AMLOG: an Amalgamated Equational Logic Programming Language

GLENN MANSFIELD*, ATSUSHI TOGASHI* and SHOICHI NOGUCHI*

AMLOG is an equational logic programming language based on the concept introduced by Fribourg [6]. In this language, an amalgamation of logic programming and equational programming is achieved by combining, in the computation procedure, the capability of inferring solutions by means of goal reduction as in logic programming and the term rewriting feature of equational programming. The logical basis of equational logic programming languages is established using deductive reasoning about programs. Two deductive systems are proposed, the notions of confluence and Church-Rosser property of programs are introduced based on these deductive systems, and their equivalence is proved. It is shown that solutions obtained by executing an equational logic program are deducible from the program in the deductive system for equational definite clauses. This result gives the soundness of the computation mechanism. The converse of this result is not true in general. However, for confluent programs, we have shown the following completeness result: if a goal has a solution deducible from a confluent program, under certain conditions, then there is a successfully terminating computation for the goal with a more general solution. Some implementation issues and features of the language are discussed.

1. Introduction

The phenomenal reduction in the cost of computing has resulted in the emergence of several descriptive languages in the horizon of practical computing. These descriptive languages are gifted with elegant semantics based on mathematical logic and independent of the actual method of computation. Two of the more promising languages, not unrelated as we will see below, are logic programming languages and equational (or functional) programming languages. Logic programming languages, *e.g.* Prolog [5, 16], are based on first order predicate logic, and involve the direct application of the resolution principle [3] introduced by Robinson. In these languages programs are expressed as sentences, the successive derivation of new goals from goals, using resolution, is the computation process. Problems (Goals) are solved by showing the inconsistency of the given goals with respect to the given program. The instances of contradiction are results of the computation. The power of these, basically relational, languages lies in the capability of inferring solutions. The two major drawbacks of logic programming are its awkward handling of the equality relation and the meaningless way functions are handled.

Equational languages, for instance in [10, 21], use equations to specify programs. From the computational point of view, equations may be viewed as reduction rules allowing the reduction of a left-hand expression to its corresponding right-hand expression. In the process of computation successive reductions are carried out on a given expression by applying equations until no further reduction is possible. The expression is then said to have been reduced to its normal form. This expression is the result of the computation. The rules are equivalence relations; thus, equality is builtin. But the input-output relationship is strictly defined. This precludes the possibility of inference.

The complementary properties of the two programming approaches have inspired researchers to look for a formalism that combines the two approaches. The FUNLOG of Subrahmanyam *et al.* [19] uses function reduction and, what is called, *semantic unification*. The computation mechanism does not fully encompass *narrowing* [13] and is thus limited in its power of reasoning about equations. The EQLOG of Goguen *et al.* [8] is based on the formal foundation of many sorted Horn clause logic with equality and offers several attractive features. The methodology is, however, different from *amalgamation* in the sense that it retains predicates and functions unchanged. This is reflected in the computation mechanism which consists of separate algorithms for solving equations and handling predicates.

Operationally, *unification* (two sided *matching*) between two terms is the basis of resolution which forms the core of logic programming. On the other hand, in functional programming the evaluation mechanism for terms and subterms uses one sided *matching*. A unified approach would involve unification at subterms. The ap-

*Research Institute of Electrical Communication, Tohoku University, 2-1-1 Katahira-cho, Sendai-shi 980 Japan.

proach of SLOG [7] uses superposition which embodies the above concept. The language uses the concept of Horn oriented equational clauses. The only predicate in the language is the equality predicate. All other predicates are encoded as functions. The head of the clause is interpreted as a rewrite rule; instead of resolution, *innermost goal-superposition* is used. The completeness result is given for a restricted class of programs (*ground canonical, well-innermost-reducing*) and for goals which have answers made of constructors only.

In this paper we propose the equational logic programming language AMLOG. It is based on the concept of equational logic programming proposed by Fribourg [6]. A significant change, affecting the completeness result, is introduced in the computation mechanism. In [6] a program is seen as a dynamic object which is *extended*, possibly infinitely, by the addition of new clauses generated by *definite superposition* (superposition of clauses on clauses). Nevertheless, the computation mechanism consists of a linear derivation from a finite *extension* of the program with the initial goal statement. In contrast with this, a computation in AMLOG envisages a linear derivation from the program, which is static, with the initial goal. The completeness result given, covers a restricted (confluent) class of programs and goals which have answers made of *normal* substitutions (see section 5) only. We have adopted the deductive approach to establish the soundness and correctness of the computation mechanism. The properties of these deductive systems and their relation with the programs in the language are discussed.

The organization of this paper is as follows: in section 2 some basic definitions are stated. The formulation of a deductive system for equations is presented in section 3; soundness and completeness of the system is dealt with. In section 4, an amalgamated logic programming language is introduced, this is followed by its precise discussion in section 5. In section 6 some implementation issues and pertinent features of the language are discussed followed by concluding remarks in section 7.

2. Basic Definitions

In this section, we briefly review the definitions of terms, substitutions and other related jargon. See [9, 11, 12], for further detailed discussions. For simplicity of notation, we assume that we have only one sort. Nevertheless, all the results of this paper apply to the many sorted case as well.

Throughout this paper, it is assumed that we are given a set Σ of *function symbols* containing the distinguished symbol *true*. Each function symbol has a *rank*, a natural number. A function symbol of rank 0 is called a *constant*. The symbol *true* is a constant.

Let V be a denumerable set of *variables*. A *term* (or a Σ-*term* if we emphasize Σ) is recursively defined as follows:

—a variable $x \in V$ is a term;
—a constant $a \in \Sigma$ is a term;
—if $f \in \Sigma$ is a function symbol of rank n and each M_i is a term,
then $f(M_1, \ldots, M_n)$ is a term.

The set of all terms is denoted by $T(\Sigma, V)$, and the set of all ground terms, terms containing no variables, by $T(\Sigma)$.

For a term M, we denote its set of occurrences by $Ocr(M)$ and the *subterm* of M at the occurrence $u \in Ocr(M)$ by M/u. We say u is the occurrence of the subterm M/u in M. Given two terms M, N and an occurrence $u \in Ocr(M)$, we define $M[u \leftarrow N]$ as the term M in which the subterm M/u at the occurrence u is replaced by N, and $M[N]$ as the term M in which some subterm of M is replaced by N. $Var(M)$ denotes the set of variables in a term M.

A *substitution* is a mapping θ from a set of variables into a set of terms such that $\theta(x) \equiv x$ almost everywhere, i.e., the set of variables replaced by θ is finite. Substitutions can be extended to terms in the usual way. The *Domain* of a substitution σ is given by $D(\sigma) = \{x \mid \sigma x \neq x\}$. The *set of variables introduced* by σ is given by $I(\sigma) = \{x \mid x \in Var(\sigma y), y \in D(\sigma)\}$. The *restriction* of a substitution σ to a set of variables U is given by $(\sigma \uparrow U)x = \sigma x$ if $x \in U$ and $(\sigma \uparrow U)x = x$ if $x \notin U$.

In this paper, we use several symbols as syntactical meta variables. We use x, y, z for variables; f, g, h for function symbols; a, b, c for constants; K, L, M, N, R for terms; u, v for occurrences; Γ, Δ for sequence of equations; and, $\theta, \sigma, \eta, \zeta, \psi$ for substitutions; possibly, with primes or subscripts. The symbol $\equiv$ is used to indicate syntactical identity.

3. An Equational Definite Logic

3.1 Equational Definite Clauses

Definition An *equational definite clause* (or a *conditional equation*) is an equational clause of the form

$$L = R \leftarrow M_1 = N_1, \ldots, M_n = N_n. \qquad []$$

Note that an equation $L = R$ is an equational definite clause without conditions. In equational definite logic, a predicate can be viewed as a truth function, an atomic formula A (in first order logic) is expressed by the equation $(A = true)$.

Definition A *proof*(*figure*) of an equation $M = N$ in a deductive system DS is a tree $G_0 \ldots G_i G_{i+1} \ldots G_n$ (of equations) where G_i, $0 \leq i \leq n$, represents the set of nodes (equations) at depth i in the tree such that

(*i*) for all $0 \leq i < n$, all the nodes in G_{i+1} can be deduced from those in G_i using the inference rules of the deductive system DS, and

(*ii*) G_n is $\{M = N\}$.

We say, $M = N$ is *provable* from a set S of equational definite clauses in a deductive system DS, denoted by

$S \vdash_{DS} (M=N)$, iff there is a *proof* (*figure*) of $M=N$ from S in the deductive system. A sequence of equations Γ is said to be *provable* in DS i.e. $S \vdash_{DS} \Gamma$ iff every equation of Γ is provable in DS. The subscript DS is dropped from the notation when the deductive system chosen for the proof is clear from the context. []

We have adopted the deductive approach in proving the soundness and completeness of our computation system as, unlike the approach involving interpretations, in the deductive approach the proofs are simple, elegant, compact and based on mechanical reasoning. In the following we present a deductive system EQ for equations.

(*EQ reflection*)

$$\frac{}{M=M}$$

(*EQ commutation*)

$$\frac{M=N}{N=M}$$

(*EQ transition*)

$$\frac{M=L \quad L=N}{M=N}$$

(*EQ replacement*)

$$\frac{M=N}{L[M]=L[N]}$$

(*EQ modus ponens*)

$$\frac{\theta M_1=\theta N_1 \ldots \theta M_n=\theta N_n \quad (M=N \leftarrow M_1=N_1, \ldots, M_n=N_n \in S)}{\theta M=\theta N}$$

3.2 A Deductive System for Equations (EQ)

Given a set of equational definite clauses, the following are the deductive rules of inference for equations:

Proposition 1. If $S \vdash_{EQ} (M=N)$ then $S \vdash_{EQ} (\theta M=\theta N)$ for all substitutions θ.

Proof. Straightforward: by structural induction on the proof of $S \vdash_{EQ} (M=N)$.

3.3 Interpretations of Equational Definite Clauses

Definition An *interpretation* of Σ in the equational definite clause logic is a triple $I=\langle A, \Sigma_A, =_A \rangle$, where

—A is a non empty set, called the *carrier* of the interpretation;

—Σ_A is a set of functions f_A assigned to $f \in \Sigma$;

—$=_A$ is an equivalence relation on A such that if $\forall_i a_i =_A b_i (1 \leqq i \leqq n)$ then

$$f_A(a_1, \ldots, a_n) =_A f_A(b_1, \ldots, b_n)$$

for every function symbol f of rank n. []

In other words, an interpretation of Σ is a *Σ-algebra* [9] together with an equivalence relation compatible with every function. An equivalence relation compatible with every function is called a *congruence relation*. By abuse of notations, we identify an interpretation with a Σ-algebra on which a congruence relation is defined. A *valuation* of variables in an interpretation A is an assignment of an element in A to each variable. The definition of *valuation* can be extended to terms in the usual way.

Definition Let A be an interpretation and S be a set of equational definite clauses.

1. An equation $M=N$ is *satisfied* by a valuation v in A iff $v(M) =_A v(N)$.
2. An equational definite clause $M=N \leftarrow \Gamma$ is *valid* in A iff for every valuation v in A, $M=N$ is satisfied by v, or some equation in Γ is not satisfied by v.
3. A is a *model* of S iff each clause in S is valid in A.
4. $M=N$ is a *logical consequence* of S, denoted by $S \models M=N$, iff it is valid in every model of S. []

3.4 Soundness and Completeness of the Deductive System EQ

Lemma 1. [*Soundness of the Deductive System EQ*] *Let S be a set of equational definite clauses and $L=R$ an equation. If $S \vdash_{EQ} L=R$ then $S \models L=R$.*

Proof. Straightforward. It suffices to show that each inference rule preserves the validity of clauses. Since this can be easily checked, the details are omitted. []

To prove the completeness of the system, we will construct a special interpretation. Let S be an arbitrary set of equational definite clauses. Recall that $T(\Sigma, V)$ denotes the set of all terms constructed from Σ and V. Let us define a relation $\approx$ on $T(\Sigma, V)$ by $M \approx N$ iff $S \vdash_{EQ} M=N$. By the rules of inference, the relation $\approx$ is clearly a congruence relation on $T(\Sigma, V)$, the resulting quotient algebra $Q=T(\Sigma, V)/\approx$ is an interpretation with the identity relation on the set of all congruence classes as a congruence relation on Q. A function f_Q, for each function symbol $f \in \Sigma$ of rank n, is defined by

$$f_Q([M_1], \ldots, [M_n]) = [f(M_1, \ldots, M_n)],$$

where $[M]$ denotes the congruence class of M.

In the following, we adopt the fact, without proof, that an equational definite clause $M=N \leftarrow M_1=N_1, \ldots, M_n=N_n$ is *valid in Q* iff $\theta M \approx \theta N$ follows from $\theta M_1 \approx \theta N_1, \ldots, \theta M_n \approx \theta N_n$, for every substitution θ.

Lemma 2. *Q is a model of S.*

Proof. Let $M=N \leftarrow M_1=N_1, \ldots, M_n=N_n$ be a clause in S and θ a substitution. Suppose $\theta M_i \approx \theta N_i$ for

$i=1, \ldots, n$ then from the definition of Q $S \vdash \theta M_i = \theta N_i$, for $i=1, \ldots, n$ whence applying *modus ponence* we can deduce $S \vdash \theta M = \theta N$ so that $\theta M \approx \theta N$. Thus Q is model of S. []

Lemma 3. *$L=R$ is provable from S in the deductive system EQ, $S \vdash_{EQ} L=R$, iff it is valid in the model Q of S.*

Proof. "Only if" part is by Lemma 1 and Lemma 2. The "if" part follows from the definition of validity in Q. []

Theorem 1. [*Soundness and Completeness of the Deductive System EQ*]

$S \vdash L=R$ iff $S \models L=R$. Where, S is a set of equational definite clauses and $L=R$ is an equation.

Proof. By Lemma 1, 2, and 3. []

4. Equational Logic Programs

In an equational logic programming language, the head equation of an equational definite clause $L=R \leftarrow \Gamma$ is implicitly oriented from left to right. Thus the head of the clause can be viewed as a rewriting rule $L \rightarrow R$. In the following, we shall describe an equational logic program and its operational semantics.

Definition An *equational logic program* is a finite set **P** of *oriented equational definite clauses* of the form

$$L \rightarrow R \leftarrow L_1=R_1, \ldots, L_n=R_n. \quad []$$

Definition A *goal* for an equational logic program is a finite sequence of equations: $M_1=N_1, \ldots, M_k=N_k$. []

Computation mechanism. Goals represent the problems which will be solved by execution of programs. A *computation* (or an *execution*) is initiated by an input goal. The computation proceeds by applying suitable computation rules to derive successive new goals. At each step some equation is selected from the goal and is checked for the applicability of *reflection* or *superposition*. If reflection is possible, the equation is deleted from the goal and the substitution which unifies the two sides of the equation is applied to the resulting goal. In the case of superposition, some subterm in the equation, unifiable with the left hand side of the head of some clause in the program, is replaced by the corresponding right hand side in the head of the clause. The body of the clause is added to the goal. Finally, the unifier is applied to the resulting goal. The computation successfully terminates for an input goal if the empty goal (an empty sequence of equations) is derived. Precise definitions related to the computation are given below.

Definition Let $G=(M_1=N_1, \ldots, M_k=N_k)$ be a goal. The *reflection rule* allows the removal of an equation $M_i=N_i$, $1 \leq i \leq k$, from G where $\theta M_i \equiv \theta N_i$ by a most general unifier θ of the terms M_i and N_i. The goal

$$G' = \theta(M_1=N_1, \ldots, M_{i-1}=N_{i-1}, M_{i+1}=N_{i+1}, \ldots, M_k=N_k)$$

is called a *reflectant* of G on $M_i=N_i$. []

Definition Let $G=(M_1=N_1, \ldots, M_k=N_k)$ be a goal. The *superposition rule* yields a *superposant* G' of G by an oriented clause $L \rightarrow R \leftarrow L_1=R_1, \ldots, L_n=R_n$, on $M_i=N_i$ if either:

1. M_i has a non-variable occurrence u_i such that $\theta(M_i/u_i) \equiv \theta L$ by a most general unifier θ, and G' is the goal

$$\theta(M_1=N_1, \ldots, M_i[u_i \leftarrow R]=N_i, L_1=R_1, \ldots, L_n=R_n, M_{i+1}=N_{i+1}, \ldots, M_k=N_k), \text{ or}$$

2. N_i has a non-variable occurrence v_i such that $\theta(N_i/v_i) \equiv \theta L$ by a most general unifier θ, and G' is the goal

$$\theta(M_1=N_1, \ldots, M_i=N_i[v_i \leftarrow R], L_1=R_1, \ldots, L_n=R_n, M_{i+1}=N_{i+1}, \ldots, M_k=N_k). \quad []$$

Note that we are considering only *goal superposition* i.e. superposition of clauses on goals. *Definite superposition* cf. [6] i.e. superposition of clauses on clauses is not included in the computation. *Narrowing*, see [8, 13], is a special case of superposition where the program consists only of equations i.e. definite clauses without conditions. The computational use of narrowing and its links with Prolog have been independently investigated in [8].

For goals G and G', $G \Rightarrow G'$ indicates that G' is either a reflectant or a superposant of G. We may write $G \Rightarrow^{\theta} G'$ to specify the used unifier θ.

Definition Let P be a program. A *computation* from a goal G is a, possibly infinite, sequence of goal reductions

$$G_0(=G) \Rightarrow^{\theta_1} G_1 \Rightarrow^{\theta_2} G_2 \Rightarrow \ldots G_i \Rightarrow^{\theta_{i+1}} G_{i+1} \Rightarrow \ldots$$

A computation *successfully terminates* if G_n is an empty goal, denoted by e, for some $n \geq 0$, where an *empty goal* is the empty sequence of equations. In this case, the composition $\theta=\theta_n \ldots \theta_1$ is the *used substitution*, θ', the restriction of θ to the variables in G denoted by $\theta'=\theta \uparrow Var(G)$ is called the *answer substitution* and $\theta' G (\equiv \theta G)$ is the *result* of the computation. The *active variables* $V(G_i)$ in an intermediate goal G_i is given by

$$V(G_0)=Var(G_0)$$

$$V(G_{i+1})=(V(G_i) \cup I(\theta_{i+1} \uparrow V(G_i))) - D(\theta_{i+1} \uparrow V(G_i)).$$

Where, $G_i \Rightarrow^{\theta_{i+1}} G_{i+1}$ and $D(\theta_{i+1}) \cap I(\theta_{i+1}) = \Phi$. []

5. Soundness and Completeness

In the following we have viewed a program as a set of equational definite clauses by ignoring the orientation. This ensures that the notions such as models, validness, logical consequence and so on are also available for equational logic programs.

Lemma 4. *Let* P *be an equational logic program and G a goal. If there is a successful computation from* $G: G \Rightarrow^{\theta_1} G_1 \Rightarrow^{\theta_2} \ldots \Rightarrow^{\theta_n} \varepsilon$, *then every equation in* $\theta_n \ldots \theta_1 G$ *is provable from* P.

Proof. The proof is by induction on the length $n \geqq 0$ of computation. In the base case, $n=0$, G is an empty goal and the Lemma is trivially true.

Next, suppose that the result holds for computations of length $n-1 \geqq 0$. Let us consider a computation of length n, $G \Rightarrow^{\theta_1} G_1 \Rightarrow^{\theta_2} \ldots \Rightarrow^{\theta_{n-1}} \Rightarrow^{\theta_n} \varepsilon$, where G is of the form $(\Gamma, M=N, \Delta)$. There are two possibilities.

If G_1 is a reflectant of $(\Gamma, M=N, \Delta)$ at $M=N$, then $G_1=\theta_1(\Gamma, \Delta)$ where, $\theta_1 M \equiv \theta_1 N$. Thus, $P \vdash \theta_n \ldots \theta_2(\theta_1 M=\theta_1 N)$ by Proposition 1. Also, $P \vdash \theta_n \ldots \theta_2\theta_1(\Gamma, \Delta)$ by the induction hypothesis and Proposition 1. Combining these proofs we have $P \vdash \theta_n \ldots \theta_2\theta_1(\Gamma, M=N, \Delta)$, or, $P \vdash \theta_n \ldots \theta_2\theta_1 G$.

If G_1 is a superposant of $(\Gamma, M=N, \Delta)$ on $M=N$ by a clause $L \to R \leftarrow \Lambda \in P$, either M or N is modified. Without loss of generality, suppose M is rewritten, then $G_1=\theta_1(\Gamma, M[u \leftarrow R]=N, \Lambda, \Delta)$ where $\theta_1(M/u) \equiv \theta_1 L$ for some occurrence u in M. By induction hypothesis,

$$P \vdash \theta_n \ldots \theta_1(M[u \leftarrow R]=N) \tag{5.1}$$

$$P \vdash \theta_n \ldots \theta_1(\Gamma, \Delta) \tag{5.2}$$

$$P \vdash \theta_n \ldots \theta_1(\Lambda) \tag{5.3}$$

The proof figure for $\theta_n \ldots \theta_1 G$ may be built as follows-

$\theta_n \ldots \theta_1 \Lambda$ *(using 5.3)*
——————— *(modus ponens)*
$\theta_n \ldots \theta_1(L=R)$
——————— *(replacement)*
$\theta_n \ldots \theta_1(M=M[u \leftarrow R])$; $\theta_n \ldots \theta_1(M[u \leftarrow R]=N)$ *(using 5.1)*
——————— *(transitivity)*
$\theta_n \ldots \theta_1(M=N)$; $\theta_n \ldots \theta_1(\Gamma, \Delta)$ *(using 5.2)*
———————
$\theta_n \ldots \theta_1(G)$ []

Without loss of generality, we can assume that an input goal in an equational logic program is a single equation $M=N$. Because, solving a goal $M_1=N_1, \ldots, M_k=N_k$ is logically equivalent to solving the single equation $f(M_1, \ldots, M_k)=f(N_1, \ldots, N_k)$, where f is a fresh function symbol.

Theorem 2. [*Soundness of the Computation Mechanism*]

Let P *be an equational logic program and* $G=(M=N)$ *a goal. If there is a successfully terminating computation* $G \Rightarrow^{\theta_1} G_1 \Rightarrow^{\theta_2} G_2 \Rightarrow \ldots \Rightarrow^{\theta_n} \varepsilon$, *then the universal closure of the result* $\theta_n \ldots \theta_1 G$ *is the logical consequence of* P.

Proof. By Lemma 4 and Theorem 1. []

The converse of Theorem 2 is not true in general. Not all logical consequences of a program P have a successfully terminating computation as is shown in the following example.

Example 1. Let us consider the program $P=\{a \to b, a \to c, f(a, a) \to true\}$. Clearly, $f(b, c)=true$ is the logical consequence of P. But there is no successfully terminating computation from the goal $f(b, c)=true$. []

To handle the completeness problem we view an equational logic program as a reduction relation.

Definition Any equational logic program P defines a *reduction relation* R_P: $M_0 \to N$ belongs to R_P iff there is a successful computation from the goal $M_0=N$

$$(M_0=N) \Rightarrow^{\theta_1} (M_1=N, \Delta_1) \Rightarrow^{\theta_2} (M_2=N, \Delta_2) \Rightarrow \ldots \Rightarrow^{\theta_{n-1}} (N=N) \Rightarrow^{\theta_n} e$$

such that $\theta_n \ldots \theta_{i+1} M_i \equiv M_i$ and $\theta_n \ldots \theta_{i+1} N \equiv N$ for each i, $0 \leqq i \leqq n-1$ and N remains unchanged during the computation. By definition, $M \to N \in R_P$ if $M \to L$, $L \to N \in R_P$ for all terms M, N, and L. []

Now, we propose another deductive system RD which corresponds to the reduction relation R_p defined by a program P. This deductive system needs to take care of the inherent asymmetry in the reduction relations defined by an equational logic program.

Definition A deductive system RD for an equational logic program P consists of the following inference rules:

(RD reflection)

$$\frac{}{M=M}$$

(RD transition)

$$\frac{M=L \quad L=N}{M=N}$$

(RD replacement)

$$\frac{M=N}{L[M]=L[N]}$$

(RD modus ponens)

$$\frac{\theta M_1=K_1 \theta N_1=K_1 \ldots \theta M_n=K_n \theta N_n=K_n}{\theta M=\theta N}$$

where, $M=N \leftarrow M_1=N_1, \ldots, M_n=N_n \in P$

Note: $P \vdash_{RD} M=N$ implies $P \vdash_{RD} \theta M=\theta N$ for all θ as in EQ.

Proposition 2. $P \vdash_{RD} M=N$ implies $P \vdash_{EQ} M=N$.

Proof. Obvious. []

Definition An equation $L=R$ is *convergent* by P in RD, denoted by $\downarrow_P(L=R)$, iff there exists a K such that $L=K$ and $R=K$ are provable from P in RD i.e. $\downarrow_P(L=R)$ iff $P \vdash_{RD} (L=K), (R=K)$ for some K. A sequence of equations is *convergent* iff every equation in the sequence is *convergent*.

Theorem 3. *Let* P *be an equational logic program and M, N be terms.* $M \rightarrow N \in R_P$ *iff* $M=N$ *is provable from* P *in RD.*

Proof. "If" part is by structural induction on the proof figure of $M=N$ from P.

For the "only if" part, suppose $M \rightarrow N \in R_P$. Let $(M_0=N) \Rightarrow^{\theta_1} (M_1=N, \Delta_1) \Rightarrow^{\theta_2} (M_2=N, \Delta_2) \Rightarrow \ldots \Rightarrow^{\theta_{n-1}} (N=N) \Rightarrow^{\theta_n} \varepsilon$ be the corresponding successful computation from the goal $(M_0=N)$, where $M_0 \equiv M$. The result follows from the following assertions:

(A1) $M_{n-i}=N$ *is provable from* P *in RD, for each* $i=1, \ldots, n$.

(A2) $\theta_n \ldots \theta_{n-i+1}\Delta_{n-i}$, *is convergent by* P *in RD,* $\downarrow_P(\theta_n \ldots \theta_{n-i+1}\Delta_{n-i})$, *for each* $i=1, \ldots, n$.

The proof of the above assertions is by induction on i.

Base case: $i=1$. A1 is true as the *proof* of $N=N$ in *RD* can be constructed using the *reflection rule*. A2 is trivially true as $\Delta_{n-1} \equiv \varepsilon$.

Induction step: Suppose A1 and A2 are true for $i \geq 1$, we will show that A1 and A2 are true for $i+1$.

Let us consider the *one-step goal reduction*

$$(M_{n-i-1}=N, \Delta_{n-i-1}) \Rightarrow \theta_{n-i}(M_{n-1}=N, \Delta_{n-i})$$

There are three possibilities

(*a*) Δ_{n-i} is obtained from Δ_{n-i-1} by reflection and $M_{n-i-1} \equiv M_{n-i}$,

(*b*) Δ_{n-i} is obtained from Δ_{n-i-1} by superposition and $M_{n-i-1} \equiv M_{n-i}$, and,

(*c*) $(M_{n-i}=N, \Delta_{n-i})$ is obtained by superposing a subterm of M_{n-i-1} with the left hand side of a rule in P.

In the following each of these cases are examined—

(*a*) If Δ_{n-i} is obtained from Δ_{n-i-1} by reflection, without loss of generality let Δ_{n-i-1} be of the form $(T_1=T_2, \Gamma)$ such that $\theta_{n-i}T_1 \equiv \theta_{n-i}T_2$ and $\Delta_{n-i} \equiv \theta_{n-i}\Gamma$. According to the induction hypothesis A2 is true for Δ_{n-i}. We can easily construct the proof for Δ_{n-i-1} in accordance with A2 by chosing $\theta_{n-i}T_1 \equiv K \equiv \theta_{n-i}T_2$ and using the induction hypothesis for Δ_{n-i}. (A1 is trivially true in this case)

(*b*) If Δ_{n-i} is obtained form Δ_{n-i-1} by superposition. Without loss of generality let $\Delta_{n-i-1} \equiv (T_1=T_2, \Gamma)$; $\theta_{n-i}(T_1/u) \equiv \theta_{n-i}L$ and $\Delta_{n-i} \equiv \theta_{n-i}(T_1[u \leftarrow R]=T_2, \Lambda, \Gamma)$, where, $L=R \leftarrow \Lambda \in$ P is superposed on T_1 at its u^{th} occurrence. According to the induction hypothesis A2 is true for Δ_{n-i}. The proof for Δ_{n-i-1} can easily be constructed in accordance with A2 as follows (A1 is trivially true in this case)—

$\downarrow_P(\theta_n\theta_{n-1} \ldots \theta_{n-i}\Lambda)$ *(induction hypothesis)*

——— *By Modus Ponens*

$\theta_n \ldots \theta_{n-i}L=\theta_n \ldots \theta_{n-i}R$; $\theta_n \ldots \theta_{n-i}(T_1/u)=\theta_n \ldots \theta_{n-i}L$*(superposition cond'n) By Transitivity*

———

$\theta_n \ldots \theta_{n-i}(T_1/u)=\theta_n \ldots \theta_{n-i}R$

——— *By Replacement*

$\theta_n \ldots \theta_{n-i}(T_1)=\theta_n \ldots \theta_{n-i}(T_1[u \leftarrow R])$; $\theta_n \ldots \theta_{n-i}(T_1[u \leftarrow R])=K$*(ind'n hyp) By Transitivity*

———

$\theta_n \ldots \theta_{n-i}(T_1)=K$; $\theta_n \ldots \theta_{n-i}(T_2)=K$, $\theta_n \ldots \theta_{n-i}\Gamma$*(Induction hypothesis)*

———

$\theta_n \ldots \theta_{n-i}\Delta_{n-i-1}$

(*c*) $(M_{n-i}=N, \Delta_{n-i})$ is obtained from $(M_{n-i-1}=N, \Delta_{n-i-1})$ by superposing the clause $L=R \leftarrow \Lambda \in$ P at the occurrence u using the substitution θ_{n-i}. From the definition of the reduction relation, θ_{n-i} does not affect M_{n-i-1} and N i.e. $D(\theta_{n-i}) \cap Var(M_{n-i-1}) = \phi = D(\theta_{n-i}) \cap Var(N)$. Using the superposition condition we have $(M_{n-i-1}/u) \equiv \theta_{n-i}L$, $M_{n-i} \equiv M_{n-i-1}[u \leftarrow \theta_{n-i}R]$, $\Delta_{n-i} \equiv \theta_{n-i}(\Lambda, \Delta_{n-i-1})$. By the induction hypothesis there exist *proofs* in *RD* for $\theta_n \ldots \theta_{n-i+1}\Delta_{n-i}$ (A2) and $M_{n-i}=N$ (A1). Thus, (A2) is true for Δ_{n-i-1}. Also, $\theta_n \ldots \theta_{n-i+1}(M_{n-i}=N) \equiv (M_{n-i}=N)$, whence, $\theta_n \ldots \theta_{n-i}(M_{n-i-1}=N) \equiv (M_{n-i-1}=N)$ as $D(\theta_{n-i}) \cap Var(N) = \phi = D(\theta_{n-i}) \cap Var(M_{n-i-1})$.

The *proof* for $M_{n-i-1}=N$ is constructed as follows—

$\downarrow_P(\theta_n \ldots \theta_{n-i}\Lambda)$*(induction hypothesis)*

——— *By Modus Ponens*

$\theta_n \ldots \theta_{n-i}(L=R)$; $(M_{n-i-1}/u)=\theta_n \ldots \theta_{n-i}L$*(superposition condition) By Transitivity*

———

$(M_{n-i-1}/u)=\theta_n \ldots \theta_{n-i}R$ *By Replacement*

———

$(M_{n-i-1})=M_{n-i-1}[u \leftarrow \theta_n \ldots \theta_{n-i}R]$; $M_{n-i-1}[u \leftarrow \theta_{n-i}R]=N$*(Induction hypothesis) By transitivity*

———

$\theta_n \ldots \theta_{n-i}(M_{n-i-1}=N) \equiv (M_{n-i-1})=N$

note: $M_{n-i-1}[u \leftarrow \theta_n \ldots \theta_{n-i}R] \equiv M_{n-i-1}[u \leftarrow \theta_{n-i}R]$

This completes the proof of the theorem. []

Definition A program P is *confluent* iff for any terms M, N_1 and N_2, $P \vdash_{RD} (M=N_1)$, $(M=N_2)$ implies $P \vdash_{RD} (N_1=L)$, $(N_2=L)$ for some L. []

Definition A program P has a *Church-Rosser property* iff for any terms M and N, $P \vdash_{EQ} (M=N)$ implies $P \vdash_{RD} (M=L)$, $(N=L)$ for some L. []

It is evident that P is confluent, whenever P has a Church-Rosser property.

Proposition 3. *Let* P *be an equational logic program.* P *is confluent if and only if* P *has the Church-Rosser property.*

Proof. It suffices to show "only if" part only. Assume that P is confluent. Let $M=N$ be an equation. Suppose that $M=N$ is provable from P in the deductive system EQ. Let π be a proof figure of $M=N$ from P in EQ. To prove the result, we show by structural induction on π that both equations $M=K$ and $N=K$ are provable from P in RD for some K.

Base case is clear by (*RD reflection*). Now, suppose π is a proof figure of $M=N$ consisting of more than one quation, then $M=N$ is inferred, for instance, from equations $M=L$ and $L=N$ by applying (*EQ transition*). By inductive hypothesis, there are terms K_1 and K_2 such that $M=K_1$, $L=K_1$, $L=K_2$, and $N=K_2$ are provable from P in RD. Since P is confluent, $K_1=K$ and $K_2=K$ are provable from P in RD for some K. Hence, $M=K$ and $N=K$ are provable from P in RD.

Other cases are proved using the same line of argument. []

Corollary 1. *Let* P *be a confluent program. If a single equation* $M=N$ *is the logical consequence of* P, *then there is a successfully terminating computation from* $M=N$ *with the empty answer substitution.*

Proof. By Theorem 1, 3 and Proposition 3.

Definition Given a program P, a term M is said to be *R_P-normal* if there is no term N except M itself such that $M \to N \in R_P$. A substitution θ is *R_P-normalized* if $\theta(x)$ is *R_P-normal*, for every variable x. An *R_P-normal instance* of a term t is ηt where η is an Rp-normal substitution. []

The connection between rewriting relations and narrowing relations has been given for equations in [13]. This result has been extended to the case of conditional equations in [15, 14, 4]. It has been shown: Given a program P, if there exists a rewriting relation from ηG where η is an R_P-normal substitution and G is a goal then there exists a computation from G in P. In the following, we present a more general result eliciting the connection between a successful computation which uses an empty answer substitution (*reference computation*) and other successful computations, emerging respectively, from an R_P-normal instance of a goal and the goal itself. Note that, though the referenced successful computations involve empty answer substitutions only; intermediate variables may be introduced into the goal by the referenced computation and may be freely instantiated as the occasion arises.

Proposition 4. Given an equational program P, a goal G', and a successful computation from $G_0'(=G') \Rightarrow^{\theta_0} G_1' \Rightarrow^{\theta_1} G_2' \Rightarrow \ldots \Rightarrow^{\theta_n} \varepsilon$ where $\theta_n \ldots \theta_1\theta_0$ is an empty (answer) subsitution i.e. $\theta_{n-1} \ldots \theta_1\theta_0 \uparrow V(G_0)$ is an empty substitution; suppose at some step, say i, the computation involves superposition by a clause C: $L=R \leftarrow Q \in$ P at some non-variable occurrence u of a term M' in G_i' ($Var(G_i') \cap Var(C) = \Phi$) using a substitution θ_i ($D(\theta_i) \cap V(G_i') = \Phi$ as $\theta_n \ldots \theta_1\theta_0$ is an empty (answer) substitution) and $M' \equiv \eta M$ where η is an R_P-normal substitution, $D(\eta) = U \subseteq Var(M)$, and $Var(\eta x) \subseteq V(M')$ for all $x \in U$, then

(i) u corresponds to a non-variable subterm in M.

Proof. (i) Let $M'/u \equiv \eta M/u \equiv P'$. By the superposition condition $\theta P' \equiv \theta L$. There are three possibilities-

(a) u corresponds to a non variable occurrence in M,

(b) u corresponds to a variable occurrence in M i.e. $M/u = x \in Var(M)$,

(c) $\exists v \in Ocr(M)$ such that $M/v = x \in Var(M)$ and $M'/u \equiv P'$ is a proper subterm of ηx.

In the following we will examine all the three possibilities:

(a) In this case (i) is true.

(b) $M/u \equiv x$. There are two possibilities—

$x \in U$: by the hypothesis of the proposition $Var(\eta x) \subseteq V(M')$ and since θ is an empty answer substitution $D(\theta) \cap Var(\eta x) = \Phi$. Thus, $\theta\eta x \equiv \eta x$ and the superposition condition reduces to $\eta x = \theta L$. Also, the conditions of successful computation yield $\theta Q^* \Rightarrow^{\psi} \varepsilon$ for some empty answer substitution ψ. This indicates that $\eta x \to \theta R \in R_P$ which contradicts the assumption that η is Rp-normal.

$x \notin U$: by the hypothesis of the proposition $\eta x \equiv x$. Once again from the superposition condition $\theta x \equiv \theta L$. This shows that u is a variable occurrence in M' itself which contradicts the assumption that u is a non variable occurrence in M'.

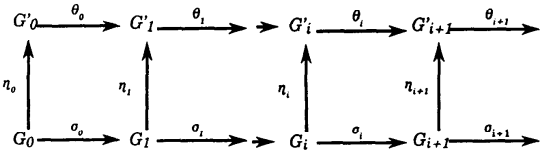
(c) Suppose P' corresponds to some subterm of ηx i.e. $\eta x/w \equiv P'$. The superposition condition gives $\theta P' \equiv \theta(\eta x/w) \equiv \theta L$. Now, $\theta(\eta x/w) = \eta x/w$ as $D(\theta) \cap Var(\eta x) = \Phi$ so that $\eta x/w \equiv \theta L$. Once again using the logic in (b) above we can show that this contradicts the assumption that η is Rp-normal.

Thus by *reductio ad absurdum* u corresponds to a non variable occurrence in M []

Lemma 5. Given an equational program P, a goal G, and an Rp-normal substitution η, if there exists a successful computation from $G_0'(=\eta G) \Rightarrow^{\theta_0} G_1' \Rightarrow^{\theta_1} G_2' \Rightarrow \ldots \Rightarrow^{\theta_n} \varepsilon$ where $\theta_n \ldots \theta_1\theta_0$ is an empty (answer) substitution i.e. $\theta_n \ldots \theta_1\theta_0 \uparrow V(G_0)$ is an empty substitution, then there is a successful computation $G_0(=G) \Rightarrow^{\sigma_0} G_1 \Rightarrow^{\sigma_1} \ldots \Rightarrow^{\sigma_n} \varepsilon$ emerging from G such that for all i, $0 \le i \le n$, there exists η_i and ρ_i such that

$D(\eta_i) \subseteq V(G_i)$ (I)

η_i is R_p-normal (II)



$\eta_0 = \eta_i \rho_i \uparrow V(G_0)$ (III) where, $\rho_0 = \varepsilon$; $\rho_{i+1} = \sigma_i \rho i$

$\eta_i G_i \equiv G_i'$ (IV)

Proof. By induction on i:

Base case. $i=0$; choosing $\eta_0 = \eta$, conditions I-IV are trivially satisfied.

Induction step. Suppose it is true for some $i \geq 0$. Let us consider the derivation $G_i' \rightarrow^{\theta_i} G_{i+1}'$.

Suppose G_i' is of the form $(\Gamma', M' = N', \Delta')$. According to the induction hypothesis G_i is of the form $(\Gamma, M = N, \Delta)$, and, $\Gamma' \equiv \eta_i \Gamma$, $M' \equiv \eta_i M$, $N' \equiv \eta_i N$, $\Delta \equiv \eta_i \Delta$.

There are two possibilities:

(A) G_{i+1}' is obtained by applying *reflection* on $M' = N'$ i.e. $\theta_i M' \equiv \theta_i N'$ since $M' \equiv \eta_i M$ and $N' \equiv \eta_i N$, we have $\theta_i \eta_i M \equiv \theta_i \eta_i N$ so that M and N may be unified using $\theta_i \eta_i$.

(B) G_{i+1}' is obtained by *superposing* a clause $(C: A = B \leftarrow Q \in \mathrm{P})$ on $M' = N'$. Without loss of generality, suppose the superposition is at an occurrence u of M'. Let $M'/u = P'$. The superposition condition gives $\tau(P') \equiv \nu(A)$, $\theta_i = (\tau \cup \nu)$ and $G_{i+1}' \equiv (\theta_i \Gamma', \theta_i M'[u \leftarrow B] = \theta_i N', \theta_i Q, \theta_i \Delta')$. By renaming C we can arrange so that $V(G_i') \cap D(\nu) = \Phi = I(\eta_i) \cap D(\nu)$ i.e. $V(G_i') \cap D(\nu) = \Phi$. P is a non-variable subterm in G_i as $P' \equiv \eta_i P$, $D(\eta_i) \subseteq V(G_i)$, η_i is $\mathrm{R_p}$-normal, and θ_i is an empty answer substitution i.e. $D(\theta_i) \cap V(G_i') = \Phi$ (Proposition 4-(i)). Thus, $\tau \eta_i P \equiv \nu(A)$ which shows superposition is possible using $\psi = \tau \eta_i \cup \nu = \tau \eta_i \cup \nu \eta_i = (\tau \cup \nu)\ \eta_i = \theta_i \eta_i$ as a unifier.

Thus in both cases the computation may be simulated using the unifier $\theta_i \eta_i$. Suppose σ_i is the minimum unifier i.e. $\theta_i \eta_i = \eta' \sigma_i$ for some η'. Note, since θ_i is an empty answer substitution and $D(\eta_i) \subseteq V(G_i)$, we have $D(\theta_i) \cap Var(\eta_i x) = \Phi$, for all $x \in V(G_i)$, whence $\theta_i \eta_i = \theta_i \cup \eta_i$.

With $V(G_{i+1}) = \{V(G_i) \cup I(\sigma_i \uparrow V(G_i))\} - D(\sigma_i \uparrow V(G_i)) \ldots (1)$

chosing $\eta_{i+1} = \eta' \uparrow V(G_{i+1}) \ldots (2)$

we have $D(\eta_{i+1}) \subseteq V(G_{i+1})$ (I) *proved.*

$$\eta_i = (\theta_i \eta_i) \uparrow V(G_i) = (\eta' \sigma_i) \uparrow V(G_i)$$
$$= (\eta_{i+1} \sigma_i) \uparrow V(G_i) \quad \ldots(3) \text{ from (1) and (2).}$$

Also, it is clear $\theta_i = \eta_{i+1} \sigma_i \uparrow (\{Var(G_i) \cup Var(A)\} - V(G_i))$

since, $D(\theta_i) \cap Var(\eta_i x) = \Phi$, and θ_i, σ_i are *mgu's.*

$\therefore \theta_i \eta_i = \eta_{i+1} \sigma_i \quad \ldots(4)$

From definitions

$I(\sigma_{i-1} \ldots \sigma_1 \sigma_0 \uparrow V(G_0)) \subseteq V(G_i) \quad \ldots(5)$

and $V(G_i) \subseteq V(G_0) \cup I(\sigma_{i-1} \ldots \sigma_1 \sigma_0 \uparrow V(G_0)) \quad \ldots(6)$

Also, $\eta = \eta_i \rho_i \uparrow V(G_0)$ ...(III) induction hypothesis

$$= ((\eta_{i+1} \sigma_i) \uparrow V(G_i)) \rho_i \uparrow V(G_0) \quad \text{from (3)}$$
$$= ((\eta_{i+1} \sigma_i) \uparrow V(G_i)) \sigma_{i-1} \ldots \sigma_1 \sigma_0 \uparrow V(G_0)$$
$$= (\eta_{i+1} \sigma_i \rho_i) \uparrow V(G_0) \quad \text{using (5) and (6)}$$
$$= (\eta_{i+1} \rho_{i+1}) \uparrow V(G_0) \quad \text{(III) } \textit{proved.}$$

In the case of reflection η_{i+1} is trivially normal.

In the case of superposition let us consider any variable x in $V(G_{i+1})$, there are two possibilities:

(i) $x \in I(\sigma_i \uparrow V(G_i))$ then $\exists y \in V(G_i)$ such that $x \in Var(\sigma_i(y))$. Since $D(\theta_i) \cap Var(\eta_i x) = \Phi$ we have, using (4), $\theta_i \eta_i y \equiv \eta_i y \equiv \eta_{i+1} \sigma_i y$, which shows η_{i+1} is Rp-normal as η_i is Rp-normal.

(ii) $\sigma_i(x) = x$ i.e. $x \notin D(\sigma_i)$ and $x \in V(G_i)$. Once again since $D(\theta_i) \cap Var(\eta_i x) = \Phi$ we have using (4) $\theta_i \eta_i x \equiv \eta_i x \equiv \eta_{i+1} \sigma_i x \equiv \eta_{i+1} x$, which shows η_{i+1} is R_P-normal as η_i is R_P-normal. (II) *proved.*

Finally,

in case of reflection

$G_{i+1}' = \theta_i G_i' = \theta_i \eta_i G_i = \eta_{i+1} \sigma_i G_i = \eta_{i+1} G_{i+1}$ and,

for superposition

$$G_{i+1}' = (\theta_i \Gamma', \theta_i M'[u \leftarrow B] = \theta_i N', \theta_i Q, \theta_i \Delta')$$
$$= \theta_i \eta_i (\Gamma, M[u \leftarrow B] = N, Q, \Delta)$$
$$= \eta_{i+1} \sigma_i (\Gamma, M[u \leftarrow B] = N, Q, \Delta)$$
$$= \eta_{i+1}(G_{i+1}) \quad \text{(IV) } \textit{proved.}$$

This completes the proof of the Lemma.

Theorem 4. [*R_P-normalized Completeness for Confluent Programs*]

Let P be a confluent equational logic program, $G = (M = N)$ a goal, and θ an Rp-normal substitution. If $\theta(M = N)$ is the logical consequence of the program P, then there is a successfully terminating computation from $M = N$ with an answer substitution σ such that $\theta = \zeta\sigma$ for some ζ.

Proof. By Corollary 1 and Lemma 5. []

6. Amlog: Implementation Issues & Pertinent Features

6.1 Implementation issues

The computation rules involve *reflection* and *superposition* only. We have proved that the computation is complete for confluent systems and *R_P-normalised* answer substitutions. The completeness implies that there is a successfully terminating computation for every logical conseqence of the program. However this does not preclude the possibility of non-terminating computations. In the following we will examine the effect of some computation strategies.

6.1.1 Reflection vs Superposition: The sequence of reflection and superposition seriously effects the computation-

e.g. given the program

$F(x) = F(a) \leftarrow F(x) = F(a)$ and the goal

$\leftarrow F(a) = F(a)$

applying superposition first i.e. the *Superposition first* strategy, will continue *ad infinitum*. While, *reflection first* strategy trivially yields the success goal.

On the other hand with *reflection first* the goal

$$\leftarrow Fact(a)=x$$

terminates with the trivial solution $x=Fact(a)$.

6.1.2 Innermost vs. Outermost: The choice of subterms has an important effect on the termination and efficiency of the computation. In the *innermost* strategy the innermost subterm with a defined function symbol at its head is chosen for superposition. While in the *outermost* strategy the outermost subterm is chosen for superposition.

Innermost computation sequences tend to be infinite whenever possible; for the case of unconditional equations O'Donell [18] has argued that the *innermost* strategy may be very expensive and inefficient and has shown that some *outermost* strategies may be optimal. For conditional equations, under certain conditions, Bergstra *et al.* [2] have shown that the *parallel outermost* strategy is optimal and terminating.

6.1.3 Simplification: An important aspect of the computation strategy adopted that retains the efficiency of functional programming over logic programming is *simplification*. It involves-

trivial removal: removal of identities $T=T$ from the goal and,

goal rewriting: unconditional rewritings on goals, as much as possible.

Note that *simplification* is a special case of *reflection* and *superposition*. With the added property of irrevocability. I.e. *simplifcations* (when confluence is guaranteed) are non-backtrackable. This results in significant savings of resources in terms of memory usage-backtrack information need not be retained for simplifications, and in terms of computation time since backtracking over reductions would be senseless for confluent programs.

Thus the strategy adopted is-

(1) Simplify as much as possible-
(2) Try reflection, if successful, go to step (1)
(3) Superpose one step, if failure backtrack
(4) Go to (1).

6.2 Features of AMLOG

The combination of the two programming paradigms *viz.* logic programming and functional programming in AMLOG has resulted in several interesting and powerful features some of which are listed below.

6.2.1 Solving equations: The capability of using embedded functions coupled with the power of inference allows one to obtain solutions of simultaneous equations in a very natural and simple manner in AMLOG.

Example.

Suppose we want to solve the two simple linear equations—

$$Add(Add(*x, *x), *y)=s(s(s(s(0)))).$$

$$Add(*x, *y)=s(s(s(0))).$$

Where the signature Σ consists of the function Add the constructor s and the constant 0. The program consists of the definition of Add as follows—

$$Add(0, *x)=*x.$$

$$Add(s(*x), *y)=s(Add(*x, *y)).$$

To obtain the solution the user will simply have to pose the query

$$\leftarrow Add(Add(*x, *x), *y)=s(s(s(s(0)))), Add(*x, *y)=s(s(s(0)))$$

The answer substitution $*x=s(0)$; $*y=s(s(0))$; is obtained.

6.2.2 Functionality: Simplification does not allow backtracking. Thus backtracking over functional dependencies is avoided. Thereby a major portion of the *cut* usage in the normal PROLOG programs is made unnecessary.

Example. Suppose we want to solve the following equation for a given n and $P(x)$—

$$P(x)=D(x, Fact(n)).$$

The signature contains the function symbols P, D, $Fact$, the constructor s and the constant 0. The program consists of the definitions of D, $Fact$ and P.

Now in the computation- $Fact(n)$ is computed by simplification using the (unconditional) definition of $Fact(n)$ and there is no backtracking over it.

6.2.3 Call by need implementation: The outermost strategy results in a call by need type implementation in the case of superposition.

Using the program

$$Car([])=[].$$

$$Car([A \mid L])=L.$$

Given the query

$$\leftarrow Car([M \mid Fact(*x)])=*y.$$

The answer substitution immediately yields $*y \equiv M$ without going into the evaluation of $Fact(*x)$ which may even be undefined in some cases.

6.2.4 Infinite data structures: By virtue of the call by need approach of superposition infinite structures may be handled. Note that the "simplify as much as possible" policy of simplification apparently contravenes this tenet. Actually, the call by need is effected only in superposition and in case the user intends making use of this feature even when the relation is functional it is only required to put the corresponding equation (definition) in the conditional form.

Example.

The program P:

$$G(*x)=[*x \mid G(s(*x))] \leftarrow true=true.$$

$$Sum([*x \mid *y])=Add(*x, Sum(*y)).$$

may be used to carry out the following pattern matching

$$Sum(G(s(0)))=Add(*x, Sum(*l))$$

to obtain the match $*x=s(0)$, $*l=Sum(s(s(0)))$. Thus the user has greater flexibility. The efficiency of functional programs to deliver answers directly by simple rewriting is retained. At the same time the user may exert control to handle divergent functions by using dummy condition $true=true$ cf. [20, 4].

7. Concluding Remarks

The language AMLOG is presented and its soundness and correctness has been established using the deductive approach. At first we have introduced a deductive system for equations, and proved its soundness and completeness. Then we have presented the equational logic programming language and investigated the relationship between the operational semantics and the declarative semantics of equational logic programs using the deductive systems. The soundness and completeness of the interpreter is proved. Some attractive features of the language have been shown.

The interpreter for AMLOG has been implemented [17]. Currently it supports both selection modes: innermost and outermost. The efficiency considerations and other studies are being carried out.

References

1. BELLIA, M., DEGANO, P. and LEVI, G. The Call-by-name semantics of a clause language with functions, in *Logic Programming* (Clark, K.L. and Tärnlund, S-Ä. eds.) (1982), 281–295.
2. BERGSTRA, J. A. and KLOP, J. W. Conditional Rewrite rules: Confluence and Termination. *Journal of Computer and System Sciences* **32** (1986), 323–362.
3. CHANG, C. L. and LEE, R. C. T. *Symbolic logic and mechanical theorem proving*, Academic press (1973).
4. DERSHOWITZ, N. and PLAISTED, D. A. Equational Programming *Report* no. UIUCDCS-R-86-1265, Dep't of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois (1986).
5. VAN EMDEN, M. H. and KOWALSKI, R. A. The Semantics of Predicate Logic as a Programming language, *J. ACM*, **23** (1976), 733–742.
6. FRIBOURG, L. Oriented Equational Clauses as a Programming language, *J. Logic Programming* **1** (1984), 165–177.
7. FRIBOURG, L. SLOG: A Logic Programming Language Interpreter based on Clausal Superposition and Rewriting. *Proc. of the 1985 symposium on Logic Programming, Boston, MA* (July, 1985), 172–184.
8. GOGUEN, J. A. and MESEGUER, J. Equality, Types, Modules and Generics for Logic programming, *J. of Logic Programming*, **1:2** (1984) 179–210.
9. GOGUEN, J. A., THATCHER, J. W., WAGNER, E. G. and WRIGHT, J. B. An Initial algebra approach to the Specification, Correctness, and Implementation of Abstract data types, in *Current Trends in Programming Methodology*, **4**, ed. Yeh, R., Prentice-Hal (1978), 80–149.
10. HOFFMANN, M. and O'DONNELL, M. J. Programming with Equations, *ACM TOPLAS*, **4** (1982), 83–112.
11. HUET, G. Confluent reductions: Abstract properties and Applications to Term rewriting systems, *J. ACM*, **27:4** (1980), 797–821.
12. HUET, G. and OPPEN, D. C. Equations and Rewrite rules: A survey, in *Formal Languages: Perspectives and open Problems*, ed. Book, R., Academic Press (1980), 349–405.
13. HOLLOT, J-M. Canonical forms and Unification, in *5th Conference on Automated Deduction* (1980), 318–334.
14. HUSSMAN, H. Unification in Conditional-Equational theories. *Technical Report* MIP-8502 Dep't of Mathematics and Information sciences, University of Passau (Jan. 1985).
15. KAPLAN, S. Simplifying Conditional Term rewriting systems: Unification, Termination and Confluence. *Research Report no. 316, University de Paris-Sud, LRI Orsay*, France (1986).
16. LLOYD, J. W. *Foundations of logic programming*, Springer-Verlag (1984).
17. MIYAKE, N., TOGASHI, A. and NOGUCHI, S. *Amalgamated Programming Languages and their implementations*, to be published in the Journal of the Japan Software Science Society.
18. O'DONELL, M. J. Computing in Systems described by Equations, *Lecture notes in Computer science*, **58**, Springer Verlag (1977).
19. SUBRAHMANYAM, P. A. and YOU, J-H. Pattern Driven Lazy reduction: A Unifying evaluation mechanism for Functional and Logic programs, *Proc. of the Eleventh ACM Symposium on Principles of Programming Languages* (1984).
20. TAMAKI, H. and SATO, T. Program transformation through Meta-shifting, *New Generation Computing*, **1** (1983), 93–98.
21. TOGASHI, A. and NOGUCHI, S. A Program transformation from Equational programs into Logic programs, *J. of Logic Programming* (1987: 4), 85–103.

(Received October 15, 1987; revised June 9, 1988)