PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

THE SEMANTICS OF A FUNCTIONAL LOGIC LANGUAGE WITH INPUT MODE

D.W. Shin, J.H. Nang, S.R. Maeng, and J.W. Cho

Department of Computer Science
Korea Advanced Institute of Science and Technology
P.O. Box 150 Cheongryang Seoul 131-650 Korea
dwshin@casun.kaist.ac.kr(Internet) ...mcvax!sorak!casun!dwshin (UUCP)

ABSTRACT

There have been several functional logic languages which distinguish the variables appearing in function terms from logical variables. Aflog, Funlog and LEAF are such languages that restrict the variables in function terms to be used only in the input mode. Although this restriction makes the languages effective, it prevents them from having semantics in the conventional logical framework.

In this paper, a new framework is presented to give logical semantics to Aflog. This framework is essentially based on Levi and Palamidessi's approach, however, it is modified and extended to meet equational theories. In addition, an extension of *SLD-resolution* based on E-unification is introduced to provide the operational semantics of Aflog programs, and the soundness results are shown. Finally, the cases are discussed in which completeness results are not obtained.

1 INTRODUCTION

During the last several years, there have been many approaches to combine functional languages and logic languages (see (Bellia 1986) for a more detailed survey of these approaches). These combinations are considered promising as they provide two different programming styles in one system which would be useful in many applications.

In functional programming, one of the central concepts is the evaluation of an expression when it is ready to be computed. This characteristic makes the formal parameters of functional languages seen one way and the evaluation deterministic. In contrast, logic programming is well known by its non-determinism caused by the non-deterministic proof procedure and I/O non-determinism (see (Reddy 1985) for a detailed comparison of these two programming styles).

In the amalgamation of functional programming and logic programming, four different approaches are usually attempted as follows:

(1) combination of logic language and LISP,
(2) use of (conditional) rewrite rules with unification,
(3) transformation of equations into clausal forms,
(4) use of extended unification

The first approach includes the ones which try to integrate logic language and LISP. LOGLISP (Robinson 1982) is a typical example. In LOGLISP, a user can invoke logic from LISP and vice versa. The latter three approaches differ from (1) in that they try to unify in a single coherent framework, providing uniform semantics.

As an attempt in the second class, Darlington et al. (Darlington et al. 1986) proposed an extended functional language which uses unification instead of pattern matching and set abstraction to obtain the effect of logical variables. This language provides all of the expressive power of logic programs whilst retaining underlying functional simplicity. Dershowitz and Plaisted (Dershowitz 1985) also proposed a programming language that unifies logic programming and functional programming by using conditional rewrite rules.

The languages which belong to the third class are based on the transformation of equations to clausal forms and on one inference rule - *SLD*-resolution. LEAF (Barbuti et al. 1986) and K-LEAF (Levi et al. 1987) belong to this class. In these languages, equations are first transformed to flat forms, where function compositions are eliminated and replaced by the logical operator *and*, and *SLD*-resolution is applied to these flat forms.

The main idea of the languages in the fourth class is extending unification into E-unification so that equality theory comes in to support functional notations. Eqlog (Goguen 1984), Jaffar et al.'s approach (Jaffar et al. 1986), Yamamoto's approach (Yamamoto 1987), Funlog (Subrahmayam 1986) and Aflog (Shin et al. 1987) belong to this class. Eqlog is based on Horn clauses and canonical equational theories, and completeness is obtained when narrowing (Hullot 1980) is used as an E-unification algorithm. Jaffar et al. defined a logic program which consists of the usual set of definite clauses and *Horn equality clauses*. They proved that *Horn equality theory* has a finest congruence relation, and Least E-model and least fixpoint can be given to a program. Yamamoto introduced a narrowing procedure into logic pro-

gramming in order to treat equational theories. He proposed a system in which both *SLD*-resolution and narrowing are considered equally and proved the completeness results. In these languages, functions are evaluated in two ways: if all arguments of a function are instantiated to ground terms, the function is evaluated as usual, and otherwise, the ground instances of the arguments are tried to be found.

The latter two languages - Funlog and Aflog are also based on Horn clauses and equational theories, however, these languages are basically different from the former languages in that they restrict the variables in function terms to be used only in one way. Although this restriction makes the languages effective, it hinders them from having semantics in conventional logical framework, because there developed no way to give the semantics to variables used only in input mode.

This paper concerns mainly on the framework giving the formal semantics to Aflog - a functional logic language with input mode. Based on this framework, Aflog is given model-theoretic and fixpoint semantics. This framework is essentially based on Levi and Palamidessi's approach (Levi 1985) however, it is modified and extended to meet equational theories. Furthermore, a refinement of Canonical unification presented in (Shin et al. 1987) will be made in this paper and the operational semantics will be introduced by the resolution with Canonical unification. Finally the soundness results will be shown and the cases will be also discussed in which completeness results are not obtained.

2 THE LANGUAGE

2.1 The Syntax

In this section, the syntax of Aflog is presented rigorously. The syntax defined below is slightly different from that presented in (Shin et al. 1987). The new language is constructor-based, and does not include guard construct for simplicity.

The language alphabet A is an 8-tuple $<C,F,P,V,=,\equiv,C^{\wedge},F^{\wedge}>$, where

i) C is a set of data constructor symbols,
ii) F is a set of function symbols,
iii) P is a set of predicate symbols,
iv) V is a set of variable symbols,
v) $=$ is a special equality symbol,
vi) $\equiv$ is a special evaluable equality symbol,
vii) $C^{\wedge}$ is a set of annotated data constructor symbols defined as $C^{\wedge} = \{c^{\wedge} \mid c \in C\}$,
viii) $F^{\wedge}$ is a set of annotated function symbols defined as $F^{\wedge} = \{f^{\wedge} \mid f \in F\}$.

Evaluable equality is slightly different from equality in that equality represents that two terms are equal, while evaluable equality informs which terms are produced via unification as well as tests equality. Syntactically speaking, equality only serves to define a function and, evaluable equality is used for evaluating a function in an argument and binding its result to the other argument. In this view, "is" relation in Prolog falls under evaluable equality.

In the language alphabet, symbols in the set i) through vi) are shown in user Aflog programs, while symbols in vii) or viii) are not. In other words, the annotated constructor (function) symbols are not included in user Aflog programs, but involved in *evaluable equality axioms*, which are used to discover the nature of evaluable equality. In the latter part of this section, *evaluable equality axioms* are defined precisely.

A *data term* is:

i) a variable symbol, or
ii) a data constructor application $c(d_1,\ldots,d_n)$, where c is an n-adic data constructor symbol and $d_1,\ldots,d_n$ are data terms. If $n = 0$, then c is considered as a constant.

A *term* is:

i) a data term, or
ii) a data constructor application $c(t_1,\ldots,t_n)$, where c is an n-adic data constructor symbol and $t_1,\ldots,t_n$ are terms, or
iii) a function application $f(t_1,\ldots,t_n)$, where f is an n-adic function symbol and $t_1,\ldots,t_n$ are terms.

A *function definition* is:

$f(d_1,\ldots,d_n) = t$, where f is an n-adic function symbol, $d_1,\ldots,d_n$ are data terms, t is a term and the following conditions are satisfied:

i) *left-linearity*: multiple occurrences of the same variable in $(d_1,\ldots,d_n)$ are not allowed;
ii) all the variables occurring in t must also occur in $(d_1,\ldots,d_n)$;
iii) all function terms are reduced to their normal forms in finite steps.

These conditions are necessary for function definitions to be canonical. In other words, if function definitions satisfy condition i), ii) and *non-ambiguity rule* (i.e., there is no superposition between equations.), they are confluent. And with condition iii), they are said to be canonical. Canonical property guarantees that any function term has its unique normal form, and Canonical unification presented in Section 4 terminates. Now, atoms are defined into two classes, according to the positions in which they are used.

head atom is:

a relational atom $p(t_1,...,t_n)$, where p is an n-adic predicate symbol and $t_1,...,t_n$ are data terms.

body atom is:

i) an *evaluable equality* $t_1 \equiv t_2$, where t_1 and t_2 are terms, or

ii) a relational atom $p(t_1,...,t_n)$, where p is an n-adic predicate symbol and $t_1,...,t_n$ are terms.

A *definite clause* is:

$A \leftarrow B_1,...,B_n$ $(n \geq 0)$, where A is a head atom and $B_1,...,B_n$ are body atoms.

A definite clause has two important characteristics. First, function terms should not appear in a head atom. It reflects that a head atom is used for procedure call through pattern matching, and procedure execution or function evaluation is carried out in body atoms. Second, evaluable equality is used instead of equality in definite clauses. In Aflog, equality is used for defining functions, while evaluable equality is employed for evaluating functions.

A *goal statement* is a formula of the goal:

$\leftarrow B_1,...,B_n$ $(n \geq 0)$

where $B_1,...,B_n$ are body atoms. If $n = 0$, then the goal is called *empty goal*, and is written $\leftarrow$.

An *Aflog program Pr* is:

a set of clauses consisting of three parts: E, P and $\{X \equiv X\}$, where P is a set of *definite clauses* $C_1,...,C_m$ and E is a set of function definitions such that for each pair of function definitions $f(d_1,...,d_n)=t$ and $f(e_1,...,e_n)=u$, $f(d_1,...,d_n)$ and $f(e_1,...,e_n)$ are not unifiable (*non-ambiguity*). *E(Pr)* and *P(Pr)* denote E and $P \bigcup \{X \equiv X\}$ respectively.

We now present an Aflog program. This is the program computing the list of factorials for a given input list.

Example 2.1

```
Pr =
   [fact(0)     = 1.
    fact(s(N)) = s(N) * fact(N).

    compute([ ],[ ]).
    compute([H|T],[NH|NT]) <-
       fact(H)≡NH, compute(T,NT).
    X ≡ X. ].
```

Then,

```
E(Pr) =
   [fact(0)     = 1.
    fact(s(N)) = s(N) * fact(N).]
P(Pr) =
   [compute([ ],[ ]).
    compute([H|T],[NH|NT]) <-
       fact(H)≡NH, compute(T,NT).
    X ≡ X. ].
```

2.2 Evaluable Equality and Annotated Programs

equality axioms prescribe the characteristics of equality in first-order logic with equality, which is defined as follows:

$EQ = \{E_I : X = X$ (the identity axiom clause)$\}$

$\bigcup \{E_S : Y=X \leftarrow X=Y\}$

$\bigcup \{E_T : X=Z \leftarrow X=Y, Y=Z\}$

$\bigcup \{E_f : f(X_1,...,X_n)= f(Y_1,...,Y_n) \leftarrow X_1=Y_1,...,X_n=Y_n;\ f \in \{C,F,C\hat{},F\hat{}\}\}$

$\bigcup \{E_p : p(X_1,...,X_n) \leftarrow X_1=Y_1,..., X_n=Y_n, p(Y_1,...,Y_n);\ p \in P,\ p \neq '='\}$

In contrast to equality $=$, evaluable equality $\equiv$ is intended to inform which terms are produced via unification as well as test equality between two arguments. Modifying *equality axioms* slightly differently, we can define *evaluable equality*.

$EV = \{V_I : X \equiv Y \leftarrow X = Y\}$

$\bigcup \{V_S : Y \equiv X \leftarrow X \equiv Y\}$

$\bigcup \{V_T : X \equiv Z \leftarrow X \equiv Y, Y \equiv Z\}$

$\bigcup \{V_f : f(X_1,...,X_n) \equiv f(Y_1,...,Y_n) \leftarrow X_1 \equiv Y_1,...,X_n \equiv Y_n;\ f \in \{C,F,C\hat{},F\hat{}\}\}$

$\bigcup \{V_{af} : f(X_1,...,X_n) \equiv f\hat{}(Y_1,...,Y_n) \leftarrow X_1 \equiv Y_1,...,X_n \equiv Y_n;\ f \in \{C,F\}\}$

Axiom V_{af} is added to model that all terms can be computed through evaluable equality $\equiv$. In $X\hat{}$, $\hat{}$ implies that X can be produced. V_{af} means that if a term $f(X_1,...,X_n)$ is provided in an argument of evaluable equality, a term which has the same value as $f(X_1,...,X_n)$ can be produced in the other argument. In the next section, some useful results from *evaluable equality axioms* are presented on the domain specially designed for Aflog.

As noted before, Aflog distinguishes the variables in function terms from logical variables. Variables in function terms are used only one way, i.e., variables should be instantiated to ground terms when their functions are to be evaluated, while logical variables are used two way as usual in logic programming. For this purpose, variables in function terms in clauses are annotated by "?", which represents the variables are used only one way and the functions only consume the values of the vari-

ables. Hence, employing the following simple translation rule, it is easy to model such behavior in Aflog.

Translation rule:

Variables in a function term in a cluase are annotated by "?".

From now on, the new program obtained by applying the translation rule to a program P is called the *annotated program* of P and denoted as $P?$. Employing the above rule, we obtain the following annotated program from Example 2.1.

Example 2.2

```
E(Pr?) =
    {fact(0)     = 1.
     fact(s(N?)) = s(N) * fact(N?).}
P(Pr?) =
    {compute([], []).
     compute([H|T],[NH|NT]) <-
        fact(H?)≡NH, compute(T,NT).
     X ≡ X. }.
```

3 DENOTATIONAL SEMANTICS

3.1 Models of Functional Logic Programs

In this section, we present interpretations and models on the universe especially introduced for modeling terms which can be produced, or consumed.

A *poset* (partial orered set) is a structure $(D, \leq)$ where D is a set and $\leq$ is a partial order relation on D. D_M denotes the set of maximal elements and D_m denotes the set of minimal elements of D.

Let $(D,\leq)$ be a poset. An n-adic left-closed relation on D is any subset R of D^n such that for every $t_1,\ldots,t_n$, $u_1,\ldots,u_n$ belonging to D, if R contains $(t_1,\ldots,t_n)$ and $u_1 \leq t_1,\ldots,u_n \leq t_n$, then R contains also $(u_1,\ldots,u_n)$. In other words, D is said to be left-closed with respect to R.

Definition 3.1 Let $Pr?$ be an annotated program. The *term universe of* $Pr?$ is a *poset* $(T, \leq)$ such that:

i) for each variable x, x belongs to T,

ii) for each n-ary constructor c of $Pr?$, $c(t_1,\ldots,t_n)$ and $c\hat{}(t_1,\ldots,t_n)$ belong to T, *if* $t_1,\ldots,t_n$ belong to T. 0-ary constructor denotes constant,

iii) for each n-ary function f of $Pr?$, $f(t_1,\ldots,t_n)$ and $f\hat{}(t_1,\ldots,t_n)$ belong to T, if $t_1,\ldots,t_n$ belong to T and are not annotated by $\hat{}$ (*function term*).

The aim of the terms annotated by "^" is to represent data structures which can be produced (computed). Rule ii) reflects that a constructor can not only be consumed, but produced. Rule iii) reflects that arguments of a function are only consumed and can not be produced. It also shows that if all arguments of a function are instantiated to ground terms, the result of the function can be produced, which is represented by $f\hat{}(t_1,\ldots,t_n)$.

The relation $\leq$ is the relation satisfying the following rules:

i) for each constant c, $c \leq c$, $c \leq c\hat{}$ and $c\hat{} \leq c\hat{}$,

ii) for each variable x, $x \leq x$,

iii) for each n-ary functor(function or constructor) c, and $t_1,\ldots,t_n,u_1,\ldots,u_n$ belonging to T, if $t_1 \leq u_1,\ldots,t_n \leq u_n$ then,

a) $c(t_1,\ldots,t_n) \leq c(u_1,\ldots,u_n)$,

b) $c(t_1,\ldots,t_n) \leq c\hat{}(u_1,\ldots,u_n)$,

c) $c\hat{}(t_1,\ldots,t_n) \leq c\hat{}(u_1,\ldots,u_n)$.

Now we define some notions necessary for introducing interpretation.

Definition 3.2 The *Universe* $(U,\leq)$ of an annotated program $Pr?$ is the substructure of the term universe $(T, \leq)$ *of* $Pr?$ containing only the ground elements of T, that is, the terms containing no occurrences of variables.

Definition 3.3 The *Base* B_{Pr} of an annotated program $Pr?$ is the set of all the expressions of the form $p(t_1,\ldots,t_n)$ such that p is an n-ary predicate, $=$ or $\equiv$ of $Pr?$ and $t_1,\ldots,t_n$ belong to $(U,\leq)$. For a subset of I of B_{Pr}, let $E(I)$ denote the set of the atoms $p(t_1,\ldots,t_n)$ in I, where I is '=', and $P(I)$ denote I - $E(I)$, respectively. Then I is said to be an *interpretation* of $Pr?$, if $P(I)$ is left-closed.

An interpretation models that if a term can be produced, it is also be able to be consumed. As a term denoting *produced* is always greater than the term denoting *consumed*, an interpretation must be left-closed with respect to any relation other than $=$. $=$ (equality) does not represent whether a term can be produced or consumed, but only represents that a term is equal to others in equational theories.

The *truth values* of ground clauses and atom conjunctions, in a given interpretation I, are defined as follows:

- A ground atom $p(t_1,\ldots,t_n)$ is true if it belongs to I.
- A ground conjunction of atoms $A_1,\ldots,A_n$ is true in I if every A_i is true in I.
- A ground clause $A \leftarrow B_1,\ldots,B_n$ is true if A is true whenever $B_1,\ldots,B_n$ is true.

Let $(D, \leq)$ be a poset. A *compatible vector* on D is any finite vector $[t_1,\ldots,t_n]$ of elements of D

such that there exists in D a least upper bound d for a set of its elements. The set of *compatible vectors* on D is denoted by C_D. For example, [f(a, b^), f(a^, b), f(a^, b^)] is a *compatible vector* on U and the least upper bound is f(a^,b^). But, [f(a, b^), g(a^,b)] is not a *compatible vector* since there exists no ordering between f and g, and we can not find the least upper bound of them. Now we are in a position to introduce valid instances of a clause involving read-only variables.

Definition 3.4 Let $A \leftarrow B_1,\ldots,B_n$ be an annotated clause. An *assignment* is a mapping α from the set of variables to C_U such that

i) The length of $\alpha(x)$ is equal to the number of occurrences of x in the clause,

ii) Let $[\alpha(x)]_n$ denote the *n-th* component of $\alpha(x)$. If there exist k occurrences of x in the clause head, and h occurrences in the clause body then;
   a) if $h>0$, $[\alpha(x)]_1 = \ldots = [\alpha(x)]_k = \text{lub}\{[\alpha(x)]_i \mid k+1 \leq i \leq k+h\}$,
   b) if $h=0$, $[\alpha(x)]_1 = \ldots = [\alpha(x)]_k$ belong to U_m,

iii) If the *i-th* occurrence of x (in the clause body) is marked by "?" then $[\alpha(x)]_i$ *is not-annotated*, and moreover;
   a) x occurs also in the clause head, or
   b) there exists j such that the main functor of $[\alpha(x)]_j$ is annotated by ^,

iv) If an occurrence of x (in the clause head) is marked by "?", then $[\alpha(x)]_i$ belongs to U_m for every i-th occurrence of x in the clause.

An assignment provides a way of obtaining a valid instance of a clause and it is similarly defined as in (Levi 1985). In first-order logic, we can assign any term to a variable, making an instance of a clause. In contrast, as read-only variable is introduced in the language, some instantiations mislead the meaning of this variable. For instance, in Example 2.2, an assignment 1^ to the read-only variable H? is not allowed, because H? means that H can only consume values and never produce 1.

Condition ii) represents that a variable in the head part of a clause can produce a term, when it is the least upper bound of the terms which are produced by another occurrences of the variable in the body part. Condition ii)-b) informs that if a variable does not have an occurrence in the body part, it only consumes terms, because it has no way of producing them. For example, if three occurrences of x in the body part produce [a^,b,c], [a,b^,c] and [a,b,c^] respectively, the occurrences of x in the head part produce [a^,b^,c^] - the least upper bound of [a^,b,c], [a,b^,c] and [a,b,c^]. Condition iii) represents that a read-only variable should receive a value from an occurrence of the same variable in the head part, or another occurrence in the body part.

Condition iv) represents that if an occurrence of a variable in the head part is annotated by "?", all occurrences of this variable should only consume a value. This condition does not apply to definite clauses in an Aflog program becuase there is no function term in the head part of a definite clause and thus no read-only variable in the head part in the corresponding annotated clause.

If C is a clause and α is an assignment, then Cα denotes the expression obtained from C by substituting, for each variable x, its i-th occurrence with $[\ \alpha\ ]_i$. For instance, if C is $p(X,Y) \leftarrow q(X,Z), r(Z,Y)$, and $\alpha = \{\{[X]_1 = a,\ [X]_2 = a\}, \{[Y]_1 = b^,\ [Y]_2 = b^\}, \{[Z]_1 = c^,\ [Z]_2 = c^\}\}$ is an assignment of C, then Cα is p(a,b^) ← q(a,c^), r(c,b^).

Definition 3.5 Let $A \leftarrow B_1,\ldots,B_n$ be an annotated clause. Let $A^$ denote the atom obtained from A by annotating by ^ all the constants and the constructors which do not contain variables whose another occurrences are involved in function terms. $A \leftarrow B_1,\ldots,B_n$ is true in I iff for every assignment α, $(A^ \leftarrow B_1,\ldots,B_n)\alpha$ is true in I. If there is no assignment, $A \leftarrow B_1,\ldots,B_n$ is defined false.

In the second clause of Example 2.2, A^ = compute([H|T], [NH|NT]^) since the first argument contains a variable H whose another occurrence is involved in fact(H?). It reflects that the first argument can not produce values, since H must be instantiated to a ground term and consumed only.

Definition 3.6 Let *Pr?* be an annotated program. An interpretation I of *Pr?* is a *model* iff for every definite clause and equation of *Pr?* is true in I, and denoted as Mod(*Pr?*). And we say a goal G is a *logical consequence* of *Pr?* if, for every interpretation I of *Pr?*, I is a model for *Pr?* implies that I is also a model for G.

Now let $\bar{s}$ be a term where all ^ are deleted in s. Then we obtain an important property of *EV* and *EQ* defined in Section 2 by the following lemma.

Lemma 3.1 For any term $s \in (U,\leq)$, $\bar{s} \equiv s$ is a logical consequence of $EV? \cup EQ?$. Moreover, $s \equiv t$ is a logical consequence of $EV? \cup EQ?$ iff $\bar{s} \equiv t$ is a logical consequence of $EV? \cup EQ?$.

Definition 3.7 Let $A_1,\ldots,A_n$ be a conjunction of atoms (goal). An *annotated substitution* θ is a mapping from variables to C_T such that

i) the length of $\theta(x)$ is equal to the number of occurrences of x in $A_1,\ldots,A_n$,

ii) if x is annotated by the symbol "?" in the posi-

tion i, then $[\theta(x)]_i$ is *not annotated* and there exists j such that $[\theta(x)]_j$ has the main functor annotated.

3.2 Minimal Model and Least Fixpoint

This section begins with introducing the Minimal model of an Aflog program. We show that an Aflog program always has the Minimal model. Next, we present the fixpoint semantics of a program and show that this semantics is identical with M_{Pr} - model theoretic semantics.

Theorem 3.2 Let $M_{Pr} = \bigcap$ Mod $(Pr? \bigcup EQ? \bigcup EV?)$ where Pr is an Aflog program. Then, M_{Pr} is also a model (Shin et al. 1988).

By the above theorem, the intersection of all models is again a model called, the *Minimal model*, for Pr. We denote this model by M_{Pr}.

Now let us introduce three transformations defined on the set of interpretations which are the bases for presenting fixpoint semantics.

Definition 3.8 Let us associate to an annotated program $Pr? = \{P? \bigcup E?\}$ a mapping T_P on the set of interpretations where E is a set of equations and P is other *clauses*. T_P is defined as follows:

$T_P(I) = \{t|\ t \leq (A\hat{})\alpha$, where $A \leftarrow B_1,\ldots,B_n$ is a clause of $P?$, α is an assignment and $B_1\alpha,\ldots,B_n\alpha$ belong to $I\}$

Theorem 3.3 T_P is monotonic and continuous (Shin et al. 1988).

Second, we define a mapping concerning on equality as follows.

Definition 3.9 Let E be a set of equations. Then we define a mapping T_E:

$T_E(I) = \{\ p(t_1,\ldots,t_n)\ |\ p(t_1,\ldots,t_n) \in I$ or $t_i \rightarrow_E u_i$ and $p(t_1,\ldots,u_i,\ldots,t_n) \in I$ for some argument t_i and $u_i\}$.

In this mapping, $t_i \rightarrow_E u_i$ means that t_i is reduced to u_i in an equational theory E. The reflexive transtive closure of relation $\rightarrow_E$ is denoted by $\stackrel{*}{\rightarrow}_E$.

Theorem 3.4 T_E is continuous (Yamamoto 1987).

Now we define inductively the set $T_E\uparrow k$ for every positive integer k as like in (Yamamoto 1987):

$T_E\uparrow 1 = Id$, where $Id = \{s = s;\ s \in (U,\leq)\}$,

$T_E\uparrow(k+1) = T_E\uparrow T_E\uparrow(k)$.

Id is the least model for $\{X=X\}$. Moreover we define

$T_E\uparrow\omega = \bigcup_{k\in\omega}(T_E\uparrow k)$.

Then we obtain the following properties for $T_E\uparrow\omega$.

Lemma 3.5 $T_E\uparrow\omega = \{\ s = t|\ s,t \in (U,\leq)$ there exists u such that $s \stackrel{*}{\rightarrow}_E u$ and $t \stackrel{*}{\rightarrow}_E u\}$ (Yamamoto 1987)

Lemma 3.6 $\bigcap$Mod$(E? \bigcup EQ?) = T_E\uparrow\omega$ iff E is confluent with respect to ground terms (Yamamoto 1987).

Finally, we present a mapping concerning on evaluable equality. Before defining the mapping, a notation denoted as $\equiv\{t_1,t_2\}$ is introduced to represent $t_1 \equiv t_2$ or, $t_2 \equiv t_1$.

Definition 3.10 Let E be a set of equations. Then we define a mapping T_V:

$T_V(I) = \{t|\ t \leq t_1 \equiv t_2$ such that $t_1 \equiv t_2 \in I$, or $t_i \rightarrow_E u_i$ and $\equiv\{t_i, u_i\} \in I$ for some argument t_j and u_j, where $i \neq j\}$.

Theorem 3.7 T_V is continuous (Shin et al. 1988).

Now we define inductively the set $T_V\uparrow k$ for every positive integers k as follows:

$T_V\uparrow 1 = \{\ t \equiv s|\ t = \bar{s} \in T_E\uparrow 0\}$,

$T_V\uparrow(k+1) = T_V\uparrow T_V\uparrow(k)$.

And we define

$T_V\uparrow\omega = \bigcup_{k\in\omega}(T_V\uparrow k)$.

From these definitions, we obtain the following important lemma and theorem.

Lemma 3.8 $T_V\uparrow\omega = \{\ s \equiv t|$ there exists u such that $\bar{s} \stackrel{*}{\rightarrow}_E u$ and $t \stackrel{*}{\rightarrow}_E u.\}$ (Shin et al. 1988).

Let

$T_{EV}(I) = T_E(I) \bigcup T_V(I)$,

and

$T_{EV}\uparrow 0 = \emptyset$,

$T_{EV}\uparrow(k+1) = T_{EV}\uparrow(T_{EV}\uparrow k)$.

And we define

$T_{EV}\uparrow\omega = \bigcup_{k\in\omega}(T_{EV}\uparrow k)$.

Theorem 3.9 $\bigcap$Mod$(E? \bigcup EQ? \bigcup EV?) = T_{EV}\uparrow\omega$ iff E is confluent with respect to ground terms (Shin et al. 1988).

To denote the refutation procedure, we can define a mapping $S_{pr}: 2^{B(Pr)} \rightarrow 2^{B(Pr)}$ by:

$S_{Pr}(I) = T_{EV(Pr)}(I) \bigcup T_{P(Pr)}(I)$.

Then we easily know that S_{Pr} is continuous.

Now we define the set $S_{Pr}\uparrow k$ for non-negative integers inductively.

$S_{Pr}\uparrow 0 = \emptyset$,

$S_{Pr}\uparrow(k+1) = S_{Pr}\uparrow(S_{Pr}\uparrow k))$.

Theorem 3.10 Let *Pr?* be an annotated program. If *E(Pr)* is confluent with respect to ground terms, then $\bigcap \text{Mod}(Pr? \bigcup EQ? \bigcup EV?) = S_{Pr}\uparrow\omega$ (Shin et al. 1988).

By the above theorem, it is shown that the Minimal model M_{Pr} and $S_{Pr}\uparrow\omega$ are equal and they are the declarative semantics of an Aflog program *Pr*, which is represented by the following expression:

$M_{Pr} = S_{Pr}\uparrow\omega = \text{lfp}(S_{Pr})$.

Another comment on the declative semantics is that $T_E\uparrow\omega$ - a subset of $S_{Pr}\uparrow\omega$ can never be obtained operationally because an equation is not allowed in a query (Refer the previous section which specifies the forms of programs and queries). In other words, as evaluable equality provides the way of evaluating a function term or testing equality, it is used instead of equality in a query.

4 OPERATIONAL SEMANTICS

This section is concerned with the operational semantics of Aflog programs.

4.1 *SLDC*-resolution

In this subsection, Canonical unification is presented formally. Canonical unification is a restricted version of *E-unification* in that it imposes a restriction that function terms should be instantiated to ground terms.

Canonical unification

Input : Two Terms T and S
Output : The E-unifier θ of T and S if unifiable, otherwise Fail message.

1) Replace each term by its normal form.
   - T_n = complete_rewriting(T)
   - S_n = complete_rewriting(S)
2) Unify T_n and S_n.
3) If unifiable, return θ which is the mgu of T_n and S_n. Otherwise, return Fail message.

Function complete_rewriting(T)

Input : a term T
Output: T_n - the normal form of T

```
case
begin
  If T is a variable, then return(T).
  If T is a constant, then return(T).
  If T is a constructor c(t1,...,tn), then for each ti,
    si = complete_rewriting(ti).
    return(c(s1,...,sn)).

  If T is a function term f(t1,...,tn), then
  S = one_step_reduction(T).
  case
  begin
    If T = S, then for each ti,
        si = complete_rewriting(ti)
        return( complete_rewriting( f(s1,...,sn)))

      If T ≠ S, then
      case
      begin
          If S is a constant, then

          return(S).
          If S is c(u1,...,un) and c is a construc-
          tor, then for each ui,
              vi = complete_rewriting(ui).
              return(c(v1,...,vn)).
          If S is a function term, then
          return( complete_rewriting( S)).
      end
  end
end
```

Function one-step-reduction(S)

Input : a term S
Output : one step reduced form of S - S_{re}

Find an equation $f = t$ whose left-hand side can be unified with S.
If $f\theta = S$, **then**
  $S_{re} = t\theta$
else
  $S_{re} = S$ •

Theorem 4.1 (Canonical unification theorem) Let E be a set of equations which provides a canonical equational theory. And let S be a pair of first-order terms, say (P,Q), whose function terms are ground. Then S is E-unifiable in the equational theory given by E, iff Canonical unification algorithm terminates and gives an E-unifier of S. If S is not E-unifiable, then the unification algorithm terminates and reports fail (Shin et al. 1988).

To introduce *SLDC*-refutation, *safe* computation rule is defined first. A computation rule R is *safe* if R always selects a literal in which all arguments of functions are instantiated to ground terms. Definitions described below are much similar to those in (Lloyd 1984).

Definition 4.1 Let G_i and C_{i+1} be $\leftarrow A_1,\ldots,A_m,\ldots,A_k$, and $A \leftarrow B_1,\ldots,B_q$, respectively, and R be a *safe* computation rule. Then G_{i+1} is *derived* from G_i and C_{i+1} using the E-unifier θ_{i+1} via R if all of the following conditions hold:

i) A_m is selected by R,

ii) $A_m\ \theta_{i+1} = A\theta_{i+1}$ (θ_{i+1} is the E-unifier of A_m and A),

iii) $G_{i+1} = \leftarrow(A_1,\ldots,A_{m-1},(B_1,\ldots,B_q),A_{m+1},\ldots,A_k)\theta_{i+1}$.

Definition 4.2 An *SLDC-derivation* of $Pr \bigcup \{G\}$ via a safe computation rule R consists of a sequence $G_0{=}G,G_1,\cdots$ of goals, a sequence $C_1,C_2,\cdots$ of program clauses and a sequence $\theta_1,\theta_2,\cdots$ of E-unifiers such that each G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1} via R.

Definition 4.3 An *SLDC-refutation* of $Pr \bigcup \{G\}$ via a safe computation rule R is a finite *SLDC*-derivation of $Pr \bigcup \{G\}$ via R which has the empty clause $\square$ as the last goal in the derivation.

Definition 4.4 Let $Pr = \{ P \bigcup E \bigcup X{\equiv}X \}$ be a program, G a goal and R a safe computation rule. An *R-computed answer substitution* θ for $Pr \bigcup \{G\}$ is the substitution obtained by the composition $\theta_1 \cdots \theta_n$, where $\theta_1,\ldots,\theta_n$ is the sequence of E-unifier's used in an *SLDC*-refutation of $Pr \bigcup \{G\}$ via R.

Now we come to the point to define the *success set* which is the procedural counterpart of the Minimal model and show the soundness results. For a substitution θ, let $\theta^{\wedge}$ be a new substitution such that the substituted term c of a variable is replaced by c^. For example, if θ = {c(x,a)/z, b/u}, then $\theta^{\wedge}$ = {c^(x,a^)/z, b^/u}.

Definition 4.5 Let Pr be a program and A^o be a new atom from $A \in B_{Pr}$ by replacing all subterms annotated ^ in A by new variables. The *Success set* of Pr is the set of all $A \in B_{Pr}$ such that $Pr \bigcup \{\leftarrow A^o\}$ has an *SLDC*-refutation with an R-computed answer substitution θ via a safe computation rule R and $A^o\theta^{\wedge} = A$.

Theorem 4.2 (Soundness of R-computed answer substitution) *Let* $Pr = \{P \bigcup E \bigcup X \equiv X\}$ be an Aflog program and let $A_1,\ldots,A_k$ be a goal. If the computation of $A_1,\ldots,A_k$ terminates successfully computing a substitution θ, then there exists an annotated substitution ψ such that for each successfully computing a substitution θ, then there exists an annotated substitution ψ such that for each variable x occurring in $A_1,\ldots,A_k$, $\theta^{\wedge}(x) = \mathrm{lub}(\psi(x))$ and $\forall(A_1,\ldots,A_k)\psi$ is a logical consequence of $\{Pr \bigcup EQ? \bigcup EV?\}$ (Shin et al. 1988).

Corollary 4.3 The success set of a program Pr is contained in M_{Pr}.

We explain what does the soundness mean, taking Example 2.2.

Example 4.1 Consider the program presented in Example 2.2 and a goal ← compute([2,3], X). This goal terminates successfully with the substitution θ = {X ← [2,6]}. Let ψ = {X ← [2^,6^]^}. Then $X\theta^{\wedge} = X\psi$ and compute([2,3], X)ψ = (compute([2,3], [2^,6^]^)) is a logical consequence of $\{Pr? \bigcup EQ? \bigcup EV?\}$.

4.2 Discussion on Incompleteness

In this subsection, the cases of programs where declarative semantics and operational semantics are not equal are discussed. Such programs can be classified into two categories. One is the programs deriving *dead-lock* and the other including a *meaningless* evaluable equality.

case 1) Programs deriving *dead-lock*

dead-lock means the situation where no *SLDC*-derivation is produced because we can not find a *safe* computation rule. That is, if there is no literal in a goal whose function terms are all instantiated to ground terms, this goal is said to be fallen into *dead-lock*. In general, dead-lock occurs when two or more literals require read-only variables to be produced in each other literal simultaneously. For example, in the clause

p(X) ← q(X,Y,f(Z?)), r(g(Y?),Z),

literal q and r fall into dead-lock because q requires that a read-only variable Z is instantiated in r and r requires that Y is instantiated in q. Hence, if a goal contains these two literals, a safe computation rule can not be found. However, as an assignment does not exclude a dead-lock situation, results which can not be inferred operationally may logically follow from a program. In the above clause, from an assignment

{{[X_1] = 1^, [X_2] = 1^}, {[Y_1] = 2^, [Y_2] = 2}, {[Z_1] = 3, [Z_2] = 3^},

p(1^) belongs to the Minimal model, while p never produces a value operationally.

case 2) Programs involving a meaningless evaluable equality.

Evaluable equality is used for evaluating function terms or unifying two terms in equational theories. Furthermore, it prescribes which terms are produced or consumed. In Example 2.2, the literal *fact(H?)* ≡ *NH* represents that *NH* is bound to the results of *fact(H?)*. For instance, if *H* is instantiated to 3, *NH* is bound to 6 - the result of fact(3). In fact, the literal fact(3) ≡ 6^ logically follows from *evaluable equality axioms* (*EV?*) with

equality axioms (*EQ?*) and the definition of function *fact*. Hence, *EV* models the behavior of evaluable equality exactly in this case, making all logical consequences of *EV* inferred operationally. However *EV* is too strong that every logical consequence of *EV* is not derived operationally. For example, consider the annotated clause:

compute([H|T], [NH|NT]) ←
fact(H?) = NH, compute(T,NT)

If we modify this clause as following:

compute([H|T], [NH|NT]) ← H ≡ Temp,
fact(Temp?) = NH, compute(T,NT)

then, the first argument of *compute* can produce values declaratively, while it can not produce any value operationally. It is because that terms can be produced through the literal $H \equiv Temp$ declaratively. However, literals like $H \equiv Temp$ are meaningless and can be removed without changing the semantics, which consequently makes that all results in the declarative semantics are deduced operationally.

5 CONCLUSIONS

We provide a logical framework in which Aflog programs have declarative semantics. This is essentially based on Levi and Palamidessi's approach (Levi 1985), however it is modified and extended to meet equational theories. In this framework, in order to model the evaluation of function terms through equality, another type of equality called evaluable equality is suggested and it is shown to be able to inform which terms are produced via unification as well as equality. Two kinds of semantics are presented based on this framework. One is the Minimal model and the other is the least fixpoint, and these two semantics are shown to be identical.

To deal with equational theories, a refinement of Canonical unification presented in (Shin et al. 1987) and *SLDC*-resolution are introduced. Consequently, soundness results are shown. However, it is not possible to obtain the completeness results because the declarative semantics does not always have dead-lock informations and every logical consequence of evaluable equality is not deduced operationally. Finally, the cases in which completeness results are not obtained are discussed.

ACKNOWLEDGEMENTS

This work has been supported in part by the Ministry of Science and Technology in Korea as a national project for the next generation computer system. We wish to thank S.B. Kim and S.K. Han for their many fruitful discussions.

REFERENCES

Barbuti, R., Bellia M., Levy, G., and Martelli, M., LEAF: A Language which integrates Logic, Equations and Functions, in: D. Degroot and G. Lindstrom (eds.), *LOGIC PROGRAMMING Functions, Relations, and Equations*, Prentice-Hall, 1986 pp. 201-238.

Bellia, M. and Levy, G., The Relation between Logic and Functional Languages: A Survey, *J. Logic Programming* 3(3):217-236 (Oct. 1986).

Darlington, J., Field A.J., and Pull H., The Unification of Functional and Logic Language, in: D. DeGroot and G. Lindstrom(eds.), *LOGIC PROGRAMMING Functions, Relations, and Equations*, Prentice-Hall, 1986, pp. 37-72.

Dershowitz, N. and Plaisted, D.A., Logic Programming cum Applicative Programming, in: *Proceedings of 1985 Symposium on Logic Programming*, Boston, 1985, pp. 54-66.

Goguen, J.A. and Meseguer, J., Equality, Types, Modules, and(Why not?) Generics for Logic Programming, *J. Logic Programming* 1(2):179-210 (1984).

Hullot, J.M., Canonical Forms and Unification, in: *Proceedings of 5th Conference on Automated Deduction*, 1980, pp. 318-334.

Jaffar, J., Lassez, J.L. and Maher, J., A Logic Programming Language Scheme, in: D. Degroot and G. Lindstrom (eds.), *LOGIC PROGRAMMING Functions, Relations, and Equations*, Prentice-Hall, 1986 pp. 441-468.

Levi, G. and Palamidessi, C., The Declarative Semantics of Logical Read-Only Variables, in: *Proceedings of 1985 Symposium on Logic Programming*, Boston, 1985, pp. 128-137.

Levi, G., Palamidessi, C., Bosco, P.G., Giovannetti E., and Moiso, C., A Completeness Semantic Characterization of K-leaf: A Logic Language with Partial Functions, in: *Proceedings of 1987 Symposium on Logic Programming*, San Francisco, 1987, pp. 318-327.

Lloyd, J.W., *Foundation of Logic Programming*, Springer-Verlag, 1984.

Reddy, U.S., On the Relationship between Logic and Functional Language, in: D. DeGroot and G. Lindstrom (eds.), *LOGIC PROGRAMMING Functions, Relations, and Equations*, Prentice-Hall, 1986, pp. 3-26.

Robinson, J.A. and Sibert, E.E., LOGLISP: Motivation, Design and Implementation, in: K.L. Clark and S.-A. Tarnlund (eds.), *Logic Programming*, Academic, London, 1982, pp. 299-313.

Shin, D.W., Nang, J.H., Han, S., and Maeng, S.R., A Functional Logic Language Based on Canonical Unification, in: *Proceedings of 1987 Symposium on Logic Programming*, San Francisco, 1987, pp. 328-334.

Shin, D.W., Nang, J.H., Maeng, S.R., and Cho, J.W., Modeling The Semantics of a Functional Logic Language with Input Mode, submitted for publication to *J. Logic Programming*, 1988.

Subrahmanyam, P.A. and You, J.H., FUNLOG: A Computational Model Integrating Logic Programming and Functional Programming, in: D. Degroot and G. Lindstrom(eds.) *LOGIC PROGRAMMING: Functions, Relations, and Equations*, Prentice-Hall, 1986, pp. 157-200.

Yamamoto, A., A Theoretic Combination of SLD-Resolution and Narrowing, in: J.L. Lassez(ed.), *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne, 1987, pp. 470-487.