Incorporating Inductions and Game Semantics into Logic Programming

Keehang Kwon
Dept. of Computer Engineering, DongA University
Busan 604-714, Korea
khkwon@dau.ac.kr

Abstract: Inductions and game semantics are two useful extensions to traditional logic programming. To be specific, inductions can capture a wider class of provable formulas in logic programming. Adopting game semantics can make logic programming more interactive.

In this paper, we propose an execution model for a logic language with these features. This execution model follows closely the reasoning process in real life.

keywords: induction, game semantics, read, computability logic.

1 Introduction

Fixed-point definitions, inductions and game semantics are all useful extensions to the theory of logic programming. In this paper, we propose an execution model that combines these three concepts.

First, logic programming with fixed-point definitions has been studied by several researchers [6, 10]. In this setting, clauses of the form \( A \overset{\Delta}{=} B \) – called definition clauses – are used to provide least fixed-point definitions of atoms. We assume that a set \( \mathcal{D} \) of such definition clauses – which we call a program – has been fixed. The following definition-right rule, which is a variant of the one used in LINC [10], is used in this paper as an inference rule which introduces atomic formulas on the right.

\[
pv(\sigma, \mathcal{G} \vdash A) \text{ if } A' \overset{\Delta}{=} B \in \mathcal{D} \text{ and } A' \theta = A\sigma \text{ and } pv(\sigma\theta, \mathcal{G} \vdash B).
\]

This rule is similar to backchaining in Prolog with the difference that a current answer substitution \( \sigma \) (also called a run) is maintained and applied to formulas in a lazy way here. The definition-left rule represents a case analysis in reasoning.
\begin{align*}
pv(\sigma, A : G \vdash D) \text{ if, for each } \theta \text{ which is the } \text{mgu}(A\sigma, A') \text{ for some } A' \triangleq B \in \mathcal{D}, pv(\sigma\theta, B : G \vdash D).
\end{align*}

Here, \( D \) represents a goal. This rule is well-known and used to instantiate the free variables of the sequent by \( \theta \), which is a most general unifier (mgu) for atoms \( A\sigma \) and \( A' \). If there is no such \( \theta \), the sequent is proved.

Natural number induction is also useful in many applications. We use 0 for zero and \( x + 1 \) for a successor of \( x \). The following \textit{nat-right} rules introduce natural numbers on the right.

\begin{align*}
pv(\sigma, G \vdash \text{nat}(z)).
\end{align*}

\begin{align*}
pv(\sigma, G \vdash \text{nat}(I + 1)) \text{ if } pv(\sigma, G \vdash \text{nat}(I)).
\end{align*}

The \textit{nat-left} rule corresponds to an induction in reasoning.

\begin{align*}
pv(\sigma, \text{nat}(n) \vdash G) \text{ if } pv(\sigma, \emptyset \vdash G(n/0)) \text{ and } pv(\sigma, G(n/j) \vdash G(n/(j + 1))) \text{ where } j \text{ is a new variable.}
\end{align*}

This rule is a well-known induction rule \cite{6} and used to prove a goal \( G \) for all natural numbers using only trivial inductions. As we shall see later, even simple inductions make their implementation difficult.

The operational semantics of these languages \cite{6} is typically based on intuitionistic provability. In the operational semantics based on provability, solving the universally quantified goal \( \forall x D \) from a definition \( \mathcal{D} \) simply \textit{terminates} with a success if it is provable.

In this paper, we make the above operational semantics more “interactive” by adopting the game semantics in \cite{2, 3}. That is, our approach in this paper involves a modification of the operational semantics to allow for more active participation from the user. Solving \( \forall x D \) from a program \( \mathcal{D} \) now has the following two-step operational semantics:

- Step (1): the machine tries to prove \( \forall x D \) from a program \( \mathcal{D} \). If it fails, the machine returns the failure. If it succeeds, goto Step (2).

- Step (2): the machine requests the user to choose a constant \( c \) for \( x \) and then proceeds with solving the goal, \([c/x]D\).

As an illustration of this approach, let us consider the following program.
\[
\{ \text{fact}(0, 1) \triangleq \top, \\
\text{fact}(X + 1, XY + Y) \triangleq \text{fact}(X, Y) \} 
\]

As a particular example, consider a goal task \( \forall x(\text{nat}(x) \supset \exists y \text{fact}(x, y)) \).

To prove that this goal is valid, we need to use induction. Most theorem provers simply terminates with a success as it is solvable. However, in our context, execution requires more. To be specific, execution proceeds as follows: the system requests the user to select a particular number for \( x \). After the number – say, 5 – is selected, the system returns \( y = 120 \). As seen from the example above, universally quantified goals in intuitionistic logic can be used to model the \textit{read} predicate in Prolog.

In this paper we present the syntax and semantics of this language called Prolog\(^{\text{Ind,G}}\). The remainder of this paper is structured as follows. We describe Prolog\(^{\text{Ind,G}}\) in the next section. Section 3 describes the new semantics. Section 4 concludes the paper.

2 An Overview of Prolog\(^{\text{Ind,G}}\)

Our language is a variant of the level 0/1 prover in \([10]\) extended with simple inductions. Therefore, we closely follow their presentation in \([10]\). We assume that a program – a set of definition clauses \( D \) – is given. We have two kinds of goals given by \( G \)- and \( D \)-formulas below:

\[
G ::= \top | \bot | \text{nat}(x) | A | G \land G | \exists x \ G \\
\]

\[
D ::= \top | \bot | \text{nat}(x) | A | D \land D | \exists x \ D | \forall x \ D | G \supset D | \text{nat}(x) \supset G \\
\]

In the rules above, \( A \) represents an atomic formula.

The formulas in this languages are divided into level-0 goals, given by \( G \) above, and level-1 goals, given by \( D \). We assume that atoms are partitioned level-0 atoms and level-1 atoms. Goal formulas can be level-0 or level-1 formulas, and in a definition \( A \triangleq B \), \( A \) and \( B \) can be level-0 or level-1 formulas, provided that \( \text{level}(A) \geq \text{level}(B) \).

Proving Level-0 formulas and Level-1 formulas is similar to proving goal formulas in Prolog. However, there are some major differences:
• when the Level-1 prover meets the implication $G \supset D$ where $G$ is not $\text{nat}(x)$, it attempts to solve $G$ (in level-0 mode). If $G$ is solvable with all the possible answer substitutions $\Sigma_1, \ldots, \Sigma_n$, then the Level-1 prover checks that, for every substitution $\Sigma_i$, $D\Sigma_i$ holds. If Level-0 finitely fails, the implication is proved.

• when the Level-1 prover meets the implication $\text{nat}(x) \supset G$, the choices for $x$ can be infinite. Therefore the machine needs to prove $G$ using induction (in induction mode). In induction mode, the machine attempts to decompose the induction hypothesis $G(x/n)$ (in level-0 submode) into a set atomic formulas $A$. Then it attempts to solve $G(x/n+1)$ (in level-1 submode ) relative to $A$. If $G(x/n + 1)$ is solvable with respect to $G(x/n)$ with an (partial) answer substitution $\Delta_n$, then the machine concludes that $G(x/k)$ holds with an (total) answer substitution $\Delta_k \ldots \Delta_0$ (i.e., by composing answer substitutions) for each natural number $k$.

We will present the standard operational semantics for this language as inference rules [1]. Below the notation $G : \mathcal{G}$ denotes $\{G\} \cup \mathcal{G}$. Note that execution alternates between two phases: the left rules phase and the right rules phase. In this fragment, all the left rules (excluding the defL in in) are invertible and therefore the left-rules (excluding the defL) take precedence over the right rules. Note that our semantics is a lazy version of the semantics of level 0/1 prover in the sense that an answer substitution is applied as lazily as possible. Below, the proof procedure for some formula returns a final run $\Sigma$ in normal mode and a final run $\Delta$ in induction mode. Note that it is not always possible to obtain the final run due to the presence of induction. In such a case, we assume that the machine returns a Failure.

Definition 1. Let $\sigma, \delta$ be answer substitutions, let $G, D$ be a goal, let $\mathcal{G}$ be a set of $G$-formulas. Then the task of

• proving $D$ from $\emptyset$ (empty premise) with respect to $\sigma, D$ and returns a total run $\Sigma - \text{pv}(l_1, \sigma, \emptyset, D, \Sigma) - \%$ in level 1,

• proving $D$ from $G : \mathcal{G}$ with respect to $\sigma, D$ and returns a total run $\Sigma - \text{pv}(l_0, \sigma, G : \mathcal{G}, D, \Sigma) - \%$ in level 0,

• proving $G$ from $G : \mathcal{G}$ with respect to $\sigma, \delta, D$ and returns a partial run $\Delta - \text{pv}(i_0, \sigma, \delta, G : \mathcal{G}, G, \Delta) - \%$ induction mode, level 0
• proving $G$ from $G : \mathcal{G}$ with respect to $\sigma, \delta, \mathcal{D}$ and returns a partial run

$\Delta - pv(i_1, \sigma, \delta, G : \mathcal{G}, G, \Delta)$ % induction mode, level 1

are defined as follows:

1. $pv(l_0, \sigma, \perp : \mathcal{G} \vdash D, \sigma)$. % This is a success.
2. $pv(l_0, \sigma, \top : \mathcal{G} \vdash D, \Sigma)$ if $pv(l_0, \sigma, \mathcal{G} \vdash D, \Sigma)$. % $\top$ in the premise is redundant.
3. $pv(l_0, \sigma, A : \mathcal{G} \vdash D, \Sigma)$ if, for each $\theta$ which is the mgu($A\sigma, A'$) for some $A' \triangleq B \in \mathcal{D}$, $pv(l_0, \sigma\theta, B : \mathcal{G} \vdash D, \Sigma)$. % DefL rule
4. $pv(l_0, \sigma, \text{nat}(n) : \mathcal{G} \vdash G, \text{Failure})$ if $%$ invokes induction $pv(l_1, \sigma\{(n, 0)\}, \emptyset : \mathcal{G} \vdash G, \Sigma)$ % prove base case and $pv(i_0, \sigma\{(n, j)\}, \emptyset, \mathcal{G} \vdash G(n/n + 1, \Delta)$. % process $G$ if it contains a nonatomic formula. % In induction step, $\delta$ – a partial substitution – is initialized to an empty substitution. Failure means that it is not possible to obtain the final run.
5. $pv(l_0, \sigma, (G_0 \land G_1) : \mathcal{G} \vdash D, \Sigma)$. % Below is the description of the level-0 prover in induction phase
6. $pv(l_0, \sigma, \exists xG : \mathcal{G} \vdash D, \Sigma)$ if $pv(l_0, \sigma, [y/x]G : \mathcal{G} \vdash D, \Sigma)$ where $y$ is a new free variable. % Below is the description of the level-1 prover in induction phase
7. $pv(i_0, \sigma, \delta, A \vdash G, \Delta)$ if $pv(i_1, \sigma, \delta, A \vdash G, \Delta)$, % switch from $i_0$ to $i_1$.
8. $pv(i_0, \sigma, \delta, A : \mathcal{G} \vdash G, \Delta)$ if (a nonatomic $G$ is in $G$) and $pv(i_0, \sigma, \delta, A : \mathcal{G} \vdash G', \Delta)$, % process $\mathcal{G}$ if it contains a nonatomic formula.
9. $pv(i_0, \sigma, \delta, (G_0 \land G_1) : \mathcal{G} \vdash G, \Delta)$. % Below is the description of the level-1 prover in induction phase
10. $pv(i_0, \sigma, \delta, \exists xG_1 : \mathcal{G} \vdash G, \Delta)$ if $pv(i_0, \sigma, \delta, [y/x]G_1 : \mathcal{G} \vdash G, \Delta)$ where $y$ is a new free variable. % This is a success via induction hypothesis $A$.
(12) $pv(i_1, \sigma, \delta, A \vdash G_0 \land G_1, \Delta'_0 \cup \Delta'_1)$ if $pv(i_1, \sigma, \delta, A \vdash G_0, \Delta_0)$ and $pv(i_1, \sigma, \delta, A \vdash G_1, \Delta_1)$.

Here, the answer substitution $\Delta'_0$ is identical to $\Delta_0$ but locations of the form $loc(x)$ in $\Delta'_0$ are adjusted to new locations properly. Similarly for $\Delta'_1$.

(13) $pv(i_1, \sigma, \delta, A \vdash \exists x G, \Delta)$ if $pv(i_1, \sigma, \delta_1, A \vdash [y/x]G, \Delta)$ where $y$ is a new free variable, $\delta_1 = \{(loc(x), t)\} \{(y, t)\}$ and $t$ is a term. Note that we assume that $loc(x)$ represents a unique location in the sequent.

% Below is the description of the level-1 prover

(14) $pv(l_1, \sigma, \emptyset \vdash \top, \sigma)$. % solving a true goal

(15) $pv(l_1, \sigma, \emptyset \vdash A, \Sigma)$ if $A' \triangleright B \in D$ and $A' \theta = A\sigma$ and $pv(l_1, \sigma\theta, \emptyset \vdash B, \Sigma)$. % DefR

(16) $pv(l_1, \sigma, \emptyset \vdash D_0 \land D_1, \Sigma_0 \cup \Sigma_1)$ if $pv(l_1, \sigma, G \vdash D_0, \Sigma_0)$ and $pv(l_1, \sigma, G \vdash D_1, \Sigma_1)$. % conjunctive goals

(17) $pv(l_1, \sigma, \emptyset \vdash G \supset D, \Sigma)$ if $pv(l_0, \sigma, G \vdash D, \Sigma)$. % switch from level 1 to level 0

(18) $pv(l_1, \sigma, \emptyset \vdash \forall x D, \Sigma)$ if $pv(l_1, \sigma, \emptyset \vdash [y/x]D, \Sigma)$ where $y$ is a new free variable.

(19) $pv(l_1, \sigma, \emptyset \vdash \exists x D, \Sigma)$ if $pv(l_1, \sigma \sigma_1, \emptyset \vdash [y/x]G, \Sigma)$ where $y$ is a new free variable, $\sigma_1 = \{(y, t)\}$ and $t$ is a term.

The following is a proof tree (from bottom up) of the example given in Section 1. Note that a proof tree is represented as a list. Now, a proof tree of a proof formula is a list of tuples of the form $\langle E, \Sigma, Ch \rangle$ where $E$ is a proof formula, $\Sigma$ is a final run for $E$, and $Ch$ is a list of the form $i_1 :: \ldots :: i_n :: nil$ where each $i_k$ is the address of its $k$th child (actually the distance to $E$'s $k$th child in the proof tree).

% base case

\[
\text{l}_1, \{(h_0, 0), (w_0, 1)\}, \emptyset \vdash \top, \Sigma, \text{nil} \text{ \% success} \\
\text{l}_1, \{(h_0, 0), (w_0, 1)\}, \emptyset \vdash \text{fact}(h_0, w_0), \Sigma, \text{1::nil} \text{ \% defR}
\]
3 An Alternative Operational Semantics

Adding game semantics requires some changes to the previous execution model. To be precise, our new execution model – adapted from [2] – solves the goal relative to the program using the proof tree built in the proof search.

To be precise, execution proceeds in two different phases: normal phase and induction phase. In normal phase, execution simply follows the proof tree because the proof tree encodes all the possible total runs. In induction phase, things are more complicated. Note that the proof tree in induction mode encodes only the partial run (from \(i\)th inductive step to \(i+1\)th inductive step). Therefore, a total run must be obtained from composing all the partial runs, not from the proof tree.

In addition, to deal with the universally quantified goals properly, the execution needs to maintain an input substitution \(F\) of the form \(\{y_0/c_0, \ldots, y_n/c_n\}\) where each \(y_i\) is a variable introduced by a universally quantified goal in the proof phase and each \(c_i\) is a user input during the execution phase.

**Definition 2.** Let \(L\) be a fixed proof tree. Let \(i\) be an index to a proof tree and let \(F\) be an input substitution. In addition, let \(\sigma\) be an answer substitution,
let \( \Delta \) be an answer substitution (obtained from composing induction steps). Then executing \( L_i \) (the \( i \) element in \( L \)) with \( F \) in normal phase – written as \( ex(i, F) \) – and executing \( G \) with \( \sigma, \Delta, F \) in induction phase – written as \( ex(ind, \sigma, \Delta, \emptyset \vdash G, F) \) – are defined as follows:

(1) \( ex(i, F) \) if \( L_i = (E, \text{nil}) \). % no child. This is a success.

(2) \( ex(i, F) \) if \( L_i = ((l_1, \sigma, \emptyset, D_0 \land D_1, \Sigma), m :: 1 :: \text{nil}) \) and
\( ex(i - m, F) \) and % execute \( D_0 \)
\( ex(i - 1, F) \). % execute \( D_1 \)

(3) \( ex(i, F) \) if \( L_i = ((l_1, \sigma, G, \forall x D, \Sigma), 1 :: \text{nil}) \) and
\( L_{i-1} = ((l_1, \sigma, G, [y/x] D, \Sigma), \_ \) and
% read a user input
\( \text{read}(r) \) % read \( y \) and
\( ex(i - 1, F \cup \{y/c\}) \) % update \( F \) for universal quantifiers
where \( c \) is the user input (the value stored in \( r \)).

(4) \( ex(i, F) \) if \( L_i = ((l_0, \sigma, A : G, D, \Sigma), i_1 :: \ldots :: i_n :: \text{nil}) \) and
choose a \( i_k \) such that \( L_{i-k} = ((l_0, \sigma \theta_k, B, G, D, \Sigma), \_ \) and
\( F \) and \( \theta_k \) agree on the variables appearing in \( F \)
and \( ex(i - i_k, F) \). % choose a correct one using \( F \) among many paths in defL

(5) \( ex(i, F) \) if \( L_i = ((l_0, \sigma, \text{nat}(n) \vdash G, \text{Failure}), p :: q :: \text{nil}) \) and
\( L_{i-p} = ((l_1, \sigma \{(n, 0)\}, \emptyset \vdash G, \Sigma_B), \_ \) and % base case
\( L_{i-q} = ((l_0, \sigma \{(n, j)\}, \emptyset, G \vdash G(n/n + 1), \Delta), \_ \) and % induction step
\( ex(ind, \sigma, \Delta_{\text{total}}, \emptyset \vdash G, F) \) % run in induction mode
where \( k = F(n) \)
and \( \Delta_{\text{total}} = (\Delta \|(j, k - 1) \ldots \Delta \|(j, 0) \Sigma_B)) \) (j, k - 1) represents a total run for \( G \)

(6) \( ex(i, F) \) if \( L_i = ((l_1, \sigma, \emptyset \vdash \exists x D), 1 :: \text{nil}) \) and
\( L_{i-1} = ((l_1, \sigma \{(y, t)\}, \emptyset \vdash [y/x] D), \_ \)
and (print \( x = y \sigma F \)) and \( ex(i - 1, F) \). Hence the value of \( x \) is \( y \)
imstantiated by \( F \) and \( \sigma \).

(7) \( ex(i, F) \) if \( L_i = (E, 1 :: \text{nil}) \) and \( ex(i - 1, F) \). % otherwise

(8) \( ex(ind, \sigma, \Delta, \emptyset \vdash A, F) \). % success in induction mode
ex(ind, σ, Δ, ∅ ⊢ G₀ ∧ G₁, F) if
ex(ind, σ, Δ, ∅ ⊢ G₀, F) and % execute the first goal.
ex(ind, σ, Δ, ∅ ⊢ G₁, F). % execute the second goal.

ex(ind, σ, Δ, ∅ ⊢ ∃xG, F) if (print x = t) and
ex(ind, σ, Δ, ∅ ⊢ [t/x]G, F)
where t = loc(x) ΔσF. % apply Δ, σ and then F to loc(x).

Initially, σ, F are empty substitutions.
In the above, Δtotal = ((Δ| (j, k − 1) . . . Δ| (j, 0) Σ_B))| (−k + 1) is used to correctly obtain a total run for G. To be precise, the notation Δ|(j, i) is used

- to rename each variable wᵣ to wᵣ+im,
- to replace j with i

where m is the number of existentially quantified variables in G. Thus the composition Δ|(j, k−1) . . . Δ|(j, 0) Σ_B contains all the answer substitutions obtained in inductive steps up to the number k. Thus it contains all the answer substitutions for km variables. Then to produce correct answers in solving G, we must undo the renaming via |(−k + 1), deleting unnecessary answer substitutions. Note that each Δ|(j, i) may contain location variables of the form loc(x) and we assume that loc(x) is adjusted properly in obtaining Δtotal.

The following is an execution sequence of the goal ∀x(nat(x) ⊃ ∃y fib(x, y)) using the proof tree above. We assume that the user chooses 3 for x. Note that the last component represents F.

% execution (from bottom up)

ind, i₀, ∅, Δtotal, fact(h₀ + 1, 6), 1::nil % success, print z = 6.
ind, i₀, ∅, Δtotal, ∃z fact(h₀ + 1, z), 1::nil % ∃-L

l₀, ∅, nat(h₀) ⊢ ∃y fib(h₀, y), ∅ 5::1::nil, {(h₀, 3)} % defL
l₁, ∅, ∅ ⊢ nat(h₀) ⊃ ∃y fact(h₀, y), ∅ 5::1::nil, {(h₀, 3)} % the user input is 3.
update F
l₁, ∅, ∅ ⊢ ∀x(nat(x) ⊃ ∃y fact(x, y)), ∅ 5::1::nil, ∅ % ∀-R

In the above, Δtotal is obtained as follows:
(1) From the base case in the proof tree, we obtain $\Sigma_B = \{(h_0, 0), (w_0, 1)\}$.

(2) From the inductive case in the proof tree, we obtain $\{(h_0, j)\}$ and a run $\Delta = \{(loc(z), (j+1)w_0), (w_1, (j+1)w_0)\}$.

(3) Then $\Delta|(j, i) = \{(loc(z), (i+1)w_i), (w_{i+1}, (i+1)w_i)\}$

(4) $\Delta|(j, 2) \ldots \Delta|(j, 0) \Sigma_B = \{(w_3, (2+1)w_2)\}\{(w_2, (1+1)w_1)\}\{(w_1, 1w_0)\}\{(w_0, 1)\} = \{(w_3, 6), (w_2, 2), (w_1, 1), (w_0, 1)\}$. It also contains answer substitutions for $loc(z_0), \ldots$ which we will not show here.

(5) $\Delta_{total} = (\Delta|(j, 2) \ldots \Delta|(j, 0) \Sigma_B)|(j, -k + 1) = \{(w_1, 6)\}$

(6) In the above, for simplicity, we omit the answer substitutions for $loc(z)$ variables in $\Delta_{total}$.

4 Conclusion

In this paper, we have considered a new execution model for a subset of the level 0/1 prover, enhanced with simple inductions and game semantics. This new model is interesting in that it gives a logical status to the $read$ predicate in Prolog. We plan to connect our execution model to Japaridze’s Computability Logic [2, 3] in the near future.

References

[1] G. Kahn, “Natural Semantics”, In the 4th Annual Symposium on Theoretical Aspects of Computer Science, LNCS vol. 247, 1987.

[2] G. Japaridze, “Introduction to computability logic”, Annals of Pure and Applied Logic, vol.123, pp.1–99, 2003.

[3] G. Japaridze, “Sequential operators in computability logic”, Information and Computation, vol.206, No.12, pp.1443-1475, 2008.

[4] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov, “Uniform proofs as a foundation for logic programming”, Annals of Pure and Applied Logic, vol.51, pp.125–157, 1991.
[5] David Baelde, Andrew Gacek, Dale Miller, Gopalan Nadathur, and Alwen Tiu. A User Guide to Bedwyr, November 2006.

[6] Raymond McDowell and Dale Miller. A logic for reasoning with higher-order abstract syntax. In Proc. LICS 1997, pp. 434–445, IEEE Comp. Soc. Press, 1997.

[7] Peter Schroeder-Heister. Rules of definitional reflection. In Proc. LICS 1993, pages 222–232. IEEE Comp. Soc. Press, 1993.

[8] Alwen Tiu. A Logical Framework for Reasoning about Logical Specifications. PhD thesis, Pennsylvania State University, May 2004.

[9] Alwen Tiu. Model checking for π-calculus using proof search. In M. Abadi and L. de Alfaro, editors, CONCUR, volume 3653 of LNCS, pages 36–50. Springer, 2005.

[10] Alwen Tiu, Gopalan Nadathur, and Dale Miller. Mixing finite success and finite failure in an automated prover. In Proc. of ESHOL’05: Empirically Successful Automated Reasoning in Higher-Order Logics, pages 79 – 98, December 2005.