A Method for Implementing Equational Theories as Logic Programs

M.H.M. Cheng
M.H. van Emden*
Department of Computer Science
University of Victoria
Victoria, B.C., V8W 2Y2 Canada

D. Stott Parker†
Computer Science Department
University of California
Los Angeles, CA 90024-1596 USA

Abstract

Equational theories underly many fields of computing, including functional programming, symbolic algebra, theorem proving, term rewriting and constraint solving. In this paper we show a method for implementing many equational theories with a limited class of logic programs. We define *regular equational theories*, a useful class of theories, and illustrate with a number of examples how our method can be used in obtaining efficient implementations for them.

The significance of our method is that:

- It is simple and easy to apply.
- Although executable, it supports separation of concerns between specification and implementation.
- Its class of logic programs execute with impressive efficiency using Prolog.
- It permits interesting compilation and optimization techniques that can improve execution efficiency still further.
- It offers perspectives on term rewriting and functional programming evaluation strategies, how they can be compiled, and how they can be integrated with logic programming effectively.

1 Introduction

Every scientific discipline rests on particular notions of equivalence or equality, and these notions differ in both their theoretical and computational properties. To appreciate the diversity of notions one need look no further than the large literature for symbolic algebra, functional programming, logic programming, algebraic program semantics, automated theorem proving, term rewriting systems, and constraint satisfaction.

How should one go about implementing equational theories? This is apparently a difficult question to answer well, if the volume of literature is any indicator. Still, the more one understands this question and different approaches for answering it, the more one senses common themes with variations. It would be useful if these themes could be brought out and made explicit.

This paper describes one framework for answering this question. Our contribution is to identify a suitable axiomatization, in a suitable form, of a subtheory of the usual theory of equality, and to argue that it can be used as a formal theme, or paradigm, in (re)constructing many variations. Since the usual theory of equality is a Horn theory, subtheories can be expressed naturally with logic programming. Nevertheless, having a suitable form for its axiomatization is crucial in obtaining useful computational implementations.

*Supported by the Natural Sciences and Engineering Research Council of Canada.
†This research supported by an NSF grant.

We call theories generated by our axiomatization **Regular Equational Theories (RETs)**, since they can be specified with something like regular expressions. The contribution is similar in ways to the development of Definite Clause Grammars (DCGs) [15], in that it tries to identify a general equational 'paradigm' for logic programming, in which programmers can express computationally useful subtheories of the standard equality theory. The paradigm is very simple. We will show also that it works very well in practice, running surprisingly fast on Prolog systems and having a theory that supports compilation techniques. It also gives new perspective on laziness and control, and the relationship between functional programming, logic programming, term rewriting, and automated theorem proving.

A critical part of any research like this is experimentation on a large number of examples. Paradigms take time to evolve from the ooze. Also, paradigms are sometimes best appreciated both when worked out in the privacy of the reader's brain, and when studied as a large ensemble. A paper like this is not the right place to present a large ensemble, so we encourage the reader to consider working with some sample RETs. About twenty RET programs in Prolog are available from the authors, and can be retrieved over the Internet as `file://pop.cs.ucla.edu/pub/eq/Distribution.shar.Z`. A complete version of this paper, with proofs of theorems, is available there also.

1.1 The Axioms of Equality

The equality relation is one of the most important in logic. It alone is sufficient for developing very useful mathematical theories in first-order logic.

The set **Eq** of axioms of equality in first-order logic is defined as follows:

R: Reflexivity axiom

$$\forall x \quad x = x.$$

S: Symmetry axiom

$$\forall x \forall y \quad x = y \ \leftarrow \ y = x.$$

T: Transitivity axiom

The standard transitivity axiom for equality is:

$$\forall x \forall y \forall z \quad x = z \ \leftarrow \ x = y, \ y = z.$$

F: Function substitutivity axioms

For each n-ary function symbol f, and for each i, $1 \leq i \leq n$:[1]

$$\forall x_1 \ldots \forall x_n \forall y_i \quad f(x_1, \ldots x_{i-1}, x_i, x_{i+1}, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, y_i, x_{i+1}, \ldots x_n) \ \leftarrow \ x_i = y_i.$$

P: Predicate Substitutivity axioms

For all n-ary predicate symbols p:

$$\forall x_1 \ldots \forall x_n \forall y_1 \ldots \forall y_n \quad p(x_1, \ldots, x_n) \ \leftarrow \ p(y_1, \ldots, y_n), \ x_1 = y_1, \ \ldots, \ x_n = y_n.$$

The set of axioms **Eq** is augmented by a set **E** of (user-defined) assertions. For now, these assertions can be assumed to take the Horn form

$$(l \ \Rightarrow \ r) \quad \leftarrow \quad q$$

where l and r are first-order terms, and q is an optional query (conjunction), rendering the equation **conditional**. The query q must obey restrictions discussed later.

[1] There are many equivalent forms of equality axioms. We believe that it is widely agreed that the above are their most natural forms, but for one exception: functional substitutivity. The following alternative form seems to have equal claim to being most natural:

F′: For each n-ary function symbol f,

$$\forall x_1 \ldots \forall x_n \ \forall y_1 \ldots \forall y_n \quad f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \ \leftarrow \ x_1 = y_1, \ \ldots, \ x_n = y_n.$$

Nevertheless in this paper we will rely on **F** instead of **F′**.

1.2 Previous Work

The axiomatization above has positive and negative aspects. On the positive side, the axioms are all Horn sentences, and so are easily implemented in computational logic systems. On the negative side, the axioms can be composed (via resolution) in so many ways that it is not feasible to search for proofs using them naively. One learns quickly that careless use of the equality axioms can yield computationally useless systems — systems that are not practical in any implementation.

Since the initial work on implementing equality-processing subsystems in automated theorem provers in the 1960's, hundreds of attempts have been made to develop useful systems. Such concepts as paramodulation, demodulation and simplification, unification with equality, resolution with equality, narrowing, and ultimately what we now refer to as 'term rewriting' are descendants of this continuing development. A good recent review of this evolution may be found in Chapter 1 of [7].

To date, successful approaches have concentrated on implementing specialized equational theories that are important for some field. Functional programming, for example, typically uses the axioms only in an applicative form, in which any user-defined equation $f(t_1, \ldots, t_n) = g$ is represented as

$$f(x_1, \ldots, x_n) \ = \ y \quad \leftarrow \quad x_1 = t_1, \ \ldots, \ x_n = t_n, \ \ g = y$$

— a consequence of transitivity, function substitutivity, and the user's equation. Logic programming has succeeded because programmers have sublimated their desire to use the equality relation; for example using `sum(X,Y,Z)` instead of `X+Y=Z`. Prolog has managed quite well with only the reflexivity axiom. A variety of other specialized theories are summarized in [9].

A complication of concentrating on specialized equational theories is that every implementation carries involved assumptions and restrictions on use. It can be difficult to 'reuse' such work in developing new programs. Our interest has recently been in going back to axioms of **Eq** in an attempt to logically reconstruct some of the results in handling equational theories, and (hopefully) extract themes of general applicability.

This work is an outcome of several years' work in developing and reasoning about equational systems, functional programming systems, and term rewriting systems in logic programming. Cheng [2], Narain [13], and Parker [14] each developed fairly extensive implementations of such systems. Van Emden and Yukawa [20] has in common with the present paper that it obtains an executable logic program for rewriting by selecting computationally useful theorems about equality. The resulting program has the disadvantage that a separate check for canonicality is needed, thereby detracting from efficiency. This defect was corrected by Cheng and Yukawa [5] at the expense of adding a third argument to some of the equality predicates to serve as a flag. This detracted from the declarative quality of their logic program. Richards [18] solved the problem with [20] in a declarative way by explicitly reasoning about the one-step rewrite relation. We have used this idea in the second proof of Theorem 1 below.

Our method is not intended to be compared side-by-side with any scheme for implementing a specialized class of theories. The emphasis here is not really on performance, although the method does perform well on a wide range of theories. Nor is it on completeness, although it is not difficult to implement complete search strategies, or to integrate tools like the Knuth-Bendix completion procedure. The emphasis is on identifying subtheories of **Eq** that are of general applicability in programming, and which permit some separation of specification from execution. This emphasis also permits us to logically reconstruct different compilation techniques for equations (such as [12]), and get perspective on different proposals for implementing equality.

1.3 Considerations in Implementing Equality

Overall, our objective is to identify and implement *computationally useful subtheories of equality*, i.e., subtheories of the full equality theory that are amenable to practical computer implementation. We are interested in implementations of equality using logic programming systems with a fixed computation rule, such as Prolog with its left-to-right goal selection and top-to-bottom clause selection rules. In other words, we are interested in logic programming implementations of equality, with some emphasis on the word *programming*.

The following principles have influenced the design of our method:

- *Subtheories of equality are enough*
  Implementing the full equality theory is only rarely necessary. The symmetry axiom can often be discarded by following the convention that the equality theory be used as a 'simplifier', so the right argument is always 'simpler' than the left. Some substitutivity axioms can also often be discarded, as we will show in Section 4.

- *Equality should succeed only in 'interesting' cases*
  In any equality theory there are many valid but uninteresting equations [20]. However, the equality predicate should succeed only for interesting argument values, such as for normal forms (canonical terms), etc. We will see later how an evaluation strategy, such as innermost or outermost evaluation, determines its success.

- *Equality theories should be compilable*
  As in [3, 5, 18] we can implement an equational system in Prolog by first developing a Prolog specification for the equality theory as suggested here, and then relying on compilation techniques to speed it up. This approach has the advantage that it is provably correct, and yields efficient programs when the equality theory is well-written.

- *Computational behavior of equality theories should be comprehensible*
  There must be a simple model of the execution of the program; it is not enough that there be an 'equality theory'. It must be clear where the equality predicate is being used, what its evaluation strategy is, and what computational complexity it will exhibit.

2 Computational Implementation of the Axioms of Equality

The method we propose develops **directed** equational theories as rewrite relations, which derive the equation $s = t$ when the term s can be rewritten to t as a result of a finite reduction sequence. Rather than base the derivation on the standard axioms given above, we will formulate mini-theories of equality that can be justified as subtheories of **Eq**. These mini-theories define subsets of the equality relation (regarding these relations as sets of pairs). The subsets are large enough to contain the desired results of rewriting and are defined by axioms of a form that qualifies as a logic program. As a result the efficient Prolog theorem prover can obtain the result of rewriting at a cost in time and memory that makes this method of practical interest.

To obtain our mini-theories, we introduce of different predicate symbols eq_i for $i = 1, 2, \ldots$ that, though different, all have as same meaning the equality symbol used above. Thus we add to **Eq** the clauses $\mathrm{eq}_i(X, Y) \leftrightarrow X = Y$ for whatever values of i we need.

We then collect theorems about the eq_i from the axioms in **Eq**. These theorems can be used as sole definition of eq_i, that is, not using the axiom $\mathrm{eq}_i(X, Y) \leftrightarrow (X \Rightarrow Y)$. In this way the theorems define a subrelation of equality; that is, they constitute a mini-theory as referred to above.

2.1 The Regular Equational Theory Paradigm

Our method is to use the following equational paradigm when developing logic programs:

$\mathrm{eq}(x, y) \;\leftarrow\; \mathrm{eq}_2(x, y)$	(directed equality)
$\mathrm{eq}_2(x, z) \;\leftarrow\; \mathrm{eq}_1(x, y),\ \mathrm{eq}_2(y, z)$	(transitivity)
$\mathrm{eq}_2(x, x) \;\leftarrow$	(reflexivity)
$\mathrm{eq}_1(x, y) \;\leftarrow\; (x \;\Rightarrow\; y)$	(rule reference clause)
$\mathrm{eq}_1(f(x_1, y, ..., z),\ f(x_2, y, ..., z)) \;\leftarrow\; \mathrm{eq}_1(x_1, x_2)$	(substitutivity for f)
$\mathrm{eq}_1(f(x, y_1, ..., z),\ f(x, y_2, ..., z)) \;\leftarrow\; \mathrm{eq}_1(y_1, y_2)$	(substitutivity for f)
$\vdots$	$\vdots$
$\mathrm{eq}_1(f(x, y, ..., z_1),\ f(x, y, ..., z_2)) \;\leftarrow\; \mathrm{eq}_1(z_1, z_2).$	(substitutivity for f)

The **rule reference clause** $\mathrm{eq}_1(X, Y) \leftarrow (X \Rightarrow Y)$ states that two terms are equal if one rewrites to the other. The substitutivity rules are included for each function symbol f of positive arity appearing in the user-defined equations **E**. As mentioned, equations in **E** have the general conditional form

$$(l \;\Rightarrow\; r) \quad \leftarrow \quad q$$

defining the binary predicate '$\Rightarrow$'. For reasons that we will clarify later, we require that the condition q always have a finite proof tree [11]. (Specifically, this requirement prohibits use of eq or eq_2 in any condition q, in order to exclude recursions that would go beyond our paradigm.)

This paradigm discards the symmetry axiom. It adopts a directed view of equality in which the left argument of eq is always found equal to the right argument after some number of user-defined rewriting steps. It may be possible to incorporate symmetry or other extensions in the paradigm, but we will not consider that here.

All programs shown below adopt this paradigm. It includes most of the axioms of equality in **Eq**, as labeled, but with a particular structure: the predicate eq_2 forms the reflexive transitive closure of the rewrite relation defined by eq_1, which comprises user equations and function substitutivity.

It is important to stress that, with this paradigm, *the ordering of equational axioms can matter*. In Section 4 this is actually shown to be useful. Logic programming systems rely on a computation rule and search rule in finding proofs to queries [11]. For example, Prolog's SLD interpreter consumes goals completely before dismissing them, and processes rules top-to-bottom. This requires us to find orderings for the equality axioms that will guarantee completeness. Without care, infinite recursion and prodigious backtracking result. Using the ordering above and the top-to-bottom search rule, SLD resolution will find longer equational proofs before it finds shorter proofs; we believe this corresponds well to finding 'interesting' equations.

We call this the **regular equational theory (RET) paradigm**, and the logic programs developed using it **regular equational theories**, for reasons explained later. With it we (the 'user') wish to derive consequences of these assertions using the equality relation, and do so in a computationally useful way. First, we run through several examples illustrating how this can be done.

2.2 Example 1: Combinatory/Functional Programming

This equational theory, while limited, is computationally very useful and can be implemented directly in logic programming systems like Prolog. Let us illustrate this with a few examples. Consider the Prolog clauses

```
succ : X   =>  s(X).
twice : F  =>  t(F).
t(F) : X   =>  F : (F : X).
```

defining the combinators `twice` and `succ`, where '`:`' is a left-associative application operator. The equality theory above can be specialized for the binary function symbol `:` to obtain:

```
eq(X,Y) :- eq2(X,Y).

eq2(X,Z) :- eq1(X,Y), eq2(Y,Z).       % transitivity
eq2(X,X).                             % reflexivity

eq1((X1:Y),(X2:Y)) :- eq1(X1,X2).     % substitutivity for ':'
eq1((X:Y1),(X:Y2)) :- eq1(Y1,Y2).     % substitutivity for ':'
eq1(X,Y) :- (X => Y).                 % rule reference clause
```

This equality theory, along with the above equations, is sufficient to implement combinatory logic (and functional programming with higher-order functions) [3]. The goal

```
?- eq( twice:twice:twice:succ:0, X ).
```

immediately yields the solution 16, or

```
X = s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(0)))))))))))))))).
```

In fact, Prolog is *faster* in obtaining this solution than are many Lisp systems, which rely on uninspired implementations of combinators ('lambdas'); see [3].

2.3 Example 2: Algebraic Simplification

As another example, let us consider a simple set of equations for simplifying conjunctions in Boolean algebras to normal form, adapted directly from [16]:

```
(X * Y) * Z    =>     X * (Y * Z).
X * Y          =>     Y * X           :-  greater(X,Y).
X * (Y * Z)    =>     Y * (X * Z)     :-  greater(X,Y).
X * X          =>     X.
X * (X * Y)    =>     X * Y.
(X + Y) + Z    =>     X + (Y + Z).
X + Y          =>     Y + X           :-  greater(X,Y).
X + (Y + Z)    =>     Y + (X + Z)     :-  greater(X,Y).
X + X          =>     0.
X + (X + Y)    =>     Y.
X * 1          =>     X.
1 * X          =>     X.
X + 0          =>     X.
0 + X          =>     X.
X * 0          =>     0.
0 * X          =>     0.
X * (Y + Z)    =>     X * Y + X * Z.
(X + Y) * Z    =>     X * Z + Y * Z.


eq(X,Y) :- eq2(X,Y).

eq2(X,Z) :- eq1(X,Y), eq2(Y,Z).          % transitivity
eq2(X,X).                                % reflexivity

eq1((X1*Y),(X2*Y)) :- eq1(X1,X2).        % substitutivity
eq1((X*Y1),(X*Y2)) :- eq1(Y1,Y2).        % substitutivity
eq1((X1+Y),(X2+Y)) :- eq1(X1,X2).        % substitutivity
eq1((X+Y1),(X+Y2)) :- eq1(Y1,Y2).        % substitutivity
eq1(X,Z) :- (X => Y).                    % rule reference clause
```

Here '*' is Boolean conjunction, and '+' is 'exclusive or'. This self-contained equality theory is sufficient to implement simplification in any Boolean algebra, including theorem proving: simplifying formulas to 1 (validity) or 0 (falsehood). For example, the goal

```
?- eq(   (((p*(1*q))*r)*((1*1)*s)) + (((p*q)*r)*s),   NormalForm  ).
```

directly produces the solution:

```
NormalForm  =  0.
```

This approach uses conditional rules, in what is called 'ordered term rewriting' [6, 16]. Ordered term rewriting extends standard term rewriting in that user-defined rules are permitted to include ordering constraints. The condition `greater(X,Y)` here insists that `X` be larger than `Y` in some term ordering (such as the lexicographic path ordering [6]), sufficient to guarantee that the rules in question terminate when rewriting.

2.4 Example 3: Primes

A commonly given example for relatively advanced functional programming is the computation of primes by the method of sieves. The idea is to model `primes` as a **stream** (a potentially infinite list) and compute `first(10,primes)`, the list of the first 10 primes. The rules below carry this out, permitting computation of the first `N` prime numbers by evaluating `first(N,primes)`:

```
first(0,_)       =>  [].
first(_,[])      =>  [].
first(N,[X|Xs])  =>  [X|first(N1,Xs)]  :-  successor(N1,N).

primes  =>  sieve(intsFrom(2)).

intsFrom(N1)  =>  [N1|intsFrom(N)]  :-  successor(N1,N).

sieve([])      =>  [].
sieve([X|Xs])  =>  [X|sieve(filter(Xs,X))].

if(true, X,_)  =>  X.
if(false,_,Y)  =>  Y.

filter([],_)      =>  [].
filter([X|Xs],P)  =>  if(divides(P,X), filter(Xs,P), [X|filter(Xs,P)]).

divides(N,M)  =>  true   :-  M mod N =:= 0.
divides(N,M)  =>  false  :-  M mod N =\= 0.
```

This theory is interesting in that it raises an issue that is often treated as secondary in equational logic or term rewriting, although it is of central importance in functional programming: if this RET is thought of as a program, *the evaluation strategy matters.* Since the stream of integers is infinite, we cannot evaluate **first(10,primes)** by ordinary innermost evaluation: evaluation of **intsFrom(2)** will never terminate. Fortunately, however, outermost evaluation [17] is sufficient for the job.

We will show later that the RET paradigm permits implementation of outermost evaluation very naturally, by considering the ordering of clauses. In a Prolog implementation of RET, positioning the rule reference clause at the *start* of the `eq1` definition causes rule matching at the outermost level to be attempted first (before substitivity considers matching on subterms). Thus, outermost rewriting is obtained. A consequence of putting the rule reference clause before substitutivity clauses (at least those defining substitutivity for the arguments of functions whose evaluation can require potentially infinite computations) is that potentially infinite computations are deferred as long as is possible. Furthermore, as will be discussed in Section 4, some substitutivity axioms can be omitted with no change in semantics. With the RET below, Prolog will solve `?- eq(first(10,primes), P)` with `P = [2,3,5,7,11,13,17,19,23,29]`:

```
eq1( X,            Y            ) :- (X => Y).     % rule reference clause
eq1( [H|T1],       [H|T2]       ) :- eq1(T1,T2).
eq1( if(X1,Y,Z),   if(X2,Y,Z)   ) :- eq1(X1,X2).
eq1( first(X,Y1),  first(X,Y2)  ) :- eq1(Y1,Y2).   % stream argument
eq1( sieve(X1),    sieve(X2)    ) :- eq1(X1,X2).   % stream argument
eq1( filter(X1,Y), filter(X2,Y) ) :- eq1(X1,X2).   % stream argument
```

3 Semantics of Regular Equational Theories

The semantics of RET programs are easy to summarize informally. Basically, the predicate eq_1 corresponds to a single narrowing step, while eq_2 corresponds to a (possibly empty) sequence of such steps. Such a sequence need not end in a canonical form; we are interested in conditions under which such would be the case. In this section we present results that more formally characterize the semantics (actually SLD trees) of regular equational theories.

3.1 Preliminaries

To make the discussion more precise we use some of the concepts familiar from related literature. We briefly review them here, assuming the reader is familiar with SLD trees [11].

Rewriting and narrowing are closely related — in fact the former is a special case of the latter. Hence we begin with the definition of narrowing:

Given a term σ (possibly containing variables) and a set R of rewrite rules, we say that σ narrows in one step to τ if
(1) σ has, at position p, a nonvariable subterm σ' that unifies, with most general unifier θ, with the left-hand side of a rule $(l \Rightarrow r)$ in R (in which each variable has been renamed to differ from any in σ, and any variable in r also appears in l) and if
(2) τ is the result of replacing by $r\theta$ the subterm at position p in $\sigma\theta$.

Suppose now that σ is a ground term. Then θ leaves σ unchanged and acts as a ground substitution on l to give the subterm σ'. In this special case the above definition becomes the one for rewriting.

Earlier we stated that conditional rules of the form

$$(l \ \Rightarrow \ r) \quad \leftarrow \quad q$$

are permitted provided that the condition q always has a finite proof tree. This is a useful practical generalization from the situation in which rules must have the conditionless form $(l \ \Rightarrow \ r)$. However it is indistinguishable as far as rewriting or narrowing is concerned, since any correct answer substitutions θ resulting from a proof with a conditional rule is indistinguishable from a substitution obtained from a rule with no condition. Losing no generality, then, proofs in the remainder of this section will assume that rules have no conditions, Here we will also assume that every variable in r also appears in l, so that rewriting of a ground term σ always produces a ground term τ.

3.2 SLD Trees obtained by Regular Equational Theories

Let the set P of clauses be the result of translating a set R of rewrite rules. If σ is any ground term, let T_1 be an SLD-tree for P and the query $\mathrm{eq}_1(\sigma, Z)$.

Lemma 1 *T_1 is finite.*

Theorem 1 *T_1 has a successful derivation that substitutes τ for Z iff σ rewrites to τ under R in one rewrite step.*

Theorem 1 can be demonstrated either proof-theoretically or model-theoretically. Model-theoretically, we interpret eq_1 as a *binary relation* among ground terms that holds iff the first element of a pair rewrites in one step to the second element.

Corollary 1 *T_1 is finitely failed iff σ is a canonical term.*

Corollary 2 *Let T_2 be an SLD-tree for P and the query $eq_2(\sigma, Z)$.
Then T_2 has a successful derivation that substitutes τ for Z iff σ rewrites to τ under R.*

Note that Corollary 2 holds independently of the ordering of clauses in P. Hence it does not say anything about where in T_2 the success nodes occur. In other words, it does not say anything about whether Prolog finds a success node.

Corollary 3 *If there is no infinite rewriting sequence starting from σ and if the reflexive clause for eq_2 occurs after the transitive clause for eq_2, then Prolog execution of $eq_2(\sigma, Z)$ with program P yields as first answer a canonical form of σ.*

3.3 Regular Equational Theories as Regular Expressions

The previous discussion shows that the semantics of the equational paradigm are naturally summarized using binary relations: $\mathrm{eq}_2 = \mathrm{eq}_1^*$, the reflexive transitive closure of eq_1. These are potentially infinite binary relations over the terms in the Herbrand universe U_L (ground terms that can be formed with function symbols and constants of a particular logic language L [11]). Viewing regular equational theories as relations gives tremendous perspective on their semantics.

Any proof of an equation using the equational axioms **T**, **R**, **F**, **E** clearly can be summarized by a sequence of these symbols. Here, the right-linear recursion of the RET paradigm restricts application of the axioms, so that its *equational theories can be represented by sequences of symbols like regular expressions.* This is the whole justification for calling it the Regular Equational Theory paradigm.

Corresponding to the equational axioms, define the following binary relations on the Herbrand Universe:

- $\mathbf{R}$ denotes the identity or reflexivity relation, satisfying $x \; \mathbf{R} \; x$ for all terms x in U_L.
- $\mathbf{E}$ denotes the binary relation of (user-defined) rewriting, denoting the set of pairs $x \; \mathbf{E} \; y$ such that $(x \; {=>} \; y)$ is a ground instance of a user-defined rule.
- If P is any binary relation, and $\mathbf{F}$ is a set of **indexed function symbols** (function symbols f along with the index i of some argument of f), then $\mathbf{F}[P]$ denotes one-step functional substitutivity with respect to P, denoting the binary relation

$$f(t_1, \ldots, t_{i-1},\; x,\; t_{i+1}, \ldots, t_n) \;\; \mathbf{F}[P] \;\; f(t_1, \ldots, t_{i-1},\; y,\; t_{i+1}, \ldots, t_n)$$

for each x and y such that $(x \;\, P \;\, y)$, and for every function symbol f having argument index i in $\mathbf{F}$.

If P and Q are binary relations:

- $(P \cup Q)$ denotes the union of P and Q.
- $(P \; , \; Q)$ denotes the binary relation composition of P and Q. That is, $(P \; , \; Q) \; = \; \{ \; (x, z) \,|\, (x, y) \in P, \; (y, z) \in Q \; \}$
- P^* denotes the reflexive transitive closure of P, and satisfies the recursive definition

$$P^* \quad \stackrel{def}{=} \quad (P \;\; , \;\; P^*) \;\; \cup \;\; \mathbf{R}.$$

- The relation of $\mathbf{F}$-substitutivity with respect to P is defined by (overloading the '$*$' notation mildly)

$$\mathbf{F}^*[P] \quad \stackrel{def}{=} \quad P \;\; \cup \;\; \mathbf{F}[\,\mathbf{F}^*[P]\,] \quad = \quad P \;\; \cup \;\; \mathbf{F}[P] \;\; \cup \;\; \mathbf{F}[\mathbf{F}[P]] \;\; \cup \;\; \mathbf{F}[\mathbf{F}[\mathbf{F}[P]]] \;\; \cup \;\; \cdots$$

For example, suppose $(l \; \mathbf{E} \; r)$. Let τ be a term containing l as a subterm at p, a position corresponding to a path that traverses only arguments of function symbols in $\mathbf{F}$. Then $(\tau \;\, \mathbf{F}^*[\mathbf{E}] \;\, \tau')$ where τ' is the result of replacing l at position p in τ by r.

A **regular binary relation (RBR)** is a binary relation, defined inductively as follows:

1. $\mathbf{R}$ and $\mathbf{E}$ are regular binary relations.
2. If P is a regular binary relation, then $\mathbf{F}[P]$ and $\mathbf{F}^*[P]$ are also.
3. If P and Q are regular binary relations, then $(P \cup Q)$, $(P \; , \; Q)$, and P^* are also.

Regular binary relations have an interesting theory of their own. Letting '$\equiv$' denote equality among binary relations, a simple result that will be exploited later is as follows:

Theorem 2 *If P is any regular binary relation and $\mathbf{F} = \{f_i\}$ where f_i is any indexed function symbol, then*

$$(\,\mathbf{F}[P]\,)^* \quad \equiv \quad \mathbf{F}[P^*].$$

Theorem 3 *Every theory implementable with the RET paradigm is a regular binary relation.*

Proof With the RET paradigm, $\mathrm{eq}_1 \; = \; \mathbf{F}^*[\mathbf{E}]$, $\mathrm{eq}_2 \; = \; \mathrm{eq}_1^*$, $\mathrm{eq} \; = \; \mathrm{eq}_2$, and thus $\mathrm{eq} \; = \; (\,\mathbf{F}^*[\mathbf{E}]\,)^*$.
□

Theorem 3 gives a nice characterization of the expressive power of the RET paradigm. It also suggests the paradigm is well-suited to implementation in Prolog, which favors linear (stack-oriented) recursion. However, Prolog relies upon a computation rule that orders rules top-to-bottom. An extension of this theorem that takes rule ordering into account is derivable only with a deeper understanding of rule ordering.

4 Evaluation Strategies and Rule Ordering

It is natural to view the RET paradigm as an **evaluation paradigm** — a way to reduce functional expressions (first-order terms) to 'values' (canonical terms) that cannot be rewritten further. This section justifies this view. In fact, Regular Equational Theories can implement not only standard evaluation strategies like innermost and outermost evaluation, but also nonstandard strategies with interesting properties.

4.1 Evaluation Strategies

An **evaluation strategy** is a mapping from terms to terms that specifies the selection and replacement of a (sub)term by another term. It thus defines a one-step rewrite relation. Usually one is interested in computationally useful (effective, recursive) strategies for evaluation of a term.

Here let us assume that evaluation is functional evaluation, defined by **rules** such as

$$f(t_1, \ldots, t_n) \quad = \quad g.$$

In this rule, f is an n-ary function symbol ($n \geq 0$), $t_1, \ldots, t_n$ are terms possibly containing variables, and g is a term such that every variable in g also appears among $t_1, \ldots, t_n$.

A non-variable term is called **canonical** if neither it nor any of its subterms unifies with the left side of a rule.

Perhaps the best known way to evaluate a functional expression is to evaluate all arguments before beginning the evaluation of the function itself. This is called **eager**, **applicative order**, **call-by-value**, or **innermost evaluation**. It can be defined recursively by:

$$\mathrm{inner}(f(s_1, \ldots, s_n)) \quad = \quad \begin{cases} f(s_1, \ldots, \mathrm{inner}(s_i), \ldots, s_n) & \text{if } s_i \text{ is the leftmost non-canonical argument} \\ g\theta & \text{otherwise,} \\ & \text{if } \theta \text{ is a mgu for } f(s_1, \ldots, s_n) \text{ and } f(t_1, \ldots, t_n). \end{cases}$$

A more sophisticated evaluation strategy is to interleave evaluation of a function with evaluation of its arguments, doing just enough evaluation of the arguments (possibly none!) before going ahead and rewriting the whole expression using a rule for the function. This is called **lazy**, **normal order**, **call-by-name**, or **outermost evaluation**. It can be defined recursively by:

$$\mathrm{outer}(f(s_1, \ldots, s_n)) \quad = \quad \begin{cases} g\theta & \text{if } \theta \text{ is a mgu for } f(s_1, \ldots, s_n) \text{ and } f(t_1, \ldots, t_n) \\ f(s_1, \ldots, \mathrm{outer}(s_i), \ldots, s_n) & \text{otherwise,} \\ & \text{if } s_i \text{ is the leftmost non-canonical argument.} \end{cases}$$

Outermost evaluation is essential when operating on (potentially) infinite structures, such as streams. Innermost evaluation cannot be used successfully for these. A commonly given example of stream processing is the program that computes primes by sieving the sequence of all natural numbers, as we saw earlier.

There are important performance tradeoffs between evaluation strategies. Innermost evaluation has a disadvantage when it is not necessary to evaluate the arguments of a function completely in order to obtain its value. On the other hand, outermost evaluation is slower when function arguments must be evaluated completely, as partially-evaluated expressions are often copied multiple times and these copies evaluated independently. Only recent improvement in compilation techniques have put into question the traditional belief that lazy (outermost) functional programming languages are impractical. Laziness can pay off, but only when it is used intelligently.

4.2 Implementing Evaluation Strategies

The most interesting result of this section is that with the RET paradigm we can freely combine innermost and outermost evaluation — and we can do this merely by reordering the clauses of eq_1. The RET paradigm offers new insights into innermost and outermost evaluation, and is very good for developing new evaluation strategies.

The eq_1 relation defines an evaluation strategy. Consider a goal `eq1(U,V)`, where `U` is a ground term. If this goal resolves with the rule reference clause `eq1(X,Y) :- (X => Y)`, then an outermost reduction step is performed. Otherwise, if such a goal resolves with a substitutivity clause, then any outermost reduction is suspended and an innermost reduction step is attempted. As Prolog clauses are presented for resolution in textual order, one can see that our translation has the following property:

> *Innermost reduction is performed on those arguments for which the substitutivity clause occurs before the rule reference clause, and outermost reduction for the other arguments.*

Thus for each function symbol, and for each argument place within each function symbol, we can select whether to evaluate innermost or outermost.

Below we express three evaluation strategies with regular equational theories in Prolog, and illustrate their differences with the following sample set of rewrite rules:

```
plus(X,0)       =>  X.
plus(X,s(Y))    =>  s(plus(X,Y)).

if(true, X,_)   =>  X.
if(false,_,Y)   =>  Y.
```

Both of the RETs below extend the definitions for the 'inner' and 'outer' functions in a significant way: they omit the word "otherwise" from their definitions. Instead of a *function*, `eq1` defines a *relation*. The rule ordering of Prolog considers the use of the rule reference clause in the same order as in the definitions of 'inner' and 'outer', but permits alternative evaluations as well (via backtracking). The effect is identical to these functions whenever the first solution obtained by Prolog is all that is wanted (which is usual), but in some situations the relational semantics of `eq1` can be exploited. Also, the duality between these two evaluation strategies is clearest when the word "otherwise" is omitted.

Innermost Rewriting

Innermost evaluation first evaluates arguments of a function, and the function itself afterwards. As formalized above, arguments are evaluated left-to-right, giving **leftmost innermost evaluation**; but of course any selection rule is permissible. A RET implementing leftmost innermost evaluation for the rules above is:

```
eq(X,Y) :- eq2(X,Y).

eq2(X,Z) :- eq1(X,Y), eq2(Y,Z).                        % T
eq2(X,X).                                              % R

eq1( s(X1),         s(X2)        ) :- eq1(X1,X2).   % {s_1}
eq1( plus(X1,Y ),   plus(X2,Y )  ) :- eq1(X1,X2).   % {plus_1}
eq1( plus(X, Y1),   plus(X, Y2)  ) :- eq1(Y1,Y2).   % {plus_2}
eq1( if(X1,Y, Z ), if(X2,Y, Z ) ) :- eq1(X1,X2).   % {if_1}
eq1( if(X, Y1,Z ), if(X, Y2,Z ) ) :- eq1(Y1,Y2).   % {if_2}
eq1( if(X, Y, Z1), if(X, Y, Z2) ) :- eq1(Z1,Z2).   % {if_3}
eq1( X,             Y            ) :- (X => Y).     % E:  Innermost evaluation
```

Outermost Rewriting

Outermost evaluation evaluates an expression first without evaluating its arguments if possible, and if not then incrementally evaluates arguments in some fixed order (usually left-to-right, giving **leftmost outermost rewriting**). A RET implementing leftmost outermost evaluation for the rules above is:

```
eq(X,Y) :- eq2(X,Y).

eq2(X,Z) :- eq1(X,Y), eq2(Y,Z).                        % T
eq2(X,X).                                              % R

eq1( X,             Y            ) :- (X => Y).     % E:  Outermost evaluation
eq1( s(X1),         s(X2)        ) :- eq1(X1,X2).   % {s_1}
eq1( plus(X1,Y ),   plus(X2,Y )  ) :- eq1(X1,X2).   % {plus_1}
eq1( plus(X, Y1),   plus(X, Y2)  ) :- eq1(Y1,Y2).   % {plus_2}
eq1( if(X1,Y, Z ), if(X2,Y, Z ) ) :- eq1(X1,X2).   % {if_1}
eq1( if(X, Y1,Z ), if(X, Y2,Z ) ) :- eq1(Y1,Y2).   % {if_2}
eq1( if(X, Y, Z1), if(X, Y, Z2) ) :- eq1(Z1,Z2).   % {if_3}
```

These RETs also extend functional programming and conventional term rewriting in another noteworthy way. Above, in the rule

$$f(t_1, \ldots, t_n) \quad = \quad g.$$

the terms $t_1, \ldots, t_n$ were not really restricted (except that the variables in g were required to appear among them). This is significant:

- In functional programming, the t_i are normally restricted to be distinct variables.
- In term rewriting, the t_i are restricted to be either variables or canonical terms, and often further that no variable appears more than once among the t_i.

These restrictions certainly result in useful systems, and the examples in this paper follow the term rewriting restrictions, since they make it easier to reason about the resulting semantics. However there are useful RETs that do not obey these restrictions. For example:

```
twice : F : X  =>  F : (F : X).
succ : X       =>  s(X).
```

The rules given earlier for `twice` give better performance, but it is important to realize that these two sets of rules have virtually identical semantics.

'Just-in-Time' Rewriting

Often the substitutivity axioms are not useful, particularly when we are interested in outermost (lazy) rewriting. For example, with the definition of `plus`

```
plus(X,0)      =>  X.
plus(X,s(Y))   =>  s(plus(X,Y)).
```

substitutivity on the first argument will *never* help in successful unification, since the rules are *linear* (i.e., no variable appears more than once in the left of any rule) and the first argument of `plus` is always variable. Use of substitutivity on this argument in any proof will have the effect of evaluating the argument, even though the value of the argument may not be needed later. A more striking example comes from the rules

```
if(true, X,_)  =>  X.
if(false,_,Y)  =>  Y.
```

in which it is clear that evaluation of the second and third argument is not wanted in general: the if-then-else function is **non-strict**, i.e., its value does not necessarily depend on the value of its arguments.

Just-in-time rewriting is like outermost rewriting then, but differs in that substitutivity rules are included *only* for those function symbols appearing as the outermost symbol in a user-defined rule, and then only for those arguments for which one or more user-defined rules have a nonvariable term in that argument position. For instance, with the rules above we obtain the following RET:

```
eq(X,Y) :- eq2(X,Y).

eq2(X,Z) :- eq1(X,Y), eq2(Y,Z).                       % T
eq2(X,X).                                             % R

eq1( X,             Y            ) :- (X => Y).       % E:  Outermost evaluation
eq1( plus(X, Y1),  plus(X, Y2)  ) :- eq1(Y1,Y2).      % {plus_2}
eq1( if(X1,Y, Z ), if(X2,Y, Z ) ) :- eq1(X1,X2).      % {if_1}
```

Just-in-time rewiting can give significant performance improvements. The just-in-time RET for the `primes` program shown earlier runs about twice as fast as the ordinary outermost RET.

5 Implementation Method

The method described above is already an implementation method, since for example the Prolog programs above are all executable as shown. For small problems it is adequate. Compilation techniques can, however, yield faster programs.

5.1 Compilation Techniques

We have used three techniques in compiling the RET paradigm:

1. **Strategy compilation** improves performance by applying a program transformation that replaces the clauses for `eq1`, `eq2`, etc. by an equivalent but faster program, exploiting some fact about the evaluation strategy (outermost, innermost, etc.).

2. **Partial evaluation** specializes the equational theory for particular queries, and evaluates subexpressions at compile time wherever possible [19]. We have partially evaluated `eq1` first, and partially evaluated `eq2` afterwards.

3. **Relationalization** converts goals such as `eq(f(X),Y)` to the relational form `eq_f(X,Y)`. In systems like Prolog, this change can increase speed by exploiting built-in indexing and reducing structure building. Relationalization is actually a folding strategy used in terminating partial evaluation [19].

These techniques can be used singly or in combination. We have used them together, in the order above.

The first technique is the most interesting. Equivalences among regular binary relations give program transformation rules. The equivalence

$$(\,[\mathbf{E}]\mathbf{F}^*\,)^* \quad \equiv \quad (\,\mathbf{E}^*\ ,\ \ \mathbf{F}[\,[\mathbf{E}]\mathbf{F}^*\,]\,)^*$$

says that, in outermost evaluation, argument evaluation can be deferred until outermost evaluation fails. This is a standard way to compile lazy evaluation. The equivalence

$$(\,\mathbf{F}^*[\mathbf{E}]\,)^* \quad \equiv \quad (\,(\,\mathbf{F}[\,\mathbf{F}^*[\mathbf{E}]\,]\,)^*\ ,\ \ \mathbf{E}\,)^*$$

says that innermost evaluation permits function arguments to be completely evaluated first, with function evaluation only afterwards. Again, this is a standard functional program compilation technique.

We can do still better. Consider innermost evaluation with the RET paradigm. Its ordered regular expression $(\,\mathbf{F}^*[\mathbf{E}]\,)^*$ corresponds to the program:

```
eq(X,Y) :- eq2(X,Y).                 %  eq2       =  eq   =  (F*[E])*

eq2(X,Z) :- eq1(X,Y), eq2(Y,Z).      %  (eq1)+
eq2(X,X).                            %  (eq1)*    =  eq2

eq1(f(X1),f(X2)) :- eq1(X1,X2).      %   F+
eq1(X,Y) :- (X => Y).                %   F*[E]    =  eq1
```

This program can be improved upon with one another result:

Theorem 4

$$(\,\mathbf{F}^*[\mathbf{E}]\,)^* \quad \equiv \quad (\,\mathbf{F}^*[(\,\mathbf{E}\ \ ,\ \ (\mathbf{F}^*[\mathbf{E}]\,)^*\,)]\,)^*$$

For example, if $\mathbf{F} = \{f_1\}$, the right hand expression in Theorem 4 corresponds to the program:

```
eq(X,Y) :- eq2(X,Y).                 %  eq2                 =  eq   =  (F*[E])*

eq2(X,Z) :- eq1(X,Y), eq2(Y,Z).      %  (eq1)+
eq2(X,X).                            %  (eq1)*              =  eq2

eq1(f(X1),f(X2)) :- eq1(X1,X2).      %   F+
eq1(X,Z) :- (X => Y), eq2(Y,Z).      %   F*[( E, eq2 )]     =  eq1
```

With most Prolog systems this program will perform better than the original because it applies **E** rules as much as possible before exiting recursions initiated by substitutivity, rather than merely apply one **E** step at a time. This saves not only in `eq1` steps and stack management, but with WAM-like Prolog implementations saves also in structure creation and garbage collection, since the substitutivity rules create structures such as `f(X2)` only when their arguments `X2` are the results of *complete* (eq_2) evaluations of `X1`, instead of the one-step (eq_1) evaluations of `X1` that would occur with the unmodified RET paradigm.

5.2 The Effect of Compilation Techniques on Performance

In [3], some analysis of the performance of logic programs using a variant of the basic RET paradigm indicated that it executes very fast with modern Prolog implementations. In fact, for the `twice` program shown earlier, Prolog is *faster* than many Lisp systems since they lack efficient implementations of combinators.

In this section, let us apply all of the techniques discussed in this paper, and show that, from this already fairly fast program, *we can further extract large factors in improved performance with appropriate compilation techniques.* For all the reasons discussed earlier (simplicity, comprehensibility, compilability, etc.) this suggests the RET paradigm can implement computationally interesting subtheories of equality effectively, and can be useful as a general programming tool.

Consider the `twice` program shown earlier, revised according to Theorem 4 as just described, and revised to use machine arithmetic in order to deemphasize the role of structure creation in its performance:

```
eq(X,Y) :- eq2(X,Y).                 % eq2              = eq   =  (F*[E])*

eq2(X,Z) :- eq1(X,Y), eq2(Y,Z).      % (eq1)+
eq2(X,X).                            % (eq1)*           = eq2

eq1((X1:Y),(X2:Y)) :- eq1(X1,X2).    % {:_1}+
eq1((X:Y1),(X:Y2)) :- eq1(Y1,Y2).    % F+
eq1(X,Z) :- (X => Y), eq2(Y,Z).      % F*[( E , eq2 )]  = eq1

succ : X   =>  Y   :-   Y is X+1.
twice : F  =>  t(F).
t(F) : X   =>  F : (F : X).
```

This program is much faster than the original; see the table below. But we can do still better. Partial evaluation can now be applied to this program. With a system like the one described in [19], partial evaluation of `eq1` yields:

```
eq1(A:B, C:B) :- eq1(A, C).
eq1(A:B, A:C) :- eq1(B, C).
eq1(succ:B, A) :- C is B+1, eq2(C, A).
eq1(twice:B, A) :- eq1(t(B), C), eq1(C, D), eq2(D, A).
eq1(twice:B, A) :- eq1(t(B), A).
eq1(twice:A, t(A)).
eq1(t(B):A, B:(B:A)).
```

This can then be relationalized, giving:

```
eq1(A:B,C) :- 'eq1_:'(A,B,C).

'eq1_:'(A,B,C:B) :- eq1(A,C).
'eq1_:'(A,B,A:C) :- eq1(B,C).
'eq1_:'(succ,A,B) :- C is A+1, eq2(C,B).
'eq1_:'(twice,A,t(A)).
'eq1_:'(t(A),B,C) :- 'eq1_:'(A,A:B,D), eq2(D,C).
'eq1_:'(t(A),B,A:(A:B)).
```

Finally `eq2` can be partially evaluated and relationalized, yielding:

```
eq2(A:B,C) :- 'eq2_:'(A,B,C).
eq2(A,A).

'eq2_:'(A,B,C) :- eq1(A,D), 'eq2_:'(D,B,C).
'eq2_:'(A,B,C) :- eq1(B,D), 'eq2_:'(A,D,C).
'eq2_:'(succ,A,B) :- C is A+1, eq2(C,B).
'eq2_:'(twice,A,t(A)).
'eq2_:'(t(A),B,C) :- 'eq2_:'(A,A:B,D), eq2(D,C).
'eq2_:'(A,B,A:B).
```

Further compilation techniques can take advantage of the specific structure of a problem. Consider:

Theorem 5 *If* $\mathbf{F}$ *consists of two different indexed function symbols* $\mathbf{F}_1$ *and* $\mathbf{F}_2$*, then*

$$(\ \mathbf{F}^*[\mathbf{E}]\ )^* \quad \equiv \quad (\ \mathbf{F}_1[(\ \mathbf{F}^*[\mathbf{E}])^*\ ]\ ,\ \mathbf{F}_2[\ (\mathbf{F}^*[\mathbf{E}])^*\ ]\ ,\ \mathbf{E}\ )^*.$$

The proof relies directly on Theorem 2 shown earlier. Theorem 5 then implies that for this program, an extremely fast `eq` program is possible:

```
eq(X1:Y1,Z) :-  eq(X1,X2), eq(Y1,Y2), (X2:Y2 => XY), eq(XY,Z).
eq(X,X).
```

Below is a table of resulting timings for the `twice` program when this and the preceding compilation techniques are used. All times were obtained on a Sun Sparc 10 with Sicstus Prolog 2.1 #9 and compiled execution, averaged over a number of repetitions ranging from 5000 for the first benchmark goal to 10 for the last. The '(overflow)' entry below is a stack overflow, a result of excessive structure creation.

Benchmark goal	`eq` *program* $(\mathbf{F}^*[\mathbf{E}])^*$	*revised* `eq` $(\mathbf{F}^*[\mathbf{E},\ (\mathbf{F}^*[\mathbf{E}])^*])^*$	*partially evaluated revised* `eq`	`eq` *specialized for* ':'
eq(twice:twice:twice:succ:0, 16)	2.16 msec	2.30 msec	1.28 msec	1.06 msec
eq(twice:twice:(twice:twice:twice:succ):0, 64)	8.39 msec	7.10 msec	3.67 msec	2.91 msec
eq(twice:twice:(twice:twice):succ:0, 256)	36.79 msec	25.68 msec	12.94 msec	10.30 msec
eq(twice:twice:(twice:twice):(twice:twice:succ):0, 1024)	168.4 msec	98.7 msec	49.4 msec	39.1 msec
eq(twice:twice:twice:twice:succ:0, 65536)	(overflow)	6340.8 msec	3215.3 msec	2562.4 msec

6 Conclusions

Some problems (like the normalization of conjunctions considered earlier) are just naturally term rewriting problems; others are strict functional programs; others are inherently lazy. Rather than demand a single approach for handling all of these problems, it is advantageous to accommodate many. The Regular Equational Theory (RET) paradigm unifies many of these problem-solving approaches, and is a useful complement to logic programming environments. Despite its simplicity, it yields useful implementations immediately, and with various compilation techniques can attain impressive speed.

The RET paradigm gives new perspectives on several approaches for implementing equational theories. Often implementations of term rewriting, algebraic simplification, functional programming, etc. can all be viewed as instances of the method here, with specific evaluation strategies and compilation techniques. Surveying these approaches with this common framework in mind suggests implementation variations and directions for further work.

References

[1] J.A. Bergstra and J.W. Klop. Conditional rewrite rules: Confluence and termination. *J. Comput. System Sci.*, 32:323–362, 1986.

[2] M.H.M. Cheng. *Lambda-Equational Logic Programming.* Doctoral dissertation, Dept. of Computer Science, University of Waterloo, 1987.

[3] M.H.M. Cheng, M.H. van Emden, B. Richards. On Warren's Method for Functional Programming in Logic. *Proc. Seventh Intnl. Conf. on Logic Programming*, Jerusalem. Boston, MA: MIT Press, 546–560 (1990).

[4] M.H.M. Cheng, D.S. Parker, M.H. van Emden. Prolog Technology Term Rewriting: Applicative Systems. *Proceedings of HOA'93*, Amsterdam, CWI, September 1993.

[5] Mantis H.M. Cheng and Keitaro Yukawa. AP: an assertional programming system. In R.W. Wilkerson, editor, *Advances in Logic Programming and Automated Reasoning, volume 1.* Ablex Publishing, pp. 120–134, 1992.

[6] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, volume B*, pages 243–320. North Holland, 1990.

[7] S. Hölldobler. *Foundations of Equational Logic Programming.* NY: Springer-Verlag, Lecture Notes in Artificial Intelligence No. 353, 1989.

[8] J.-P. Jouannaud, C. Kirchner. Solving Equations in Abstract Algebras: A Rule-Based Survey of Unification. in: *Computational Logic: Essays in Honor of Alan Robinson*, MIT Press, 257–321, 1991.

[9] C. Kirchner, ed. *Unification.* NY: Academic Press, 1990.

[10] R.A. Kowalski. *Logic for Problem-Solving.* Elsevier North-Holland, 1979.

[11] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2nd edition, 1987.

[12] S. Narain. A Technique for Doing Lazy Evaluation in Logic. *J. Logic Programming* **3**:3, 259–276 (October 1986).

[13] S. Narain. *LOG(F): An Optimal Combination of Logic Programming, Rewrite Rules and Lazy Evaluation.* Ph.D. Dissertation, UCLA Computer Science Dept., 1988.

[14] D. Stott Parker. *Bop 0.2 Manual.* Technical Report CSD-9200027, UCLA Computer Science Department, June 1992. Available as `file://pop.cs.ucla.edu/pub/bop/BopManual.ps.Z`.

[15] F.C.N. Pereira and D.H.D. Warren. Definite Clause Grammars for Language Analysis — a Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence* **13**:3, 231–278 (1980).

[16] G.E. Peterson. Complete Sets of Reductions with Constraints. M.E. Stickel, ed., *Proc. 10th CADE*, Springer LNCS #449, 381–395, 1990.

[17] S.L. Peyton Jones. *The Implementation of Functional Programming Languages.* Englewood Cliffs, NJ: Prentice-Hall, 1987.

[18] B. Richards. *Contributions to Functional Programming in Logic.* M.S. Thesis, Department of Computer Science, University of Victoria, 1990.

[19] D. Sahlin. An Automatic Partial Evaluator for Full Prolog. Ph.D. dissertation, Dept. of Telecommunications and Computer Systems, Royal Institute of Technology (KTH), S-100 44 Stockholm Sweden. Also available from: SICS, Box 1263, S-164 28 Kista, Sweden.

[20] M.H. van Emden and K. Yukawa. Logic programming with equations. *The Journal of Logic Programming*, 4:265–288, 1987.