(m,Z))))
a(a(X,a(Y,Y)),a(X,a(m,Y)))->a(m,a(X,a(Y,Y)))
a(a(a(X,X),Y),a(a(m,X),Y))->a(m,a(a(X,X),Y))
a(b,a(X,a(a(m,b),X)))->a(m,a(a(m,b),X))
a(X,a(m,a(1,X)))->a(m,a(1,X))
a(a(m,a(X,X)),a(m,Y))=a(a(m,a(m,X)),a(Y,Y))
a(a(m,a(m,X)),a(m,a(Y,Y)))=a(a(m,a(X,X)),a(m,a(m,Y)))
a(a(m,a(m,X)),a(m,Y))=a(a(m,a(X,X)),a(Y,Y))
a(a(m,a(m,1)),a(X,X))=a(a(m,a(m,1)),a(m,a(m,X)))
a(a(m,a(m,1)),a(X,X))->a(a(m,a(m,1)),X)
a(a(m,a(1,1)),a(X,X))=a(a(m,a(1,1)),a(m,a(m,a(m,X))))
a(a(m,a(1,1)),a(m,a(X,X)))->a(a(m,a(1,1)),X)
a(a(m,X),a(m,a(Y,Y)))=a(a(X,X),a(m,a(m,Y)))
a(a(m,X),a(m,Y))=a(a(X,X),a(Y,Y))
a(a(X,X),a(m,a(m,Y)))=a(a(m,X),a(m,a(Y,Y)))
a(a(m,1),a(m,X))=a(1,a(m,a(X,X)))
a(a(m,1),X)->a(1,a(m,X))
a(X,a(Y,a(m,a(Z,Z))))=a(X,a(Y,a(m,a(m,Z))))
a(a(X,X),a(Y,Y))=a(a(m,X),a(m,Y))
a(a(m,a(X,X)),a(Y,Y))=a(a(m,a(m,X)),a(m,Y))
a(a(m,X),a(Y,Y))=a(a(X,X),a(m,Y))
a(a(m,X),a(m,a(m,Y)))=a(a(X,X),a(m,a(Y,Y)))
a(X,a(a(b,X),Y))=a(a(m,a(b,X)),Y)
a(a(m,a(m,1)),a(m,a(m,X)))->a(a(m,a(m,1)),X)
a(a(m,X),a(a(b,a(X,X)),a(m,a(b,a(m,X)))))
  ->a(m,a(m,a(b,a(X,X))))
a(m,a(1,1))->a(m,a(m,1))
a(1,a(m,a(X,X)))=a(1,a(m,a(m,X)))
a(m,a(m,a(m,X)))=a(m,a(m,a(X,X)))
.
proof found
false:-
   prolog(\+f_chaining)
   a(a(a(b,m),1),f(a(a(b,m),1)))
    =a(m,a(1,f(a(a(b,m),1)))):-
      prolog(f_chaining)
      input(a(b,m)=a(b,m))
      input(a(a(b,m),l)=a(a(b,m),l))
      input(a(a(a(b,m),l),f(a(a(b,m),l)))
            =a(a(a(b,m),l),f(a(a(b,m),l))))
      input(a(l,f(a(a(b,m),l)))=a(l,f(a(a(b,m),l))))
      input(a(m,a(l,f(a(a(b,m),l))))
            =a(m,a(l,f(a(a(b,m),l)))))
   a(m,a(l,f(a(a(b,m),l))))
     =a(f(a(a(b,m),l)),a(m,a(l,f(a(a(b,m),l))))):-
      prolog(f_chaining)
      input(a(m,a(l,f(a(a(b,m),l))))
            =a(m,a(l,f(a(a(b,m),l)))))
      a(a(l,f(a(a(b,m),l))),a(l,f(a(a(b,m),l))))
        =a(f(a(a(b,m),l)),a(m,a(l,f(a(a(b,m),l))))):-
         prolog(f_chaining)
         input(a(l,f(a(a(b,m),l)))=a(l,f(a(a(b,m),l))))
         input(a(a(l,f(a(a(b,m),l))),a(l,f(a(a(b,m),l))))
               =a(a(l,f(a(a(b,m),l))),a(l,f(a(a(b,m),l)))))
         a(a(l,f(a(a(b,m),l))),a(l,f(a(a(b,m),l))))
           =a(m,a(l,f(a(a(b,m),l)))):-
            prolog(f_chaining)
            a(m,a(l,f(a(a(b,m),l))))
              =a(a(l,f(a(a(b,m),l))),a(l,f(a(a(b,m),l)))):-
               prolog(f_chaining)
               input(a(m,a(l,f(a(a(b,m),l))))
                     =a(m,a(l,f(a(a(b,m),l)))))
               input(a(a(l,f(a(a(b,m),l))),a(l,f(a(a(b,m),l))))
                     =a(a(l,f(a(a(b,m),l))),a(l,f(a(a(b,m),l)))))
         input(a(f(a(a(b,m),l)),a(m,a(l,f(a(a(b,m),l)))))
               =a(f(a(a(b,m),l)),a(m,a(l,f(a(a(b,m),l))))))
size of proof 32
clause count 17
919.716 cpu seconds used
1813 inferences done
```

Since the statement of the theorem is the same as before, the proof starts with the generation of the same subgoals. The first equational subgoal matches the definition of the BLUEBIRD combinator, and generates 5 subgoal equations, all discharged as instances of the reflexivity axiom, here the only ultimate grounder of subgoals. The second matches the definition of the MOCKINGBIRD, and so adds two more equations to derive. The first is again satisfied immediately as an instance of `X = X`, and the second takes us into the definition of LARK. (Note $lxlx$ here appears as mlx). Its first two equational subgoals as well as the fourth are instances of the reflexivity axiom, and the third calls on symmetry. The subgoal equation matches the MOCKINGBIRD definition, whose two equations are both grounded in the reflexivity axiom again. Unlike the previous proofs, this proof is constructed almost entirely through forward-chaining, and so the order of the proof does not reflect at all that of its construction.

2.5 Results

2.5.1 The Problems

We tested the three representations on 16 combinator problems drawn from [15]. The results appear in Figure 14. The convention followed in labeling the problems is that the given combinators are listed to the left of the underscore, and what is to be derived, usually another combinator, to the right.

Two problems proved too difficult for sprfn, at least with the weights given. McCune and Wos [16] were able with much human effort and hours of computer time to guide their prover to a derivation of a fixed-point combinator (a 'sage') from the b and w combinators. We were not able to get sprfn to derive this in one step. When given a pair of lemmas in the form of two intermediate combinators suggested by Smullyan, a and c, the problem becomes soluble for sprfn under all problem representations. Similarly, we were unable to derive a complicated permutor, psi, from b,c (a different c, called a 'cardinal' in [15]) and w, but it is relatively simple when done via the derivation of a 'dovekie' and a 'hummingbird.' The difficulty with both BW_SAGE and BCW_PSI is most likely the 'warbler' combinator:

$$\mathsf{w}xy = xyy$$

both sides of the definition unify, rendering simplification impossible.

The derivation of the 'cardinal', a simple three-place permutor, from the composition combinator 'bluebird' and the two-place permutor 'thrush' (discovered by Church) proved surprisingly difficult, so another three-place permutor, the 'robin', was introduced as an intermediary in a pair of problems (suggested by Smullyan).

In KW_MOCK we derive the duplicative 'mockingbird' from the aforementioned 'warbler' and the cancellative 'kestrel':

$$\mathsf{k}xy = x.$$

L_EGO involves the construction of an "egocentric" function, which returns itself when given itself as an argument, from a 'lark'. This problem was solved in [17]. The final problem constructs a combinator that commutes with every other combinator from the 'thrush' and the condition that every function has a fixed-point.

2.5.2 Data

CPU times and the number of inferences required for the problems are tabulated in Figure 14. The problem were run under C-Prolog on a Sun 3/60. No effort was made to control for load, which varied from light to moderate. Blanks indicate that the problem was not solved, usually by running out of resources. In the one case marked by an asterisk, the problem was solved only under the 'nosave' option, which turns off the caching feature, and results in faster but more repetitive inferences using comparatively little memory.

The surprising result was that the no-rewrite representation performed the best in terms of shortest time. It should be noted that the no-rewrite version would have had an unsolved problem, however, had we not disabled the caching feature on the BT_CARD problem. Nonetheless, the equational axioms did perform unexpectedly well, as is shown in another way in Figure 15. Here we list the first and last place totals for each representation.

One bright spot for the critical pair representation is that is does have the lowest average number of inferences, even if not CPU time. Focussing on this number not only sidesteps the inaccuracy of time statistics in UNIX, but also discounts the overhead involved in the rewriting and completion phases, and so points out the potential for efficient implementations of these phases. Unfortunately, this measure is skewed badly by the 'nosave' run, which quickly piles up repeated inferences. Dropping the inference counts for BT_CARD from both columns, we find that the critical pair representation averages over twice the number of inferences made by the no-rewrite representation, and the modification format averages over three times as many.

2.6 Discussion

The rewriting strategies we have employed do not exhibit the efficiency we had expected. Clearly directing equations has not provided the additional control promised. It must be emphasized that the data are insufficient to warrant any firm conclusions. We have compared only 16 problems of a specific type, using but one setting of the various switches sprfn provides. Nonetheless, some provisional morals suggest themselves.

	CPU seconds			# of Inferences		
Theorem	NRW	CP	MOD	NRW	CP	MOD
ac_sage	33	49	41	107	139	158
b_dovk	24	21	52	119	133	258
bcw_humm	243	493	1325	812	1094	2095
bcw_psi	—	—	—	—	—	—
bdh_psi	13	67	127	73	263	397
blm_sage	72	210	920	240	633	1813
bs_phi	23	370	120	639	660	347
bt_card	377*	1022	14963	15026	3663	42379
bt_rob	7	15	35	44	114	189
bw_a	38	42	112	142	186	383
bw_c	157	3522	10082	547	4852	7876
bw_sage	—	—	—	—	—	—
kw_mock	330	83	53	654	412	260
l_ego	297	53	34	522	155	132
r_card	4	6	20	23	35	103
t_comm	8	6	12	50	29	58
AVERAGE1†:	116	426	1993	1357	883	4032
AVERAGE2‡:	96	378	995	306	670	1082

*: caching turned off.

†: BCW_PSI and BW_SAGE not included.

‡: BCW_PSI, BW_SAGE, and BT_CARD not included.

Figure 14: Statistics for combinator problems

	CPU time		Inferences	
	1st	3rd	1st	3rd
NRW	10	2	9	2
CP	2	2	2	1
MOD	2	9	3	11

Figure 15: First and Last Place Finishes

One surprise was how well sprfn can handle the axiomatic approach to equality. In fact this was a surprise, since only late in the testing did we come to seriously apply the sorts of efficiency guards to the no-rewrite version we had worked out for the other representations. We believe that the back-chaining nature of the prover worked well with this approach, and that the ability to limit some of the axioms to either the forward and backward chaining phases of the prover helped significantly.

We do not yet understand well enough how this integration of completion and theorem-proving works. For example, the input files were originally written to work with an alphabetical lexicographic ordering. It turned out that performance was significantly improved when terms were reverse alphabetically ordered! Some problems took longer, but most were solved more quickly, and one theorem previously unprovable using the modification format was proven under this orientation.[3] This was surprising since the lexical ordering of atoms has an effect only if terms are tied at the top level and on all subterms considered up to that point. At most a few percent of the critical pairs generated in our examples are oriented by considering the alphabetical ranking of identifiers. Presumably this ordering has more effect in rewriting instances, which is invisible to the user.

This case and other surprises show that we do not yet understand our technique well enough to write it off. A better use of the orienter (or perhaps one using different principles), or a more clever setting of the switches sprfn provides may make all the difference for the rewriting approach.

Nonetheless, the integration of completion into the prover did not work as well as we had hoped. We suspect that the amount of information coming in under the forward chaining phase in the form of new directed equations simply overwhelmed the normally efficient goal-orientation of sprfn. Note that the modification representation, in which all but the opening stage of the proof is derived in forward chaining, performed the poorest overall.

The tactic of carrying out completion in the forward-chaining phase and the rest of the proof primarily in the back-chaining phase would undoubtedly be aided by more sophisticated forward chaining, currently perhaps the least intelligent part of sprfn. The general problem of introducing search priorities within the iterative deepening stategy is being investigated by Xumin Nie. Tests on experimental priority systems devised by Nie have sometimes caused dramatic improvements for the rewriting techniques described here, and it would be worth investigating which representation benefits the most

[3] *viz.*, BW_C.

from the various priority schemes.

Even granting the preliminary status of our results, it is reasonable to reconsider the strategy of integrating completion into theorem-proving as a way of handling equality. After all, by its very nature, completion will introduce premises not particularly helpful for the proof at hand. Another possibility we have been exploring is a more directed representation taking advantage of the left-linear nature of combinator definitions, more akin to **narrowing** (*cf.* [18]) than completion.

References

[1] G. Birkhoff. "On the structure of abstract algebras." *Proceedings of the Cambridge Philosophical Society* **31** (1935), pp. 433 – 454.

[2] N. Dershowitz. "Orderings for term-rewriting systems." *The Journal of Theoretical Computer Science* **17** (1982), pp. 279 – 301.

[3] M.H.A. Newman. "On theories with a combinatorial definition of 'equivalence'." *Annals of Mathematics* **43** (1942), pp. 223 – 243.

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

[5] D. Knuth and P. Bendix. "Simple word problems in universal algebras." In *Computational Problems in Abstract Algebra*, J. Leech, ed. Pergamon Press, NY (1970), pp. 263 – 297.

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

[7] N. Dershowitz. "Completion and its applications." *Proceedings of the Colloquium on the Resolution of Equations in Algebraic Structures*, H. Ait-Kaci & M. Nivat, eds. Austin, Texas. (1987).

[8] L. Bachmair, N. Dershowitz, and D. Plaisted. "Completion without failure." *Proceedings of the Colloquium on the Resolution of Equations in Algebraic Structures*, H. Ait-Kaci & M. Nivat, eds. Austin, Texas. (1987).

[9] G. Peterson and M. Stickel. "Complete sets of reductions for some equational theories." *Journal of the Association for Computing Machinery* **28** (1981), pp. 233 – 264.

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

[11] L. Bachmair and N. Dershowitz. "Completion for rewriting modulo a congruence." *Proceedings of the Second International Conference on Rewriting Techniques and Applications* (1987).

[12] D. Brand. "Proving theorems with the modification method." *SIAM Journal of Computing* 4 (1975), pp. 412 – 430.

[13] D. A. Plaisted and S. Greenbaum. "Problem representations for back chaining and equality in resolution theorem proving." *proceedings of the First IEEE Conference on Artificial Intelligence Applications* (1984), pp. 417 – 422.

[14] D. A. Plaisted. "Non-Horn clause logic programming without contrapositives." *Journal of Automated Reasoning*, forthcoming.

[15] R. Smullyan. *To Mock a Mockingbird.* Knopf, NY (1985).

[16] W. McCune and L. Wos. "A case study in automated theorem proving: finding sages in combinatory logic." *Journal of Automated Reasoning* **3** (1987), pp. 91 – 107.

[17] B. Glickfield and R. Overbeek. "A foray into combinatory logic." *Journal of Automated Reasoning* **2** (1986), pp. 419 – 431.

[18] J.-M. Hullot. "Canonical forms and unification." *Proceedings of the Fifth Conference on Automated Deduction*, printed as *Lecture Notes in computer Science* **87**, Springer-Verlag (1980), pp. 318 – 334.