

9 ЛОГИЧЕСКИЙ ВЫВОД В ЛОГИКЕ ПЕРВОГО ПОРЯДКА

В этой главе будут определены эффективные процедуры получения ответов на вопросы, сформулированные в логике первого порядка.

В главе 7 определено понятие **логического вывода** и показано, как можно обеспечить непротиворечивый и полный логический вывод для пропозициональной логики. В данной главе эти результаты будут дополнены для получения алгоритмов, позволяющих найти ответ на любой вопрос, сформулированный в логике первого порядка и имеющий ответ. Обладать такой возможностью очень важно, поскольку в логике первого порядка можно сформулировать практически любые знания, приложив для этого достаточные усилия.

В разделе 9.1 представлены правила логического вывода для кванторов и показано, как можно свести вывод в логике первого порядка к выводу в пропозициональной логике, хотя и за счет значительных издержек. В разделе 9.2 описана идея **унификации** и показано, как эта идея может использоваться для формирования правил логического вывода, которые могут применяться непосредственно к высказываниям в логике первого порядка. После этого рассматриваются три основных семейства алгоритмов вывода в логике первого порядка: **прямой логический вывод** и его применение к **дедуктивным базам данных** и **продукционным системам** рассматриваются в разделе 9.3; процедуры **обратного логического вывода** и системы **логического программирования** разрабатываются в разделе 9.4; а системы **доказательства теорем** на основе резолюции описаны в разделе 9.5. Вообще говоря, в любом случае следует использовать наиболее эффективный метод, позволяющий охватить все факты и аксиомы, которые должны быть выражены в процессе логического вывода. Но следует учитывать, что формирование рассуждений с помощью полностью общих высказываний логики первого порядка на основе метода резолюции обычно является менее эффективным по сравнению с формированием рассуждений с помощью определенных выражений с использованием прямого или обратного логического вывода.

9.1. СРАВНЕНИЕ МЕТОДОВ ЛОГИЧЕСКОГО ВЫВОДА В ПРОПОЗИЦИОНАЛЬНОЙ ЛОГИКЕ И ЛОГИКЕ ПЕРВОГО ПОРЯДКА

В этом и следующих разделах будут представлены идеи, лежащие в основе современных систем логического вывода. Начнем описание с некоторых простых правил логического вывода, которые могут применяться к высказываниям с кванторами для получения высказываний без кванторов. Эти правила естественным образом приводят к идее, что логический вывод в логике первого порядка может осуществляться путем преобразования высказываний в логике первого порядка, хранящихся в базе знаний, в высказывания, представленные в пропозициональной логике, и дальнейшего использования пропозиционального логического вывода, а о том, как выполнять этот вывод, нам уже известно из предыдущих глав. В следующем разделе указано одно очевидное сокращение, которое приводит к созданию методов логического вывода, позволяющих непосредственно манипулировать высказываниями в логике первого порядка.

Правила логического вывода для кванторов

Начнем с кванторов всеобщности. Предположим, что база знаний содержит следующую стандартную аксиому, которая передает мысль, содержащуюся во многих сказках, что все жадные короли — злые:

$$\forall x \text{ King}(x) \wedge \text{Greedy}(x) \heartsuit \text{ Evil}(x)$$

В таком случае представляется вполне допустимым вывести из нее любое из следующих высказываний:

$$\text{King}(\text{John}) \wedge \text{Greedy}(\text{John}) \heartsuit \text{ Evil}(\text{John})$$

$$\text{King}(\text{Richard}) \wedge \text{Greedy}(\text{Richard}) \heartsuit \text{ Evil}(\text{Richard})$$

$$\text{King}(\text{Father}(\text{John})) \wedge \text{Greedy}(\text{Father}(\text{John})) \heartsuit \text{ Evil}(\text{Father}(\text{John}))$$

...

Согласно правилу \ni **конкретизации высказывания с квантором всеобщности** (сокращенно UI — Universal Instantiation), мы можем вывести логическим путем любое высказывание, полученное в результате подстановки **базового термина** (терма без переменных) вместо переменной, на которую распространяется квантор всеобщности¹. Чтобы записать это правило логического вывода формально, воспользуемся понятием **подстановки**, введенным в разделе 8.3. Допустим, что $\text{Subst}(\theta, \alpha)$ обозначает результат применения подстановки θ к высказыванию α . В таком случае данное правило для любой переменной v и базового термина σ можно записать следующим образом:

$$\frac{\forall v \alpha}{\text{Subst}(\{v/\sigma\}, \alpha)}$$

¹ Эти подстановки не следует путать с расширенными интерпретациями, которые использовались для определения семантики кванторов. В подстановке переменная заменяется термом (синтаксической конструкцией) для получения нового высказывания, тогда как любая интерпретация отображает некоторую переменную на объект в проблемной области.

Например, три высказывания, приведенные выше, получены с помощью подстановок $\{x/John\}$, $\{x/Richard\}$ и $\{x/Father(John)\}$.

Соответствующее правило \ni **конкретизации высказывания с квантором существования** (Existential Instantiation — EI) для квантора существования является немного более сложным. Для любых высказывания α , переменной v и константного символа k , который не появляется где-либо в базе знаний, имеет место следующее:

$$\frac{\exists v \alpha}{\text{Subst}(\{v/k\}, \alpha)}$$

Например, из высказывания

$$\exists x \text{Crown}(x) \wedge \text{OnHead}(x, \text{John})$$

можно вывести высказывание

$$\text{Crown}(C_1) \wedge \text{OnHead}(C_1, \text{John})$$

при условии, что константный символ C_1 не появляется где-либо в базе знаний. По сути, в этом высказывании с квантором существования указано, что существует некоторый объект, удовлетворяющий определенному условию, а в процессе конкретизации просто присваивается имя этому объекту. Естественно, что это имя не должно уже принадлежать другому объекту. В математике есть прекрасный пример: предположим, мы открыли, что имеется некоторое число, которое немного больше чем 2,71828 и которое удовлетворяет уравнению $d(x^y) / dy = x^y$ для x . Этому числу можно присвоить новое имя, такое как e , но было бы ошибкой присваивать ему имя существующего объекта, допустим, π . В логике такое новое имя называется \ni **сколемовской константой**. Конкретизация высказывания с квантором существования — это частный случай более общего процесса, называемого **сколемизацией**, который рассматривается в разделе 9.5.

Конкретизация высказывания с квантором существования не только сложнее, чем конкретизация высказывания с квантором всеобщности, но и играет в логическом выводе немного иную роль. Конкретизация высказывания с квантором всеобщности может применяться много раз для получения многих разных заключений, а конкретизация высказывания с квантором существования может применяться только один раз, а затем соответствующее высказывание с квантором существования может быть отброшено. Например, после того как в базу знаний будет добавлено высказывание $\text{Kill}(\text{Murderer}, \text{Victim})$, становится больше не нужным высказывание $\exists x \text{Kill}(x, \text{Victim})$. Строго говоря, новая база знаний логически не эквивалентна старой, но можно показать, что она \ni **эквивалентна с точки зрения логического вывода**, в том смысле, что она выполнима тогда и только тогда, когда выполнима первоначальная база знаний.

Приведение к пропозициональному логическому выводу

Получив в свое распоряжение правила вывода высказываний с кванторами из высказываний без кванторов, мы получаем возможность привести вывод в логике первого порядка к выводу в пропозициональной логике. В данном разделе будут изложены основные идеи этого процесса, а более подробные сведения приведены в разделе 9.5.

Основная идея состоит в следующем: по аналогии с тем, как высказывание с квантором существования может быть заменено одной конкретизацией, высказы-

вание с квантором всеобщности может быть заменено множеством всех возможных конкретизаций. Например, предположим, что наша база знаний содержит только такие высказывания:

$$\begin{aligned} & \forall x \text{ King}(x) \wedge \text{Greedy}(x) \heartsuit \text{ Evil}(x) \\ & \text{King}(\text{John}) \\ & \text{Greedy}(\text{John}) \\ & \text{Brother}(\text{Richard}, \text{John}) \end{aligned} \quad (9.1)$$

Затем применим правило конкретизации высказывания с квантором всеобщности к первому высказыванию, используя все возможные подстановки базовых термов из словаря этой базы знаний — в данном случае $\{x/\text{John}\}$ и $\{x/\text{Richard}\}$. Мы получим следующие высказывания:

$$\begin{aligned} & \text{King}(\text{John}) \wedge \text{Greedy}(\text{John}) \heartsuit \text{ Evil}(\text{John}) \\ & \text{King}(\text{Richard}) \wedge \text{Greedy}(\text{Richard}) \heartsuit \text{ Evil}(\text{Richard}) \end{aligned}$$

и отбросим высказывание с квантором всеобщности. Теперь база знаний становится по сути пропозициональной, если базовые атомарные высказывания ($\text{King}(\text{John})$, $\text{Greedy}(\text{John})$ и т.д.) рассматриваются как пропозициональные символы. Поэтому теперь можно применить любой из алгоритмов полного пропозиционального вывода из главы 7 для получения таких заключений, как $\text{Evil}(\text{John})$.

Как показано в разделе 9.5, такой метод \heartsuit **пропозиционализации** (преобразования в высказывания пропозициональной логики) может стать полностью обобщенным; это означает, что любую базу знаний и любой запрос в логике первого порядка можно пропозиционализировать таким образом, чтобы сохранялось логическое следствие. Таким образом, имеется полная процедура принятия решения в отношении того, сохраняется ли логическое следствие... или, возможно, такой процедуры нет. Дело в том, что существует такая проблема: если база знаний включает функциональный символ, то множество возможных подстановок базовых термов становится бесконечным! Например, если в базе знаний упоминается символ Father , то существует возможность сформировать бесконечно большое количество вложенных термов, таких как $\text{Father}(\text{Father}(\text{Father}(\text{John})))$. А применяемые нами пропозициональные алгоритмы сталкиваются с затруднениями при обработке бесконечно большого множества высказываний.

К счастью, имеется знаменитая теорема, предложенная Жаком Эрбраном [650], согласно которой, если некоторое высказывание следует из первоначальной базы знаний в логике первого порядка, то существует доказательство, которое включает лишь конечное подмножество этой пропозиционализированной базы знаний. Поскольку любое такое подмножество имеет максимальную глубину вложения среди его базовых термов, это подмножество можно найти, формируя вначале все конкретизации с константными символами (Richard и John), затем все термы с глубиной 1 ($\text{Father}(\text{Richard})$ и $\text{Father}(\text{John})$), после этого все термы с глубиной 2 и т.д. до тех пор, пока мы не сможем больше составить пропозициональное доказательство высказывания, которое следует из базы знаний.

Выше был кратко описан один из подходов к организации вывода в логике первого порядка с помощью пропозиционализации, который является **полным**, т.е. позволяет доказать любое высказывание, которое следует из базы знаний. Это — важное достижение, если учесть, что пространство возможных моделей является бесконечным. С другой стороны, до тех пор пока это доказательство не составлено, мы не

знаем, следует ли данное высказывание из базы знаний! Что произойдет, если это высказывание из нее не следует? Можем ли мы это определить? Как оказалось, для логики первого порядка это действительно невозможно. Наша процедура доказательства может продолжаться и продолжаться, вырабатывая все более и более глубоко вложенные термы, а мы не будем знать, вошла ли она в безнадежный цикл или до получения доказательства остался только один шаг. Такая проблема весьма напоминает проблему останова машин Тьюринга. Алан Тьюринг [1518] и Алонсо Черч [255] доказали неизбежность такого состояния дел, хотя и весьма различными способами.

☞ *Вопрос о следствии для логики первого порядка является полуразрешимым; это означает, что существуют алгоритмы, которые позволяют найти доказательство для любого высказывания, которое следует из базы знаний, но нет таких алгоритмов, которые позволяли бы также определить, что не существует доказательства для каждого высказывания, которое не следует из базы знаний.*

9.2. УНИФИКАЦИЯ И ПОДНЯТИЕ

В предыдущем разделе описан уровень понимания процесса вывода в логике первого порядка, который существовал вплоть до начала 1960-х годов. Внимательный читатель (и, безусловно, специалисты в области вычислительной логики, работавшие в начале 1960-х годов) должен был заметить, что подход на основе пропозиционализации является довольно неэффективным. Например, если заданы запрос $Evil(x)$ и база знаний, приведенная в уравнении 9.1, то становится просто нерациональным формирование таких высказываний, как $King(Richard) \wedge Greedy(Richard) \heartsuit Evil(Richard)$. И действительно, для любого человека вывод факта $Evil(John)$ из следующих высказываний кажется вполне очевидным:

$$\begin{aligned} & \forall x King(x) \wedge Greedy(x) \heartsuit Evil(x) \\ & King(John) \\ & Greedy(John) \end{aligned}$$

Теперь мы покажем, как сделать его полностью очевидным для компьютера.

Правило вывода в логике первого порядка

Процедура вывода того факта, что Джон — злой, действует следующим образом: найти некоторый x , такой, что x — король и x — жадный, а затем вывести, что x — злой. Вообще говоря, если существует некоторая подстановка θ , позволяющая сделать предпосылку импликации идентичной высказываниям, которые уже находятся в базе знаний, то можно утверждать об истинности заключения этой импликации после применения θ . В данном случае такой цели достигает подстановка $\{x/John\}$.

Фактически можно обеспечить выполнение на этом этапе вывода еще больше работы. Предположим, что нам известно не то, что жаден Джон — $Greedy(John)$, а что жадными являются все:

$$\forall y Greedy(y) \tag{9.2}$$

Но и в таком случае нам все равно хотелось бы иметь возможность получить заключение, что Джон зол — $Evil(John)$, поскольку нам известно, что Джон — ко-

роль (это дано) и Джон жаден (так как жадными являются все). Для того чтобы такой метод мог работать, нам нужно найти подстановку как для переменных в высказывании с импликацией, так и для переменных в высказываниях, которые должны быть согласованы. В данном случае в результате применения подстановки $\{x/John, y/John\}$ к предпосылкам импликации $King(x)$ и $Greedy(x)$ и к высказываниям из базы знаний $King(John)$ и $Greedy(y)$ эти высказывания становятся идентичными. Таким образом, теперь можно вывести заключение импликации.

Такой процесс логического вывода может быть представлен с помощью единственного правила логического вывода, которое будет именоваться \ni **обобщенным правилом отделения** (Generalized Modus Ponens): для атомарных высказываний p_i , p_i' и q , если существует подстановка θ , такая, что $Subst(\theta, p_i') = Subst(\theta, p_i)$, то для всех i имеет место следующее:

$$\frac{p_1', p_2', \dots, p_n', (p_1 \wedge p_2 \wedge \dots \wedge p_n \heartsuit q)}{Subst(\theta, q)}$$

В этом правиле имеется $n+1$ предпосылка: n атомарных высказываний p_i' и одна импликация. Заключение становится результатом применения подстановки θ к следствию q . В данном примере имеет место следующее:

$$\begin{array}{ll} p_1' - \text{это } King(John) & p_1 - \text{это } King(x) \\ p_2' - \text{это } Greedy(y) & p_2 - \text{это } Greedy(x) \\ \theta - \text{это } \{x/John, y/John\} & q - \text{это } Evil(x) \\ Subst(\theta, q) - \text{это } Evil(John) & \end{array}$$

Можно легко показать, что обобщенное правило отделения — непротиворечивое правило логического вывода. Прежде всего отметим, что для любого высказывания p (в отношении которого предполагается, что на его переменные распространяется квантор всеобщности) и для любой подстановки θ справедливо следующее правило:

$$p \models Subst(\theta, p)$$

Это правило выполняется по тем же причинам, по которым выполняется правило конкретизации высказывания с квантором всеобщности. Оно выполняется, в частности, в любой подстановке θ , которая удовлетворяет условиям обобщенного правила отделения. Поэтому из p_1', \dots, p_n' можно вывести следующее:

$$Subst(\theta, p_1') \wedge \dots \wedge Subst(\theta, p_n')$$

а из импликации $p_1 \wedge \dots \wedge p_n \heartsuit q$ — следующее:

$$Subst(\theta, p_1) \wedge \dots \wedge Subst(\theta, p_n) \heartsuit Subst(\theta, q)$$

Теперь подстановка θ в обобщенном правиле отделения определена так, что $Subst(\theta, p_i') = Subst(\theta, p_i)$ для всех i , поэтому первое из этих двух высказываний точно совпадает с предпосылкой второго высказывания. Таким образом, выражение $Subst(\theta, q)$ следует из правила отделения.

Как принято выражаться в логике, обобщенное правило отделения представляет собой \ni **поднятую** версию правила отделения — оно поднимает правило отделения из пропозициональной логики в логику первого порядка. В оставшейся части этой главы будет показано, что могут быть разработаны поднятые версии алгоритмов прямого логического вывода, обратного логического вывода и резолюции, представленных в главе 7. Основным преимуществом применения поднятых правил логиче-

ского вывода по сравнению с пропозиционализацией является то, что в них предусмотрены только те подстановки, которые требуются для обеспечения дальнейшего выполнения конкретных логических выводов. Единственное соображение, которое может вызвать недоумение у читателя, состоит в том, что в определенном смысле обобщенное правило вывода является менее общим, чем исходное правило отделения (с. 303): правило отделения допускает применение в левой части импликации любого отдельно взятого высказывания α , а обобщенное правило отделения требует, чтобы это высказывание имело специальный формат. Но оно является обобщенным в том смысле, что допускает применение любого количества выражений p_i .

Унификация

Применение поднятых правил логического вывода связано с необходимостью поиска подстановок, в результате которых различные логические выражения становятся идентичными. Этот процесс называется λ **унификацией** и является ключевым компонентом любых алгоритмов вывода в логике первого порядка. Алгоритм *Unify* принимает на входе два высказывания и возвращает для них λ **унификатор**, если таковой существует:

$$\text{Unify}(p, q) = \theta \text{ где } \text{Subst}(\theta, p) = \text{Subst}(\theta, q)$$

Рассмотрим несколько примеров того, как должен действовать алгоритм *Unify*. Предположим, что имеется запрос $\text{Knows}(\text{John}, x)$ — кого знает Джон? Некоторые ответы на этот запрос можно найти, отыскивая все высказывания в базе знаний, которые унифицируются с высказыванием $\text{Knows}(\text{John}, x)$. Ниже приведены результаты унификации с четырьмя различными высказываниями, которые могут находиться в базе знаний.

$$\begin{aligned} \text{Unify}(\text{Knows}(\text{John}, x), \text{Knows}(\text{John}, \text{Jane})) &= \{x/\text{Jane}\} \\ \text{Unify}(\text{Knows}(\text{John}, x), \text{Knows}(y, \text{Bill})) &= \{x/\text{Bill}, y/\text{John}\} \\ \text{Unify}(\text{Knows}(\text{John}, x), \text{Knows}(y, \text{Mother}(y))) &= \{y/\text{John}, x/\text{Mother}(\text{John})\} \\ \text{Unify}(\text{Knows}(\text{John}, x), \text{Knows}(x, \text{Elizabeth})) &= \text{fail} \end{aligned}$$

Последняя попытка унификации оканчивается неудачей (*fail*), поскольку переменная x не может одновременно принимать значения *John* и *Elizabeth*. Теперь вспомним, что высказывание $\text{Knows}(x, \text{Elizabeth})$ означает “Все знают Элизабет”, поэтому мы обязаны иметь возможность вывести логически, что Джон знает Элизабет. Проблема возникает только потому, что в этих двух высказываниях, как оказалось, используется одно и то же имя переменной, x . Возникновения этой проблемы можно избежать, λ **стандартизируя отличие** (*standardizing apart*) одного из этих двух унифицируемых высказываний; под этой операцией подразумевается переименование переменных в высказываниях для предотвращения коллизий имен. Например, переменную x в высказывании $\text{Knows}(x, \text{Elizabeth})$ можно переименовать в z_{17} (новое имя переменной), не меняя смысл этого высказывания. После этого унификация выполняется успешно:

$$\text{Unify}(\text{Knows}(\text{John}, x), \text{Knows}(z_{17}, \text{Elizabeth})) = \{x/\text{Elizabeth}, z_{17}/\text{John}\}$$

С дополнительными сведениями о том, с чем связана необходимость в стандартизации отличия, можно ознакомиться в упр. 9.7.

Возникает еще одна сложность: выше было сказано, что алгоритм `Unify` должен возвращать такую подстановку (или унификатор), в результате которой два параметра становятся одинаковыми. Но количество таких унификаторов может быть больше единицы. Например, вызов алгоритма `Unify(Knows(John, x), Knows(y, z))` может вернуть $\{y/John, x/z\}$ или $\{y/John, x/John, z/John\}$. Первый унификатор позволяет получить в качестве результата унификации выражение `Knows(John, z)`, а второй дает `Knows(John, John)`. Но второй результат может быть получен из первого с помощью дополнительной подстановки $\{z/John\}$; в таком случае принято считать, что первый унификатор является более общим по сравнению со вторым, поскольку налагает меньше ограничений на значения переменных. Как оказалось, для любой унифицируемой пары выражений существует единственный **наиболее общий унификатор** (Most General Unifier — MGU), который является уникальным вплоть до переименования переменных. В данном случае таковым является $\{y/John, x/z\}$.

Алгоритм вычисления наиболее общих унификаторов приведен в листинге 9.1. Процесс его работы очень прост: рекурсивно исследовать два выражения одновременно, “бок о бок”, наряду с этим формируя унификатор, но создавать ситуацию неудачного завершения, если две соответствующие точки в полученных таким образом структурах не совпадают. При этом существует один дорогостоящий этап: если переменная согласуется со сложным термом, необходимо провести проверку того, встречается ли сама эта переменная внутри терма; в случае положительного ответа на данный вопрос согласование оканчивается неудачей, поскольку невозможно сформировать какой-либо совместимый унификатор. Из-за этой так называемой **проверки вхождения** (occure check) сложность всего алгоритма становится квадратично зависимой от размера унифицируемых выражений. В некоторых системах, включая все системы логического программирования, просто исключается такая проверка вхождения и поэтому в результате иногда формируются противоречивые логические выводы, а в других системах используются более развитые алгоритмы со сложностью, линейно зависящей от времени.

Листинг 9.1. Алгоритм унификации. Алгоритм действует путем поэлементного сравнения структур входных высказываний. В ходе этого формируется подстановка θ , которая также является параметром функции `Unify` и используется для проверки того, что дальнейшие сравнения совместимы со связываниями, которые были определены ранее. В составном выражении, таком как $F(A, B)$, функция `Op` выбирает функциональный символ F , а функция `Args` выбирает список параметров (A, B)

```
function Unify(x, y,  $\theta$ ) returns подстановка, позволяющая сделать
    x и y идентичными
inputs: x, переменная, константа, список или составной терм
        y, переменная, константа, список или составной терм
         $\theta$ , подстановка, подготовленная до сих пор (необязательный
            параметр, по умолчанию применяется пустой терм)

if  $\theta = failure$  then return failure
else if  $x = y$  then return  $\theta$ 
else if Variable?(x) then return Unify-Var(x, y,  $\theta$ )
else if Variable?(y) then return Unify-Var(y, x,  $\theta$ )
else if Compound?(x) and Compound?(y) then
```



```

        return Unify(Args[x], Args[y], Unify(Op[x], Op[y],  $\theta$ ))
    else if List?(x) and List?(y) then
        return Unify(Rest[x], Rest[y], Unify(First[x], First[y],  $\theta$ ))
    else return failure

function Unify-Var(var, x,  $\theta$ ) returns подстановка
    inputs: var, переменная
           x, любое выражение
            $\theta$ , подстановка, подготовленная до сих пор

    if {var/val}  $\in$   $\theta$  then return Unify(val, x,  $\theta$ )
    else if {x/val}  $\in$   $\theta$  then return Unify(var, val,  $\theta$ )
    else if Occur-Check?(var, x) then return failure
    else return добавление {var/x} к подстановке  $\theta$ 

```

Хранение и выборка

В основе функций Tell и Ask, применяемых для ввода информации и передачи запросов в базу знаний, лежат более примитивные функции Store и Fetch. Функция Store(*s*) сохраняет некоторое высказывание *s* в базе знаний, а функция Fetch(*q*) возвращает все унификаторы, такие, что запрос *q* унифицируется с некоторым высказыванием из базы знаний. Описанная выше задача, служившая для иллюстрации процесса унификации (поиск всех фактов, которые унифицируются с высказыванием Knows(*John*, *x*)), представляет собой пример применения функции Fetch.

Проще всего можно реализовать функции Store и Fetch, предусмотрев хранение всех фактов базы знаний в виде одного длинного списка, чтобы затем, после получения запроса *q*, можно было просто вызывать алгоритм Unify(*q*, *s*) для каждого высказывания *s* в списке. Такой процесс является неэффективным, но он осуществим, и знать об этом — это все, что нужно для понимания последней части данной главы. А в оставшейся части данного раздела описаны способы, позволяющие обеспечить более эффективную выборку, и он может быть пропущен при первом чтении.

Функцию Fetch можно сделать более эффективной, обеспечив, чтобы попытки унификации применялись только к высказываниям, имеющим определенный шанс на унификацию. Например, нет смысла пытаться унифицировать Knows(*John*, *x*) и Brother(*Richard*, *John*). Такой унификации можно избежать, \sphericalangle **индексируя** факты в базе знаний. Самая простая схема, называемая \sphericalangle **индексацией по предикатам**, предусматривает размещение всех фактов Knows в одном сегменте, а всех фактов Brother — в другом. Сами сегменты для повышения эффективности доступа можно хранить в хэш-таблице².

Индексация по предикатам является удобной, когда имеется очень много предикатных символов, но лишь небольшое количество выражений в расчете на каждый символ. Однако в некоторых приложениях имеется много выражений в расчете на

² Хэш-таблица — это структура данных для хранения и выборки информации, индексируемой с помощью фиксированных ключей. С точки зрения практики хэш-таблица может рассматриваться как имеющая постоянные временные показатели хранения и выборки, даже если эта таблица содержит очень большое количество элементов.

каждый конкретный предикатный символ. Например, предположим, что налоговые органы желают следить за тем, кто кого нанимает, с использованием предиката $Employs(x, y)$. Такой сегмент, возможно, состоящий из миллионов нанимателей и десятков миллионов наемных работников, был бы очень большим. Для поиска ответа на такой запрос, как $Employs(x, Richard)$ (“Кто является нанимателем Ричарда?”), при использовании индексации по предикатам потребовался бы просмотр всего сегмента.

Поиск ответа на данный конкретный запрос стал бы проще при использовании индексации фактов и по предикату, и по второму параметру, возможно, с использованием комбинированного ключа хэш-таблицы. В таком случае существовала бы возможность просто формировать ключ из запроса и осуществлять выборку именно тех фактов, которые унифицируются с этим запросом. А для ответа на другие запросы, такие как $Employs(AIMA.org, y)$, нужно было бы индексировать факты, комбинируя предикат с первым параметром. Поэтому факты могут храниться под разными индексными ключами, что позволяет моментально сделать их доступными для разных запросов, с которыми они могли бы унифицироваться.

Если дано некоторое высказывание, которое подлежит хранению, то появляется возможность сформировать индексы для всех возможных запросов, которые унифицируются с ними. Применительно к факту $Employs(AIMA.org, Richard)$ возможны следующие запросы:

$Employs(AIMA.org, Richard)$	Является ли организация AIMA.org нанимателем Ричарда?
$Employs(x, Richard)$	Кто является нанимателем Ричарда?
$Employs(AIMA.org, y)$	Для кого является нанимателем организация AIMA.org?
$Employs(x, y)$	Кто для кого является нанимателем?

Как показано на рис. 9.1, а, эти запросы образуют **решетку обобщения**. Такая решетка обладает некоторыми интересными свойствами. Например, дочерний узел любого узла в этой решетке может быть получен из его родительского узла с помощью единственной подстановки, а “наибольший” общий потомок любых двух узлов является результатом применения наиболее общего унификатора для этих узлов. Та часть решетки, которая находится выше любого базового факта, может быть сформирована систематически (упр. 9.5). Высказывание с повторяющимися константами имеет несколько иную решетку, как показано на рис. 9.1, б. Наличие функциональных символов и переменных в высказываниях, подлежащих хранению, приводит к появлению еще более интересных структур решетки.

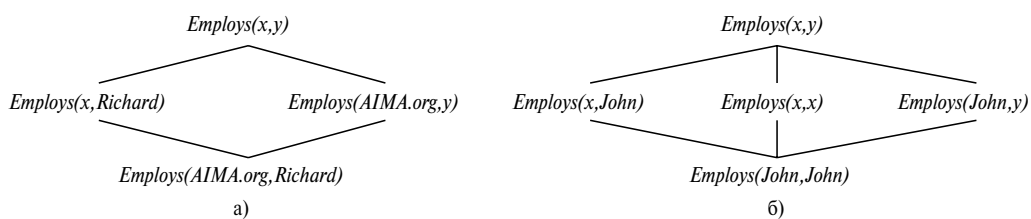


Рис. 9.1. Примеры решеток обобщения: решетка обобщения, самым нижним узлом которой является высказывание $Employs(AIMA.org, Richard)$ (а); решетка обобщения для высказывания $Employs(John, John)$ (б)

Только что описанная схема применяется очень успешно, если решетка содержит небольшое количество узлов. А если предикат имеет n параметров, то решетка включает $O(2^n)$ узлов. Если же разрешено применение функциональных символов, то количество узлов также становится экспоненциально зависимым от размера термов в высказывании, подлежащем хранению. Это может вызвать необходимость создания огромного количества индексов. В какой-то момент затраты на хранение и сопровождение всех этих индексов перевесят преимущества индексации. Для выхода из этой ситуации можно применять какой-либо жесткий подход, например сопровождать индексы только на ключах, состоящих из некоторого предиката плюс каждый параметр, или адаптивный подход, в котором предусматривается создание индексов в соответствии с потребностями в поиске ответов на запросы того типа, которые встречаются наиболее часто. В большинстве систем искусственного интеллекта количество фактов, подлежащих хранению, является достаточно небольшим для того, чтобы проблему эффективной индексации можно было считать решенной. А что касается промышленных и коммерческих баз данных, то эта проблема стала предметом значительных и продуктивных технологических разработок.

9.3. ПРЯМОЙ ЛОГИЧЕСКИЙ ВЫВОД

Алгоритм прямого логического вывода для пропозициональных определенных выражений приведен в разделе 7.5. Его идея проста: начать с атомарных высказываний в базе знаний и применять правило отделения в прямом направлении, добавляя все новые и новые атомарные высказывания до тех пор, пока не возникнет ситуация, в которой невозможно будет продолжать формулировать логические выводы. В данном разделе приведено описание того, как можно применить этот алгоритм к определенным выражениям в логике первого порядка и каким образом он может быть реализован эффективно. Определенные выражения, такие как *Situation*♥*Response*, особенно полезны для систем, в которых логический вывод осуществляется в ответ на вновь поступающую информацию. Таким образом могут быть определены многие системы, а формирование рассуждений с помощью прямого логического вывода может оказаться гораздо более эффективным по сравнению с доказательством теорем с помощью резолюции. Поэтому часто имеет смысл попытаться сформировать базу знаний с использованием только определенных выражений, чтобы избежать издержек, связанных с резолюцией.

Определенные выражения в логике первого порядка

Определенные выражения в логике первого порядка весьма напоминают определенные выражения в пропозициональной логике (с. 312): они представляют собой дизъюнкции литералов, среди которых положительным является один и только один. Определенное выражение либо является атомарным, либо представляет собой импликацию, антецедентом (предпосылкой) которой служит конъюнкция положительных литералов, а консеквентом (следствием) — единственный положительный литерал. Ниже приведены примеры определенных выражений в логике первого порядка.

$$\begin{aligned} & King(x) \wedge Greedy(x) \heartsuit Evil(x) \\ & King(John) \\ & Greedy(y) \end{aligned}$$

В отличие от пропозициональных литералов, литералы первого порядка могут включать переменные, и в таком случае предполагается, что на эти переменные распространяется квантор всеобщности. (Как правило, при написании определенных выражений кванторы всеобщности исключаются.) Определенные выражения представляют собой подходящую нормальную форму для использования в обобщенном правиле отделения.

Не все базы знаний могут быть преобразованы в множество определенных выражений из-за того ограничения, что положительный литерал в них должен быть единственным, но для многих баз знаний такая возможность существует. Рассмотрим приведенную ниже задачу.

Закон гласит, что продажа оружия недружественным странам, осуществляемая любым американским гражданином, считается преступлением. В государстве Ноуноу, враждебном по отношению к Америке, имеются некоторые ракеты, и все ракеты этого государства были проданы ему полковником Уэстом, который является американским гражданином.

Мы должны доказать, что полковник Уэст совершил преступление. Вначале все имеющиеся факты будут представлены в виде определенных выражений в логике первого порядка, а в следующем разделе будет показано, как решить эту задачу с помощью алгоритма прямого логического вывода.

- “...продажа оружия враждебным странам, осуществляемая любым американским гражданином, является преступлением”:

$$\begin{aligned} & American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \\ & \heartsuit Criminal(x) \end{aligned} \quad (9.3)$$

- “В государстве Ноуноу... имеются некоторые ракеты”. Высказывание $\exists x Owns(Nono, x) \wedge Missile(x)$ преобразуется в два определенных выражения путем устранения квантора существования и введения новой константы M_1 :

$$Owns(Nono, M_1) \quad (9.4)$$

$$Missile(M_1) \quad (9.5)$$

- “...все ракеты этого государства были проданы ему полковником Уэстом”:

$$Missile(x) \wedge Owns(Nono, x) \heartsuit Sells(West, x, Nono) \quad (9.6)$$

Нам необходимо также знать, что ракеты — оружие:

$$Missile(x) \heartsuit Weapon(x) \quad (9.7)$$

Кроме того, мы должны знать, что государство, враждебное по отношению к Америке, рассматривается как “недружественное”:

$$Enemy(x, America) \heartsuit Hostile(x) \quad (9.8)$$

- “...полковником Уэстом, который является американским гражданином”:

$$American(West) \quad (9.9)$$

- “В государстве Ноуноу, враждебном по отношению к Америке...”:

$$Enemy(Nono, America) \quad (9.10)$$

Эта база знаний не содержит функциональных символов и поэтому может служить примером класса баз знаний языка λ **Datalog**, т.е. примером множества определенных выражений в логике первого порядка без функциональных символов. Ниже будет показано, что при отсутствии функциональных символов логический вывод становится намного проще.

Простой алгоритм прямого логического вывода

Как показано в листинге 9.2, первый рассматриваемый нами алгоритм прямого логического вывода является очень простым. Начиная с известных фактов, он активизирует все правила, предпосылки которых выполняются, и добавляет заключения этих правил к известным фактам. Этот процесс продолжается до тех пор, пока не обнаруживается ответ на запрос (при условии, что требуется только один ответ) или больше не происходит добавление новых фактов. Следует отметить, что факт не является “новым”, если он представляет собой λ **переименование** известного факта. Одно высказывание называется переименованием другого, если оба эти высказывания идентичны во всем, за исключением имен переменных. Например, высказывания $Likes(x, IceCream)$ и $Likes(y, IceCream)$ представляют собой переименования по отношению друг к другу, поскольку они отличаются лишь выбором имени переменной, x или y ; они имеют одинаковый смысл — все любят мороженое.

Листинг 9.2. Концептуально простой, но очень неэффективный алгоритм прямого логического вывода. В каждой итерации он добавляет к базе знаний **KB** все атомарные высказывания, которые могут быть выведены за один этап из импликационных высказываний и атомарных высказываний, которые уже находятся в базе знаний

```

function FOL-FC-Ask(KB,  $\alpha$ ) returns подстановка или значение false
  inputs: KB, база знаний - множество определенных выражений
            первого порядка
             $\alpha$ , запрос - атомарное высказывание
  local variables: new, новые высказывания, выводимые
                    в каждой итерации

  repeat until множество new не пусто
    new  $\leftarrow$  {}
    for each высказывание r in KB do
      ( $p_1 \wedge \dots \wedge p_n \heartsuit q$ )  $\leftarrow$  Standardize-Apart(r)
      for each подстановка  $\theta$ , такая что  $\text{Subst}(\theta, p_1 \wedge \dots \wedge p_n) =$ 
         $\text{Subst}(\theta, p_1' \wedge \dots \wedge p_n')$  для некоторых  $p_1', \dots, p_n'$ 
        в базе знаний KB
         $q' \leftarrow \text{Subst}(\theta, q)$ 
        if высказывание  $q'$  не является переименованием
          некоторого высказывания, которое уже находится
          в KB, или рассматривается как элемент множества
          new then do
            добавить  $q'$  к множеству new
             $\phi \leftarrow \text{Unify}(q', \alpha)$ 
            if значение  $\phi$  не представляет собой fail
              then return  $\phi$ 
            добавить множество new к базе знаний KB
  return false

```

Для иллюстрации работы алгоритма FOL-FC-Ask воспользуемся описанной выше задачей доказательства преступления. Импликационными высказываниями являются высказывания, приведенные в уравнениях 9.3, 9.6–9.8. Требуются следующие две итерации.

- В первой итерации правило 9.3 имеет невыполненные предпосылки. Правило 9.6 выполняется с подстановкой $\{x/M_1\}$ и добавляется высказывание $Sells(West, M_1, Nono)$. Правило 9.7 выполняется с подстановкой $\{x/M_1\}$ и добавляется высказывание $Weapon(M_1)$. Правило 9.8 выполняется с подстановкой $\{x/Nono\}$ и добавляется высказывание $Hostile(Nono)$.
- На второй итерации правило 9.3 выполняется с подстановкой $\{x/West, y/M_1, z/Nono\}$ и добавляется высказывание $Criminal(West)$.

Сформированное дерево доказательства показано на рис. 9.2. Обратите внимание на то, что в этот момент невозможны какие-либо новые логические выводы, поскольку каждое высказывание, заключение которого можно было бы найти с помощью прямого логического вывода, уже явно содержится в базе знаний KB. Такое состояние базы знаний называется **фиксированной точкой** (fixed point) в процессе логического вывода. Фиксированные точки, достигаемые при прямом логическом выводе с использованием определенных выражений первого порядка, аналогичны фиксированным точкам, возникающим при пропозициональном прямом логическом выводе (с. 315); основное различие состоит в том, что фиксированная точка первого порядка может включать атомарные высказывания с квантором всеобщности.

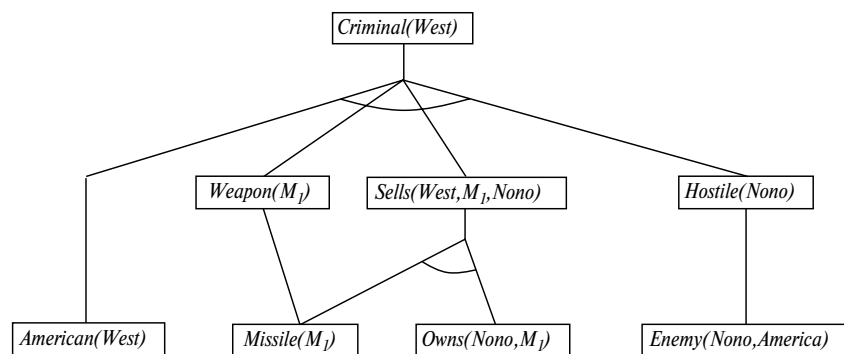


Рис. 9.2. Дерево доказательства, сформированное путем прямого логического вывода в примере доказательства преступления. Первоначальные факты показаны на нижнем уровне, факты, выведенные логическим путем в первой итерации, — на среднем уровне, а факт, логически выведенный во второй итерации, — на верхнем уровне

Свойства алгоритма FOL-FC-Ask проанализировать несложно. Во-первых, он является **непротиворечивым**, поскольку каждый этап логического вывода представляет собой применение обобщенного правила отдаления, которое само является непротиворечивым. Во-вторых, он является **полным** применительно к базам знаний

с определенными выражениями; это означает, что он позволяет ответить на любой запрос, ответы на который следуют из базы знаний с определенными выражениями. Для баз знаний Datalog, которые не содержат функциональных символов, доказательство полноты является довольно простым. Начнем с подсчета количества возможных фактов, которые могут быть добавлены, определяющего максимальное количество итераций. Допустим, что k — максимальная **арность** (количество параметров) любого предиката, p — количество предикатов и n — количество константных символов. Очевидно, что может быть не больше чем pn^k различных базовых фактов, поэтому алгоритм должен достичь фиксированной точки именно после стольких итераций. В таком случае можно применить обоснование приведенного выше утверждения, весьма аналогичное доказательству полноты пропозиционального прямого логического вывода (см. с. 315). Подробные сведения о том, как осуществить переход от пропозициональной полноты к полноте первого порядка, приведены применительно к алгоритму резолюции в разделе 9.5.

При его использовании к более общим определенным выражениям с функциональными символами алгоритм FOL-FC-Ask может вырабатывать бесконечно большое количество новых фактов, поэтому необходимо соблюдать исключительную осторожность. Для того случая, в котором из базы знаний следует ответ на высказывание запроса α , необходимо прибегать к использованию теоремы Эрбрана для обеспечения того, чтобы алгоритм мог найти доказательство (случай, касающийся резолюции, описан в разделе 9.5). А если запрос не имеет ответа, то в некоторых случаях может оказаться, что не удастся нормально завершить работу данного алгоритма. Например, если база знаний включает аксиомы Пеано:

$$\begin{aligned} & \text{NatNum}(0) \\ & \forall n \text{ NatNum}(n) \heartsuit \text{ NatNum}(S(n)) \end{aligned}$$

то в результате прямого логического вывода будут добавлены факты $\text{NatNum}(S(0))$, $\text{NatNum}(S(S(0)))$, $\text{NatNum}(S(S(S(0))))$ и т.д. Вообще говоря, избежать возникновения этой проблемы невозможно. Как и в общем случае логики первого порядка, задача определения того, следуют ли высказывания из базы знаний, сформированной с использованием определенных выражений, является полурезолюционной.

Эффективный прямой логический вывод

Алгоритм прямого логического вывода, приведенный в листинге 9.2, был спроектирован не с целью обеспечения эффективного функционирования, а, скорее, с целью упрощения его понимания. Существуют три возможных источника осложнений в его работе. Во-первых, “внутренний цикл” этого алгоритма предусматривает поиск всех возможных унификаторов, таких, что предпосылка некоторого правила унифицируется с подходящим множеством фактов в базе знаний. Такая операция часто именуется **согласованием с шаблоном** и может оказаться очень дорогостоящей. Во-вторых, в этом алгоритме происходит повторная проверка каждого правила в каждой итерации для определения того, выполняются ли его предпосылки, даже если в базу знаний в каждой итерации вносится лишь очень немного дополнений. В-третьих, этот алгоритм может вырабатывать много фактов, которые не имеют отношения к текущей цели. Устраним каждый из этих источников неэффективности по очереди.

Согласование правил с известными фактами

Проблема согласования предпосылки правила с фактами, хранящимися в базе знаний, может показаться достаточно простой. Например, предположим, что требуется применить следующее правило:

$$\text{Missile}(x) \vee \text{Weapon}(x)$$

Для этого необходимо найти все факты, которые согласуются с выражением $\text{Missile}(x)$; в базе знаний, индексированной подходящим образом, это можно выполнить за постоянное время в расчете на каждый факт. А теперь рассмотрим правило, подобное следующему:

$$\text{Missile}(x) \wedge \text{Owns}(\text{Nono}, x) \vee \text{Sells}(\text{West}, x, \text{Nono})$$

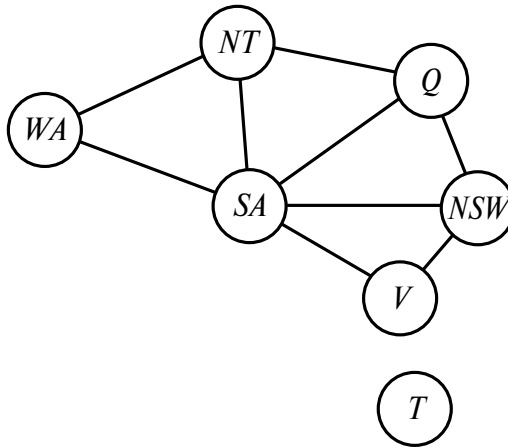
Найти все объекты, принадлежащие государству Ноуноу, опять-таки можно за постоянное время в расчете на каждый объект; затем мы можем применить к каждому объекту проверку, является ли он ракетой. Но если в базе знаний содержится много сведений об объектах, принадлежащих государству Ноуноу, и лишь немного данных о ракетах, то было бы лучше вначале найти все ракеты, а затем проверить, какие из них принадлежат Ноуноу. Это — проблема **упорядочения конъюнктов**: поиск упорядочения, позволяющего решать конъюнкты в предпосылке правила таким образом, чтобы общая стоимость решения была минимальной. Как оказалось, задача поиска оптимального упорядочения является NP-трудной, но имеются хорошие эвристики. Например, эвристика с **наиболее ограниченной переменной**, применявшаяся при решении задач CSP в главе 5, подсказывает, что необходимо упорядочить конъюнкты так, чтобы вначале проводился поиск ракет, если количество ракет меньше по сравнению с количеством всех известных объектов, принадлежащих государству Ноуноу.

Между процедурами согласования с шаблоном и удовлетворения ограничений действительно существует очень тесная связь. Каждый конъюнкт может рассматриваться как ограничение на содержащиеся в нем переменные; например, $\text{Missile}(x)$ — это унарное ограничение на x . Развивая эту идею, можно прийти к выводу, что \iff *существует возможность представить любую задачу CSP с конечной областью определения как единственное определенное выражение наряду с некоторыми касающимися ее базовыми фактами*. Рассмотрим приведенную на рис. 5.1 задачу раскраски карты, которая снова показана на рис. 9.3, а. Эквивалентная формулировка в виде одного определенного выражения приведена на рис. 9.3, б. Очевидно, что заключение $\text{Colorable}()$ можно вывести из этой базы знаний, только если данная задача CSP имеет решение. А поскольку задачи CSP, вообще говоря, включают задачи 3-SAT в качестве частных случаев, на основании этого можно сделать вывод, что \iff *задача согласования определенного выражения с множеством фактов является NP-трудной*.

То, что во внутреннем цикле алгоритма прямого логического вывода приходится решать NP-трудную задачу согласования, может показаться на первый взгляд довольно неприятным. Тем не менее, есть следующие три фактора, благодаря которым эта проблема предстает немного в лучшем свете.

- Напомним, что большинство правил в базах знаний, применяемых на практике, являются небольшими и простыми (подобно правилам, используемым в примере доказательства преступления), а не большими и сложными (как в формулировке задачи CSP, приведенной на рис. 9.3). В мире пользователей

баз данных принято считать, что размеры правил и арности предикатов не превышают некоторого постоянного значения, и принимать во внимание только \approx **сложность данных**, т.е. сложность логического вывода как функции от количества базовых фактов в базе данных. Можно легко показать, что обусловленная данными сложность в прямом логическом выводе определяется полиномиальной зависимостью.



а)

$$\begin{aligned}
 & Diff(wa, nt) \wedge Diff(wa, sa) \wedge \\
 & Diff(nt, q) \wedge Diff(nt, sa) \wedge \\
 & Diff(q, nsw) \wedge Diff(q, sa) \wedge \\
 & Diff(nsw, v) \wedge Diff(nsw, sa) \wedge \\
 & Diff(v, sa) \Rightarrow Colorable() \\
 \\
 & Diff(Red, Blue) \quad Diff(Red, Green) \\
 & Diff(Green, Red) \quad Diff(Green, Blue) \\
 & Diff(Blue, Red) \quad Diff(Blue, Green)
 \end{aligned}$$

б)

Рис. 9.3. Иллюстрация связи между процессами согласования с шаблоном и удовлетворения ограничений: граф ограничений для раскрашивания карты Австралии (см. рис. 5.1) (а); задача CSP раскрашивания карты, представленная в виде единственного определенного выражения (б). Обратите внимание на то, что области определения переменных заданы неявно с помощью констант, приведенных в базовых фактах для предиката `Diff`

- Могут рассматриваться подклассы правил, для которых согласование является наиболее эффективным. По сути, каждое выражение на языке Datalog может рассматриваться как определяющее некоторую задачу CSP, поэтому согласование будет осуществимым только тогда, когда соответствующая задача CSP является разрешимой. Некоторые разрешимые семейства задач CSP описаны в главе 5. Например, если граф ограничений (граф, узлами которого являются переменные, а дугами — ограничения) образует дерево, то задача CSP может быть решена за линейное время. Точно такой же результат остается в силе для согласования с правилами. Например, если из карты, приведенной на рис. 9.3, будет удален узел `SA`, относящийся к Южной Австралии, то результирующее выражение примет следующий вид:

$$Diff(wa, nt) \wedge Diff(nt, q) \wedge Diff(q, nsw) \wedge Diff(nsw, v) \heartsuit Colorable()$$

что соответствует сокращенной задаче CSP, показанной на рис. 5.7. Для решения задачи согласования с правилами могут непосредственно применяться алгоритмы решения задач CSP с древовидной структурой.

- Можно приложить определенные усилия по устранению излишних попыток согласования с правилами в алгоритме прямого логического вывода, что является темой следующего раздела.

Инкрементный прямой логический вывод

Когда авторы демонстрировали в предыдущем разделе на примере доказательства преступления, как действует прямой логический вывод, они немного схитрили; в частности, не показали некоторые из согласований с правилами, выполняемые алгоритмом, приведенным в листинге 9.2. Например, во второй итерации правило

$$\text{Missile}(x) \heartsuit \text{Weapon}(x)$$

согласуется с фактом $\text{Missile}(M_1)$ (еще раз), и, безусловно, при этом ничего не происходит, поскольку заключение $\text{Weapon}(M_1)$ уже известно. Таких излишних согласований с правилами можно избежать, сделав следующее наблюдение: \heartsuit *каждый новый факт, выведенный в итерации t , должен быть получен по меньшей мере из одного нового факта, выведенного в итерации $t-1$* . Это наблюдение соответствует истине, поскольку любой логический вывод, который не требовал нового факта из итерации $t-1$, уже мог быть выполнен в итерации $t-1$.

Такое наблюдение приводит естественным образом к созданию алгоритма инкрементного прямого логического вывода, в котором в итерации t проверка правила происходит, только если его предпосылка включает конъюнкт p_i , который унифицируется с фактом p_i' , вновь выведенным в итерации $t-1$. Затем на этапе согласования с правилом значение p_i фиксируется для согласования с p_i' , но при этом допускается, чтобы остальные конъюнкты в правиле согласовывались с фактами из любой предыдущей итерации. Этот алгоритм в каждой итерации вырабатывает точно такие же факты, как и алгоритм, приведенный в листинге 9.2, но является гораздо более эффективным.

При использовании подходящей индексации можно легко выявить все правила, которые могут быть активизированы любым конкретным фактом. И действительно, многие реальные системы действуют в режиме “обновления”, при котором прямой логический вывод происходит в ответ на каждый новый факт, сообщенный системе с помощью операции Tell. Операции логического вывода каскадно распространяются через множество правил до тех пор, пока не достигается фиксированная точка, а затем процесс начинается снова, вслед за поступлением каждого нового факта.

Как правило, в результате добавления каждого конкретного факта в действительности активизируется лишь небольшая доля правил в базе знаний. Это означает, что при повторном конструировании частичных согласований с правилами, имеющими некоторые невыполненные предпосылки, выполняется существенный объем ненужной работы. Рассматриваемый здесь пример доказательства преступления слишком мал, чтобы на нем можно было наглядно показать такую ситуацию, но следует отметить, что частичное согласование конструируется в первой итерации между следующим правилом:

$$\text{American}(x) \wedge \text{Weapon}(y) \wedge \text{Sells}(x, y, z) \wedge \text{Hostile}(z) \heartsuit \text{Criminal}(x)$$

и фактом $\text{American}(\text{West})$. Затем это частичное согласование отбрасывается и снова формируется во второй итерации (в которой данное правило согласуется успешно). Было бы лучше сохранять и постепенно дополнять частичные согласования по мере поступления новых фактов, а не отбрасывать их.

В \heartsuit **rete-алгоритме**³ была впервые предпринята серьезная попытка решить эту проблему. Алгоритм предусматривает предварительную обработку множества пра-

³ Здесь rete — латинское слово, которое переводится как сеть и читается по-русски “рете”, а по-английски — “рити”.

вил в базе знаний для формирования своего рода сети потока данных (называемой *rete*-сетью), в которой каждый узел представляет собой литерал из предпосылки какого-либо правила. По этой сети распространяются операции связывания переменных и останавливаются после того, как в них не удастся выполнить согласование с каким-то литералом. Если в двух литералах некоторого правила совместно используется какая-то переменная (например, $Sells(x, y, z) \wedge Hostile(z)$ в примере доказательства преступления), то варианты связывания из каждого литерала пропускаются через узел проверки равенства. Процессу связывания переменных, достигших узла n -арного литерала, такого как $Sells(x, y, z)$, может потребоваться перейти в состояние ожидания того, что будут определены связывания для других переменных, прежде чем он сможет продолжаться. В любой конкретный момент времени состояние *rete*-сети охватывает все частичные согласования с правилами, что позволяет избежать большого объема повторных вычислений.

Не только сами *rete*-сети, но и различные их усовершенствования стали ключевым компонентом так называемых \approx **продукционных систем**, которые принадлежат к числу самых первых систем прямого логического вывода, получивших широкое распространение⁴. В частности, с использованием архитектуры продукционной системы была создана система *Xcon* (которая первоначально называлась *R1*) [1026]. Система *Xcon* содержала несколько тысяч правил и предназначалась для проектирования конфигураций компьютерных компонентов для заказчиков *Digital Equipment Corporation*. Ее создание было одним из первых очевидных успешных коммерческих проектов в развивающейся области экспертных систем. На основе той же базовой технологии, которая была реализована на языке общего назначения *Ops-5*, было также создано много других подобных систем.

Кроме того, продукционные системы широко применяются в \approx **когнитивных архитектурах** (т.е. моделях человеческого мышления), в таких как *ACT* [31] и *Soar* [880]. В подобных системах “рабочая память” системы моделирует кратковременную память человека, а продукции образуют часть долговременной памяти. В каждом цикле функционирования происходит согласование продукции с фактами из рабочей памяти. Продукции, условия которых выполнены, могут добавлять или удалять факты в рабочей памяти. В отличие от типичных ситуаций с большим объемом данных, наблюдаемых в базах данных, продукционные системы часто содержат много правил и относительно немного фактов. При использовании технологии согласования, оптимизированной должным образом, некоторые современные системы могут оперировать в реальном времени больше чем с миллионом правил.

Не относящиеся к делу факты

Последний источник неэффективности прямого логического вывода, по-видимому, свойствен самому этому подходу и также возникает в контексте пропозициональной логики (см. раздел 7.5). Прямой логический вывод предусматривает выполнение всех допустимых этапов логического вывода на основе всех известных фактов, даже если они не относятся к рассматриваемой цели. В примере доказательства преступления не было правил, способных приводить к заключениям, не относящимся к делу, поэтому такое отсутствие направленности не вызывало каких-либо проблем. В других случаях (например, если бы в базу знаний было внесено несколь-

⁴ Слово **продукция** в названии **продукционная система** обозначает правило “условие-действие”.

ко правил с описанием кулинарных предпочтений американцев и цен на ракеты) алгоритм FOL-FC-Ask выработывал бы много нерелевантных заключений.

Один из способов предотвращения формирования нерелевантных заключений состоит в использовании обратного логического вывода, как описано в разделе 9.4. Еще одно, второе решение состоит в том, чтобы ограничить прямой логический вывод избранным подмножеством правил; этот подход обсуждался в контексте пропозициональной логики. Третий подход сформировался в сообществе пользователей дедуктивных баз данных, для которых прямой логический вывод является стандартным инструментальным средством. Идея этого подхода состоит в том, чтобы перезаписывать множество правил с использованием информации из цели так, что в процессе прямого логического вывода рассматриваются только релевантные связывания переменных (принадлежащие к так называемому \bowtie **магическому множеству**). Например, если целью является $Criminal(West)$, то правило, приводящее к заключению $Criminal(x)$, может быть перезаписано для включения дополнительного конъюнкта, ограничивающего значение x , следующим образом:

$$Magic(x) \wedge American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge \\ Hostile(z) \heartsuit Criminal(x)$$

Факт $Magic(West)$ также добавляется в базу знаний. Благодаря этому в процессе прямого логического вывода будет рассматриваться только факт о полковнике Уэсте, даже если база знаний содержит факты о миллионах американцев. Полный процесс определения магических множеств и перезаписи базы знаний является слишком сложным для того, чтобы мы могли заняться в этой главе его описанием, но основная идея состоит в том, что выполняется своего рода “универсальный” обратный логический вывод от цели для выяснения того, какие связывания переменных нужно будет ограничивать. Поэтому подход с использованием магических множеств может рассматриваться как гибридный между прямым логическим выводом и обратной предварительной обработкой.

9.4. ОБРАТНЫЙ ЛОГИЧЕСКИЙ ВЫВОД

Во втором большом семействе алгоритмов логического вывода используется подход с **обратным логическим выводом**, представленный в разделе 7.5. Эти алгоритмы действуют в обратном направлении, от цели, проходя по цепочке от одного правила к другому, чтобы найти известные факты, которые поддерживают доказательство. Мы опишем основной алгоритм, а затем покажем, как он используется в **логическом программировании**, представляющем собой наиболее широко применяемую форму автоматизированного формирования рассуждений. В этом разделе будет также показано, что обратный логический вывод имеет некоторые недостатки по сравнению с прямым логическим выводом, и описаны некоторые способы преодоления этих недостатков. Наконец, будет продемонстрирована тесная связь между логическим программированием и задачами удовлетворения ограничений.

Алгоритм обратного логического вывода

Простой алгоритм обратного логического вывода, FOL-BC-Ask, приведен в листинге 9.3. Он вызывается со списком целей, содержащим единственный элемент

(первоначальный запрос) и возвращает множество всех подстановок, которые удовлетворяют этому запросу. Список целей можно рассматривать как “стек” целей, ожидающих отработки; если все они могут быть выполнены, то текущая ветвь доказательства формируется успешно. В алгоритме берется первая цель из списка и выполняется поиск в базе знаний всех выражений, положительный литерал которых (или **голова**) унифицируется с целью. При обработке каждого такого выражения создается новый рекурсивный вызов, в котором предпосылки (или **тело**) выражения добавляются к стеку целей. Напомним, что факты представляют собой выражения с головой, но без тела, поэтому, если какая-то цель унифицируется с известным фактом, то к стеку не добавляются какие-то подцели, а сама эта цель считается получившей решение. На рис. 9.4 показано дерево доказательства для получения факта $Criminal(West)$ из высказываний, приведенных в уравнениях 9.3–9.10.

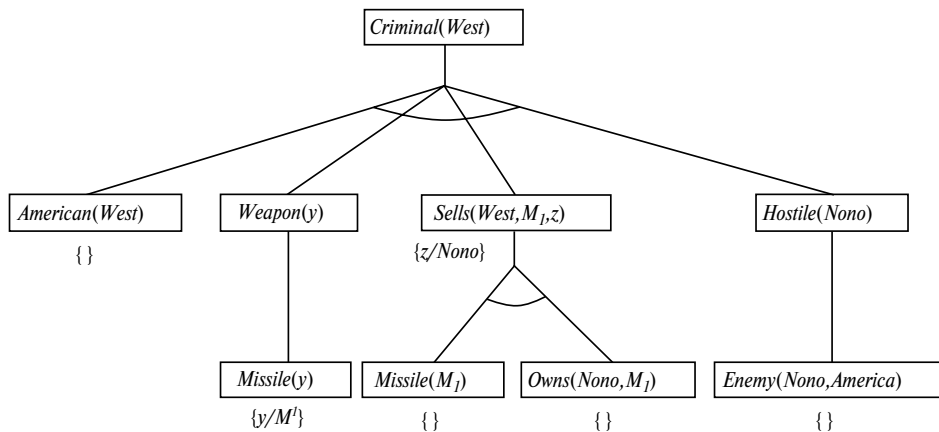


Рис. 9.4. Дерево доказательства, сформированное путем обратного логического вывода для доказательства того, что полковник Уэст совершил преступление. Это дерево следует читать в глубину, слева направо. Чтобы доказать факт $Criminal(West)$, необходимо доказать четыре конъюнкта, находящихся под ним. Некоторые из них находятся в базе знаний, а другие требуют дальнейшего обратного логического вывода. Связывания для каждой успешной унификации показаны после соответствующей подцели. Обратите внимание на, что после успешного достижения одной подцели в конъюнкции ее подстановка применяется для последующих подцелей. Таким образом, к тому времени, как алгоритм FOL-BC-Ask достигает последнего конъюнкта, первоначально имевшего форму $Hostile(z)$, переменная z уже связана с $Nono$

Листинг 9.3. Простой алгоритм обратного логического вывода

```

function FOL-BC-Ask(KB, goals,  $\theta$ ) returns множество подстановок
inputs: KB, база знаний
           goals, список конъюнктов, образующих запрос (подстановка  $\theta$ 
           уже применена)
            $\theta$ , текущая подстановка, первоначально пустая подстановка {}
local variables: answers, ответы - множество подстановок,
                   первоначально пустое

if список goals пуст then return {}
 $q' \leftarrow \text{Subst}(\theta, \text{First}(\text{goals}))$ 
  
```

```

for each высказывание  $r$  in  $KB$ , где  $\text{Standardize-Apart}(r) =$ 
     $(p_1 \wedge \dots \wedge p_n \heartsuit q)$  и  $\theta' \leftarrow \text{Unify}(q, q')$  является
    выполнимым
     $\text{new\_goals} \leftarrow [p_1, \dots, p_n | \text{Rest}(\text{goals})]$ 
     $\text{answers} \leftarrow \text{FOL-BC-Ask}(KB, \text{new\_goals}, \text{Compose}(\theta', \theta)) \cup \text{answers}$ 
return  $\text{answers}$ 

```

В этом алгоритме используется \heartsuit композиция подстановок. Здесь $\text{Compose}(\theta_1, \theta_2)$ — это подстановка, результат которой идентичен результату применения каждой подстановки по очереди, следующим образом:

$$\text{Subst}(\text{Compose}(\theta_1, \theta_2), p) = \text{Subst}(\theta_2, \text{Subst}(\theta_1, p))$$

В данном алгоритме текущие связывания переменных, которые хранятся в подстановке θ , komponуются со связываниями, возникающими в результате унификации цели с головой выражения, что приводит к получению нового множества текущих связываний для рекурсивного вызова.

Алгоритм обратного логического вывода в том виде, в каком он был приведен в этом разделе, безусловно, представляет собой алгоритм поиска в глубину. Это означает, что его потребности в пространстве линейно зависят от размера доказательства (если на данный момент пренебречь тем, какой объем пространства требуется для накопления решений). Это также означает, что обратный логический вывод (в отличие от прямого логического вывода) страдает от проблем, обусловленных наличием повторяющихся состояний и неполноты. Эти проблемы и некоторые потенциальные решения будут рассматриваться ниже, но вначале покажем, как обратный логический вывод используется в системах логического программирования.

Логическое программирование

Логическое программирование — это технология, позволяющая довольно близко приблизиться к воплощению декларативного идеала, описанного в главе 7, согласно которому системы должны конструироваться путем представления знаний на некотором формальном языке, а задачи решаться путем применения процессов логического вывода к этим знаниям. Такой идеал выражен в следующем уравнении Роберта Ковальского:

$$\text{Алгоритм} = \text{Логика} + \text{Управление}$$

Одним из языков логического программирования, намного превосходящим все прочие по своей распространенности, является \heartsuit **Prolog**. Количество его пользователей насчитывает сотни тысяч. Он используется в основном в качестве языка быстрой разработки прототипов, а также служит для решения задач символических манипуляций, таких как написание компиляторов [1536] и синтаксический анализ текстов на естественном языке [1208]. На языке Prolog было написано много экспертных систем для юридических, медицинских, финансовых и других проблемных областей.

Программы Prolog представляют собой множества определенных выражений, записанных в системе обозначений, немного отличающейся от используемой стандартной логики первого порядка. В языке Prolog прописные буквы применяются для обозначения переменных, а строчные — для обозначения констант. Выражения записываются с головой, предшествующей телу; символ $:-$ служит для обозначения

импликации, направленной влево, запятые разделяют литералы в теле, а точка обозначает конец высказывания, как показано ниже.

```
criminal(X) :- american(X), weapon(Y), sells(X,Y,Z), hostile(Z).
```

Язык Prolog включает “синтаксические упрощения” (syntactic sugar) для обозначения списков и арифметических выражений. Например, ниже приведена программа Prolog для предиката `append(X, Y, Z)`, которая выполняется успешно, если список `Z` представляет собой результат дополнения списка `Y` списком `X`.

```
append([], Y, Y).
append([A|X], Y, [A|Z]) :- append(X, Y, Z).
```

На естественном языке эти выражения можно прочесть так: во-первых, дополнение списка `Y` пустым списком приводит к получению того же списка `Y`, и, во-вторых, `[A|Z]` — это результат дополнения списка `Y` списком `[A|X]`, при условии, что `Z` — это результат дополнения списка `Y` списком `X`. Такое определение предиката `append` на первый взгляд кажется весьма подобным соответствующему определению на языке Lisp, но фактически является гораздо более мощным. Например, в систему можно ввести запрос `append(A, B, [1, 2])` — какие два списка можно дополнить один другим, чтобы получить `[1, 2]`? Система возвратит следующие решения:

```
A=[]      B=[1, 2]
A=[1]    B=[2]
A=[1, 2] B=[]
```

Выполнение программ Prolog осуществляется по принципу обратного логического вывода с поиском в глубину, при котором попытка применения выражений выполняется в том порядке, в каком они записаны в базу знаний. Но некоторые описанные ниже особенности языка Prolog выходят за рамки стандартного логического вывода.

- В нем предусмотрено множество встроенных функций для выполнения арифметических операций. Литералы, в которых используются соответствующие функциональные символы, “доказываются” путем выполнения кода, а не осуществления дальнейшего логического вывода. Например, цель “`X is 4+3`” достигается успешно после связывания переменной `X` со значением 7. С другой стороны, попытка достижения цели “`5 is X+Y`” оканчивается неудачей, поскольку эти встроенные функции не обеспечивают решения произвольных уравнений⁵.
- В языке предусмотрены встроенные предикаты, вызывающие при их выполнении побочные эффекты. К ним относятся предикаты ввода-вывода и предикаты `assert/retract` для модификации базы знаний. Такие предикаты не имеют аналогов в логике и могут порождать некоторые эффекты, вызывающие путаницу, например, если факты подтверждаются (и вводятся в базу знаний) некоторой ветвью дерева доказательства, которая в конечном итоге оканчивается неудачей.

⁵ Следует отметить, что если бы в некоторой программе Prolog были предусмотрены аксиомы Пеано, то такие цели могли быть решены с помощью логического вывода.

- В языке Prolog допускается определенная форма отрицания — **отрицание как недостижение цели**. Отрицаемая цель `not P` считается доказанной, если системе не удастся доказать `P`. Таким образом, следующее высказывание:
`alive(X) :- not dead(X).`
 можно прочесть так: “Любого следует считать живым, если нельзя доказать, что он мертв”.
- В языке Prolog предусмотрен оператор равенства, “=”, но он не обладает всей мощностью логического равенства. Цель с оператором равенства достигается успешно, если в ней два терма являются унифицируемыми, а в противном случае попытка ее достижения оканчивается неудачей. Таким образом, цель `X+Y=2+3` достигается успешно, после связывания переменной `X` со значением 2, а `Y` — со значением 3, а попытка достижения цели `morningstar=eveningstar` оканчивается неудачей. (В классической логике последнее равенство может быть или не быть истинным.) Не могут быть подтверждены (введены в базу знаний) какие-либо факты или правила, касающиеся равенства.
- Из алгоритма унификации Prolog исключена **проверка вхождения**. Это означает, что могут быть сделаны некоторые противоречивые логические выводы; такая проблема возникает редко, за исключением тех ситуаций, когда язык Prolog используется для доказательства математических теорем.

Решения, принятые при проектировании языка Prolog, представляют собой компромисс между стремлениями обеспечить декларативность и вычислительную эффективность (по крайней мере, эффективность в той ее трактовке, которая существовала в период разработки языка Prolog). Мы вернемся к этой теме после рассмотрения того, как реализована система Prolog.

Эффективная реализация логических программ

Подготовка программ Prolog к выполнению может осуществляться в двух режимах: интерпретация и компиляция. Интерпретация по существу представляет собой применение алгоритма `FOL-BC-Ask`, приведенного в листинге 9.3, к программе, представленной в виде базы знаний. Предыдущее предложение включает слова “по существу”, поскольку в интерпретаторах Prolog предусмотрены всевозможные усовершенствования, предназначенные для максимального повышения скорости работы. Здесь рассматриваются только два таких усовершенствования.

Во-первых, вместо формирования списка всех возможных ответов для каждой подцели перед переходом к следующей подцели интерпретаторы Prolog вырабатывают один ответ и “дают обещание” выработать остальные после того, как будет полностью исследован текущий ответ. Такое “обещание” оформляется как **точка выбора** (choice point). После того как в процессе поиска в глубину завершается исследование возможных решений, вытекающих из текущего ответа, и происходит возврат к точке выбора, в этой точке используемые структуры данных дополняются, чтобы в них можно было включить новый ответ для данной подцели и сформировать новую точку выбора. Такой подход позволяет экономить и время, и пространство, а также обеспечивает создание очень простого интерфейса для отладки, поскольку постоянно существует лишь единственный путь решения, подлежащий рассмотрению.

Во-вторых, в приведенной в данной главе простой реализации алгоритма FOL-BC-Ask много времени затрачивается на выработку и компоновку подстановок. В языке Prolog подстановки реализуются с использованием логических переменных, позволяющих сохранить в памяти их текущее связывание. В любой момент времени каждая переменная в программе является либо несвязанной, либо связанной с некоторым значением. Эти переменные и значения, вместе взятые, неявно определяют подстановку для текущей ветви доказательства. Любая попытка продления пути позволяет лишь добавить новые связывания переменных, поскольку стремление ввести другое связывание для уже связанной переменной приводит к неудачному завершению унификации. После того как попытка продления пути поиска оканчивается неудачей, система Prolog возвращается к предыдущей точке выбора и только после этого получает возможность отменить связывания некоторых переменных. Такая операция отмены выполняется благодаря тому, что данные обо всех переменных, для которых было выполнено связывание, отслеживаются в стеке, называемом **контрольным стекком** (trail). По мере того как в функции Unify-Var осуществляется связывание каждой новой переменной, эта переменная задвигается в контрольный стек. Если попытка достижения некоторой цели оканчивается неудачей и наступает время возвратиться к предыдущей точке пункта выбора, отменяется связывание каждой из этих переменных по мере их выталкивания из контрольного стека.

Но даже в самых эффективных интерпретаторах Prolog, в связи с издержками на поиск по индексу, унификацию и формирование стека рекурсивных вызовов, требуется выполнение нескольких тысяч машинных команд в расчете на каждый этап логического вывода. В действительности интерпретатор постоянно ведет себя так, как если бы он никогда до сих пор не видел данную программу; например, ему приходится каждый раз находить выражения, которые согласуются с целью. С другой стороны, откомпилированная программа Prolog представляет собой процедуру логического вывода для конкретного множества выражений, поэтому ей известно, какие выражения согласуются с целью. В процессе компиляции система Prolog по сути формирует миниатюрную программу автоматического доказательства теоремы для каждого отдельного предиката, устраняя тем самым основную часть издержек интерпретации. Эта система позволяет также применять **открытый код** для процедуры унификации каждого отдельного вызова, что позволяет избежать необходимости проведения явного анализа структуры терма (подробные сведения об унификации с открытым кодом приведены в [1557]).

Наборы команд современных компьютеров плохо согласуются с семантикой Prolog, поэтому большинство компиляторов Prolog компилирует программу в промежуточный язык, а не непосредственно в машинный язык. Наиболее широко применяемым промежуточным языком является язык WAM (Warren Abstract Machine — абстрактная машина Уоррена) получивший название в честь Дэвида Г.Д. Уоррена, одного из создателей первого компилятора Prolog. Язык WAM представляет собой абстрактное множество команд, которое подходит для преобразования в него программ Prolog и может интерпретироваться или транслироваться в машинный язык. Другие компиляторы транслируют программу Prolog в программу на языке высокого уровня, таком как Lisp или C, а затем используют компилятор этого языка для трансляции в машинный язык. Например, определение предиката Append может быть откомпилировано в код, показанный в листинге 9.4. Ниже приведено несколько замечаний, заслуживающих упоминания в этой связи.


Листинг 9.4. Псевдокод, представляющий собой результат компиляции предиката `Append`. Функция `New-Variable` возвращает новую переменную, отличную от всех других переменных, использовавшихся до сих пор. Процедура `Call(continuation)` продолжает выполнение с заданным продолжением `continuation`

```

procedure Append(ax, y, az, continuation)

    trail ← Global-Trail-Pointer()
    if ax = [] and Unify(y, az) then Call(continuation)
    Reset-Trail(trail)
    a ← New-Variable(); x ← New-Variable(); z ← New-Variable()
    if Unify(ax, [a | x]) and Unify(az, [a | z])
        then Append(x, y, z, continuation)

```

- Выражения предиката `Append` преобразуются в процедуру, а этапы логического вывода осуществляются путем вызова этой процедуры, поэтому не приходится выполнять поиск соответствующих выражений в базе знаний.
- Как было описано выше, текущие связывания переменных хранятся в контрольном стеке. На первом этапе выполнения этой процедуры текущее состояние контрольного стека сохраняется в памяти, поэтому оно может быть восстановлено с помощью функции `Reset-Trail`, если попытка выполнения первого выражения окончится неудачей. Это приводит к отмене всех связываний, сформированных при первом вызове процедуры `Unify`.
- Сложнейшей частью этой программы является использование  **продолжений** для реализации точек выбора. *Продолжение* может рассматриваться как структура данных, в которой упакованы процедура и список параметров, вместе взятые, определяющая, что следует делать дальше, после успешного достижения текущей цели. Дело в том, что после достижения цели не было бы достаточно просто вернуть управление из процедуры, подобной `Append`, поскольку успех может быть достигнут несколькими способами, и каждый из них должен быть исследован. Параметр `continuation`, определяющий продолжение, позволяет решить эту проблему, поскольку он может быть вызван после каждого успешного достижения цели. В приведенном здесь коде процедуры `Append`, если первый параметр является пустым, то предикат `Append` достиг успеха. Затем вызывается продолжение с помощью процедуры `Call` (притом что в контрольном стеке находятся все подходящие связывания) для того, чтобы определить, что делать дальше. Например, если процедура `Append` вызвана на верхнем уровне дерева доказательства, то структура данных `continuation` будет использоваться для вывода информации о связывании переменных.

До того как Уоррен выполнил свою работу по внедрению компиляции в системы логического вывода на языке `Prolog`, средства логического программирования работали слишком медленно для того, чтобы они действительно могли найти широкое применение. Компиляторы, разработанные Уорреном и другими специалистами, позволили достичь скоростей обработки кода `Prolog`, способных конкурировать с языком `C` в самых различных стандартных эталонных тестах [1536]. Безусловно, тот факт, что теперь появилась возможность написать планировщик или синтаксиче-

ский анализатор текста на естественном языке, состоящий из нескольких десятков строк на языке Prolog, говорит о том, что этот язык становится более подходящим (по сравнению с языком С) для создания прототипов большинства исследовательских проектов в области искусственного интеллекта небольших масштабов.

Значительное повышение скорости позволяет также обеспечить распараллеливание работы. Организация параллельного выполнения может осуществляться по двум направлениям. Первое направление, получившее название \sphericalangle **ИЛИИ-параллелизма**, исходит из возможности унификации цели со многими различными выражениями в базе знаний. Каждая из этих операций унификации приводит к появлению независимой ветви в пространстве поиска, которая может привести к потенциальному решению, и поиск решений по всем таким ветвям может осуществляться параллельно. Второе направление, называемое \sphericalangle **И-параллелизмом**, исходит из возможности параллельного решения каждого конъюнкта в теле некоторой импликации. Задача достижения И-параллелизма является более сложной, поскольку для поиска решений всей конъюнкции требуются совместимые связывания для всех переменных. Поэтому при обработке каждой конъюнктивной ветви необходимо обеспечивать обмен данными с другими ветвями для гарантированного достижения глобального решения.

Избыточный логический вывод и бесконечные циклы

Теперь рассмотрим, в чем состоит ахиллесова пята языка Prolog: несогласованность между организацией поиска в глубину и деревьями поиска, которые включают повторяющиеся состояния и бесконечные пути. Рассмотрим следующую логическую программу, позволяющую определить, существует ли путь между двумя точками в ориентированном графе:

```
path(X,Z) :- link(X,Z).
path(X,Z) :- path(X,Y), link(Y,Z).
```

На рис. 9.5, *a* приведен простой граф, состоящий из трех узлов, который описан с помощью фактов `link(a,b)` и `link(b,c)`. При использовании этой программы запрос `path(a,c)` вырабатывает дерево доказательства, показанное на рис. 9.6, *a*. С другой стороны, если эти два выражения расположены в таком порядке:

```
path(X,Z) :- path(X,Y), link(Y,Z).
path(X,Z) :- link(X,Z).
```

то система Prolog следует по бесконечному пути, как показано на рис. 9.6, *b*. Поэтому система Prolog является **неполной**, как и программа автоматического доказательства теоремы для определенных выражений (как показано в этом примере, даже применительно к программам, соответствующим формату языка Datalog), поскольку для некоторых баз знаний эта система оказывается неспособной доказать высказывания, которые из них следуют. Следует отметить, что алгоритм прямого логического вывода не подвержен этой проблеме: сразу после вывода фактов `path(a,b)`, `path(b,c)` и `path(a,c)` процесс прямого логического вывода останавливается.

Обратный логический вывод с поиском в глубину сталкивается также с проблемами, обусловленными излишними вычислениями. Например, при поиске пути от узла A_1 к узлу J_4 на рис. 9.5, *b* система Prolog выполняет 877 этапов логического вывода, большинство из которых связано с поиском всех возможных путей к узлам, не позволяющим достичь цели. Такая проблема аналогична проблеме повторяющихся состоя-

ний, которая описывалась в главе 3. Общее количество этапов логического вывода может определяться экспоненциальной зависимостью от количества базовых фактов, вырабатываемых в процессе вывода. А если бы вместо этого применялся прямой логический вывод, то можно было бы ограничиться выработкой, самое большее, n^2 фактов $\text{path}(X, Y)$, связывающих n узлов. При этом для решения задачи, приведенной на рис. 9.5, б, потребовалось бы только 62 этапа логического вывода.

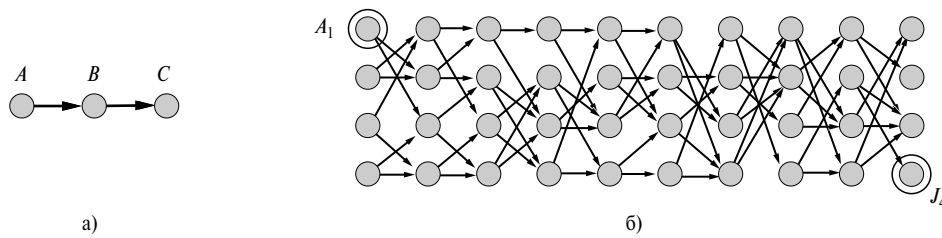


Рис. 9.5. Иллюстрация недостатков языка Prolog: поиск пути от A до C может привести систему Prolog к созданию бесконечного цикла (а); граф, в котором каждый узел связан с двумя случайно выбираемыми преемниками на следующем уровне; для поиска пути от A_1 до J_4 требуется 877 этапов логического вывода (б)

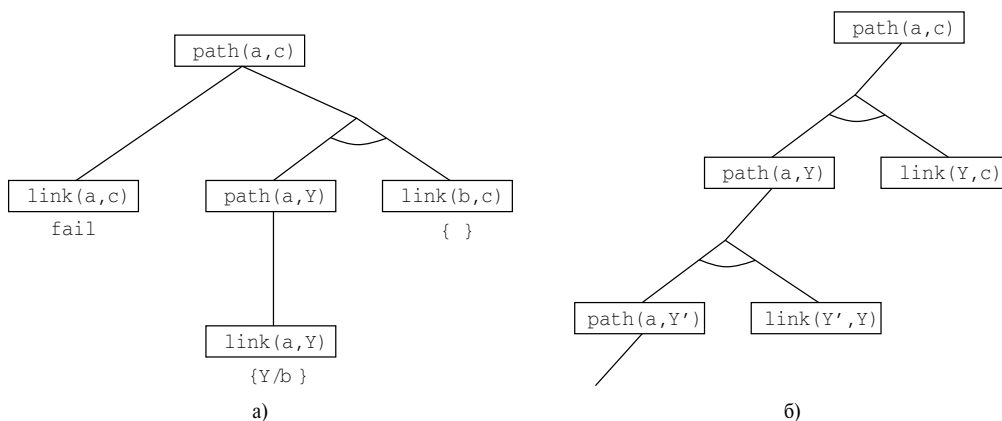


Рис. 9.6. Иллюстрация того, что работа программы Prolog зависит от расположения взаимосвязанных выражений: успешное доказательство того, что путь от A до C существует (а); бесконечное дерево доказательства, формируемое из-за того, что выражения находятся в “неправильном” порядке (б)

Применение прямого логического вывода при решении задач поиска в графе представляет собой пример **динамического программирования**, в котором решения подзадач формируются инкрементно из решений меньших подзадач и кэшируются для предотвращения повторного вычисления. Аналогичный эффект в системе обратного логического вывода может быть достигнут с помощью **запоминания** (memoization), т.е. кэширования решений, найденных для подцели, по мере их обнаружения, а затем повторного применения этих решений после очередного появления той же подцели, вместо повторения предыдущих вычислений. Этот подход применяется в системах **табулированного логического программирования** (tabled logic programming), в которых для реализации метода запоминания используются

эффективные механизмы сохранения и выборки. В табулированном логическом программировании объединяется целенаправленность обратного логического вывода с эффективностью динамического программирования, присущей прямому логическому выводу. Кроме того, эти системы являются полными применительно к программам в формате Datalog, а это означает, что программисту приходится меньше беспокоиться о бесконечных циклах.

Логическое программирование в ограничениях

В приведенном выше описании прямого логического вывода (раздел 9.3) было показано, как можно представить задачи удовлетворения ограничений (Constraint Satisfaction Problem — CSP) в виде определенных выражений. Стандартный язык Prolog позволяет решать подобные задачи точно таким же способом, как и алгоритм поиска с возвратами, приведенный в листинге 5.1.

Поскольку поиск с возвратами предусматривает полный перебор областей определения переменных, он может применяться только для решения задач CSP с **конечными областями определения**. В терминах Prolog это можно перефразировать таким образом, что для любой цели с несвязанными переменными должно существовать конечное количество решений. (Например, цель `diff(q, sa)`, которая определяет, что штаты Квинсленд и Южная Австралия должны быть окрашены в разные цвета, имеет шесть решений, если допускается применение трех цветов.) Задачи CSP с бесконечными областями определения (например, с целочисленными или вещественными переменными) требуют применения совсем других алгоритмов, таких как распространение пределов или линейное программирование.

Приведенное ниже выражение выполняется успешно, если подставленные в него три числа удовлетворяют неравенству треугольника.

```
triangle(X,Y,Z) :-
  X>=0, Y>=0, Z>=0, X+Y>=Z, Y+Z>=X, X+Z>=Y.
```

Если системе Prolog передан запрос `triangle(3,4,5)`, он будет выполнен безукоризненно. С другой стороны, после передачи запроса `triangle(3,4,Z)` решение не будет найдено, поскольку подцель `Z>=0` не может быть обработана системой Prolog. Возникающая при этом сложность состоит в том, что переменные в системе Prolog должны находиться только в одном из двух состояний: несвязанные или связанные с конкретным термом.

Связывание переменной с конкретным термом может рассматриваться как крайняя форма ограничения, а именно как ограничение равенства. **Логическое программирование в ограничениях** (Constraint Logic Programming — CLP) позволяет ограничивать, а не связывать переменные. Решением для программы в логике ограничений является наиболее конкретное множество ограничений, налагаемых на переменные запроса, которое может быть определено с помощью базы знаний. Например, решением запроса `triangle(3,4,Z)` является ограничение `7>=Z>=1`. Программы в стандартной логике представляют собой частный случай программ CLP, в которых ограничения решения должны быть не ограничениями сравнения, а ограничениями равенства, т.е. связываниями.

Системы CLP включают различные алгоритмы решения задач с ограничениями для таких вариантов ограничений, которые разрешены к использованию в языке.

Например, система, позволяющая использовать линейные неравенства с переменными, имеющими вещественные значения, может включать алгоритм линейного программирования для решения этих ограничений. Кроме того, в системах CLP принят гораздо более гибкий подход к решению запросов стандартного логического программирования. Например, вместо использования поиска в глубину, слева направо, с возвратами, в них может применяться любой из более эффективных алгоритмов, описанных в главе 5, включая эвристическое упорядочение конъюнктов, обратный переход, определение условий формирования множества отсечения и т.д. Поэтому в системах CLP сочетаются элементы алгоритмов удовлетворения ограничений, логического программирования и дедуктивных баз данных.

Кроме того, системы CLP позволяют воспользоваться преимуществами различных методов оптимизации поиска в задачах CSP, описанных в главе 5, таких как упорядочение переменных и значений, предварительная проверка и интеллектуальный возврат. В частности, разработаны проекты нескольких систем, позволяющих программисту получить больший контроль над порядком поиска для логического вывода. Например, язык MRS [539], [1321] позволяет программисту-пользователю записывать λ **метаправила** для определения того, какие конъюнкты должны быть опробованы в первую очередь. Например, пользователь может сформулировать правило с указанием, что в первую очередь следует пытаться достичь цели с наименьшим количеством переменных, или оформить характерные для проблемной области правила, касающиеся конкретных предикатов.

9.5. РЕЗОЛЮЦИЯ

Последнее из трех рассматриваемых в данной главе семейств логических систем основано на **резолуции**. Как было показано в главе 7, пропозициональная резолюция — это полная процедура логического вывода для пропозициональной логики на основе опровержения. В этом разделе будет показано, как распространить резолюцию на логику первого порядка.

Проблема существования полных процедур доказательства всегда является предметом непосредственного внимания математиков. Если бы удалось найти полную процедуру доказательства для математических утверждений, это повлекло бы за собой два последствия: во-первых, вывод всех заключений мог бы осуществляться механически; во-вторых, всю математику можно было бы построить как логическое следствие некоторого множества фундаментальных аксиом. Поэтому вопрос о полноте доказательства стал в XX веке предметом наиболее важных математических работ. В 1930 году немецкий математик Курт Гёдель доказал первую λ **теорему о полноте** для логики первого порядка, согласно которой любое высказывание, являющееся следствием заданных аксиом, имеет конечное доказательство. (Но никакая действительно применимая на практике процедура доказательства не была найдена до тех пор, пока Дж.Э. Робинсон не опубликовал в 1965 году алгоритм резолюции.) В 1931 году Гёдель доказал еще более знаменитую λ **теорему о неполноте**. В этой теореме утверждается, что любая логическая система, которая включает принцип индукции (а без этого принципа удастся построить лишь очень малую часть дискретной математики), обязательно является неполной. Поэтому существуют такие высказывания, которые следуют из заданных аксиом, но в рамках данной логиче-

ской системы для них невозможно найти конечное доказательство. Иголка действительно может быть в метафорическом стоге сена, но ни одна процедура не позволяет гарантировать, что она будет найдена.

Несмотря на то, что теорема Гёделя о неполноте устанавливает определенные пределы, программы автоматического доказательства теорем на основе резолюции широко применялись и применяются для обоснования математических теорем, включая несколько таких теорем, для которых до сих пор не было известно доказательство. Кроме того, программы автоматического доказательства теорем успешно использовались для проверки проектов аппаратных средств, формирования логически правильных программ, а также во многих других приложениях.

Конъюнктивная нормальная форма для логики первого порядка

Как и в случае пропозициональной логики, для резолюции в логике первого порядка требуется, чтобы высказывания находились в **конъюнктивной нормальной форме** (Conjunctive Normal Form — CNF), т.е. представляли собой конъюнкцию выражений, в которой каждое выражение представляет собой дизъюнкцию литералов⁶. Литералы могут содержать переменные, на которые, согласно принятому предположению, распространяется квантор всеобщности. Например, высказывание

$$\forall x \text{ American}(x) \wedge \text{Weapon}(y) \wedge \text{Sells}(x, y, z) \wedge \text{Hostile}(z) \heartsuit \text{Criminal}(x)$$

принимает в форме CNF следующий вид:

$$\neg \text{American}(x) \vee \neg \text{Weapon}(y) \vee \neg \text{Sells}(x, y, z) \vee \neg \text{Hostile}(z) \vee \text{Criminal}(x)$$

☞ Каждое высказывание в логике первого порядка может быть преобразовано в эквивалентное с точки зрения логического вывода высказывание CNF. В частности, высказывание CNF является невыполнимым тогда и только тогда, когда невыполнимо первоначальное высказывание, поэтому мы получаем основу для формирования доказательств от противного с помощью высказываний CNF.

Процедура преобразования любого высказывания в форму CNF весьма подобна процедуре, применяемой в пропозициональной логике, которая показана на с. 308. Принципиальное различие связано с необходимостью устранения кванторов существования. Проиллюстрируем эту процедуру на примере преобразования в форму CNF высказывания “Каждого, кто любит всех животных, кто-то любит”, или

$$\forall x [\forall y \text{ Animal}(y) \heartsuit \text{Loves}(x, y)] \heartsuit [\exists y \text{ Loves}(y, x)]$$

Ниже приведены этапы этого преобразования.

- **Устранение импликаций:**

$$\forall x [\neg \forall y \neg \text{Animal}(y) \vee \text{Loves}(x, y)] \vee [\exists y \text{ Loves}(y, x)]$$

- **Перемещение связок \neg внутрь выражений.** Кроме обычных правил для отрицаемых связок, нам нужны правила для отрицаемых кванторов. Поэтому получаем следующие правила:

⁶ Как показано в упр. 7.12, любое выражение может быть представлено как импликация с конъюнкцией атомов слева и дизъюнкцией атомов справа. Эта форма, иногда называемая **формой Ковальского**, при использовании записи с символом импликации, ориентированным справа налево [Ошибка! Источник ссылки не найден.], часто бывает намного более удобной для чтения по сравнению с другими формами.

$\neg\forall x p$ принимает вид $\exists x \neg p$

$\neg\exists x p$ принимает вид $\forall x \neg p$

Рассматриваемое высказывание проходит через такие преобразования:

$\forall x [\exists y \neg(\neg Animal(y) \vee Loves(x, y))] \vee [\exists y Loves(y, x)]$

$\forall x [\exists y \neg\neg Animal(y) \wedge \neg Loves(x, y)] \vee [\exists y Loves(y, x)]$

$\forall x [\exists y Animal(y) \wedge \neg Loves(x, y)] \vee [\exists y Loves(y, x)]$

Обратите внимание на то, что квантор всеобщности ($\forall y$) в предпосылке импликации стал квантором существования. Теперь это высказывание приобрело такое прочтение: “Либо существует какое-то животное, которого x не любит, либо (если это утверждение не является истинным) кто-то любит x ”. Очевидно, что смысл первоначального высказывания был сохранен.

- **Стандартизация переменных.** В высказываниях наподобие $(\forall x P(x)) \vee (\exists x Q(x))$, в которых дважды используется одно и то же имя переменной, изменим имя одной из переменных. Это позволит в дальнейшем избежать путаницы после того, как будут удалены кванторы. Поэтому получим следующее:

$\forall x [\exists y Animal(y) \wedge \neg Loves(x, y)] \vee [\exists z Loves(z, x)]$

- **Сколемизация.** \sphericalangle **Сколемизация** — это процесс устранения кванторов существования путем их удаления. В данном простом случае этот процесс подобен применению правила конкретизации с помощью квантора существования, приведенного в разделе 9.1: преобразовать $\exists x P(x)$ в $P(A)$, где A — новая константа. Однако, если это правило будет непосредственно применено к высказыванию, рассматриваемому в качестве примера, то будет получено следующее высказывание:

$\forall x [Animal(A) \wedge \neg Loves(x, A)] \vee Loves(B, x)$

которое имеет полностью неправильный смысл: в нем утверждается, что каждый либо не способен любить какое-то конкретное животное A , либо его любит некоторая конкретная сущность B . В действительности первоначальное высказывание позволяет каждому человеку не быть способным любить какое-то другое животное или быть любимым другим человеком. Поэтому желательно, чтобы сущности, определяемые в процессе сколемизации, зависели от x :

$\forall x [Animal(F(x)) \wedge \neg Loves(x, F(x))] \vee Loves(G(x), x)$

где F и G — \sphericalangle **сколемовские функции**. Общее правило состоит в том, что все параметры сколемовской функции должны быть переменными, на которые распространяются кванторы всеобщности, в область действия которых попадает соответствующий квантор существования. Как и при использовании конкретизации с помощью квантора существования, сколемизированное высказывание является выполнимым тогда и только тогда, когда выполнено первоначальное высказывание.

- **Удаление кванторов всеобщности.** В данный момент на все оставшиеся переменные должны распространяться кванторы всеобщности. Кроме того, данное высказывание эквивалентно тому, в котором все кванторы всеобщности перенесены влево. Поэтому кванторы всеобщности могут быть удалены следующим образом:

$$[Animal(F(x)) \wedge \neg Loves(x, F(x))] \vee Loves(G(x), x)$$

- **Распределение связки \vee по \wedge :**

$$[Animal(F(x)) \vee Loves(G(x), x)] \wedge [\neg Loves(x, F(x)) \vee Loves(G(x), x)]$$

На этом этапе может также потребоваться выполнить раскрытие скобок во вложенных конъюнкциях и дизъюнкциях.

Теперь рассматриваемое высказывание находится в форме CNF и состоит из двух выражений. Оно полностью недоступно для восприятия. (Помочь его понять может такое пояснение, что сколемовская функция $F(x)$ указывает на животное, которое потенциально может быть нелюбимым лицом x , а $G(x)$ указывает на кого-то, кто может любить лицо x .) К счастью, людям редко приходится изучать высказывания в форме CNF, поскольку показанный выше процесс преобразования может быть легко автоматизирован.

Правило логического вывода с помощью резолюции

Правило резолюции для выражений в логике первого порядка представляет собой поднятую версию правила резолюции для пропозициональной логики, приведенного на с. 307. Два выражения, которые, согласно принятому предположению, должны быть стандартизированными таким образом, чтобы в них не было общих переменных, могут быть подвергнуты операции резолюции, если они содержат взаимно дополнительные литералы. Пропозициональные литералы являются взаимно дополнительными, если один из них представляет собой отрицание другого, а литералы в логике первого порядка являются взаимно дополнительными, если один из них унифицируется с отрицанием другого. Поэтому имеет место следующее правило:

$$\frac{\ell_1 \vee \dots \vee \ell_k, m_1 \vee \dots \vee m_n}{\text{Subst}(\theta, \ell_1 \vee \dots \vee \ell_{i-1} \vee \ell_{i+1} \vee \dots \vee \ell_k \vee m_1 \vee \dots \vee m_{j-1} \vee m_{j+1} \vee \dots \vee m_n)}$$

где $\text{Unify}(\ell_i, \neg m_j) = \theta$. Например, можно применить операцию резолюции к следующим двум выражениям:

$$[Animal(F(x)) \vee Loves(G(x), x)] \text{ и } [\neg Loves(u, v) \vee \neg Kills(u, v)]$$

путем устранения взаимно дополнительных литералов $Loves(G(x), x)$ и $\neg Loves(u, v)$ с помощью унификатора $\theta = \{u/G(x), v/x\}$ для получения следующего выражения, называемого **резольвентой**:

$$[Animal(F(x)) \vee \neg Kills(G(x), x)]$$

Только что приведенное правило называется правилом \bowtie **бинарной резолюции**, поскольку в нем происходит удаление с помощью резолюции двух и только двух взаимно дополнительных литералов. Но правило бинарной резолюции, отдельно взятое, не позволяет получить полную процедуру логического вывода. С другой стороны, правило полной резолюции позволяет удалять в каждом выражении подмножества литералов, которые являются унифицируемыми. Альтернативный подход состоит в том, чтобы распространить **операцию факторизации** (удаления избыточных литералов) на логику первого порядка. В пропозициональной факторизации два литерала сводятся к одному, если они являются идентичными, а в факторизации первого порядка два литерала сводятся к одному, если они являются унифицируемыми.

Унификатор должен быть применен ко всему выражению. Сочетание бинарной резолюции и факторизации позволяет создать полную процедуру логического вывода.

Примеры доказательств

При использовании резолюции доказательство того, что $KB \models \alpha$ (из базы знаний следует высказывание α) осуществляется путем доказательства невыполнимости выражения $KB \wedge \neg\alpha$, т.е. путем получения пустого выражения. Алгоритмический подход, применяемый в логике первого порядка, идентичен подходу в пропозициональной логике, который показан в листинге 7.5, поэтому мы не будем здесь его повторять. Вместо этого приведем два примера доказательства. Первым из них является пример доказательства преступления, описанного в разделе 9.3. Соответствующие высказывания, преобразованные в форму CNF, выглядят следующим образом:

$$\begin{aligned} &\neg American(x) \vee \neg Weapon(y) \vee \neg Sells(x, y, z) \vee \neg Hostile(z) \vee \\ &Criminal(x) \\ &\neg Missile(x) \vee \neg Owns(Nono, x) \vee Sells(West, x, Nono) \\ &\neg Enemy(x, America) \vee Hostile(x) \\ &\neg Missile(x) \vee Weapon(x) \\ &Owns(Nono, M_1) \\ &Missile(M_1) \\ &American(West) \\ &Enemy(Nono, America) \end{aligned}$$

В число этих высказываний должна быть включена также отрицаемая цель $\neg Criminal(West)$. Процедура доказательства по методу резолюции показана на рис. 9.7. Обратите внимание на его структуру: единственный “хребет” начинается с целевого выражения, и операция резолюции применяется к выражениям из базы знаний до тех пор, пока не образуется пустое выражение. В этом состоит характерная особенность применения метода резолюции к базам знаний, представленным в виде хорновских выражений. В действительности выражения, расположенные вдоль главного хребта, точно соответствуют последовательным значениям целевых переменных в алгоритме обратного логического вывода, приведенном в листинге 9.3. Это связано с тем, что для резолюции всегда выбирается выражение, положительный литерал которого унифицируется с самым левым литералом “текущего” выражения в хребте; именно это происходит при обратном логическом выводе. Таким образом, обратный логический вывод в действительности представляет собой просто частный случай резолюции, в котором применяется конкретная стратегия управления для определения того, какая операция резолюции должна быть выполнена в следующую очередь.

В рассматриваемом здесь втором примере используется сколемизация и применяются выражения, которые не являются определенными. Это приводит к созданию немного более сложной структуры доказательства. На естественном языке эта задача формулируется, как описано ниже.

Каждого, кто любит всех животных, кто-то любит.
 Любого, кто убивает животных, никто не любит.
 Джек любит всех животных.
 Кота по имени Тунец убил либо Джек, либо Любопытство.
 Действительно ли этого кота убило Любопытство?

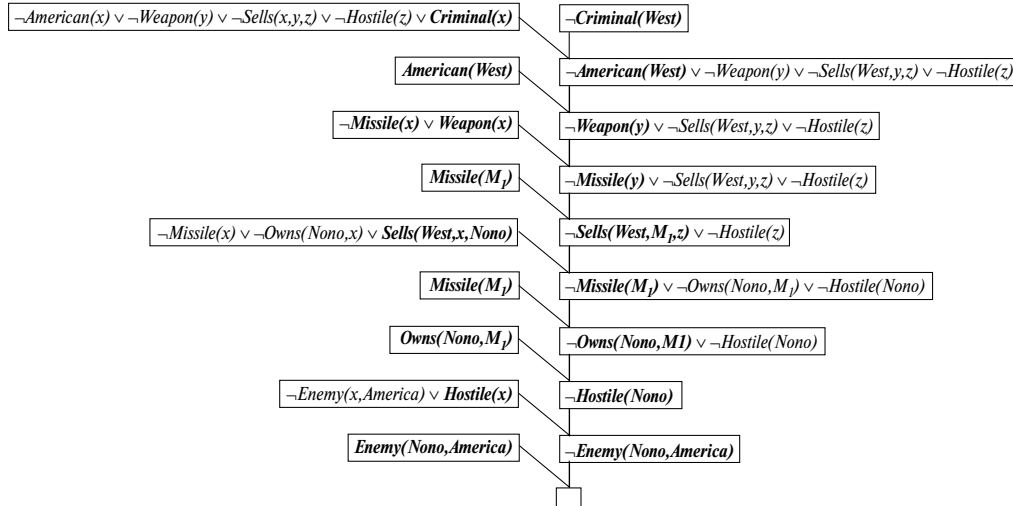


Рис. 9.7. Процедура доказательства с помощью резолюции того, что полковник Уэст совершил преступление

Вначале представим в логике первого порядка первоначальные высказывания, некоторые фоновые знания и отрицаемую цель G :

- A. $\forall x [\forall y Animal(y) \heartsuit Loves(x,y)] \heartsuit [\exists y Loves(y,x)]$
- B. $\forall x [\exists y Animal(y) \wedge Kills(x,y)] \heartsuit [\forall z \neg Loves(z,x)]$
- C. $\forall x Animal(x) \heartsuit Loves(Jack,x)$
- D. $Kills(Jack,Tuna) \vee Kills(Curiosity,Tuna)$
- E. $Cat(Tuna)$
- F. $\forall x Cat(x) \heartsuit Animal(x)$
- $\neg G$. $\neg Kills(Curiosity,Tuna)$

Затем применим процедуру преобразования, чтобы преобразовать каждое высказывание в форму CNF:

- A1. $Animal(F(x)) \vee Loves(G(x),x)$
- A2. $\neg Loves(x,F(x)) \vee Loves(G(x),x)$
- B. $\neg Animal(y) \vee \neg Kills(x,y) \vee \neg Loves(z,x)$
- C. $\neg Animal(x) \vee Loves(Jack,x)$
- D. $Kills(Jack,Tuna) \vee Kills(Curiosity,Tuna)$
- E. $Cat(Tuna)$
- F. $\neg Cat(x) \vee Animal(x)$
- $\neg G$. $\neg Kills(Curiosity,Tuna)$

Доказательство с помощью метода резолюции того, что кот убило Любопытство, приведено на рис. 9.8. На естественном языке это доказательство может быть перефразировано, как показано ниже.

Предположим, что кот Тунца убило не Любопытство. Мы знаем, что это сделал либо Джек, либо Любопытство; в таком случае это должен был сделать Джек. Итак, Тунец — кот, а коты — животные, поэтому Тунец — животное. Любого, кто убивает животное, никто не любит, поэтому мы делаем вывод, что никто не любит Джека. С другой стороны, Джек любит всех животных, поэтому кто-то его любит; таким образом, возникает противоречие. Это означает, что кот убило Любопытство.

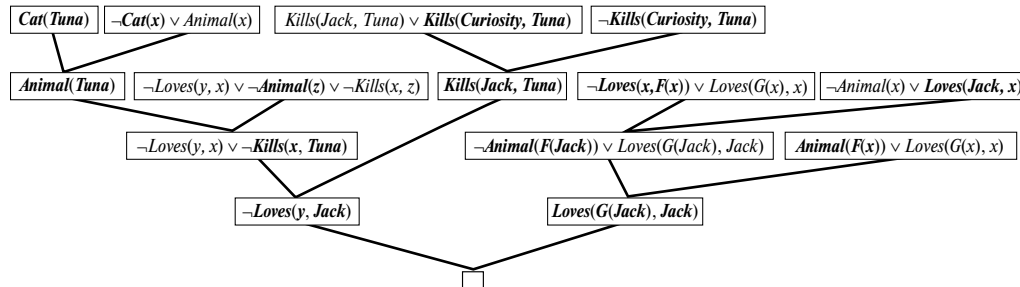


Рис. 9.8. Процедура доказательства с помощью резолюции того, что коту убило Любопытство. Обратите внимание на то, что при выводе выражения $\text{Loves}(G(\text{Jack}), \text{Jack})$ использовалась факторизация

Такое доказательство действительно отвечает на вопрос “Действительно ли этого кота убило Любопытство?”, но часто требуется найти ответ на более общие вопросы, такие как “Кто убил кота?” Резолюция позволяет это сделать, но для получения ответа требует немного больше работы. Данная цель может быть представлена в виде выражения $\exists w \text{Kills}(w, \text{Tuna})$, которое после его отрицания принимает в форме CNF вид $\neg \text{Kills}(w, \text{Tuna})$. Повторяя доказательство, показанное на рис. 9.8, с новой отрицаемой целью, мы получим аналогичное дерево доказательства, но с подстановкой $\{w/\text{Curiosity}\}$ в одном из этапов. Поэтому в данном случае для поиска того, кто убил кота, достаточно проследить за связываниями, которые применяются к переменным запроса в этом доказательстве.

К сожалению, в методе резолюции могут вырабатываться **неконструктивные доказательства** для существующих целей. Например, в выражении $\neg \text{Kills}(w, \text{Tuna})$ после применения операции резолюции удаляется взаимно дополнительный литерал, входящий в состав выражения $\text{Kills}(\text{Jack}, \text{Tuna}) \vee \text{Kills}(\text{Curiosity}, \text{Tuna})$, с получением выражения $\text{Kills}(\text{Jack}, \text{Tuna})$, к которому снова применяется операция резолюции с использованием выражения $\neg \text{Kills}(w, \text{Tuna})$, что приводит к получению пустого выражения. Обратите внимание на то, что в этом доказательстве переменная w имеет два различных связывания, а правило резолюции сообщает нам, что кто-то действительно убил кота Тунца — либо Джек, либо Любопытство. Но в этом для нас нет ничего нового! Одно из решений данной проблемы состоит в том, чтобы регламентировать допустимые этапы резолюции так, чтобы переменные запроса могли быть связанными только один раз в каждом конкретном доказательстве; в таком случае можно будет предусмотреть применение перехода с возвратом по всем возможным связываниям. Еще одно решение состоит в добавлении специального **литерала ответа** к отрицаемой цели, которая принимает вид $\neg \text{Kills}(w, \text{Tuna}) \vee \text{Answer}(w)$. Теперь процесс резолюции вырабатывает ответ каждый раз, когда формируется выражение, содержащее только единственный литерал ответа. Для доказательства, приведенного на рис. 9.8, таковым является выражение $\text{Answer}(\text{Curiosity})$. Неконструктивное доказательство привело бы к выработке выражения $\text{Answer}(\text{Curiosity}) \vee \text{Answer}(\text{Jack})$, которое не может рассматриваться как ответ и отбрасывается.

Полнота резолюции

В настоящем разделе приведено доказательство полноты резолюции. Это доказательство может пропустить без ущерба для дальнейшего понимания текста любой читатель, который готов принять его на веру.

Мы покажем, что резолюция обеспечивает \approx **полноту опровержения** (refutation completeness), а это означает, что если множество высказываний является невыполнимым, то резолюция всегда позволяет прийти к противоречию. Резолюцию нельзя использовать для выработки всех логических следствий из множества высказываний, но она может применяться для подтверждения того, что данное конкретное высказывание следует из множества высказываний. Поэтому резолюция может служить для поиска всех ответов на данный конкретный вопрос с помощью метода отрицаемой цели, который был описан выше в настоящей главе.

Примем как истинное такое утверждение, что любое высказывание в логике первого порядка (без использования равенства) может быть перезаписано в виде множества выражений в форме CNF. Это можно доказать по индукции на основе анализа формы высказывания, применяя в качестве базового случая атомарные высказывания [336]. Поэтому наша цель состоит в том, чтобы доказать следующее: \Leftarrow *если S — невыполнимое множество выражений, то применение к S конечного количества этапов резолюции приведет к противоречию.*

Схема нашего доказательства повторяет первоначальное доказательство, приведенное Робинсоном, с некоторыми упрощениями, которые были внесены Генезеретом и Нильссоном [537]. Основная структура этого доказательства показана на рис. 9.9; оно осуществляется, как описано ниже.

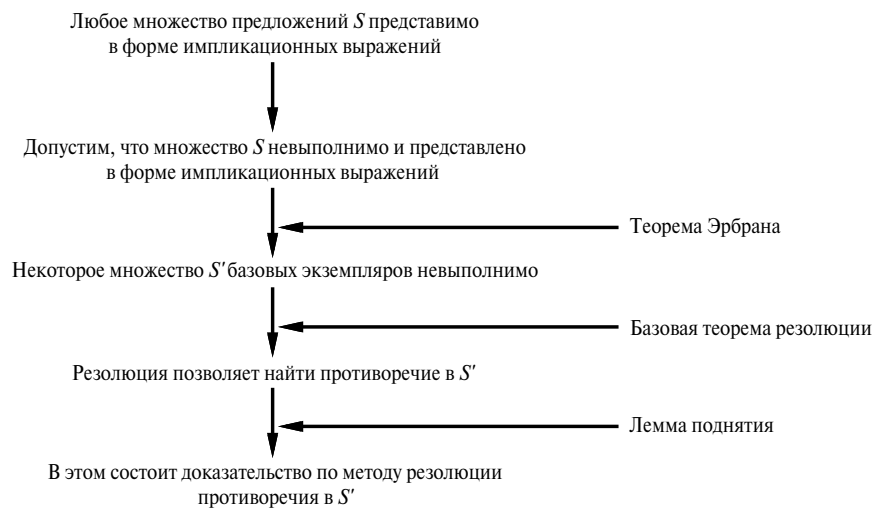


Рис. 9.9. Структура доказательства полноты резолюции

1. Вначале отметим, что если множество выражений S невыполнимо, то существует такое конкретное множество базовых экземпляров выражений S , что это множество также невыполнимо (теорема Эрбрана).

2. Затем прибегнем к **базовой теореме резолюции** (ground resolution theorem), приведенной в главе 7, в которой утверждается, что пропозициональная резолюция является полной для базовых высказываний.
3. После этого воспользуемся **леммой поднятия**, чтобы показать, что для любого доказательства по методу пропозициональной резолюции, в котором применяется множество базовых высказываний, существует соответствующее доказательство по методу резолюции первого порядка с использованием высказываний в логике первого порядка, из которых были получены базовые высказывания.

ТЕОРЕМА ГЁДЕЛЯ О НЕПОЛНОТЕ

Немного дополнив язык логики первого порядка для обеспечения возможности применять **схему математической индукции** в арифметике, Гёдель сумел показать в своей **теореме о неполноте**, что существуют истинные арифметические высказывания, которые не могут быть доказаны.

Полное доказательство этой теоремы о неполноте немного выходит за рамки настоящей книги, поскольку в своем непосредственном виде оно занимает не меньше 30 страниц, но мы здесь приведем его набросок. Начнем с логической теории чисел. В этой теории существует единственная константа, 0, и единственная функция, S (функция определения преемника). В намеченной модели интерпретации этой теории $S(0)$ обозначает 1, $S(S(0))$ обозначает 2 и т.д., поэтому в рассматриваемом языке имеются имена для всех натуральных чисел. Кроме того, словарь языка включает функциональные символы $+$, \times и $Exp t$ (возведение в степень), а также обычное множество логических связок и кванторов. Прежде всего, следует отметить, что множество высказываний, которые могут быть записаны на этом языке, может быть пронумеровано. (Для этого достаточно представить себе, что определен алфавитный порядок символов, а затем в алфавитном порядке расположено каждое из множеств высказываний с длиной 1, 2 и т.д.) Затем можно обозначить каждое высказывание α уникальным натуральным числом $\# \alpha$ (которое называется **гёделевским номером**). Это — самый важный момент доказательства; в нем утверждается, что теория чисел включает отдельное имя для каждого из ее собственных высказываний. Аналогичным образом, с помощью гёделевского номера $G(P)$ можно пронумеровать каждое возможное доказательство P , поскольку любое доказательство — это просто конечная последовательность высказываний.

Теперь предположим, что имеется рекурсивно перечислимое множество A высказываний, которые представляют собой истинные утверждения о натуральных числах. Напомним, что высказывания из множества A можно именовать с помощью заданного множества целых чисел, поэтому можно представить себе, что на нашем языке записывается высказывание $\alpha(j, A)$ такого рода:

- $\forall i \ i$ — не гёделевский номер доказательства высказывания, гёделевским номером которого является j , где в доказательстве используются только предпосылки из множества A .

Затем допустим, что σ представляет собой высказывание $\alpha(\# \sigma, A)$, т.е. высказывание, в котором утверждается его собственная недоказуемость из A . (Утверждение о том, что такое высказывание всегда существует, является истинным, но его нельзя назвать полностью очевидным.)

Теперь применим следующий остроумный довод: предположим, что высказывание σ доказуемо из A ; в таком случае высказывание σ ложно (поскольку в высказывании σ утверждается, что оно не может быть доказано). Но это означает, что имеется некоторое ложное высказывание, которое доказуемо из A , поэтому A не может состоять только из истинных высказываний, а это противоречит нашей предпосылке. Поэтому высказывание σ не доказуемо из A . Но именно это и утверждает о самом себе высказывание σ , а это означает, что σ — истинное высказывание.

Итак, мы доказали (и сэкономили 29 с половиной страниц), что для любого множества истинных высказываний теории чисел и, в частности, для любого множества базовых аксиом существуют другие истинные высказывания, которые не могут быть доказаны из этих аксиом. Из этого, кроме всего прочего, следует, что мы никогда не сможем доказать все теоремы математики в пределах любой конкретной системы аксиом. Очевидно, что это открытие имело для математики очень важное значение. Значимость этого открытия для искусственного интеллекта была предметом широких обсуждений, начиная с размышлений самого Гёделя. Мы вступим в эти дебаты в главе 26.

Для того чтобы выполнить первый этап доказательства, нам потребуются три новых понятия, описанных ниже.

- **Универсум Эрбрана.** Если S — множество выражений, то H_S , универсум Эрбрана для множества S , представляет собой множество всех базовых термов, которые могут быть сформированы из следующего:
 - а) функциональные символы из множества S , если они имеются;
 - б) константные символы из множества S , если они имеются; если они отсутствуют, то константный символ A .

Например, если множество S содержит только выражение $\neg P(x, F(x, A)) \vee \neg Q(x, A) \vee R(x, B)$, то H_S представляет собой следующее бесконечное множество базовых термов:

$$\{A, B, F(A, A), F(A, B), F(B, A), F(B, B), F(A, F(A, A)), \dots\}$$

- **Насыщение.** Если S — множество выражений, а P — множество базовых термов, то $P(S)$, насыщение S по отношению к P , представляет собой множество всех базовых выражений, полученное путем применения всех возможных совместимых подстановок базовых термов из P вместо переменных в S .
- **База Эрбрана.** Насыщение множества выражений S по отношению к его универсуму Эрбрана называется базой Эрбрана множества S и записывается как $H_S(S)$. Например, если S содержит только приведенное выше выражение, то $H_S(S)$ представляет собой следующее бесконечное множество выражений:

$$\begin{aligned} & \{ \neg P(A, F(A, A)) \vee \neg Q(A, A) \vee R(A, B), \\ & \neg P(B, F(B, A)) \vee \neg Q(B, A) \vee R(B, B), \\ & \neg P(F(A, A), F(F(A, A), A)) \vee \neg Q(F(A, A), A) \vee R(F(A, A), B), \\ & \neg P(F(A, B), F(F(A, B), A)) \vee \neg Q(F(A, B), A) \vee R(F(A, B), B), \dots \} \end{aligned}$$

Эти определения позволяют сформулировать одну из форм \aleph **теоремы Эрбрана** [650]:

Если множество выражений S является невыполнимым, то существует конечное подмножество $H_S(S)$, которое также является невыполнимым.

Допустим, что S' — конечное подмножество базовых высказываний. Теперь можно прибегнуть к базовой теореме резолюции (с. 311), чтобы показать, что **резолюционное замыкание** $RC(S')$ содержит пустое выражение. Это означает, что доведение до конца процесса пропозициональной резолюции применительно к S' приводит к противоречию.

Теперь, после определения того, что всегда существует доказательство по методу резолюции, в котором применяется некоторое конечное подмножество базы Эрбрана множества S , на следующем этапе необходимо показать, что существует доказательство по методу резолюции, в котором используются выражения из самого множества S , которые не обязательно являются базовыми выражениями. Начнем с рассмотрения одного приложения правила резолюции. Из базовой леммы Робинсона следует приведенный ниже факт.

Допустим, что C_1 и C_2 — два выражения без общих переменных, а C_1' и C_2' — базовые экземпляры C_1 и C_2 . Если C' — резольвента C_1' и C_2' , то существует выражение C , такое, что, во-первых, C — резольвента C_1 и C_2 , и, во-вторых, C' — базовый экземпляр C .

Это утверждение называется \aleph **леммой поднятия** (lifting lemma), поскольку оно позволяет поднять любой этап доказательства от базовых выражений к общим выражениям в логике первого порядка. Для того чтобы доказать свою основную лемму поднятия, Робинсону пришлось изобрести унификацию и определить все свойства наиболее общих унификаторов. Мы здесь не будем повторять доказательство Робинсона, а просто проиллюстрируем применение этой леммы следующим образом:

$$\begin{aligned} C_1 &= \neg P(x, F(x, A)) \vee \neg Q(x, A) \vee R(x, B) \\ C_2 &= \neg N(G(y), z) \vee P(H(y), z) \\ C_1' &= \neg P(H(B), F(H(B), A)) \vee \neg Q(H(B), A) \vee R(H(B), B) \\ C_2' &= \neg N(G(B), F(H(B), A)) \vee P(H(B), F(H(B), A)) \\ C' &= \neg N(G(B), F(H(B), A)) \vee \neg Q(H(B), A) \vee R(H(B), B) \\ C &= \neg N(G(y), F(H(y), A)) \vee \neg Q(H(y), A) \vee R(H(y), B) \end{aligned}$$

Очевидно, что C' — действительно базовый экземпляр выражения C . Вообще говоря, для того чтобы выражения C_1' и C_2' имели какие-либо резольвенты, они должны быть получены путем предварительного применения к выражениям C_1 и C_2 наиболее общего унификатора для пары взаимно дополнительных литералов в C_1 и C_2 . Из леммы поднятия можно легко получить аналогичное, приведенное ниже утверждение о любой последовательности применений правила резолюции.

Для любого выражения C' в резолюционном замыкании множества выражений S' существует выражение C в резолюционном замыкании множества выражений S , такое, что C' — базовый экземпляр выражения C и логический вывод C имеет такую же длину, как и логический вывод C' .

Из этого факта следует, что если в резолюционном замыкании множества выражений S' появляется пустое выражение, оно должно также появиться в резолюционном замыкании множества выражений S . Это связано с тем, что пустое выражение не может быть базовым экземпляром любого другого выражения. Подведем итог: мы показали, что если множество выражений S невыполнимо, то для него существует конечная процедура логического вывода пустого выражения с помощью правила резолюции.

Поднятие способа доказательства теорем от базовых выражений к выражениям первого порядка обеспечивает огромное увеличение мощи доказательства. Это увеличение связано с тем фактом, что теперь в доказательстве первого порядка конкретизация переменных может выполняться только по мере того, как это потребуется для самого доказательства, тогда как в методах с использованием базовых выражений приходилось исследовать огромное количество произвольных конкретизаций.

Учет отношения равенства

Ни в одном из методов логического вывода, описанных до сих пор в этой главе, не учитывалось отношение равенства. Для решения этой задачи может быть принято три различных подхода. Первый подход состоит в аксиоматизации равенства — включении в базу знаний высказываний, касающихся отношения равенства. При этом необходимо описать, что отношение равенства является рефлексивным, симметричным и транзитивным, а также сформулировать утверждение, что мы можем в любом предикате или функции заменять равные литералы равными. Таким образом, требуются три базовые аксиомы и еще по одной аксиоме для каждого предиката и функции, как показано ниже.

$$\begin{aligned} \forall x \ x = x \\ \forall x, y \ x = y \ \heartsuit \ y = x \\ \forall x, y, z \ x = y \wedge y = z \ \heartsuit \ x = z \\ \forall x, y \ x = y \ \heartsuit \ (P_1(x) \Leftrightarrow P_1(y)) \\ \forall x, y \ x = y \ \heartsuit \ (P_2(x) \Leftrightarrow P_2(y)) \\ \dots \\ \forall w, x, y, z \ w = y \wedge x = z \ \heartsuit \ (F_1(w, x) = F_1(y, z)) \\ \forall w, x, y, z \ w = y \wedge x = z \ \heartsuit \ (F_2(w, x) = F_2(y, z)) \\ \dots \end{aligned}$$

При наличии в базе знаний таких высказываний любая стандартная процедура логического вывода, такая как резолюция, позволяет решать задачи, требующие формирования рассуждений с учетом отношения равенства, например находить решения математических уравнений.

Еще один способ учета отношения равенства состоит в использовании дополнительного правила логического вывода. В простейшем правиле, правиле **демодуляции**, берется единичное выражение $x=y$, после чего терм y подставляется вместо любого терма, который унифицируется с термом x в каком-то другом выражении. Более формально эту идею можно представить, как описано ниже.

- **Демодуляция.** Для любых термов x , y и z , где $\text{Unify}(x, z) = \theta$ и $m_n[z]$ — литерал, содержащий z , справедливо следующее:

$$\frac{x = y, m_1 \vee \dots \vee m_n[z]}{m_1 \vee \dots \vee m_n[\text{Subst}(\theta, y)]}$$

Демодуляция обычно используется для упрощения выражений с помощью коллекции утверждений, таких как $x+0=x$, $x^1=x$ и т.д. Это правило может быть также дополнено, чтобы можно было учитывать неединичные выражения, в которых появляется литерал со знаком равенства, как показано ниже.

- **Парамодуляция.** Для любых термов x , y и z , где $\text{Unify}(x, z)=\theta$, справедливо следующее:

$$\frac{\ell_1 \vee \dots \vee \ell_k \vee x = y, m_1 \vee \dots \vee m_n[z]}{\text{Subst}(\theta, \ell_1 \vee \dots \vee \ell_k \vee m_1 \vee \dots \vee m_n[y])}$$

В отличие от демодуляции, парамодуляция позволяет получить полную процедуру логического вывода для логики первого порядка с отношением равенства.

В третьем подходе формирование логических рассуждений с учетом равенства полностью осуществляется с помощью расширенного алгоритма унификации. Это означает, что термы рассматриваются как унифицируемые, если можно доказать, что они становятся равными при некоторой подстановке; здесь выражение “можно доказать” допускает включение в определенном объеме рассуждений о равенстве. Например, термы $1+2$ и $2+1$ обычно не рассматриваются как унифицируемые, но алгоритм унификации, в котором известно, что $x+y=y+x$, способен унифицировать их с помощью пустой подстановки. **Унификация с учетом равенства** (equational unification) такого рода может выполняться с помощью эффективных алгоритмов, разработанных с учетом данных конкретных используемых аксиом (коммутативность, ассоциативность и т.д.), а не с помощью явного логического вывода на основе этих аксиом. Программы автоматического доказательства теорем с использованием этого метода очень близки к системам логического программирования в ограничениях, описанным в разделе 9.4.

Стратегии резолюции

Как известно, повторные применения правила логического вывода на основе резолюции позволяют в конечном итоге найти доказательство, если оно существует, а в этом подразделе рассматриваются стратегии, позволяющие находить доказательства не методом перебора, а более эффективно.

Преимущественное использование единичных выражений

В этой стратегии преимущество отдается таким операциям резолюции, в которых одним из высказываний является единственный литерал (известный также как **единичное выражение** — unit clause). В основе этой стратегии лежит такая идея, что если осуществляются попытки получения пустого выражения, то может оказаться целесообразным отдавать предпочтение таким операциям логического вывода, в которых вырабатываются более короткие выражения. Применение операции резолюции к единичному высказыванию (такому как P) в сочетании с любым другим высказыванием (таким как $\neg P \vee \neg Q \vee R$) всегда приводит к получению выражения (в данном случае $\neg Q \vee R$), более короткого, чем это другое высказывание. Когда стратегия с преимущественным использованием единичных выражений была впервые опи-

робована в пропозициональном логическом выводе в 1964 году, она привела к резкому ускорению работы, обеспечив возможность доказывать теоремы, с которыми не удавалось справиться без использования этого метода предпочтения. Тем не менее метод предпочтения единичных выражений, отдельно взятый, не позволяет уменьшить коэффициент ветвления в задачах средних размеров до такой степени, чтобы можно было обеспечить возможность их решения с помощью резолюции. Несмотря на это, он представляет собой полезный эвристический метод, который может успешно использоваться в сочетании с другими стратегиями.

☞ **Единичная резолюция** (unit resolution) — это ограниченная форма резолюции, в которой на каждом этапе резолюции должно участвовать единичное выражение. В общем случае метод единичной резолюции является неполным, но становится полным при его применении к хорновским базам знаний. Процесс доказательства по методу единичной резолюции применительно к хорновским базам знаний напоминает прямой логический вывод.

Множество поддержки

Применение метода предпочтений, в котором в первую очередь осуществляется попытка выполнить определенные операции резолюции, вполне оправдано, но, вообще говоря, более эффективный метод может быть основан на том, что следует попытаться полностью устранить некоторые потенциальные этапы резолюции. В стратегии с использованием множества поддержки выполняется именно это. Применение данной стратегии начинается с выявления подмножества высказываний, называемого ☞ **множеством поддержки** (set of support). На каждом этапе резолюции высказывание из множества поддержки комбинируется с другим высказыванием, а резольвента добавляется к множеству поддержки. Если множество поддержки является небольшим по сравнению со всей базой знаний, это позволяет резко сократить пространство поиска.

При использовании этого подхода необходимо соблюдать осторожность, поскольку при неправильном выборе множества поддержки алгоритм может стать неполным. Однако, если множество поддержки S будет выбрано так, чтобы оставшиеся высказывания, вместе взятые, оставались выполнимыми, то резолюция с помощью множества поддержки становится полной. Общепринятый подход, основанный на предположении, что первоначальная база знаний является непротиворечивой, состоит в том, чтобы в качестве множества поддержки применялся отрицаемый запрос. (В конце концов, если база знаний не является непротиворечивой, то сам факт, что запрос является следствием из нее, становится избыточным, поскольку из противоречия можно доказать все, что угодно.) Стратегия с использованием множества поддержки имеет дополнительное преимущество в том, что в ней часто вырабатываются деревья доказательства, легко доступные для понимания людей, поскольку само формирование доказательства осуществляется целенаправленно.

Резолюция с входными высказываниями

В стратегии ☞ **резолюции с входными высказываниями** (input resolution) на каждом этапе резолюции комбинируется одно из входных высказываний (из базы знаний или запроса) с некоторым другим высказыванием. В доказательстве, показанном на рис. 9.7, использовались только этапы резолюции с входными высказываниями и поэтому дерево доказательства имело характерную форму в виде единого “хребта” с отдельными высказываниями, комбинирующимися с этим хребтом. Очевидно, что

пространство деревьев доказательства такой формы меньше по сравнению с пространством всех возможных графов доказательства. В хорновских базах знаний как своего рода стратегия резолюции с входными высказываниями может рассматриваться правило отдаления, поскольку при использовании этого правила некоторая импликация из первоначальной базы знаний комбинируется с некоторыми другими высказываниями. Таким образом, нет ничего удивительного в том, что метод резолюции с входными высказываниями является полным применительно к базам знаний, которые находятся в хорновской форме, но в общем случае он неполон. Стратегия \simeq **линейной резолюции** (linear resolution) представляет собой небольшое обобщение, в котором допускается применять в одной операции резолюции высказывания P и Q , если P находится в первоначальной базе знаний или P является предком Q в дереве доказательства. Метод линейной резолюции является полным.

Обобщение

В методе \simeq **обобщения** (subsumption) устраняются все высказывания, которые обобщаются некоторым существующим высказыванием из базы знаний (т.е. являются более конкретными по сравнению с ним). Например, если в базе знаний есть высказывание $P(x)$, то нет смысла вводить в нее высказывание $P(A)$ и еще меньше смысла вводить $P(A) \vee Q(B)$. Обобщение позволяет поддерживать небольшие размеры базы знаний и тем самым ограничивать размеры пространства поиска.

Средства автоматического доказательства теорем

Средства автоматического доказательства теорем (называемые также *средствами автоматизированного формирования рассуждений*) отличаются от языков логического программирования в двух отношениях. Во-первых, большинство языков логического программирования поддерживает только хорновские выражения, тогда как средства автоматического доказательства теорем поддерживают полную логику первого порядка. Во-вторых, в программах на таком типичном языке логического программирования, как Prolog, переплетаются логика и управление. Например, выбор программистом выражения $A :- B, C$ вместо $A :- C, B$ может повлиять на выполнение программы. С другой стороны, в большинстве средств автоматического доказательства теорем синтаксическая форма, выбранная для высказываний, не влияет на результаты. Для средств автоматического доказательства теорем все еще требуется управляющая информация, чтобы они могли функционировать эффективно, но эта информация обычно хранится отдельно от базы знаний, а не входит в состав самого представления знаний. Большинство исследований в области средств автоматического доказательства теорем посвящено поиску стратегий управления, которые приводят к общему повышению эффективности, а не только к увеличению быстродействия.

Проект одного из средств автоматического доказательства теорем

В этом разделе описана программа автоматического доказательства теорем Otter (Organized Techniques for Theorem-proving and Effective Research) [1018]; в этом описании особое внимание будет уделено применяемой в ней стратегии управления. Подготавливая любую задачу для программы Otter, пользователь должен разделить знания на четыре описанные ниже части.

- Множество выражений, известное как **множество поддержки** (или *sos* — set of support), в котором определяются важные факты о данной задаче. На каждом этапе резолюции операция резолюции применяется к одному из элементов множества поддержки и к другой аксиоме, поэтому поиск сосредоточивается на множестве поддержки.
- Множество **полезных аксиом** (usable axiom), которое выходит за пределы множества поддержки. Эти аксиомы предоставляют фоновые знания о проблемной области. Определение границы между тем, что должно войти в состав задачи (и поэтому в множество *sos*) и что относится к фоновым знаниям (и поэтому должно войти в число полезных аксиом), передается на усмотрение пользователя.
- Множество уравнений, известных как **правила перезаписи** (rewrites), или **деמודуляторы** (demodulators). Хотя демодуляторы представляют собой уравнения, они всегда применяются в направлении слева направо, поэтому определяют каноническую форму, в которой должны быть представлены все упрощенные термы. Например, демодулятор $x+0=x$ указывает, что любой терм в форме $x+0$ должен быть заменен термом x .
- Множество параметров и выражений, который определяет стратегию управления. В частности, пользователь задает эвристическую функцию для управления поиском и функцию фильтрации для устранения некоторых подцелей как не представляющих интереса.

Программа Otter действует по принципу постоянного применения правила резолюции к одному из элементов множества поддержки и к одной из полезных аксиом. В отличие от системы Prolog, в этой программе используется определенная форма поиска по первому наилучшему совпадению. Ее эвристическая функция измеряет “вес” каждого выражения с учетом того, что наиболее предпочтительными являются выражения с наименьшими весами. Задача точной формулировки эвристической функции возлагается на пользователя, но, вообще говоря, вес любого выражения должен коррелировать с его размером или сложностью. Единичные выражения оцениваются как имеющие наименьший вес, поэтому такой метод поиска может рассматриваться как обобщение стратегии с преимущественным использованием единичных выражений. На каждом этапе программа Otter перемещает выражение “с наименьшим весом” из множества поддержки в список полезных аксиом и добавляет в множество поддержки некоторые непосредственные следствия применения операции резолюции к выражению с наименьшим весом и к элементам списка полезных аксиом. Программа Otter останавливается, если обнаруживает противоречие или если возникает такая ситуация, что в множестве поддержки не остается больше выражений. Алгоритм работы этой программы показан более подробно в листинге 9.5.

Листинг 9.5. Набросок структуры программы автоматического доказательства теорем Otter. Эвристическое управление применяется при выборе выражения “с наименьшим весом” и в функции Filter, которая устраняет из рассмотрения такие выражения, которые не представляют интереса

```

procedure Otter(sos, usable)
  inputs: sos, множество поддержки - выражения, определяющие
           решаемую задачу (глобальная переменная)
           usable, множество фоновых знаний, которые потенциально

```

могут быть релевантными для данной задачи

repeat

clause ← элемент множества *sos* с наименьшим весом
переместить выражение *clause* из множество *sos*
в множество *usable*
Process(Infer(*clause*, *usable*), *sos*)

until *sos* = [] **or** обнаружится опровержение

function Infer(*clause*, *usable*) **returns** множество выражений *clauses*

применить правило резолюции к выражению *clause* и каждому элементу
множества *usable*
возвратить полученное множество *clauses* после применения
функции Filter

procedure Process(*clauses*, *sos*)

for each *clause* **in** *clauses* **do**

clause ← Simplify(*clause*)
выполнить слияние идентичных литералов
отбросить выражение *clause*, если оно представляет
собой тавтологию

sos ← [*clause* | *sos*]

if *clause* не имеет литералов, то обнаружено опровержение

if *clause* имеет один литерал, то искать
единичное опровержение

Расширение системы Prolog

Еще один способ создания средства автоматического доказательства теорем состоит в том, чтобы начать с компилятора Prolog и дополнить его в целях получения непротиворечивого и полного средства формирования рассуждений для полной логики первого порядка. Именно этот подход был принят при создании программы РТТР (Prolog Technology Theorem Prover) [1463]. Как описано ниже, программа РТТР включает пять существенных дополнений к системе Prolog, позволяющих восстановить полноту и выразительность алгоритма обратного логического вывода.

- В процедуру унификации снова вводится проверка вхождения для того, чтобы эта процедура стала непротиворечивой.
- Поиск в глубину заменяется поиском с итеративным углублением. Это позволяет добиться того, чтобы стратегия поиска стала полной, а увеличение продолжительности поиска измерялось лишь постоянной зависимостью от времени.
- Разрешается применение отрицаемых литералов (таких как $\neg P(x)$). В этой реализации имеется две отдельные процедуры; в одной из них предпринимается попытка доказать P , а в другой — доказать $\neg P$.
- Выражение с n атомами хранится в виде n различных правил. Например, при наличии в базе знаний выражения $A \leftarrow B \wedge C$ должно быть также предусмотрено хранение в ней этого выражения, представленного как $\neg B \leftarrow C \wedge \neg A$ и как $\neg C \leftarrow B \wedge \neg A$. Применение такого метода, известного под названием

☞ **блокирование** (locking), означает, что текущая цель требует унификации только с головой каждого выражения, и вместе с тем позволяет должным образом учитывать отрицание.

- Логический вывод сделан полным (даже для нехорновских выражений) путем добавления правила резолюции с линейным входным выражением: если текущая цель унифицируется с отрицанием одной из целей в стеке, то данная цель может рассматриваться как решенная. В этом состоит один из способов рассуждения от противного. Предположим, что первоначальной целью было высказывание P и что эта цель свелась в результате применения ряда этапов логического вывода к цели $\neg P$. Тем самым установлено, что $\neg P \heartsuit P$, а это выражение логически эквивалентно P .

Несмотря на эти изменения, программа РТТР сохраняет свойства, благодаря которым обеспечивается высокое быстродействие системы Prolog. Операции унификации все еще осуществляются посредством непосредственной модификации переменных, а отмена связывания выполняется путем разгрузки контрольного стека во время возврата. Стратегия поиска все еще основана на резолюции с применением входных выражений, а это означает, что в каждой операции резолюции участвует одно из выражений, содержащихся в первоначальной формулировке задачи (а не какое-то производное выражение). Такой подход позволяет осуществить компиляцию всех выражений из первоначальной формулировки задачи.

Основным недостатком программы РТТР является то, что пользователь должен отказаться от любых попыток взять на себя управление поиском решений. Каждое правило логического вывода в этой системе используется и в его первоначальной, и в контрапозитивной форме. Это может привести к выполнению таких операций поиска, которые противоречат здравому смыслу. Например, рассмотрим следующее правило:

$$(f(x, y) = f(a, b)) \Leftarrow (x = a) \wedge (y = b)$$

Если бы оно рассматривалось как правило Prolog, то применялся бы разумный способ доказательства того, что два терма f равны. Но в системе РТТР должно быть также сформировано контрапозитивное правило:

$$(x \neq a) \Leftarrow (f(x, y) \neq f(a, b)) \wedge (y = b)$$

По-видимому, попытка доказать, что любые два терма, x и a , являются разными, привела бы к непроизводительным затратам ресурсов.

Применение средств автоматического доказательства теорем в качестве помощников

До сих пор в этой книге любая система формирования рассуждений рассматривалась как независимый агент, который должен был принимать решения и действовать самостоятельно. Еще одно направление использования средств автоматического доказательства теорем состоит в том, что они должны служить в качестве помощников, предоставляя консультации, скажем, математикам. При эксплуатации подобных систем в режиме помощи математик действует в роли руководителя, очерчивая стратегию определения того, что должно быть сделано в следующую очередь, а затем передавая системе автоматического доказательства теорем просьбу проработать все детали. Это позволяет в определенной степени устранить проблему полурешимости, поскольку руководитель научной разработки может отменить запрос и опробовать другой подход, если поиск ответа на запрос потребовал слишком много

времени. Любая система автоматического доказательства теорем может также действовать в качестве **средства проверки доказательства**; при ее использовании в таком режиме доказательство предоставляется человеком в виде ряда довольно крупных этапов; отдельные операции логического вывода, которые требуются для того, чтобы показать, что каждый из этих этапов является непротиворечивым, определяются системой.

В частности, **сократовское средство формирования рассуждений** (socratic reasoner) представляет собой такую систему автоматического доказательства теорем, в которой функция Ask является неполной, но эта система всегда позволяет прийти к определенному решению, если ей будет задан правильный ряд вопросов. Таким образом, сократовские средства формирования рассуждений становятся хорошими помощниками, при условии, что есть руководитель, способный составить правильный ряд вызовов функции Ask. Одной из сократовских систем формирования рассуждений для математиков является Ontic [1005].

Области практического использования средств автоматического доказательства теорем

Средства автоматического доказательства теорем позволили получить новейшие математические результаты. Программа SAM (Semi-Automated Mathematics) была первой программой, с помощью которой удалось доказать одну из лемм в теории решеток [602]. Кроме того, программа Aura позволила найти ответ на многие открытые вопросы в нескольких областях математики [1621]. Программа автоматического доказательства теорем Бойера–Мура [165] применялась и дорабатывалась в течение многих лет, и ею воспользовался Натараджан Шанкар для получения первого полностью строгого формального доказательства теоремы Гёделя о неполноте [1393]. Одним из самых строгих средств автоматического доказательства теорем является программа Otter; она использовалась для решения некоторых открытых задач в области комбинаторной логики. Наиболее известные из них касаются **алгебры Роббинса**. В 1933 году Герберт Роббинс предложил простое множество аксиом, которые, на первый взгляд, могли служить для определения булевой алгебры, но ни одного доказательства этой гипотезы найти не удавалось (несмотря на напряженную работу нескольких выдающихся математиков, включая самого Альфреда Тарского). Наконец, 10 октября 1996 года после восьми дней вычислений программа EQR (одна из версий программы Otter) нашла такое доказательство [1019].

Средства автоматического доказательства теорем могут также применяться для решения задач, связанных с **проверкой** и **синтезом** как аппаратных, так и программных средств, поскольку для обеих этих проблемных областей могут быть предусмотрены правильные варианты аксиоматизации. Поэтому исследования доказательства теорем проводятся не только в искусственном интеллекте, но и в таких областях, как проектирование аппаратных средств, языки программирования и разработка программного обеспечения. В случае программного обеспечения аксиомы определяют свойства каждого синтаксического элемента языка программирования. (Процесс формирования рассуждений о программах полностью аналогичен процессу формирования рассуждений о действиях в ситуационном исчислении.) Программный алгоритм проверяется путем демонстрации того, что его выходы соответствуют спецификациям для всех входов. Таким образом были проверены алгоритм шифрования RSA с открытым ключом и алгоритм согласования строк Бойера–Мура [166]. В случае аппаратного обеспечения аксиомы описывают способы взаи-

модействия сигналов и элементов схемы (один из примеров приведен в главе 8). Программой Auga [1610] был проверен проект 16-битового сумматора. Системы формирования логических рассуждений, предназначенные специально для проверки аппаратных средств, оказались способными проверять целые процессоры, включая свойства синхронизации этих процессоров [1455].

Формальный синтез алгоритмов был одним из первых направлений использования средств автоматического доказательства теорем, как и было намечено Корделлом Грином [591], который опирался на идеи, высказанные ранее Саймоном [1416]. Общий замысел состоит в том, что должна быть доказана теорема с утверждением, что “существует программа p , удовлетворяющая некоторой спецификации”. Если удастся ограничить это доказательство конструктивной формой, то появляется возможность извлечь из результатов доказательства требуемую программу. Хотя полностью автоматизированный \approx **дедуктивный синтез**, как стало называться это направление, еще не осуществим применительно к программированию задач общего назначения, дедуктивный синтез, управляемый вручную, оказался успешным при проектировании нескольких новейших и сложнейших алгоритмов. Кроме того, активной областью исследования является синтез программ специального назначения. В области синтеза аппаратных средств программа автоматического доказательства теорем Auga применялась для проектирования схем, оказавшихся более компактными по сравнению с разработанными во всех предыдущих проектах [1609]. Для многих проектов логических схем достаточно применить пропозициональную логику, поскольку множество интересующих высказываний является фиксированным благодаря конечным размерам множества схемных элементов. В наши дни применение пропозиционального логического вывода в аппаратном синтезе является стандартным методом, имеющим много крупномасштабных областей использования (см., например, работу Новика и др. [1149]).

Те же методы теперь начинают также применяться для проверки программного обеспечения с помощью таких систем, как программа проверки моделей Spin [672]. Например, с ее помощью была проверена до и после полета программа управления космическим аппаратом Remote Agent [633].

9.6. РЕЗЮМЕ

В этой главе приведен анализ логического вывода в логике первого порядка и многих алгоритмов его выполнения.

- В первом подходе используется правило логического вывода для конкретизации кванторов в целях преобразования задачи логического вывода в форму пропозициональной логики. Как правило, этот подход характеризуется очень низким быстродействием.
- Использование **унификации** для выявления подходящих подстановок для переменных позволяет устранить этап конкретизации в доказательствах первого порядка, в результате чего этот процесс становится гораздо более эффективным.
- В поднятой версии **правила отделения** унификация применяется для получения естественного и мощного правила логического вывода — **обобщенного правила**

отделения. В алгоритмах **прямого логического вывода** и **обратного логического вывода** это правило применяется к множествам определенных выражений.

- Обобщенное правило отделения является полным применительно к определенным выражениям, но проблема логического следствия остается **полуразрешимой**. Для программ **Datalog**, состоящих из определенных выражений без функций, проблема логического следствия разрешима.
- Прямой логический вывод используется в **дедуктивных базах данных**, где он может сочетаться с реляционными операциями баз данных. Он также применяется в **продукционных системах**, которые обеспечивают эффективное обновление при наличии очень больших наборов правил.
- Прямой логический вывод для программ Datalog является полным и выполняется за время, определяемое полиномиальной зависимостью.
- Обратный логический вывод используется в **системах логического программирования**, таких как **Prolog**, в которых реализована сложная технология компиляции для обеспечения очень быстрого логического вывода.
- Недостатками обратного логического вывода являются излишние этапы логического вывода и бесконечные циклы; эти недостатки можно устранить путем **запоминания**.
- Обобщенное правило логического вывода на основе **резолуции** позволяет создать полную систему доказательства для логики первого порядка с использованием баз знаний в конъюнктивной нормальной форме.
- Существует несколько стратегий сокращения пространства поиска для систем с резолюцией без потери полноты. Эффективные средства автоматического доказательства теорем на основе резолюции использовались для доказательства интересных математических теорем, а также для проверки и синтеза программных и аппаратных средств.

БИБЛИОГРАФИЧЕСКИЕ И ИСТОРИЧЕСКИЕ ЗАМЕТКИ

Логический вывод широко исследовался древнегреческими математиками. Аристотель тщательно исследовал один из типов логического вывода, называемый **силлогизмом**, который представляет собой своего рода правило логического вывода. Силлогизмы Аристотеля включали элементы логики первого порядка, такие как кванторы, но были ограничены унарными предикатами. Силлогизмы классифицировались по “фигурам” и “модусам”, в зависимости от порядка термов (которые следовало бы назвать предикатами) в высказываниях и от степени общности (которую теперь принято интерпретировать с помощью кванторов), применяемой к каждому терму, а также с учетом того, является ли каждый терм отрицаемым. Наиболее фундаментальным силлогизмом является тот, который относится к первой фигуре первого модуса:

Все S суть M .

Все M суть P .

Следовательно, все S суть P .

Аристотель пытался доказать истинность других силлогизмов, “приводя” их к силлогизмам первой фигуры. Описание того, в чем должно состоять такое “приведение”, оказалось гораздо менее точным по сравнению с описанием, в котором были охарактеризованы сами фигуры и модусы силлогизмов.

Готтлоб Фреге, который разработал полную логику первого порядка в 1879 году, основал свою систему логического вывода на большой коллекции логически правильных схем и единственном правиле логического вывода — правиле отделения (Modus Ponens). Фреге воспользовался тем фактом, что результат применения любого правила логического вывода в форме “Из P вывести Q ” можно моделировать путем применения к высказыванию P правила отделения наряду с логически правильной схемой $P \heartsuit Q$. Такой “аксиоматический” стиль представления знаний, в котором наряду с правилом отделения использовался целый ряд логически правильных схем, был принят на вооружение после Фреге многими логиками; наиболее замечательным является то, что этот стиль использовался в основополагающей книге *Principia Mathematica* [1584].

В подходе на основе **натуральной дедукции** (natural deduction), который был введен Герхардом Генценом [541] и Станиславом Яськовским [725], основное внимание было сосредоточено не на аксиоматических системах, а на правилах логического вывода. Натуральная дедукция получила название “натуральной”, поскольку в ней не требуется преобразование в нормальную форму (неудобную для восприятия человеком), а также в связи с тем, что в ней применяются такие правила логического вывода, которые кажутся естественными для людей. Правитц [1235] посвятил описанию натуральной дедукции целую книгу. Галье [515] применил подход Генцена для разъяснения теоретических основ автоматизированной дедукции.

Крайне важным этапом в разработке глубокого математического анализа логики первого порядка явилось изобретение формы представления в виде импликационных выражений (clausal form). Уайтхед и Рассел [1584] описали так называемые правила прохождения (rules of passage) (фактически этот термин принадлежит Эрбрану [650]), которые используются для перемещения кванторов в переднюю часть формул. Торальфом Сколемом [1424] были достаточно своевременно предложены сколемовские константы и сколемовские функции. Общая процедура сколемизации, наряду с важным понятием универсума Эрбрана, описана в [1425].

Крайне важную роль в разработке автоматизированных методов формирования рассуждений, как до, так и после введения Робинсоном правила резолюции, играет теорема Эрбрана, названная в честь французского логика Жака Эрбрана [650]. Это отражается в применяемом нами термине “универсум Эрбрана”, а не “универсум Сколема”, даже несмотря на то, что это понятие в действительности было открыто Сколемом. Кроме того, Эрбран может считаться изобретателем операции унификации. Гёдель [565], опираясь на идеи Сколема и Эрбрана, сумел показать, что для логики первого порядка имеется полная процедура доказательства. Алан Тьюринг [1518] и Алонсо Черч [255] практически одновременно продемонстрировали, используя очень разные доказательства, что задача определения общезначимости в логике первого порядка не имеет решения. В превосходной книге Эндертонна [438] все эти результаты описаны в строгой, но труднодоступной для понимания манере.

Хотя Маккарти [1009] предложил использовать логику первого порядка для представления знаний и формирования рассуждений в искусственном интеллекте, первые подобные системы были разработаны логиками, заинтересованными в получе-

нии средств автоматического доказательства математических теорем. Впервые применение метода пропозиционализации и теоремы Эрбрана предложено Абрахамом Робинсоном, а Гилмор [556] написал первую программу, основанную на этом подходе. Дэвис и Патнем [336] применили форму представления в виде импликационных выражений и разработали программу, в которой предпринимались попытки поиска противоречий путем подстановки элементов универсума Эрбрана вместо переменных для получения базовых выражений, а затем поиска пропозициональных противоречивостей среди этих базовых выражений. Правиц [1234] разработал ключевую идею, позволяющую использовать для управления процессом поиска тенденцию к обнаружению пропозициональных противоречивостей и вырабатывать термы из универсума Эрбрана, только если это необходимо для определения пропозициональной противоречивости. После дальнейшей разработки этой идеи другими исследователями Дж.Э. Робинсон (который не связан родством с Абрахамом Робинсоном) пришел к созданию метода резолюции [1298]. Примерно в то же время советским исследователем С. Масловым [994], [995] на основе немного иных принципов был разработан так называемый *инверсный метод*, который характеризуется некоторыми вычислительными преимуществами над пропозиционализацией. **Метод соединения** Вольфганга Бибеля [123] может рассматриваться как расширение этого подхода.

После разработки метода резолюции исследования в области логического вывода первого порядка стали развиваться в нескольких разных направлениях. В искусственном интеллекте метод резолюции применялся для создания систем поиска ответов на вопросы Корделлом Грином и Бертрамом Рафаэлем [593]. Несколько менее формальный подход был принят Карлом Хьюиттом [651]. Разработанный им язык Planner, хотя и не был полностью реализован, явился предшественником логического программирования и включал директивы для прямого и обратного логического вывода и для отрицания как недостижения цели. А подмножество этого языка, известное как Micro-Planner [1475], было реализовано и использовалось в системе понимания естественного языка Shrdlu [1601]. В ранних реализациях систем искусственного интеллекта большие усилия направлялись на разработку структур данных, которые должны были обеспечить эффективную выборку фактов; эти работы описаны в книгах по программированию для искусственного интеллекта [240], [479], [1148].

В начале 1970-х годов в искусственном интеллекте полностью утвердился метод **прямого логического вывода** как легко доступная пониманию альтернатива методу резолюции. Прямой логический вывод использовался в самых различных системах, начиная от программы автоматического доказательства геометрических теорем Невинса [1123] и заканчивая экспертной системой R1 для разработки конфигурации компьютеров VAX [1026]. Приложения искусственного интеллекта обычно охватывают большое количество правил, поэтому было важно разработать эффективную технологию согласования с правилами, особенно для инкрементных обновлений. Для поддержки таких приложений была разработана технология **продукционных систем**. Язык продукционных систем Ops-5 [197], [482] использовался для экспертной системы R1 и для когнитивной архитектуры Soar [880]. В языке Ops-5 был включен процесс согласования с помощью rete-алгоритма [483]. Архитектура Soar, позволяющая вырабатывать новые правила для кэширования результатов предыдущих вычислений, способна создавать очень большие множества правил; например, в системе TacAir-Soar, предназначенной для управления тренажером, моделирующим самолет-истребитель [743], количество правил превышало один миллион. Язык

CLIPS [1626] продукционных систем на основе языка С, разработанный в NASA, обеспечивал лучшую интеграцию с другими программными, аппаратными и сенсорными системами и использовался для автоматизации космической станции и разработки нескольких военных приложений.

Большой вклад в понимание особенностей прямого логического вывода внесли также работы в области исследований, известной как **дедуктивные базы данных**. Исследования в этой области начались с симпозиума, организованного в Тулузе в 1977 году Джеком Минкером, который собрал вместе специалистов в области логического вывода и систем баз данных [514]. В опубликованном сравнительно недавно историческом обзоре [1264] сказано: “Дедуктивные системы [баз данных] были попыткой адаптировать язык Prolog, воплощающий видение мира с «малыми данными», к миру «больших данных»”. Таким образом, цель разработок в этой области состоит в объединении технологии баз данных, которая предназначена для выборки больших множеств фактов, с технологией логического вывода на основе языка Prolog, в которой обычно осуществляется выборка одновременно только одного факта. К числу работ в области дедуктивных баз данных относятся [228] и [1525].

Важная работа, выполненная Чандрой и Нарелом [231], а также Ульманом [1524], привела к признанию языка Datalog в качестве стандартного языка для дедуктивных баз данных. Кроме того, стал стандартным “восходящий” логический вывод, или прямой логический вывод, отчасти потому, что данный метод позволяет избежать проблем, обусловленных незавершаемыми и избыточными вычислениями, которые возникают при обратном логическом выводе, и отчасти потому, что имеет более естественную реализацию в терминах основных операций реляционной базы данных. Разработка метода **магических множеств** для перезаписи правил Бансильоном и др. [67] позволила воспользоваться в прямом логическом выводе преимуществами целенаправленности, свойственными обратному логическому выводу. Методы табулированного логического программирования (с. 1), вступившие в конкурентную борьбу с другими методами, заимствовали преимущества динамического программирования от прямого логического вывода.

Большой вклад в понимание сложностей логического вывода внесло сообщество пользователей дедуктивных баз данных. Чандра и Мерлин [232] впервые показали, что задача согласования единственного нерекурсивного правила (в терминологии баз данных — **конъюнктивного запроса**) может оказаться NP-трудной. Купер и Варди [870] предложили использовать понятие **сложности данных** (т.е. сложности как функции размера базы данных, в которой размер правила рассматривается как постоянный) в качестве подходящего критерия эффективности получения ответов на запросы. Готтлоб и др. [586] обсудили связь между конъюнктивными запросами и задачами удовлетворения ограничений, показав, как можно использовать способ декомпозиции гипердерева для оптимизации процесса согласования.

Как уже упоминалось выше, процедуры **обратного логического вывода**, применяемые для логического вывода, впервые появились в разработанном Хьюиттом языке Planner [651]. Но логическое программирование как таковое развивалось независимо от этого направления разработок. Ограниченная форма линейной резолюции, называемая **SL-резолюцией**, была разработана Ковальским и Кюннером [853] на основе метода **устранения моделей** Лавленда [947]; после применения этого метода к определенным выражениям он принял вид метода **SLD-резолюции**, который предоставил возможность осуществлять интерпретацию определенных выражений как про-

грамм [849–851]. Между тем в 1972 году французский исследователь Ален Колмерор разработал и реализовал **Prolog** в целях синтаксического анализа текста на естественном языке; первоначально выражения Prolog предназначались для использования в качестве правил контекстно-свободной грамматики [285], [1311]. Основная часть теоретических основ логического программирования разработана Ковальским в сотрудничестве с Колмерором. Семантическое определение с использованием наименьших фиксированных точек предложено Ван Эмденом и Ковальским [1530]. Ковальский и Коэн [274], [852] подготовили хорошие исторические обзоры истоков языка Prolog. Теоретический анализ основ языка Prolog и других языков логического программирования приведен в книге *Foundations of Logic Programming* [940].

Эффективные компиляторы Prolog главным образом основаны на модели абстрактной машины Уоррена (Warren Abstract Machine — WAM), разработанной Дэвидом Г.Д. Уорреном [1556]. Ван Рой [1536] показал, что благодаря применению дополнительных методов организации работы компилятора, таких как логический вывод типов, программы Prolog становятся способными конкурировать по быстродействию с программами С. Рассчитанный на 10 лет исследовательский проект создания компьютера пятого поколения, который был развернут в Японии в 1982 году, опирался полностью на язык Prolog, применяемый в качестве средства разработки интеллектуальных систем.

Методы предотвращения нежелательного заикливания в рекурсивных логических программах были разработаны независимо Смитом и др. [1434], а также Тамаки и Сато [1487]. Кроме того, последняя статья включала данные о методе запоминания, предназначенном для логических программ, который интенсивно разрабатывался в качестве метода **табулированного логического программирования** Дэвидом С. Уорреном. Свифт и Уоррен [1483] показали, как дополнить машину WAM для обеспечения табуляции, что позволяет добиться быстродействия программ Datalog, превышающего на порядок быстродействие дедуктивных систем баз данных с прямым логическим выводом.

Первые теоретические работы по логическому программированию в ограничениях были выполнены Джаффаром и Лассе [722]. Джаффар и др. [723] разработали систему CLP(R) для обработки ограничений с действительными значениями. В [724] приведено описание обобщенной машины WAM, на основе которой создана машина CLAM (Constraint Logic Abstract Machine — абстрактная машина логики ограничений) для разработки спецификаций различных реализаций систем CLP. В [10] описан сложный язык Life, в котором методы CLP сочетаются с функциональным программированием и формированием рассуждений в логике наследования. В [820] описан перспективный проект использования логического программирования в ограничениях в качестве основы архитектуры управления в реальном времени, которая может применяться для создания полностью автоматических средств вождения самолетов (автопилотов).

Объем литературы по логическому программированию и языку Prolog очень велик. Одной из первых книг по логическому программированию явилась книга *Logic for Problem Solving* [851]. Языку Prolog посвящены, в частности, книги [175], [270] и [1405]. Превосходный обзор тематики CLP приведен в [987]. До его закрытия в 2000 году официальным журналом для публикаций в этой области был *Journal of Logic Programming*; теперь вместо него выпускается журнал *Theory and Practice of Logic Programming*. К числу основных конференций по логическому программированию

относятся *International Conference on Logic Programming (ICLP)* и *International Logic Programming Symposium (ILPS)*.

Исследования в области **автоматического доказательства математических теорем** начались еще до того, как были впервые разработаны полные логические системы первого порядка. В разработанной Гербертом Гелернтером программе *Geometry Theorem Prover* [532] использовались методы эвристического поиска в сочетании с диаграммами для отсекаания ложных подцелей; с помощью этой программы удалось обосновать некоторые весьма сложные результаты математических исследований в области евклидовой геометрии. Но с тех пор взаимодействие таких научных направлений, как автоматическое доказательство теорем и искусственный интеллект, не слишком велико.

На первых порах основные усилия ученых были сосредоточены на проблемах полноты. Вслед за появлением оригинальной статьи Робинсона в работах [1619] и [1620] были предложены правила демодуляции и парамодуляции для формирования рассуждений с учетом отношения равенства. Эти правила были также разработаны независимо в контексте систем перезаписи термов [811]. Внедрение средств формирования рассуждений с учетом отношения равенства в алгоритм унификации было осуществлено Гордоном Плоткиным [1217]; применение таких средств было также предусмотрено в языке Qlisp [1339]. В [752] приведен обзор средств унификации с учетом отношения равенства на основе процедур перезаписи термов. Эффективные алгоритмы для стандартной унификации были разработаны Мартелли и Монтанари [989], а также Патерсоном и Вегманом [1181].

Кроме средств формирования рассуждений с учетом отношения равенства, в программы автоматического доказательства теорем были включены всевозможные процедуры принятия решений специального назначения. В [1120] предложена получившая широкое распространение схема интеграции подобных процедур в общую схему формирования рассуждений; к другим методам относятся “резолуция теории” Стикеля [1462] и “специальные отношения” Манна и Валдингера [978].

Для метода резолуции был предложен целый ряд стратегий управления, начиная со стратегии предпочтения единичного выражения [1616]. В [1617] была предложена стратегия с использованием множества поддержки, которая позволяет обеспечить определенную целенаправленность резолуции. Линейная резолуция впервые была предложена в [948]. В [537, глава 5] приведен краткий, но исчерпывающий анализ всего разнообразия стратегий управления.

В [602] описана одна из первых программ автоматического доказательства теорем, *Sam*, которая позволила решить одну из открытых проблем в теории решеток. В [1621] приведен краткий обзор того, какой вклад был внесен с помощью программы автоматического доказательства теорем *Aura* в решение открытых проблем в различных областях математики и логики. Это описание продолжено в [1018], где перечислены достижения программы *Otter*, преемника программы *Aura*, в решении открытых проблем. В [1563] описана программа *Spass* — одна из сильнейших современных программ автоматического доказательства теорем. Основным справочником по программе автоматического доказательства теорем Бойера–Мура является книга *A Computational Logic* [165]. В [1463] рассматривается система РТТР (*Prolog Technology Theorem Prover*), в которой сочетаются преимущества компиляции *Prolog* с полнотой устранения моделей [947]. Еще одной широко применяемой программой автоматического доказательства теорем, основанной на этом подходе, является *SETHEO* [915]; она способна выполнять несколько миллионов логических выводов

в секунду на рабочих станциях модели 2000. Эффективной программой автоматического доказательства теорем, реализованной всего лишь в 25 строках на языке Prolog, является LeanTaP [91].

Одни из первых работ в области автоматизированного синтеза программ были выполнены Саймоном [1416], Грином [591], а также Манна и Валдингером [976]. В трансформационной системе Бурстолла и Дарлингтона [211] используется формирование рассуждений с учетом отношения равенства для синтеза рекурсивных программ. Одной из самых сильных современных систем является Kids [1435], [1436]; она действует в качестве помощника эксперта. В [979] приведено учебное введение с описанием современного состояния дел в этой области, в котором основное внимание уделено описанию собственного дедуктивного подхода этих авторов. В книге *Automating Software Design* [954] собрано множество статей из этой области. Обзор примеров использования логики в проектировании аппаратных средств приведен в [791]; в [267] рассматривается применение метода проверки по модели для диагностирования аппаратных средств.



Хорошим справочником по темам полноты и неразрешимости является книга *Computability and Logic* [150]. Многие ранние статьи в области математической логики можно найти в книге *From Frege to Gödel: A Source Book in Mathematical Logic* [1532]. Официальным журналом для публикаций в области чистой математической логики (в отличие от автоматизированного дедуктивного логического вывода) является *The Journal of Symbolic Logic*. К числу учебников, посвященных автоматизированному дедуктивному логическому выводу, относятся классическая книга *Symbolic Logic and Mechanical Theorem Proving* [233], а также более новые работы [124], [776] и [1618]. Антология *Automation of Reasoning* [1408] включает много важных ранних статей по автоматизированному дедуктивному логическому выводу. Другие исторические обзоры приведены в [206] и [949]. Основным журналом для публикаций в области автоматического доказательства теорем является *Journal of Automated Reasoning*, а главной конференцией — ежегодно проводимая конференция *Conference on Automated Deduction (CADE)*. Кроме того, исследования в области автоматического доказательства теорем тесно связаны с работами по использованию логики при анализе программ и языков программирования, которым посвящена основная конференция *Logic in Computer Science*.

УПРАЖНЕНИЯ

- 9.1. Докажите на основании главных логических принципов, что процедура конкретизации с помощью квантора всеобщности является непротиворечивой и что процедура конкретизации с помощью квантора существования позволяет получить базу знаний, эквивалентную с точки зрения логического вывода.
- 9.2. Представляется вполне обоснованным утверждение, что из отдельного факта $Likes(Jerry, IceCream)$ можно вывести высказывание $\exists x Likes(x, IceCream)$. Запишите общее правило логического вывода, правило **введения квантора существования**, позволяющее узаконить такой логический вывод. Тщательно сформулируйте условия, которым должны удовлетворять переменные и термы, участвующие в этом выводе.

- 9.3.** Предположим, что база знаний содержит только одно высказывание, $\exists x \text{ AsHighAs}(x, \text{Everest})$. Какие из следующих фактов являются действительными результатами применения правила конкретизации с помощью квантора существования?
- $\text{AsHighAs}(\text{Everest}, \text{Everest})$.
 - $\text{AsHighAs}(\text{Kilimanjaro}, \text{Everest})$.
 - $\text{AsHighAs}(\text{Kilimanjaro}, \text{Everest}) \wedge \text{AsHighAs}(\text{BenNevis}, \text{Everest})$ (после двух применений).
- 9.4.** Для каждой приведенной ниже пары атомарных высказываний укажите наиболее общий унификатор, если он существует.
- $P(A, B, B), P(x, y, z)$.
 - $Q(y, G(A, B)), Q(G(x, x), y)$.
 - $\text{Older}(\text{Father}(y), y), \text{Older}(\text{Father}(x), \text{John})$.
 - $\text{Knows}(\text{Father}(y), y), \text{Knows}(x, x)$.
- 9.5.** Рассмотрите решетки обобщения, приведенные на рис. 9.1.
- Составьте решетку для высказывания $\text{Employs}(\text{Mother}(\text{John}), \text{Father}(\text{Richard}))$.
 - Составьте решетку для высказывания $\text{Employs}(\text{IBM}, y)$ (“Компания IBM является нанимателем для всех”). Не забудьте включить запрос любого рода, который унифицируется с этим высказыванием.
 - Предположим, что функция *Store* индексирует каждое высказывание под каждым узлом в его решетке обобщения. Объясните, как должна работать функция *Fetch*, если некоторые из этих высказываний содержат переменные; воспользуйтесь в качестве примера высказываниями, приведенными в упр. 9.5, а и 9.5, б, а также запросом $\text{Employs}(x, \text{Father}(x))$.
- 9.6.** Предположим, что в логическую базу данных помещена часть данных переписи населения США с указанием возраста, города проживания, даты рождения и имени матери каждого лица, с использованием номеров карточек социального страхования в качестве идентифицирующих констант для каждого лица. Таким образом, например, возраст Джорджа задается выражением $\text{Age}(443-65-1282, 56)$. Какая из приведенных ниже схем индексации S1–S5 позволяет эффективно находить ответы на каждый из запросов Q1–Q4 (при условии, что применяется обычный метод обратного логического вывода)?
- Схемы индексации.
 - S1. Индекс для каждого атомарного термина в каждой позиции.
 - S2. Индекс для каждого первого параметра.
 - S3. Индекс для каждого атомарного термина предиката.
 - S4. Индекс для каждой комбинации предиката и первого параметра.
 - S5. Индекс для каждой комбинации предиката и второго параметра и индекс для каждого первого (нестандартного) параметра.
 - Запросы.
 - Q1. $\text{Age}(443-44-4321, x)$.
 - Q2. $\text{ResidesIn}(x, \text{Houston})$.

- Q3. $Mother(x, y)$.
 - Q4. $Age(x, 34) \wedge ResidesIn(x, TinyTownUSA)$.
- 9.7. Можно было бы предположить, что стандартизация раз и навсегда отличий всех высказываний в базе знаний позволяет избежать проблемы конфликта переменных при унификации в процессе обратного логического вывода. Покажите, что для некоторых высказываний этот подход не применим. (*Подсказка.* Рассмотрите высказывание, одна часть которого унифицируется с другой.)
- 9.8. Объясните, как записать любую конкретную формулировку задачи 3-SAT произвольного размера с использованием единственного определенного выражения первого порядка и не больше 30 базовых фактов.
- 9.9. Запишите логические представления для приведенных ниже высказываний, применимые для использования с обобщенным правилом отделения.
- а) Лошади, коровы и свиньи — млекопитающие.
 - б) Рожденный лошастью — лошадь.
 - в) Блюбёрд — лошадь.
 - г) Блюбёрд — родитель Чарли.
 - д) Отношения “быть рожденным” и “быть родителем” — обратные.
 - е) Каждое млекопитающее имеет родителя.
- 9.10. В этом упражнении для получения ответов на вопросы с помощью алгоритма обратного логического вывода используются высказывания, записанные при решении упр. 9.9.
- а) Нарисуйте дерево доказательства, сформированное исчерпывающим алгоритмом обратного логического вывода для запроса $\exists h Horse(h)$ (“Существует некоторая лошадь”), в котором выражения согласуются в указанном порядке.
 - б) Какие особенности этой проблемной области вы обнаружили?
 - в) Какое количество решений для h фактически следует из ваших высказываний?
 - г) Можете ли вы предложить способ поиска всех этих решений? (*Подсказка.* Вам может потребоваться обратиться к [1434].)
- 9.11. Одной из известных детских английских загадок является следующая: “Brothers and sisters have I none, but that man's father is my father's son” (Братьев и сестер у меня нет, но отец этого человека — сын моего отца). С использованием правил из области семейных отношений (глава 8) определите, кто этот человек, о котором говорится в загадке. Вы можете применять любые методы логического вывода, описанные в этой главе. Почему, по вашему мнению, эту загадку трудно отгадать сразу?
- 9.12. Проследите за выполнением алгоритма обратного логического вывода, приведенного в листинге 9.3, при его применении для решения задачи доказательства преступления. Покажите, какую последовательность значений принимает переменная $goals$, и расположите эти значения в виде дерева.
- 9.13. Приведенный ниже код Prolog определяет предикат P :
- $$P(X, [X|Y]) .$$
- $$P(X, [Y|Z]) :- P(X, Z) .$$

- а) Покажите деревья доказательства и решения для запросов $P(A, [1, 2, 3])$ и $P(2, [1, A, 3])$.
- б) Какую стандартную операцию со списками представляет предикат P ?
- 9.14.**  В этом упражнении рассматривается применение сортировки в языке Prolog.
- а) Напишите выражения Prolog, которые определяют предикат $sorted(L)$, принимающий истинное значение тогда и только тогда, когда список L отсортирован в возрастающем порядке.
- б) Напишите на языке Prolog определение предиката $perm(L, M)$, который принимает истинное значение тогда и только тогда, когда L — перестановка M .
- в) Определите предикат $sort(L, M)$ (M — отсортированная версия L) с использованием предикатов $perm$ и $sorted$.
- г) Применяйте предикат $sort$ ко все более длинным и длинным спискам, пока вам это не надоест. Какова временная сложность вашей программы?
- д) Реализуйте на языке Prolog более быстрый алгоритм сортировки, такой как сортировка вставкой ($insert\ sort$) или быстрая сортировка ($quicksort$).
- 9.15.**  В этом упражнении рассматривается рекурсивное применение правил перезаписи с использованием логического программирования. Правилom перезаписи (или **демоулятором**, в терминологии программы Otter) является уравнение с указанным направлением применения. Например, правило перезаписи $x+0 \rightarrow x$ указывает, что любое выражение, которое согласуется с $x+0$, должно заменяться выражением x . Средства применения правил перезаписи составляют центральную часть систем формирования рассуждений с учетом отношения равенства. Для представления правил перезаписи мы будем использовать предикат $rewrite(X, Y)$. Например, приведенное выше правило перезаписи может быть представлено как $rewrite(x+0, x)$. Некоторые термы являются примитивными и не могут подвергаться дальнейшим упрощениям, поэтому мы будем использовать запись $primitive(0)$ для указания на то, что 0 — примитивный терм.
- а) Запишите определение предиката $simplify(X, Y)$, который принимает истинное значение, если Y — упрощенная версия X , т.е. к каким-либо подвыражениям Y больше не применимы какие-либо правила перезаписи.
- б) Запишите коллекцию правил перезаписи для упрощения выражений, в которых применяются арифметические операторы, и примените ваш алгоритм упрощения к некоторым примерам выражений.
- в) Запишите коллекцию правил перезаписи для символического дифференцирования и примените их наряду с определенными вами правилами упрощения для дифференцирования и упрощения выражений, в которых есть арифметические выражения, включая возведение в степень.
- 9.16.** В этом упражнении рассматривается реализация алгоритмов поиска на языке Prolog. Предположим, что предикат $successor(X, Y)$ принимает истинное значение, если состояние Y является преемником состояния X , и что предикат $goal(X)$ принимает истинное значение, если X — целевое состояние. Запишите определение для предиката $solve(X, P)$, который означает, что P — путь (список состояний), начинающийся от X , оканчивающийся в целевом

состоянии и состоящий из последовательности допустимых шагов, которые определены предикатом *successor*. Вы обнаружите, что простейшим способом решения этой задачи является поиск в глубину. Насколько легко будет ввести эвристическое управление поиском?

- 9.17.** Как можно воспользоваться методом резолюции для демонстрации того, что некоторое высказывание является общезначимым? Невыполнимым?
- 9.18.** Из высказывания “Лошади — животные” следует, что “голова лошади — голова животного”. Продемонстрируйте, что этот логический вывод является допустимым, выполнив приведенные ниже этапы.
- а)** Преобразуйте предпосылку и вывод этого высказывания в язык логики первого порядка. Воспользуйтесь тремя предикатами: *HeadOf*(*h*, *x*) (который означает, что “*h* — голова *x*”), *Horse*(*x*) и *Animal*(*x*).
 - б)** Примените отрицание к заключению и преобразуйте предпосылку и отрицаемое заключение в конъюнктивную нормальную форму.
 - в)** Воспользуйтесь правилом резолюции, чтобы показать, что заключение следует из предпосылки.
- 9.19.** Ниже приведены два высказывания на языке логики первого порядка.
- A.* $\forall x \exists y (x \geq y)$
B. $\exists y \forall x (x \geq y)$
- а)** Допустим, что переменные пробегают по всем натуральным числам $0, 1, 2, \dots, \infty$ и что предикат \geq означает “больше или равно”. При использовании такой интерпретации переведите высказывания *A* и *B* на естественный язык.
 - б)** Является ли высказывание *A* истинным при этой интерпретации?
 - в)** Является ли высказывание *B* истинным при этой интерпретации?
 - г)** Является ли *B* логическим следствием *A*?
 - д)** Является ли *A* логическим следствием *B*?
 - е)** С использованием правила резолюции попытайтесь доказать, что *A* следует из *B*. Сделайте эту попытку, даже если вы считаете, что *A* не следует логически из *B*; продолжайте свои усилия до тех пор, пока доказательство не оборвется и вы не сможете продолжать дальше (поскольку оно оборвалось). Покажите унифицирующую подстановку для каждого этапа резолюции. Если доказательство окончилось неудачей, точно объясните, где, как и почему оно оборвалось.
 - ж)** *A* теперь попытайтесь доказать, что *B* следует из *A*.
- 9.20.** Метод резолюции способен вырабатывать неконструктивные доказательства для запросов с переменными, поэтому приходится вводить специальные механизмы для извлечения только определенных ответов. Объясните, почему такая проблема не возникает при использовании баз знаний, содержащих только определенные выражения.
- 9.21.** В этой главе было указано, что метод резолюции не может использоваться для формирования всех логических следствий из некоторого множества высказываний. Позволяет ли какой-то другой алгоритм решить эту задачу?