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:

$\forall x\ King(x) \wedge Greedy(x) \Rightarrow Evil(x)$

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

$King(John) \wedge Greedy(John) \Rightarrow Evil(John)$
$King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)$
$King(Father(John)) \wedge Greedy(Father(John)) \Rightarrow Evil(Father(John))$
...

Согласно правилу **конкретизации высказывания с квантором всеобщности** (сокращенно UI — Universal Instantiation), мы можем вывести логическим путем любое высказывание, полученное в результате подстановки **базового терма** (терма без переменных) вместо переменной, на которую распространяется квантор всеобщности[1]. Чтобы записать это правило логического вывода формально, воспользуемся понятием **подстановки**, введенным в разделе 8.3. Допустим, что $Subst(\theta, \alpha)$ обозначает результат применения подстановки θ к высказыванию α. В таком случае данное правило для любой переменной v и базового терма g можно записать следующим образом:

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

[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 **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:

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

Например, из высказывания

$$\exists x\ Crown(x) \wedge OnHead(x,John)$$

можно вывести высказывание

$$Crown(C_1) \wedge OnHead(C_1,John)$$

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 **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 *Kill*(*Murderer*,*Victim*), становится больше не нужным высказывание $\exists x\ Kill(x,Victim)$. Строго говоря, новая база знаний логически не эквивалентна старой, но можно показать, что она **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:

$$\forall x\ King(x) \wedge Greedy(x) \Rightarrow Evil(x)$$
$$King(John)$$
$$Greedy(John)$$
$$Brother(Richard, John) \qquad (9.1)$$

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 $\{x/John\}$ и $\{x/Richard\}$. Мы получим следующие высказывания:

$$King(John) \wedge Greedy(John) \Rightarrow Evil(John)$$
$$King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)$$

и отбросим высказывание с квантором всеобщности. Теперь база знаний становится по сути пропозициональной, если базовые атомарные высказывания (*King*(*John*), *Greedy*(*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 *Evil*(*John*).

Как показано в разделе 9.5, такой метод **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 *Father*(*Father*(*Father*(*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 (*Father*(*Richard*) и *Father*(*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 $King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)$. И действительно, для любого человека вывод факта $Evil(John)$ из следующих высказываний кажется вполне очевидным:

$$\forall x\ King(x) \wedge Greedy(x) \Rightarrow Evil(x)$$
$$King(John)$$
$$Greedy(John)$$

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:

$$\forall y\ Greedy(y) \qquad (9.2)$$

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 **обобщенным правилом отделения** (Generalized Modus Ponens): для атомарных высказываний p_i, p_i' и q, если существует подстановка θ, такая, что $\text{Subst}(\theta, p_i') = \text{Subst}(\theta, p_i)$, то для всех i имеет место следующее:

$$\frac{p_1',\ p_2',\ \ldots,\ p_n',\ (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{\text{Subst}(\theta, q)}$$

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:

p_1' — это $King(John)$ $\quad$ p_1 — это $King(x)$
p_2' — это $Greedy(y)$ $\quad$ p_2 — это $Greedy(x)$
θ — это $\{x/John, y/John\}$ $\quad$ q — это $Evil(x)$
$\text{Subst}(\theta, q)$ — это $Evil(John)$

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:

$$p \models \text{Subst}(\theta, p)$$

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 $p_1', \ldots, p_n'$ можно вывести следующее:

$$\text{Subst}(\theta, p_1') \wedge \ldots \wedge \text{Subst}(\theta, p_n')$$

а из импликации $p_1 \wedge \ldots \wedge p_n \Rightarrow q$ — следующее:

$$\text{Subst}(\theta, p_1) \wedge \ldots \wedge \text{Subst}(\theta, p_n) \Rightarrow \text{Subst}(\theta, q)$$

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

Как принято выражаться в логике, обобщенное правило отделения представляет собой **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 **унификацией** и является ключевым компонентом любых алгоритмов вывода в логике первого порядка. Алгоритм `Unify` принимает на входе два высказывания и возвращает для них **унификатор**, если таковой существует:

```
Unify(p,q) = θ где Subst(θ,p) = Subst(θ,q)
```

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

```
Unify(Knows(John,x), Knows(John,Jane)) = {x/Jane}
Unify(Knows(John,x), Knows(y,Bill)) = {x/Bill,y/John}
Unify(Knows(John,x), Knows(y,Mother(y))) = {y/John,x/Mother(John)}
Unify(Knows(John,x), Knows(x,Elizabeth)) = fail
```

Последняя попытка унификации оканчивается неудачей (*fail*), поскольку переменная *x* не может одновременно принимать значения *John* и *Elizabeth*. Теперь вспомним, что высказывание *Knows*(*x*,*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, **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 *Knows*(*x*,*Elizabeth*) можно переименовать в z_{17} (новое имя переменной), не меняя смысл этого высказывания. После этого унификация выполняется успешно:

```
Unify(Knows(John,x), Knows(z17,Elizabeth)) = {x/Elizabeth,z17/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 возвратить `{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** (occur 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, θ) returns подстановка, позволяющая сделать
         x и y идентичными
    inputs: x, переменная, константа, список или составной терм
            y, переменная, константа, список или составной терм
            θ, подстановка, подготовленная до сих пор (необязательный
               параметр, по умолчанию применяется пустой терм)

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

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

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

    if {var/val} ∈ θ then return Unify(val, x, θ)
    else if {x/val} ∈ θ then return Unify(var, val, θ)
    else if Occur-Check?(var, x) then return failure
    else return добавление {var/x} к подстановке θ
```

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, ✍ **индексируя** факты в базе знаний. Самая простая схема, называемая ✍ **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-таблице[2].

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 — эта структура данных для хранения и выборки информации, индексируемой с помощью фиксированных ключей. С точки зрения практики хэш-таблица может рассматриваться как имеющая постоянные временны*е* 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.

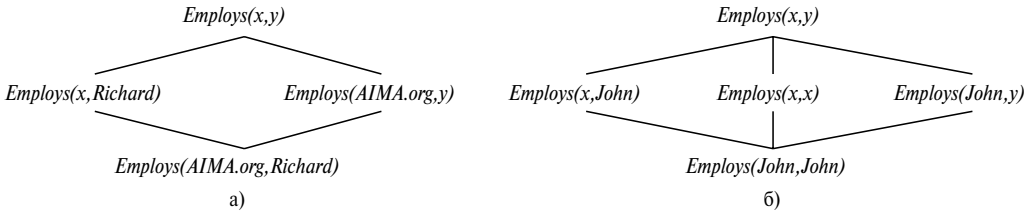


Рис. 9.1. Примеры решеток обобщения: решетка обобщения, самым нижним узлом которой является высказывание Employs(AIMA.org,Richard) *(а); решетка обобщения для высказывания* Employs(John,John) *(б)*

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.

$King(x) \wedge Greedy(x) \Rightarrow Evil(x)$
$King(John)$
$Greedy(y)$

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":

  $$American(x) \wedge Weapon(y) \wedge Sells(x,y,z) \wedge Hostile(z) \Rightarrow Criminal(x) \quad (9.3)$$

- "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":

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

  Нам необходимо также знать, что ракеты — оружие:

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

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

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

- "…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 ✍ **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 ✍ **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, α) returns подстановка или значение false
    inputs: KB, база знаний - множество определенных выражений
                первого порядка
            α, запрос - атомарное высказывание
    local variables: new, новые высказывания, выводимые
                          в каждой итерации

    repeat until множество new не пусто
          new ← {}
          for each высказывание r in KB do
                (p1 ∧ ... ∧ pn ♥ q) ← Standardize-Apart(r)
                for each подстановка θ, такая что Subst(θ,p1∧...∧pn) =
                          Subst(θ,p1'∧...∧pn') для некоторых p1',...,pn'
                          в базе знаний KB
                      q' ← Subst(θ, q)
                      if выражение q' не является переименованием
                         некоторого высказывания, которое уже находится
                         в KB, или рассматривается как элемент множества
                         new then do
                            добавить q' к множеству new
                            φ ← Unify(q', α)
                            if значение φ не представляет собой fail
                                 then return φ
          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.

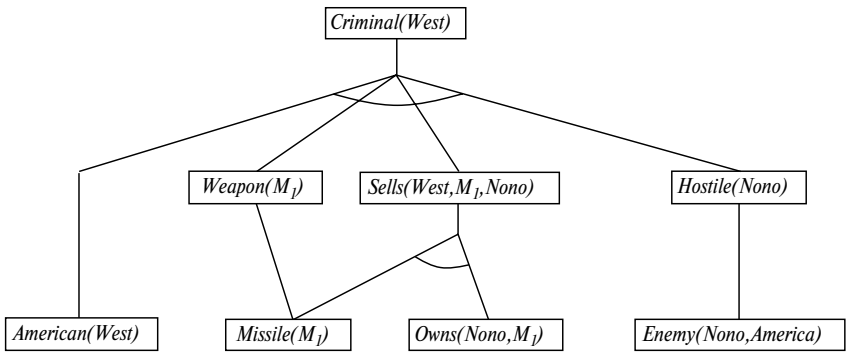


Рис. 9.2. Дерево доказательства, сформированное путем прямого логического вывода в примере доказательства преступления. Первоначальные факты показаны на нижнем уровне, факты, выведенные логическим путем в первой итерации, — на среднем уровне, а факт, логически выведенный во второй итерации, — на верхнем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

Свойства алгоритма FOL-FC-Ask проанализировать несложно. Во-первых, он является **непротиворечивым**, поскольку каждый этап логического вывода представляет собой применение обобщенного правила отделения, 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 q, 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 удается нормально завершить работу данного алгоритма. Например, если база знаний включает аксиомы Пеано:

$NatNum(0)$

$\forall n\ NatNum(n) \Rightarrow NatNum(S(n))$

то в результате прямого логического вывода будут добавлены факты $NatNum(S(0))$, $NatNum(S(S(0)))$, $NatNum(S(S(S(0))))$ 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 полуразрешимой.

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:

$$Missile(x) \Rightarrow Weapon(x)$$

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

$$Missile(x) \wedge Owns(Nono,x) \Rightarrow Sells(West,x,Nono)$$

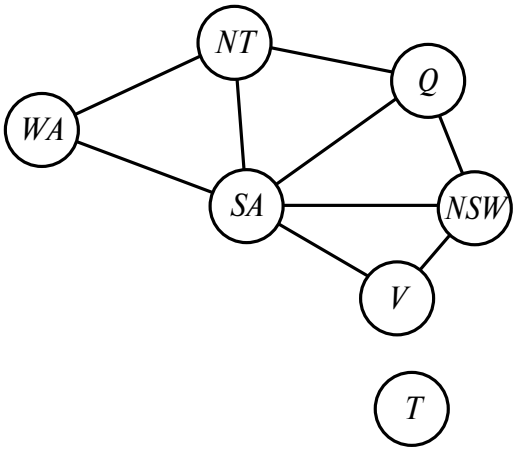
Найти все объекты, принадлежащие государству Ноуноу, опять-таки можно за постоянное время в расчете на каждый объект; затем мы можем применить к каждому объекту проверку, является ли он ракетой. Но если в базе знаний содержится много сведений об объектах, принадлежащих государству Ноуноу, и лишь немного 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, $Missile(x)$ — это унарное ограничение на x. Развивая эту идею, можно прийти к выводу, что ☞ *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 $Colorable()$ 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 ☞ *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 ✍ **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)

Diff (wa, nt) ∧ *Diff (wa, sa)* ∧
Diff (nt, q) ∧ *Diff (nt, sa)* ∧
Diff (q, nsw) ∧ *Diff (q, sa)* ∧
Diff (nsw, v) ∧ *Diff(nsw, sa)* ∧
Diff (v, sa) ⇒ *Colorable()*

Diff (Red, Blue) *Diff (Red, Green)*
Diff (Green, Red) *Diff (Green, Blue)*
Diff (Blue, Red) *Diff (Blue, Green)*

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) ∧ Diff(nt,q) ∧ Diff(q,nsw) ∧ Diff(nsw,v) ♥ 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

$Missile(x) \Rightarrow Weapon(x)$

согласуется с фактом $Missile(M_1)$ (еще раз), и, безусловно, при этом ничего не происходит, поскольку заключение $Weapon(M_1)$ уже известно. Таких излишних согласований с правилами можно избежать, сделав следующее наблюдение: ☞ *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:

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

и фактом $American(West)$. 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 ✍ **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 удается 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 **продукционных систем**, которые принадлежат к числу самых первых систем прямого логического вывода, получивших широкое распространение[4]. 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 **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 продукций 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 вырабатывал 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 **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:

$$Magic(x) \land American(x) \land Weapon(y) \land Sells(x,y,z) \land Hostile(z) \Rightarrow Criminal(x)$$

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