Bounded Inputs for Logic Programming

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

Abstract: Adding I/O to logic programming is an essential task. Unfortunately, none of the existing logic languages can model bounded inputs. Executing the goal \texttt{read}(x)G from a program \mathcal{P} simply tries to unify \texttt{x} with the keyboard input. This is an unpredictable and unguided situation, because the user can type in anything.

We propose to modify the operational semantics above to allow for more controlled and more guided participation from the user. We illustrate our idea via Prolog$^B$, an extension of Prolog with bounded inputs.

keywords: input, logic programming, bounded input, read.

1 Introduction

Adding I/O to logic programming is an essential task. The input statement that has been used in logic programming has been restricted to the \texttt{read} statement. The \texttt{read} statement is of the form \texttt{read}(X) where \texttt{X} can have any value. Hence, it is a form of an unbounded and unbounded input statement. However, there are several situations where the system requires the user to choose one among many alternatives. In other words, they require a form of guided and bounded inputs. Examples are provided by most interactive systems such as airline ticketing systems.

The use of bounded inputs thus provides the user with a useful facility in representing interactive systems. For this, this paper introduces a new goal statement \texttt{read}(X,S)G where \texttt{X} is a variable and \texttt{S} is a set of terms. This has the following semantics:

$$\text{ex}(D, \text{read}(X, \{t_1, \ldots, t_n\})G) \text{ if } \text{ex}(D, [t_i/x]G)$$

where \texttt{i} is chosen by the user and \texttt{D} is a program. In the above definition, the system requests the user to choose \texttt{i} and then proceeds with solving \texttt{G} with the chosen term, \texttt{t_i}. It can be easily seen that our new statement has
the advantage over the old statement: the former reduces the human errors in most interactive systems.

As an illustration of this approach, let us consider a fast-food restaurant where you can have the hamburger set or the fishburger set. For a hamburger set, you can have a hamburger and a side-dish vegetable (onion or corn). For a fishburger set, you can have a fishburger and a side-dish vegetable (onion or corn). The menu is provided by the following definition:

\[ \text{price}(h, 3). \quad \% \text{hamburger, three dollars} \]
\[ \text{price}(f, 4). \quad \% \text{fishburger, four dollars} \]
\[ \text{price}(o, 1) \quad \% \text{onion, one dollar} \]
\[ \text{price}(c, 2) \quad \% \text{corn, two dollars} \]

As a particular example, consider a goal task

\[ \text{read}(x, \{h, f\}) \quad \text{read}(y, \{o, c\}) \quad \text{price}(x, W), \text{price}(y, Z), (U = W + Z). \]

In our context, execution proceeds as follows: the system requests the user to select a particular burger and a vegetable. After they are selected, the system computes the total price \( U \).

To present our idea as simple as possible, this paper focuses on 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 our language in the next section. In Section 3 we present some examples of Prolog\textsuperscript{BI}. Section 4 concludes the paper.

## 2 Prolog with Bounded Input

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

\[ G ::= \ A \mid G \land G \mid \exists x \ G \mid \text{read}(x) G \mid \text{read}(x, S) G \]
\[ D ::= \ A \mid G \supset A \mid \forall x \ D \mid D \land D \]

In the rules above, \( A \) represents an atomic formula, \( S \) is a set of terms. A \( D \)-formula is called a Horn clause, or simply a clause.
In the transition system to be considered, $G$-formulas will function as queries and a $D$-formula will constitute a program. We will present the standard operational semantics for this language as inference rules [1]. The rules for executing queries in our language are based on uniform provability [4, 6]. Note that execution alternates between two phases: the goal reduction phase and the backchaining phase.

**Definition 1.** Let $G$ be a goal and let $D$ be a program. Then executing $G$ from $D$ – written as $ex(D, G)$ – is defined as follows:

1. $bc(A, D, A)$. % This is a success.
2. $bc(G \supset A, D, A)$ if $ex(D, G)$.
3. $bc(\forall x D_0, D, A)$ if $bc([t/x]D_0, D, A)$.
4. $bc(D_0 \land D_1, D, A)$ if $bc(D_0, D, A)$.
5. $bc(D_0 \land D_1, D, A)$ if $bc(D_1, D, A)$.
6. $ex(D, A)$ if $bc(D, D, A)$.
7. $ex(D, G_0 \land G_1)$ if $ex(D, G_0)$ and $ex(D, G_1)$.
8. $ex(D, \exists x G)$ if $ex(D, [t/x]G)$. Typically, selecting the true term can be achieved via the unification process.
9. $ex(D, read(x)G)$ if $ex(D, [kbd/x]G)$ where $kbd$ is the keyboard input.
10. $ex(D, read(x, S)G)$ if $ex(D, [t_i/x]G)$ where $i$ is chosen by the user.

In the above rules, the symbols $read(x, S)$ provides choice operations by the user.

3 Examples

As an example, let us consider the following database which contains the today’s flight information for major airlines such as Panam and Delta airlines.
Consider a goal \( \text{read}(S) \ \text{read}(D) \ \exists dt \exists at \ \text{panam}(\text{paris}, \text{nice}, dt, at). \) This goal expresses the task of diagnosing whether the user has a flight in Panam to fly from \( S \) to \( D \) today. The system then requests the user to type in a particular source and a destination. Note that this goal is difficult to use and error-prone because the user may type in an invalid source or a destination.

Our \( \text{read} \) statement is useful to avoid this kind of human errors. Consider a goal

\[
\text{read}(S, \{\text{paris, nice}\}) \ \text{read}(D, \{\text{tokyo, london}\}) \ \exists dt \exists at \ \text{panam}(S, D, dt, at)
\]

Again, this goal expresses the task of diagnosing whether the user has a flight in Panam to fly from \( S \) to \( D \) today. Note that this goal is much easier and much safer to use. The system now requests the user to select among three – rather than type in – a particular source and destination. After they are selected, the system produces the departure and arrival time of the flight of the Panam airline.

4 Conclusion

In this paper, we have considered an extension to Prolog with bounded input statement. This extension allows goals of the form \( \text{read}(x, S)G \) where \( G \) is a goal. This goal makes it possible for Prolog to model decision steps from the user.

We plan to connect our execution model to Japaridze’s Computability Logic \[2, 3\] in the near future.
5 Acknowledgements

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

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] 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.

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

[6] 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.