TM-0828

A Formalization of Reflection in Logic Programming

by
H. Sugano (Fujitsu)

December, 1989

A Formalization of Reflection in Logic Programming

Hiroyasu SUGANO

International Institute for Advanced Study
of Social Information Science, FUJITSU LIMITED

1-17-25 Shin-Kamata, Ota-ku, Tokyo 144, Japan

E-mail: suga@iias.fujitsu.co.jp

Abstract

We propose reflective logic programming language R-Prolog and formalize its operational and declarative semantics. R-Prolog is obtained from pure Prolog by two step extensions; meta-level extension which employs new symbols quote, up, down and reflective extension which incorporates reflective computation by reflective predicates. These two extensions allows us to redefine some of extra-logical predicates of actual Prolog from a consistent framework. We also show the soundness and completeness results based on a given semantics.

1 Introduction

Logic programming language Prolog utilizes many kinds of "meta" facilities, such as predicates which can manipulate variables, atoms and clauses as data. By means of these predicates, Prolog has capabilities which pure logic language does not have as it is. For instance, a meta-interpreter of Prolog can be written easily, in 3 or 4 lines, in itself. From logical point of view, however, the semantics of Prolog is rather complicated and opaque because of the employment of these predicates. In a logical sense, variable are not data by themselves but are carriers of data. Atoms and clauses provide relations among data. Dealing with them as data can be said to be a confusion of object(data) and meta(data-processing) levels. Furthermore, several extra-logical predicates in Prolog, such as database management predicates and input/output predicates and so on, can be given no logically reasonable semantics. Even though these predicates are introduced for satisfying demands as an actual programming language, they should be given semantics in a uniform framework in order to keep logic language as a transparent programming language.

Reflection (or computational reflection), proposed by B. C. Smith [7, 8], is a notion by which we can make programming languages and computational systems compact and transparent. Smith introduced 3-lisp as a reflective dialect of lisp and showed lots of advantages of reflective computation. A reflective computational system can recognize its own computational states as data and modify them in a formerly described manner. In that sense, a reflective system is said to be a self-modifying system. In 3-lisp, environments and continuations are dealt with as its computational state, and it is shown that, by recognizing and modifying these states, many kinds of system functions of lisp can be redefined as user-defined functions in it. Taking advantage of reflection, we can amalgamate meta-level feature of programming language in a consistent manner.

Regarding the importance of reflection, several programming languages with reflective capability are proposed after 3-lisp [5, 11, 10, 9]. In the logic programming area, Tanaka [9] introduced reflective operation in concurrent logic programming language GHC, and illustrated its usefulness by constructing several applications in logic language. However, the semantics and the behavioral nature of programs of the language are not yet made clear. In this paper, we propose a reflective logic programming language *R-Prolog* from a rather more theoretical motivation.

As stated above, Prolog as an actual programming language has two problems in the logical point of view; treatments of variables and extra-logical predicates. R-Prolog can be obtained by two step extension from pure logic language corresponding to these logical problems of Prolog. These steps are called meta-level extension and reflective extension. Meta-level extension is the employment of quote, up, down symbols; we can solve the problems of variables by means of them. Reflective extension is the introduction of reflective computation by means of reflective predicates. Reflective extension allows us to redefined several extra-logical predicates in Prolog. In this paper, we provide an operational and a declarative semantics and prove the soundness and completeness of R-Prolog computation with respect to the declarative semantics. Further, we show the advantages of reflective computation in logic programming, by reconstructing Prolog from a consistent and logical viewpoint in R-Prolog.

The organization of this paper is as follows; we introduce reflective logic programming language R-Prolog in the next section; in Subsection 2.1 the syntax of R-Prolog is presented introducing up , down, quote symbols and reflective predicates; in Subsection 2.2 its computational semantics, with new unification called $\eta\mu$-unification and reflective computation rule, is described. In Section 3 a declarative semantics of R-Prolog is provided as an extension of that of pure logic programs, and some semantic properties (soundness and completeness) of R-Prolog are shown; in the section 4 we make some discussions and concluding remarks.

2 Reflective logic language R-Prolog

In this section, we introduce the syntax and an operational semantics of R-Prolog.

2.1 Syntax of R-Prolog

The syntax of R-Prolog is an extension of Horn clause logic (pure Prolog). Its language has extra three symbols, $'$ (quote), $\uparrow$ (up), $\downarrow$ (down). Furthermore, besides usual predicate symbols, a special kind of predicate symbols, called ***reflective predicates***, are included to materialize the reflective computation.

Definition. 2.1 (Language of R-Prolog)
Language of R-Prolog L is a sextuple;

$$L = \langle VAR, FUN, OP, RP, DL, SS \rangle$$

where

1. VAR is the countable set of variables.

   $VAR = \{X_1, \ldots, X_i, \ldots\}$

2. FUN is the finite set of function symbols.

   $Fun = \{f_1, \ldots, f_n\}$

   Each function symbols f_l is associated with its arity k_l, where k_l is a natural number. We always assume the existence of the special nullary function symbol *nil* and the special binary function symbol *cons* among $f_1, \ldots, f_n$.

3. OP is the finite set of ordinary predicate symbols.

   $OP = \{p_1, \ldots, p_n\}$

   Each ordinary predicate symbols p_i is associated with its arity k_i, where k_i is a natural number.

4. RP is the finite set of reflective predicate symbols.

   $RP = \{r_1, \ldots, r_n\}$

   Each reflective predicate symbols r_j is associated with its arity k_j, where k_j is a natural number.

5. DL is the set of delimiters, which are listed as follows;

   comma(,), period(.), implication symbol($\leftarrow$), parentheses(( )).

6. SS is the set of special symbols, which are listed as follows;

   $\downarrow$(down), $\uparrow$(up), $'$(quote).

□

Terms and atoms of R-Prolog are defined as follows.

Definition. 2.2 (Terms and atoms)
Terms and atoms of R-Prolog are defined recursively as follows;

1. Terms.
   (a) A variable is a term.
   (b) If t is a term, then $\downarrow t$ is a term. This term is called *a downed term.*
   (c) If t is a term or an atom, then $\uparrow t$ is a term. This term is called *an upped term.*
   (d) If s is a function symbol, an ordinary predicate symbol, a reflective predicate symbol, a term or an atom, then $'s$ is a term. This term is called *a quoted term.*
   (e) Let f be an n-ary function symbol and $t_1, \ldots, t_n$ be terms. $f(t_1, \ldots, t_n)$ is a term. This term is called *a compound term.*

2. Atoms
   (a) If $t_1, \ldots, t_n$ are terms and p is an n-ary ordinary predicate symbol, then $p(t_1, \ldots, t_n)$ is an atom. This atom is called *an ordinary atom.*

(b) If $t_1, \ldots, t_n$ are terms and r is an n-ary reflective predicate symbol, then $r(t_1, \ldots, t_n)$ is an atom. This atom is called *a reflective atom.*

(c) A downed term is an atom.

□

We write $TERM$ for the set of terms and $ATOM$ for the set of atoms. Following the conventional list notation, nil is denoted by [] and $cons(t_1, cons(t_2, \ldots, cons(t_n, nil)\ldots))$ is denoted by $[t_1, t_2, \ldots, t_n]$. These terms are called *list.* If t is a term and l is a list, term $cons(t, l)$ is also a list and denoted by $[t|l]$.

Quoted terms, upped terms and downed terms are newly introduced in R-Prolog, which allow us to deal with meta-level object legally in its own language. So we sometimes call them multi-level terms. As we describe their meaning in detail in section 2.2, a quoted term represents its quoted "term" as syntactic object. They are dealt with as ground terms because they are data as they are. Contrasted with that, upped and downed terms have somewhat dynamic feature. Variables in these terms can be binded to some terms by unifications when goals including them are executed, and after that they are transformed to their name(quoted form). In other words, they are used as information carrier from object (meta) level to meta (object) level. In fact, they are closely related to variable handling concept *freeze* and *melt* proposed by Nakashima et al. [6]. The relation between our primitives and them are discussed in section 4. Atoms of R-Prolog are defined almost same as that of usual logic programs. The distinctive difference is that downed terms can be used as atoms.

As stated above, terms without up and down symbol are considered as ones staying in the same level everytime. This leads to the following definition.

Definition. 2.3 (S-term and S-atom)

1. A term t is called *S(Static)-term* if t has no occurrence of up or down symbols. The set of S-terms is denoted by $STERM$.

2. An atom a is called *S(Static)-atom* if a is not a downed variable and each term occurring in a is an S-term. The set of S-atoms is denoted by $SATOM$. Furthermore, $SATOM_p$ denote the set of S-atoms whose predicates are p.

□

S-term and S-atoms play important roles in the operational and declarative semantics of R-Prolog.

Clauses of R-Prolog can be classified into the following three ; ordinary clauses, reflective clauses and reflective definition clauses. The next definition is for the first two clauses.

Definition. 2.4 (Ordinary and reflective clauses)

Let a_0 be an ordinary S-atom and $a_1, \ldots, a_n (n \geq 0)$ be ordinary or reflective atoms.

1. If a_i is a reflective atom for some $i (1 \leq i \leq n)$, Then

$$a_0 \leftarrow a_1, \ldots, a_n.$$

is called *reflective clause*(RC).

2. Otherwise,

$$a_0 \leftarrow a_1, \ldots, a_n.$$

is called *ordinary clause* (OC). Particularly, in the case that $n = 0$, we write it

$$a.$$

and we call it *ordinary unit clauses* (OUC).

□

Definition. 2.5 (Reflective definition clause)
Let r be a reflective predicate and $a_1, \ldots, a_n$ be ordinary (or reflective) atoms. *Reflective definition clause (RDC)* for r is

$r(arg, db, ndb, env, nenv) \leftarrow a_1, \ldots, a_n.$

where $arg, db, ndb, env, nenv$ are S-terms. $r(arg, db, ndb, env, nenv)$ is called *an LS(level shifting)-atom*. Note that an LS-atom is not an atom. □

Definition. 2.6 *A Program P* of R-Prolog is a pair of finite sets of clauses,

$$P = [P_O, P_R],$$

where P_O is a finite set of ordinary or reflective clauses, P_R is a finite set of reflective definition clauses. □

Definition. 2.7 Let $a_1, \ldots, a_n$ be ordinary or reflective atoms. *A Goal clause* of R-Prolog is defined as follows,

$\leftarrow a_1, \ldots, a_n.$ □

We present some examples of R-Prolog programs below.

In the actual representation of R-Prolog programs, we declare each reflective predicate to be reflective to distinguish it from ordinary predicates. In the followings, we employ a declarator "reflective" to specify reflective predicate symbols. The example below is a definition of *assert* in R-Prolog.

reflective $assert(X)$
$assert([X], Pr, Pr1, Sub, Sub) \leftarrow insert\_clause(X, Pr, Pr1).$

where *insert_clause* embeds X in a suitable place in Pr and return $Pr1$.

In the following, we fix the language (symbols) L of R-Prolog. Atoms, clauses, programs, and so on, in the language L are called respectively atoms, clauses, programs of L and so on .

2.2 Computation in R-Prolog

In this subsection, we describe the operational semantics of R-Prolog. Computations of R-Prolog programs are sequential (left-to-right, depth first) search with backtracking.

We first have to define the unification in order to deal with meta-level terms, i. e. upped, downed and quoted terms.

Definition. 2.8 (Substitution)
A *substitution* σ is a mapping from VAR to $STERM$ such that $\{X | \sigma X \neq X\}$, which we call the domain of σ, is finite. □

In such a case that the domain of a substitution is empty, we call it *identity substitution* and write it as ϵ. When the domain of a substitution σ is $\{X_1, \ldots, X_n\}$ and $\sigma(X_i) = t_i$ for $1 \leq i \leq n$, we write $\sigma = \{X_1/t_1, \ldots, X_n/t_n\}$.

The domains of substitutions are extended to S-terms as follows. If t is a compound term $f(t_1, \ldots, t_n)$ and σ is a substitution, application of σ to t, $t\sigma$ (following the conventional notation) , is defined as $f(t_1\sigma, \ldots, t_n\sigma)$. If t is a quoted term, $t\sigma = t$, i. e. t is dealt with as a ground term and σ does not affect t. Composition of substitutions are also defined as usual.

In R-Prolog computation, instead of usual unification, a special unification called $\eta\mu$-unification is employed to deal with upped and downed variables in actual computation. $\eta\mu$-unification is decomposed into two aspects; the partial mapping η transforming a term into its S-form and μ-unification among S-terms which are defined below. In order to define $\eta\mu$-unification, we first have to introduce an ordering relation and an equivalence relation on the set of S-terms.

Definition. 2.9

1. We define a relation $\geq_M$ on the set of S-term $STERM$ as the smallest one satisfying following conditions.
   (a) If t and s are S-terms and $s = t$, then $s \geq_M t$.
   (b) Let f be an n-ary function symbol, and $t_1, \ldots, t_n, s_1, \ldots, s_n$ be S-terms. If $t_i \geq_M s_i$ for any $i (1 \leq i \leq n)$, then $f(t_1, \ldots, t_n) \geq_M f(s_1, \ldots, s_n)$.
   (c) Let f be an n-ary function symbol, and $t_1, \ldots, t_n, s_1, \ldots, s_n$ be S-terms. If $t_i \equiv_M s_i$ for any $i (1 \leq i \leq n)$, then $['f,' s_1, \ldots, ' s_n] \geq_M 'f(t_1, \ldots, t_n)$.
2. The symmetric transitive closure of $\geq_M$ is denoted by $\equiv_M$.

□

The relation $\geq_M$ defined above is clearly a partial order relation and the relation $\equiv_M$ is clearly an equivalence relation. In the followings, $STERM / \equiv_M$ is denoted by $ESTERM$, $SATOM / \equiv_M$ is denoted by $ESATOM$. In R-Prolog computation, if two terms are equivalent in the above sense, they are identified. The μ-unification defined below unifies two terms under that constraint.

Definition. 2.10 (μ-unifiability)

1. Let t and s be S-terms. t and s are said to be ***μ-unifiable*** if there exists a substitution σ such that $t\sigma \equiv_M s\sigma$.

2. Let a and b be a pair of S-atoms or LS-atoms. a and b are said to be ***μ-unifiable*** if they have the same predicate symbol and each corresponding arguments are μ-unifiable.

□

Now, we present the μ-unification algorithm.

μ-unification algorithm

μ-unification algorithm is described as the following function μ-unify.

μ-unify(t,s):

Input: Two S-terms t and s.

Output: Returns the most general μ-unifier of t and s if it exists, and otherwise returns *fail*.

```
case
  variable($t$): return $\{t/s\}$
  variable($s$): return $\{s/t\}$
  quote($t$): Let $t =' u$.
        case
          quote($s$): Let $s =' v$.
             if $u = v$ then return   $\epsilon$
             else return   $fail$
          $u = f(t_1, \ldots, t_n)$:
               begin
                 if $s = [s_0, s_1, \ldots, s_n]$ then
                    return   $\mu$-unify-quote($t$,$s$)
                 else return   $fail$
               end

          otherwise: return   $fail$
        endcase

  quote($s$): Let $s =' u$.
        case
          $u = f(t_1, \ldots, t_n)$:
               begin
                 if $t = [t_0, t_1, \ldots, t_n]$ then
                    return   $\mu$-unify-quote($s$,$t$)
                 else return   $fail$
               end

          otherwise: return   $fail$
        endcase

  compound($t$): Let $t = f(t_1, \ldots, t_n)$.
        case
          $s = f(s_1, \ldots, s_n)$:
             return   $\mu$-unify-compound($t$,$s$)
          otherwise: return   $fail$
```

```
            endcase

        endcase
```

μ-unify-compound(t,s):

Input: Two compound S-terms $t = f(t_1, \ldots, t_n)$ and $s = f(s_1, \ldots, s_n)$.

Output: Returns the most general μ-unifier of t and s if it exists, and otherwise returns *fail*.

```
    begin
        $\theta = \epsilon$, $\xi = \epsilon$, $i = 1$
        while $\xi \neq fail$ and $i \leq n$ do
          begin
            $\theta = \mu$-unify($t_i\theta$,$s_i\theta$)
            if $\theta \neq fail$ then
               $\xi = \xi\theta$
            else $\xi = fail$ endif
          end
        return   $\xi$
    end
```

μ-unify-quote(t,s):

Input: qouted terms $t =' f(t_1, \ldots, t_n)$ and list $s = [s_0, s_1, \ldots, s_n]$.

Output: Returns the most general μ-unifier of t and s if it exists, and otherwise returns *fail*.

```
    begin
        if $\mu$-unify($'f$,$s_0$) = $fail$ then
           return   $fail$
        else
          begin
            $\theta = \epsilon$, $\xi = \epsilon$, $i = 1$
            while $\xi \neq fail$ and $i \leq n$ do
              begin
                $\theta = \mu$-unify($'t_i$,$s_i\theta$)
                if $\theta \neq fail$ then
                   $\xi = \xi\theta$
                else $\xi = fail$ endif

              end
            return   $\xi$
        end

    end
```

Definition. 2.11 Let s and t be S-terms, σ and τ be μ-unifier of s and t. σ is said to be *more general than* τ if, there exists a substitution θ s. t. $s\sigma\theta \geq_M s\tau$. □

The generality relation on μ-unifiers above is a pre-order relation. The next theorem shows the existence of the most general μ-unifier of two S-terms up to renaming.

Theorem. 2.1 (Uniqueness of most general μ-unifier)
Let s and t be S-terms. If σ and τ are maximally general μ-unifiers of s and t, then there exist substitutions θ and ξ s. t. $\sigma\theta = \tau$ and $\tau\xi = \sigma$.

proof. trivial. □

Definition. 2.12

1. A partial mapping η from $TERM \times SUBST$ to $STERM \cup SATOM$ is defined recursively as follows. Let t be a term and σ be a substitution.

$$\eta(t,\sigma) = \begin{cases} t\sigma & \text{if } t \text{ is a variable or a quoted term,} \\ '(\eta(s,\sigma)) & \text{if } t \text{ is an upped term } \uparrow s \text{ and } \eta(s,\sigma) \text{ is defined.} \\ u & \text{if } t \text{ is a downed term } \downarrow s \text{ and } \eta(s,\sigma) \text{ is defined} \\ & \text{and } \eta(s,\sigma) \equiv'_M u \\ & \text{where } u \text{ is a term or an atom.} \\ f(\eta(t_1,\sigma),\ldots,\eta(t_n,\sigma)) & \text{if } t \text{ is a compound term } f(t_1,\ldots,t_n)\text{, and each} \\ & \eta(t_i,\sigma)(1 \leq i \leq n) \text{ is defined and it is a term.} \\ \text{undefined} & \text{otherwise} \end{cases}$$

2. Let $a = p(t_1,\ldots,t_n)$ and σ be a substitution and assume that $\eta(t_i,\sigma) \in STERM$. The mapping η is extended for an atom a as follows.

$$\eta(a,\sigma) = p(\eta(t_1,\sigma),\ldots,\eta(t_n,\sigma))$$

□

The partial mapping η transforms some terms to an S-term by a substitution.

Definition. 2.13

1. A term t is $\eta\mu$-unifiable with an S-term t' in a substitution σ if $\eta(t,\sigma)$ is defined and it is μ-unifiable with t'.

2. An atom a is $\eta\mu$-unifiable with an S-atom a' in a substitution σ if $\eta(a,\sigma)$ is μ-unifiable with a'.

□

Definition. 2.14 A program and a substitution can be associated with terms representing themselves as follows.

1. Let $P = [\{c_1, \ldots, c_n\}, \{d_1, \ldots, d_m\}]$ be a program and $\sigma = \{X_{i_1}/t_1, \ldots, X_{i_l}/t_l\}$ be a substitution. An associated term $\hat{P}$ with P is defined as $['c_1, \ldots, 'c_n, 'd_1, \ldots, 'd_m]$ and an associated term $\hat{\sigma}$ with σ is defined as $[['X_{i_1}, 't_1], \ldots, ['X_{i_l}, 't_l]]$.
2. If t is an associated term with a program P or a substitution σ, then $\bar{t}$ denote the program P or the substitution σ respectively.

□

We now describe states of R-Prolog computation. Let *PROG* be the set of programs of L, *Subst* be the set of substitution of L, and *GOAL* be the set of goals of L.

Definition. 2.15 *Continuation* γ of R-Prolog is defined as a finite sequence of elements of $Goal \times Var \times Var$. The set of continuations is denoted by *Cont*. □

Definition. 2.16 The set of computational states of R-Prolog is defined as follows.

$$State = Goal \times Cont \times Prog \times Subst.$$

□

Definition. 2.17 R-computation beginning at the state $s \in State$ is a (finite or infinite) sequence of elements of $STATE$, $s_0, s_1, \ldots, s_i, \ldots$, satisfying the following conditions.

1. $s_0 = s$,
2. Assume $s_i = \langle G_i, C_i, P_i, \sigma_i \rangle (i \geq 0)$.
   (a) If G_i is empty goal,
      i. When C_i is empty, there is no descendent $s_j (j > i)$.
      ii. Otherwise, let $C_i = [c_{first}|C_{rest}]$ and $c_{first} = \langle G, V_1, V_2 \rangle$. There exists the next state $s_{i+1} = \langle G_{i+1}, C_{i+1}, P_{i+1}, \sigma_{i+1} \rangle$, with the following form;

$$\begin{aligned} G_{i+1} &= G \\ C_{i+1} &= C_{rest} \\ P_{i+1} &= \overline{V_1 \sigma_i} \\ \sigma_{i+1} &= \overline{V_2 \sigma_i} \end{aligned}$$

   (b) If $G_i = \leftarrow a_1, \ldots, a_n (n > 0)$, then
      i. The case that a_1 is an ordinary atom.
      There is a fresh variant of *OC* or *RC* $cl = b \leftarrow b_1, \ldots, b_m$ in P_O where a_1 is $\eta\mu$-unifiable with the head b in σ_i, and τ be the most general μ-unifier of $\eta(a_1, \sigma_i)$ and b. There exists the next state $s_{i+1} = \langle G_{i+1}, C_{i+1}, P_{i+1}, \sigma_{i+1} \rangle$, with the following form;

$$\begin{aligned} G_{i+1} &= \leftarrow b_1, \ldots, b_m, a_2, \ldots, a_n. \\ C_{i+1} &= C_i \\ P_{i+1} &= P_i \\ \sigma_{i+1} &= \sigma_i \cdot \tau \end{aligned}$$

ii. The case that a_1 is a reflective atom $r(t_1, \dots, t_l)$.
There is a fresh variant of RDC $cl = b \leftarrow b_1, \dots, b_m$ whose head b is unifiable with the reflective definition atom $rd = r([\eta(t_1, \sigma_i), \dots, \eta(t_l, \sigma_i)], \hat{P}_i, Y, \hat{\sigma}_i, Z)$ where Y and Z are variables not appearing before, and τ is the most general unifier of b and rd. There exists the next state $s_{i+1} = \langle G_{i+1}, C_{i+1}, P_{i+1}, \sigma_{i+1} \rangle$, with the following form;

$$\begin{aligned} G_{i+1} &= \leftarrow b_1, \dots, b_m. \\ C_{i+1} &= [\langle \leftarrow a_2, \dots, a_n., Y, Z \rangle | c_i] \\ P_{i+1} &= P_i \\ \sigma_{i+1} &= \sigma \cdot \tau \end{aligned}$$

□

In the above definition, *a fresh variant* of a clause means a variant of the clause which does not include any variables which appeared before.

Definition. 2.18 R-computation of goal G in program P is defined as an R-computation beginning at the state $s_0 = \langle G, [], P, \epsilon \rangle$. □

Definition. 2.19

1. If there is a finite R-computation $s_0, s_1, \dots, s_n$ of a goal G in a program P, it is called an R-refutation of G in P (of length n). Furthermore, if $s_n = \langle [], [], P', \sigma \rangle$, P' is called *the final program* of the R-refutation and σ is called *the final substitution* of the R-refutation.

2. Assume there is an R-refutation of a goal G in a program P. Let σ be the final substitution of the R-refutation and $FV(G)$ be the set of free variables in G. Restriction of σ to $FV(G)$, $\sigma|_{FV(G)}$, is called *an answer substitution of G in P.*

□

The program in the figure 1 is a meta-interpreter of R-Prolog.

3 Declarative Semantics of R-Prolog

In this section, we present a declarative semantics of R-Prolog. Because computational reflection is a procedural notion, we cannot adopt the usual declarative semantics given as logical consequence of programs. In order to incorporate a procedural aspect of reflective computation, we define the extended notion of interpretations and models.

3.1 R-interpretation and R-model

We first define the equivalence relation on the set of programs $PROG$.

```
solve([], Prog, Prog, Subst, Subst).
solve([A|Rest], Prog, Prog1, Subst, Subst1)←
                    reduce(A, Rest, Prog, Prog1, Subst, Subst1).

reduce(A, Rest, Prog, Prog1, Subst, Subst1)← ordinary(A),
                    get_clause(A, Prog, Subst, Cl, Mgu),
                    make_new_subst(Mgu, Subst, NewSub),
                    get_body(Cl, Body),
                    append(Body, Rest, NewGl),
                    solve(NewGl, Prog, Prog1, NewSub, Subst1).

reduce(A, Rest, Prog, Prog1, Subst, Subst1)← reflective(A),
                    A=[Pred,Arg],
                    ↓ [Pred, Arg, ↑ Prog, ↑ Prog2, ↑ Subst, ↑ Subst2],
                    solve(Rest, Prog2, Prog1, Subst2, Subst1).
```

Figure 1: Meta-interpreter of R-Prolog

Definition. 3.1 A relation $\equiv_P$ on $PROG$ is defined as follows. Let P and P' be programs.

$P \equiv_P P' \iff$ For each clause cl in P there exist a clause cl' in P' and substitutions σ and τ such that $cl\sigma = cl'$ and $cl'\tau = cl$, and vice versa.

$\equiv_P$ is clearly an equivalence relation. We define $EPROG$ as $PROG/\equiv_P$. □

Two programs are equivalent in the above sense if they are same up to renaming. The next definition introduces the reflective variant of interpretation.

Definition. 3.2 (IO-pair and R-interpretation)

1. The set IO is defined as follows.

$$IO = ESATOM \times (EPROG \times EPROG) \times (SUBST \times SUBST)$$

The element of IO is called *an IO-pair.*

2. A subset of IO is called *R-interpretation.*

□

Theorem. 3.1 The set of all R-interpretation 2^{IO} is a complete lattice with respect to set inclusion. □

In the following, elements of $EPROG$, $ESTERM$ and $ESATOM$ are denoted by $\tilde{P}$, $\tilde{t}$ and $\tilde{a}$ respectively, where P, t, a are their reprsentatives. However, equivalence classes will sometimes be denoted by representatives of themselves for simplicity in case that it is obvious from context. An IO-pair can be denoted by, for example $\langle a, P_1, P_2, \sigma_1, \sigma_2\rangle$, or $\langle \tilde{a}, \tilde{P}_1, \tilde{P}_2, \sigma_1, \sigma_2\rangle$, where a is an S-atom, P_1, P_2 are programs, σ_1, σ_2 are substitutions.

Definition. 3.3 Let $\tilde{P} \in EPROG$, $cl = a \leftarrow a_1, \ldots, a_n (n \geq 0)$ be OC, RC or RDC in P. *IO-description* of cl in program P is defined as follows.

1. If cl is OC or RC, *an IO-description* of cl in P is

$$\langle b, \tilde{P}_0, \tilde{P}_1, \ldots, \tilde{P}_n, \sigma_0, \sigma_1, \ldots, \sigma_n \rangle$$

   where

   (a) b is an S-atom μ-unifiable with a and σ_0 is an $mgmu$ of a and b,

   (b) $\sigma_1, \ldots, \sigma_n$ are substitutions, such that $\sigma_i = \sigma_0 \sigma'_i$ for some σ'_i for each i $(1 \leq i \leq n)$,

   (c) $\tilde{P}_0, \ldots, \tilde{P}_n \in EPROG$ and $\tilde{P}_0 = \tilde{P}$.

2. If cl is RDC and $a = r(t_1, \ldots, t_5)$ is LS-atom of cl where r is an n-ary reflective predicate, *an IO-description* of cl in P is

$$\langle b, \tilde{P}_0, \tilde{P}_1, \ldots, \tilde{P}_n, \sigma_0, \sigma_1, \ldots, \sigma_n \rangle$$

   where

   (a) $b = r(s_1, s_2, s_3, s_4, s_5)$ is an LS-atom unifiable with a and σ_0 is an $mgmu$ of a and b.

   (b) $\sigma_1, \ldots, \sigma_n$ are substitutions, such that $\sigma_i = \sigma_0 \sigma'_i$ for some σ'_i for each $i (1 \leq i \leq n)$

   (c) $\tilde{P}_0, \ldots, \tilde{P}_n \in EPROG$ and $\tilde{P}_0 = \tilde{P}$,

   (d) $s_1 = [u_1, \ldots, u_n]$, where $u_1, \ldots, u_n$ are S-terms,

   (e) $s_2 = \hat{P}$, and $s_4 = \hat{\phi}$ for some substitution ϕ in which each free variable in cl does not appear,

   (f) s_3 and s_5 are variables,

   (g) $\overline{s_3 \sigma_n}$ is a program and $\overline{s_5 \sigma_n}$ is a substitution.

□

The next definition is a preparation of the following definitions.

Definition. 3.4 Let P be a program. For an R-interpretation $I \in 2^{IO}$, the set of programs appearing in I, $W_P(I)$, is defined as follows;

$$W_P(I) = \begin{cases} \{\tilde{P}\} & \text{if } I = \emptyset \\ \{\tilde{P}' \in EPROG | \text{ there exist } i \in I \text{ and } \tilde{P}'' \in EPROG, i = \langle a, P', P'', \sigma_1, \sigma_2 \rangle \text{ or} \\ i = \langle a, P'', P', \sigma_1, \sigma_2 \rangle\} & \text{otherwise} \end{cases}$$

□

In the followings, we use the notation $r[s]$ for $r(u_1, \ldots, u_n)$ if r is a predicate symbol and $s = [u_1, \ldots, u_n]$.

A notion of model in R-Prolog is introduced as follows.

Definition. 3.5 Let P be a program and I be an R-interpretation. I is said to be *an R-model of P* if the followings hold;

1. $\tilde{P} \in W_P(I)$.
2. For any $\tilde{Q} \in W_P(I)$, $Q \in \tilde{Q}$, any OC or RC in Q, say $cl = a \leftarrow a_1, \ldots, a_n$, any IO-description of cl in Q,
$$\langle b, \tilde{P}_0, \ldots, \tilde{P}_n, \sigma_0, \ldots, \sigma_n \rangle,$$
and any substitution ϕ in which each free variable in cl does not appear, if for any i $(0 \le i \le n-1)$,
$$\langle \eta(a_{i+1}, \sigma_i), \tilde{P}_i, \tilde{P_{i+1}}, \phi\sigma_i, \phi\sigma_{i+1} \rangle \in I$$
then $\langle b, P_0, P_n, \phi, \phi\sigma_n \rangle \in I$.
3. For any $\tilde{Q} \in W_P(I)$, $Q \in \tilde{Q}$ any RDC in Q, say $cl = a \leftarrow a_1, \ldots, a_n$, any IO-description of cl in Q,
$$\langle b, \tilde{P}_0, \ldots, \tilde{P}_n, \sigma_0, \ldots, \sigma_n \rangle$$
where $b = r(s_1, \ldots, s_5)$ and $s_4 = \hat{\phi}$, if for any i $(0 \le i \le n-1)$,
$$\langle \eta(a_{i+1}, \sigma_i), \tilde{P}_i, \tilde{P_{i+1}}, \phi\sigma_i, \phi\sigma_{i+1} \rangle \in I$$
then $\langle r[s_1], \overline{s_2}, \overline{s_3\sigma_n}, \overline{s_4}, \overline{s_5\sigma_n} \rangle \in I$

□

Theorem. 3.2 If $J = \{I_i\}$ is a non-empty set of R-models of program P, then the intersection $\bigcap J$ is also an R-model of P.

proof. trivial. □

The intersection of all R-models of program P is denoted by $M(P)$. $M(P)$ is the smallest R-model of program P.

3.2 Fixed point semantics

In this subsection, we show the smallest R-model of P, $M(P)$, is obtained as the least fixed point of a certain continuous function on 2^{IO} determined by P.

We define the function T_P on 2^{IO} as follows. It is a reflective variant of the usual characterization function of pure Prolog.

Definition. 3.6 Let P be a program. The function $T_P : 2^{IO} \rightarrow 2^{IO}$ is defined as follows. Let $I \in 2^{IO}$.
$$T_P(I) = U_P(I) \cup V_P(I)$$
where

$U_P(I) = \bigcup_{\tilde{Q} \in W_P(I)} \bigcup_{Q \in \tilde{Q}} \bigcup_{cl \in Q_O} \{ \langle b, \tilde{P}_0, \tilde{P}_n, \phi, \phi\sigma_n \rangle | \langle b, \tilde{P}_0, \ldots, \tilde{P}_n, \sigma_0, \ldots, \sigma_n \rangle$ be an IO-description of $cl = a \leftarrow a_1, \ldots, a_n$ in $\tilde{P}_0 = \tilde{Q}$, ϕ is a substitution in which each free variable in cl does not appear, and for any i $(0 \le i \le n-1)$, $\langle \eta(a_{i+1}, \sigma_i), \tilde{P}_i, \tilde{P_{i+1}}, \phi\sigma_i, \phi\sigma_{i+1} \rangle \in I \}$

$V_P(I) = \bigcup_{\tilde{Q} \in W_P(I)} \bigcup_{Q \in \tilde{Q}} \bigcup_{cl \in Q_R} \{ \langle r[s_1], \overline{s_2}, \overline{s_3\sigma_n}, \overline{s_4}, \overline{s_5\sigma_n} \rangle | \langle b, \tilde{P}_0, \ldots, \tilde{P}_n, \sigma_0, \ldots, \sigma_n \rangle$ be an IO-description of $cl = a \leftarrow a_1, \ldots, a_n$ in $\tilde{P}_0 = \tilde{Q}$ where $b = r(s_1, \ldots, s_5)$, and $s_4 = \hat{\phi}$ and for any i $(0 \le i \le n-1)$, $\langle \eta(a_{i+1}, \sigma_i), \tilde{P}_i, \tilde{P_{i+1}}, \phi\sigma_i, \phi\sigma_{i+1} \rangle \in I \}$

□

Theorem. 3.3 Function $T_P : 2^{IO} \to 2^{IO}$ is continuous.

proof. Let X be a directed set in 2^{IO}. We have to prove $T_P(\bigcup X) = \bigcup T_P(X)$. Let $z = \langle a, \tilde{P}_0, \tilde{P}_1, \sigma_0, \sigma_1 \rangle \in IO$ be an IO-pair. Note that if $z \in T_P(I)$ for some I, $P_0 \in W_P(I)$.

First, we assume $z \in T_P(\bigcup X)$. By definition, z is in $U_P(\bigcup X)$ or $V_P(\bigcup X)$. If $z \in U_P(\bigcup X)$, there are a program $Q \in \tilde{P}_0$, an IO-description $\langle a, \tilde{Q}_0, \ldots, \tilde{Q}_n, \tau_0, \ldots, \tau_n \rangle$ of some OC or RC in Q, say $b \leftarrow b_1, \ldots, b_n$, and a substitution ϕ s. t. $\tilde{Q}_0 = \tilde{P}_0$, $\tilde{Q}_n = \tilde{P}_1$, $\phi\tau_0 = \sigma_0$, $\phi\tau_n = \sigma_1$, and $\alpha_i = \langle \eta(b_{i+1}, \tau_i), \tilde{P}_i, \tilde{P}_{i+1}, \phi\tau_i, \phi\tau_{i+1} \rangle \in \bigcup X$ for each i $(1 \leq i \leq n-1)$. By directedness of X, there exists $I \in X$ to which each of α_i belongs. Therefore, $z \in T_P(I)$, and $z \in \bigcup T_P(X)$.

If $z \in V_P(\bigcup X)$, there are a program $Q \in \tilde{P}_0$, an IO-description $\langle a', \tilde{Q}_0, \ldots, \tilde{Q}_n, \tau_0, \ldots, \tau_n \rangle$ of RDC, $b \leftarrow b_1, \ldots, b_n$, in Q, s. t. $a' = r(s_1, \ldots, s_5)$, $a = r[s_1]$, $P_0 = \overline{s_2}$, $P_1 = \overline{s_3\tau_n}$, $\sigma_0 = \overline{s_4}$, $\sigma_1 = \overline{s_5\tau_n}$, and $\alpha_i = \langle \eta(b_{i+1}, \tau_i), P_i, P_{i+1}, \sigma_0\tau_i, \sigma_0\tau_{i+1} \rangle \in \bigcup X$ for each i $(1 \leq i \leq n-1)$. By directedness of X, there exists $I \in X$ to which each of α_i belongs. Therefore, $z \in T_P(I)$, and $z \in \bigcup T_P(X)$.

Now, we prove the opposite direction. Assume $z \in \bigcup T_P(X)$. Then, for some $I \in X$, $z \in T_P(I)$. Therefore, by the fact $I \subseteq \bigcup X$ and almost same argument as above, we can conclude $z \in T_P(\bigcup X)$. □

It is well known that a continuous function on a complete lattice has the least fixed point given as the lub of ω-chain beginning at the bottom. We now write $lfp(T_P)$ for the least fixed point of T_P.

Theorem. 3.4 Let P be a program.

$$M(P) = lfp(T_P)$$

proof. From the definition of $M(P)$ and T_P, we can show that

$$I \text{ is an R-model of } P \iff T_P(I) \subseteq I$$

for R-interpretation I. Then, by the fixed point theorem on complete lattice (see Prop.5.1 in[4]),

$$\begin{aligned} M(P) &= \bigcap\{I | I \text{ is an R-model of } P\} \\ &= \bigcap\{I | T_P(I) \subseteq I\} \\ &= lfp(T_P) \end{aligned}$$

□

3.3 Some Results

The next theorem show soundness of R-refutation with respect to its declarative semantics.

Theorem. 3.5 Let P be a program and $s_0 = \langle G_0, C_0, P_0, \sigma_0 \rangle$ be a computational state, where G_0 is a goal $\leftarrow a_1, \ldots, a_n.$, $C_0 = [c_1, \ldots, c_m]$, and for each j $(1 \leq j \leq m)$, $c_j = \langle \leftarrow a_1^j, \ldots, a_{h_j}^j., X_1^j, X_2^j \rangle$. If there is an R-refutation of the length more than 1 beginning at s_0 with the final substitution ν and the final program Q', and $\tilde{P}_0 \in W_P(lfp(T_P))$, then there are a sequence of substitutions and a sequence of programs,

$$\sigma_1, \ldots, \sigma_n, \sigma_0^1, \ldots, \sigma_{h_1}^1, \ldots, \sigma_0^m, \ldots, \sigma_{h_m}^m = \nu$$
$$P_1, \ldots, P_n, P_0^1, \ldots, P_{h_1}^1, \ldots, P_0^m, \ldots, P_{h_m}^m = Q'$$

satisfying that

1. there are σ_i', $\sigma_k^{j'}$ such that $\sigma_0\sigma_i' = \sigma_i$, $\sigma_0^j\sigma_k^{j'} = \sigma_k^j$,
2. $\sigma_0^1 = X_1^1\sigma_n$, $P_0^1 = X_2^1\sigma_n$, $\sigma_0^j = X_1^j\sigma_{h_{j-1}}^{j-1}$, $P_0^j = X_2^j\sigma_{h_{j-1}}^{j-1}$ for each j $(2 \leq j \leq m)$,

and further, for each i $(1 \leq i \leq n)$, $\langle \eta(a_i, \sigma_{i-1}), P_{i-1}, P_i, \sigma_{i-1}, \sigma_i \rangle \in lfp(T_P)$, and for each j $(1 \leq j \leq m)$ and k $(1 \leq k \leq h_j)$, $\langle \eta(a_k^j, \sigma_{k-1}^j), P_{k-1}^j, P_k^j, \sigma_{k-1}^j, \sigma_k^j \rangle \in lfp(T_P)$.

proof. We will prove by induction of the length of R-refutation.

If the length of R-refutation $l = 1$, then s_0 must be $\langle \leftarrow a, [], P_0, \sigma_0 \rangle$, and there is a unit clause (OC) $b.$ in P_0 with which $\eta(a, \sigma_0)$ is μ-unifiable by an mgmu σ'. Then, $\alpha = \langle \eta(a, \sigma_0), \tilde{P}_0, \sigma' \rangle$ is an IO-description of $b.$ and by definition of T_P, $\langle \eta(a, \sigma_0), \tilde{P}_0, \tilde{P}_0, \sigma_0, \sigma_0\sigma' \rangle \in T_P(\emptyset)$. Therefore, $\langle \eta(a, \sigma_0), \tilde{P}_0, \tilde{P}_0, \sigma_0, \sigma_0\sigma' \rangle \in lfp(T_P)$.

Now we assume the proposition holds when $l = n - 1$. Let the R-refutation be $s_0, s_1, \ldots, s_{n-1}$ with the length n. We prove it according to the construction of definition2.17. If G_0 is empty, C_0 must not be empty, and s_1 is $\langle \leftarrow a_1^1, \ldots, a_{h_1}^1., [c_2, \ldots, c_m], X_1^1\sigma_0, X_2^1\sigma_0 \rangle$. By induction hypothesis, there are substitutions and programs

$$\sigma_1^1, \ldots, \sigma_{h_1}^1, \ldots, \sigma_0^m, \ldots, \sigma_{h_m}^m$$
$$P_1^1, \ldots, P_{h_1}^1, \ldots, P_0^m, \ldots, P_{h_m}^m$$

Therefore, if we let $\sigma_0^1 = X_1^1\sigma_0$ and $P_0^1 = X_2^1\sigma$, the required result is obtained.

Assume G_0 is non-empty. If a_1 is an ordinary atom and there exists a clause (OC) $cl = b \leftarrow b_1, \ldots, b_r.(r \geq 0)$ in P_0 such that b is μ-unifiable with $\eta(a_1, \sigma_0)$ by mgmu σ', and $s_1 = \langle \leftarrow b_1, \ldots, b_r, a_2, \ldots, a_n., C_0, P_0, \sigma_0\sigma' \rangle$. By the induction hypothesis, there are substitutions and programs,

$$\tau_1, \ldots, \tau_r, \sigma_2, \ldots, \sigma_n, \sigma_0^1, \ldots, \sigma_{h_1}^1, \ldots, \sigma_0^m, \ldots, \sigma_{h_m}^m$$
$$Q_1, \ldots, Q_r, P_2, \ldots, P_n, P_0^1, \ldots, P_{h_1}^1, \ldots, P_0^m, \ldots, P_{h_m}^m$$

such that $\tau_0 = \sigma_0\sigma'$, $Q_0 = P_0$, $\tau_i = \tau_0\tau_i'$ for some τ_i' for each i, and for each i $(1 \leq i \leq r)$, $\langle \eta(b_i, \tau_{i-1}), Q_{i-1}, Q_i, \tau_{i-1}, \tau_i \rangle \in lfp(T_P)$ and so on. Then, by the definition of T_P, with an IO-description of cl, $\langle \eta(a_1, \sigma_0), P_0, Q_1, \ldots, Q_r, \sigma', \sigma\tau_1', \ldots, \sigma\tau_r' \rangle$, we can have $\langle \eta(a_1, \sigma_0), P_0, Q_r, \sigma_0, \sigma_0\sigma'\tau_r' \rangle \in lfp(T_P)$ where $\tau_r = \sigma_0\sigma'\tau_r'$. Therefore, if we take $P_1 = Q_r$ and $\sigma_1 = \tau_r$, the desired result is derived.

In case that a_1 is a reflective atom, there is an RDC $cl = b \leftarrow b_1, \ldots, b_r.$ such that $s_1 = \langle \leftarrow b_1, \ldots, b_r., [\langle \leftarrow a_2, \ldots, a_n., X_1, X_2 \rangle | C_0], P_0, \sigma_0\sigma' \rangle$. By the induction hypothesis, there are substitutions and programs,

$$\tau_1, \ldots, \tau_r, \sigma_0, \ldots, \sigma_{n-1}, \sigma_0^1, \ldots, \sigma_{h_1}^1, \ldots, \sigma_0^m, \ldots, \sigma_{h_m}^m$$
$$Q_1, \ldots, Q_r, P_0, \ldots, P_{n-1}, P_0^1, \ldots, P_{h_1}^1, \ldots, P_0^m, \ldots, P_{h_m}^m$$

and $\tau_0 = \sigma_0\sigma'$, $Q_0 = P_0$, $\sigma_1 = X_2\tau_r$ and $P_1 = X_1\tau_r$ such that, for each i $(1 \leq i \leq r)$, $\langle \eta(b_i, \tau_{i-1}), Q_{i-1}, Q_i, \tau_{i-1}, \tau_i \rangle \in lfp(T_P)$ and so on. Then, by the definition of T_P, with an IO-description of cl, $\langle r(s_1, \ldots, s_5), \tilde{P}_0, \tilde{Q}_1, \ldots, \tilde{Q}_r, \sigma', \tau_1, \ldots, \tau_r \rangle$, we can have $\langle r[s_1], P_0, P_1, \sigma_0, \sigma_1 \rangle \in lfp(T_P)$.

□

Theorem. 3.6 Let P be a program and $G = \leftarrow a$ be a goal clause for an atom a. If G has R-refutation in P with the final substitution σ, and the final program P'

$$\langle a, \tilde{P}, \tilde{P}', \epsilon, \sigma \rangle \in lfp(T_P).$$

proof. Let $s_0 = \langle \leftarrow a., [], P, \epsilon \rangle$. Because there is an R-refutation beginning at s_0 with the final substitution σ, and the final program P' , by theorem 3.5, the result is obtained. □

The next theorem shows completeness of R-refutation.

Theorem. 3.7 Let a be an S-atom, P, P', P'' be a program and σ be substitutions. If

$$\langle a, \tilde{P}', \tilde{P}'', \sigma, \sigma' \rangle \in lfp(T_P).$$

then there exists an R-refutation beginning at $s_0 = \langle \leftarrow a., [], P', \sigma \rangle$ with final substitution σ' and the final program P''.

proof. As stated before, $lfp(T_P)$ is given as the lub of ω-chain beginning at the bottom, i. e.

$$lfp(T_P) = T_P \uparrow \omega = \bigcup_{n \in \omega} \{T_P \uparrow n | T_P \uparrow 0 = \emptyset, \text{ and } T_P \uparrow n = T_P(T_P \uparrow (n-1))\}$$

We prove this theorem by the induction of n of $T_P \uparrow n$.

Assume $lfp(T_P)$ is non-empty because the proof is trivial if it is empty. Let

$$\alpha = \langle a, \tilde{P}', \tilde{P}'', \sigma, \sigma' \rangle \in lfp(T_P).$$

We first assume $\alpha \in T_P \uparrow 1 = T_P(\emptyset)$. Note that $W_P(\emptyset) = \{P\}$. If a is an ordinary atom, there exist an ordinary unit clause $cl = b.$ in $Q \in \tilde{P} = \tilde{P}'$ with which $a\sigma$ is μ-unifiable by τ, and there is an IO-description of cl in $\tilde{P}$, $\langle a, \tilde{P}, \tau \rangle$. Then s_0 has a descendent $\langle [], [], P, \sigma' \rangle$ because $\sigma' = \sigma\tau$, and it can have no descendent. If $a = r(t_1, \ldots, t_n)$ is a reflective atom, there exist a unit RDC $cl = b.$ in $Q \in \tilde{P}$ with which $r(s_1, \ldots, s_5)$ is μ-unifiable by τ, where $s_1 = [t_1\sigma, \ldots, t_n\sigma]$, $s_2 = \tilde{P}$, $s_3 = X_1$, $s_4 = \tilde{\sigma}$. Then s_0 has a descendent $s_1 = \langle [], [\langle [], X_1, X_2 \rangle], P, \sigma\tau \rangle$, and $s_2 = \langle [], [], P'', \sigma' \rangle$ where $X_1\sigma\tau = P''$, $X_2\sigma\tau = \sigma'$ and it can have no descendent.

Now we assume $\alpha \in T_P \uparrow n$. For some $Q \in \tilde{Q} \in W_P(T_P \uparrow (n-1))$, there exists a clause $cl = b \leftarrow b_1, \ldots, b_n$ which support it. If a is an ordinary atom, cl is OC or RC, and there is the mgmu τ of $a\sigma$ and b and IO-description of cl in $\tilde{Q}$, $\langle a\sigma, \tilde{Q}_0, \ldots, \tilde{Q}_n, \tau_0, \ldots, \tau_n \rangle$, where $\tilde{Q}_0 = \tilde{Q}$, $\tau_0 = \tau$, and for each i $(0 \leq i \leq n-1)$, $\langle \eta(b_{i+1}, \sigma\tau_i), \tilde{Q}_i, Q_{i+1}, \sigma\tau_i, \sigma\tau_{i+1} \rangle \in T_P \uparrow (n-1)$. By induction hypothesis, for each i $(1 \leq i \leq n)$, there exist an R-refutation beginning at $s_0^i = \langle \eta(b_i, \sigma\tau_{i-1}), [], Q, \sigma\tau_{i-1} \rangle$. Then it is easy to construct an R-refutation beginning at s_0 by combining these. If a is a reflective atom, we have the result by the almost same argument. □

Theorem. 3.8 Let a be an S-atom, P, P' be programs, σ, σ' be substitutions. If

$$\langle a, \tilde{P}, \tilde{P}', \sigma, \sigma' \rangle \in lfp(T_P)$$

then there exists an R-refutation of goal $\leftarrow a.$ in P.

proof. If we let s_0 be $\langle \leftarrow a., [], P, \epsilon \rangle$, the result is derived from theorem 3.7. □

4 Discussions and Concluding Remarks

We have introduced a reflective logic programming language R-Prolog and investigated its semantics rather formally. We have not given, however, the illustrating examples which show the benefit of R-Prolog. Although we will discuss those in the other paper, we will remark on some properties and possibilities of R-Prolog in this section.

4.1 Up and Down

As stated above, up and down construction introduced in this paper is closely related to *freeze* and *melt* proposed by Nakashima *et al.* [6]. Freeze and melt were introduced in order to manipulate variables themselves as data, and it is shown that these concepts enable us to distinguish several features provided in actual Prolog. Although our motivation is quite similar to theirs, we can indicate some distinctions between our apparatus and them.

One of the distinctions is a syntactic matter that freeze and melt are provided as special predicates. For example, $freeze(X, Y)$ freezes a term to which X is bound into its frozen form to which Y is to be bound, and $melt(X, Y)$ melts the frozen form to the original one. In R-Prolog, $\uparrow X$ denotes the frozen form of term X is bound to, and $\downarrow X$ melts the frozen form to the original one.

A critical difference is that the existence of the notion of "name" or "meta", i. e. a quoting apparatus. Using freeze predicate, if X is bound to a ground term,

$$freeze(X, Y), X == Y$$

always holds. On the contrary, the corresponding R-Prolog expression,

$$X == \uparrow X$$

always fails, because upped terms ($\uparrow t$) have its quoted form ($'t$) explicitly. Conceptually in R-Prolog, quoted terms denote a names of theirs [1], and it is assumed that an object and a name of the object are different. This notion of "name" is necessary to give the consistent semantics from a logical point of view. In fact, freeze and melt are somewhat computational notions and their logical meanings are not clear.

4.2 Some Applications

Advantages of R-Prolog can be shown in several applications. In this subsection, we will mention two basic applications among them: global variables and meta-level reasoning.

[1]This notion of "name" is similar to the notion of Barklund[1]

Reflective operation on current substitutions allows us to realize global variables which are distinguished from usual logical variables. For example, the following reflective definition realizes the global variable declaration.

$$
\begin{aligned}
&\text{reflect } global(X,Y).\\
&global([X,Y],Pr,Pr1,Sub,Sub1) \leftarrow \quad get\_binding(Sub,X,Z),\\
&\qquad Z1 =\downarrow Z,\\
&\qquad Y = Z1,\\
&\qquad set\_binding(Sub,X,\uparrow Z1).
\end{aligned}
$$

By this definition, we can utilize global variables as follows;

$$
\begin{aligned}
&p(X,Y) \leftarrow s(X,Z), global('GX,Z), t(X,Y).\\
&q(X,Y) \leftarrow global('GX,Z), r(Z,X,Y).
\end{aligned}
$$

This program employs a global variable GX for communication between two predicates p and q. Note that *global* can be used for both way communication.

Meta-level reasoning is also realizable by means of reflective operations. Using meta-interpreter presented in section 2.2, well known predicate *demo* [3] can be defined in R-Prolog as follows;

$$
\begin{aligned}
&\text{Reflective } demo(Db,Gl).\\
&demo([Db,Gl],Pr,Pr,Sub,Sub1) \leftarrow solve(Gl,Db,Db1,Sub,Sub1).
\end{aligned}
$$

It has been shown that this kind of *demo* predicate makes many kind of applications realizable, such as database management, knowledge representation, and so on [3, 2]. For example, with the following programs,

$$
\begin{aligned}
&believe(Person,Knowledge) \leftarrow haskb(Person,KB), demo(KB,Knowledge).\\
&haskb(john,['lazy(paul),'lazy(\ldots),\ldots,]).
\end{aligned}
$$

the goal $\leftarrow believe(john,\uparrow lazy(X))$. retreives the person who John believes to be lazy.

4.3 Concluding remarks

We formalized the semantics of R-Prolog and proved soundness and completeness of its operational semantics with respect to the declarative one. Based on these fundamental and important properties, we believe we can discuss the formal properties of behaviors of programs with reflective operations. Works now we are concentrating on are described as follows.

In R-Prolog programming, we can say that anything is allowed to be changeable by users in some sense. Although this increases the language's flexibility, it involves somewhat dangerous situation, e. g. a given program might be a self-destroying one. We have to investigate in which case programs describe meaningful computation and in which case it leads to inconsistency. In order to enable that, much finer arguments about R-Prolog programs is required. Among programs of R-Prolog, following three classes have particular significance in studying behaviors of R-Prolog programs; programs which have no reflective predicates, programs with reflective predicates which operate on current programs only, programs with reflective predicates which operate on current substitutions only. We are very interested in making behavioral characterization of these classes.

We are also trying to make more illustrative applications for reflective computation in logic programming. Such an extention of the scope of the applications of reflection will help people to realize the substance and the importance of reflection. It is also quite important to implement this reflective language.

Acknowledgements

This research has been carried out as a part of Fifth Generation Computer System project of Japan. I wish to thank Dr. Jiro Tanaka, Dr. Masaki Murakami, Dr. Youji Kohda and Mr. Munenori Maeda for the fruitful discussions and their helpful comments.

References

[1] Jonas Barklund. What is a meta-variable in prolog? In *The Workshop on Meta-Programming in Logic Programming (META 88)*, pp. 281–292, 1988.

[2] K. Bowen. Meta-level programming and knowledge representation. *New Generation Computing*, Vol. 3, pp. 359–383, 1985.

[3] K. Bowen and R. Kowalski. Amalgamating language and metalanguage in logic programming. In S. Tarnlund, editor, *Logic Programming*, pp. 153–172. Academic Press, 1982.

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

[5] P. Maes. Reflection in an object-oriented language (draft). In *Preprint of the Workshop on Meta-Level Architecture and Reflection*, 1986.

[6] H. Nakashima, S. Tomura, and K. Ueda. What is a variable in prolog? In *FGCS '84*, pp. 327–332, 1984.

[7] B. C. Smith. Reflection and semantics in a procedural language. Technical Report 272, MIT LCS, 1982.

[8] B. C. Smith. Reflection and semantics in lisp. In *Proc. 11th ACM Symposium on Principles of Programming Languages*, pp. 23–35, 1984.

[9] J. Tanaka. Meta-interpreters and reflective operations in GHC. In *FGCS '88*, pp. 774–783, 1988.

[10] M. Wand and D. P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. In *ACM Symposium on Lisp and Functional Programming*, pp. 298–307, 1986.

[11] T. Watanabe and A. Yonezawa. Reflection in an object-oriented concurrent language. In *Proc. of ACM Conf. on OOPSLA*, September 1988.