Logic Programming with Macro Connectives

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

Abstract: Logic programming such as Prolog is often sequential and slow because each execution step processes only a single, micro connective. To fix this problem, we propose to use macro connectives as the means of improving both readability and performance.

keywords: Prolog, macro connectives, synthetic connectives.

1 Introduction

Modern imperative languages such as Java, Perl support macro connectives to improve readability and performance of a program. The switch statement is such an example. To be precise, the switch statement (n-ary branch) is redundant in the sense that it can be converted to the if-then-else (binary branch) statement. However, this switch statement has proven essential in many programs.

Unfortunately, the situation is quite different in logic and logic programming. For example, first-order logic (FOL) requires logical connectives ∧, ∨ to be restricted to binary. For example, ∧(A, B, C) ((A ∧ B ∧ C) in infix notation) must be written as either ∧(A, ∧(B, C)) or ∧(∧(A, B), C). Similarly, it requires $\vec{x}$ in $\forall \vec{x}$, $\exists \vec{x}$ to be a single variable where $\vec{x} = x_1, \ldots, x_n$. This restriction is rather unnatural and has some unpleasant consequences, known as syntactic bureaucracy.

- It increases the complexity of formulas and, therefore, makes formulas more difficult to read and write.
- It makes proof search more sequential and less parallel. It forces proof steps that are parallel in nature to be written in a sequential order.
- It makes (already highly nondeterministic) proof search less atomic and more nondeterministic.
To fix this problem of syntactic bureaucracy, we extend FOL to FOL $^+$ to include the following macro formulas (called generalized conjunction/disjunction, block universal/existential quantifiers, respectively):

- $\land(F_1, \ldots, F_n), \lor(F_1, \ldots, F_n)$ are formulas for $i = 2, 3, \ldots$
- $\exists \bar{x} F, \exists \bar{\bar{x}} F, \forall \bar{x} F, \forall \bar{\bar{x}} F$ are formulas if $F$ is a formula.

In the above (and in the sequel as well), $\bar{x}$ represents $x_1, \ldots, x_n$, $\bar{\bar{x}}$ represents $[x_1, \ldots, x_n]$ and $\bar{x}$ represents $\{x_1, \ldots, x_n\}$. The meaning of these formulas are the following:

- $\exists \bar{x} F$ is identical to $\exists x_1 \ldots \exists x_n F$.
- $\exists \bar{\bar{x}} F$ is identical to $\exists x_1 \ldots \exists x_n F$ with the additional constraint that, in the former, $x_1, \ldots, x_n$ must be processed consecutively in that order. $\exists \bar{\bar{x}} F$ is called a block sequential existential quantifier.
- $\exists \bar{x} F$ is identical to $\exists \bar{\bar{x}} F$ with the difference that, in the former $x_1, \ldots, x_n$ must be processed consecutively but in arbitrary order. $\exists \bar{x} F$ is called a block parallel existential quantifier.

$\forall \bar{x} F, \forall \bar{\bar{x}} F, \forall \bar{x} F$ are similarly defined.

A sequent calculus for FOL $^+$ can be easily obtained by extending the standard sequent rules of Gentzen’s LK for $\land, \lor, \forall, \exists$ with new synthetic rules. Thus, in the new calculus, a small consecutive local inference steps can be combined into a single synthetic step, thus making proof search more parallel and more deterministic.

In this paper, our focus is on applying this idea to logic programming for improved conciseness and improved performance.

For example, we adopt the following operational semantics for $\land$ and $\lor$.

- $ex(D, \land(G_1, \ldots, G_n))$ if $ex(D, G_1) \text{ pand } \ldots \text{ pand } ex(D, G_n)$
- $ex(D, \lor(G_1, \ldots, G_n))$ if $ex(D, G_1) \text{ por } \ldots \text{ por } ex(D, G_n)$

where $\lor$ represents classical disjunction, $\text{pand}$ represents a parallel conjunction\[4], and $\text{por}$ represents a parallel disjunction\[4].

This paper proposes Prolog$^{\text{macro}}$, an extension of Prolog with macro connectives. The remainder of this paper is structured as follows. We describe Prolog$^{\text{macro}}$ in the next section. Section\[3] concludes the paper.
2 The Language

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

\[
G ::= A \mid \land (G_1, \ldots, G_n) \mid \lor (G_1, \ldots, G_n) \mid \exists \vec{x}G \mid \exists \tilde{x}G \mid \exists \ddot{x}G
\]

\[
D ::= A \mid G \supset A \mid \forall \vec{x}D \mid \forall \tilde{x}D \mid \forall \ddot{x}D \mid \land (D_1, \ldots, D_n)
\]

In the rules above, \( A \) represents an atomic formula. A \( D \)-formula is called a Horn clause with macro connectives.

The logic programming paradigm such as Prolog was originally founded on the resolution method. But this approach was difficult to extend to richer logics. The use of sequent calculus allows us to overcome this limit. In particular, uniform proofs [7] allows us to execute logic programs in an efficient way by integrating two separate phases – the proof phase and the execution phase – into a single phase. We adopt this approach below.

Note that execution alternates between two phases: the goal-reduction phase and the backchaining phase. In the goal-reduction phase (denoted by \( ex(D, G) \)), the machine tries to solve a goal \( G \) from a clause \( D \) by simplifying \( G \). If \( G \) becomes an atom, the machine switches to the backchaining mode. In the backchaining mode (denoted by \( bc(D_1, D, A) \)), the machine tries to solve an atomic goal \( A \) by first reducing a Horn clause \( D_1 \) to simpler forms and then backchaining on the resulting clause (via rule (1) and (2)).

**Definition 1.** Let \( G \) be a goal and let \( D \) be a program. Then the notion of executing \( \langle D, G \rangle \) – \( ex(D, G) \) – is defined as follows:

1. \( bc(A, D, A) \). % This is a success.
2. \( bc((G_0 \supset A), D, A) \) if \( ex(D, G_0) \). % backchaining
3. \( bc(\land (D_1, \ldots, D_n), D, A) \) if \( bc(D_1, D, A) \) por \ldots por \( bc(D_n, D, A) \).
4. \( bc(\forall x_1, \ldots, x_nD_1, D, A) \) if \( bc(\forall x_2 \ldots x_n[t_1/x_1]D_1, D, A) \). Thus it processes only \( x_1 \).
5. \( bc(\forall \ddot{x}D_1, D, A) \) if \( bc([t_1/x_1] \ldots [t_n/x_n]D_1, D, A) \) where \( t_1, \ldots, t_n \) are terms. Thus, the variables \( x_1, \ldots, x_n \) are processed both consecutively and sequentially.

3
(6) \( bc(\forall x D_1, D, A) \) if \( bc([t_{i_1}/x_{i_1}] \ldots [t_{i_n}/x_{i_n}]D_1, D, A) \) where \( t_1, \ldots, t_n \) are terms. Thus, the variables \( x_1, \ldots, x_n \) are processed both consecutively and in parallel.

(7) \( ex(D, A) \) if \( bc(D, D, A) \). \% switch to backchaining mode

(8) \( ex(D, \land (G_1, \ldots, G_n)) \) if \( ex(D, G_1) \) \( pand \ldots pand \ ex(D, G_n) \).

(9) \( ex(D, \lor (G_1, \ldots, G_n)) \) if \( ex(D, G_1) \) \( por \ldots por \ ex(D, G_n) \).

(10) \( ex(D, \exists x_1, \ldots, x_n G) \) if \( ex(D, \exists x_2, \ldots, x_n[t_1/x_1]G) \) where \( t_1 \) is a term. Thus, it processes only \( x_1 \).

(11) \( ex(D, \exists \bar{x} G) \) if \( ex(D, [t_{i_1}/x_{i_1}] \ldots [t_{i_n}/x_{i_n}]G) \) where \( t_1, \ldots, t_n \) are terms. Thus, the variables \( x_1, \ldots, x_n \) are processed both consecutively and sequentially.

(12) \( ex(D, \exists \bar{x} G) \) if \( ex(D, [t_{i_1}/x_{i_1}] \ldots [t_{i_n}/x_{i_n}]G) \) where \( t_1, \ldots, t_n \) are terms. Thus, the variables \( x_1, \ldots, x_n \) are processed both consecutively and in parallel.

These rules are straightforward to read.

As an example, consider the following specification for computing binomial coefficients, denoted by \( c(n, k, z) \).

\[
\begin{align*}
\forall N & \ c(N, 1, N) \ . \ % \ select \ one \ out \ of \ n \\
\forall N & \ c(N, N, 1) \ . \ % \ select \ n \ out \ of \ n \\
\forall \{N, K\} & \ c(N, K, 0) : - N < K. \\
\forall \{N, K, W, Z\} & \ c(N, K, W + Z) : - c(N - 1, K - 1, W) \land c(N - 1, K, Z).
\end{align*}
\]

The above program is a little simpler and more efficient than Prolog due to the use of block universal quantifiers. The correctness of the above program is guaranteed from the focalization property of traditional logic.

While it does not seem like much, it is easy to see that the benefits of using macro connectives will be substantial for highly complex formulas.
3 Conclusion

In this paper, we have considered an extension to Prolog\cite{1} with some macro connectives. This extension makes Prolog programs easier to read, write and execute.

Our macro connectives is a simple yet practical subset of a wider class of connectives called *synthetic* connectives. These synthetic connectives – proposed originally by Girard – is theoretically interesting and is based on the notion of focalization in linear logic. In the near future, we plan to investigate the possibility of including these synthetic connectives into logic programming.

References

[1] I. Bratko, “Prolog:programming for AI ”, Addison Wesley, 2001 (3rd edition).

[2] J.Y. Girard, “Linear Logic”, Theoretical Computer Science, vol.50, pp.1–102, 1987.

[3] J. Hodas and D. Miller, “Logic Programming in a Fragment of Intuitionistic Linear Logic”, Information and Computation, vol.110, pp.327–365, 1994.

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

[5] E. Komendantskaya and V. Komendantsky, “On uniform proof-theoretical operational semantics for logic programming”, In J.-Y. Beziau and A.Costa-Leite, editors, Perspectives on Universal Logic, pages 379–394. Polimetrica Publisher, 2007.

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

[7] 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.