УДК 519.7

Особенности программной реализации логических задач на языке Prolog

О.Н. Половикова, В.В. Ширяев, Н.М. Оскорбин, Л.Л. Смолякова

Алтайский государственный университет (Барнаул, Россия)

Features of Software Implementation of Logical Tasks in Prolog

O.N. Polovikova, V.V. Shiryaev, N.M. Oskorbin, L.L. Smolyakova

Altai State University (Barnaul, Russia)

Одним из перспективных направлений использования языка Prolog является решение логических задач. В данном исследовании обозначен подход поиска ответа на основе процедуры генерации состояния и процедуры проверки. Представлены решение логической задачи, которое демонстрирует на практике предлагаемый подход, и способ задания процедуры для генерации состояний. В предложенном примере генерируется битовая цепочка, которая определяет код для буквы в решении прикладной задачи. Построение ответа средствами генерации кода с проверкой позволяет не хранить в базе знаний бинарное дерево всех возможных кодов. Процесс генерации новых состояний можно связать с обучением программы и с динамическим формированием базы знаний. Подход базируется на возможностях программных сред для добавления фактов и правил к уже имеющимся, которые были получены в качестве результатов работы программы либо ее этапов. В данном случае генерирующим правилом является вся программа. Анализ построенных и апробированных процедур динамической генерации состояний и генерации фактов позволяет говорить о применимости такого решения для определенных прикладных задач.

Ключевые слова: Prolog, база знаний, предикаты, логические задачи, дерево решений, процедура генерации состояний.

One of the promising areas for using Prolog-systems is to solve logical tasks. This study outlines a solution approach based on the state generation procedure and the verification procedure. A solution to a logical task is presented, which demonstrates in practice the proposed approach and method of specifying a procedure for generating states. In the proposed example, a bit chain is generated that defines the code of a particular letter in the solution of the applied problem. Building a solution by means of code generation with verification allows not storing in the knowledge base a binary tree of all possible codes. The process of generating new states can be associated with the training of the program, with the dynamic formation of the knowledge base. The approach is based on the capabilities of software environments for adding facts and rules to existing ones, which were obtained as the results of the program or its stages. In this case, the entire program is the generating rule. An analysis of the constructed and tested procedures for the dynamic generation of states and the generation of facts allows us to talk about the applicability of such a solution for certain applied problems.

Key words: Prolog, knowledge base, predicates, logical tasks, decision tree, procedure for generating states.

DOI: 10.14258/izvasu(2021)1-20

Введение

Логические языки нашли широкое применение во многих прикладных областях, в том числе и для решения специфических задач, где требуется применять узко настраиваемые модели знаний и аппарат получения новых знаний. Среди областей применения декларативных логических языков можно выделить следующие направления: решение нетривиальных математических задач, экспертные системы, теория игр, графы, машинное обучение, а также задачи и проекты в рамках развития семантического анализа данных [1–5].

Среди логических языков особое место занимает язык Prolog (Actor Prolog, Visual Prolog и др. среды), как широко используемый инструмент декларативного логического программирования. Такая применимость обусловлена поддержкой не только логического программирования, но и объектно-ориентированной

парадигмы, программирования, управляемого шаблонами. Кроме этого современные реализации языка Prolog позволяют кодировать различные модели представления знаний [6, 7].

Доступность синтаксиса и простота интерпретации семантических конструкций языка позволяют его использовать в качестве интеллектуального («разумного») инструмента решения задач. Программирование на языке логики, во-первых, привлекает к процессу разработки не только программистов и инженеров знаний, но и других специалистов с различными научными интересами, например математиков. Во-вторых, определяет одно из перспективных направлений использования Prolog-систем — в качестве аппарата для решения логических задач. Работы в этом направлении ведутся со времени основания логического языка, и на сегодняшний день разработано множество программ (баз знаний фактов и правил), которые развиваются вместе с прикладными научными направлениями [8–11].

Приемы решения логических задач базируются на выдвижении некоторой гипотезы и ее подтверждении (или опровержении) согласно некоторой методике. Такие приемы соответствуют вычислению ответа на основе связанной многократной работы двух модулей: построение некоторого состояния, проверка полученного состояния. Подход вычисления ответа, а точнее — построения ответа согласуется с решением задач на языке Prolog. В этом случае Prolog-программу можно рассматривать как базу знаний, реализующую выполнение двух процедур:

1) построение претендента на решение и

2) согласование претендента с имеющимся набором фактов и правил.

Чтобы закодировать в терминах языка Prolog процедуру проверки на соответствие (согласование), нужно описать правилами все требования, предъявляемые к искомому состоянию. Примеры таких процедур фактов и правил рассмотрены в различных прикладных работах по логическим языкам [1, 8]. Построение процедуры построения претендента на ответ является нетривиальной задачей, решение которой привязано к специфике предметной области и требует от исследователя использования в том числе и эвристических приемов. Поэтому формализация процедуры для генерации состояний претендентов на ответ является актуальной задачей, которая имеет практическую значимость.

Целью данной работы является анализ вычислительной базы Prolog-системы для построения генераторов состояний. Основными методами исследования выступают системный анализ и логический вывод.

Следует понимать, что подход на основе генерации состояний отвечает логике поиска решения Prolog-системой. Суть такого подхода определяется возможностью получать на очередном шаге поиска новое состояние, которое нужно проверить. Проверяемые состояния являются терминальными узлами дерева решений программы, но при этом само дерево в памяти не хранится. База знаний хранит процедуры получения и проверки узлов. Обход дерева решений Prolog-системой отвечает процессу формирования решения и его проверки на соответствие набору предложений из базы знаний.

1. Решение логических задач с процедурой для генерации состояний

В исследовании рассматривается один из способов построения новых элементов базы знаний — процедура генерации состояний. Как правило, в процессе построения каждого очередного состояния происходит проверка допустимых условий. Проверяемые условия или правила также могут корректироваться по ходу поиска Prolog-системой решения, например с учетом уже добавленных состояний в динамическую базу знаний. Динамическое формирование множества возможных состояний объекта и способов его поведения позволяет выполнить поиск решения задачи в тех случаях, когда либо все дерево состояний хранить нецелесообразно, исходя из специфики решения, либо предварительное построение и хранение такого дерева может привести к комбинаторному взрыву.

Кроме этого, этап динамической генерации состояний будет ключевым в решении тех задачах, где пользователь программы управляет ходом решения, когда выбор программой соответствующей ветви решения зависит от цепочки ходов пользователя. В таких условиях до этапов выполнения программы неизвестны знания, на основе которых будет выстраиваться целевое построение. Программа в этом случае хранит знания о способах получения множества состояний в виде совокупности правил (процедуры правил), а также знания об одном или нескольких начальных или переходных состояний объекта или его поведения. При этом совокупность регламентирующих правил для перехода из одного состояния в другое может явно отсутствовать в постановке задачи. В этом случае на этапе проектирования базы знаний программы их следует вывести и формализовать либо получить знания о правилах динамической генерации самих правил, поскольку можно построить и реализовать обобщенную модель знаний о знаниях предметной области решаемой задачи. Другими словами, использование динамической базы знаний предоставляет возможность оперировать не только с множеством состояний по заранее сформированным правилам, но также и управлять генерацией самих правил, по которым будет строиться совокупность состояний.

Рассмотрим простой пример, демонстрирующий на практике генерацию состояний по правилу.

2. Демонстрация подхода на тестовой задаче

Условие задачи. По каналу связи передаются сообщения, содержащие только 4 буквы П, О, С, Т;

для передачи используется двоичный код, допускающий однозначное декодирование. Для букв Т, О, П используются такие кодовые слова: Т: 111, О: 0, П: 100. Необходимо найти кратчайшее кодовое слово для буквы С, при котором код будет допускать однозначное декодирование [12].

Для данной задачи актуальным является определение правила (или процедуры правил) для построения очередного кода буквы — генерация последовательностей из 0 и 1. Если сгенерированный код пройдет проверку «однозначное декодирование», тогда процесс построения кода следует считать выполненным, решение найдено. Если проверка не пройдена, тогда генерируется следующий код, который также подвергается проверке.

Таким образом, база знаний программы будет включать:

- факты, декларирующие уже занятые бинарные коды;
- процедуру проверки на «однозначное декодирование»;
- процедуру генерации бинарного кода.

На рисунке представлен вариант реализации решения для рассматриваемой задачи с использованием процедур генерации состояний и проверки построенного состояния-претендента условиям задачи.

```
/*Факты и Вспомогательные процедуры*/
kod(т, [1, 1, 1]).
kod(о, [0]).
kod(п, [1, 0, 0]).
concat([],L,L).
concat([X|L],M,[X|L1]):- concat(L,M,L1).
/* Процедура  проверки условий*/
inspection(M):- kod(_,K),(concat(K,_,M); concat(M,_,K)).
verification(M):-not(inspection(M)).
verificationlist([],_).
verificationlist([H|T],Rezult):- verification(H), nl,write("
verificationlist(T,Rezult1).
verificationlist([_|T],Rezult):- verificationlist(T,Rezult).
/*процедура генерации состояния */
generate([],[]).
generate([H|T],[X,Y|T1]):- X=[0|H], Y=[1|H], generate(T,T1).

search(_,5,Rezult):- nl, write(Rezult).
search(L,N,Rezult):- not(N>4), N1 is N+1, generate(L,L1),
    verificationlist(L1,Rezult), search(L1,N1,Rezult).
```

```
search([[]],1,[]).

yes - [1, 1, 0]
yes - [1, 0, 1]
yes - [1, 1, 0, 0]
yes - [1, 0, 1, 0]
yes - [1, 1, 0, 1]
yes - [1, 0, 1, 1]
[]
true
Next  10  100  1,000

?- search([[]],1,[]).
```

Программный код, демонстрирующий подход решения с генерацией состояний

В процедуре генерации (*generate/2*) на очередном шаге рекурсии обновляется список вариантов для нового кода. Вспомогательная процедура собирает результаты генерации, передает их на проверку и следит за длиной списков, которые формируются правилами. Для решения данной задачи сформировано ограничение на длину искомого кода. Такое ограничение не влияет на построенную базу фактов и правил, может формироваться динамически, исходя из длины уже известных кодов букв или длины результирующего списка с уже проверенными вариантами кодов.

В решении отсутствует процедура правил для ввода именно *кратчайшего кодового слова* согласно условию задачи. Поиск всех возможных вариантов (с учетом максимальной длины кода) выполнен намеренно, чтобы подчеркнуть возможности процедуры генерации состояний для объекта (кодовых слов) для решения подобных задач. Например, если требуется построить несколько кодовых слов, или всевозможные кодовые слова определенной длины, или кодовое слово с наименьшим значением в числовом бинарном представлении.

Правило, генерирующее коды, формирует кодовые слова списком из элементов равной длины. Проверяемые коды создаются от самых коротких ([[0], [1]]) до слов с пороговой длиной (см. табл.). Искомый кратчайший код добавляется в результирующий список самым первым. Поэтому решение описанной выше задачи можно завершить при первой удачной проверке сгенерированного программой кода.

Таблица

Генерация проверяемых кодовых слов по шагам

Шаг	Сгенерированные списки для кодовых слов
1	[0] [1]
2	[0, 0] [1, 0] [0, 1] [1, 1]
3	[0, 0, 0] [1,0,0] [0,1, 0] [1, 1, 0] [0, 0, 1] [1, 0, 1] [0, 0, 1] [1, 0, 1]
4	[0, 0, 0, 0] [1, 0, 0, 0] [0,1,0,0] [1,1,0,0]...[0, 0, 0, 1] [1, 0, 0, 1] [0,1,0,1] [1,1,0,1]

3. Возможности подхода и области применения

Построение решения средствами генерации кода с проверкой позволяет не хранить в базе знаний бинарное дерево всех возможных кодов. Без генерации кода решение можно свести к описанию бинарного дерева всех кодов и к поиску незанятых в нем узлов. Но альтернативный подход ставит встречную задачу на определение уровня хранимого программой бинарного дерева. Глубина разветвления бинарного дерева должна гарантировать необходимый поиск по основной задаче.

Немаловажным фактором для оценки выбранного подхода к построению поиска решения является корректный отклик декларативной программы (базы знаний и процедуры вывода) на ситуацию, когда целевой предикат не может быть согласован при заданных условиях. Prolog-система должна выдать отрицательный ответ для случая, когда нельзя построить кодовое слово для искомой буквы (см. условие задачи), например, если все вершины дерева заняты другими буквами (нет свободных вариантов построения кода с учетом условия «однозначное декодирование»). Анализ базы знаний программы позволяет сделать вывод, что в этом случае сформированное решение выполнит корректный поиск по всем возможным вариантам и построит пустой результирующий список. Такой ответ следует интерпретировать как отсутствие решения (само решение программа построит и найдет).

Процесс генерации новых состояний можно связать с обучением программы, с динамическим формированием базы знаний. Подход базируется на возможностях программных сред для добавления фактов и правил к уже имеющимся, которые были получены в качестве результатов работы программы либо ее этапов. В данном случае генерирующим правилом является вся программа.

Продемонстрировать работу механизма генерации фактов с целью обучения программы можно на примерах элементарных вычислительных задач: поиск факториала или чисел Фибоначчи.

Процедура правил и фактов для вычисления чисел Фибоначчи основана на рекурсивном вызове данных процедур и декларируется базовыми средствами Prolog-системы. Полученное программой очередное число Фибоначчи добавляется в начало базы знаний в виде факта. При последующем запуске программы поиск решения осуществляет среди построенных фактов. Если соответствующий факт не найден, тогда срабатывает рекурсивное правило. Если сработало правило, то поиск фактов (а затем и правила) выполняется с меньшим числовым значением параметр и новый построенный факт опять добавляется в начало базы знаний. Так как количество рекурсивных вычислений с каждым запуском программы сокращается, происходит экономия стековой памяти, которая используется для передачи значений переменных в рекурсивном вызове процедур правил.

Заключение

Сохранение для последующего использования сгенерированных программой фактов требует привлечения дополнительных ресурсов памяти, но позволяет сократить время рекурсивного вычисления. Генерация фактов для вычислительного процесса демонстрирует подход построения решения, хотя вычислительные задачи не являются основной областью применения логических языков. Добавлять в базу знаний можно построенные на ходу факты, решая задачи из разных прикладных областей, для которых можно задекларировать базу знаний и правила поиска решения.

Также следует отметить, что в задачах, где построение поисковых процедур происходит в условиях неопределенности, которая разрешается только в момент работы программы, рассматриваемый подход построения решения является приемлемым. Можно также провести сравнение эффективности подхода, основанного на динамической генерации состояний, с подходом, основанным на программном хранении деревьев всех рассматриваемых комбинаций. После анализа такого сравнения можно говорить, что использование процедуры генерации для построения новых фактов в процессе работы программы позволяет сократить время рекурсивного вычисления для задач расчета факториала, чисел Фибоначчи и им подобным.

Подход решения логических задач с использованием процедур генерации и проверки состояний апробирован на практических задачах в рамках изучения дисциплин в Институте математики и информационных технологий в Алтайском госуниверситете.

Библиографический список

1. Лорьер Ж.Л. Системы искусственного интеллекта. URL: http://lib.alnam.ru/book_sii.php (дата обращения: 19.06.2020).

2. Balai E., Gelfond M., Zhang Y. P-log: refinement and a new coherency condition // Annals of Mathematics and Artificial Intelligence. 2019. Vol. 86. DOI: 10.1007/s10472-019-09620-2.

3. Elkhatib O., Pontelli E., Son T.C. Integrating an Answer Set Solver into Prolog: ASP-PROLOG // Lecture Notes in Computer Science. Berlin, Heidelberg, 2005. DOI: 10.1007/11546207_35.

4. Половикова О.Н. Формализация процесса построения решения с использованием списков для класса логических задач в программах на языке Пролог // Известия Алтайского гос. ун-та. 2011. № 1/1 (69).

5. Nilsson U., Maluszynski. J. Logic, Programming and Prolog (2ed) Previously. URL: http://www.cs.ubbcluj.ro/~csatol/log_funk/prolog/NilsonMaluszynski_Prolog.pdf (дата обращения: 19.06.2020).

6. Bramer M. Logic Programming with Prolog. URL: http://athena.ecs.csus.edu/~logicp/Logic_Prog_Prolog.pdf. (дата обращения: 19.06.2020).

7. Карпов В.Э. Фреймы. Пролог // Электронная библиотека. URL: http://rema44.ru/resurs/study/ai/present/L05-02-frame.pdf. (дата обращения: 19.06.2020).

8. Математическая логика и логическое программирование // Математический форум MathHelpPlanet. URL: http://mathhelpplanet.com/static.php?p=matematicheskaya-logika-i-logicheskoye-programmirovaniye. (дата обращения: 19.06.2020).

9. Решение логических задач на Prolog // Блог программиста: программирование и алгоритмы (версия от 28.05 2018). URL: https://pro-prof.com/archives/1299. (дата обращения: 19.06.2020).

10. Santos Costa V. On Just in Time Indexing of Dynamic Predicates in Prolog // Lecture Notes in Computer Science. Berlin, Heidelberg, 2009. DOI:10.1007/978-3-642-04686-5

11. Повторение и рекурсия. Откат // Проект: project:prolog:povtorenie_i_rekursija. URL: http://verim.org/project/prolog/povtorenie_i_rekursija. (дата обращения: 19.06.2020).

12. ЕГЭ по информатике (2018) // Сайт К.Ю. Полякова. URL: http://kpolyakov.spb.ru/school/ege.htm. (дата обращения: 19.06.2020).