数理解析研究所講究録
第511巻 1984年 259-273

Foundation of Logic Programming Based on Inductive Definition

Masami Hagiya

Research Institute for Mathematical Sciences
Kyoto University
Oiwake-cho Kitashirakawa, Sakyo-ku, Kyoto 606

Takafumi Sakurai

Department of Information Science
Faculty of Science
University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113

ABSTRACT

A logical system of inference rules intended to give the foundation of logic programs is presented. The distinguished point of the approach taken here is the application of the theory of inductive definitions, which allows us to uniformly treat various kinds of induction schema and also allows us to regard *negation as failure* as a kind of induction schema. This approach corresponds to the so called least fixpoint semantics. Moreover, in our formalism, logic programs are extended so that a condition of a clause may be any first order formula. This makes it possible to write a quantified specification as a logic program. It also makes the class of induction schemata much larger to include the usual course-of-values inductions.

0. Introduction

The purpose of this paper is to present a logical system, which is intended to be the basis of verification and automatic programming of logic programs. We followed the theory of iterated inductive definitions of Martin-Löf[10], in formulating the rules.

Logic programming brings the idea that the semantics of a programming language consists of a description of a declarative (logical) meaning of a program and a control to execute it. The latter corresponds to the operational semantics and the former to a part of the denotational semantics. The advantage of logic programming is that, since a program is mapped to a formula of a logical system, its declarative meaning is naturally given by the formula and many of its properties such as partial correctness, termination or equivalence of programs are expressible within the framework of the logical system. Note, however, that the complete semantics of the programming language cannot be defined in the logical system alone. We need to describe it in some other way and prove that executing a program according to the complete semantics does not contradict the declarative meaning of the program given in the logical system. In the case of pure Prolog, the logical system is the first order predicate logic, a program is a set of Horn clauses and the declarative meaning of the program is a conjunction of the Horn clauses, while how to execute a program, i.e. the operational semantics of pure Prolog is SLD-resolution with a certain strategy e.g. depth-first or breadth-first.

It is possible to express many properties of programs in the framework of the usual first order logic. Clark and Tärnlund[7] proposed to use the first order predicate logic. (See also Clark and Darlington[6], Hansson and Tärnlund[8].) In their system, data structure is defined by some predicate and a predicate is defined by first order recursion equations. But as is pointed out by them, in order to characterize the data structure completely, it is necessary to introduce an induction schema on that data structure, i.e. to add an extremal clause to the predicate definitions. For example, consider the set of natural numbers. To characterize it as a data structure, the following clauses are introduced.

(1) 0 is a natural number.

(2) x is a natural number iff the successor of x is a natural number.

These clauses, formulated in the form of recursion equations, do not completely characterize the set of natural numbers. According to Peano axioms, it is necessary to introduce the induction schema on natural numbers besides the above two clauses. However, in general, there is no guarantee that the predicate definition and the induction schema are compatible with each other. The most significant difference between our system and Clark and Tärnlund[7]'s system is that we incorporate into our system a rule to derive such an induction schema from predicate definitions. The rule is called *production elimination.*

Given a set of recursion equations which define a predicate, there are two alternative ways to define semantics of the predicate. One is to regard the predicate as the least fixpoint of the equations and another is to regard it as the greatest fixpoint. (See Apt and van Emden[2], Sato[14] for the investigations of the greatest fixpoint semantics.) The recursion equations themselves do not determine which semantics to choose, but introducing production elimination forces us to choose the least fixpoint semantics; production elimination is a syntactic representation of the least fixpoint semantics. A similar rule can also be introduced which represents the greatest fixpoint semantics. However, the important distinction is that production elimination derives a wide variety of induction schemata. This is one of the reasons why we choose the least fixpoint semantics.

The declarative meaning of a logic program can be regarded as its specification. But generally the specifications of programs often contain universal quantifications and other logical symbols. By using the theory of generalized inductive definitions, we can define predicates of higher *levels* in terms of full first order formulas constructed from predicates of lower levels; a higher level predicate serves as a specification of lower level ones. Usually, higher level predicates are not (or can not be) executed directly, but are transformed to lower level ones, or are used to prove some properties of lower level predicates. But, of course, there is no reason why we *should* not execute them directly.

In our extension of logic programs presented here, a condition of a clause may contain any logical symbols. This makes it possible to write a quantified specification as a logic program. It also makes the class of induction schemata much larger to include the usual course-of-values inductions.

How to treat negation is one of the most problematic points in logic programming. We also discuss the problem of negation in our formalism. The point is that the so called *negation as failure* rule (Clark[5]) is derivable in our system.

1. ID

We introduce a logical system called **ID**.

1.1. Symbols

The symbols of **ID** consist of the following.

(L1) Constants

Countably many individual constants:

a string of roman alphabetical characters which begins with an uppercase character e.g. Zero, Nil.

Countably many n-ary function constants for each n $\geqslant 0$:

a string of roman lowercase characters which may have indices *e.g.* cons, s, fact.

A nullary predicate constant of level 0: $\perp$.

A binary predicate constant of level 0: $=$.

Countably many n-ary predicate constants of level m for each n $\geqslant$ 0 and m $\geqslant$ 1:

a string of roman alphabetical characters which begins with an uppercase character which may have indices

e.g. Nat, List, Fib.

(L2) Variables

Countably many individual variables:

a single lowercase roman alphabetical character which may have indices

e.g. x, y, a, b, l.

(L3) Logical symbols

$\wedge, \vee, \supset, \forall, \exists$

1.2. Terms, Formulas

Terms are defined as follows:

(T1) An individual constant is a term.

(T2) A variable is a term.

(T3) If f is an n-ary function constant and $t_1, \cdots, t_n$ are terms, the $f(t_1, \cdots, t_n)$ is a term.

Atomic formulas (*predicates*) are defined as follows:

(A1) If P is an n-ary predicate constant and $t_1, \cdots, t_n$ are terms, then $P(t_1, \cdots, t_n)$ is an atomic formula.

We call n the *arity* of P. We impose a restriction that syntactically identical predicate constants should have a fixed arity and level. We abbreviate $\perp()$, $=(s, t)$ as $\perp$, $s = t$ and if $\boldsymbol{s}$ and $\boldsymbol{t}$ are sequences of terms of length n, i.e. $\boldsymbol{s}$ is $s_1, \cdots, s_n$ and $\boldsymbol{t}$ is $t_1, \cdots, t_n$, we abbreviate $s_1{=}t_1, \cdots, s_n{=}t_n$ as $\boldsymbol{s} = \boldsymbol{t}$.

Formulas are defined as usual, where $\neg A$, $A \leftrightarrow B$ and $A_1, \cdots, A_n \supset B$ are abbreviations of $A \supset \perp$, $(A \supset B) \wedge (B \supset A)$ and $A_1 \supset (\cdots (A_n \supset B) \cdots)$ respectively.

Free and *bound occurrence* of a variable in a formula is defined as usual.

To define *P-forms*, we introduce countably many symbols $*_1, *_2, \cdots$ which are foreign to **ID**. P-forms are defined as follows:

(P1) $*_i$ is a P-form.

(P2) A formula is a P-form.

(P3) If **P** and **Q** are P-forms, then $\mathbf{P} \wedge \mathbf{Q}$, $\mathbf{P} \vee \mathbf{Q}$, $\forall x.\mathbf{P}$ and $\exists x.\mathbf{P}$ are P-forms.

(P4) If F is a formula and **P** is a P-form, then $F \supset \mathbf{P}$ is a P-form.

The *degree* of a P-form **P** is the largest n such that $*_n$ occurs in **P**. If **P** is a P-form of degree n and $F_1, \cdots, F_n$ are formulas, $\mathbf{P}[F_1, \cdots, F_n]$ denotes the result of replacing the symbols $*_1, \cdots, *_n$ in **P** by $F_1, \cdots, F_n$ respectively.

For syntactic variables, we use

x, y, z, a for variables,

$\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{z}$ for sequences of variables,

r, s, t, u for terms,

$\boldsymbol{r}$, $\boldsymbol{s}$, $\boldsymbol{t}$, $\boldsymbol{u}$ for sequences of terms,

P, Q, R for predicate constants,

A, B, C, F, G, H for formulas,

P, **Q** for P-forms,
Γ, Δ for sequences of formulas.

They may have indices.

Sequents are defined as follows:

(S1) $\Gamma \rightarrow F$ is a sequent.

The notation for *substitution* is defined as follows:

(ST) Let $\boldsymbol{x}$ be $x_1, \cdots, x_n$ and $\boldsymbol{t}$ be $t_1, \cdots, t_n$. $s_{\boldsymbol{x}}(\boldsymbol{t})$ represents a term obtained by simultaneously substituting t_i for all occurrences of x_i.

(SF) Let $\boldsymbol{x}$ be $x_1, \cdots, x_n$ and $\boldsymbol{t}$ be $t_1, \cdots, t_n$. $A_{\boldsymbol{x}}(\boldsymbol{t})$ represents a formula obtained by simultaneously substituting t_i for all occurrences of x_i with renaming of bound variables in A if variables in $\boldsymbol{t}$ would be bound in A.

If A and B are formulas such that A is $B_{\boldsymbol{x}}(\boldsymbol{t})$, we express this fact by writing A as $B(\boldsymbol{t})$ and B as $B(\boldsymbol{x})$.

We say d' is a *variant of* d *with respect to* e (d, e are sequences of terms or formulas) if d' is obtained by renaming variables of d, and d' and e do not have free variables in common.

1.3. Inference rules

An *inference rule* is of the form

$$\frac{S_1 \ \cdots \ S_n}{S} \qquad n \geqslant 0$$

where S_i, S are sequents.

We call $S_1, \cdots, S_n$ the *premises* and S the *consequence* of the inference rule.

A *proof* is defined as usual.

The inference rules consist of the following.

(i) intuitionistic logic

(ii) inference rules for equality

(iii) production introduction and production elimination

1.3.1. Logic

Logical part of **ID** is the intuitionistic logic.

The rules are

$$(Ax) \quad \frac{}{\Gamma, A, \Delta \rightarrow A}$$

$$(Wk) \quad \frac{\Gamma \rightarrow A}{B, \Gamma \rightarrow A}$$

$$(Cntr) \quad \frac{A, A, \Gamma \rightarrow B}{A, \Gamma \rightarrow B}$$

$$(Ex) \quad \frac{\Gamma, A, B, \Delta \rightarrow C}{\Gamma, B, A, \Delta \rightarrow C}$$

$$(Cut) \quad \frac{\Gamma \rightarrow A \quad A, \Delta \rightarrow B}{\Gamma, \Delta \rightarrow B}$$

$$(\wedge I) \quad \frac{\Gamma \rightarrow A \quad \Gamma \rightarrow B}{\Gamma \rightarrow A \wedge B} \qquad (\wedge E) \quad \frac{\Gamma \rightarrow A \wedge B}{\Gamma \rightarrow A} \quad \frac{\Gamma \rightarrow A \wedge B}{\Gamma \rightarrow B}$$

$$(\vee I) \quad \frac{\Gamma \rightarrow A}{\Gamma \rightarrow A \vee B} \quad \frac{\Gamma \rightarrow B}{\Gamma \rightarrow A \vee B} \qquad (\vee E) \quad \frac{\Gamma \rightarrow A \vee B \quad A, \Gamma \rightarrow C \quad B, \Gamma \rightarrow C}{\Gamma \rightarrow C}$$

$$(\supset I) \quad \frac{A, \Gamma \rightarrow B}{\Gamma \rightarrow A \supset B} \qquad (\supset E) \quad \frac{\Gamma \rightarrow A \supset B \quad \Gamma \rightarrow A}{\Gamma \rightarrow B}$$

$$(\forall I)\ \frac{\Gamma \rightarrow A(a)}{\Gamma \rightarrow \forall x.A(x)} \qquad (\forall E)\ \frac{\Gamma \rightarrow \forall x.A(x)}{\Gamma \rightarrow A(t)}$$

$$(\exists I)\ \frac{\Gamma \rightarrow A(t)}{\Gamma \rightarrow \exists x.A(x)} \qquad (\exists E)\ \frac{\Gamma \rightarrow \exists x.A(x) \quad A(a),\Gamma \rightarrow B}{\Gamma \rightarrow B}$$

$$(\bot E)\ \frac{\Gamma \rightarrow \bot}{\Gamma \rightarrow A}$$

where in $(\forall I)$ a must not occur free in Γ and in $(\exists E)$ a must not occur free in B and Γ.

1.3.2. Inference rules for equality

The intended meaning of a predicate constant $=$ is, of course, equality. We need some rules to characterize equality. The basic rules are

$$\frac{}{\rightarrow t = t} \qquad \frac{}{s = t \rightarrow t = s} \qquad \frac{}{r = s,\ s = t \rightarrow r = t}$$

$$\frac{}{\boldsymbol{s} = \boldsymbol{t} \rightarrow r_x(\boldsymbol{s}) = r_x(\boldsymbol{t})} \qquad \frac{}{A(\boldsymbol{s}),\ \boldsymbol{s} = \boldsymbol{t} \rightarrow A(\boldsymbol{t})}$$

where $\boldsymbol{s}$ and $\boldsymbol{t}$ are of the same length.

Moreover, as for equality and falsity, we can introduce any rules so long as they do not violate the constraint on levels. (See 1.3.3.) For example, we can introduce the following rules by Clark[5].

$$\frac{c = c'}{\bot} \quad \textit{for distinct individual constants } c,\ c'$$

$$\frac{f(x_1, \cdots, x_n) = g(y_1, \cdots, y_m)}{\bot} \quad \textit{for distinct function constants } f,\ g$$

$$\frac{f(x_1, \cdots, x_n) = f(y_1, \cdots, y_n)}{x_i = y_i} \quad \textit{for any function constant } f$$

$$\frac{f(x_1, \cdots, x_n) = c}{\bot} \quad \textit{for any function constant } f \textit{ and individual constant } c$$

$$\frac{t = x}{\bot} \quad \textit{for any term } t \textit{ in which } x \textit{ ocuurs}$$

They are formulated as what we call productions. (See 1.3.3.) With these rules, we can explain the mechanism of unification used in Prolog. (See 4.1.) We call these rules Peq.

On the other hand, we can interpret function constants by introducing some rules, e.g.

$$\frac{}{\mathrm{fact}(0) = \mathrm{s}(0)} \qquad \frac{}{\mathrm{fact}(\mathrm{s}(\mathrm{x})) = \mathrm{times}(\mathrm{s}(\mathrm{x}),\mathrm{fact}(\mathrm{x}))}$$

. . .

1.3.3. Production introduction and production elimination

These rules are most important and useful when we prove formulas in **ID**. These rules are the elaboration of those in Martin-Löf[10]. First we define production.

1.3.3.1. Production

Productions are schemata for defining predicates inductively. It has a figure of the form

$$\frac{F_1 \ \cdots \ F_n}{P(\boldsymbol{t})} \quad n \geqslant 0 \qquad \text{(p)}$$

where $\boldsymbol{t}$ is a sequence of terms, P is a predicate constant, F_i is a formula

$$\mathbf{P}_i[Q_{i1}(\boldsymbol{t}_{i1}),\ \cdots,\ Q_{ik_i}(\boldsymbol{t}_{ik_i})]$$

$\mathbf{P}_i$ is a P-form of degree k_i, Q_{ij} is a predicate constant and the condition on levels

(*) the levels of Q_{ij} should be less than or equal to the level of P and the levels of predicates in $\mathbf{P}_i$ should be less than the level of P

is satisfied.

We call $F_1, \cdots, F_n$ the conditions of the production (p).

Example

$$\frac{}{\text{List(Nil)}} \qquad \frac{\text{List(l)}}{\text{List(cons(x,l))}}$$

1.3.3.2. Production introduction

A *production introduction* of a production (p) is .

$$\frac{\Gamma,\boldsymbol{s}=\boldsymbol{t}',\Delta\to F_1' \quad \cdots \quad \Gamma,\boldsymbol{s}=\boldsymbol{t}',\Delta\to F_n'}{\Gamma,\boldsymbol{s}=\boldsymbol{t}',\Delta\to P(\boldsymbol{s})} \qquad \text{(pI)}$$

where $\boldsymbol{t}'$, F_1', $\cdots$, F_n' is a variant of $\boldsymbol{t}$, F_1, $\cdots$, F_n with respect to $\boldsymbol{s}$.

Example

$$\frac{\text{cons(u,cons(v,w))} = \text{cons(x,l)} \to \text{List(l)}}{\text{cons(u,cons(v,w))} = \text{cons(x,l)} \to \text{List(cons(u,cons(v,w)))}}$$

1.3.3.3. Production elimination

To define *production elimination*, we introduce the definition of *link* which is a relation between predicate constant.

(1) A predicate constant is linked with itself.

(2) If a predicate constant P occurs in the conclusion of the following production

$$\frac{\cdots \ \mathbf{P}_i[\cdots,Q_{ij}(\boldsymbol{s}_{ij}),\cdots] \ \cdots}{P(\boldsymbol{s})}$$

then P is linked with every predicate constant which is linked with Q_{ij}.

Production elimination of the inductively defined predicate constant P is of the form

$$\frac{\Gamma\to P(\boldsymbol{t}) \quad \textit{minor premises}}{\Gamma\to F} \qquad \text{(pE)}$$

We explain how to make minor premises. First, we choose an arbitrary set **Ps** of predicate constant such that it contains P and its members are linked with P. We associate a formula and a sequence of terms with each predicate constant in **Ps** as follows.

(1) We associate F and $\boldsymbol{t}$ with P.

(2) For a predicate constant Q other than P, we associate an arbitrary formula and sequence of terms whose length is the arity of Q.

A minor premise is constructed for each pair of a predicate constant Q in **Ps** and a production whose conclusion contains Q. Let the production be of the form

$$\frac{\cdots \ \mathbf{Q}_i[\cdots,R_{ij}(\boldsymbol{s}_{ij}),\cdots] \ \cdots}{Q(\boldsymbol{s})}$$

and let

$\boldsymbol{r}'$, G', $\boldsymbol{r}_{ij}'$, G_{ij}' be a variant of $\boldsymbol{r}$, G, $\boldsymbol{r}_{ij}$, G_{ij} with respect to Γ,

$\boldsymbol{z}_{ij}$ be a sequence of all the variables in $\boldsymbol{r}_{ij}$,

$\boldsymbol{s}'$, F_i, H_i be a variant of $\boldsymbol{s}$, $\mathbf{Q}_i[\cdots,R_{ij}(\boldsymbol{s}_{ij}),\cdots]$, $\mathbf{Q}_i[\cdots,H_{ij},\cdots]$

with respect to $\boldsymbol{r}'$, G', $\boldsymbol{r}_{ij}'$, G_{ij}', $\Gamma$,

where

$G$ and $\boldsymbol{r}$ are associated with $Q$,

$$H_{ij} = \begin{cases} \forall z_{ij}(\boldsymbol{r}_{ij}'=\boldsymbol{s}_{ij} \supset G_{ij}') & \textit{if } G_{ij} \textit{ and } \boldsymbol{r}_{ij} \textit{ are associated with } R_{ij} \\ R_{ij}(\boldsymbol{s}_{ij}) & \textit{otherwise.} \end{cases}$$

The corresponding minor premise is

$$\boldsymbol{r}'=\boldsymbol{s}', \quad \cdots, F_i, H_i, \quad \cdots, \Gamma \rightarrow G' \qquad \text{(PpE)}$$

For P, $t$ and $F$, we may have several production eliminations according to **Ps** and the association which we make.

Example

For a predicate constant ‘List’ defined in the above example, production elimination is of the form

$$\frac{\Gamma \rightarrow \mathrm{List}(t) \quad t_{\mathrm{y}}(\mathrm{z})=\mathrm{Nil},\Gamma \rightarrow \mathrm{F}(\mathrm{z}) \quad t_{\mathrm{y}}(\mathrm{z})=\mathrm{cons}(\mathrm{x},\mathrm{l}),\mathrm{List}(\mathrm{l}),\forall \mathrm{z}.(t_{\mathrm{y}}(\mathrm{z})=\mathrm{l} \supset \mathrm{F}_{\mathrm{y}}(\mathrm{z})),\Gamma \rightarrow \mathrm{F}(\mathrm{z})}{\Gamma \rightarrow \mathrm{F}(\mathrm{y})}$$

where y is the only variable in $t$, and x and l do not occur free in $\Gamma$. To help the understanding of the reader, we list the correspondence between the constructs in the second minor premise of the above schema and those of (PpE).

$t \cdots \boldsymbol{r}$
$t_{\mathrm{y}}(\mathrm{z}) \cdots \boldsymbol{r}'$
$\mathrm{z} \cdots z_{ij}$
$\mathrm{cons}(\mathrm{x},\mathrm{l}) \cdots \boldsymbol{s}'$
$\mathrm{F}(\mathrm{y}) \cdots G$
$\mathrm{F}(\mathrm{z}) \cdots G'$
$\mathrm{List} \cdots Q$

These are our definition of **ID**. If we want to make explicit that **ID** has a set of productions $I$, we use the notation **ID**($I$). Provability symbol $\vdash_{\mathbf{ID}(I)}$ is used as usual.

2. Production

2.1. Meaning of production elimination

Production elimination may seem very complicated. We explain its meaning by an example.

$$\text{(n1)} \ \frac{}{\mathrm{Nat}(0)} \qquad \text{(n2)} \ \frac{\mathrm{Nat}(\mathrm{x})}{\mathrm{Nat}(\mathrm{s}(\mathrm{x}))}$$

0 is an abbreviation of an individual constant Zero. The intended meaning of Nat($x$) is that $x$ is an natural number. One of the production eliminations of Nat is

$$\frac{\Gamma \rightarrow \mathrm{Nat}(\mathrm{x}) \quad \mathrm{y}=0,\Gamma \rightarrow \mathrm{F}(\mathrm{y}) \quad \mathrm{y}=\mathrm{s}(\mathrm{z}),\mathrm{Nat}(\mathrm{z}),\forall \mathrm{y}.(\mathrm{y}=\mathrm{z} \supset \mathrm{F}(\mathrm{y})),\Gamma \rightarrow \mathrm{F}(\mathrm{y})}{\Gamma \rightarrow \mathrm{F}(\mathrm{x})}$$

As its derived rule, we have

$$\frac{\Gamma \rightarrow \mathrm{Nat}(\mathrm{x}) \quad \Gamma \rightarrow \mathrm{F}(0) \quad \mathrm{Nat}(\mathrm{z}),\mathrm{F}(\mathrm{z}),\Gamma \rightarrow \mathrm{F}(\mathrm{s}(\mathrm{z}))}{\Gamma \rightarrow \mathrm{F}(\mathrm{x})}$$

This is exactly the induction schema on natural numbers. This also means that Nat is the minimal solution of the equation

$$X(0) \wedge \forall \mathrm{x}.(X(\mathrm{x}) \supset X(\mathrm{s}(\mathrm{x})))$$

where $X$ is a predicate variable which is unknown.

Similarly, we are able to derive an induction schema from the production elimination

$$\frac{\Gamma \rightarrow P(x) \quad \textit{minor premises}}{\Gamma \rightarrow F}$$

and it implies that P is the minimal solution.

Roughly speaking, minor premises of a production elimination are obtained by replacing the predicate of productions by the formula associated with it, but the replacement is allowed only when the argument of the predicate belongs to some domain which is determined by the sequence of terms associated with the predicate constant. ($\mathbf{r}'=\mathbf{s}'$ and $r_{ij}'=s_{ij}'$ of (PpE) defines the domain.) This means that production elimination expresses the minimality of the restricted predicate.

2.2. Advantage of production elimination

The reason why we use production is that it is very natural to define predicate. For example, a set of productions

$$\frac{}{\mathrm{List(Nil)}} \qquad \frac{\mathrm{List(l)}}{\mathrm{List(cons(x,l))}}$$

with a production elimination is more natural than

$$\forall \mathrm{x.(List(x)} \leftrightarrow \mathrm{x{=}Nil} \vee \exists \mathrm{yl.(x{=}cons(y,l)} \wedge \mathrm{List(l)))}$$

and easy to treat. Moreover, the latter form of the predicate definition, i.e. the predicate definition by if and only-if, does not imply the minimality of the defined predicate. We give an easy example to show that the if and only-if definition does not imply minimality. The definition of a predicate constant 'False'

$$\mathrm{False()} \leftrightarrow \mathrm{False()}$$

does not decide 'False' at all, while the definition by a production with a production elimination implies that False() $\leftrightarrow \perp$ as is seen in Example 3 in 2.3.

Another reason why we adopt the production elimination is that it provides a wide variety of induction schemata. It is natural that the production elimination takes a form of the induction, because the least fixpoint of the transformation associated with productions (see Apt and van Emden[2]) is equal to the union of the finite powers of the transformation applied to the least element. (The least element corresponds to the base case and the transformation to the step of the induction.)

2.3. Examples of production elimination of plain production

Now we give some examples of productions in which predicate constants are of at most level 1 and the conditions are all atomic. We call such a production a plain production.

Example 1

Production elimination does not necessarily derive a ordinary induction schema. Let Nat be a predicate constant defined in 2.1. Another production elimination of Nat is

$$\frac{\Gamma \rightarrow \mathrm{Nat(s(x))} \quad \mathrm{s(y){=}0}, \Gamma \rightarrow \mathrm{F(y)} \quad \mathrm{s(y){=}s(z),Nat(z)}, \forall \mathrm{y.(s(y){=}z} \supset \mathrm{F(y))}, \Gamma \rightarrow \mathrm{F(y)}}{\Gamma \rightarrow \mathrm{F(x)}}$$

Since the intended meaning of Nat(s(x)) is that s(x) is a natural number, i.e. x is a natural number, the proof of Nat(s(x))→$F(x)$ and that of Nat(x)→$F(x)$ will have some relation. If the following rules (some of Peano axioms)

$$\frac{\mathrm{s(x) = s(y)}}{\mathrm{x = y}} \qquad \frac{\mathrm{s(x) = 0}}{\perp}$$

are introduced, we have

$$\frac{\Gamma \rightarrow \mathrm{Nat(s(x))} \quad \mathrm{Nat(z)}, \Gamma \rightarrow \mathrm{F(z)}}{\Gamma \rightarrow \mathrm{F(x)}}$$

as its derived rule.

Example 2

If we have productions

$$\overline{\text{Isblock}(A)} \qquad \overline{\text{Isblock}(B)} \qquad \overline{\text{Isblock}(C)}$$

then,

$$\frac{\Gamma \rightarrow \text{Isblock}(x) \quad y{=}A,\Gamma \rightarrow F(y) \quad y{=}B,\Gamma \rightarrow F(y) \quad y{=}C,\Gamma \rightarrow F(y)}{\Gamma \rightarrow F(x)}$$

is a production elimination, which permits us to argue by case-analysis. In the above production elimination, y should not appear in Γ, but x may appear in Γ. However, if x appears in Γ, it is desirable that the minor premises are $x{=}A,\Gamma \rightarrow F(x)$ etc. We are able to derive such a rule as follows.

$$\dfrac{\dfrac{\Gamma(x) \rightarrow \text{Isblock}(x) \quad \dfrac{y{=}A,\Gamma(x),\Gamma(y) \rightarrow F(y)}{y{=}A,\Gamma(x) \rightarrow \Gamma(y) \supset F(y)} \quad \dfrac{y{=}B,\Gamma(x),\Gamma(y) \rightarrow F(y)}{y{=}B,\Gamma(x) \rightarrow \Gamma(y) \supset F(y)} \quad \dfrac{y{=}C,\Gamma(x),\Gamma(y) \rightarrow F(y)}{y{=}C,\Gamma(x) \rightarrow \Gamma(y) \supset F(y)}}{\Gamma(x) \rightarrow \Gamma(x) \supset F(x)}}{\Gamma(x) \rightarrow F(x)}$$

Example 3

We define a nullary predicate False().

$$\frac{\text{False}()}{\text{False}()}$$

For any formula F, we can prove False() → F using production elimination of False. The proof is

$$\frac{\overline{\text{False}() \rightarrow \text{False}()} \qquad \overline{\text{False}(),F,\text{False}() \rightarrow F}}{\text{False}() \rightarrow F}$$

So we can prove

$$\text{False}() \leftrightarrow \perp$$

As is explained in section 3, we can regard the above production as a program of Prolog. If we execute False(), it does not terminate because of an infinite recursion. But as the meaning of False() is ⊥, it is desirable that the execution fails. We implemented Prolog which has a facility to make this possible. Note that this facility is reasonable because we adopt the least fixpoint semantics.

3. Prolog and its foundation

3.1. Pure Prolog

We are concerned with a program of Prolog which corresponds to a set of productions of **ID**. We suppose that

(i) every predicate has a fixed arity

(ii) an argument is a term of **ID**.

For example, if we execute the following goal

```
  <- Add(s(s(0)), s(0), z)
```

under a program

```
  Add(0, y, y) <-
  Add(s(x), y, s(z)) <- Add(x, y, z)
```

we get a response

$$z = s(s(s(0)))$$

When a predicate A is executed under a program P and execution terminates with success, we write

$$P \vdash_{\text{Prolog}} A$$

and when a predicate A is executed under a program P and execution fails, we write

$$P \not\vdash_{\text{Prolog}} A$$

A program of Prolog can be regarded as a set of plain productions such that the levels of the predicate constants are 1. The above program is converted to

$$\frac{}{\text{Add}(0, y, y)} \qquad \frac{\text{Add}(x, y, z)}{\text{Add}(s(x), y, s(z))}$$

Executing a predicate in Prolog corresponds to generating its proof in **ID**. In the case of the above example, the following proof is generated.

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\Gamma \to \text{Add}(x_2, y_2, z_2)}}{\Gamma \to \text{Add}(x_1, y_1, z_1)}}{\Gamma \to \text{Add}(s(s(0)), s(0), z)}}{\Gamma \to \exists z.\text{Add}(s(s(0)), s(0), z)}$$

where Γ is

$$s(s(0))=s(x_1),s(0)=y_1,z=s(z_1),x_1=s(x_2),y_1=y_2,z_1=s(z_2),x_2=0,y_2=y_3,z_2=y_3$$

Compare this proof with the *procedural interpretation* (Kowalski[9]). Applying the inference rule corresponds to calling the body and Γ to the unification. We get an answer $z = s(s(s(0)))$ by analyzing Γ.

Note that the above proof is normal. (For the definition of a normal proof see Prawitz[12], Martin-Löf[10].)

We say Γ is a *u-seq* if Γ is a sequence of formulas $s=t$ and Γ is consistent. Therefore, whether Γ is a u-seq or not depends on equality rules.

3.2. Foundation of Prolog

Usually it is explained that Prolog is based on SLD-resolution. (Kowalski[9], Apt and van Emden[2]) But it is more natural to regard a Prolog program and execution of it as a set of productions and generation of a normal proof than to regard them as a set of Horn clauses and SLD-resolution, since it reflects more faithfully the procedural interpretation of predicate logic and the completeness is proved more easily and naturally using the normalization theorem. (See 3.4.) Relations between natural deduction and resolution are investigated in e.g. Bibel[3], Andrew[1]. Some resolution procedures are more clearly understood in terms of deduction, even if deduction and refutation are equivalent. In our case, equivalence is almost trivial, but we have another advantage when we extend a Horn clause to a more general first order formula. (See 4.2 and 4.3.)

3.3. Partial correctness of Prolog

For a Prolog program P.p, we can make a set of productions P.

According to the (rather informal) explanation of Prolog in section 3.1,

$$P.\text{p} \vdash_{\text{Prolog}} A \quad \Rightarrow \quad \vdash_{\text{ID}(P)} \Gamma \to \exists x.A \quad \textit{for some u-seq } \Gamma$$

where A is a predicate defined in P, and x is a sequence of free variables in A.

This means partial correctness. Furthermore, in the case of Prolog with occur-check,

$$P.\mathrm{p}\vdash_{\mathrm{Prolog}} A \quad\Rightarrow\quad \vdash_{\mathbf{ID}(P\cup\mathrm{Peq})} \to \exists x.A$$

3.4. Completeness of Prolog

Completeness can be described by

$$\vdash_{\mathbf{ID}(P\cup\mathrm{Peq})} \to \exists x.A \quad\Rightarrow\quad P.\mathrm{p}\vdash_{\mathrm{Prolog}} A$$

Does this really hold? There are two problems.

The first problem is that the proof of $\to\exists x.A$ in $\mathbf{ID}(P\cup\mathrm{Peq})$ must be normal, for A to terminate with success under P.p. In general, the proof of $\to\exists x.A$ is not necessarily normal. But according to the normalization theorem (Martin-Löf[10]), if the proof of formula F exists, there is a normal proof of F. In this case, since A is a predicate, a normal proof of $\to\exists x.A$ consists of only production introductions except for the last rule $\exists I$.

The second problem is as follows. As is explained in section 3.1, Prolog searches for a normal proof of A. If the search is ideal, Prolog will find a normal proof of A if A holds. But usually, Prolog uses a depth-first search and it may result in non-termination. If a breadth-first search is introduced, Prolog will find a normal proof whenever it exists, i.e. the completeness holds.

To conclude, if $\vdash_{\mathbf{ID}(P\cup\mathrm{Peq})}\to\exists x.A$, $P.\mathrm{p}\vdash_{\mathrm{Prolog}} A$ holds or the execution of A does not terminate. But it never happens that A fails in spite of $\vdash_{\mathbf{ID}(P\cup\mathrm{Peq})}\to\exists x.A$.

Compare our treatment of the completeness to that of SLD-resolution in Apt and van Emden[2]. Ours is much simpler, though the simplicity depends partly on the fact that we have not strictly defined the execution procedure of Prolog. As mentioned above, if Prolog is ideal (i.e. uses breadth-first strategy), it is complete. This corresponds to the completeness of SLD-resolution. When Prolog searches a proof, there are two choices, that is,

(i) order of proving premises,

(ii) selection of productions.

Our approach gives an intuition that (i) does not influence completeness and in fact it is right. Corresponding result is also proved in the case of SLD-resolution (Apt and van Emden[2]). If our proof of the fact that (i) need not be considered is described strictly, it will be essentially not so different as the case of SLD-resolution, but our approach has an advantage of ease of understanding.

4. Extensions

4.1. Negation as failure

Negation as failure can be interpreted as a series of successive production eliminations. We explain the situation by an example.

Example

Consider the following Prolog program

(N1) Nat(0) <−
(N2) Nat(s(x)) <− Nat(x)
(N3) <− Nat(s(t(0)))

Since execution of (N3) fails, it is supposed that ¬Nat(s(t(0))) holds if we use *negation as failure.* (N1) and (N2) are converted to the productions (n1) and (n2) in 2.1 and ¬Nat(s(t(0))) is proved in $\mathbf{ID}(\{(\mathrm{n1}),(\mathrm{n2})\}\cup\mathrm{Peq})$ as follows.

$$\dfrac{\dfrac{}{\mathrm{Nat}(s(t(0))),\Gamma \to \mathrm{Nat}(s(t(0)))} \quad \dfrac{\cdots}{s(t(0))=0,\Gamma' \to \bot} \quad \dfrac{\dfrac{\Pi}{\cdots}}{s(t(0))=s(x),\mathrm{Nat}(x),s(t(0))=x\supset\bot,\Gamma' \to \bot}}{\dfrac{\mathrm{Nat}(s(t(0))),\Gamma \to \bot}{\Gamma \to \neg\mathrm{Nat}(s(t(0)))}}$$

where Γ is a u-seq in which x and y do not occur free, Γ′ is Nat(s(t(0))), Γ and Π is

$$\dfrac{\dfrac{}{\mathrm{Nat(t(0)),\Gamma' \to Nat(t(0))}} \quad \dfrac{\cdots}{\mathrm{t(0)=0,\Gamma'' \to \bot}} \quad \dfrac{\cdots}{\mathrm{t(0)=s(y),Nat(y),t(0)=y \supset \bot,\Gamma'' \to \bot}}}{\mathrm{Nat(t(0)),\Gamma' \to \bot}}$$

where Γ″ is Nat(t(0)), Γ′.

Note that $\mathrm{s(t(0))=0 \to \bot}$, $\mathrm{t(0)=0 \to \bot}$, $\mathrm{t(0)=s(y) \to \bot}$ correspond to three failures in executing (N3), i.e. s(t(0)) does not match with 0, t(0) with 0, t(0) with s(y).

In general, we can describe *negation as failure* by

$$P.\,\mathrm{p} \not\vdash_{\mathrm{Prolog}} A \;\Rightarrow\; \vdash_{\mathbf{ID}(P\cup \mathrm{Peq})} \to \forall \boldsymbol{x}.\neg A$$

and justify it by converting a failure tree in the sense of Clark[5] into a proof of **ID**. Since it is almost obvious, we leave it to the reader.

As is known from the above discussion, unification of Prolog can be regarded as a built-in equality test procedure in **ID**(Peq). Therefore, if we have some other equality rules and the procedure to test the equality and use it instead of unification, we have a variation of Prolog, e.g. Prolog whose terms are functions.

4.2. Introducing higher level predicates

4.2.1. Condition on levels

Up to now, we have only considered what we call plain productions. Plain productions are enough for the so-called pure Prolog, but, as was discussed in 4.1, the usual Prolog interpreter can actually treat negations of predicates by *negation as failure* and in fact a predicate constant can be defined in terms of the negation of other predicates. But allowing arbitrary productions, which may not satisfy the condition on levels defined in 1.3.3.1, we can soon prove a contradiction. The simplest example is the following production:

$$\frac{\neg\mathrm{Liar()}}{\mathrm{Liar()}}.$$

Considering Liar() as the minimal solution of

$$\neg X \supset X,$$

we can conclude that Liar() is true. But using the following production elimination

$$\frac{\mathrm{Liar()\to Liar()} \quad \mathrm{\neg Liar(),\neg\neg Liar(),Liar()\to\neg Liar()}}{\mathrm{Liar()\to\neg Liar()}}$$

and assuming Liar() to be true, we immediately get a contradiction (⊥). This means that the production elimination and the naive least fixpoint semantics are not compatible, when the condition on levels is violated. It is because the (illegal) production does not introduce a monotone transformation in the sense of Apt and van Emden[2], as was discussed in 2.2.

The condition on levels requires that the predicate constants of lower levels should have been completely defined before the process of defining the predicate constants of the higher levels. It guarantees that the associated transformation is monotone at each level of the definition, so that production elimination is a valid rule with respect to the least fixpoint semantics.

4.2.2. Extension with negation

We consider Prolog programs in which a negation of a predicate may appear as a condition of a clause. With this extension, when transforming clauses to productions, we should explicitly check the condition on levels.

Example Member and Insert

$$\frac{}{\text{Member}(x,\text{cons}(x,l))} \qquad \frac{\text{Member}(x,l)}{\text{Member}(x,\text{cons}(y,l))}$$

$$\frac{\text{Member}(x,l)}{\text{Insert}(x,l,l)} \qquad \frac{\neg\text{Member}(x,l)}{\text{Insert}(x,l,\text{cons}(x,l))}$$

If we assign level 1 to Member and level 2 to Insert, the above productions satisfy the condition on levels.

Analyzing existing Prolog programs, we have come to believe that almost all the *logical* Prolog programs satisfy the condition on levels with an appropriate assignment of levels.

For the extended programs, *negation as failure* can be justified by transforming the execution with *negation as failure* into a series of successive production introductions and production eliminations, just as in 4.1.

4.3. Towards verification and synthesis

4.3.1. Notion of verification and synthesis

As was discussed in Clark and Tärnlund[7] and also mentioned in 0, since a Prolog program can be mapped to a logical formula (in our case, to a set of inference rules), many of the properties of the program are naturally formalized and proved inside pure logic, i.e. without resort to any device such as Hoare's axioms and rules. In the case of logic programming, the correctness problem of a program is formulated rather as the equivalence problem of two programs. The typical example is the following two definitions of Fibonacci sequences:

$$\frac{}{\text{Fib}_1(0,1)} \qquad \frac{}{\text{Fib}_1(1,1)} \qquad \frac{\text{Fib}_1(x,y) \quad \text{Fib}_1(x+1,z)}{\text{Fib}_1(x+2,y+z)}$$

$$\frac{}{G(0,1,1)} \qquad \frac{G(x,y,z)}{G(x+1,z,y+z)} \qquad \frac{G(x,y,z)}{\text{Fib}_2(x,y)}$$

Fib_2 can be regarded as an implementation of Fib_1 (Fib_1 is the specification of Fib_2), or, more moderately, Fib_2 is an optimization of Fib_1. The correctness of Fib_2 relative to Fib_1 can be stated as

$$\forall xy(\text{Fib}_1(x,y) \leftrightarrow \text{Fib}_2(x,y))$$

To prove this formula is the verification of Fib_2 with respect to Fib_1. To generate (automatically) the definition of Fib_2 from the definition of Fib_1, possibly with the proof of the above formula, is the synthesis of Fib_2 from Fib_1, or the transformation of Fib_1 to Fib_2.

4.3.2. Specification with quantifiers

In general, the specification of a program often contains universal quantifiers and other logical symbols. Consider the following production:

$$\frac{\text{Member}(y,l) \quad \forall x(\text{Member}(x,l) \supset \text{Leq}(x,y))}{\text{Max}(y,l)}$$

This production satisfies the condition on levels, if we assign level 2 to Max and level 1 to other predicate constants. Since quantifiers and implication are not allowed in ordinary Prolog, the above production is indeed an extension to Prolog. The higher level predicate Max, which is to be used for the program specification, is defined by a production. (We may execute Max by invoking the general theorem prover of our logical system.) One of the possible implementations of Max is the following set of productions:

$$\frac{}{\text{Max}_1(x,\text{cons}(x,\text{Nil}))} \qquad \frac{\text{Max}_1(x,l) \quad \text{Leq}(x,y)}{\text{Max}_1(y,\text{cons}(y,l))} \qquad \frac{\text{Max}_1(x,l) \quad \text{Leq}(y,x)}{\text{Max}_1(x,\text{cons}(y,l))}$$

In the case of Max, we may add the following formula

$$\forall yl(\text{Max}(y,l) \leftrightarrow \text{Member}(y,l) \wedge \forall x(\text{Member}(x,l) \supset \text{Leq}(x,y)))$$

as an axiom, since it is equivalent to the introduction and the elimination of the production. But, in our system, all the nonlogical informations, such as programs and specification, are formulated in the form of productions, so that the condition on levels always guarantees the consistency of the system.

4.3.3. Course-of-values induction

In proving properties of a logic program, we need various kind of induction schema according to the recursion structure of the program. In our system, each induction schema can be derived as production elimination of an appropriate set of (possibly non-plain) production.

Example course-of-values induction on natural numbers

$$\frac{\mathrm{Nat}(y) \quad \forall x(\mathrm{Nat}(x) \wedge \mathrm{Less}(x,y) \supset \mathrm{Natc}(x))}{\mathrm{Natc}(y)}$$

For the condition on levels, the level of Natc should be bigger than that of other predicate constants. From the elimination of the above production, we can derive the following schema

$$\frac{\Gamma \rightarrow \mathrm{Natc}(t) \quad \forall x(\mathrm{Nat}(x) \wedge \mathrm{Less}(x,y) \supset F(x)), \Gamma \rightarrow F(y)}{\Gamma \rightarrow F(t)}$$

If we have proved

$$\forall x(\mathrm{Nat}(x) \leftrightarrow \mathrm{Natc}(x)),$$

then we can replace Natc(t) by Nat(t) in the schema and get the ordinary course-of-values induction schema on natural numbers.

Usually, such induction schemata are formulated as meta schemata, and in order to justify them on the basis of the primitive schema, one should carry out meta-level arguments. In our system, however, the corresponding justification takes the form of an ordinary formula, as above, and needs no meta-level devices. This is because of the generality (and complexity) of the production system.

5. Concluding remarks

First, let us briefly summarize the model theory of our system. Remember that a specific system of **ID** is determined by a set of rules concerning with the equality and the falsity and a set of productions with a conclusion whose predicate constant is of level $\geqslant 1$. For introducing functions into the system, we did not specify the rules for the equality in advance, but allow each system to have its own equality rules. The rules for the equality are formulated as productions. For building the model for a specific system, we should first define the domain of individuals and assign an interpretation for each function constant. The equality symbol will be interpreted as the equality on that domain.

Once the model for the equality has been constructed, as we pointed out in 4.2, the model for the system is constructed by taking a least fixed point at each level, starting from the set of productions with a conclusion whose predicate constant is of level 1, and proceeding in the increasing order of levels, using the model for the lower level predicates. Since a condition of a production may contain quantifiers, the associated transformation is not necessarily continuous.

Since the schema of production elimination corresponds to the least fixpoint semantics it has a strong relationship to the formalization of non-monotonic logic. In fact, the circumscription of McCarthy[11] is almost the same as the elimination schema of Martin-Löf[10]. The difference is the use of the level hierarchy, which guarantees the existence of the minimal model.

In Bowen[4], he proposed programming in full first order logic by relating sequent calculus with logic programming. His formalism almost applies to ours, since sequent calculus and natural deduction do not differ so much, as far as intuitionistic logic is concerned.

References

[1] Andrew, P. B.: Transforming matings into natural deduction proofs, *5th Conference on Automated Deduction,* Lecture Notes in Computer Science 87, Springer-Verlag, 1980, pp.281-292.

[2] Apt, R. K. and van Emden, M. H.: Contributions to the theory of logic programming, *J.ACM, vol.29, No.3* (Jul. 1982), 841-862.

[3] Bibel, W.: A syntactic connection between proof procedures and refutation procedures, *Theoretical Computer Science 3rd GI Conference,* Lecture Notes in Computer Science 48, Springer-Verlag, 1978, pp.215-224.

[4] Bowen, K. A.: Programming with full first-order logic, *Machine Intelligence 10,* 1982, pp.421-440.

[5] Clark, K. L.: Negation as failure, *Logic and Data Bases (ed. H. Gallaire and J. Miker),* Plenum Press, New York, 1978, pp.293-324.

[6] Clark, K. L. and Darlington, J.: Algorithmic classification through synthesis, *Compt. J., vol.23, No.1* (1980), 61-65.

[7] Clark, K. L. and Tärnlund, S-Å.: A first order theory of data and programs, *Proc. IFIP-77 Congress,* North-Holland, 1977, pp.937-944.

[8] Hansson, Å. and Tärnlund, S-Å.: A natural programming calculus, *Proc. 6th Int. Joint Conf. on Artificial Intelligence,* 1979, pp.348-355.

[9] Kowalski, R. A.: Predicate logic as a programming language, *Information Processing 74 (ed. Rosenfeld J.),* North-Holland, 1974, pp.569-574.

[10] Martin-Löf, P.: Hauptsatz for the intuitionistic theory of iterated inductive definitions, *Proc. Second Scandinavian Logic Symposium,* North-Holland, Amsterdam, 1970, pp.179-216.

[11] McCarthy, J.: Circumscription --- A form of non-monotonic reasoning, *Artificial Intelligence, vol.13* (1980), 27-39.

[12] Prawitz, D.: *Natural deduction,* Almquist and Wksell, Stockholm, 1965.

[13] Prawitz, D.: Ideas and results in proof theory, *Proc. Second Scandinavian Logic Symposium,* North-Holland, Amsterdam, 1970, pp.235-307.

[14] Sato, T.: Negation and semantics of Prolog programs, *Proc. First Int. Logic Programming Conf.,* 1982, pp.169-174.