УДК 004.43

Библиотечная реализация Пролог-решателя

© 2012 г. И. В. Струков

iley@iley.ru

Кафедра алгоритмических языков

1 Введение

Язык логического программирования Пролог был разработан в 70-х годах XX века для решения задач обработки естественного языка. Со временем область успешного применения этого языка расширилась, и сейчас она включает в себя такие задачи искусственного интеллекта, как автоматическое доказательство теорем, экспертные системы, игровые программы и многое другое [2]. Ярким примером применения Пролога в современном программировании может служить компьютерная система Watson [5], известная тем, что в 2011 году выступила в качестве оппонента человеку в интеллектуальной игре Jeopardy и одержала победу. Часть её программного обеспечения написана на Прологе [3].

Несмотря на все достоинства данного языка и логической парадигмы программирования в целом, Пролог в наше время применяется при создании крупных программных систем чрезвычайно редко, даже в ситуациях, когда разработчики хорошо владеют данным языком. Основной причиной этого является, по всей видимости, тот факт, что любая крупная программа включает в себя большое число компонентов, и, как правило, лишь часть из них целесообразно выражать в терминах логического программирования. К примеру, уже упомянутая система Watson использует Пролог лишь для решения некоторых

задач, в частности обработки естественного языка, другие же части системы написаны на языках Java и C++.

Данная работа посвящена описанию экспериментальной реализации языка Пролог при помощи метода непосредственной интеграции в рамках библиотеки InteLib [1]. Эта реализация может упростить использование Пролога в качестве дополнительного языка программирования в программных проектах на C++.

2 Пролог как дополнительный язык программирования

При использовании Пролога в программном проекте, где основным является один из широко применяемых сегодня объектно-ориентированных языков программирования, возникают определённые трудности, связанные с интеграцией частей системы, написанных на разных языках. Для интеграции программного кода на Прологе в систему, написанную на объектно-ориентированном языке программирования, чаще всего применяют один из следующих двух подходов.

1. Написание двух раздельных программ, которые взаимодействуют посредством каких-либо средств межпроцессных коммуникаций.

2. Использование встраиваемого интерпретатора.

Оба эти подхода требуют написания дополнительного программного кода, служащего связующим звеном между частями системы. В первом случае этот код представляет собой реализацию некоторого протокола взаимодействия для каждого из языков системы. Во втором случае это так называемые обёртки, которые адаптируют структуры данных и операции, реализованные на базовом языке, для использования в коде на

дополнительном языке, который исполняется с помощью интерпретатора. В обоих случаях дополнительный код усложняет систему и затрудняет её разработку, отладку, тестирование и поддержку. Кроме того, каждый из этих подходов имеет свои ограничения. Например, использование встраиваемого интерпретатора не всегда допустимо из-за специфики целевой платформы или по административным причинам. Указанные проблемы в совокупности могут свести на нет преимущества, приобретаемые от применения дополнительного языка программирования.

Одним из подходов к совместному применению различных языков программирования в рамках одного проекта, призванных решить описанные проблемы, является метод непосредственной интеграции. Данный метод состоит в имитации синтаксиса и семантики дополнительного языка выразительными средствами языка базового. Примером использования метода непосредственной интеграции является библиотека InteLib, которая позволяет встраивать код на языках Лисп, Scheme [7], Рефал [6] и Плэнер [8] в программы на C++. Примечателен тот факт, что InteLib изначально спроектирована таким образом, чтобы с её помощью можно было реализовать вычислительные модели довольно широкого класса языков. Условно можно выделить в данной библиотеке две основные части. Первая часть включает в себя классы, реализующие общие для всех дополнительных языков структуры данных. Эти классы, помимо прочего, обеспечивают поддержку динамического определения типов во время исполнения программы и упрощают управление памятью с помощью техники так называемых «умных указателей». Вторая часть библиотеки включает в себя реализации вычислительных моделей альтернативных языков программирования. Благодаря тому, что все структуры данных, используемые реализациями языков в рамках InteLib, построены на основе общих базовых классов, передача данных между частями программы, написанными на различных языках програм-

мирования, максимально упрощается.

Оставшаяся часть статьи описывает InteLib Prolog, реализацию языка Пролог в рамках библиотеки InteLib. Полученная реализация не только позволяет упростить интеграцию кода на Прологе в программные проекты на C++, но также даёт возможность использовать Пролог совместно с языками программирования, уже реализованными в InteLib — Лисп, Scheme, Рефал, Плэнер. Разработанная реализация расширяет библиотеку InteLib классами, реализующими представление данных и вычислительную модель языка Пролог. Кроме того, InteLib Prolog позволяют имитировать синтаксис Пролога настолько, насколько это возможно в исходном коде на C++.

3 Представление программ и данных

Программа на Прологе представляет собой набор *правил*, которые могут быть как *фактами*, выражающими элементарные отношения между сущностями, так и сложными предложениями. Правила записываются при помощи *термов*. Терм по определению — это либо число, либо *атом*, либо список, либо *переменная*, либо *составной терм*. Составной терм состоит из *функтора*[1] и списка *аргументов*. Важен тот факт, что термы используются для представления как программ, так и данных в Прологе[2]. Можно провести аналогию между термами в Прологе и S-выражениями [4], используемыми в языке Лисп. Прологовскому атому соответствует *символ* в Лиспе, списки и числа используются аналогично в этих двух языках. Для обеспечения совместимости между программами на Лиспе и Прологе в рамках InteLib необходимо было повторно использовать указанные типы данных. Кроме того, поскольку InteLib, так же как и многие другие современные реализации языка Лисп,

[1] Функтор состоит из атома и числа, так называемой арности, т.е. числа аргументов

[2] Данное свойство языка программирования называют гомоиконностью

Таблица 1. Сравнение синтаксических элементов

Пролог	InteLib Prolog	Комментарий
`f`	`f`	атом
`X`	`X`	переменная
`f(1, 2, 3)`	`f(1, 2, 3)`	составной терм
`f(1, 2, 3).`	`*f(1, 2, 3)`	факт
`[1, 2, 3]`	`(S|1, 2, 3)`	список
`[H|T]`	`H^T`	список
`f(X) :- g(X).`	`f(X) <<= g(X)`	правило
`f(X), g(X)`	`f(X) & g(X)`	конъюнкция
`f(X); g(X)`	`f(X) | g(X)`	дизъюнкция
`f(X), !`	`f(X) & cut`	отсечение
`X + Y * Z`	`X + Y * Z`	арифм. выражение
`Z is X + Y`	`Z.is(X + Y)`	операция `is`

расширяет понятие S-выражения рядом дополнительных типов, например, таких, как строка (string), хэш-таблица, вектор и очередь, важно было обеспечить совместимость между языками и при использовании таких типов. В связи с этим, модель данных и реализация вычислителя InteLib Prolog была разработана таким образом, чтобы повторно использовать как можно большее число уже реализованных типов данных. Так, например, для представления списков в InteLib Prolog непосредственно используются ранее описанный в InteLib класс `SExpressionCons`. В тех случаях, когда непосредственное использование существующих классов было невозможным, повторное использование достигалось при помощи наследования. Примером может служить класс `PlgExpressionAtom`, представляющий атом в Прологе.

Для имитации синтаксиса Пролога в библиотеке предусмотрены перегрузки операторов. Для иллюстрации рассмотрим примеры выражений в стандартном синтаксисе Пролога и их аналоги в InteLib Prolog, представленные в таблице 1.

Как видно из таблицы, запись атомов, переменных и составных термов в InteLib Prolog полностью соответствует стандартному синтаксису. Для построения составных термов используется перегруженный оператор `()`. Для записи списков используется принятый в InteLib синтаксис, это позволяет добиться единообразия при использовании нескольких дополнительных языков программирования, например, Лиспа и Пролога.

В качестве замены оператора `:-`, который отсутствует в C++, используется оператор присваивания с побитовым сдвигом `<<=`, который визуально схож с `:-` и символом ←, спользуемым в математической логике. Факт в InteLib Prolog можно записать либо при помощи оператора `<<=`, как и при определении составных правил, либо при помощи оператора `*`. Второй способ позволяет визуально выделить факты и сделать запись короче.

Вместо запятой и точки с запятой, обозначающих соответственно конъюнкцию и дизъюнкцию в Прологе, используются символы `&` и `|`. Решение о выборе данных символов было принято в связи с тем, что точка с запятой в C++ имеет специальное значение: она разделяет операторы и не может быть перегружена, а запятая, хотя и может быть перегружена, обладает очень низким приоритетом, что не соответствует приоритету запятой в Прологе. Выбранные символы `&` и `|`, хотя и не схожи визуально с их эквивалентами в Прологе, обладают аналогичным смыслом в C++. В C++ эти символы обозначают побитовые конъюнкцию и дизъюнкцию, и их смысл в программе на InteLib Prolog должен быть интуитивно понятен программисту на C++.

Для записи арифметических операций используется перегузка всех необходимых операторов C++. Кроме того, для имитации бинарной операции `is` предусмотрен специальный метод в классе, представляющем переменные. Для замены символов, для которых не удалось найти адекватного представления в синтаксисе C++, используется текстовая запись. Так, напри-

мер, вместо символа отсечения `!` используется идентификатор `cut`.

Зачастую может быть удобнее писать программы при помощи оригинального синтаксиса Пролога, а не его имитации, и, кроме того, может возникнуть необходимость адаптировать программу, написанную на традиционном Прологе, для выполнения при помощи InteLib Prolog. В связи с этим была разработана программа-транслятор, которая преобразует исходный код на Прологе в синтаксис InteLib Prolog.

4 Вычислительная модель языка Пролог

Как уже упоминалось в статье, программа на Прологе состоит из набора правил. Все эти правила в совокупности составляют так называемую базу данных, а выполнение программы представляет собой вычисление некоторого запроса к базе данных. Для выполнения запроса Пролог-система производит поиск такого набора значений переменных, который сделал бы искомый запрос истинным в терминах правил, составляющих базу данных. Поиск производится при помощи обхода дерева решений в глубину.

Чтобы реализовать вычислительную модель Пролога в рамках InteLib Prolog, в библиотеку был введён класс, представляющий состояние вычислителя (`PlgContinuation`) . В ответ на любой запрос пользователь библиотеки получает экземпляр класса `PlgContinuation`, который может использоваться для последовательного получения всех решений. Экземпляр `PlgContinuation` содержит в себе три важные структуры данных: контекст, хранящий связи переменных, очередь текущих целей и стек точек возврата. Алгоритм поиска очередного решения для заданного запроса состоит из следующих шагов.

1. Проверить, есть ли в очереди текущих целей запросы, для которых ещё не найдены решения. Если нет — попытать-

ся произвести *откат*. Если откат произвести не удалось, решения не существует.

2. Извлечь из очереди целей первый элемент и произвести для него замену имён переменных на их значения в текущем контексте. Если очередь пуста — поиск успешно завершён.

3. Найти *предикат*, связанный с функтором текущей цели, и вычислить его.

4. Если вычисление предиката неуспешно, попытаться произвести откат. Если откат произвести не удалось, решения не существует. Иначе перейти к пункту 2.

Рассмотрим более подробно отдельные пункты алгоритма. Первый пункт необходим для проверки граничного условия. Благодаря нему алгоритм поиска сообщит о неудаче в случае, когда все решения уже были найдены.

Пункты с второго по четвёртый образуют основной цикл поиска. На первый взгляд может показаться, что для процесса поиска решения, который по сути является рекурсивным, использование рекурсии должно было бы привести к более простой реализации, однако, из-за различий в вычислительных моделях Пролога и C++ итеративная реализация оказывается более простой и эффективной. Этот факт подтверждает предположение, выдвинутое в [7] и состоящее в том, что вычислительные модели, основанные на рекурсии, следует реализовывать без использования рекурсии.

Используемый в третьем пункте предикат может быть реализован как на C++ (такой предикат назовём специальным), так и на Прологе. Такие операции, как конъюнкция и дизъюнкция, представляют собой специальные предикаты, поэтому их реализацию не пришлось напрямую предусматривать в алгоритме поиска. Важно отметить, что пользователь библиотеки может определять собственные специальные предикаты

при помощи того же механизма, с помощью которого реализованы конъюнкция и дизъюнкция. Этот факт даёт пользователю возможность произвольным образом расширять семантику языка.

Рассмотрим подробнее, как реализовано вычисление предикатов, написанных на Прологе. Каждый такой предикат состоит из некоторого списка правил и идентифицируется функтором. В InteLib Prolog ссылка на указанный список хранится в объекте, представляющем атом функтора. При вычислении предиката происходит поиск в списке первого подходящего правила при помощи *унификации*. Если такое правило найдено, его тело заносится в список целей. Позиция в списке, на которой был прекращён поиск, сохраняется в так называемой *точке возврата*. Если при откате потребуется найти другое решение для данного предиката, поиск будет продолжен с той же позиции.

5 Унификация

Одним из ключевых механизмов, на которых основана вычислительная модель языка Пролог, является унификация. Унификация — это операция, применяемая к двум произвольным термам. В процессе унификации Пролог-решатель производит поиск такого набора значений для несвязанных переменных, входящих в термы, что при подстановке этих значений вместо всех вхождений соответствующих переменных термы становятся тождественными. В упрощённом виде стандартный алгоритм унификации в Прологе состоит из следующих шагов.

1. Если один из термов является несвязанной переменной, задать её значение равным второму терму в текущем контексте. В таком случае унификация считается успешной.

2. Если оба терма являются списками, произвести попарную унификацию элементов списков. Если все элементы были

унифицированы, унификация списков считается успешной.

3. Если оба терма являются составными, проверить, что их функторы равны, а списки аргументов унифицируются. В таком случае унификация составного терма успешна.

4. Если оба терма являются числами или атомами, унификация сводится к простой проверке на равенство.

5. Если ни одно из условий 1-4 не выполнено, унификация считается неуспешной.

К реализации унификации в InteLib Prolog выдвигались следующие требования. Во-первых, унификация должна работать для уже существующих типов данных InteLib, причём, желательно, без написания дополнительного кода для каждого отдельного типа. Во-вторых, механизм унификации должен быть расширяемым. То есть, реализация унификации для дополнительных типов данных, которые в будущем могут быть добавлены в InteLib Prolog, не должна вызывать больших затруднений.

В результате были приняты следующие конструктивные решения. Во-первых, для унификации примитивных типов данных использовать стандартную операцию сравнения, предусмотренную InteLib. Во-вторых, для реализации унификации составных типов данных, таких как составные термы, использовать механизм виртуальных функций. Наконец, для уже существующих составных типов, таких как `SExpressionCons`, в качестве исключения выполнять унификацию специальным образом. С учётом перечисленных решений алгоритм унификации двух термов в InteLib Prolog выглядит следующим образом.

1. Если один из термов реализует виртуальный метод Unify, результатом унификации будет вычисление данного мето-

да с соответствующими параметрами. В частности, реализация данного метода в классе `PlgExpressionVariable` добавляет в текущий контекст связь данной переменной со вторым аргументом унификации, а реализация в `PlgExpressionTerm` проверяет, что второй аргумент также является составным термом и производит унификацию функторов и списков аргументов.

2. Если оба терма являются списками, итеративно произвести попарную унификацию элементов списка.

3. Если пункты 1-2 не выполнены, результатом унификации является вычисление метода `IsEqual`, реализующего стандартную операцию сравнения в InteLib. Это верно для чисел, атомов, строк и других типов данных.

Для использования каких-либо новых примитивных типов данных в программе на InteLib Prolog достаточно реализовать операцию сравнения. Для использования новых составных типов данных необходимо применить наследование и реализовать виртуальный метод `Unify`. Как правило, реализация данного метода не составляет большого труда, так как унификация для какого-либо составного типа данных в большинстве случаев тривиальным образом сводится к унификации объектов, которые экземпляр данного типа в себе содержит.

6 Откат

Другим важным механизмом, используемым в Прологе, является откат. При выполнении отката Пролог-вычислитель возвращается к состоянию[3], которое он принимал в прошлом в момент выбора пути в дереве решений. Такой выбор, в частно-

[3]Здесь под состоянием вычислителя подразумеваются значения переменных и список целей.

сти, осуществляется при обработке дизъюнкций и выборе правила из базы данных.

Для реализации отката в InteLib Prolog используются точки возврата. Каждая точка возврата при её создании сохраняет информацию о состоянии контекста переменных и очереди целей. В текущей реализации контекст, представляемый классом `PlgContext`, состоит из хэш-таблицы, предназначенной для хранения значений связанных переменных, и стека связей. Каждый элемент стека связей (будем называть его *фреймом*) — это список имён переменных, для которых были добавлены связи начиная с момента создания некоторой точки возврата. Когда создаётся очередная точка возврата, в стек связей добавляется новый пустой фрейм, а ссылка на предыдущую вершину стека сохраняется в точке возврата. Во время отката при помощи некоторой точки возврата T достаточно удалить из контекста связи для всех переменных, чьи имена встречаются в стеке связей выше фрейма, ссылка на который хранится в T. Таким образом будет восстановлено состояние контекста на момент создания T.

В работе со стеком точек возврата есть важный нюанс, связанный с использованием отсечения. Операция отсечения в Прологе удаляет все точки возврата, созданные с момента начала выполнения текущего предиката. Для того, чтобы реализовать такое поведение в InteLib Prolog, в стек точек возврата помещаются специальные метки, которые позволяют при выполнении отсечения определить, какие из точек следует удалить. В результате, алгоритм отсечения состоит в том, чтобы удалять точки возврата с вершины стека, пока не будет встречена метка начала выполнения предиката, либо стек не окажется пуст.

7 Пример программы

Ниже представлен полный пример программы, реализующей алгоритм быстрой сортировки [9] (quicksort) при помощи InteLib Prolog. Надо заметить, что данная программа отличается от своего аналога на стандартном Прологе только дополнительными объявлениями, необходимыми в C++, и деталями синтаксиса.

```
#include <stdio.h>
#include <prolog/prolog.hpp>

int main() {
  // необходимо для использования стандартных предикатов
  using namespace PlgStdLib;

  // вспомогательные объекты для конструирования списков
  SListConstructor S;
  SReference &nil = *PTheEmptyList;

  // определения всех используемых атомов и переменных
  PlgAtom qsort("qsort"), split("split");
  PlgVariable H("H"), T("T"),
    R("R"), L("L"),
    X("X"), Res("Res"),
    LS("LS"), RS("RS");

  // предикат сортировки qsort(+In, -Out)
  // In - входной список, Out - результат
  *qsort(nil, nil);

  qsort(H^T, Res) <<=
    split(H, T, L, R) &
    qsort(L, LS) &
    qsort(R, RS) &
```

```
    append(LS, H ^ RS, Res);

  // предикат для разбиения списка
  // split(+Pivot, +In, -Lower, -Upper)
  // Pivot - граничное значение, In - входной список,
  // Lower и Upper - результирующие списки
  *split(X, nil, nil, nil);

  split(X, H^T, H^LS, RS) <<=
    (H <= X) &
    split(X, T, LS, RS);

  split(X, H^T, LS, H^RS) <<=
    (H >  X) &
    split(X, T, LS, RS);

  // входной список
  SReference list = (S|3, 1, 2, 5, 4);

  // получаем объект для выполнения запроса
  PlgContinuation cont = qsort(list, X).Query();

  // вычисляем первый результат запроса
  // в нашем случае вычисление всегда успешно
  cont->Next();

  // получаем значение переменной X, с которой
  // при выполнении запроса был связан результат
  SReference sortedList = cont->GetValue(X);

  // выводим результат на экран
  puts(sortedList->TextRepresentation().c_str());
  return 0;
}
```

8 Заключение

В результате выполненной работы библиотека InteLib была дополнена реализацией языка программирования Пролог, которая на данный момент, хотя и не является полной, вполне работоспособна и может быть использована для интеграции несложных программ на Прологе в проекты на C++. Полученная реализация включает в себя библиотеку классов на C++, которая позволяет представлять и исполнять программный код на Прологе, а также транслятор для адаптации существующих программ на Прологе к данной реализации. Как показывает практика, InteLib Prolog позволяет в отдельных случаях значительно упростить интеграцию кода на Прологе в программные проекты на C++, что подтверждает перспективность метода непосредственной интеграции.

Список литературы

[1] Elena Bolshakova, Andrey Stolyarov. Building functional techniques into an object-oriented system. // *Knowledge-Based Software Engineering. Proceedings of the 4th JCKBSE*, volume 62 of *Frontiers in Artificial Intelligence and Applications*, pages 101–106, Brno, Czech Republic, 2000. IOS Press, Amsterdam.

[2] I. Bratko. *Prolog programming for artificial intelligence.* International computer science series. Addison Wesley, 2001.

[3] Adam Lall, Paul Fodor. Natural language processing with prolog in the ibm watson system. Technical report, IBM Thomas J. Watson Research Center, 2011.

[4] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, April 1960.

[5] IBM Systems, Technology Group. Watson — a system designed for answers. Technical report, IBM Corporation, 2011.

[6] И. Е. Бронштейн, А. В. Столяров. Библиотечная поддержка объектно-ориентированной модели языка Рефал. // *Сборник статей молодых учёных факультета ВМиК МГУ*, pages 36–46, Москва, 2009. Издательский отдел факультета ВМиК МГУ.

[7] А. В. Столяров. Импорт вычислительной модели языка scheme в объектно-ориентированное окружение. // *Сборник статей молодых учёных факультета ВМиК МГУ*, pages 119–130, Москва, 2008. Издательский отдел факультета ВМиК МГУ.

[8] О. Г. Фролова. Библиотечная реализация вычислительной модели языка Плэнер. // *Сборник статей молодых учёных факультета ВМиК МГУ*, pages 24–33, Москва, 2008. Издательский отдел факультета ВМиК МГУ.

[9] Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. Глава 7. Быстрая сортировка // *Алгоритмы: построение и анализ = Introduction to Algorithms / Под ред. И. В. Красикова*, с. 198-219, Москва, 2005.