Equational Logic Programming: An Extension to Equational Programming

Jia-Huai You
Department of Computer Science
Rice University

P.A. Subrahmanyam
AT&T Bell Laboratories Research

Abstract

The paradigm of equational programming potentially possesses all the features provided by Prolog-like languages. In addition, the ability to reason about equations, which is not provided by Prolog, can be accommodated by equational languages. In this paper, we propose an extended equational programming paradigm, and describe an *equational logic* programming language which is an extension of the equational language defined in [Hoff82]. Semantic foundations for the extension are discussed. The extended language is a powerful logic programming language in the sense of Prolog and thus enjoys the programming features that Prolog possesses. Furthermore, it provides an ability to solve equations, which captures the essential power of equational programming.

1. Introduction

Equations provide a powerful computational paradigm and may be used to program all of the computable functions. Among a number of proposals, Hoffmann and O'Donnell introduced an equational programming language which was based on equational logic [Hoff82, Hoff84, O'Do77]. A program written in the language is a set of equations which are computationally used as rewrite rules, and the execution of a program is a simplification process involving the replacement of equal by equals using *reduction*. The theoretical basis of the language is that if each equation in a given equational program is *left-linear* and the set of all equations as a whole is *closed*, then the program possesses the confluence property which is sufficient to guarantee the completeness of logical consequences of equality. The language is not constrained to have the strong termination property (which requires that *every* reduction sequence terminate), provides a form of lazy evaluation, and can be used to specify infinite data structures, non-terminating computations. Associated with the language are two major merits: its simple semantics which is based on logical consequences of equality and the mechanisms that enable an efficient implementation.

In other developments, logic programming has emerged as a very promising programming paradigm in recent years, mainly due to Kowalski's formalization [Kowa74], and various efforts at efficient implementations, most notably the DEC-10 PROLOG system [Pere78]. From the perspective of functional and equational programming, Prolog offers new programming features, such as (a) rule-based program construction; (b) computations based on relations rather than functions; and (c) logical variables that enable suspended variable bindings in a computation. All these features contribute to a key characteristic of Prolog that makes it different from other programming paradigms: it generates solutions by *reasoning* (or "solving for variables") rather than direct computing. This reasoning ability is mainly responsible for the declarative programming style provided in the logic programming paradigm. Prolog, however, does not provide facilities for reasoning about equalities, which, as argued by Kornfeld [Korn83], will be likely to play an important role in future logic programming practices.

Traditional equality-based equational languages, like Hoffmann and O'Donnell's language, provide mechanisms to compute a semantically equivalent expression from a given expression by the mechanism called reduction or rewriting. The equivalence of a computed expression and the original expression is therefore a *logical consequence* of the given equational program. Expressions here are called (first-order) *terms* which are composed of function symbols and variables. A logical consequence of a program is an assertion which must be true whenever the assertions of the program are all true, no matter how the program is *interpreted*.

We are interested in another computational paradigm in which the user programs by writing abstract equations, and then supplies a question asking whether an equation has a solution. The machine either responds with an answer (or answers) by generating a set of variable bindings which constitute a solution, or never answers. We refer to this computational paradigm as *equational logic programming*: equational programs are accordingly called *equational logic programs*. The paradigm of equational logic programming is actually one of reasoning with equations. Equational logic programs here are assertions of equality. Given an equation, the machine computes a solution, if there exists one, based on all the information provided in the program but nothing else.

Is such a computational paradigm useful? We argue that the answer is yes. First of all, it allows an emulation of Horn clause logic programming. As has been argued by several researchers (see, for example, [Redd85]), Prolog programs may be systematically "translated" into equations defining truth-value functions. An argument along similar lines applies to equational logic programming. Horn clauses of the forms

A.
B :- Q_1, Q_2 , ..., Q_n.

can be rewritten as equations

A = true
B = and(Q_1, Q_2 , ..., Q_n)

where *true* is a constant function symbol, and *and* is a meta-function and returns *true* if all its arguments are "evaluated" to *true*. A Prolog goal of the form

?- P_1, ..., P_n.

can accordingly be rewritten as a conjunction of equations

?- P_1 = true, ..., P_n = true,

or equivalently,

?- H(P_1, ..., P_n) = H(true, ..., true),

where H serves as a new constructor symbol. A solution to this equation is a substitution σ such that the term σH(P_1,...,P_n) is *E-equivalent* to H(true,...,true), i.e., they are equivalent under the *equational theory* described by the corresponding equational logic program. The existence of such a substitution implies that the equation of the goal statment is a logical consequence of the given equational logic program. If there is more than one solution, we may be interested in a set of solutions containing all *most general* ones, i.e., all of the other solutions are instances of, or can be obtained from some solutions in the set. The problem of solving equations described here is actually the one of unification in equational theories as originally formalized in [Plot72]. It has been shown that there exists such a general algorithm that can be used to generate all of the most general solutions [You85b]. It is clear that such an equational logic programming language can have the same expressive power as Prolog-like languages. In addition, it allows negative information to be explicitly specified; higher-order facilities can also be supported (see [Subr84]); last, but not the least, it provides a natural framework into which equality reasoning is incorporated. Because both equational programming and equational logic programming fall into the *same semantic framework* which is based on logical consequences of equality, a language can be designed and implemented in a way that both programming styles are supported. In view of these features, it is clear that the equational logic programming paradigm is useful.

General algorithms for solving equations tend to be very inefficient. The one given in [You85b], for example, is too inefficient to be practically useful. The challenging problem is then to identify useful classes of systems for which there exist efficient solution procedures.

In this paper we describe an equational logic language. In particular, we show how Hoffmann and O'Donnell's language can be extended to an equational logic language. The formal semantics for the proposed language is the classical theory of equality, while the operational semantics consists of two rules—*narrowing* and *deletion upon unification*. Narrowing is an extended mechanism of reduction: reduction uses a matching process (or one-way unification) while narrowing exploits the full power of (two-way) unification. The operational semantics yields a complete evaluation procedure for a large subclass of closed linear equational logic programs (cf. Section 4). We also show how the underlying semantics can be adjusted to reflect the *lazy narrowing* on "conditional terms"—the terms that contain conditional function symbols. such as if_then_else and if_then. Some implementational aspects are also discussed.

2. Previous Efforts and Related Work

A number of authors have recently focused attention on extensions of functional programming languages so that some programming features, initially solely enjoyed by Prolog, can also be provided in the functional programming paradigm.

Functional languages have traditionally used variations of the notion of *reduction* as their operational semantics: replacing equals by equals in ground terms. A common theme of the more recent extensions has been to use a more general replacement mechanism—narrowing or restricted narrowing.

Let P/u denote the subterm of P at occurrence u. A term P *narrows* to a term Q at occurrence u, denoted by

$$P \sim>_{[u,k,\rho]} Q,$$

if and only if there exists an equation $\alpha_k = \beta_k$ such that ρ is the most general unifier of P/u and α_k, and

$$Q = \rho(P[u <\!\!- \rho\beta_k]),$$

where $P[u <\!\!- \rho\beta_k]$ denotes the term obtained by substituting the subterm at occurrence u by the term $\rho\beta_k$. Note that equations here are used as rewrite rules.

Perhaps Bandes was among the first people who realized that the reduction mechanism was incapable of capturing the full semantics of equational specifications. A mechanism, called *constraining-unification* was then proposed [Band84]. The idea of viewing computations as constraint activities among entities or "devices" can be traced back to the work on *constraint programming* [Born81, Stee79]. Constraint programming is like logic programming in that it generates solutions by *solving*. Constraint programming, however, appears to be more of an engineering art than a rigorously based science at the present. Dershowitz introduced an equational programming language based on the Knuth-Bendix completion procedure, which only handles terminating term rewriting systems [Ders84]. Recent interesting work includes: an inclusion of logical variables in the functional programming language FEL [Lind85]; the use of narrowing as the operational semantics for a functional programming language [Redd85]; and regarding programs as (terminating) conditional rewrite rules [Ders85]. The research reported in [Darl85, Sato85, Smol85] is also along these lines.

Some of these extensions are not based on equality; instead they generally rely on some rather complex denotational semantics. For those that do provide a form of equality reasoning, the strong termination property is imposed. The language we are proposing is based on the classical theory of equality, the use of which accrues semantic simplicity and the ability to reason about equations. Furthermore, no *strong* termination property is imposed in our language.

3. Equational Logic Programs

Definition 1. An equational logic program is a set of equations

$$\alpha_0 = \beta_0,\ \alpha_1 = \beta_1,\ \ldots\ldots,\ \alpha_n = \beta_n$$

where variables appearing in β_k must also appear in α_k.
A *goal of equalities* is a conjunction of equalities, and is of the form

$$?\text{-}\ A_0 = B_0,\ A_1 = B_1,\ \ldots,\ A_m = B_m,$$

where A_i and B_i are terms.
There is a distinguished goal, called the empty goal.
□

Two of the commonly used conditional functions can be defined as:

if_then_else(true, X, Y) = X
if_then_else(false, X, Y) = Y
if_then(true, X) = X

where *true* and *false* are two constant function symbols and X and Y are variables.

3.1. Operational Semantics

The operational semantics of the equational logic programs defined here consists of two inference rules: narrowing and deletion upon unification.

Definition 2. Let G be a goal of equalities

$$?\text{-}\ A_0 = B_0,\ A_1 = B_1,\ \ldots,\ A_m = B_m.$$

Narrowing: If A_i is narrowable at occurrence u, $0 \le i \le m$, i.e.,

$$A_i \sim>_{[u,k,\sigma]} A'_i,$$

then G' is a goal derived from G and is of the form

$$?\text{-}\ \sigma(A_0 = B_0),\ \ldots,\ A'_i = \sigma(B_i),\ \ldots,\ \sigma(A_m = B_m).$$

Deletion Upon Unification: If A_i and B_i, $0 \le i \le m$, are unifiable with the most general unifier σ, then G' is a goal derived from G, and is of the form

$$?\text{-}\ \sigma(A_0 = B_0,\ \ldots,\ A_{i-1} = B_{i-1},\ A_{i+1} = B_{i+1},\ \ldots,\ A_m = B_m).$$

We denote by a pair $<G', \sigma>$, the derived goal G' and the unifier it is associated with. □

Definition 3. Let G be a goal of equalities. A successful computation from G is an N-derivation sequence (narrowing based derivations) which is a finite sequence of pairs

$$<G_0, \rho_0>, <G_1, \rho_1>, ..., <G_q, \rho_q>,$$

where $G_0 = G$, ρ_0 is the identity substitution, G_{i+1} is a goal derived from G_i and ρ_{i+1} the unifier associated with this derivation, and G_q is the empty goal.

Let W be the set of variables in G. The computed answer with respect to the above sequence is a substitution restricted to W, and is defined as:

$$\sigma = (\rho_q \circ \rho_{q-1} \circ \cdots \circ \rho_1 \circ \rho_0)|_W. \quad \square$$

Example:

We use the function *append* to illustrate the operational semantics defined above:

append([], L) = L
append(A^X, Y) = A ^ append(X, Y)

The goal

?- append(a^Y^[], c^d^[]) = append(a^b^c^[], d^[])

generates a substitution {Y <— b}.

As was alluded to earlier, Prolog programs may be "translated" into equations in a straightforward way. For example, the predicate append in Prolog

append([], L, L).
append(A^X, Y, A^Z) :- append(X, Y, Z).

can be translated into the following equational logic program

append([], L, L) = true
append(A^X, Y, A^Z) = append(X, Y, Z)

The goal

?- append(a^Y^[], c^d^[], a^b^c^d^[]) = true

also generates the binding {Y <— b}.

The program above can be written in a Lisp-like manner as follows:

append(X, Y, Z) = if X ≡ [] then Y ≡ Z
else and(first(X) ≡ first(Z),
append(rest(X), Y, rest(Z)))

where we assume the availability of certain built-in functions, which may be defined as:

first(A^X) = A
rest(A^X) = X
and(true, true) = true
and(false, X) = false
and(X, false) = false
A ≡ A = true.

The goal

?- append(a^Y^[], c^d^[], a^b^c^d^[]) = true

also generates the binding {Y <— b}.

The equality predicate ≡ defined above can be directly supported by the operational semantics. An equality A ≡ B is simply "evaluated" by narrowing and deletion upon unification, returning *true* when the empty goal is derived, and *false* when derivation stops with a non-empty goal (because of undecidability, computations of this type may not always terminate). We can thus write equations like:

f(X) = if p(a) ≡ q(a) then g(X)

which simulates the following clause in Prolog with equality

f(X) = g(X) :- p(a) = q(a).

4. Semantic Foundations

In this section we show that operational semantics defined in the last section are faithful to the classical theory of equality for a large subclass of the *closed linear* equational logic programs. That is, for this subclass of equational logic programs, whenever there exists a solution for a given equation, the generation of this solution (or a more general one) is guaranteed by the operational semantics; and whenever the empty goal is derived, the computed answer is indeed a solution of the equation.

Definition 4. Let E be an equational logic program. Let F be the set of function symbols that occur in E and V be a set of variables, such that $V \cap F = \phi$. The *free algebra* over V, denoted by $T(F,V)$ contains the set of all terms constructed from the function symbols in F and the variables in V.

The classical theory of equality is defined as the E-equality, denoted by $=_E$, which is the finest congruence on $T(F,V)$, closed under instantiation, i.e., if A = B is in E, then $\sigma A = \sigma B$ is also in E for any substitution σ that maps from variables to terms in $T(F,V)$. $\square$

Definition 5. A substitution σ is a solution of the system of simultaneous equations

$$A_0 = B_0, A_1 = B_1, ..., A_m = B_m,$$

if and only if

$$\sigma A_0 =_E \sigma B_0, \sigma A_1 =_E \sigma B_1, ..., \sigma A_m =_E \sigma B_m.$$

An equational logic program P denotes the set of all systems of simultaneous equations that have a solution. $\square$

Note that solving a goal of equalities of the form

$$?- A_0 = B_0, A_1 = B_1, ..., A_m = B_m$$

is equivalent to solving the goal

$$?- H(A_0, A_1, ..., A_m) = H(B_0, B_1, ..., B_m),$$

where H is a function symbol that is not in F.

Definition 6 Let E be an equational logic program. Let W be the set of variables in an equation P = Q. The notion of E-equality is extended to substitutions *restricted* to W as follows:

$\sigma =_E \rho$ [W] iff $\forall x \in W\ \sigma x =_E \rho x$.

Let both σ and ρ be solutions of P = Q. We say σ is more general than ρ under the equational logic program E and restricted to W, denoted by

$$\sigma \leq_E \rho \text{ [W]},$$

if and only if there exists a substitution η such that the composition $\eta \circ \sigma$ is E-equivalent to ρ, i.e.,

$$\eta \circ \sigma =_E \rho \text{ [W]}.$$

□

It turns out that the operational semantics may not necessarily always coincide with the denotation defined above for an arbitrary equational logic program. The correctness of the operational semantics follows from the work of Hullot in [Hull80]; the problematic aspect is completeness. Take, for example, the rewrite rule a → h(a). The goal of equality

$$?- g(X, X) = g(Y, h(Y))$$

is solvable with the following substitution σ = {X <— h(a), Y <— a}. The operational semantics based on narrowing, however, cannot generate any solution. Solving equations, in its most general setting, is the problem of *unification in equational theories* [Plot72]. An equational theory is described by a set of equations. In this context there can be more than one most general solution for a given equation. For the purposes of efficient computation, equations can be treated as rewrite rules. The completeness of using rewrite rules in making deductions equationally is expressed by the Church-Rosser property, or equivalently, the confluence property, which says that two terms are E-equivalent if and only if they can be reduced to an identical term. Fay has shown that the narrowing method is complete for canonical term rewriting systems [Fay79]. But canonical term rewriting systems do not usually make use of conditional functions, for the reason that conditional functions often introduce non-terminating reduction sequences. Consider, for example, the familiar definition for factorial

factorial(N) = if N = 0 then 1 else N * factorial (N–1).

In almost all of the existing functional programming languages, the conditional if_then_else is treated as being *non-strict*, thus ensuring termination for appropriate computations. Viewed as a term rewriting system, however, the termination property is lost; for example, the invocation of factorial(–4) will result in an infinite reduction sequence. This appears to be a major difficulty in attempts to formalize an equality-based computational model for non-canonical term rewriting systems.

Our attention is focused on the class of closed linear term rewriting systems; this class allows non-terminating term rewriting systems, and is basically the formalization of Hoffmann and O'Donnell's language. The linearity property says that no variables can occur more than once in the left-hand side of an equation and is required in defining the closure property. Without the linearity property we could not identify the exact *rearrangements* of subterms by a reduction. For example, consider an equation f(X,X) = X. If we reduce f(a,a) to produce the term a, we will then have a problem in identifying which a in f(a,a) the resulting term a is a rearrangement of. However, if a variable disappears in the right-hand side, the linearity condition on the lefthand side can be removed. This is why equations like

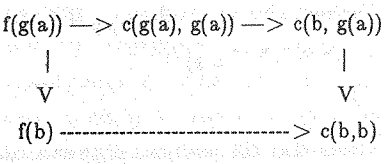
append(A^X, Y, A^Z) = append(X, Y, Z)
A ≡ A = true

do not pose any problems.

The closure property is a special case of confluence property; it ensures a (closed) relationship between reductions at "outer" occurrences and those at "inner" occurrences. As a simple example, consider the equations

f(X) = c(X, X)
g(a) = b.

For the term f(g(a)), we have the following picture embodying the closure property:



We continue in the next section to give a more formal account of the class of closed linear term rewriting systems, and show the soundness and completeness of the operational semantics.

4.1 Closed Linear Term Rewriting Systems

To refer to terms, as well as subterms, function symbols and variables in a term, we need the following definition.

Definition 7. Terms in $T(F,V)$ can be viewed as labeled trees in the following way: a term A is a partial function from N^*, the set of finite sequences of positive integers, to $F \cup V$, such that its domain $D(A)$ satisfies:

(i) $\Delta \in D(A)$
(ii) $u \in D(t_i) \Rightarrow i.u \in D(f(t_1,...,t_i,...,t_n))$ $1 \leq i \leq n$.

$D(A)$ is called a set of occurrences of A. The set of occurrences are partially ordered:

$$u \leq v \text{ iff } \exists w \ u.w = v$$
$$u < v \text{ iff } u \leq v \ \& \ u \neq v.$$

The quotient u−v of two occurrences u and v is defined as: $u-v = w$ iff $v.w = u$.

If $u < v$ we then say that u is *outer* to v and v is *inner* to u.

We say that two occurrences are independent, denoted by $v <> u$, iff $v \not\geq u \ \& \ u \not\geq v$.

We denote by $V(A)$ the set of variables appearing in A. □

The rearrangement of subterms by a reduction is characterized by the notion of the *residue map*, which is defined below.

Definition 8. Let R be an equational logic program. The residue map r with respect to a linear term rewriting system is a function, as defined below:

$$r[A \rightarrow_{[u,k]} B]v$$
$$= \{u.w.(v-v') \mid \alpha_k(v') \in V(R) \ \& \ \alpha_k(v')=\beta_k(w) \ \& \ v' \leq v\} \quad \text{if } v>u \text{ and } v \in D(A)$$
$$= \{v\} \quad \text{if } u <> v \text{ and } v \in D(A)$$
$$= \phi \quad \text{otherwise,}$$

where $A \rightarrow_{[u,k]} B$ denotes that the term A reduces to B at occurrence u using the kth defining equation.

We say that an occurrence w is a residue of v with respect a reduction $A \rightarrow_{[u,k]} B$ if and only if

$$w \in r[A \rightarrow_{[u,k]} B]v.$$

□

Note that the positions of the variables in a rewrite rule indicate the result of this rearrangement process.

The definition of the residue map r can be extended to show how a set of occurrences is rearranged by a sequence of reductions.

Definition 9. Let

$$S = A_0 \rightarrow_{u_0} \cdots \rightarrow_{u_{n-1}} A_n.$$

The extended residue map $\overset{*}{r}$ is a function, as defined below:

for any $N \subseteq D(A_0)$

$$\overset{*}{r}[A_0, \{u_0,...,u_{n-1}\}]N$$
$$= \overset{*}{r}[A_{n-1}, \{u_{n-1}\}]\overset{*}{r}[A_0, \{u_0,...,u_{n-2}\}]N$$

for any $M \subseteq D(A_0)$

$$\overset{*}{r}[A_i, \{u_i\}]M$$
$$= \bigcup_{v \in M} r[A_i \rightarrow_{u_i} A_{i+1}]v$$

□

Intuitively, the above definition says that the residues of a set of occurrences are the union of the individual residues, and that a residue by a sequence of reductions is the cascaded residue yielded by the individual residues in the sequence.

We are now in a position to define the closure property.

Definition 10. An equational logic program R is *closed* if and only if

(i) $\forall u,v \in D(A)$ $((u < v \ \& \ A \rightarrow_u B \ \& \ A \rightarrow_v C) \supset$
$\exists D \ (B \rightarrow_{r[A \rightarrow B]v} D \ \& \ C \rightarrow_u D))$

(ii) $\forall u \in D(A)$ $(A \rightarrow_u B \ \& \ A \rightarrow_u C) \supset \ B = C$

□

Clause (i) of the closure property is illustrated in Figure 1. The second clause says that if two different lefthand sides match the same term, then the corresponding righthand sides must be the same. The second clause is necessary because from the first clause alone one cannot derive the confluence property.

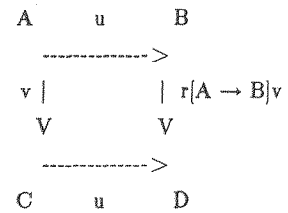


Figure 1: The closure property (when $u < v$).

The closure property can be generalized to reduction sequences. For this and other detailed exploration, the reader should consult [O'Do77, You85a].

It is argued by O'Donnell in the context of subtree replacement systems [O'Do77] (an equational logic program defines a subtree replacement system), that closure is a natural and common property, and the nonclosed subtree replacement systems often represent undesirable axiom systems.

Theorem 1. Let E be a closed linear equational logic program, and G be a goal of equalities of the form

?- $H(A_0, A_1, ..., A_k) = H(true, true, ..., true)$.

If G has a solution σ, then there exists an N-derivation which generates a computed answer ρ which is more general than ρ. i.e., there exists a substitution η such that the composition $\eta\rho$ is E-equivalent to σ.

Conversely, any computed answer σ by an N-derivation from G is a solution to the equation in the goal, i.e,

$\sigma H(A_0, A_1, ..., A_k) =_E H(true, true, ..., true)$.

□

Proof: (Sketch)

Let $P = H(A_0, A_1, ..., A_k)$
$Q = H(B_0, B_1, ..., B_k)$.

Note that Q is neither reducible nor narrowable.

Suppose the equation P = Q has a solution σ. It can be shown (see, for example, [O'Do77]) that E is confluent if it is closed. From the confluence property, there exists a reduction sequence of the form

(a) $\sigma h(P,Q) = A_0 \rightarrow_{u_0} \cdots \rightarrow_{u_{n-1}} A_n = h(Q,Q)$,

where h is a new function symbol, serving as a constructor.

Let $U(\sigma)$ denote the set of occurrences in $D(\sigma h(P,Q))$ that are *introduced* by the substitution σ on h(P,Q). It can be shown that there exist another substitution σ' which is E-equivalent to σ and a reduction sequence

(b) $\sigma' h(P,Q) = B_0 \rightarrow_{v_0} \cdots \rightarrow_{v_{m-1}} B_m = h(Q,Q)$,

such that none of occurrences v_i is a residue of any subterms in σ'. That is,

$\forall v_i,\ 0 \le i \le m-1,\ v_i \notin r^*[B_0, \{v_0,...,v_{i-1}\}]U(\sigma')$.

It can be further shown that corresponding to each reduction sequence of the above form, there exists an N-derivation sequence

$<G_0, \gamma_0>, <G_1, \gamma_1> ..., <G_{m-1}, \gamma_{m-1}>$,

such that the composition

$\nu = \eta \circ \gamma_{m-1} \circ \gamma_{m-2} \circ \cdots \circ \gamma_0$

is a solution of the equation, i.e., $\nu P =_E \nu Q = Q$, and is more general than σ.

Conversely, for any N-derivation sequence issuing from h(P,Q), let ν be the computed answer generated by the sequence. It can be shown that there exists a reduction sequence

$\nu h(P,Q) \xrightarrow{*} h(Q,Q)$.

This implies that ν is indeed a solution to the equation P = Q. This completes the proof. □

A complete proof of Theorem 1 involves several lemmas, proofs of which can be found in [You85a]. The role that the closure property plays here is that a reduction sequence of the form (a) can be "transformed" to a reduction sequence of the form (b) in a finite number of steps. The proof of the existence of an N-derivation sequence corresponding to a reduction sequence of the form (b) is a variant of the one presented in [Hull80]. It should be mentioned that there may not always exist an N-derivation sequence for an arbitrary reduction sequence of the form (a) [You85c].

When a goal of equalities is of the form

?- $H(A_0, A_1, ..., A_m) = H(true, true, ..., true)$,

the rule of deletion upon unification can be ignored. This is described in the following proposition.

Proposition 2. Let $P = H(A_0,...,A_m)$ and $Q = H(true,...,true)$. Let also h be a function symbol not in F. For a goal of equalities of the form ?- P = Q, there exists a narrowing sequence

$h(P,Q) = C_0$

$C_0 \sim>_{[u_0,k_0,\gamma_0]} \cdots \sim>_{[u_{q-1},k_{q-1},\gamma_{q-1}]} C_q$

$C_q = h(Q,Q)$,

such that the composition $\gamma_{q-1} \circ \gamma_{q-2} \circ \cdots \circ \gamma_0$ is a solution of P = Q. □

Proof: Since Q is composed of constant function symbol *true*, the rule of deletion upon unification is not applicable unless A_i, for some i, has narrowed to *true*. Applications of the rule of deletion upon unification on trivial equations of the form true = true can obviously be suspended until no narrowing is possible. Theorem 1 therefore guarantees the existence of a narrowing sequence leading to h(Q,Q). □

Now when a goal is an arbitrary equation, completeness is not always guaranteed even for closed linear term rewriting systems. For example, the rewrite rule a → h(a) mentioned earlier is trivially closed linear. A further property, called the *non-repetition property* needs to be imposed to eliminate such "undesirable" equational logic programs. Intuitively, if a term rewriting system is non-terminating, there are non-terminating reduction sequences, in which some rewrite rule(s) must be used for reductions infinitely many times. For example, suppose that $f(t_1,...,t_n)$ $g(s_1,...,s_m)$ is a rewrite rule that is used infinitely many times in a non-terminating reduction

sequence. We then have an infinite number of subterms (within the reduction sequence) that are reduced by this rule:

$$f(t_{[1,1]},\ldots,t_{[1,n]}),\ f(t_{[2,1]},\ldots,t_{[2,n]}),\ \ldots,\ f(t_{[k,1]},\ldots,t_{[k,n]}),\ \ldots.$$

This yields an infinite set of subterms Q_f, elements of which are reduced by the *same rewrite rule* defining the function f. Some of these terms may be equivalent, i.e.,

$$f(t_{[i,1]},\ldots,t_{[i,n]}) =_E f(t_{[j,1]},\ldots,t_{[j,n]})$$

for some i and j. These terms are thus partitioned into equivalence classes. The non-repetition property requires that all such equivalence classes be finite. Accordingly, the allowable non-terminating sequences are those that have an infinite number of such equivalence classes, each class being finite.

We argue that "normal" equational programs, where constructors are heavily used, possess the non-repetition property. As an example, rewrite rules of the form

$$f(x) \rightarrow x \hat{}\ f(c(x))$$

are often used to generate infinite data structures in programming languages with a lazy evaluation mechanism, where c is a constructor. If we do not have a rewrite rule of the form, say, $a \rightarrow c(a)$, then f(x) and f(c(x)) will not be equivalent no matter how the variable x is instantiated, and the program hence possesses the non-repetition property. Furthermore, such undesirable equational logic programs may be syntactically detected, or even detected at execution time so that the user would know there is a possibility of not getting an answer even if one exists [You85c].

For the class of closed linear term rewriting systems with the non-repetition property, the soundness and completeness of the operational semantics is ensured by the following theorem.

Theorem 3. Let E be a closed linear equational logic program with the non-repetition property, and let G be a goal of equalities of the form

$$?\text{-}\ H(A_0, A_1, \ldots, A_m) = H(B_0, B_1, \ldots, B_m).$$

If G has a solution σ, then there exists an N-derivation generating a computed answer ρ which is more general than ρ. i.e., there exists a substitution η such that the composition $\eta \circ \rho$ is E-equivalent to

Conversely, any computed answer σ by an N-derivation from G is a solution to the equation in the goal, i.e,

$$\sigma H(A_0, A_1, \ldots, A_m) =_E \sigma H(B_0, B_1, \ldots, B_m).$$

□

Proof: (Sketch) Notice that here the term $H(B_0,B_1,\ldots,B_m)$ is no longer composed of constant function symbols. In such a case, for an arbitrary reduction sequence and some term R

(i) $\sigma h(P,Q) = A_0 \rightarrow_{u_0} \ldots \rightarrow_{u_{n-1}} A_n = h(R,R)$,

there exists a reduction sequence

(ii) $\sigma' h(P,Q) = B_0 \rightarrow_{v_0} \ldots \rightarrow_{v_{m-1}} B_m = h(R',R')$,

such that σ' is E-equivalent to and none of occurrences v_i is a residue of any occurrence in $U(\sigma')$. Given this, the remainder of the proof of Theorem 1 follows through unmitigated. □

5. Lazy Narrowing on Conditional Terms

A term involving a conditional function symbol, such as if_then_else and if_then, is said to be a "conditional term". In almost all functional languages, conditional functions are treated as being non-strict; only the conditional part is evaluated at first, and the expression then gets reduced according to the result of that evaluation.

To incorporate a similar strategy into equational logic programming, we need to adjust the definition for E-equality. Recall that the free algebra $T(F,V)$ is divided into congruence classes, closed under instantiation, each of which contains equivalent terms. Let T be a congruence class, which is the union of two disjoint subsets

$$T = T_c \cup T_n$$

where T_c is the set of all conditional terms and T_n the set of all non-conditional terms.

We now interpret $=_E$ to be the congruence of T_n's, that is, all the conditional terms are taken out. We believe that this interpretation also reflects the intuition of using conditional equations in programming: conditional terms are not considered as "final results" but intermediate results that need to be evaluated further to yield nonconditional terms. This interpretation yields the desired result by forcing the operational semantics to perform narrowing on the conditional part, and never perform the rule of deletion upon unification for conditional terms, since equalities are no longer considered for conditional terms.

Although this semantic adjustment seems to violate the pure meaning of *symbolic computation* and has an "operational" flavor, we believe that the equality-based semantics is not greatly impacted, and the simplicity of the semantics is preserved.

6. Implementation Considerations

We have described the basic formalization of an equational logic programming language wherein closed linear term rewriting systems serve as programs. Several programming constructs and implementation strategies might be useful under this formalization.

* *A User-Controlled Reduction Construct* — The use of this construct in a program will force the reduction to be performed until the subterm under consideration is not reducible. The need for this construct is based on the fact that although we have used narrowing as part of the operational semantics, the reduction mechanism is still a useful one. First, the matching procedure (used in reduction) can be implemented more efficiently than unification (used in narrowing). This is particularly true for left-linear equational programs. Secondly, because of the confluence property, reduction steps are not subject to backtracking while narrowing steps are. Moreover, the reduction construct itself serves as an evaluation mechanism for an equational language.
* *Ignoring the Rule of Deletion upon Unification* — Recall that the operational semantics consists of two rules, one of which is called deletion upon unification. When a goal is of the form A = true, only narrowing derivations need be carried out on A, as described in Proposition 2. The rule of deletion upon unification needs to be applied *only* when no further narrowing is possible.
* *Embedding the Rule of Deletion upon Unification into Narrowing* — At each point during the execution of a goal, both the rules of narrowing and deletion upon successful unification should be tried. Also, narrowing steps are subject to backtracking (if so implemented). Redundant narrowing sequences should be eliminated in an implementation. For example, an equation like g(a) = h(f(X)), where g and h are constructors, is not solvable no matter how many narrowing steps are performed on f(X). Furthermore, the work done by deletion upon successful unification should not entirely ignored when backtracking occurs. A method that accommodates these observations is described in [You85c].

7. Conclusion

We have proposed an equational logic computational paradigm and argued its usefulness. An equational logic language was described and some practical considerations were discussed.

The equational logic language described here can be embedded into Prolog to obtain a language which provides logic programming with equality, and in which usual Horn clause logic programming and functional programming are both supported. The theoretical basis for this approach stems from the results in [Plot72] where it was shown how equational theories could be built into resolution procedures. Theorem 3 presented in Section 4 is actually a completeness theorem for the class of closed linear term rewriting systems with the non-repetition property.

Acknowledgments

We would like to thank Gary Lindstrom for his helpful comments on an earlier version of this paper. Jia-Huai You would like to thank Uday Reddy for several inspiring discussions with him on the issues relating to this work.

The work by Jia-Huai You was supported in part by ONR Contract N00014-83-K-0317, and was done when he was at the Computer Science Department, University of Utah.

References

Band84.
Bandes, R.G., "Constraining-unification and the programming language Unicorn," in *Proc. 11th POPL*, pp. 106-110, Salt Lake City, Utah, January, 1984.

Born81.
Borning, A., "The Programming language aspects of ThingLab, a constraint-oriented simulation laboratory," *ACM TOPLAS*, vol. 3, no. 4, pp. 353-387, October, 1981.

Darl85.
Darlington, J., A.J. Field, and H. Pull, "The unification of functional and logic programming," in *Logic programming: relations, functions, and equations*, ed. G. Lindstrom, Prentice-Hall, 1985.

Ders84.
Dershowitz, N., "Equations as programming language," in *Proc. 4th Jerusalem Conference on Information Technology*, pp. 114-123, May, 1984.

Ders85.
Dershowitz, H. and D. Plaisted, "Logic programming cum applicative programming," in *Proc. 1985 International Symposium on Logic Programming*, Boston, Mass., July, 1985.

Fay79.
Fay, M.J., "First-order unification in an equational theory," in *Proc. 4th Workshop on Automated Deduction*, pp. 161-167, Austin, Texas, July, 1985.

Hoff82.
Hoffmann, C.M. and M. O'Donnell, "Programming with equations," *ACM Transaction on Programming Languages and Systems*, vol. 4, no. 1, pp. 83-112, January, 1982.

Hoff84.
Hoffmann, C.M. and M. O'Donnell, "Implementation of an interpreter for abstract equations," in *Proc. 11th POPL*, pp. 111-121, Salt Lake City, Utah, January, 1984.

Hull80.
Hullot, J.M., "Canonical forms and unification," in *Proc. 5th Conference on Automated Deduction*, pp. 318-334, 1980.

Korn83.
Kornfeld, W.A., "Equality in Prolog," in *Proc. 8th International Joint Conference on Artificial Intelligence*, Karlsruhe, West Germany, 1983.

Kowa74.
Kowalski, R.A., "Predicate logic as programming language," in *Proc. IFIP 74*, pp. 556-574, Amsterdam: North Holland, 1974.

Lind85.
Lindstrom, G., "Functional programming and the logical variables," in *Proc. 12th POPL*, New Orleans, January, 1985.

O'Do77.
O'Donnell, M., "Computing in systems described by equations," in *Lecture notes in computer science*, vol. 58, Springer-Verlag, New York, 1977

Pere78.
Pereira, L.M., F.C.N. Pereira, and D.H.D. Warren, *User's guid to DECsystem-10 Prolog*, University of Edinburgh, 1978.

Plot72.
Plotkin, G., "Building-in equational theories," in *Machine Intelligence 7*, pp. 73-90, Edinburgh University Press, 1972.

Redd85.
Reddy, U., "Narrowing as the operational semantics of functional languages," in *Proc. 1985 International Symposium on Logic Programming*, Boston, Mass., July, 1985.

Sato85.
Sato, M. and T. Sakurai, "QUTE: a functional language based on unification," in *Logic programming: relations, functions, and equations*, ed. G. Lindstrom, Prentice-Hall, 1985.

Smol85.
Smolka, G. and P. Panangaden, "A higher-order applicative language based on unification and generic quantification," in *Logic programming: relations, functions, and equations*, ed. G. Lindstrom, Prentice-Hall, 1985.

Stee79.
Steele, G.L. and G.J. Sussman, "Constraints," in *Proc. APL 79 (Part 1)*, pp. 208-225, June, 1979.

Subr84.
Subrahmanyam, P.A. and J.-H. You, "Pattern driven lazy reduction: a unifying evaluation mechanism for functional and logic programming," in *Proc. 11th POPL*, pp. 228-234, Salt Lake City, Utah, January, 1984.

You85a.
You, J.-H. and P.A. Subrahmanyam, *A class of term rewriting systems and unification*, May, 1985. (unpublished manuscript).

You85b.
You, J.-H. and P.A. Subrahmanyam, *On the completeness of first-order unification in equational theories*, June, 1985. (unpublished manuscript).

You85c.
You, J.-H., First-order unification in equational theories and its application to logic programming, Ph.D. Thesis, Computer Science Department, University of Utah, 1985.