http://dx.doi.org/10.3745/KIPSTA.2012.19A.1.069

논리 프로그래밍을 위한 for-loop 구문

권 기 항[†] · 하 홍 표[††]

요 약

고전 논리나 선형 논리에 기반한 논리 프로그래밍에서는 순차적 작업이나 순차적 순환 작업을 표현하는 구문이 결여되어 있다. 최근 순차적 작업을 표현하는 구문 - $G_1 \cap G_2$ 이 제시되었는데 이는 Japaridze의 game 모델에 기반을 두고 있다. 본 논문에서는 $\cap_x^L G$ 의 형태를 갖고 있는 순차적 순환 작업을 제안하고 있다. 여기서 x는 변수이고, L은 리스트이고, G는 목표작업을 의미한다. 기호 $\cap_x^L$는 순차적 유한 정량자라고 불린다.
위 구문을 다음과 같은 작업을 의미한다: x에 L의 원소들의 값을 차례로 대입하여 순차적으로 반복 수행하시오.

키워드 : For-loop, Sequentiality, Iteration, Bounded Quantifier, Computability Logic

For-loop for Logic Programming

KeeHang, Kwon[†] · Hong Pyo, Ha[††]

ABSTRACT

Logic programming based on classical or linear logic has traditionally lacked devices for expressing sequential tasks and sequential iterative tasks. Expressing sequential goal tasks has been addressed by a recent proposal of sequential goals of the form $G_1 \cap G_2$ which is based on the game semantics of Japaridze.

This paper proposes sequential iterative goal formulas of the form $\cap_x^L G$ where G is a goal, x is a variable, and L is a list. $\cap_x^L$ is called a sequential bounded quantier. These goals allow us to specify the following task: sequentially iterate G with x ranging over all the elements of L.

Keywords : For-loop, Sequentiality, Iteration, Bounded Quantifier, Computability Logic

1. Introduction

Logic programming based on classical or linear logic has traditionally lacked mechanisms that permit some tasks to be (1) sequentially executed and (2) sequentially iterated.

These two deficiencies are an outcome of using a weak logic as the basis for logic programming. Lacking sequential conjunctive goals, logic programming such as Prolog has been relying on adopting SLD-like resolution as a proof procedure. The SLD-like resolution simulates sequential goals by sequentially processing parallel-conjunctive goals.

This approach is, however, unsatisfactory because declarative meanings of the resulting programs tend to be awkward to formulate lacking looping constructs, logic programming relies on recursion to perform iterative goal tasks. One of the disadvantages of this approach is that simple goal tasks have unnecessarily complex solutions in which recursion are used, when a simpler solution using iteration exists. The latter is easier to write and understand, and is closer to the original specification. Also, iteration can be implemented more efficiently than recursion.

To deal with the first deficiency, attention has been given to adding sequential goals on top of parallel-conjunctive goals to logic programming [6]. A sequential goal is of the form $G_1 \cap G_2$ where G_1, G_2 are goals. Executing this goal has the following intended semantics: solve G_1 and G_2 sequentially.

※ This paper was supported by the Dong-A University Research Fund.
† 정 회 원 : 동아대학교 컴퓨터공학과 교수
†† 준 회 원 : 서강대학교 컴퓨터공학과 석사과정
논문접수 : 2011년 12월 12일
심사완료 : 2012년 1월 11일

Both executions must succeed for executing $G_1 \cap G_2$ to succeed. This logic is built on the recent notion on the logic of task [3] and has the advantage that a well-defined declarative meaning can be obtained. This is in contrast to other approaches [1, 2] based on classical logic where no well-defined declarative meaning of sequentiality can be found.

To deal with the second deficiency, our approach in this paper involves the direct enrichment of the underlying logic in [6] to allow for sequential iterative goals. A sequential iterative goal is of the form $\cap_x^L G$ where G is a goal, x is a variable, and L is a list. Executing this goal has the following intended semantics: iterate G with x ranging over all elements of the list L. All executions must succeed for executing $\cap_x^L G$ to succeed.

An illustration of this facet is provided by the following definition of the relation which sequentially writes all the elements in a list:

write list(L) : - *write*("*List* : ") $\cap$ ($\cap_x^L$*write*(x)):
which replaces the tedious logic program
(with no declarative semantics for commas ingoals)

shown below:

write list(L) : - *write*("*List* : "),
*write list*1(L).
*write list*1([]).
*write list*1([X | T]) : - *write*(X),
*write list*1(T):

The body of the new definition above contains a sequential conjunction, denoted by $\cap$ and an iterative goal. As a particular example, solving the query *write_list*([1,2,3]) would result in solving the goal $\cap_x^{[1,2,3]}$ after writing *List* :. The given goal will succeed after writing 1, 2, 3 in sequence.

As seen from the example above, sequential iterative goals can be used to perform looping tasks. This paper proposes Prolog^{loop}, an extension of Prolog with sequential iterative operators in goal formulas.

As mentioned earlier, we also adopt sequential conjunctive goals, introduced in [6], which are of the form $G_1 \cap G_2$ where G_1, G_2 are goals. Executing this goal has the following intended semantics: execute both G_1 and G_2 in sequence. Both executions must succeed for executing $G_1 \cap G_2$ to succeed.

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^{loop} based on a first-order sequential Horn clauses in the next section. In Section 3, we present some examples of SProlog.Section 4 concludes the paper.

2. The Language

The language is a version of Horn clauses with sequential iterative goals. It is described by G- and D-formulas given by the syntax rules below:

G::= $A \mid G \wedge G \mid \exists\ x\ G \mid G \cap G \mid G$
D::= $A \mid G \supset A \mid \forall x D \mid D \wedge D$

In the rules above, x represents a variable, L represents a list of terms, and A represents an atomic formula. A D-formula is called a sequential Horn clause with sequential iterative goals.

In the transition system to be considered, G-formulas will function as queries and a set of D-formulas will constitute a set of instructions. For this reason, we refer to a G-formula as a query, to a set of D-formula as an instruction set.

We will present an operational semantics for this language as inference rules. To be specific, we encode such inference rules as theories in the (higher-order) logic of task, *i.e.*, a simple variant of Computability Logic [3]. 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.

These rules in fact depend on the top-level constructor in the expression, a property known as uniform provability [7, 8].

Definition 1. Let G be a goal and let $\mathcal{P}$ be a finite set of instructions. Then the notion of executing <$\mathcal{P}$, G> executing G relative to $\mathcal{P}$ is defined as follows:

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

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

(3) *exec*($\mathcal{P}$, $G_1 \wedge G_2$) if exec($\mathcal{P}$, G_1) pand
exec($\mathcal{P}$, G_2). Thus, the two goal tasks must be done in parallel and both tasks must succeed for the current task to succeed.

(4) *exec*($\mathcal{P}$, $\exists x G_1$) if (select the true term t) sand

exec($\mathcal{P}$, $[t/x]G_1$). Typically, selecting the true term can be achieved via the unification process.

(5) *exec*($\mathcal{P}$, $G_1 \cap G_2$) if *exec*($\mathcal{P}$, G_1) sand *exec*($\mathcal{P}$, G_2).

Thus, the two goal tasks must be done in sequence and both tasks must succeed for the current task to succeed.

(6) *exec*($\mathcal{P}$, $\cap_x^{nil} G$). The current execution terminates with a success.

(7) *exec*($\mathcal{P}$, $\cap_x^{[a1,\ldots,an]} G$) if *exec*($\mathcal{P}$, $[a_1/x]G$) *sand exec* ($\mathcal{P}$, $\cap_x^{[a2,\ldots,an]} G$).

In the above rules, the $\cap_x^L$ provides iterations: they allow for the repeated sequential conjunctive execution of the instructions.

An alternative yet tedious way to giving semantics of our language is by transformation to plain logic programming. For example, our loop construct $\cap_x^L$ can be defined by introducing a recursive auxiliary predicate such as *write_list*1 in Section 1. This method is discussed in detail in [2].

While the operational notion of execution defined above is intuitive enough and is quite similar to imperative languages, it is interesting to ask what its declarative meaning is. Clearly, sequential goals cannot be handled in classical logic or linear logic. Fortunately, this limitation can be overcome by a new declarative semantics for logical formulas, *i.e.*, using Japaridze's Computability Logic(CL)[3, 4, 5]. CL is a new semantic platform for reinterpreting logic as a theory of tasks. Formulas in CL stand for instructions that can carry out some tasks. Proving the soundness and completeness of our proof procedure with respect to CL is currently under way.

3. Example

An example is provided by the following the "factorial" program.

fact(0, 1): % base case
fact(X + 1, XY + Y) : - *fact*(X, Y).

Our language in Section 2 permits iterative goals. An example of this construct is provided by the program which does the following sequential tasks: output 10!, 11!, 12!,13!

sequentially:
*query*1 : $\cap_N^{[10,11,12,13]}$ % for i = 10 to 13 begin
(*fact*(N, O) $\cap$
write(N) $\cap$*write*('*factorial*:') $\cap$
write(O)) % for end

For example, consider a goal *query*1. Solving this goal has the effect of executing *query*1 with respect to the factorial program for four times.

Our language in Section 2 permits variables to appear in the list in iterative goals. These variables can be used only for controlling iteration and must be instantiated at run-time. An example of this construct is provided by the program which does the following sequential tasks: read a number N from the user, and then repeatedly output the factorials of the numbers from 1 to N.

*query*2 :.
(*read*(N) $\cap_x^{[1-N]}$% for x=1 to N begin (*fact*(x, O) $\cap$
write(x) $\cap$ *write*('*factorial is* :') $\cap$ *write*(O)) % for end

In the above, note that [1..N] is a shorthand notation for [1,2,$\cdots$,N].

4. Conclusion

In this paper, we have considered an extension to logic programming with sequential iterations in goals. This extension allows goals of the form $\cap_x^L G$ where G is a goal, x is a variable and L is a list of terms. These goals are particularly useful for the bounded looping executions of instructions, making logic programming more concise, more readable, and more friendly to imperative programmers.

Although sequential iterative goals do provide a significant gain in expressive elegance, some tasks with dynamic termination conditions cannot be expressed at all using them.

We plan to look at some variations [2] such as the *fromto* statements in the future to improve expressibility.

Regarding implementing our language, the handling of sequential bounded quantifications does not pose any major complications. The treatment of a goal of the form $G_1 \cap G_2$ that is indicated by the operational semantics essentially requires G_1 and G_2 to be processed sequentially, as is done in most Prolog implementations.

Reference

[1] K. Apt, "Arrays, bounded quantification and iteration in logic

and constraint logic programming", Science of Computer Programming, Vol.26, pp.133-148, 1996.
[2] J. Schimpf, "Logical loops", ICLP, pp.224-238, 2002.
[3] G. Japaridze, "Introduction to computability logic", Annals of Pure and Applied Logic, Vol.123, pp.1-99, 2003.
[4] G. Japaridze, "Sequential operators in computability logic", Information and Computation, Vol.206, No.12, pp.1443-1475, 2008.
[5] G. Japaridze, "A new face of the branching recurrence of computability logic", Applied Mathematics Letters (to appear).
[6] K. Kwon and S. Hur, "Adding sequential conjunctions to Prolog", IJCTA, Vol.1, No.1, pp.1-3, 2010.
[7] D. Miller, "A logical analysis of modules in logic programming", Journal of Logic Programming, Vol.6, pp.79-108, 1989.
[8] 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.

권 기 항

e-mail : khkwon@dau.ac.kr
1983년 3월 서울대학교 컴퓨터공학과 (공학사)
1985년 9월 Georgia Tech Computer Science (공학석사)
1994년 12월 Duke University Computer Science(공학박사)
1995년 9월~현 재 동아대학교 컴퓨터공학과 교수
관심분야 : 소프트웨어공학, computability logic

하 홍 표

e-mail : hompoyo@hotmail.com
2011년 2월 동아대학교 컴퓨터공학과 (공학사)
현 재 서강대학교 컴퓨터공학과 석사과정
관심분야 : 소프트웨어공학, computability logic, Software Modeling, Software Quality Assurance