ISSN 1392–124X INFORMATION TECHNOLOGY AND CONTROL, 2012, Vol. 41, No. 4

Forward Reasoning via Sequential Queries in Logic Programming

Keehang Kwon

DongA University, Department of Computer Engineering
Busan 604-714, Korea
E-mail: khkwon@dau.ac.kr

crossref http://dx.doi.org/10.5755/j01.itc.41.4.2364

Abstract. Most Prolog implementations are based on backward chaining techniques. However, there are many applications in which forward chaining ones are desirable such as in dynamic programming. In this paper, we first introduce a variant of a Prolog interpreter that computes interpolations and then introduce the notion of sequential queries. These two notions allow a combination of both forms of reasoning in Prolog.

Keywords: backward chaining; forward chaining; sequential queries.

1. Introduction

The procedural intepretation of the commas between Prolog goals as sequentiality has been popular. However, this intepretation is purely wrong and has very little to do with its logical foundation. Consequently this generates many well-known semantic problems, *i.e.*, the mismatch between the declarative meaning and the procedural meaning of the commas between goals.

There has been much interest in giving a $logical$ account of the sequential construct in goals in the context of logic programming. For example, Computability Logic(CL) [1, 2] uses the logic of task to give a logical foundation for additive(choice) sequentiality [2] and multiplicative sequentiality [4] .

In this paper we are concerned with applying sequential queries introduced in [4] to generating lemmas and forward reasoning in computation. A sequence of well chosen lemmas is important in presenting a proof in mathematical proof. Producing such lemmas can also be important in logic programming as lemmas can improve the search for proof. Unfortunately, the underlying proofs using backward chaining in logic programming are usually lemma-free, and lemmas do not have a role in improving the search for proofs.

Our solution to the above problem is the notion of $sequential$ queries introduced in [4]. To be specific, proving the query G using some lemma G_1 from a logic program $\mathcal{P}$ can be represented via a sequential-conjuctive goal of the form $G_1 \prec G$. Solving this goal corresponds to first attempting to prove the lemma G_1 from $\mathcal{P}$ and then G from $\mathcal{P}, G_1$. Using lemmas in this fashion can be considered as a form of (*guided*) forward reasoning in the sense that a knowledge base grows during computation.

In the logic programming setting, there are some complications due to the indefinite goal formulas of the form $\exists xG$ (or $G \vee G$ in some versions of Prolog). In such a case, its definite version $[t/x]G_1$ needs to be extracted from the proof of G_1 from $\mathcal{P}$. We call this process *interpolation*. This process converts an indefinite goal formula to a set of atoms by producing a set of atomic instances of the goal formula. This process is integrated into the Prolog interpreter shown in Section 2.

Another approach to generating lemmas and forward reasoning in logic programming is *tabled logic programming* [8–10, 5]. The idea behind tabling is the following: goals encountered in solving a query are maintained in a table. If a goal is re-encountered, solving that goal reuses information from the table rather than re-solving it. This idea supports dynamic programming and termination of some nonterminating Prolog programs.

Compared to ours, their approaches have the following disadvantages:

- They restrict lemmas to be subgoals of the original query. We do not have such a restriction: the lemmas in our language can be any formulas.
- In their approaches, due to the lack of sequential goals, it is *not* possible for the user to specify and guide a list of lemmas that should be tabled. As a consequence, they store all the previously proved atoms, regardless of whether these atoms have a role in improving the search for proofs.

In a sense, our approach supports *guided* tabling.

This paper proposes an extension of Prolog with a limited form of sequentiality. That is, sequential goals are allowed only in top-level Prolog goals.

We make this choice to provide a smooth transition from Prolog to this new language from the perspective of implementation overheads. To be specific, a sequential-conjunctive goal is of the form $G_1 \prec G$ where G_1, G are Prolog goals to be executed sequentially. Proving this goal with respect to $\mathcal{P}$ has the following intended semantics: prove G_1 with respect to $\mathcal{P}$ and then prove G with respect to $\mathcal{P} \cup G_1'$ where G_1' is a set of atomic instances of G_1.

We also provide an iterative goal of the form $\prec_x^{i..j} G$ where G is a goal, x is a variable, and i, j are natural numbers. This goal also appears in [7]. Proving this goal has the following intended semantics: iterate G with x ranging over all elements in the set $\{i, i+1, \ldots, j\}$.

In this paper we present the syntax and semantics of this extended language, show some examples of its use and study the interactions among the newly added constructs. To be specific, we describe an operational semantics for Prolog sequential goals in terms of CL, which can be viewed operationally as extending the logic program from the logic of truth to the logic of task. Sequential queries introduce an element of forward reasoning into logic programming, as we shall see in Section 4. The sequential queries allow us to program in a natural and declarative way many applications based on dynamic programming. An example is memoization in logic programming.

The remainder of this paper is structured as follows. We describe a new interpreter with interpolation for first-order Horn clauses in the next section and its extension in Section 3. In Section 4, we present some examples of SProlog. Section 5 concludes the paper.

2. The Prolog Interpreter that computes interpolation

The Prolog language is based on Horn clauses. It is described by G- and D-formulas given by the syntax rules below:

$$\begin{array}{rl} G ::= & A \mid G \wedge G \mid \exists x\, G \\ D ::= & A \mid G \supset A \mid \forall x\, D \mid D \wedge D \end{array}$$

In the rules above, A represents an atomic formula. A D-formula is called a Horn clause, or a definite clause.

In the transition system to be considered, G-formulas will function as queries and a set of D-formulas will constitute a program. We will present the standard operational semantics for this language as inference rules [3].

The rules for executing queries in our language are based on uniform provability [6].

Definition 1. Let G be a goal and let $\mathcal{P}$ be a program. Then the notion of $intp_o(\mathcal{P}, G)$ – proving G from $\mathcal{P}$ – is defined as follows:

(1) $intp_o(\mathcal{P}, A)$ if A is identical to an instance of a program clause in $\mathcal{P}$.

(2) $intp_o(\mathcal{P}, A)$ if A is an instance of a program clause in $\mathcal{P}$ of the form $G_1 \supset A$ and $intp_o(\mathcal{P}, G_1)$.

(3) $intp_o(\mathcal{P}, G_1 \wedge G_2)$ if $intp_o(\mathcal{P}, G_1)$ and $intp_o(\mathcal{P}, G_2)$.

(4) $intp_o(\mathcal{P}, \exists x G_1)$ if $intp_o(\mathcal{P}, [t/x]G_1)$.

Now we will present another operational semantics for this language based on CL. Further, this is a new form of a Prolog interpreter that performs additional work: that of extracting its interpolations as well. Basically, the set of interpolations is a table that consists of all atomic instances of a goal G that are on the goal side of the proof.

To be specific, we encode such inference rules as theories in the logic of task, *i.e.*, a simple variant of CL [1]. Below the expression $A\ sand\ B$ denotes a sequential conjunction of the task A and the task B and the expression $A\ pand\ B$ denotes a parallel conjunction of the task A and the task B.

Definition 2. Let G be a goal and let $\mathcal{P}$ be a program. Then the notion of $intp_G(\mathcal{P}, G, S)$ – proving G from $\mathcal{P}$ and converting G to a set of atoms S – is defined as follows:

(1) $intp_G(\mathcal{P}, A, \{A\})$ if A is identical to an instance of a program clause in $\mathcal{P}$.

(2) $intp_G(\mathcal{P}, A, \{A\})$ if A is an instance of a program clause in $\mathcal{P}$ of the form $G_1 \supset A\ pand$ $intp_G(\mathcal{P}, G_1, \_)$. Here _ denotes a don't-care condition.

(3) $intp_G(\mathcal{P}, G_1 \wedge G_2, S_1 \cup S_2)$ if $intp_G(\mathcal{P}, G_1, S_1)$ $pand$ $intp_G(\mathcal{P}, G_2, S_2)$.

(4) $intp_G(\mathcal{P}, \exists x G_1, S)$ if $intp_G(\mathcal{P}, [t/x]G_1, S)$.

It is easily observed that the notion of converting a goal to a set of atoms requires little extra overheads.

3. The Sequential Queries

The language we use is an expanded version of goal formulas with top-level sequential conjunctive goals. It is described by E-formulas given by the syntax rules below:

$$E ::= G \mid \prec_x^{i..j} E \mid E \prec E$$

In the rules above, a E-formula is called a sequential-conjunctive goal or an extended goal.

In the transition system to be considered, E-formulas will function as extended queries. We will present an operational semantics for this language.

Definition 3. Let E be an extended goal and let $\mathcal{P}$ be a program. Then the notion of $intp_E(\mathcal{P}, E, S)$ – proving E from $\mathcal{P}$ and converting E to a set of atoms S – is defined as follows:

(1) $intp_E(\mathcal{P}, G, S)$ if $intp_G(\mathcal{P}, G, S)$.

(2) $intp_E(\mathcal{P}, E_1 \prec E_2, S \cup T)$ if $intp_E(\mathcal{P}, E_1, S)$ $sand$ $intp_E(S \cup \mathcal{P}, E_2, T)$. Thus, the first goal E_1 serves as a lemma.

(3) $intp_E(\mathcal{P}, \prec_x^{i..i}\ E, \{\})$. Thus, this task terminates with a success.

(4) $intp_E(\mathcal{P}, \prec_x^{i..j}\ E, S \cup T)$ if $(i < j)$ $pand$ $(intp_G(\mathcal{P}, [i/x]E, S)\ sand\ intp_E(S \cup \mathcal{P}, \prec_x^{(i+1)..j}\ E, T))$. Thus, the first goal $[i/x]E$ serves as a lemma.

In the above rules, the goal $E_1 \prec E_2$ provides sequential executions of instructions: it allows for solving the extended goal E_2 via the lemmas extracted from the proof of E_1.

4. Examples

Prolog permits the modification of the current logic program through the $assert$ and $retract$: assert D adds clause D to the program, while retract D removes D. To avoid the re-computation of previously solved goals, lemmas are memoized by applying assert to goals. It is well-known that the resulting program becomes obscure and complicated.

Our approach is immune to this problem. To see this, consider the following Horn clause specification for computing Fibonacci numbers.

$fib(0, 1)$. % base case
$fib(1, 1)$. % base case
$fib(X + 2, Y + W) :- fib(X, Y) \land$
$\qquad fib(X + 1, W)$.

Now consider the following query $query1$.

$query1$:
$\exists x fib(100, x)$

Solving this goal executes $query1$ in "backward chaining" with respect to the instructions in the Fibonacci program. If f_n denotes the nth Fibonacci number, then it is well-known that the size of the only proof of the goal $fib(n, f_n)$ is exponential in n.

The $query1$ has the problem of re-computing previously solved goals. To avoid this problem, our language allows us to employ sequential goals to store previously computed Fibonacci numbers. Consider the following query $query2$.

$query2$:
$\prec_i^{1..100} \exists y fib(i, y)$

Solving this query stores previously computed Fibonacci numbers in a forward chaining fashion. It tables a list of goal instances $fib(2, 1), fib(3, 2), \ldots,$ $fib(99,)$ which are extracted from a list of lemmas $\exists x fib(2, x)$, …, $\exists x fib(99, x)$. Note that each new lemma instances in the list can be computed using the previous lemma instances in constant time. In this query, there exists a proof of $fib(n, f_n)$ and that proof has a size proportional to n.

As a second example, consider the following Horn clause specification for computing binomial coefficients, denoted by $c(n, k, z)$.

$c(N, 1, N)$. % select one out of n
$c(N, N, 1)$. % select n out of n
$c(N, K, 0) :- \quad N < K$.
$c(N, K, W + Z) :- \quad c(N - 1, K - 1, W) \land$
$\qquad c(N - 1, K, Z)$.

Now suppose we want to compute the value of $c(100, 45)$ by invoking the following query: $\exists z\, c(100, 45, z)$. It is well-known that this query leads to redundant computations. Instead, the following query, which employs sequential goals to store previously computed numbers, will do the job in an efficient way.

$query3$:
$\prec_i^{1..100} \prec_j^{1..45} \exists z c(i, j, z)$

Solving this query requires "forward chaining" which is linear to the input size. It tables a list of lemma instances $c(1, 1, 1), c(1, 2, 0), \ldots, c(100, 1, 100), \ldots, c(100, 45,)$, as we desired.

5. Conclusion

In this paper, we have considered an extension to Prolog with some limited form of sequential goal formulas. This extension allows goals of the form $E_1 \prec E$ where E_1, E are a sequence of Prolog goals to be executed sequentially. These goals are particularly useful for sequential executions of goal tasks at the top-level and forward chaining. We plan to generalize our language so that sequential goals can be embedded in goal formulas.

Higher-order logic programming based on CL is another extension we have to consider in the future. The vehicle for this work is λProlog, a logic programming language supporting higher-order predicates introduced by Miller et al.[6]. λProlog extends Prolog in several directions: It provides the simply-typed higher-order terms as a data type and incorporates modules. We plan to interpret this language via CL, and add sequential queries to make it to perform more complex tasks.

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*, 2003, 123, 1–99.

[2] **G. Japaridze.** Sequential operators in computability logic. *Information and Computation*, 2008, 206, 1443–1475.

[3] **G. Kahn.** Natural Semantics. *the 4th Annual Symposium on Theoretical Aspects of Computer Science*, *Lecture Notes in Computer Science*,1987,247,22–39.

[4] **K. Kwon, S. Hur.** Adding sequential conjunctions to Prolog. *International Journal of Computer Technology and Applications*, 2010,1,1–3.

[5] **D. Miller, V. Nigam.** Incorporating tables into proof. *Computer Science Logic 2007*, *Lecture Notes in Computer Science*,2007,4646,466–480.

[6] **D. Miller, G. Nadathur, F. Pfenning, A. Scedrov.** Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*,1991,51,pp.125–157.

[7] **J. Schimpf.** Logical loops. *International Conference on Logic Programming*, 2002, 224–238.

[8] **A. Roychoudhury et al.** Justifying proofs using memo tables. *International Symposium on Principles and Practice of Declarative Programming*,2000,178–189.

[9] **H. Tamaki, T. Sato.** OLD resolution with tabulation. *International Conference on Logic Programming*, 1986,84–98.

[10] **D.S. Warren.** Memoing for logic programs. *Comm. of the ACM*,1992,35,93–111.

Received September 2011.