*Regular Paper*

# Abstract Machine Approach to Operational Semantics of Prolog

TETSUO IDA*, ATSUSHI NAKAMURA**, TARO SUZUKI** and KOJI NAKAGAWA***

Operational semantics of Prolog based on abstract Prolog machines is studied. We start from the SLD-resolution calculus and a simple abstract Prolog machine to realize the SLD-resolution calculus. We then systematically refine the abstract Prolog machine in three stages; by (1) introduction of continuation, (2) realization of substitutions on a stack, and (3) speed-up of a backtrack. The result of the refinement is the abstract machine that is amenable to the procedural realization of Prolog. From the refined abstract machine it is straightforward to develop an algorithm which transforms a Prolog program to an equivalent Scheme program. The Scheme program hides certain details of the computation mechanism of Prolog, but clearly exhibits essential control and data structure for Prolog. Our study has another positive result in that Prolog programs can be run in Scheme environment.

## 1. Introduction

One of the objectives of the operational semantics of a programming language is to prescribe its implementation. Detailed description of operational semantics will help to implement language systems. In this respect a *runnable* operational semantics is most desirable. In this paper we are concerned with this aspect of the operational semantics of Prolog.

Too much detailed description, however, has the danger of committing the specification of a language to one particular implementation, and moreover, it sometimes hinders our abstract reasoning about the semantics of the language. A declarative language like Prolog enables us, by virtue of being mathematically well-founded, to manipulate and reason about programs of the language in a formal way within the logic and/or a metalogical system associated with the language. Hence, we can avoid 'detailed nonsense' by a formal treatment of underlying computation models. We exemplify this by systematically refining abstract Prolog machines.

We take the approach of viewing computation of Prolog programs as a rewrite process of atoms. We give several layers of operational semantics, starting with the SLD-resolution calculus, and concluding at the generation of Scheme programs [9]. In our view Scheme programs can describe the underlying computing mechanism of Prolog sufficiently clearly, and furthermore they are runnable on Scheme systems. Thus, the purpose of this paper is twofold. One is to give operational semantics of Prolog, which specifies the implementation of Prolog systems. The other is to generate, from a Prolog program, an equivalent Scheme program based on the operational semantics.

The operational semantics of Prolog have been given in several ways. WAM is an example which gives the semantics of Prolog in terms of an abstract Prolog machine [12]. Deransart and Ferrand give an operational formal definition of Prolog [5]. Debray and Mishra discuss the relationship between denotational and operational semantics of Prolog [4]. Baeton et al. attempt to give semantics of Prolog via a priority-ordered term rewriting system [1]. We view computation of Prolog programs as rewriting of states of abstract Prolog machines, instead of trying to understand Prolog via term rewriting systems. To view Prolog programs as a term rewriting system has an intrinsic difficulty in treating variables that only appear in the body of program clauses.

This paper is organized as follows. In section 2 we introduce a simple abstract Prolog machine based on the SLD-resolution calculus. In section 3 the abstract Prolog machine is refined in a formal way. We obtain a refined abstract Prolog machine with continuation. In section 4 a method to generate a Scheme program from a Prolog program is presented.

Research supported in part by Grants-in-Aid for Scientific Research No. 03680022 and No. 03235201, the Ministry of Education, Science and Culture.

*Institute of Information Sciences and Electronics, University of Tsukuba.
**Doctoral program in Engineering, University of Tsukuba.
***College of Information Sciences, University of Tsukuba.

## 2. An Abstract Prolog Machine

The operational semantics of Prolog is usually given via the SLD-resolution calculus defined by the following inference rule

$$\frac{\{\neg p\}\cup G,\ \{q\}\cup Q}{\theta(G\cup Q)}\ \theta p\equiv\theta q$$

where $G$ and $Q$ are (possibly empty) sets of negative atoms[1], and $p$ and $q$ are atoms that are unifiable by an mgu (most general unifier) $\theta$. The expression $\{\neg p\}\cup G$ represents a goal clause and $\{q\}\cup Q$ an input clause. The calculus captures the behavior of a Prolog interpreter in the sense that the computed answer substitutions which the Prolog interpreter delivers are included with those obtained by the SLD-resolution calculus. In other words, a Prolog interpreter is a sound implementation of the SLD-resolution calculus. The calculus itself does not specify deterministically which input clauses are to be selected, nor which atoms in the goal are selected for the resolution.

The process of the computation of the SLD-resolution calculus is seen by rewriting a set of negative atoms, i.e. using a relation $\overset{\theta}{\rightarrow}_{\alpha}$ between sets of negative atoms

$$\{\neg p\}\cup G \overset{\theta}{\rightarrow}_{\alpha} \theta(G\cup Q)$$

where $\alpha\equiv\{q\}\cup Q$. This view is the starting point of the operational semantics of Prolog based on abstract machines.

We define the operational semantics of Prolog by an abstract Prolog machine which replaces the so-called do-not-know non-determinism inherent in the SLD-resolution calculus. The abstract Prolog machine operates on the sequences of atoms, called *goals*. Since manipulation of a set is an expensive operation in computing, and furthermore it involves, in general, indeterminacy, we replace a set of negative atoms by a sequence of atoms. This entails that the order of atoms in a sequence is significant in the following discussion. The resulting abstract Prolog machine (to be abbreviated APM) is an extension of Colmerauer's abstract Prolog machine [3], to a determinate one.

A Prolog program consists of a goal clause and one or more program clauses. We regard program clauses as rewrite rules, and a goal clause as a sequence of atoms which is rewritten by the application of rewrite rules. For the clarity, we write $p\rhd q_1 \ldots q_n$, $n\geq 0$ for a program clause $p{:}-q_1, \ldots, q_n$. An empty sequence is denoted by $\varepsilon$; thus $p\rhd\varepsilon$ corresponds to program clause $p$. Hereafter we call a sequence of atoms simply an *atom sequence*. Relation $\overset{\theta}{\rightarrow}_R$, where $\theta$ is an mgu as before and $R$ is a rewrite rule, is redefined as a relation between goals. Apart from this notational difference, we use the standard terminology of logic programming as expounded in [7].

We first give several notions needed to describe APM. Rewrite rules, which have the same head symbol, i.e. the predicate symbol in the head (left-hand side) of rewrite rules, constitute a procedure for that predicate. Rewrite rules of the same head symbol are priority-ordered according to the order in which they appear in the program text. We model this as a partially ordered set of rewrite rules defined as follows.

**Definition 1**

Given a Prolog program, let $\mathbf{R}$ be a set of rewrite rules associated with the Prolog program.

(i) Quasi-order $\sqsubset$ on $\mathbf{R}$ is defined as follows; $\forall R_1, R_2\in\mathbf{R}$ $R_2\sqsubset R_1$ iff $R_1$ and $R_2$ have the same head symbol and the program clause corresponding to $R_1$ appears textually before the program clause corresponding to $R_2$ in the Prolog program.

(ii) Partial order $\sqsubseteq$ on $\mathbf{R}$ is a union of relations $\equiv$ and $\sqsubset$, where $\equiv$ is the syntactic equality modulo renaming of variables.

For a technical reason that will become apparent shortly, we assume that the least element $\perp$ and the greatest element $\top$ are included in $\mathbf{R}$. $\perp$ conceptually designates the last rewrite rule, and $\top$ the first rewrite rule. $(\mathbf{R}, \sqsubseteq)$ is thus a complete lattice.

A selection of a rewrite rule during the rewriting of goals according to the predefined priority necessitates a backtrack for securing higher chances of successful rewriting into $\varepsilon$ (which means successful refutation in resolution terminology, as we will see later). Suppose $Q\overset{\theta}{\rightarrow}_R Q'$, and a backtrack from $Q'$ to $Q$ occurs. To realize the backtrack, and to enable appropriate choice of an alternative rewrite rule, we essentially need to save $Q$ and $R$ at each rewrite. Hence, we have the following definition of a trail.

**Definition 2**

(i) A *rewrite record* is a pair of a goal and a rewrite rule.

(ii) A *trail* is a sequence of rewrite records.

$[Q, R]$ represents a rewrite record consisting of a goal $Q$ and a rewrite rule $R$. Hereafter, we use $P$ and $Q$ to denote an atom sequence, $R$ to denote a rewrite rule, and $p$ and $q$ to denote an atom. They may be subscripted. $qQ$ designates a non-empty sequence.

With trail $T$, a state of APM is defined as a triple $(Q, R, T)$. In the sequel we use $\mathbf{S}$ to denote a set of states. An empty sequence of rewrite records is also denoted by $\varepsilon$. A possible confusion between $\varepsilon$ of an atom sequence and of a rewrite record sequence should be avoided from the context. The first component $Q$ of the state is a goal to be rewritten. The second component $R$, the candidate rewrite rule, is the least upper bound of a subset of $R$ that consists of the rewrite rules potentially applicable to the rewrite of the leftmost atom of $Q$.

The candidate rewrite rule $R$ is used in the following way. Let $q$ be the leftmost atom of $Q$. Following function *Sel*: $\mathbf{R}\times\mathscr{A}\rightarrow\mathbf{R}$, where $\mathscr{A}$ is a set of atoms, selects a

[1]We call negated atoms *negative atoms* in this paper.

rewrite rule which is less than or equal to ($\sqsubseteq$) $R$, and whose head unifies with $q$:

$$Sel(R, q) = \bigsqcup \{X \in \mathbf{R} \mid X \sqsubseteq R \text{ and } \theta(\mathrm{hd}(X)) \equiv \theta q \text{ for some mgu } \theta\}$$

where $\mathrm{hd}(X)$ is the head of a rewrite rule $X$.

Note that when no rewrite rule $X$ is found in the above, $Sel(R, q) = \bot$.

In conjunction with $Sel(R, q)$ we use the following function *next*: $\mathbf{R} \to \mathbf{R}$.

$$next(R) = \begin{cases} R' & \text{if } \exists R'(R' \sqsubset R \text{ and } \forall X \sqsubset R, X \sqsubseteq R') \\ \bot & \text{otherwise} \end{cases}$$

Now we give a formal definition of APM.

**Definition 3**

An abstract Prolog machine $\mathcal{M}$ is defined as a triple $(\mathbf{R}, s_0, \sigma)$, where $\mathbf{R}$, $s_0$ and $\sigma$ are as follows.

- $\mathbf{R}$ is a complete lattice of rewrite rules.
- $s_0 \equiv (Q_0, \top, \varepsilon)$ is an initial state, where $Q_0$ is an initial goal.
- $\sigma$: $\mathbf{S} \to \mathbf{S}$ is a state transition function defined below:

$$\sigma[\![(\varepsilon, R, T)]\!] = (\varepsilon, R, T) \qquad (1)$$

$$\sigma[\![(qQ, R, T)]\!] = (\theta(PQ), \top, T[qQ, R']) \quad \text{if } Sel(R, q) = R'(\equiv p \triangleright P) \qquad (2)$$

where $\theta$ is an mgu such that $\theta q \equiv \theta p$

$$\sigma[\![(qQ, R, T[Q', R'])]\!] = (Q', next(R'), T) \quad \text{if } Sel(R, q) = \bot \qquad (3)$$

$$\sigma[\![(qQ, R, \varepsilon)]\!] = (qQ, R, \varepsilon) \quad \text{if } Sel(R, q) = \bot \qquad (4)$$

We assume that whenever a rewrite rule is used for the state transition, namely in equation (2), a new variant of a rewrite rule is always taken, so that no variable name conflict occurs between those of goals and of a rewrite rule.

It is easy to see that $\sigma$: $\mathbf{S} \to \mathbf{S}$ defined by equations (1) ~ (4) is indeed a function. Hence APM is a determinate machine.

We write $(Q, R, T) \rightharpoonup (Q', \top, T')$ when equation (2) is used for the state transition from $(Q, R, T)$ to $(Q', \top, T')$, and $(qQ, R, T[Q', R']) \hookrightarrow (Q', R', T)$ when equation (3) is used. The latter state transition is a backtrack. We let $\rightsquigarrow = \rightharpoonup \cup \hookrightarrow$ and let $\rightsquigarrow^*$ be a reflexive and transitive closure of $\rightsquigarrow$. Note that $\rightsquigarrow^*$ contains the identity relations defined in equations (1) and (4).

APM $\mathcal{M}$ is said to halt if its state is a fixed point of $\sigma$. The fixed points of $\sigma$ are $(\varepsilon, R, T)$ and $(qQ, R, \varepsilon)$ such that $Sel(R, q) = \bot$, as is easily seen from equations (1) and (4). The fixed points are also called final states. We call a state transition path leading to the fixed points *trace* (*generated by* $\rightsquigarrow$) and the number of $\rightsquigarrow$ in the trace *length* of the trace. When $\mathcal{M}$ halts we have either of the following two traces:

(i) $(Q_0, \top, \varepsilon) \rightsquigarrow^* (\varepsilon, R_n, T_n)$, or

(ii) $(Q_0, \top, \varepsilon) \rightsquigarrow^* (qQ, R_n, \varepsilon)$

In case (i) the rewriting of $Q_0$ is said to be *successful*, and in case (ii) the rewriting of $Q_0$ *unsuccessful*.

**Example 1**

$$\text{Let } \mathbf{R} = \{ \top,\ R^1\!: p \triangleright a_1a_2,\ R^2\!: a_1 \triangleright b,\ R^3\!: a_1 \triangleright \varepsilon,\ R^4\!: a_2 \triangleright \varepsilon,\ \bot \}$$

where $p$, $a_1$, $a_2$ and $b$ are ground atoms.

$$\begin{aligned}(p, \top, \varepsilon) &\rightharpoonup (a_1a_2, \top, [p, R^1]) \\ &\rightharpoonup (ba_2, \top, [p, R^1][a_1a_2, R^2]) \\ &\hookrightarrow (a_1a_2, R^3, [p, R^1]) \\ &\rightharpoonup (a_2, \top, [p, R^1][a_1a_2, R^3]) \\ &\rightharpoonup (\varepsilon, \top, [p, R^1][a_1a_2, R^3][a_2, R^4])\end{aligned}$$

The following proposition links APM and the SLD-resolution calculus.

**Proposition 1** *Suppose*

$$(Q_0, R_0, T_0) \rightsquigarrow^* (\varepsilon, R_n, T_n), \qquad (5)$$

*where*

$$T_n \equiv T_0[Q_0, R_{i_0}][Q_{i_1}, R_{i_1}] \ldots [Q_{i_k}, R_{i_k}], 0 < i_0 < i_1 < \ldots < i_k < n.$$

*Then,* $Q_0, Q_{i_1}, \ldots, Q_{i_k}, \varepsilon$ *is a refutation derivation of* $Q_0$. *Here goals* $Q_j$, $j \in \{0, i_1, \ldots, i_k\}$ *are interpreted as a disjunction of negative atoms, and* $\varepsilon$ *as an empty clause.*

The proposition can be proved easily by the induction on the number of backtracks in the trace (5).

In particular, when $R_0 \equiv \top$ and $T_0 \equiv \varepsilon$, the proposition assures the soundness of APM with respect to the SLD-resolution calculus. Therefore in the case of refutation, we can extract from the trail the sequence of rewrites

$$Q_0 \to Q_{i_1} \to \cdots \to Q_{i_k} \to \varepsilon,$$

which is what we want from the computation on APM.

## 3. Refinement of the Abstract Prolog Machine

APM is transformed to a machine which represents more faithfully abstract Prolog machines used in the implementation of Prolog systems. We are going to refine the APM in the following ways; (i) introduction of continuation, (ii) machine-oriented representation of substitutions, and (iii) speed-up of a backtrack.

### 3.1 Segment of Atom Sequences and Continuation

The first idea to this end is to partition an atom sequence into subsequences, called *segments*, of atoms. For example, an atom sequence $q_0q_1 \ldots q_n$ may be represented as

$$[q_{i_0} \ldots q_{i_1}][q_{i_1+1} \ldots q_{i_2}] \ldots [q_{i_{k-1}+1} \ldots q_{i_k}]$$

where $q_{i_0} \equiv q_0$ and $q_{i_k} \equiv q_n$. The subsequences enclosed by [and] are segments of the atom sequence. Atoms within the same segment come from the same body (i.e. the variant of the right-hand side) of a rewrite rule. Suppose we have an atom sequence

$$[qQ_0][Q_1] \ldots [Q_k]$$

which is rewritten by $p \rhd P$ with an mgu $\theta$, we have the following rewrite.

$$[qQ_0][Q_1] \ldots [Q_k] \to \theta([P][Q_0][Q_1] \ldots [Q_k])$$

The reason for the segmented representation of atom sequences is that atoms in a body are treated as a whole in the implementation of Prolog systems rather than concatenated in front of the remaining goal. Logically, a segmented atom sequence and corresponding non-segmented one have the same meaning.

Suppose we have an ordered subset of rewrite rules $\mathbf{X} = \{p_1 \rhd P_1, \ldots, p_n \rhd P_n\}$ where each rewrite rule has the same head symbol, say p. According to the usual procedural interpretation of Prolog, **X** is compiled into the procedure p and the atoms in the bodies are considered as procedure calls.

Thus, a rewrite sequence

$$[qQ_0][Q_1] \ldots [Q_k] \to \theta([P][Q_0] \ldots [Q_k])$$
$$\to^{*} \theta'([Q_0] \ldots [Q_k])$$

is interpreted procedurally as follows:

1. $q$ is a procedure call to p.
2. $P$ is the body of procedure p. All the atoms in $P$ are then processed.
3. Next, $Q_0$ is processed.

For a call to procedure p from $[qQ_0]$, the segments $[Q_0] \ldots [Q_k]$ are called the *continuation* of the call of p, since when $[P]$ has been processed, the rewrite continues on $[Q_0] \ldots [Q_k]$.

During rewriting, the focus of attention is always on the first segment of the atom sequence. Therefore, the atom sequence can be split into two parts; one is the first segment and the other is the continuation.

## 3.2 Manipulation of Substitution

The second idea is to separate mgus from goals. Applying an mgu to a goal in order to create a new goal is an expensive operation in Prolog systems. Therefore a pair consisting of the mgu and the goal is used instead. When we write $\theta(PQ)$, for example, $\theta(PQ)$ is not meant to be a result of the application of $\theta$ to $PQ$, but a pair $(\theta, PQ)$ hereafter.

Since notations concerning substitutions differ slightly among authors, we summarize here our notations used in the sequel:

- $\mathscr{V}(\alpha)$, where $\alpha$ is any syntactic object, is the set of variables in $\alpha$.
- A substitution $\theta$ is represented as a set $\{t_1/x_1, \ldots, t_n/x_n\}$, where $t_i/x_i$, $i=1, \ldots, n$ is a binding of a variable $x_i$ to a term $t_i$. By definition, $x_i \not\equiv t_i$, $i=1, \ldots, n$. For this $\theta$, its domain $\mathscr{D}(\theta)$ and codomain $\mathscr{CD}(\theta)$ are defined as follows:

$$\mathscr{D}(\theta) = \{x_1, \ldots, x_n\}$$
$$\mathscr{CD}(\theta) = \cup_i \mathscr{V}(t_i)$$

- $\theta \mid V$ is a substitution whose domain is restricted to $V$, i.e.,

$$\theta \mid V(x) = \begin{cases} \theta(x) & x \in V \\ x & \text{otherwise} \end{cases}$$

- The composition of $\theta_1$ with $\theta_2$, i.e. $\theta_2(\theta_1(x))$, is written simply as $\theta_2\theta_1(x)$. $\theta_2\theta_1$ in the set representation is

$$(\theta_2 \cup \{\theta_2 s/x \mid s/x \in \theta_1\}) - \{\theta_2 s/x \mid \theta_2 s \equiv x,\ s/x \in \theta_1\}$$
$$- \{s/x \in \theta_2 \mid x \in \mathscr{D}(\theta_1)\}.$$

Suppose we have a refutation:

$$Q_0 \to^{\theta_1}_{R_1} \mu_1 Q_1 \to^{\theta_2}_{R_2} \cdots \to^{\theta_n}_{R_n} \mu_n \varepsilon \qquad (6)$$

where $\mu_i = \theta_i \mu_{i-1}$, $i=1, \ldots, n$ and $\mu_0 = \phi$.

At the $i$-th step

$$\mu_{i-1}(qQ'_{i-1})(\equiv \mu_{i-1}Q_{i-1}) \to^{\theta_i}_{R_i} \mu_i Q_i,$$

we have $\theta_i \mu_{i-1} q \equiv \theta_i p_i$ where $R_i \equiv p_i \rhd P_i$, and $Q_i \equiv P_i Q'_{i-1}$ since, because of $\mathscr{D}(\mu_{i-1}) \cap \mathscr{V}(R_i) = \phi$, the following holds:

$$(\theta_i P_i)(\theta_i \mu_{i-1} Q'_{i-1}) = (\theta_i \mu_{i-1} P_i)(\theta_i \mu_{i-1} Q'_{i-1})$$
$$= \mu_i (P_i Q'_{i-1}).$$

Therefore atoms in $Q_i$, $i=1, \ldots, n-1$, are either atoms in $Q_0$ or variants of atoms of the right-hand side of rewrite rules $R_i$, $i=1, \ldots, n-1$. It is these $\theta_i$'s and $\mu_i$'s that are computed during the refutation.

In the standard implementation of Prolog systems, the unification algorithm is designed to generate mgus that have the following properties:

(i) For any variable renaming binding $y/x$,

$$y/x \in \theta_i \Rightarrow y \in \mathscr{V}(\mu_{i-1}Q_{i-1}).$$

(ii) Mgus are idempotent, i.e. for any mgu $\theta_i$,

$$\mathscr{D}(\theta_i) \cap \mathscr{CD}(\theta_i) = \phi.$$

We also note that

$$\mathscr{V}(R_i) \cap \mathscr{V}(R_j) = \phi,\ i \neq j,\ i, j \in \{1, \ldots, n\}$$

since $R_i$s are new variants of rewrite rules.

Then, we have the following proposition concerning the substitutions that are generated in the refutation (6).

**Proposition 2** *Let*

$$Q_0 \overset{\theta_1}{\to}_{R_1} \mu_1 Q_1 \overset{\theta_2}{\to}_{R_2} \cdots \overset{\theta_n}{\to}_{R_n} \mu_n \varepsilon \qquad (7)$$

*be a refutation, where* $\mu_i = \theta_i \mu_{i-1}$, $i=1, \ldots, n$ *and* $\mu_0 = \phi$ and $\theta_i$, $i=1, \ldots, n$, *satisfy properties* (i) *and* (ii).

*Then, for* $i=1, \ldots, n$, *we have*

(a) $\mathscr{D}(\theta_i)\cap\mathscr{D}(\mu_{i-1})=\phi$,

(b) $\mu_i=\theta_i\cup\{\theta_i t/x \mid t/x\in\mu_{i-1}\}$, (8)

(c) $\mu_i$ *is idempotent.*

The proof is given in the appendix.

Let $\mu_{i-1}=\{t_j/x_j \mid j\in J_{i-1}\}$ and $\Delta\theta_i=\theta_i \mid \bigcup_{j\in J_{i-1}}\mathscr{V}(t_j)$, and we write

$$\Delta\theta_i\cdot\mu_{i-1}\equiv\{\Delta\theta_i t_j/x_j \mid j\in J_{i-1}\}.$$

Then, from equations (8), we have

$$\mu_i=\theta_i\cup\Delta\theta_i\cdot\mu_{i-1}. \qquad (9)$$

With this preparation in mind, let us consider the case of a backtrack.

$$(\mu_{k-1}(qQ),\ \top,\ T)\rightharpoonup(\theta_k\mu_{k-1}(PQ),\ \top,\ T[\mu_{k-1}(qQ),\ R])$$
$$\hookrightarrow(\mu_{k-1}(qQ),\ next(R),\ T)$$

What is involved here is the recovery of $\mu_{k-1}$ from $\mu_k(=\theta_k\mu_{k-1})$. To recover $\mu_{k-1}$ from $\mu_k$ we use equation (9). Namely, subtract $\theta_k$ from $\mu_k$ and replace the subterms introduced by $\Delta\theta_k$ by the original variables. The information needed for the recovery of $\mu_{k-1}$ from $\Delta\theta_k\cdot\mu_{k-1}$ is denoted by $\Delta\theta_k^{-1}$. Discussion on how to represent $\Delta\theta_k^{-1}$ is postponed until section 4. We call $\Delta\theta_k^{-1}$ a *variable trail*. The word *trail* was originally used to refer to a stack in order to keep track of the addresses of certain variables in the goal that are instantiated by unification. It corresponds to the variable trails in our case.

## 3.3 Abstract Prolog Machine with Continuation

We are now ready to give a new machine to be called APM-C (Abstract Prolog Machine with Continuation) $\mathscr{M}_C$. A state of $\mathscr{M}_C$ is a quadruple $(Q, R, T, C)$, where $Q$, $R$, $T$ and $C$ are a goal, an applicable rule, a trail, and a continuation respectively.

**Definition 4** An abstract Prolog machine with continuation, $\mathscr{M}_C$, is defined as triple $(\mathbf{R}, s_0, \sigma_C)$, where $\mathbf{R}$, $s_0$ and $\sigma_C$ are as follows.

- $\mathbf{R}$ is a complete lattice of rewrite rules as in definition 3.
- $s_0\equiv(Q_0,\ \top,\ \varepsilon,\ \varepsilon)$ is an initial state, where $Q_0$ is an initial goal.
- $\sigma_C$: $\mathbf{S}\rightharpoonup\mathbf{S}$ is a state transition function defined below:

$$\sigma_C[\![(\varepsilon, R, T, \varepsilon)]\!]=(\varepsilon, R, T, \varepsilon) \qquad (10)$$

$$\sigma_C[\![(qQ, R, T, C)]\!]=(\theta P,\ \top,\ T[\theta(C[qQ]),\ next(R'),\ \Delta\theta^{-1}],\ \theta(C[Q])) \qquad (11)$$

if $Sel(R, q)=R'(\equiv p\rhd P)$

where $\theta$ is an mgu such that $\theta q\equiv\theta p$

$$\sigma_C[\![(\varepsilon, R, T, C[Q])]\!]=(Q, R, T, C) \qquad (12)$$

$$\sigma_C[\![(qQ, R, T[\theta(C'[Q']), R', \Delta\theta^{-1}], C)]\!]$$
$$=(Q', R', T, C') \qquad \text{if } Sel(R, q)=\bot \qquad (13)$$

$$\sigma_C[\![(qQ, R, \varepsilon, \varepsilon)]\!]=(qQ, R, \varepsilon, \varepsilon)$$
$$\text{if } Sel(R, q)=\bot \qquad (14)$$

In equation (13), $\Delta\theta^{-1}$ is used to recover $C'$ and $Q'$ from $\theta(C'[Q'])$, although its operation is implicit in the state transition. As in APM, we use $\hookrightarrow$ for the state transition of a backtrack defined by equation (13). In the case of the state transitions defined by equations (11) and (12), we use $\rightharpoonup_1$ and $\rightharpoonup_2$, respectively. Note again that $\sigma_C$ is a function.

**Example 2**

We use the same set of rewrite rules $\mathbf{R}$ as in Example 1.

$$(p,\ \top,\ \varepsilon,\ \varepsilon)\rightharpoonup_1(a_1a_2,\ \top,\ [[p],\ \bot,\ \phi],\ \varepsilon)$$
$$\rightharpoonup_1(b,\ \top,\ t_1[[a_1a_2],\ R^3,\ \phi],\ [a_2])$$
$$\text{where } t_1\equiv[[p],\ \bot,\ \phi]$$
$$\hookrightarrow(a_1a_2,\ R^3,\ t_1,\ \varepsilon)$$
$$\rightharpoonup_1(\varepsilon,\ \top,\ t_1[[a_1a_2],\ \bot,\ \phi],\ [a_2])$$
$$\rightharpoonup_2(a_2,\ \top,\ t_1t_2,\ \varepsilon)$$
$$\text{where } t_2\equiv[[a_1a_2],\ \bot,\ \phi]$$
$$\rightharpoonup_1(\varepsilon,\ \top,\ t_1t_2[[a_2],\ \bot,\ \phi],\ \varepsilon)$$

Here, $\phi$(of $\Delta\theta^{-1}$) is an empty substitution.

Note that the state transition generated by $\rightharpoonup_2$ is an additional transition compared with the case of APM. It roughly corresponds to procedure return.

We call $\mathscr{M}_C$ and $\mathscr{M}$ *equivalent* if both generate the same refutation derivation. That $\mathscr{M}_C$ and $\mathscr{M}$ with the same initial goal are equivalent can easily be seen by the following reasoning:

1. Let $\rightharpoonup=(\rightharpoonup_2\circ\rightharpoonup_1)\cup\rightharpoonup_1$, where we regard the transitions as relations defining sets $\subseteq\mathbf{S}\times\mathbf{S}$. Our intention here is to combine the transitions of $\rightharpoonup_1$ immediately followed by $\rightharpoonup_2$ into a single step of a transition. Let $\rightsquigarrow=\rightharpoonup\cup\hookrightarrow$ as in APM.
2. We compare traces of $\mathscr{M}$ and $\mathscr{M}_C$ generated by $\rightsquigarrow$.
3. We observe that the lengths of the traces are the same, and that for state $s_i\equiv(Q, R, T, C)$ of $\mathscr{M}_C$ and state $s_i'\equiv(Q', R', T')$ of $\mathscr{M}$ of the corresponding state in the trace, $Q(rev(C))\equiv Q'$ holds, where $rev([Q_1][Q_2]\ldots[Q_n])=Q_n\ldots Q_2Q_1$.

## 3.4 Repeated Backtrack

At this point, we summarize the planned changes from APM.

- Continuation is added to the state.
- The trail record consists of three items: previous goal, next applicable rule (instead of applied) and variable trail. The next applicable rule is computed

and stored in the trail, since the next applicable rule has to be computed, after all, from the applied rule at the time of a backtrack.

- The variable trail is added to recover a substitution at the time of a backtrack.

One more refinement is necessary in order to make the behavior of APM-C closer to that of Scheme programs generated from Prolog programs. In a standard implementation of Prolog systems repeated backtracks

$s_i \hookrightarrow s_{i+1} \hookrightarrow \ldots \hookrightarrow s_j$

where

$s_k=(Q_k, R_k, T_k[Q'_k, \perp, \Delta\theta_k^{-1}], C_k),\ k=i, \ldots, j-2$

and $s_j=(Q_j, R_j, T_j[Q'_j, R'_j, \Delta\theta_j^{-1}], C_j)$, where $R_j \neq \perp$

can be realized as a single backtrack. This can be formalized by modifying the definition $\sigma_C$ of equation (13). Namely, the new definition of equation (13) is

$$\begin{aligned}&\sigma_C\llbracket(qQ, R, T[\theta(C'[Q']), R', \Delta\theta^{-1}], C)\rrbracket\\ &\quad = \mathit{repeat\_backtrack}\llbracket(Q', R', T, C')\rrbracket\\ &\qquad\qquad\qquad \text{if } \mathit{Sel}(R, q)=\perp \qquad (15)\end{aligned}$$

where

$$\begin{aligned}&\mathit{repeat\_backtrack}\llbracket(qQ, \perp, T[\theta(C'[Q']), R', \Delta\theta^{-1}], C)\rrbracket\\ &\quad = \mathit{repeat\_backtrack}\llbracket(Q', R', T, C')\rrbracket\\ &\mathit{repeat\_backtrack}\llbracket(Q, R, T, C)\rrbracket = (Q, R, T, C)\\ &\qquad\qquad\qquad \text{otherwise.}\end{aligned}$$

A state transition $s_i$ to $s_j$ using equation (15) is denoted by $s_i \looparrowright s_j$.

## 4. Translation of Prolog to Scheme

### 4.1 Transformation to a Homogeneous Form

APM-C makes explicit the computing resources necessary to execute Prolog programs, i.e. trail and continuation. We further need to refine APM-C in order to map those resources to actual resources of computers. The refinement in this section is based on program transformations.

We first introduce a homogeneous form of Prolog programs [10]. A homogeneous form of a Prolog program (in a rewrite rule form) $\mathrm{p}(t_1, \ldots, t_n) \rhd P$ is

$$\mathrm{p}(x_1, \ldots, x_n) \rhd \mathrm{unify}(x_1, t_1) \cdots \mathrm{unify}(x_n, t_n) P$$

where $x_1, \ldots, x_n$ are distinct fresh variables. Predicate unify($s$, $t$) is a special predicate which unifies $s$ and $t$ if unifiable. We transform all rewrite rules in $\mathbf{R}$ to their homogeneous forms.

The reason for adopting homogeneous rewrite rules is that the unification of parameters of the heads is dealt with in the body of the rewrite rules. Then with the set of homogeneous rewrite rules, the unification between the head of a rewrite rule and an atom of a goal is successful iff the predicate symbol of the atom and the head symbol of the rewrite rule is the same.

### 4.2 Revised State Transition Function

Homogeneous rewrite rules enable us to simplify APM-C further. Let us give an example to see how APM-C can be simplified. Suppose that $\mathbf{R}$ is a set of homogeneous rewrite rules, and that we have a following rewrite sequence:

$$\begin{aligned}&(qQ, R, T, C)\\ &\rightharpoonup_1(\theta_1 P_1, \top, T[\theta_1(C[qQ]), \mathit{next}(R_1), \Delta\theta_1^{-1}], \theta_1(C[Q]))\\ &\quad(\text{where } R_1=\mathit{Sel}(R, q),\ R_1 \equiv p_1 \rhd P_1 \in \mathbf{R} \text{ and } \theta_1 q \equiv \theta_1 p_1)\\ &\leadsto^*(Q', R', T', C')\\ &\looparrowright (qQ, R_2, T, C) \qquad\qquad (\text{where } R_2=\mathit{next}(R_1))\\ &\rightharpoonup_1(\theta_2 P_2, \top, T[\theta_2(C[qQ]), \mathit{next}(R_2), \Delta\theta_2^{-1}], \theta_2(C[Q]))\\ &\qquad\qquad(\text{where } R_2 \equiv p_2 \rhd P_2 \in \mathbf{R} \text{ and } \theta_2 q \equiv \theta_2 p_2)\end{aligned}$$

Note that $R_2 \not\equiv \perp$ by the definition of $\looparrowright$. When APM-C backtracks from state $(Q', R', T', C')$ to state $(qQ, R_2, T, C)$, there is actually no need to compute $\mathit{Sel}(R_2, q)$ since $q$ always unifies with the head $p_2$ of $R_2$ because of the homogeneity of $R_2$. The state $(qQ, R_2, T, C)$ can be skipped in the rewrite sequence. This can be realized formally by combining $\looparrowright$ and $\rightharpoonup_1$ to form the composition $\rightharpoonup_1 \circ \looparrowright$. Let $\looparrowright' = \rightharpoonup_1 \circ \looparrowright$, and we call $\looparrowright'$ *quick backtrack*.

More generally, for a given rewrite sequence $s_1 \leadsto' s_2 \leadsto' \cdots \leadsto' s_n$ where $\leadsto'$ denote either $\rightharpoonup_1 \rightharpoonup_2$ or $\looparrowright$, we eliminate all the $s_i$'s that satisfy $s_{i-1} \looparrowright s_i \rightharpoonup_1 s_{i+1}$, and write $s_{i-1} \looparrowright' s_{i+1}$ instead. It is easily seen that in the thus obtained rewriting sequence the second component of all the states is $\top$. This suggests that we can obviate the second component of the state and simplify the state transition function $\sigma_C$.

Following is the new and final definition of the state transition function $\sigma'_C$. We let $\mathit{Sel}'(q)=\mathit{Sel}(\top, q)$.

[Halt with success]

$$\sigma'_C\llbracket(\varepsilon, T, \varepsilon)\rrbracket = (\varepsilon, T, \varepsilon) \qquad (16)$$

[Call a goal]

$$\begin{aligned}&\sigma'_C\llbracket(qQ, T, C)\rrbracket\\ &\quad = (\theta P, T[\theta(C[qQ]), \mathit{next}(R'), \Delta\theta^{-1}], \theta(C[Q])) \qquad (17)\\ &\qquad\qquad \text{if } \mathit{Sel}'(q)=R'(\equiv p \rhd P)\\ &\qquad\qquad \text{where } \theta \text{ is an mgu such that } \theta q \equiv \theta p\end{aligned}$$

[Return]

$$\sigma'_C\llbracket(\varepsilon, T, C[Q])\rrbracket = (Q, T, C) \qquad (18)$$

[Quick-backtrack]

$$\begin{aligned}&\sigma'_C\llbracket(qQ, Tt, C)\rrbracket\\ &\quad = \mathit{repeat\_backtrack}\llbracket(qQ, Tt, C)\rrbracket\\ &\text{if } \mathit{Sel}'(q)=\perp \qquad (19)\end{aligned}$$

[Halt with failure]

$$\sigma'_C[\![(qQ,\varepsilon,\varepsilon)]\!]=(qQ,\varepsilon,\varepsilon) \quad (20)$$
if $Sel'(q)=\perp$

We have to redefine *repeat_backtrack* for the revised states. The new definition of *repeat_backtrack* is

$$\mathit{repeat\_backtrack}\,[\![(qQ, T[\theta(C'[Q']), \perp, \Delta\theta^{-1}], C)]\!]$$
$$=\mathit{repeat\_backtrack}\,[\![(Q', T, C')]\!]$$
$$\mathit{repeat\_backtrack}\,[\![(qQ, T[\theta(C'[Q']), R', \Delta\theta^{-1}], C)]\!]$$
$$=(\theta P, T[\theta(C'[Q']), next(R'), \Delta\theta^{-1}], C)$$
$$\text{if } R'\equiv p\triangleright P$$
$$\mathit{repeat\_backtrack}\,[\![(Q,\varepsilon,\varepsilon)]\!]=(Q,\varepsilon,\varepsilon)$$

## 4.3 Mapping of Trail and Continuation to a Scheme Program

Realization of a Prolog system in Scheme is now at hand, although one would still need architectural intuition. We enumerate the major points for designing an abstract Prolog machine in Scheme.

- Homogeneous rewrite rules of the same head symbol is realized as a single procedure.
- Trail $T$ is realized as a stack. We call this stack *T-stack*. In WAM our *T-stack* is called a local stack.
- A rewrite record is encoded into a more efficient data structure called *frame* in *T-stack*. For each call of a procedure a frame is allocated.
- Close examination of equations (17) and (18) reveals that the creation/deletion of a rewrite record of the stack does not exactly synchronize with procedure entry/return.
- An mgu which is a component of a rewrite record is realized as a block of variables in the frame.
- A frame consists of $(cp, x_1, \ldots, x_n, bp, vt, nr; y_1, \ldots, y_m)$, where the components to the left of ';' are required and the components to the right is optional depending on each rewrite rule.
  - $x_1, \ldots, x_n$ are variables that appear in the head. We can choose variables in the heads of the same head symbol such that each variable at the same parameter position of the head is the same.
  - *cp* is a continuation.
  - *bp* is a value of the backtrack pointer BP (see below).
  - *vt* is a pointer to a variable trail. It is clear that variable trails can also be organized as a stack. We call this stack *v-trail*.
  - *nr* is the address of the code for the next rule. A tuple (*bp*, *vt*, *nr*) corresponds to a choice point of WAM.
  - $y_1, \ldots, y_m$ correspond to variables that only appear in the body of each rewrite rule. Actually they are pointers to variables allocated in the heap. The heap, however, does not appear explicitly in our discussion since we do not discuss the representation of terms.
- In order to implement a quick backtrack we introduce BP (called backtrack pointer) that points to the frame of the state to which the quick backtrack returns. BP always holds the most recent value of *nr*.

We do not have to manipulate explicitly the *T*-stack and continuations since they are provided in the Scheme system. The *T*-stack is simply the Scheme's system stack. Continuations are manipulated using Scheme's special procedure `call-with-current-continuation` (below it is abbreviated as call/cc). Equations (18) [Return] and (19) [Quick-backtrack] are realized by `call/cc` of Scheme. *cp* and *nr* denote the continuations created by the execution of `call/cc`. The continuation *cp* is evaluated at the time of procedure return (cf. (18)), and *bp* is evaluated at the time of a backtrack (cf. (19)). The *v*-trail is explicitly realized by an array. Associated with the *v*-trail, we have a top-of-stack pointer VT. VT and BP are realized as a global variable.

## 4.4 Generation of a Scheme Program

We describe below the case of a unary predicate definition. Generalizing to *n*-ary case is straightforward. Let

$$\{p(x)\triangleright P_1, \ldots, p(x)\triangleright P_n\}$$

be a set of rewrite rules from which we translate to a Scheme program.

We assume that $P_i$ consists of goal calls $p_{i,1}, \ldots, p_{i,k_i}$. The comments in the Scheme program to the defined procedure are hopefully sufficient to understand the program.

```
(define (p x)
 (call/cc
  (lambda (exit)
   (let ((oldbp BP)(oldvt VT))
    ; first rule
    (call/cc
     (lambda (nr)
      (let ((y11 (newvar))...(y1m1 (newvar)))
       ; newvar creates in the heap a variable
       ; whose initial value is 'undef'
       (set! BP nr)
       (p11...)...(p1k1...); goal calls in the
                           ; body of P1
       (exit nil))))
    ; next rule
    (pop-vtrail oldvt)
     ; reset the variables to 'undef' that have been
     ; instantiated since the goal call of the
     ; previous rule
    (call/cc
     (lambda (nr)
      (let ((y21 (newvar))...(y2m2 (newvar)))
       (set! BP nr)
       (p21...)...(p2k2...)
```

```
        (exit nil))))
      .
      .
      (pop-vtrail oldvt)
       ; The variables pushed between oldvt
       ; and VT are reset to 'undef', and
       ; VT is restored to the value of oldvt.
       ; We do not need a continuation here since this
       ; is the last rule.
      (set! BP oldbp)
       ; restore the old BP to the point
       ; that the state of APM returns
       ; when the following rewrites fail.
      (let ((ynl...)...(ynmn...))
        (pnl...)...(pnkn...))))))
```

We do not fully specify the algorithm of procedure `unify` since it is well understood (see [7], e.g.). The only point pertinent to our discussion is that the $v$-trail and backtrack are manipulated in `unify`. When a variable is instantiated the address of the variable is pushed to the $v$-trail. When unification fails, a backtrack is effected by evaluating the continuation held in BP. The fragment of the Scheme program for `unify` is as follows:

```
(define (unify x y)
 (cond ((var? y); Is y an unbound variable?
        (push-vtrail y)
        ; push the value of y in v-trail
        (set-var! y x))
        ; set the value of y to the value of x
      ..........
      ; When all attempts fail, the continuation
      ; held in global variable BP is evaluated.
      (t (BP 'fail))))
```

**Example 3**

The following Prolog program app concatenates two lists.

```
app([ ], X, X).
app([X|Y], Z, [X|W]):- app(Y, Z, W).
```

The above Prolog program is transformed to homogeneous rewrite rules.

$app(x_1, x_2, x_3) \triangleright unify([\ ], x_1)\ unify(x, x_2)\ unify(x, x_3)$

$app(x_1, x_2, x_3) \triangleright unify([x \mid y], x_1)\ unify(z, x_2)$

$\qquad unify([x \mid w], x_3)\ app(y, z, w)$

Then, the following Scheme program for app is obtained.

```
(define (app X1 X2 X3)
 (call/cc
  (lambda (exit)
   (let ((oldbp BP)(oldvt VT))
    (call/cc
     (lambda (nr)
      (let ((X (newvar)))
       (set! BP nr)
       (unify ( ) X1) (unify X X2) (unify X X3)
       (exit nil))))
    ; next rule
    (pop-vtrail oldvt)
    (set! BP oldbp)
    (let ((X (newvar))(Y (newvar))(Z (newvar))
          (W (newvar)))
     (unify (cons X Y) X1) (unify Z X2)
     (unify (cons X W) X3)
     (app Y Z W))))))
```

## 5. Concluding Remarks

We have shown how the operational semantics of Prolog is understood by giving several abstract Prolog machines. Prolog is known as a logic programming language based on the SLD-resolution, and theories of the operational semantics of Prolog abound. Good implementation of Prolog systems, such as DEC-10 Prolog [11], SICStus Prolog [2], and Quintus Prolog[1], exist. There is, however a big gap between the theories of the operational semantics of Prolog and their implementations. Our attempt is to bridge the theories of the operational semantics of Prolog and the implementations. The gap has been shown to be filled by refining abstract Prolog machines. The refined abstract machine exhibits the properties that are amenable to procedural realization in terms of Scheme programs.

The translation to Scheme programs relieves us from considering certain details of implementation such as allocation of variables and handling of code addresses, while at the same time it explicates the essence of the computation of Prolog, such as unification-driven goal calls and automatic backtracks. The outcome of our treatment of the operational semantics is desireble one from an application point of view since Prolog programs are runnable in Scheme systems. Furthermore our treatment makes it possible to combine Scheme programs and Prolog programs.

We have focussed on the control aspect of the compilation in this paper, and (to some extent) on goal rewriting by unification. Nilsson identified three major issues in the compilation of Prolog programs: i.e., (i) parameter passing by unification, (ii) term representation, and (iii) control structure, in his domain-based Prolog [8]. Later, Kursawe [6], based on the same observation, proposed a method of transforming Prolog programs to simpler ones that correspond more closely to WAM code [12].

In our view (iii) of the above points is most critical in understanding the semantics of Prolog. We do not discuss optimization and specialization (by partial evaluation) of unification. Kursawe's work can be adapted into our framework easily. The problem of term

[1]Quintus Prolog is a trademark of Quintus Computer Systems, Inc.

representation is not dealt with since terms in our implementation are represented in S-expressions unlike vector representation of WAM, and the facilities for structuring data are available for free.

'Cut' is not dealt with in this paper since it is straightfoward to extend our machines to allow for the cut operator. Moreover, by the discussion on this extension we would gain very little in understanding abstract operational semantics of (pure) Prolog.

Lastly, we would like to mention the relationship of our work with the works on compiling Prolog into other languages. Nilsson suggested a compilation of Prolog programs into Pascal [8], and Weiner and Ramakrishnan reported compilation into C [13], their works, however, do not address the correctness of compiling methods with respect to the formal operational semantics of Prolog. Our emphasis in this paper is to formalize the operational semantics of Prolog. It is beyond the scope of the present paper, albeit interesting to pursue, to address the problems of optimizations of compilation and of the feasibility of Scheme as a final abstract Prolog machine.

**References**

1. Baeten, J. C. M. and Weijland, W. P. *Semantics for Prolog via term rewrite systems*, Springer LNCS 308 (1987), 3-14.
2. Carlsson, M. and Wid'en, J. SICStus Prolog user's manual. SICS Research Report R 88007B, SICS, 1988.
3. Colmerauer, A. Equations and inequations on finite and infinite terms. In *FGCS*, ICOT (1984), 85-99.
4. Debray, S. K. and Mishra, P. Denotational and operational semantics for Prolog. In Wirsing, M. ed. *Formal Description of Programming Concepts—III, IFIP*, Elsevier Science Publishers B.V. (1987), 245-272.
5. Deransart, P. and Ferrand, G. An operational formal definition of Prolog. *Rapports de Recherche 763*, INRIA (Dec. 1987).
6. Kursawe, P. How to invent a Prolog machine. *New Generation Computing*, 5 (1987), 97-114.
7. Lloyd, J. W. *Foundations of Logic Programming*. Springer-Verlag, second, extended edition, 1987.
8. Nilsson, J. F. On the compilation of a domain-based Prolog. In Mason, R. E. A. ed. *Information Processing 83, IFIP*, Elsevier Science Publishers B.V. (1983), 293-298.
9. Rees, J. and Clinger, W. eds. Revised³ report on the algorithmic language Scheme. *SIGPLAN NOTICES*, 21(12) (1986), 37-43.
10. van Emden, M. H. and Lloyd, J. W. A logical reconstruction of Prolog II, *Journal of Logic Programming*, 1 (1984), 143-149.
11. Warren, D. H. D. *Applied Logic—It's Use and Implementations as Programming Tool*. PhD thesis, Department of Artifitial Intelligence, University of Edinburgh, 1977.
12. Warren, D. H. D. An abstract Prolog instruction set. Technical Note 309, SRI International, 1983.
13. Weiner, J. L. and Ramakrishnan, S. A piggy-back compiler for Prolog. *In Proceedings of the SICPLAN '88 Conference on Programming Language Design and Implementation* (June 1988), 288-296.

(Received September 2, 1991; revised February 19, 1992)

## A Proof of Proposition 2

**Proposition 2** *Let*

$$Q_0 \xrightarrow{\theta_1}_{R_1} \mu_1 Q_1 \xrightarrow{\theta_2}_{R_2} \cdots \xrightarrow{\theta_n}_{R_n} \mu_n \varepsilon$$

*be a refutation, where* $\mu_i = \theta_i \mu_{i-1}$, $i = 1, \ldots, n$ *and* $\mu_0 = \phi$ *and* $\theta_i$, $i = 1, \ldots, n$, *satisfy properties* (i) *and* (ii). *Then, for* $i = 1, \ldots, n$, *we have*

(a) $\mathcal{D}(\theta_i) \cap \mathcal{D}(\mu_{i-1}) = \phi$
(b) $\mu_i = \theta_i \cup \{\theta_i t/x \mid t/x \in \mu_{i-1}\}$,
(c) $\mu_i$ *is idempotent.*

Proof:
The proof is by the induction on $i$. For $i=1$, (a) ~ (c) are obviously true since $\mu_0 = \phi$, $\mu_1 = \theta_1$, and $\theta_1$ is idempotent. Suppose that (a) ~ (c) are true for $i > 0$. We have, by definition,

$$\begin{aligned}\mu_{i+1} &= \theta_{i+1}\mu_i \\ &= \theta_{i+1} \cup \{\theta_{i+1} t/x \mid t/x \in \mu_i\} - A_{i+1} - B_{i+1},\end{aligned}$$

where

$$A_{i+1} = \{\theta_{i+1} t/x \mid \theta_{i+1} t \equiv x,\ t/x \in \mu_i\}, \text{ and}$$
$$B_{i+1} = \{t/x \in \theta_{i+1} \mid x \in \mathcal{D}(\mu_i)\}.$$

We first show $\mathcal{D}(\theta_{i+1}) \cap \mathcal{D}(\mu_i) = \phi$, and then show that $B_{i+1} = \phi$ and $A_{i+1} = \phi$.

Suppose $\mathcal{D}(\theta_{i+1}) \cap \mathcal{D}(\mu_i) \neq \phi$. Let $x \in \mathcal{D}(\theta_{i+1}) \cap \mathcal{D}(\mu_i)$. Since $x \in \mathcal{D}(\theta_{i+1})$, there are two cases; $x \in \mathcal{V}(R_{i+1})$ or $x \in \mathcal{V}(\mu_i Q_i)$. In the former case, $x \notin \mathcal{D}(\mu_i)$ since $R_{i+1}$ is a new variant. In the latter case, $x \notin \mathcal{D}(\mu_i)$ since $\mu_i$ is idempotent by the induction hypothesis. Both cases lead to contradiction. Hence $\mathcal{D}(\theta_{i+1}) \cap \mathcal{D}(\mu_i) = \phi$. From this, we obtain $B_{i+1} = \phi$.

We next show $A_{i+1} = \phi$. Suppose $A_{i+1} \neq \phi$. Let $\theta_{i+1} t/x \in A_{i+1}$. Since $\theta_{i+1} t \equiv x$, $t$ is a variable, and hence $x/t \in \theta_{i+1}$. Since $t/x \in \mu_i$ and $\mu_i$ is idempotent, $x \notin \mathcal{V}(\mu_i Q_i)$. But this contradicts property (i). Hence $A_{i+1} = \phi$. Therefore, $\mu_{i+1} = \theta_{i+1} \cup \{\theta_{i+1} t/x \mid t/x \in \mu_i\}$.

What remains to be proved is that $\mu_{i+1}$ is idempotent. Let $y \in \mathcal{D}(\mu_{i+1})$. There are two cases; (1) $y \in \mathcal{D}(\theta_{i+1})$ or (2) $y \in \mathcal{D}(\{\theta_{i+1} t/x \mid t/x \in \mu_i\})$. Both cases are mutually exclusive by (a). In case (1), $y \notin \mathcal{CD}(\theta_{i+1})$ and $y \notin \mathcal{D}(\{\theta_{i+1} t/x \mid t/x \in \mu_i\})$ since $\theta_{i+1}$ is idempotent. In case (2), there exists a binding $\theta_{i+1} s/y$ such that $s/y \in \mu_i$. We prove $y \notin \mathcal{CD}(\theta_{i+1})$ and $y \notin \mathcal{CD}(\{\theta_{i+1} t/x \mid t/x \in \mu_i\})$ by contradiction.

Suppose $y \in \mathcal{CD}(\theta_{i+1})$. Then there would exist a term $c[y]$ which contains $y$, and a variable $z$ such that $c[y]/z \in \theta_{i+1}$. $y$ comes from either $\mu_i Q_i$ or $\mathrm{hd}(R_{i+1})$. $y \notin \mathcal{V}(\mu_i Q_i)$ since $y \in \mathcal{D}(\mu_i)$ and $\mu_i$ is idempotent. Hence $y \in \mathcal{V}(\mathrm{hd}(R_{i+1}))$. But this contradicts $s/y \in \mu_i$ since $R_{i+1}$ is a new vaviant. Therefore $y \notin \mathcal{CD}(\theta_{i+1})$.

Suppose $y \in \mathcal{CD}(\{\theta_{i+1} t/x \mid t/x \in \mu_i\})$. Then there would exist $t/x \in \mu_i$ and $\theta_{i+1} t/x$ such that $y \in \mathcal{V}(\theta_{i+1} t)$. This implies (2-1) $y \in \mathcal{CD}(\theta_{i+1})$ or (2-2) $y \in \mathcal{V}(t)$ and $y \notin \mathcal{D}(\theta_{i+1})$.

Case (2-1) contradicts what we have proved above. In case (2-2), $y \in \mathcal{CD}(\mu_i)$ since $y \in \mathcal{V}(t)$ and $t/x \in \mu_i$. This contradicts the fact that $s/y \in \mu_i$ and $\mu_i$ is idempotent. Therefore $y \notin \mathcal{CD}(\{\theta_{i+1} t/x \mid t/x \in \mu_i\})$. We conclude from cases (1) and (2) that $\mu_{i+1}$ is idempotent/