УДК 681.3.062

О.Н. Половикова

Формализация процесса построения решения с использованием списков для класса логических задач в программах на языке Пролог

O.N. Polovikova

Formalization of Process to Solve a Decision Using Lists for a Class of Logic Problems Programming in "The Prolog" Language

В исследовании рассмотрен формализованный подход к решению некоторого класса логических задач, позволяющих унифицировать используемые в решении процедуры предложений и сократить текстовый код программы за счет применения рекурсий в доказательстве предикатов. При таком построении предикатов появляется возможность повторного использования части программного кода для однотипных логических задач. Построенные в ходе исследования унифицированные процедуры предложений могут быть оформлены в виде отдельного библиотечного модуля.

Ключевые слова: логические задачи, формализация процесса решения, класс однотипных задач, списки данных, процедура предложений, заголовок и тело предиката.

The research considers formalized approach to decide logic problems of some class which allows us to unify offer procedures used in the decision and reduce a text code of the program owing to application of recursions in proving predicates. At such construction of predicates there is a possibility to reuse a part of a program code for the same logic problems. The unified offers procedures constructed during the research can be issued in the form of the separate library module.

Key words: logic problems, formalization of decision process, class of the same problems, lists of the data, offers procedure, predicates heading and body.

У каждого из языков программирования есть свой круг задач, при решении которых он используется с наибольшей эффективностью. Существуют прикладные задачи, для которых большинство классических языков программирования признаны неэффективными или малоэффективными, например, логические задачи. Для их решения необходимы языковые средства, позволяющие описать объекты с возможными значениями свойств и потенциально допустимые отношения между ними, а также искомую ситуацию, которая будет ответом. Искомая ситуация характеризуется таким соотношением свойств объектов и их связей между собой, при которых не будет противоречий с допустимыми значениями. Построение решения формируется в терминах предметной области рассматриваемой задачи.

Определить объекты (классы объектов) и взаимодействия между ними можно с использованием объектно-ориентированных языков и сред программирования, они для этого и предназначены. А вот описать искомую ситуацию взаимодействия объектов, удовлетворяющую, допустим, характеристикам их свойств и связей, а главное – заставить компилятор или транслятор языка искать такую ситуацию для большинства языков программирования практически труднореализуемо. Для императивных языков (в том числе и с поддержкой принципов объектно-ориентированной методологии) для этих целей необходимо выполнить трудоемкий и ресурсоемкий дополнительный этап решения – построение и реализацию алгоритма поиска требуемой ситуации. Методология языка Пролог позволяет описать допустимую ситуацию в терминах предметной области рассматриваемой задачи. Пролог-система автоматически строит ответ для заданной ситуации без дополнительного этапа. Поэтому язык Пролог обладает явными преимуществами для решения различного рода логических задач, где требуется определить и построить ситуацию, определяемую взаимодействием между объектами.

Несмотря на широкое применение декларативных языков для решения логических задач, подходы и методы их решения недостаточно изучены и унифицированы. Сложность формализации связана с многовариантностью и индивидуальными особенностями решений каждой задачи. При этом можно сформировать отдельные типы задач, решение которых строится на общих принципах и подчиняется одному множеству требований при описании взаимоотношений внутри предметной области. Поэтому проблема выделения класса однотипных логических задач и формализация подхода их решения с разработкой библиотечных моделей программного кода является актуальной и имеет практическое значение.

В данном исследовании описан и проанализирован один из способов формализации решения некоторого класса подобных логических задач с использованием унифицированных процедур предложения, созданных на основе списков. Выделены основные достоинства и недостатки данного подхода, а также преимущества использования списков в заголовках и доказательствах предикатов при описании искомой ситуации (фактов и правил).

Решение логических задач на языке Пролог включает следующие этапы:

1. Проектирование необходимых объектов с выделением их актуальных свойств, взаимосвязей между объектами.

2. Описание базы данных объектов с допустимыми значениями свойств.

3. Формирование искомой ситуации:

3.1. Построение базы правил взаимосвязей между объектами с конкретным сочетанием их свойств.

3.2. Описание сценария последовательного выбора правила из базы правил для достижения искомой ситуации.

В традиционном способе решения логических задач [1–4] искомая ситуация строится на предложениях (предикатах), у которых в заголовке в качестве аргументов записаны простые объекты (термы): атомы, константы и переменные, в том числе и анонимные. А вот такие термы, как структуры, которые являются сложными объектами, практически не применяются.

Использование списков в качестве аргументов в заголовках предикатов обладает своими достоинствами и недостатками. Традиционный способ решения задачи позволяет описать базу правил на языке, близком к естественному, применение списков снижает наглядность изложения искомой ситуации.

Основное достоинство использования списков в качестве аргументов в заголовках предикатов – повышение универсальности создаваемых предложений и процедур предложений и сокращение текстового кода программы за счет создания рекурсий в доказательстве предикатов. При таком построении предикатов появляется возможность повторного применения части программного кода для однотипных логических задач. Примером является предикат, который проверяет на парное различие каждого свойства всех объектов одного класса по всем актуальным свойствам.

```
различие([]).
различие([A|[]]).
различие([A|[B|B1]]):-
not(A==B),различие([A|B1]),
различие([B|B1]).
/*Процедура предикатов для проверки на
парное различие всех элементов в списке
("каждый с каждым").
Данная процедура состоит из трех предло-
жений и рекурсивно выполняет сравнение
элементов в два направления:
очередной элемент с каждым последующим:
различие([A|B1]),
следующий элемент с каждым последующим:
различие([B|B1]). */
```

Необходимо заметить, что такой неклассический подход формирования заголовка, во-первых, позволяет сократить количество аргументов, используемых в заголовке предиката; во-вторых, принципиально изменяется и хвостовая часть предложения.

Сокращение числа аргументов связано со свойством структур, так как элементами структуры могут быть в свою очередь структуры, тогда формально в заголовке любого предложения можно использовать два аргумента: структура входных данных, структура выходных данных. Доказательство предиката в этом случае строится на рекурсивном вызове правил одной или нескольких процедур правил. Рекурсию можно организовать как по внутреннему списку, перебирая все его элементы от головы до хвоста списка, так и по внешнему (главному) списку.

Чтобы показать универсальность подхода с использованием списков для некоторого класса задач, рассмотрим пример построения двух шаблонных процедур, позволяющих организовать проверку на различие элементов внутреннего списка, расположенных на определенных позициях во внешнем списке.

Пусть сформировано множество объектов, которое определяется следующим списком: $[[a_1,b_1,\dots,c_1],\ [a_2,b_2,\dots,c_2],\dots,[a_n,b_n,\dots,c_n]]$, где каждый внутренний список определяет характеристики одного объекта; a_i – это значение атрибута (свойства) одного объекта; $[a_n,b_n,\dots,c_n]$ – это список возможный значений для одной характеристики объекта.

Требуется определить – существуют ли объекты, обладающие одинаковыми значениями какого-либо свойства.

База правил решения данной задачи состоит из двух процедур правил:

```
проверить_список([[]|_]).
проверить_список (C):-
построить_список(C,B,D),различие(B),
проверить_список(D).
/*Предикат, который рекурсивно получает
однотипный список (варианты одного свой-
ства одного варианта объектов),
вызывает предикат для проверки на разли-
чие однотипных элементов списка*/
построить_список([],[],[]).
построить_список([[H|H1]|T],[H|T1],
[H1|T2]):- построить_список(T,T1,T2).
/*Предикат, который рекурсивно из задан-
ного списка(первый аргумент) формирует
два списка:
```

```
текущий список(второй аргумент) - список
одного свойства одного объекта, получен-
ные из заданного списка; список элементов
(третий аргумент), разнотипные элементы
которые не вошли в текущий список.*/
```

Рассмотренный пример показывает универсальность процедур правил, использующих списки для определенного класса однотипных задач.

Выделим класс простых логических задач на определение свойств объектов следующим образом: задача принадлежит данному классу, если в постановке задачи требуется построить ситуацию, определяемую соотношением различных свойств между несколькими объектами. Элементами данного класса являются задачи, в которых требуется сформировать объекты таким образом, чтобы выполнялись правила, регламентирующие взаимодействие между самими объектами и их свойствами, при этом условиями задачи определены количество объектов, количество их характеристик, а также дискретные множества их возможных значений. Примером такого класса задач является следующая логическая задача: *A*, *C* и *M* живут на разных этажах. *A* живет не на самом верхнем этаже и не на самом нижнем. *C* живет не на среднем этаже и не на нижнем. На каких этажах живут *A*, *C* и *M*, если они живут на разных этажах?

Предположим, что выполнен первый шаг решения задачи на языке Пролог – спроектированы объекты рассматриваемой предметной области. Следующим шагом необходимо выполнить описание базы данных объектов с допустимыми значениями свойств. Формирование базы данных программы привязано к конкретной задаче, но использование структур делает возможным полную или частичную формализацию данного шага построения решения.

свойства$([[a_{11},a_{12},…,a_{1n}],[a_{21},a_{22},…,a_{2n}],…,[a_{k1},a_{k2},…,a_{kn}]])$,

где k – это количество возможных характеристик объекта; n – количество возможных значений какой-либо характеристики.

База данных для описанного выше примера (класс простых логических задач) будет состоять из следующего факта:

```
attributs([[a,c,m],[1,2,3]]).
```

Количество элементов списка и количество подобных списков определяется спецификой задачи. Каждый список внутри основного списка определяет множество возможных значений одного свойства объекта, в отличие от традиционного подхода, где возможное значение одного свойства задается отдельным предикатом. Поэтому база фактов решения данной задачи традиционным способом будет состоять из шести одноарных предикатов: каждый предикат задает одно значение одного свойства объекта.

После формирования базы фактов необходимо выполнить построение базы правил взаимосвязей между объектами с конкретным сочетанием их свойств (шаг 3.1). Выполнение данного шага формально можно записать, используя процедуру одноименных предложений (фактов или правил):

правило$([X_{11},X_{12},…,X_{1n}]):-A_{11},A_{12},…,A_{1n1}.$
правило$([X_{21},X_{22},…,X_{2n}]):-A_{21},A_{22},…,A_{2n2}.$
…
правило$([X_{k1},X_{k2},…,X_{kn}]):-A_{k1},A_{k2},…,A_{knk}.$

База фактов для рассматриваемого примера будет реализована следующей процедурой:

```
rule([a,X]):-!,not(X=1;X=3).
rule([c,X]):-!,not(X=3;X=2).
rule([_,_]).
/*База фактов реализована процедурой
правил rule. Каждый предикат формирует
определенное условие на сочетание свойств
объекта согласно условиям задачи.*/
```

Спецификой применения структур к данному шагу реализации является рекурсивная проверка всех построенных объектов на соответствие базе фактов, что позволяет унифицировать контроль за соблюдением правил каждым построенным объектом.

```
проверить_правила([]).
проверить_правила([H|T]):-правило(H),
проверить_правила(T).
```

Проверка на соответствие базе фактов для описанного выше примера:

```
check_rules([]).
check_rules([H|T]):-rule(H),
check_rules(T).
/* Процедура check_rules выполняет
проверку на соответствие правилам базы
знаний.*/
```

Кроме построенных и описанных выше формальных предложений, в базу правил (шаг 3.1) следует включить специальные предикаты, которые позволяют назначить каждому объекту совокупность значений его атрибутов (свойств) из сформированной базы данных. Использование списков позволяет унифицировать распределение значений атрибутов по объектам для класса простых логических задач:

```
построить_объекты([],[],[]).
построить_объекты([[A|B]|T],[A|T1],[B|T2]
):- построить_объекты(T,T1,T2).
построить_объекты([[Y,A|B]|T],[A|T1],[[Y|
B]|T2]):- построить_объекты(T,T1,T2).
описать_ситуацию([[]|_],[]).
описать_ситуацию(C,[X|Y1]):-
построить_объекты(C,X,Y),
описать_ситуацию(Y,Y1).
/* описать_ситуацию – процедура, которая
отвечает за назначение объектам
определенных свойств:
первый аргумент задает список значений
для характеристик по всем объектам
(входной параметр); второй аргумент
определяет список полученных объектов
(выходной параметр).
построить_объекты – вспомогательный
предикат, формирует значение свойств для
текущего объекта.*/
```

Формализованное описание сценария последовательного выбора правила из базы правил для достижения искомой ситуации (шаг 3.2) можно представить следующим предложением:

```
?- свойства(C), описать_ситуацию(C,C1),
проверить_правила(C1), распечатать(C1).
/* C – список списков возможных значений
для свойств объектов, C1 – список
объектов, соответствующих решению задач
из выделенного класса.*/
```

Рассмотренный в исследовании формализованный подход к решению некоторого класса логических задач позволяет унифицировать используемые в решении процедуры предложений и сократить текстовый код программы за счет применения рекурсий в доказательстве предикатов. При таком построении предикатов появляется возможность повторного использования части программного кода для однотипных логических задач. Построенные в ходе исследования унифицированные процедуры предложений могут быть оформлены в виде отдельного библиотечного модуля. Данный модуль можно использовать для решения задач из определенного выше класса на языке Пролог.

Отметим, что можно найти в источниках или построить логические задачи, для решения которых проблематично использование описанного выше неклассического подхода, но при этом существует множество задач, для описания ситуаций которых практически труднореализуемым является классическое построение заголовков правил и их доказательство, а применение структур позволяет сделать возможным их решение на языке Пролог. Следует заметить, что использование структур позволяет полностью или частично унифицировать описание базы данных и построение искомой ситуации, отвечающей решению задачи.

Библиографический список

1. Братко И. Программирование на языке Пролог для искусственного интеллекта: пер. с англ. – М., 1990.
2. Адаменко А.Н., Кучуков А.М. Логическое программирование и Visual Prolog. – СПб., 2003.
3. Малпас Дж. Реляционный язык Пролог и его применение: пер. с англ. – М., 1990.
4. Шрайнер П.А. Основы программирования на языке Пролог. – М., 2005.