ICOT Technical Memorandum: TM-0894

TM-0894

Reflective Computation in Logic Language and Its Semantics

by
H. Sugano

July, 1990

Reflective Computation in Logic Language and Its Semantics

論理型言語におけるリフレクティブな計算とその意味論

Hiroyasu SUGANO

菅野 博靖

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

富士通 (株) 国際情報社会科学研究所

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** formalizing its operational and declarative semantics, and we show the soundness and the completeness results based on these semantics. In R-Prolog*, we can deal with *names* of syntactic objects and computational states explicitly by means of *quote, up, down* and *reflection* facilities. As a result of that, some of extra-logical predicates of actual Prolog can be redifined from a consistent framework. At the end of the paper, we introduce an idea of reflective concurrent logic language *Rena*, which is given by incorporating concurrency in R-Prolog*.

概　要

リフレクティブな論理型言語 R-Prolog* を提案し，その操作的，および宣言的意味論を形式化する．さらに，その上で健全性，完全性の結果が成り立つことを証明する．R-Prolog* では，構文的対象の名前や計算の状態を明示的に扱うことができ，通常の Prolog が持つ非論理的述語をリフレクションという統一的な枠組で定義しなおすことができる．さらにわれわれは，R-Prolog* に並列性を導入することを試み，リフレクティブな並列論理型言語 Rena を紹介する．

1 Introduction

In this paper, we investigate reflective computation in logic programming language and its formal semantics. We proposed a reflective logic programming language R-Prolog in[8] and gave it the formal semantics. The modified version of R-Prolog, called *R-Prolog**, is presented in this paper. Same as R-Prolog, R-Prolog* can be obtained by meta-level extension and reflective extension from pure logic language. Meta-level extension is the employment of *quote, up, down* symbols; we can solve the problems of variables [5] by means of them. Reflective extension is the introduction of computational reflection[6] by means of reflective predicates. Reflective extension allows us to redefine several extra-logical predicates in Prolog. In this paper, we provide the operational and declarative semantics and prove the soundness and completeness of R-Prolog* computation with respect to the declarative semantics. Furthermore, we try to incorporate concurrency in R-Prolog* in order to investigate what reflection is to be in concurrent logic languages. We introduce a concurrent reflective logic language *Rena*, and the idea of it is described.

In logic programming area, several works on meta-programming and reflection have been carried out so far, e.g. [10, 2, 7]. Hill and Lloyd analized meta-programs in logic programming language with negation[3]. They present a many-sorted logic language in order to distinguish object level and meta level computation. Unlike their language and some other formalizations, R-Prolog* has only one sort and it amalgamate object level and meta level, just like an idea by Bowen and Kowalski [2]. This feature is not a disadvantage of our language because levels of terms and atoms are distinguished by quote symbols. Furthermore, the language has a consistent framework including capabibity of computational reflection.

The organization of this paper is as follows; we introduce reflective logic programming language R-Prolog* in the next section; in section 2 the syntax of R-Prolog* is presented introducing up , down, quote symbols and reflective predicates, and its computational semantics, with new unification called μ-unification and reflective computation rule, is described. In Section 3, a declarative semantics of R-Prolog* is provided and some semantic properties (soundness and completeness) of R-Prolog* are shown. We try to incorporate concurrency in reflective logic language in the section 4. In the section 5, we make some discussions and concluding remarks.

2 Reflective logic language 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 VAR is the countable set of variables; FUN is the finite set of function symbols; OP is the finite set of ordinary predicate symbols; RP is the finite set of reflective predicate symbols; DL is the set of delimiters, comma(,), period(.), implication($\leftarrow$), parentheses(( )); SS is the set of special symbols, $\downarrow$(down), $\uparrow$(up), $'$(quote). □

Each element of FUN, OP, RP is related to a non-negative integer called its *arity*. We always assume FUN contains two special function symbols, *cons*(arity 2) and *nil*(arity 0).

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

Definition. 2.2 (Terms, atoms, upped form and downed form)
Terms, atoms, upped and downed forms of R-Prolog* are defined recursively as follows;

1. Terms.
   (a) A variable is a term.
   (b) An upped form and a downed form is a term. *an upped term.*
   (c) If s is a function symbol, an ordinary predicate symbol, a reflective predicate symbol, up symbol, down symbol, or a variable, then $'s$ is a term. This term is called *a quoted symbol.*
   (d) Let f be an n-ary function symbol ($n \leq 0$) 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 (reflective) predicate symbol, then $p(t_1, \ldots, t_n)$ is an atom. This atom is called *an ordinary (reflective) atom.*
   (b) A downed form is an atom.
3. Upped and downed forms.
   (a) If t is a term or an atom, then $\uparrow t$ is *an upped form.*
   (b) If t is a term, then $\downarrow t$ is *a downed form.*

□

We write $TERM$ for the set of terms and $ATOM$ for the set of atoms. Note that $TERM$ and $ATOM$ are not disjoint due to downed forms. 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 $[t|l]$. Furthermore, we adopt the following syntactic sugar. For a term $t = f(t_1, \ldots, t_n)$ and an atom $a = p(t_1, \ldots, t_n)$, $'t$ stands for $['f, 't_1, \ldots, 't_n]$ and $'a$ stands for $['p, 't_1, \ldots, 't_n]$.

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.* A quoted term represents a "term" before quoted 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.

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. We call a term without up and down symbol an *S-term*. Similarly, we call an atom without up and down symbol an *S-atom*. S-terms and S-atoms play important roles in the procedural 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. In the followings, we define these kinds of clauses and some special terms. the following three definitions are mutually recursive ones.

Definition. 2.3 (Clauses and some special terms)

1. Clauses
   (a) Let a_0 be an ordinary S-atom and $a_1, \ldots, a_n$ ($n \geq 0$) be ordinary or reflective atoms. Then, $a_0 \leftarrow a_1, \ldots, a_n$. is called *reflective clause*(*RC*) if a_i is a reflective atom for some i ($1 \leq i \leq n$), *ordinary clause* (*OC*) otherwise.
   (b) Let r be a reflective predicate and $a_1, \ldots, a_n$ be ordinary (or reflective) atoms. *Reflective definition clause (RDC)* for r is $r(arg, env1, env2) \leftarrow a_1, \ldots, a_n$. where $arg, env1, env2$ are S-terms satisfying the following condition.
      i. they are either a variable or a list structure.
      ii. if $env1$ ($env2$) is a two element list, its first element is a program term and its second element is a substitution term.

      $r(arg, env1, env2)$ is called *an RD-atom*. Note that an RD-atom is not an atom.
2. Clause term, program term and substitution term
   (a) For an ordinary, a reflective and a reflective definition clause $a \leftarrow a_1, \ldots, a_n$, its *quoted form* is a list of quoted atoms, $['a, 'a_1, \ldots, 'a_n]$. This kind of terms are called *clause terms*
   (b) If a term is a variable or a list of quoted forms of clauses, it is called a *program term*.
   (c) If a term is a variable or a list whose elements are lists of a quoted variable and a quoted term, it is called a *substitution term*.

□

A Program P of R-Prolog* is a finite set of ordinary clauses, reflective clauses and reflective definition clauses. A program form is a corresponding term to a program. *A substitution* is defined as a function σ from VAR to $TERM$ whose domain (the subset of VAR whose elements are mapped to different elements by σ) is finite. A substitution form also corresponds to a substitution. For a program P and a substitution σ, a program term corresponding to P is denoted by $\hat{P}$, a substitution term corresponding to σ is denoted by $\hat{\sigma}$. On the contrary, a program corresponding to a program term t is denoted by $\bar{t}$, a substitution corresponding to a substitution term s is denoted by $\bar{s}$.

A notion of a goal in R-Prolog* is defined as almost same as that of ordinary logic language. Let $a_1, \ldots, a_n$ be ordinary or reflective atoms. *A Goal clause* of R-Prolog* is defined as follows, $\leftarrow a_1, \ldots, a_n$. It is sometimes written as $< a_1, \ldots, a_n >$ in a semantical context because we just expect them to be sequences of atoms. 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*.

$$\begin{array}{ll} \text{reflective} & assert/1 \\ \multicolumn{2}{l}{assert([X], [Pr, Sub], [Pr1, Sub]) \leftarrow insert\_clause(X, Pr, Pr1).} \end{array}$$

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

2.2 Computation of R-Prolog*

In this subsection, we describe the operational semantics of R-Prolog*. Computations of R-Prolog* programs are sequential, depth first search without backtracking.

We first have to define the unification in order to deal with multi-level terms, i. e. upped, downed and quoted terms. The unification is based on μ-equivalence relation $\equiv_M$ on $TERM \cup ATOM$.

Definition. 2.4
We define a relation $\geq_M$ on set $TERM \cup ATOM$ as the smallest one satisfying following conditions.

1. If t and s are S-terms or S-atoms and $t = s$, then $t \geq_M s$.
2. If t and s are terms or atoms without variables and $t \geq_M s$, then $'t \geq_M \uparrow t$.
3. If t is a quoted term $'s$ for some s, then $s \geq_M \downarrow t$.
4. 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 $f(t_1, \ldots, t_n) \geq_M f(s_1, \ldots, s_n)$.

□

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, $TERM/\equiv_M$ is denoted by $ETERM$, $ATOM/\equiv_M$ is denoted by $EATOM$. Each equivalence classes in $ETERM$ and $EATOM$ has the maximum element with respect to $\geq_M$, and we write $\mu(t)$ the maximum element of the equivalence class including t. 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. Let t and s be S-terms. t and s are said to be *μ-unifiable* if there exists a substitution σ such that $t\sigma \equiv_M s\sigma$. Furthermore let a and b be a pair of S-atoms or RD-atoms. a and b are said to be *μ-unifiable* if they have the same predicate symbol and each corresponding arguments are μ-unifiable. We can also define a generality relation between μ-unifiers and can prove the existence of the most general μ-unifier of two S-terms, S-atoms up to renaming. It is also proved the most general μ-unifier of terms, atoms or unifies them into an S-terms or S-atoms.

Next, we have to introduce an important partial function to give an operational semantics of reflective operation. A partial mapping $\eta : TERM \cup ATOM \rightarrow STERM \cup SATOM$ maps well-formed terms and atoms to the greatest element with respect to $\geq_M$ in the equivalence class it belongs to. For example, $\eta(\uparrow c) = 'c$, $\eta(\downarrow' s) = s$. η transforms terms and atoms to an S-term and S-atom respectively if it is defined.

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. The set of *goal queue* GQ is defined as the set of finite sequence of goals. A goal queue is represented by list notation. *Meta Continuation* γ of R-Prolog* is defined as a finite sequence of elements of $GQ \times Var \times Var$. The set of meta continuations is denoted by MC. A meta continuation stacks the remaining goals in lower levels. Variables in a meta continuation are used as conveyers of new environments.

Now, we define the computational state of R-Prolog* as follows. The set of computational states of R-Prolog* is defined as follows.

$$STATE = GQ \times MC \times Prog \times Subst.$$

a computational state consists of goal queue of current level, meta continuation which stacks remaining goals in lower levels, current program, and current substitution (binding information). A computation of R-Prolog* is represented as a sequence of computational states, which is defined as follows.

Definition. 2.5 *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 $s_i = (G_i, C_i, P_i, \sigma_i)(i \geq 0)$.
   (a) If G_i is empty,
      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 $s_{i+1} = (G_{i+1}, C_{i+1}, P_{i+1}, \sigma_{i+1})$, with the following form;

$$G_{i+1} = G, \quad C_{i+1} = C_{rest}, \quad P_{i+1} = \overline{V_1\sigma_i}, \quad \sigma_{i+1} = \overline{V_2\sigma_i}$$

(b) If $G_i = [< a_1, \ldots, a_n > | G_{irest}] (n > 0)$, then

i. The case the selected atom $a_k (1 \leq k \leq n)$ 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 $a_k \sigma_i$ is μ-unifiable with the head b, and τ be the most general μ-unifier of $a_i \sigma_i$ 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;

$$G_{i+1} = [< b_1, \ldots, b_m >, < a_1, \ldots, a_{k-1}, a_{k+1}, \ldots, a_n > | G_{irest}]$$
$$C_{i+1} = C_i, \quad P_{i+1} = P_i, \quad \sigma_{i+1} = \sigma_i \cdot \tau$$

ii. The case that a_k is a reflective atom $r(t_1, \ldots, t_l)$.
There is a fresh variant of RDC $cl = b \leftarrow b_1, \ldots, b_m$ whose head b is unifiable with the RD-atom $d = r([\eta(\uparrow t_1 \sigma_i), \ldots, \eta(\uparrow t_l \sigma_i)], [\hat{P_i}, \hat{\sigma_i}], [Y, Z])$
where Y and Z are variables not appearing before, and τ is the most general unifier of b and d. 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;

$$G_{i+1} = [< b_1, \ldots, b_m >]$$
$$C_{i+1} = [\langle [< a_1, \ldots, a_{k-1}, a_{k+1}, \ldots, a_n > | G_{irest}], Y, Z \rangle | C_i]$$
$$P_{i+1} = P_i, \quad \sigma_{i+1} = \sigma \cdot \tau$$

□

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.

The definition 2.5 describes the whole computation in R-Prolog*. The case (a) is for the current goal quere is empty. In this case, if the meta-continuation is empty, the R-continuation terminates at that state. Otherwise, there exist the remaining goals of lower levels and the next goal should be the goal at the top of the meta-continuation. The next environment is represented by the variable in the top of the meta-continuation. When the goal queue is not empty, an atom in the goals at the top should be selected. If the atom is an ordinary one, the R-computation succeeds same as pure logic programming language. If the atom is a reflective one, say $r(t_1, \ldots, t_n)$, all the arguments are upped and the function η is applied to them, RD-atom $r([\eta(\uparrow t_1 \sigma_i), \ldots, \eta(\uparrow t_l \sigma_i)], [\hat{P_i}, \hat{\sigma_i}], [Y, Z])$ is constructed where $[\hat{P_i}, \hat{\sigma_i}]$ is the current environment, and a reflective definition clause is selected whose head is unifiable with this RD-atom.

The next definition defines an R-computation at the top level.

Definition. 2.6 R-computation of goal G in program P is defined as an R-computation beginning at the state $s_0 = \langle G, [], P, \epsilon \rangle$. □

If there is a finite R-computation $s_0, s_1, \ldots, s_n$ 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. 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.

2.3 Examples of R-Prolog* programs

Up and down

In R-Prolog[8], up and down construction was closely related to *freeze* and *melt* proposed by Nakashima *et al.* [5]. That is, up (down) transforms only terms to their name (the name of terms to terms themselves) in one direction. In R-Prolog*, however, they are used in both direction. See the following simple example.

$$p(X) \leftarrow q(\uparrow X).$$
$$q('a).$$
$$q('b).$$

In this example, goal $\leftarrow p('a)$. succeeds, but $\leftarrow p(a)$. fails. And $\leftarrow p(X)$. returns the answer $X = a$ or $X = b$. Note that $\leftarrow p(X)$ fails in R-Prolog with the same example.

However, reflective call do a kind of freeze operation. When a reflective goal is executed, its arguments are freezed and transformed into their "names". So, freeze operation can be redefined by means of reflection as follows.

reflect $\quad freeze/2, melt/2$
$freeze([X, Y], [P, S], [P, S1]) \leftarrow insert\_binding(S, [Y, \uparrow X], S1).$
$melt([X, Y], [P, S], [P, S1]) \leftarrow insert\_binding(S, [Y, \downarrow X], S1).$

Predicate *var*

The predicate *var* in Prolog is somewhat problematic in logical sense. Hill and Lloyd tried to give the logical semantics in their many-sorted logic language which strictly distinguishs meta and object levels[3]. Their language is similar to ours in the sense that a notion of quote (or name) is utilized in it. However, there are a large amount of difference between them, such that their language does not have up and down construction, it can deal with "negation" but ours cannot, and so on. Although the predicate checking whether a term is a variable is difficult to define generally, they showed it is possible by means of "negation". Because we do not have "negation" in R-Prolog*, we cannot adopt the same manner. However, if we extend R-Prolog* by allowing internal representation for quoted variables in a suitable style, we can define the predicate *variable* to check if a terms is a variable.

Meta Programming

Meta-level reasoning is also realizable by means of reflective operations. Using meta-interpreter presented in section 2.2, well known predicate *demo* [2] can be defined in R-Prolog* as follows;

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

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 [2, 1]. For example, with the following programs,

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

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

3 Declarative semantics

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

$$P \equiv_P P' \iff \begin{array}{l}\text{For each clause } cl \text{ in } P \text{ there exist a clause } cl' \text{ in } P' \text{ and substitutions } \sigma \text{ and } \tau\\ \text{such that } cl\sigma = cl' \text{ and } cl'\tau = cl, \text{ and vice versa.}\end{array}$$

$\equiv_P$ 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.

Now, we define the reflective variant of the notion of interpretation. The set of *IO-pair* IO is defined as $IO = EATOM \times ENV \times ENV$, where ENV is defined as $ENV = EPROG \times SUBST$. A subset of IO is called *R-interpretation*. It is easily shown that the set of all R-interpretation 2^{IO} is a complete lattice with respect to set inclusion. In the following, elements of $EPROG$, $ETERM$ and $EATOM$ 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 their representatives for simplicity in case that it is obvious from context.

Definition. 3.1 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

$$\langle\langle b, \tilde{P}_0, \sigma_0\rangle, \langle a_{k_1}, \tilde{P}_1, \sigma_1\rangle, \ldots, \langle a_{k_n}, \tilde{P}_n, \sigma_n\rangle\rangle$$

where

(a) $a_{k_1}, \ldots, a_{k_n}$ is a permuted sequence of $a_1, \ldots, a_n$.

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

(c) $\sigma_1, \ldots, \sigma_n$ are substitutions, such that $\sigma_i = \sigma_0\sigma_i'$ for some σ_i' for each i $(1 \leq i \leq n)$,

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

2. If cl is RDC and $a = r(t_1, t_2, t_3)$ is RD-atom of cl where r is an n-ary reflective predicate, ***an IO-description*** of cl in P is

$$\langle\langle b, \tilde{P}_0, \sigma_0\rangle, \langle a_{k_1}, \tilde{P}_1, \sigma_1\rangle, \ldots, \langle a_{k_n}, \tilde{P}_n, \sigma_n\rangle\rangle$$

where

(a) $a_{k_1}, \ldots, a_{k_n}$ is a permuted sequence of $a_1, \ldots, a_n$.

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

(c) $\sigma_1, \ldots, \sigma_n$ are substitutions, such that $\sigma_i = \sigma_0\sigma_i'$ for some σ_i' for each i $(1 \leq i \leq n)$

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

(e) $s_1 = [u_1, \ldots, u_n]$, where n is the arity of r and $u_1, \ldots, u_n$ are S-terms,

(f) $s_2 = [\dot{P}, \dot{\phi}]$ for some substitution ϕ in which each free variable in cl does not appear,

(g) $s_3 = [X_1, X_2]$ where X_1 and X_2 are variables, $\overline{X_1\sigma_n}$ is a program and $\overline{X_2\sigma_n}$ is a substitution.

□

Before defining a notion of model in R-Prolog*, we define a auxiliary function as a preparation of that. For a program P and an R-interpretation $I \in 2^{IO}$, $W_P(I)$ is defined as $\{\tilde{P}\}$ if $I = \emptyset$. Otherwise, $W_P(I)$ is defined as the set of programs occuring in I.

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

Definition. 3.2 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,

$$\langle\langle b, \tilde{P}_0, \sigma_0\rangle, \langle a_{k_1}, \tilde{P}_1, \sigma_1\rangle, \ldots, \langle a_{k_n}, \tilde{P}_n, \sigma_n\rangle\rangle$$

and any substitution ϕ in which each free variable in cl does not appear, if for any i $(0 \leq i \leq n-1)$, $\langle a_{k_{i+1}}\sigma_i, [\tilde{P}_i, \phi\sigma_i], [\tilde{P}_{i+1}, \phi\sigma_{i+1}]\rangle \in I$, if $a_{k_{i+1}}$ is an ordinary atom, $\langle \eta(up(a_{k_{i+1}}\sigma_i)), [\tilde{P}_i, \phi\sigma_i], [\tilde{P}_{i+1}, \phi\sigma_{i+1}]\rangle \in I$, if $a_{k_{i+1}}$ is a reflective atom, then $\langle b, [P_0, \phi], [P_n, \phi\sigma_n]\rangle \in I$.

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,

$$\langle\langle b, \tilde{P}_0, \sigma_0\rangle, \langle a_{k_1}, \tilde{P}_1, \sigma_1\rangle, \ldots, \langle a_{k_n}, \tilde{P}_n, \sigma_n\rangle\rangle$$

where $b = r(s_1, s_2, s_3)$ and $s_2 = [\dot{Q}, \dot{\phi}]$, if for any i $(0 \leq i \leq n-1)$, $\langle a_{k_{i+1}}\sigma_i, [\tilde{P}_i, \phi\sigma_i], [\tilde{P}_{i+1}, \phi\sigma_{i+1}]\rangle \in I$, if $a_{k_{i+1}}$ is an ordinary atom, $\langle \eta(up(a_{k_{i+1}}\sigma_i)), [\tilde{P}_i, \phi\sigma_i], [\tilde{P}_{i+1}, \phi\sigma_{i+1}]\rangle \in I$, if $a_{k_{i+1}}$ is a reflective atom, then $\langle r[s_1], [Q, \phi], \overline{[X_1\sigma_n, X_2\sigma_n]}\rangle \in I$.

□

It is easily shown that, the intersection $\bigcap J$ is also an R-model of P, where $J = \{I_i\}$ is a non-empty set of R-models of program P, 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 and some results

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. Furthermore, the soundness and completeness results based on given semantics are proved. We have shown these results for R-Prolog in a previous paper [8]. Because proofs of those result in that paper are able to apply to theorems in the followings, we omit the proofs of the following theorems.

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

$U_P(I) \;=\; \bigcup_{\tilde{Q} \in W_P(I)} \bigcup_{Q \in \tilde{Q}} \bigcup_{cl \in Q_O} \{\langle b, [\tilde{P}_0, \phi], [\tilde{P}_n, \phi\sigma_n]\rangle | \langle\langle b, \tilde{P}_0, \sigma_0\rangle, \langle a_{k_1}, \tilde{P}_1, \sigma_1\rangle, \ldots, \langle a_{k_n}, \tilde{P}_n, \sigma_n\rangle\rangle$ 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 $i\,(0 \leq i \leq n-1), \langle a_{k_{i+1}}\sigma_i, [\tilde{P}_i, \phi\sigma_i], [\tilde{P_{i+1}}, \phi\sigma_{i+1}]\rangle \in I$, if $a_{k_{i+1}}$ is an ordinary atom, $\langle \eta(up(a_{k_{i+1}}\sigma_i)), [\tilde{P}_i, \phi\sigma_i], [\tilde{P_{i+1}}, \phi\sigma_{i+1}]\rangle \in I$, if $a_{k_{i+1}}$ is a reflective atom. $\}$

$V_P(I) \;=\; \bigcup_{\tilde{Q} \in W_P(I)} \bigcup_{Q \in \tilde{Q}} \bigcup_{cl \in Q_R} \{\langle r[s_1], [Q, \phi], [\overline{X_1\sigma_n}, \overline{X_2\sigma_n}]\rangle | \langle\langle b, \tilde{P}_0, \sigma_0\rangle, \langle a_{k_1}, \tilde{P}_1, \sigma_1\rangle, \ldots, \langle a_{k_n}, \tilde{P}_n, \sigma_n\rangle\rangle$ 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 $i\,(0 \leq i \leq n-1), \langle a_{k_{i+1}}\sigma_i, [\tilde{P}_i, \phi\sigma_i], [\tilde{P_{i+1}}, \phi\sigma_{i+1}]\rangle \in I$, if $a_{k_{i+1}}$ is an ordinary atom, $\langle \eta(up(a_{k_{i+1}}\sigma_i)), [\tilde{P}_i, \phi\sigma_i], [\tilde{P_{i+1}}, \phi\sigma_{i+1}]\rangle \in I$, if $a_{k_{i+1}}$ is a reflective atom. $\}$

□

Theorem. 3.1 Function $T_P : 2^{IO} \to 2^{IO}$ is continuous. □

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. Then, we can get the following result.

Theorem. 3.2 Let P be a program.

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

□

This theorem shows that the smallest R-model of an R-Prolog* progam P is obtained as the least fixed point of the continuous function T_P. This corresponds to the well-known result on pure logic language[4].

The following theorems show the soundness and completeness of R-refutation with respect to the declarative semantics defined above.

Theorem. 3.3 (Soundness of R-computation)
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'

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

□

This theorem shows that, for an atom a, if there exists an R-refutation of $\leftarrow a$ in a program P with the final program P' and the final substitution σ, the IO-pair of the initial environment $\langle P, \epsilon\rangle$ and the final environment $\langle P', \sigma\rangle$ for a is in the minimal model of P.

The next theorem shows the converse of theorem3.3, i.e. completeness of R-refutation.

Theorem. 3.4 (Completeness of R-computation)
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. □

4 Concurrency

In this section, we describe an idea how to incorporate concurrency in reflective logic language. Concurrency brought several problems to logic language, such as one of syncronization. Following currently presented concurrent logic languages, especially GHC[9], we adopt notions of guard, committed-choice and the suspension rule of input guard. Thus, our language can be said to be a kind of reflective dialect of flat GHC. We call this concurrent reflective logic language *Rena*. Althouth the sematics of Rena in detail will be presented in other paper, we describe just a general framework of Rena in the followings.

4.1 From R-Prolog* to Rena

In R-Prolog* computation, an atom in the goal clause is selected, candidate clauses are tested if they are unifiable with the selected atom sequentially, the first one satisfying the condition is selected and other possibilities are discarded at that time. That is, R-Prolog* computation is executed sequentially without backtracking, and it has a kind of commitment mechanism. In Rena, like GHC, atoms in the goal clauses are executed concurrently, and the commitment mechanism is extended by input guard commitment rules.

Rena has an extra symbol *commit* (|) and the syntax of Rena is similar to GHC. Ordinary and reflective clauses of Rena is defined as follows;

$$h \leftarrow g_1, \ldots, g_m | b_1, \ldots, b_n$$

where h is an ordinary atom called head, $g_1, \ldots, g_m (m \geq 0)$ and $b_1, \ldots, b_n (n \geq 0)$ are atoms. The left hand side of commit is called *guard part* and the right hand side of commit is called *body part*. Reflective definition clauses are defined like in R-Prolog*. Furthermore, we give a restriction on atoms $g_1, \ldots, g_m$ in guard part. Predicates in these atoms have to be *non-productive* predicates in the program, that is, executions of these atoms do not produce any bindings. This restriction corresponds to a notion of "flat" in concurrent logic language.

What should be explicitly dealt with as a computational states in concurrent language when reflective computation occurs? The answer depends on how the semantics of GHC is given. While we do not describe it here, we give an example. We can consider input and output stream as computational states in addition to that of R-Prolog*. $ENV = PROG \times SUBST \times INS \times OUTS$ Examples of Rena reflective programs are followings;

$$\begin{array}{l} \text{reflective} \quad read/1 \\ read([X],[P,S,[A|IS],OS],[P,S1,IS,OS]) \leftarrow |insert\_binding(S,[X,A],S1). \end{array}$$

This defines input predicate *read* by means of reflection.

When we incorporate reflective computation in concurrent language, we have to consider scope of reflection. Scope of reflection means what amount of current computation should be influenced when reflective computation occurs and states has been changed. In R-Prolog*, because computation is executed sequentially, the whole current level computation pauses and meta-level computation is started when reflection occurs. In concurrent language, however, the whole computation does not have to be influenced and it depends on what components of states are changed by reflection. This brings us the critical problem of the scope of reflection. One resolution for the problem is an introduction of a notion of *process* in Rena and managing the scope of reflection by means of that. However, this is just an immature idea and a further investigation is required. We are going to present it later in other opportunity.

5 Conclusion

We proposed reflective logic language R-Prolog*, formalized its semantics and proved soundness and completeness of its operational semantics with respect to the declarative one. Based on these fundamental results, we believe we can discuss the formal properties of behaviors of programs with reflective operations. Furthermore, we introduced the idea of concurrent reflective logic language Rena. More detailed investigations on semantics of Rena is required. Other important works to be occupied with are listed as follows.

In R-Prolog* programming, 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. We are very interested in making behavioral characterization of some syntactic classes.

Rena is a new-born language and we must make more detailed investigation on its semantics. There are lots of things to make clear to understand the reflection in concurrent language. We are also trying to make more illustrative applications for reflective computation in sequential and concurrent logic language.

Acknowledgements

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

References

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

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

[3] P. M. Hill and J. W. Lloyd. Analysis of meta-programs. In *Proceedings of the Workshop on Meta-Programming in Logic Programming (META 88)*, pp. 27–42, 1988.

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

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

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

[7] V. S. Subrahmanian. Foundations of metalogic programming. In *Proceedings of the Workshop on Meta-Programming in Logic Programming (META 88)*, pp. 53–66, 1988.

[8] H. Sugano. A formalization of reflection in logic programming. Technical Report No.98, IIAS-SIS, FUJITSU LIMITED, 1989.

[9] K. Ueda. Guarded Horn Clauses. Technical Report TR-103, ICOT, 1985.

[10] Richard W. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence*, Vol. 13, pp. 133–170, 1980.