Interactive Logic Programming via Choice-Disjunctive Clauses

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

Abstract: Adding interaction to logic programming is an essential task. Expressive logics such as linear logic provide a theoretical basis for such a mechanism. Unfortunately, none of the existing linear logic languages can model interactions with the user. This is because they use provability as the sole basis for computation.

We propose to use the game semantics instead of provability as the basis for computation to allow for more active participation from the user. We illustrate our idea via Prolog⊕, an extension of Prolog with choice-disjunctive clauses.

keywords: interaction, logic programming, linear logic, computability logic.

1 Introduction

Representing interactive objects (lottery tickets, vending machines) in logic and logic programming requires interactive knowledgebases or interactive clauses. An interactive knowledgebase must be able to allow the user to select one among many alternatives. Expressive logics such as linear logic provide a theoretical basis for such a mechanism.

Unfortunately, none of the existing linear logic languages can model decision steps from the user. This deficiency is an outcome of using provability as the sole basis for executing logic programs. In the operational semantics based on provability such as uniform provability [3, 4, 5], solving a goal $G$ from the additive-disjunctive clause $D_0 \oplus D_1$ simply terminates with a success if $G$ is solvable from both $D_0$ and $D_1$. This semantics, $pv$, is shown below:

$$pv(D_0 \oplus D_1, G) \text{ if } pv(D_0, G) \text{ and } pv(D_1, G)$$

This is unsatisfactory, as the action of choosing either $D_0$ or $D_1$ by the user – the declarative reading of $\oplus$ – is not present in this operational semantics.
Our approach in this paper involves a change of the operational semantics to allow for more active participation from the user. This is inspired by the game semantics of Japaridze [1]. Solving a goal $G$ from the choice-disjunctive clause $D_0 \oplus D_1$ now has the following operational semantics:

$$ex(D_0 \oplus D_1, G) \text{ if } read(i) \text{ and } ex(D_i, G) \text{ and } pv(D_j, G)$$

where $i (= 0 \text{ or } 1)$ is chosen by the user and $j$ is $(i + 1) \mod 2$. In the above semantics, the system requests the user to choose $i$ and then proceeds with solving both the chosen goal, $G_i$, and the unchosen goal, $G_j$. Both executions must succeed. It is worth noting that solving $G_j$ must proceed using $pv$ rather than $ex$ to disallow further interactions with the user. It can be easily seen that our new semantics has the advantage over the old semantics: the former respects the declarative reading of $\&$ without losing completeness.

As an illustration of this approach, let us consider a BMW car dealer web page where you can get the information for BMW models you choose. For an engine, you can have a gasoline model or a diesel one. For a doortype, you can have a 2door or a 4door. This is provided by the following definition:

\[
\begin{align*}
!2\text{door} \oplus !4\text{door}.
!\text{diesel} \oplus !\text{gas}.
!\text{bmw}(120d) & : - 2\text{door} \otimes \text{diesel}.
!\text{bmw}(120) & : - 2\text{door} \otimes \text{gas}.
!\text{bmw}(320d) & : - 4\text{door} \otimes \text{diesel}.
!\text{bmw}(320) & : - 4\text{door} \otimes \text{gas}.
\end{align*}
\]

Here, $: -$ represents reverse implication. The definition above consists of reusable resources, denoted by $!$. As a particular example, consider a goal task $\exists x \text{ bmw}(x)$. This goal would simply terminate with no interactions from the user in the context of [3] as this goal is solvable. However, in our context, execution proceeds as follows: the system requests the user to select a particular engine model and a doortype. After they – say, $2\text{door, diesel}$ – is selected, execution eventually terminates with $x = 120d$.

As seen from the example above, choice-disjunctive clauses can be used to model interactive decision tasks.

To present our idea as simple as possible, this paper focuses on muprolog, which is a variant of a subset of Lolli[3]. The former can be obtained from
the latter by (a) disallowing linear context and \& in the clauses, (b) allowing only \(\otimes\) in goal formulas, and (c) allowing \(\oplus\) in the clauses. Prolog\(\oplus\) can also be seen as an extension of Prolog with choice-disjunctive clauses, as \(\otimes\) in Prolog\(\oplus\) corresponds to \(\land\) of Prolog.

In this paper we present the syntax and semantics of this extended language, show some examples of its use. The remainder of this paper is structured as follows. We describe Prolog\(\oplus\) based on a first-order clauses in the next section and Section 3. In Section 4 we present some examples of Prolog\(\oplus\). Section 5 concludes the paper.

2 Prolog\(\oplus\) and Its Proof Procedure

The extended language is a version of Horn clauses with choice-disjunctive clauses. It is described by \(G\)-, \(C\)- and \(D\)-formulas given by the syntax rules below:

\[
G ::= A \mid G \otimes G \mid \exists x \ G \\
C ::= A \mid G \supset A \mid \forall x \ C \\
D ::= !C \mid D \oplus D
\]

In the rules above, \(A\) represents an atomic formula. A \(C\)-formula is called a Horn clause and a \(D\)-formula is called a choice-disjunctive clause.

In the transition system to be considered, \(G\)-formulas will function as queries and a list of \(D\)-formulas will constitute a program. We will present a proof procedure for this language as sequent system. The rules for proving queries in our language are based on two different phases. The first phase is that of processing choice-disjunctive clauses, while the second phase is that of proving traditional Prolog based on uniform provability [3, 5]. Note that choice-disjunctive clauses are processed first via \(pv_D^\uparrow\). Then, execution in the second phase proceeds just like traditional logic programming. To be specific, execution in the second phase alternates between two subphases: the goal reduction subphase via \(pv_G\) (one without a distinguished clause) and the backchaining subphase via \(pv_D^\downarrow\) (one with a distinguished clause). Below in the notation \(pv_D^\downarrow(D, \mathcal{P}, G)\), the \(D\) formula is a distinguished formula (marked for backchaining). The symbol \(::\) is a list constructor.

Definition 1. Let \(G\) be a goal and let \(\Delta\) be a list of \(D\)-formulas and let \(\mathcal{P}\)
be a set of Horn clauses. Then the task of proving \( G \) from \( \Delta - pv(\Delta, G) \) – is defined as follows:

1. \( pv(\Delta, G) \) if \( pv_D^\uparrow(\text{nil}, \Delta, G) \).
2. \( pv_D^\uparrow(\mathcal{P}, !C :: \Delta, G) \) if \( pv_D^\uparrow(\{!C\} \cup \mathcal{P}, \Delta, G) \).
3. \( pv_D^\uparrow(\mathcal{P}, (D_0 \oplus D_1) :: \Delta, G) \) if \( pv_D^\uparrow(\mathcal{P}, D_0 :: \Delta, G) \) and \( pv_D^\uparrow(\mathcal{P}, D_1 :: \Delta, G) \).
4. \( pv_D^\uparrow(\mathcal{P}, \text{nil}, G) \) if \( pv_G(\mathcal{P}, G) \). % switch to goal reduction mode
5. \( pv_D^\uparrow(A, \mathcal{P}, A) \). % This is a success.
6. \( pv_D^\uparrow((G_0 \supset A), \mathcal{P}, A) \) if \( pv_G(\mathcal{P}, G_0) \).
7. \( pv_D^\uparrow(\forall x \mathcal{D}, \mathcal{P}, A) \) if \( pv_D^\uparrow([t/x] \mathcal{D}, \mathcal{P}, A) \).
8. \( pv_G(\mathcal{P}, A) \) if \( D \in \mathcal{P} \) and \( pv_D^\uparrow(D, \mathcal{P}, A) \). % switch to backchaining mode
9. \( pv_G(\mathcal{P}, G_0 \otimes G_1) \) if \( pv_G(\mathcal{P}, G_0) \) and \( pv_G(\mathcal{P}, G_1) \).
10. \( pv_G(\mathcal{P}, \exists x G_0) \) if \( pv_G(\mathcal{P}, [t/x] G_0) \).

The notion of proof defined above is intuitive enough. The following theorem – whose proof is rather obvious from the discussion in [3] and from the completeness of the focused proof system – shows the connection to linear logic.

**Theorem 1** Let \( \Delta \) be a program and \( G \) be a goal in \( \text{Prolog}_\oplus \). The procedure \( pv(\Delta, G) \) is a success if and only if \( G \) follows from \( !\Delta \) in intuitionistic linear logic.

### 3 An Execution Model for \( \text{Prolog}_\oplus \)

We now present an execution model for \( \text{Prolog}_\oplus \). This execution model is identical to the proof procedure in the previous section, except for the way choice-disjunctive clauses are handled (Rule 3 below).
In the transition system to be considered, \( G \)-formulas will function as queries and a list of \( D \)-formulas will constitute a program. We will present an operational semantics for this language as before.

**Definition 2.** Let \( G \) be a goal and let \( \Delta \) be a list of \( D \)-formulas and let \( P \) be a set of Horn clauses. Then the task of executing \( G \) from \( \Delta \) – \( \text{ex}(\Delta, G) \) – is defined as follows:

1. \( \text{ex}(\Delta, G) \) if \( \text{ex}^{\uparrow}_D(\text{nil}, \Delta, G) \).
2. \( \text{ex}^{\uparrow}_D(P, !C :: \Delta, G) \) if \( \text{ex}^{\uparrow}_D(\{!C\} \cup P, \Delta, G) \).
3. \( \text{ex}^{\uparrow}_D(P, (D_0 \oplus D_1) :: \Delta, G) \) if \( \text{read}(i) \) and \( \text{ex}^{\uparrow}_D(P, D_i :: \Delta, G) \) and \( \text{pv}^{\uparrow}_D(P, D_j :: \Delta, G) \) where \( i (= 0 \text{ or } 1) \) is chosen by the user and \( j \) is \((i + 1) \mod 2\).
4. \( \text{ex}^{\uparrow}_D(P, \text{nil}, G) \) if \( \text{ex}_G(P, G) \). % switch to traditional logic programming
5. \( \text{ex}^{\downarrow}_D(A, P, A) \). % This is a success.
6. \( \text{ex}^{\downarrow}_D((G_0 \supset A), P, A) \) if \( \text{ex}_G(P, G_0) \).
7. \( \text{ex}^{\downarrow}_D(\forall x D, P, A) \) if \( \text{ex}^{\downarrow}_D([t/x]D, P, A) \).
8. \( \text{ex}_G(P, A) \) if \( D \in P \) and \( \text{ex}^{\downarrow}_D(D, P, A) \).
9. \( \text{ex}_G(P, G_0 \otimes G_1) \) if \( \text{ex}_G(P, G_0) \) and \( \text{ex}_G(P, G_1) \).
10. \( \text{ex}_G(P, \exists x G_0) \) if \( \text{ex}_G(P, [t/x]G_0) \).

In the above rules, the symbol \( \oplus \) provides choice operations.

The following theorem – whose proof is easily obtained from the fact that the modified rule does not affect the soundness and completeness and can be shown using an induction on the length of derivations – shows the connection between our operational semantics and linear logic.

**Theorem 2** Let \( \Delta \) be a program and \( G \) be a goal in Prolog\(\oplus\). Executing \( \langle \Delta, G \rangle \) – \( \text{ex}(\Delta, G) \) – terminates with a success if and only if \( G \) follows from \( !\delta \) in
4 Examples

As an example, let us consider the following interactive database which contains tuition information for some university. The following tuition charges are in effect for this year: $40K for medical students, $30K for engineering and $20K for economics.

\[ \text{med} \oplus \text{eng} \oplus \text{eco}. \]
\[ !\text{tuition}(40K) : -\text{med}. \]
\[ !\text{tuition}(30K) : -\text{eng}. \]
\[ !\text{tuition}(20K) : -\text{eco}. \]

Consider a goal \( \exists x \ \text{tuition}(x) \). The system in Section 3 requests the user to select the current major. After the major – say, med – is selected, the system eventually produces the amount, \( i.e. \), \( x = 40K \).

5 Conclusion

In this paper, we have considered an extension to Prolog with choice-disjunctive clauses in linear logic. This extension allows clauses of the form \( D_0 \oplus D_1 \) where \( D_0, D_1 \) are Horn clauses. In particular, these clauses make it possible for Prolog to model decision steps from the user.

At this stage, clauses of more complex forms such as \( ! (D \oplus D) \) are not allowed in our language. We plan to allow them in the future. We also plan to connect our execution model to Japaridze’s expressive Computability Logic \([1, 2]\) in the near future.

6 Acknowledgements

This work was supported by Dong-A University Research Fund.

References

[1] G. Japaridze, “Introduction to computability logic”, Annals of Pure and Applied Logic, vol.123, pp.1–99, 2003.
[2] G. Japaridze, “Sequential operators in computability logic”, Information and Computation, vol.206, No.12, pp.1443-1475, 2008.

[3] J. Hodas and D. Miller, “Logic Programming in a Fragment of Intuitionistic Linear Logic”, Information and Computation, vol.110, No.2, pp.327-365, 1994.

[4] D. Miller, “A logical analysis of modules in logic programming”, Journal of Logic Programming, vol.6, pp.79–108, 1989.

[5] 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.