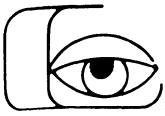


ПРОБЛЕМЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

Р. КОВАЛЬСКИ

ЛОГИКА В РЕШЕНИИ ПРОБЛЕМ

Перевод с английского
Л.Г. ОСМОЛОВСКОГО, В.Э. ВОЛЬФЕНГАГЕНА, В.Я. ЯЦУКА

С предисловием и примечаниями
Д.А. ПОСПЕЛОВА

МОСКВА "НАУКА"
ГЛАВНАЯ РЕДАКЦИЯ
ФИЗИКО-МАТЕМАТИЧЕСКОЙ ЛИТЕРАТУРЫ
1990

ББК 22.18
К56
УДК 519.76

LOGIC
FOR PROBLEM
SOLVING
ROBERT KOWALSKI
Imperial College of Science and Technology
University of London

Ковальски Р. **Логика в решении проблем**: Пер. с англ. – М.: Наука. Гл. ред. физ.-мат. лит., 1990. – (Пробл. искусств. интеллекта.) – 280 с. – ISBN 5–02–014148–8.

Изложены основы так называемой клаузальной логики, являющейся средством логического программирования. Последнее лежит в основе проектирования ЭВМ пятого поколения, над которыми в настоящее время работают специалисты ряда стран.

Для научных работников и инженеров, занимающихся проблемами искусственного интеллекта. Полезна аспирантам и студентам втузов.

Табл. 2. Ил. 146. Библиогр. 107 назв.

Рецензент академик *Г.С. Поспелов*

К $\frac{1402070000-118}{053(02)-90}$ 160-90

ISBN 5-02-014148-8

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ Д.А. ПОСПЕЛОВА

В мире существуют тысячи различных языков программирования. Подавляющее большинство известны лишь их создателям и никогда не применялись широкой армией пользователей. Законы выживаемости языков программирования пока известны не до конца, хотя многое в этом вопросе уже ясно. Например, никакой язык программирования не сможет выжить и распространиться (особенно сейчас), если он не обладает весьма большим набором вспомогательных средств, составляющих для него дружественную программную среду, куда входят сервисные программы, редакторы, трансляторы и многое другое. Язык без среды обречен на вымирание, как только исчезнет энтузиазм его создателей.

Поэтому появление в наше время нового жизнеспособного языка программирования – событие редкое. Нужен какой-то мощный стимул, чтобы новичек вошел в ряды "джентльменского набора" языков, признанных мировым сообществом программистов.

Среди пионерских исследований, связанных с решением нечисловых задач на вычислительных машинах, появившихся вскоре после появления самих вычислительных машин, были исследования по машинному переводу. К чести лингвистов надо сказать, что из всех весьма далеких от точных наук специалистов они первыми сумели оценить те возможности, которые несли в себе новые технические устройства. Попытки использовать алгоритмический подход к созданию программ для перевода предложений с одного языка на другой, а позже для автоматического реферирования текстов, фактографического поиска информации и синтеза естественно-языковых текстов выдвинули перед специалистами по программированию ряд новых задач. Обработка текстов во многом оказалась не такой, как работа с числовыми данными. Например, вся система базовых операций, из которых строятся процедуры работы с текстом, не совпадала с традиционным набором базовых операторов, оказавшимся удобным (и универсальным) для программирования вычислений. Поэтому программы обработки текстов, которые, в принципе, можно было свести к последовательности традиционных для вычислений базовых операторов, получались все же неуклюжими, плохо интерпретируемыми и неэффективными.

И поэтому с самого начала проникновения лингвистов, а вслед за ними логиков и других специалистов, которые видели в вычислительных машинах средство для автоматизации символьных преобразований, характер-

ных для их областей науки, не покидала идея о создании языков и систем программирования, ориентированных именно на такие задачи.

Недостатка в предложениях такого рода не было. Но на первом этапе, когда всякий язык программирования мыслился как алгоритмический (в том духе понимания этого термина, как оно было сформулировано создателями языка Алгол), все эти предложения были всего лишь паллиативами, не способными привести к созданию того, что нужно.

Постепенно становилось ясно, что в основе эффективных языков для автоматизации процесса символьной обработки должна лежать идея не алгоритма, а исчисления со свободным поиском в нем доказательства целевого утверждения, однозначно связанного с решением интересующей пользователя задачи.

И первый заметный шаг в этом направлении был сделан специалистами, работавшими с естественно-языковыми текстами. Это произошло в Марселе в начале семидесятых годов. Предложенный в этой группе язык был скромно назван *Q*-языком. Первые публикации о *Q*-языке сначала не привлекли к нему особого внимания специалистов. Он казался весьма специализированным языком, ориентированным, в основном, на описание процедур формирования ответов на вопросы, поступающие к базам данных, когда эти вопросы сформулированы на естественном языке.

Но повторилась история, часто встречающаяся в нашей жизни. Хорошим аналогом ее может служить возникновение мирового христианства. Учение, возникшее в Иудее и представлявшееся всему цивилизованному древнему миру какой-то очередной ересью внутри традиционной еврейской религии, наверное, так и влачило бы свое мало заметное для земного шара существование, если бы не Павел. Именно Павел усмотрел в учении Христа то зерно, которое помогло создать ему религию, охватившую огромные пространства и народы. Это исходное зерно (свод нравственно-этических норм и догматов) Павел во многом изменил и скорректировал, но сохранил то, что позволило христианству занять в мире достойное место.

Нечто подобное произошло и с *Q*-языком. Он легко бы канул в вечность, но нашелся человек, который усмотрел в нем то, что уже давно зрело в его понимании сути процесса решения задач, которые возникали при использовании вычислительных машин в автоматическом доказательстве теорем в логике. Этим человеком оказался Роберт Ковальски, книгу которого вы начали читать.

Работая в Имперском колледже в Лондоне, Ковальски читал студентам курсы, связанные с прикладной логикой и использованием вычислительных машин при решении логических задач (в частности, для доказательства теорем). Это естественным образом привело его к идее истолкования процесса решения задачи, как поиска доказательства соответствующего ей утверждения в логических исчислениях (как правило, в исчислении предикатов первого порядка). Он понимал необходимость создания нового класса языков программирования (потом они будут названы языками логического программирования), специально ориентированных на решение задач символьной обработки текстов (не только естественно-языковых, а любых символьных). Знакомство с *Q*-языком показало, что путь построения языков такого типа может быть результативным. Отталкиваясь от *Q*-языка, Р. Ковальски вызвал к жизни язык Пролог, ставший наряду с

Фортраном, Лиспом или Паскалем очередным мировым языком программирования, прародителем целого семейства языков логического программирования.

Итак, перед вами книга основоположника нового направления в программировании, в которой содержатся все основные предпосылки и идеи, положенные в основу теории таких языков и практики их использования при решении разнообразных задач символьной обработки.

Книга, как отмечает и сам ее автор, является изложением того курса лекций, которые он много лет читал в Имперском колледже и других учебных заведениях Великобритании и других стран. Именно это свойство книги, ее направленность на обучение, является, по моему мнению, особенно ценным. В нашей стране еще очень мало книг, которые могли бы выполнить роль учебника для специалистов, готовящихся овладеть логическими языками программирования и теорией таких языков. Книга Ковальски будет для них верным путеводителем и помощником на этом пути.

Несколько замечаний по поводу терминологии, встречающейся на страницах этой книги. К сожалению, в нашей стране для ряда понятий, связанных с программированием и прикладной логикой, нет однозначного перевода терминов, используемых в англоязычной литературе.

Прежде всего это касается основного для данной книги понятия "clause". На русский язык оно переводится либо как "калька", "клауза", либо как "дизъюнкт". Переводчики предпочли первый перевод. Тут они были не оригинальны. Аналогичное решение было принято при переводе на русский язык известной книги Х.Б. Карри "Основания математической логики", изданной в нашей стране в 1969 г. в издательстве "Мир". Поясняя свою позицию, переводчики данной книги мотивировали свое решение тем, что термин "дизъюнкт" плох хотя бы потому, что дизъюнкт не обязательно универсально замкнут. Другие возможные способы перевода термина "clause" ("предложение" или "клоз") были также отвергнуты. Термин "предложение" слишком перегружен другими смыслами (как и термин "утверждение"), а галлицизм "клоз" явно проигрывает по сравнению с привычным для отечественной научной терминологии латинизмом.

Кроме того, в книге везде (за исключением заглавия) известное словосочетание "problem solving" переводится, как "поиск решения", что определяется той спецификой его употребления, которая характерна для данного текста.

Роберт Ковальски проявил большое внимание к переводу своей книги на русский язык и оказал переводчикам неоценимую помощь. Переводчики и редактор книги приносят ему за это свою искреннюю и глубокую благодарность.

В заключение необходимо отметить, что в последние годы на русском языке вышел ряд отечественных и переводных книг, посвященных как математическим, так и чисто программистским аспектам логики в решении проблем. Можно смело считать, что почти все эти работы в той или иной мере развивают идеи, впервые увидевшие свет на страницах настоящей книги. Для удобства читателя ниже приведен не претендующий на полноту перечень таких работ.

1. Грей П. Логика, алгебра и базы данных: Пер. с англ. Х.И. Килова, Г.Е. Минца/ Под ред. Г.В. Орловского, А.О. Слисенко. – М.: Машиностроение. – 1989. – 368 с.
2. Кларк К., Маккейб Ф. Введение в логическое программирование на микро–Прологе: Пер. с англ. А.И. Горлина / Под ред. В.В. Мартынюка. – М.: Радио и связь. – 1987. – 312 с.
3. Клоксин У., Меллиш К. Программирование на языке Пролог: Пер. с англ. А.В. Горбунова, М.М. Комарова / Под ред. А.К. Платонова, Ю.М. Лазутина. – М.: Мир. – 1987. – 336 с.
4. Логическое программирование. Сборник статей: Пер. с англ. и фр. / Под ред. В.Н. Агафонова. – М.: Мир. – 1988. – 368 с.
5. Маслов С.Ю. Теория дедуктивных систем и ее применения. – М.: Радио и связь. – 1986. – 136 с.
6. Мейер Д. Теория реляционных баз данных: Пер. с англ. М.К. Валиева, С.М. Емельяновой и И.С. Захаревича / Под ред. М.Ш. Цаленко. – М.: Мир. – 1987. – 608 с.
7. Стерлинг Л., Шапиро Э. Искусство программирования на языке Пролог: Пер. с англ. С.Ф. Сопрунова и Л.В. Шабанова / Под ред. Ю.Г. Дадаева. – М.: Мир. – 1990.– 235 с.
8. Хоггер К. Введение в логическое программирование: Пер. с англ. М.В. Захарьящева / Под ред. Ю.И. Янова. – М.: Мир. – 1988. – 348 с.
9. ЭВМ пятого поколения: Концепции, проблемы, перспективы / Под ред. Т. Мото–ока: Пер. с англ. О.М. Вейнерова / Под ред. А.А. Рывкина, В.М. Савинкова. – М.: Финансы и статистика. – 1984. – 110 с.
10. Язык Пролог в пятом поколении ЭВМ: Пер. с англ. / Сост. Н.И. Ильинский. – М.: Мир. – 1988. – 501 с.

Д.А. Поспелов

ПРЕДИСЛОВИЕ АВТОРА К РУССКОМУ ИЗДАНИЮ

Те десять лет, что истекли с момента, когда англоязычное издание этой книги впервые увидело свет, отмечены значительным прогрессом в теории и практике логических методов поиска решений. Широкую известность и использование получил язык программирования Пролог. Начали развиваться языки параллельного логического программирования и языки логического программирования ограничений целостности. Появились дедуктивные базы данных, которые можно рассматривать как естественные преемники реляционных баз данных. Логиками и специалистами по компьютерной информатике проводились разнообразные исследования в области логического программирования и интерпретации отрицания как неудачи. Японский научный комитет по ЭВМ пятого поколения в Токио, Европейский центр исследований по компьютерной информатике в Мюнхене и Шведский институт компьютерной информатики в Стокгольме положили логическое программирование в основу своих разработок.

И все же я уверен, что, хотя много событий произошло в мире с тех пор как книга была впервые опубликована, она по-прежнему созвучна нашему времени. То особое значение, которое придавалось логике клауз Хорна в главах 3–5, да и повсюду в книге вполне соотносится с нынешним пониманием важности клауз Хорна как основы и для логического программирования, и, в общем случае, для представления знаний. Описание и сопоставление клаузальной и стандартной форм логики в главах 1, 7, 8 и 10 можно считать подходящей основой для развития логики клауз Хорна как таковой. Процедура доказательства методом графа соединений, описанная, в частности, в главе 8, в последнее время послужила предметом серьезных исследований в Стэнфордском исследовательском институте и в Кайзерслаутерн. И остальные темы, затронутые в книге (среди них использование ситуационного исчисления для задач построения плана в главе 6, глобальные стратегии поиска решений в главе 9, использование метауровневых способов рассуждения в главе 12) продолжают оставаться предметом пристального изучения. Например, метауровневые рассуждения оказались весьма полезным инструментом для построения оболочек экспертных систем.

Если бы мне представился случай написать книгу снова, то я бы придал большее значение тематике трех последних глав. Я ввел бы отрицание по неудаче много раньше главы 11 и показал бы его важность как

в логическом программировании, так и в рассуждениях по умолчанию в теории искусственного интеллекта. Я бы обратил большее внимание на изложение тех намеченных в главе 13 вопросов, где логика сочетается с изменчивостью, и в особенности, на важность абдукции для таких разнообразных и разнородных приложений, как диагностика неисправностей, понимание естественного языка и рассуждения по умолчанию. И, наконец, я отвел бы бóльшую роль изложению затронутой в главе 2 теме представления знаний, распространив область исследования и на временны́е факторы, и на соотношение между логикой, фреймами и объектами и подчеркнул бы важность как процедурного, так и декларативного представления знаний.

Однако гораздо более важным я посчитал бы возможность убедить читателя в том, что последние достижения в области логической технологии могут с успехом использоваться и компьютерами и людьми. Эти достижения, конечно, способны сделать компьютерные языки более мощными и более дружественными по отношению к человеку, но не только. Их могут использовать сами люди для того, чтобы яснее и четче понимать своих соседей по планете. И, наконец, я надеюсь, что и в своем существующем виде эта книга внесет хотя бы небольшой вклад в достижение этой заманчивой цели.

Роберт А. Ковальски

ПРЕДИСЛОВИЕ

Эта книга посвящена исследованию приложений логики к методам поиска решений и к программированию. Она не предполагает никакой предварительной подготовки в этой области, и поэтому ее можно рассматривать как введение в логику, теорию поиска решений и программирование.

Логика

Логика — это важное инструментальное средство, применяемое при анализе методов рассуждений. Логика интересуется прежде всего тем, можно ли вывести заключения из некоторых посылок, не учитывая при этом истинность или ложность этих посылок и заключений как таковых. В нашей книге делается попытка приложить эти традиционные логические методы к теоретическим аспектам таких современных прикладных научных направлений, как поиск решений и программирование.

Если взглянуть на нашу книгу как на введение в логику, то ее главным отличием от остальных руководств является то, что она основана на клаузальной форме логики. Такое построение книги обеспечивает некоторые преимущества. Клаузальная форма логики проще, чем стандартная форма, но обладает такой же выразительной силой. Она настолько проста, что ее можно изучать сразу, без обычного предварительного изучения пропозициональной логики; она также имеет бóльшее, чем стандартная форма логики, сходство с формальными методами, которые используются в обработке данных и в программировании.

Книга не затрагивает теоретическую сторону математической логики, а касается лишь ее приложений. За более интересными и полными сведениями о связях между логикой и языками читателю следует обратиться к книгам Квайна и Ходжеса [Quine 1941], [Hodges 1977].

Поиск решений

Клаузальная форма логики годится для выяснения сути моделей поиска решений, появившихся в областях когнитивной психологии и искусственного интеллекта. В настоящей книге рассматриваются модели поиска решений для эвристического поиска, редукции задач и программного исполнения и высказываются доводы в пользу того, что средства логического вывода обеспечивают построение более простых и более выразительных моделей.

Интерпретация логического вывода как метода поиска решений исходит из различия между *восходящими рассуждениями,* проводящимися по направлению от посылок к заключениям, и *нисходящими рассуждениями,* проводящимися в обратном направлении, от целей к подцелям. Интерпретация логического вывода как метода поиска решений есть прежде всего нисходящая интерпретация. При восходящем выводе решения считаются уже полученными и обоснованными, в то время как на пути нисходящего вывода решения большей частью порождаются. Восходящий вывод является *синтезом* новой информации на основе старой; нисходящей же вывод выступает как сведение *анализа* целей к анализу подцелей.

Наша книга описывает основания поиска решений почти так же, как это сделано в книге Нильсона [Nilsson 1971], Уинстона [Winston 1977] и Банди [Bundy 1978]. Но в то время как упомянутые книги используют в качестве общего метода формализации системы продукций, Лисп или Лого, наша книга последовательно применяет клаузальную форму логики.

Программирование

Если взглянуть на использование логики для общения с компьютером, то легко обнаружить, что она является более высокоуровневым и более человекоориентированным формальным средством, чем другие средства, специально созданные для компьютеров. В противовес обычной компьютерной методологии, использующей разные формальные методы для написания программ, выражения спецификаций, построения баз данных и запросных систем, установления ограничений целостности, логика обеспечивает единую форму языка для всех этих задач. Мы коснемся использования логики в базах данных, но сконцентрируем внимание на ее использовании в качестве языка программирования.

Смысл программ, написанных на обычных языках программирования, может быть определен в рамках тех процессов, которые они вызывают в компьютере как в аппарате. Смысл программ, выраженных средствами логики, может быть определен в машинно-независимых, человекоориентированных понятиях. Поэтому-то логические программы проще создавать, проще понимать, проще совершенствовать и проще адаптировать для других целей.

Те же самые методы нисходящего вывода, благодаря которым логика приобретает интерпретацию поиска решений, можно использовать для эффективного исполнения программы на компьютере. Нисходящий вывод согласует поиск решений и программирование. Вдобавок он обеспечивает такие возможности для исполнения интеллектуальных программ как недетерминизм, параллелизм, вызов процедур по образцам, которые еще недостаточно развиты в обычных языках программирования. В качестве эффективного языка программирования, применяющегося в приложениях искусственного интеллекта, базах данных и проектировании, можно указать базирующийся на клаузальной форме логики язык Пролог [Colmerauer 1972], [Roussel 1975], [Bruynooghe 1976], [Warren, Pereira, Pereira 1977], [Clark, McCabe 1979].

Автоматическое доказательство теорем

Использование клаузальной формы логики и связанных с ней систем вывода основывается на достижениях в автоматическом доказательстве теорем при помощи компьютеров. Главными предтечами систем вывода, представленных в этой книге, явились правило резолюции Робинсона [Robinson 1965a] и процедура доказательства методом исключения Лавлэнда [Loveland 1968, 1969], которые, в свою очередь, базируются на более ранних исследованиях Эрбрана [Herbrand 1930] и Правица [Pravitz 1960].

Хотя методы вывода в этой книге строятся в предположении их последующего использования на компьютере, нет никаких препятствий в использовании их людьми. Ведь стратегии поиска решений, развитые для реализации эффективных методов автоматического доказательства теорем, похожи на стратегии, вызванные к жизни исследованиями по компьютерному моделированию процессов поиска решений человеком. В частности, мы попытались представить такую точку зрения на логику, которая согласовывала бы машинноориентированную сторону метода резолюций с эвристическими процедурами доказательства Бледшоу и его коллег [Bledsoe 1979].

Эта книга может рассматриваться и как пособие по автоматическому доказательству теорем в ряду таких книг, как работы Ченя и Ли [Chang, Lee 1973], Лавлэнда [Loveland 1978] и Робинсона [Robinson 1979]. Однако наша книга менее формализована и в ней не предпринимаются попытки широкого охвата все адекватной области знания.

Структура книги

Книга может быть условно разбита на три части. Первая часть, включающая главы 1 и 2, посвящена машинно-независимой семантике клаузальной формы логики и использованию клаузальной формы для представления информации; вторая часть, включающая главы 3–8, содержит сведения о системах вывода для клаузальной формы логики; третья часть, включающая главы 9–13, предлагает расширения клаузальной формы и некоторые более сильные методы поиска решений.

В первой части книги подчеркивается, что логику, в отличие от большинства других формализмов, можно понимать, не вдаваясь в конкретные подробности. Приводятся примеры использования логики при описании программ и баз данных, а клаузальная форма соотносится с семантическими сетями в аспекте представления смысла предложений естественного языка.

Во второй части книги в порядке возрастания сложности излагаются методы вывода для клаузальной формы. В частности, главы 3–6 описывают методы вывода для клауз Хорна, которые представляют собой упрощенный вид предложений и имеют вид

$$A, \text{ если } B_1 \text{ и } B_2 \text{ и } \ldots \text{ и } B_m.$$

В главе 3 рассмотрены нисходящий и восходящий выводы, являющиеся обобщениями нисходящих и восходящих процедур синтаксического разбора для контекстно-свободных грамматик. Глава 4 интерпретирует нисходящий вывод как поиск решений, а глава 5 связана с его програм-

мной интерпретацией. Глава 6 описывает приложение логики клауз Хорна к проблемам построения планов. Методы вывода в языке нехорновских клауз и их интерпретация поиском решений составляют содержание глав 7 и 8.

Глава 9 содержит описание глобальных методов поиска решений для клаузальной формы логики, а остальные главы затрагивают уже различные расширения клаузальной формы. Стандартная форма логики и ее связь с клаузальной логикой – предмет главы 10. Определениям вида если-и-только-если посвящена глава 11. В главе 12 мы рассматриваем расширение логики, которое сочетает в себе использование и смысл предложений подобно тому, как это делается в естественном языке. Последняя глава посвящена динамике изменения информационных систем; в ней обращено особое внимание на роль противоречия в задании направленности изменения. В этом просматривается сочетание интерпретации логики как метода поиска решений с классическим ее использованием в анализе фактов и системы человеческих знаний.

О читательском назначении книги

Эта книга представляет собой расширенное изложение конспекта, подготовленного в марте 1974 г. [Kowalski 1974] для лекций по основам компьютерной информатики, прочитанных в Математическом центре в Амстердаме. Краткие курсы лекций на основе этого же материала были прочитаны автором в Эдинбурге, Милане, Риме и Стокгольме в период между 1973 и 1975 гг. С 1975 г. части этой книги использовались для вводных курсов по логики и поиску решений, прочитанных студентам Имперского Колледжа. Полный курс лекций, охватывающий весь материал книги, был прочитан в Университете Сиракуз в 1978 г.

Книга написана на неформальном уровне и почти не содержит доказательств. Она не предполагает никакой предварительной подготовки по математической логике, методам поиска решений или компьютерной информатике и поэтому доступна студентам первого курса. Правда, некоторые упражнения рассчитаны на более подготовленного читателя. Кроме того, отдельные моменты в главе 5, связанные со сравнением логики и обычных языков программирования, могут оказаться непонятными читателям, неискушенным в программировании.

Благодарности

Большая часть этой книги была написана под влиянием работ моих коллег Кейта Кларка, Алэна Колмероэ, Пата Хейеса, Маартена ван Эмдена, Дэвида Уоррена. Я благодарю их, а также Фрэнка Брауна, Алэна Банди, Тони Хоара, Уильфреда Ходжеса, Криса Хоггера, Яана Нильссона, Джорджа Полларда, Рэя Рейтера, Ричарда Уолдингера и Джорджа Уинтерстейна за ценные замечания, которые они сделали к ранним версиям этой книги; благодарю также Карен Кинг, Фрэнка Маккейба, Кевина Митчелла и Криса Мосса за помощь в полиграфической подготовке книги. Я счастлив выразить мою признательность за поддержку Научно-исследовательскому совету.

Но особенно обязан я вдохновляющему терпению моей жены Дануси и моих дочерей Тани, Дани и Янины.

Логика изучает взаимосвязь между посылками и заключениями. Она утверждает, в частности, что из посылок

Бобу нравится логика

и

Бобу нравится любой, кому нравится логика

следует заключение

Боб нравится сам себе

но не следует заключение

Бобу нравятся только те люди, которым нравится логика

Логика занимается не истинностью, ложностью или допустимостью конкретных высказываний, но связями между этими понятиями. Если заключение выведено из истинной или другой допустимой посылки, то логика приведет нас к необходимости согласиться с этим заключением. Но если из данных посылок выводится недопустимое или ложное заключение, то логика советует нам отказаться по крайней мере от одной из посылок. Так, если я отбрасываю заключение о том, что Боб нравится сам себе, то логика вынуждает меня отказаться либо от посылки, что Бобу нравится логика, либо от посылки, что Бобу нравится любой, кому нравится логика.

Если нужно проверить, что из посылок следует заключение, то полезно построить доказательство, состоящее из шагов вывода. Для того, чтобы доказательство было убедительным, отдельные шаги вывода обязаны быть ясными и очевидными и должны корректно сочетаться друг с другом. В этих условиях необходимо, чтобы высказывания не были двусмысленными, и важно, чтобы грамматика высказываний была как можно более простой. Требование недвусмысленности и грамматической простоты языка доказательств и мотивирует применение символического, а не естественного языка, такого, например, как английский.

Символический язык клаузальной формы логики, используемый в первых девяти главах этой книги чрезвычайно прост. Простейшими высказываниями в нем являются *атомарные высказывания*, которые именуют

связи между индивидами:

Бобу *нравится* логика
Джону *нравится* Мери
Джон *на два года старше* Мери

(выделенные слова суть части имен связей; таким образом, неподчеркнутыми остались имена индивидов). Более общие высказывания выражают тот факт, что из атомарных условий следуют атомарные заключения:

Мери *нравится* Джон, если Джону *нравится* Мери
Бобу *нравится* x, если x *нравится* логика

Здесь x есть переменная, которая применена для обозначения любого индивида. Высказывания могут иметь несколько совместных посылок или несколько альтернативных заключений:

Мери *нравится* Джон или Мери *нравится* Боб, если
Мери *нравится* x

(т.е., Мери нравится Джон или Боб, если ей вообще кто-либо нравится)

x *нравится* Боб, если x *является студентом* Боба
и x *нравится* логика

Высказывания также называются *клаузами**). В общем случае любая клауза выражает тот факт, что некоторое количество (возможно, нуль) совместных посылок влечет некоторое количество (возможно, нуль) альтернативных заключений. Посылки и заключения выражают отношения между индивидами. Индивиды могут быть фиксированными и обозначаться словами, как например

Боб, Джон, логика или 2

называемыми (возможно, несколько путанно) *константными символами,* а могут быть и произвольными и обозначаться *переменными,* как например

u, v, w, x, y, z

Использование *функциональных символов* для построения более сложных имен типа

папаша (Джон) (т.е. Джон – папаша)
дробь (3/4) (т.е. 3/4 – дробь)

будет рассмотрено ниже.

Приведенное неформальное введение в клаузальную форму логики будет слегка переработано и развито в следующих параграфах этой главы. Но простота клаузальной формы в сравнении с естественными языками должна быть уже достаточно очевидна. При этом не может не вызывать удивления то, что клаузальная форма обладает вдобавок значительной долей выразительности естественного языка. В последних четырех главах книги мы коснемся некоторых недостатков клаузальной формы и укажем пути их преодоления.

---

*) clause – предложение, являющееся частью сложного предложения. – *Примеч. пер.*

Пример "Родственные связи" и клаузальная форма

Атомарные формулы, которые служат посылками и заключениями клауз, принято записывать в стандартной, хотя и несколько неестественной, форме. Имя отношения записывается в начале атомарной формулы, а за ним указывается последовательность индивидов, к которым это отношение применяется. Так, мы записываем Отец (Зевс, Арес) вместо того, чтобы написать, что Зевс *является отцом* Ареса, и Царица (Гармония) вместо того, чтобы написать, что Гармония *есть царица.* Здесь, строго говоря "Царица", именует свойство индивида, а не отношение между индивидами. Однако для упрощения терминологии мы включим свойства (именуемые также *предикатами*) в число отношений. Более того, окончательно смешивания терминологию, мы будем говорить об именах отношений как о *предикатных символах.*

Будем в дальнейшем использовать стрелку ← (читается "если") для обозначения импликации. Так, например, запись

Женщина (x) ← Мать (x, y)

будет означать

x есть женщина, если x – мать y

Для того чтобы упростить обозначения, а затем и правила вывода, удобно рассматривать все клаузы как импликации, даже если у них нет посылок или заключений. Поэтому мы запишем:

Отец (Зевс, Арес) ←

вместо

Отец (Зевс, Арес)

Импликации без заключений являются отрицаниями. Клауза

← Женщина (Зевс)

выражает тот факт, что Зевс не женщина.

Следующие ниже клаузы описывают некоторые характеристики (свойства) и родственные связи древнегреческих богов:

F1 Отец (Зевс, Арес) ←
F2 Мать (Гера, Арес) ←
F3 Отец (Арес, Гармония) ←
F4 Мать (Афродита, Гармония) ←
F5 Отец (Кадм, Семела) ←
F6 Мать (Гармония, Семела) ←
F7 Отец (Зевс, Дионис) ←
F8 Мать (Семела, Дионис) ←
F9 Божество (Зевс) ←
F10 Божество (Гера) ←
F11 Божество (Арес) ←
F12 Божество (Афродита) ←
F13 Царица (Гармония) ←

Интуитивный смысл этих клауз должен быть очевиден. Смысл нижеследующих клауз проясняется благодаря задаваемым ими ограничениям.

F14 Женщина (x) ← Мать (x, y)
F15 Мужчина (x) ← Отец (x, y)
F16 Родитель (x, y) ← Мать (x, y)
F17 Родитель (x, y) ← Отец (x, y)

Эти клаузы означают, что для всех x и y

x есть женщина, если x – мать y
x есть мужчина, если x – отец y
x есть родитель y, если x – мать y
x есть родитель y, если x – отец y

Переменные в разных клаузах считаются различными, даже если у них одинаковые имена. Так, переменная x в клаузе F14 никак не связана с переменной x в F15. Конкретное имя переменной существенно только в контексте той клаузы, в которой встречается эта переменная. Две клаузы, различающиеся только именами своих переменных, являются эквивалентными, и о них говорят как о *вариантах* друг друга.

В клаузальной форме считается, что все посылки связаны конъюнктивно (т.е. при помощи связки И), в то время как все заключения связаны дизъюнктивно (т.е. при помощи связки ИЛИ). Следовательно, связки И и ИЛИ могут быть без ущерба заменены запятыми. Запятые между посылками, таким образом, читаются как И, а между заключениями – как ИЛИ. Таким образом, клаузы

F18 Дед–или–бабушка (x, y) ← Родитель (x, z),
Родитель (z, y)
F19 Мужчина (x), Женщина (x) ← Человек (x)

где x, y и z суть переменные и означают для всех x, y, z, что

x – дед или бабушка, если x есть родитель z *и* z есть родитель y
x – мужчина *или* x – женщина, если x – человек

Если несколько заключений выводятся из одних и тех же посылок, то для каждого заключения нужна своя отдельная клауза. Аналогично, если одно и то же заключение выводится из альтернативных посылок, то отдельная клауза нужна для каждой посылки. Высказывание

Женщина (x) *и* Родитель (x, y) ← Мать(x, y)

котрое может быть выражено в таком же виде в стандартной форме логики (определенной в главе 10), с помощью клауз задается так:

Женщина (x) ← Мать (x, y)
Родитель (x, y) ← Мать (x, y)

Две эти клаузы неявно соединены при помощи И, т.е. x – женщина, если x – мать y *и* x – родитель y, если x – мать y. Аналогично, высказывание

Родитель (x, y) ← Мать (x, y) *или* Отец (x, y)

может быть выражено клаузами

Родитель (x, y) ← Мать (x, y)
Родитель (x, y) ← Отец (x, y)

т.е. x – родитель y, если x – мать y *или* x – родитель y, если x – отец y.

Предикатные символы могут обозначать отношения и больше, чем между двумя индивидами. Например, атомарная формула

Родители (x, y, z)

могла бы быть использована для выражения того, что x – отец z, а y – мать z, т.е.

Родители (x, y, z) ← Отец (x, z), Мать (y, z)

Более строгое определение клаузальной формы

Определим более строго *синтаксис* (грамматику) клаузальной формы и одновременно покажем связь этой формы с естественным языком.

Клауза есть выражение вида

$$B_1, \ldots, B_m \leftarrow A_1, \ldots, A_n$$

где $B_1, \ldots, B_m, A_1, \ldots, A_n$ суть атомарные формулы, $n \geqslant 0$ и $m \geqslant 0$. Атомарные формулы $A_1, \ldots, A_n$ суть совместные *посылки* клаузы, а $B_1, \ldots, B_m$ суть альтернативные *заключения*. Если клауза содержит переменные $x_1, \ldots, x_k$, то ее следует словесно интерпретировать так:

для всех $x_1, \ldots, x_k$
B_1 или ... или B_m, если A_1 и ... и A_n

Если $n = 0$, то клаузу следует словесно интерпретировать как безусловную:

для всех $x_1, \ldots, x_k$ B_1 или ... или B_m

Если $m = 0$, то клаузу следует словесно интерпретировать так:

ни при каких $x_1, \ldots, x_k$ A_1 и ... и A_n

Если $m = n = 0$, то клаузу надо записывать как □ и интерпретировать как тождественно ложное высказывание.

Атом (или *атомарная формула*) есть выражение вида

$$P(t_1, \ldots, t_m)$$

где P – m-местный предикатный символ, $t_1, \ldots, t_m$ – термы, а $m \geqslant 1$. Атом следует интерпретировать как утверждение о том, что отношение, именуемое P, истинно для индивидов, именуемых $t_1, \ldots, t_m$.

Терм есть переменная, константный символ или выражение вида

$$f(t_1, \ldots, t_m)$$

где f – m-местный функциональный символ, $t_1, \ldots, t_m$ суть термы, а $m \geqslant 1$.

Будем считать, что множества *предикатных символов, функциональных символов, константных символов и символов переменных* суть произвольные попарно не пересекающиеся множества. Зарезервируем в дальнейшем строчные буквы u, v, w, x, y, z с добавочными значками или без них для использования в качестве переменных. Другие символы будем различать в зависимости от их положения в клаузах.

Стрелка в клаузальной форме ← записывается в направлении, противоположном тому, которое обычно используется в стандартной форме логики. Используемая в дальнейшем запись B ← A (B если A) более знакома в виде A → B (если A то B). Разница здесь, однако, чисто символическая. Обозначение B ← A используется лишь для того, чтобы обратить внимание на заключение клаузы.

Различные места в предикатном или функциональном выражении называются также его *аргументами*. В атоме $P(t_1, \ldots, t_m)$ первым аргументом является t_1, а последним – t_m.

Составные термы применяются для того, чтобы описать бесконечное множество индивидов, используя только конечное множество клауз. Например, неотрицательные целые числа могут быть представлены термами

$$0, s(0), s(s(0)), \ldots, \underbrace{s(s(\ldots s(0)\ldots))}_{\text{n раз}}, \ldots$$

где 0 есть константный символ, а s – одноместный функциональный символ*). Терм s(t) обозначает число, которое на единицу больше числа, обозначенного термом t. Оно является *последователем* в последовательности целых чисел. Клаузы

Num1 Числ (0) ←
Num2 Числ (s(x)) ← Числ (x)

означают, что 0 есть число и s(x) есть число, если x есть число.

Нисходящий и восходящий варианты определений

Общее определение клаузальной формы было сформулировано по нисходящему принципу. Это означает, что первое определение поясняет целевое понятие клаузы в терминах атомарных формул (которые еще не были определены). Атомарная формула становится новым целевым понятием, которое в следующем определении сводится к двум подцелевым понятиям предикатного символа и терма. Понятие терма определяется рекурсивно и сводится последовательно к понятиям константного символа, переменной и функционального символа. Таким образом, исходное понятие в конце концов сводится к четырем понятиям предикатного символа, константного символа, переменной и функционального символа. При этом не важно, какова природа этих символов; важно лишь только, чтобы их можно было отличить друг от друга и чтобы их нельзя было перепутать с "зарезервированными" символами:

←, (и)

*) От слова successor – последователь. – *Примеч. пер.*

Поэтому договоримся о том, что зарезервированные символы не содержатся в других символах.

Достоинством нисходящих определений является то, что они всегда хорошо обосновываются. Их недостаток заключается в том, что пока целевые понятия определяются в терминах еще не определенных подцелевых понятий, определение в целом не может быть понято, хотя законченные фрагменты его уже представлены.

Восходящие варианты определений противоположны только что приведенным. Они начинаются с понятий, которые не определены либо потому, что они "простейшие" и неопределяемые, либо потому, что они достаточно понятны. Затем в уже известных терминах определяются новые понятия. Определение заканчивается тогда, когда определяется целевое понятие. Определения становятся понятными сразу как только они даются, но их мотивировка не может быть оценена до тех пор, пока все определения не будут сформулированы.

Различие между нисходящим и восходящим проявляется не только в определениях, но также и в формулировках и построении доказательств, и в написании компьютерных программ. Доказательства могут быть представлены в традиционном, восходящем, математическом виде: сначала осмысление того, что дано, затем вывод новых заключений из предыдущих и окончание доказательства, когда цель достигается. Но с другой стороны, доказательства могут быть представлены и нисходящим способом, который отражает процесс его развертывания: процесс возврата назад от цели при помощи сведения целей к подцелям, заканчивающийся тогда, когда обнаруживается, что все подцели достигнуты.

Компьютерные программы также можно создавать как восходящие, начиная с понятных компьютеру простейших программ и продолжая запись новых программ в терминах уже созданных. При этом на каждом этапе программы могут быть выполнены компьютером и проверены. Но если уже написанные программы нижнего уровня не могут быть совместно включены в подходящую программу более высокого уровня, то они должны быть переписаны. Опыт показывает, что лучше применять нисходящий способ программирования, записывая программы высшего уровня в терминах ненаписанных программ более низких уровней. Программы нижних уровней пишутся потом, и, следовательно, гарантируется, что они будут точно сочетаться друг с другом. Более того, программы нижних уровней позднее могут быть изменены и отлажены без воздействия на остальные программы.

Различие между нисходящим и восходящим, равно как и полезность применения математической логики для представления информации, составляют одну из важнейших тем данной книги. Имеется глубокое различие между анализом (нисходящее) и синтезом (восходящее), между телеологией (нисходящее) и детерминизмом (восходящее). Более того, преимущественное применение нисходящего вывода по сравнению с восходящим выводом согласует классический, логический взгляд на рассуждение в том виде *как до́лжно его проводить* с психологическим пониманием того, *как рассуждение проводится* людьми на практике. Нисходящее рассуждение соотносит человеческую стратегию решения задач при помощи сведения целей к подцелям с методом выполнения компьютерных

программ при помощи замены процедурных вызовов телами этих процедур. Оно увязывает изучение логики как с изучением человеческих методов поиска решений, так и с изучением компьютерного программирования.

Семантика клаузальной формы

Синтаксис ведает грамматикой высказываний. Исторически он также занимается правилами вывода и доказательствами. *Семантика* же имеет дело со значением высказывания. Перевод клауз на естественный (английский) язык дает только неформальное введение в их семантику.

В естественных языках мы неявно считаем, что слова и предложения изначально обладают значениями. В логике следует быть более аккуратным. Каждое значение, которое могло бы быть связано с предикатным символом, константным символом, функциональным символом или с высказыванием, должно соотноситься с набором высказываний, выражающих все релевантные посылки. Если в примере родственных связей F1 . . 19 выражают все посылки, то нет никаких причин отбросить такую интерпретацию, в которой имеет место утверждение

F Мать (Зевс, Арес) ←

Такая возможность не противоречит высказанным посылкам F1 . . 19, ибо пока известно, что только они определяют все значения, которые могут быть сопоставлены символам

Мать, Отец, Зевс

и т.д. Для того, чтобы устранить возможность F, необходимо сделать некоторое добавочное допущение, например, такое

F20 ← Мужчина (x), Женщина (x)

F совместно с F1 . . F19, но не совместно с F1 . . 20.

Если задано множество клауз, выражающих все сведения о предметной области, то для того, чтобы понять значение любого индивидного символа или клаузы, необходимо определить, что логически следует из этих сведений. Значение такого предикатного символа "Мать" должно быть соотнесено со всем набором высказываний, которые содержат этот предикатный символ и которые логически выводятся из посылок. Поэтому к значению символа "Мать" в F1 . . 20 можно добавить отрицание

F* ← Мать (Зевс, Арес)

но к значению символа "Мать" в F1 . . 19 его не добавить.

Отсюда следует, что словесные определения значения вовсе не являются необходимыми. Все высказывания о значении могут быть переформулированы в терминах логической импликации. Следовательно для определения семантики клаузальной формы логики достаточно определить понятие логической импликации.

В клаузальной форме логики для того, чтобы определить, что множество посылок влечет некоторое заключение, мы отрицаем, что заключение выполняется и показываем, что это отрицание заключения несовместно с посылками. Семантика клаузальной формы, следовательно, сводится к понятию

несовместности. Для определения, к примеру, того, что следствие F* есть часть того значения материнства, которое задано клаузами F1 . . 20, мы показываем, что отрицание F*, т.е. утверждение F, несовместно с F1 . . 20. Сведение семантики к понятию несовместности может показаться неестественным, но оно имеет значительные вычислительные достоинства.

Несовместность множества клауз может быть продемонстрирована "семантически" при помощи показа того, что нет такой интерпретации этого множества клауз, которая делала бы их все истинными, а может быть продемонстрирована и "синтаксически" при помощи построения доказательства, состоящего из шагов вывода. Настоящая книга посвящена синтаксическому, теоретико-доказательственному методу демонстрации несовместности. Но, поскольку клаузы можно понимать и неформально, переводя их на естественный (английский) язык, а можно и более формально, рассматривая те их интерпретации, в которых они истинны, мы отложим введение в изучение правил вывода и доказательств до гл. 3.

Семантика символической логики, базирующаяся на понятии интерпретации, является независимой от использования правил вывода для манипулирования выражениями в языке. Это выделяет логику из большинства формализмов, применяемых в информатике и искусственном интеллекте. Программы, записанные на обычных языках программирования, нужно понимать в терминах того процесса, который они вызывают в компьютере. Лавина добавочных сведений обрушивается на того программиста, которому приходится программировать в машинноориентированных терминах. Однако, когда программы выражаются средствами символической логики, то они могут быть поняты в терминах их человекоориентированных, естественно-языковых эквивалентов. Лавина сведений при этом обрушивается на машину, что заставляет ее выполнять механические действия (эквивалентные шагам вывода), чтобы определить следует ли логически из информации, заключенной в программе, существование решения данной проблемы.

Нужно, чтобы машина была решателем задач. Тогда задачи построения доказательств, выполнения программ и поиска решений становятся идентичными.

Более того, одинаковые стратегии поиска решений оказываются применимыми как человеком при решении задач, сформулированных в естественных языках, так и машинами при решении задач, сформулированных в символической логике.

Перед тем, как сформулировать строгие семантические определения несовместности и интерпретации, мы проиллюстрируем примерами некоторые выразительные возможности клаузальной формы и некоторые характеристики ее семантики.

Пример "Грекам свойственно ошибаться"

Для того, чтобы показать, что из посылок

G1 Человек (Тьюринг) ←
G2 Человек (Сократ) ←
G3 Грек (Сократ) ←
G4 Ошибается (x) ← Человек (x)

следует заключение, что существует ошибающийся грек, мы сформулируем отрицание этого заключения

G5 ← Ошибается (u), Грек (u)

и покажем, что результирующее множество клауз несовместно.

Проверку несовместности можно провести так, чтобы показать причину несовместности G5 с G1 . . 4, а именно, подстановку

u = Сократ

которая идентифицирует индивида, который одновременно и ошибается и является греком. В этом смысле клауза G5 может рассматриваться как выражение задачи нахождения индивида u, который является ошибающимся греком. Подстановка u = Сократ, которая присутствует в доказательстве, может рассматриваться как решение задачи.

Пример "Грекам свойственно ошибаться" был впервые применен для пояснения работы программ, написанных на языке программирования Плэннер [Hewitt 1969]. Наши же намерения здесь иные: показать, что информация, выраженная средствами логики, может быть понята без понимания процесса, который вызывается ею внутри машины.

Пример "Факториал числа"

Пример с ошибающимся греками не типичен для программ, написанных на обычных языках программирования. Пример же с факториалом числа типичен.

Факториал 0 есть 1
Факториал x + 1 есть x + 1, умноженное на факториал x

Простейшая формулировка нашего определения использует такие функциональные символы:

факт (x)	для факториала числа x
умножить (x, y)	для произведения x и y
s(x)	для x + 1

Кроме того, используется двуместный предикатный символ, выражающий факт равенства. Этот предикат Равно (x, y) имеет место, когда x "есть" y.

Равно (факт (0), 1) ←
Равно (факт (s(x)), умножить (s(x), факт (x))) ←

Для завершения определения нужно добавочно определить характеристику "умножить" и "Равно". Нижеследующие клаузы составляют типичное определение, необходимое для равенства:

(1) Равно (x, x) ←
(2) Равно (x, y) ← Равно (x, z), Равно (z, y)
(3) Равно (факт (x), факт (y)) ← Равно (x, y)
(3′) Равно (x, y) ← Равно (y, x)

Для нахождения, например, факториала числа 2, выразим отрицание факта его существования:

(4) ← Равно (факт (s(s(0))), w)

Но (1) и (4) являются несовместными, а подстановка

w = Факт (s(s(0)))

может быть воспринята как причина несовместности. К сожалению, эта подстановка не очень информативна.

Проблема состоит в том, что функциональные символы "факт", "умножить" и "s" допускают представление чисел многими различными способами. Так, свободные от переменных термы

s(s(0)), s(1), s(факт (0)), s(факт (умножить (0, s(0))))

обозначают одно и то же число 2 и равны один другому. Проблема может быть решена, если индивидам дать различные имена. В приведенном примере это выполняется только для константного символа 0 и функционального символа s. Факториальная функция и функция умножения могут рассматриваться как отношения:

Факт (x, y) имеет место, когда факториал x есть y
Умножить (x, y, z) имеет место, когда x, умноженное на y, дает z.

Тогда клаузы:

Fact1 Факт (0, s(0)) ←
Fact2 Факт (s(x), u) ← Факт (x, v),
Умножить (s(x), v, u)

полностью определяют факториальное отношение для случая, когда имеется приемлемое определение умножения. Отношение равенства не появляется здесь и его определение не нужно. Допустим, что задано определение умножения, включающее такие клаузы:

Умножить (0, x, 0) ←
Умножить (s(0), y, y) ←

и т.д.

Теперь для решения задачи нахождения факториала 2 мы отрицаем, что он существует:

Fact3 ← Факт (s(s(0)), w)

Результирующее множество клауз Fact1 . . 3 несовместно при любом определении "Умножить", из которого можно вывести утверждения

Умножить (s(s(0)), s(0), s(s(0))) ←
Умножить (s(0), s(0), s(0)) ←

Для того, чтобы показать несовместность, нужно выделить одну единственную подстановку:

w = s(s(0))

которая и решает задачу. Такое определение "Факт", дополненное определением "Умножить", служит программой, которая может быть использована в компьютере для вычисления факториалов. Эту программу можно понять, не понимая, как работает машина.

Универсум дискурса и интерпретации

В этом и в следующем параграфе мы определим семантику клаузальной формы. Эти параграфы изложены более строго, чем остальные части главы и могут быть спокойно пропущены при первом чтении.

Две формулировки определения факториала иллюстрируют основной принцип записи клауз. Для того чтобы избавиться от проблем, связанных с тем, что индивиды обладают более, чем одним именем, надо экономно использовать константные и функциональные символы. Если индивиды именуются уникальными, свободными от переменных именами, то *универсум дискурса**) множества клауз, интуитивно представляемый в виде собрания всех индивидов, описываемых клаузами, может быть определен как набор всех свободных от переменных термов, которые можно построить из константных и функциональных символов, встречающихся во множестве клауз. Возможную *интерпретацию* множества клауз можно определить как любую замену каждого n-местного предикатного символа, встречающегося во множестве клауз, n-местным отношением над универсумом дискурса.

Посылки G1 . . 4 задачи об ошибающихся греках являются простейшим примером. Они обладают небольшим, конечным универсумом дискурса, состоящим из двух константных символов

Тьюринг и Сократ

Построение возможной интерпретации означает построение отношения над универсумом дискурса для каждого из трех предикатных символов во множестве клауз. Каждый предикатный символ может иметь четыре разные интерпретации и, таким образом, множество клауз в целом

$$4 * 4 * 4 = 64$$

различные возможные интерпретации**)

Но только две из них делают все клаузы G1 . . 4 истинными. Одна из них делает все свободные от переменных атомы

Человек (Сократ), Человек (Тьюринг),
Ошибается (Сократ), Ошибается (Тьюринг),
Грек (Сократ), Грек (Тьюринг)

истинными. Другая делает атомы

Человек (Сократ), Человек (Тьюринг),
Ошибается (Сократ), Ошибается (Тьюринг),
Грек (Сократ)

истинными, а Грек (Тьюринг) – ложным.

Более обширное множество клауз G1 . . 5 обладает таким же универсумом дискурса и таким же набором из 64 возможных интерпретаций. Однако ни одна из 64 интерпретаций не делает все пять клауз G1. .5 одновременно истинными. Обе интерпретации, которые делают G1. .4 истинны-

---

*) Или универсум Эрбрана. – *Примеч. пер.*

**)Символ * в этой книге используется для обозначения умножения.

ми, делают G5 ложной. Так, пример*) G5

G′5 ← Ошибается (Сократ), Грек (Сократ)

где u = Сократ, ложен в обеих интерпретациях, поскольку два условия Ошибается (Сократ) и Грек (Сократ), отрицаемые в G′5, истинны в обеих интерпретациях. Поскольку G′5 ложно в этих обеих интерпретациях, G5 также ложно (потому что клауза, содержащая переменные, истинна в некоторой интерпретации, если и только если все соответствующие ей примеры истинны, и ложна, если один из примеров ложен). Таким образом, совокупность G1. .5 несовместна, поскольку нет интерпретации, которая делает все клаузы истинными. Анализируя доказательство несовместности, можно установить индивид

u = Сократ

чье существование несостоятельно отрицалось клаузой G5.

Семантический метод показа несовместности множества клауз при помощи демонстрации того, что ни одна интепретация не делает все клаузы истинными, является основным методом, пригодным для любого множества клауз. Более того, рассматриваемые интепретации всегда можно свести к таким, у которых область индивидов состоит из универсума дискурса. Если множество клауз не содержит константных символов, то необходимо включить в универсум дискурса единственный, произвольно выбранный константный символ. В этом случае универсум дискурса состоит из всех свободных от переменных термов, которые могут быть построены на основе данного константного символа и любого функционального символа из тех, которые могут встретиться во множестве клауз.

Включение произвольного константного символа в универсум дискурса, если такового нет во множестве клауз, формализует допущение, что по крайней мере один индивид существует. Благодаря этому допущению из клаузы

(1) Хорошее (x) ←

которая отражает тот факт, что все на свете хорошо, следует, что хоть что-то хорошо. Этот факт не совместен с допущением, что ничто не является хорошим

(2) ← Хорошее (x)

Универсум дискурса состоит из единственного произвольного константного символа, например, ☒ . Имеются только две возможности построить интерпретацию — одна, в которой

Хорошее (☒) истинно,

а другая, в которой

Хорошее (☒) ложно.

Первая интерпретация делает ложным (2). Вторая интерпретация делает ложным (1). Таким образом, (1) и (2) не являются одновременно истин-

*)Результат подстановки в клаузу вместо переменных элементов универсума дискурса. – *Примеч. пер.*

ными ни в одной интерпретации и являются несовместными. Отметьте, что демонстрация несовместности не зависит от имени произвольного элемента универсума дискурса. Аргумент один и тот же вне зависимости от того, какой константный символ выбран.

Понятие интерпретации может быть упрощено. Для того, чтобы задать интерпретацию, достаточно задать ее действие на истинность или ложность свободных от переменных атомарных формул. *Интерпретация* множества клауз, таким образом, может быть представлена как произвольное назначение одного из двух *истинностных значений*

ИСТИНА или ЛОЖЬ

каждому свободному от переменных атому, которое может быть построено исходя из универсума дискурса и исходя из предикатного символа, содержащегося во множестве клауз.

Более строгое определение несовместности

Сейчас мы готовы сформулировать более строгое определение несовместности.

Множество клауз S *несовместно,* если и только если оно не совместно. Оно совместно, если и только если все его клаузы истинны в одной из интерпретаций S.

Клауза истинна в интерпретации множества клауз S, если и только если свободный от переменных пример клаузы, полученный при помощи замены переменных термами из универсума дискурса, истинен в этой интерпретации. В противоположном случае клауза ложна в этой интерпретации.

Свободная от переменных клауза истинна в интерпретации I, если и только если всякий раз, когда все посылки истинны в I, по крайней мере одно заключение истинно в I. Или, что то же самое, клауза истинна в интерпретации I, если и только если по крайней мере одна из ее посылок ложна в I или по крайней мере одно из ее заключений истинно в I. В противоположном случае *клауза ложна* в I

Строгое определение несовместности проясняет семантику пустой клаузы □. Поскольку пустая клауза не имеет ни условий, ни заключений, она не может быть истинной ни в одной интерпретации. Это – единственная клауза, которая является самонесовместной. Для того, чтобы показать несовместность множества клауз, достаточно показать, что из него следует пустая клауза. Пустое множество клауз, однако, совместно. Все клаузы, которые принадлежат ему, истинны во всех интерпретациях, поскольку оно вообще не содержит ни одной клаузы, которая могла бы быть ложной.

Понятия означивания и подстановки, которые мы сейчас дадим, необходимы не только для определения семантики клаузальной формы, но также и для того, чтобы в дальнейшем определить правила вывода. *Пример* (результат означивания) клаузы получается за счет применения подстановки к этой клаузе. *Подстановка* состоит в замене переменных тер-

мами. При этом каждая переменная заменяется только одним термом. Подстановку удобно задавать в виде набора независимых подстановочных компонентов:

$$\{ x_1 = t_1,\ x_2 = t_2, \ldots,\ x_m = t_m \}.$$

Каждый компонент $x_i = t_i$, этой подстановки заменяет переменную x_i термом t_i. *Результат применения подстановки* σ к выражению E есть новое выражение E_σ, которое совпадает с E во всем, за исключением того, что если σ содержит подстановочный компонент $x_i = t_i$, то во всех местах E, где встречается вхождение x_i, в новом выражении будет содержаться вхождение t_i. Применение σ к E заменяет все вхождения одинаковых переменных одинаковыми термами. При этом само *выражение* E может быть любым термом, атомом, клаузой или множеством клауз. Разные переменные могут заменяться одинаковыми термами. Отсюда следует, что разные переменные не обязательно должны сопоставляться разным индивидам.

Например, высказывания

L1 Нравится (Боб, логика) ←
L2 Нравится (Боб, x) ← Нравится (x, логика)
L3 ← Нравится (x, y), Нравится (y, y)
(т.е. никому не нравится тот, кто нравится сам себе)

несовместны, поскольку L1 и L2 несовместны с примером L3

← Нравится (Боб, Боб), Нравится (Боб, Боб)

в котором и x = Боб и y = Боб.

Семантика альтернативных заключений

Строгое определение несовместности проясняет семантику альтернативных заключений. Если у клаузы имеется несколько заключений, то ее нужно понимать как высказывание о том, что, если все посылки выполняются, то по крайней мере одно (но, может быть, и больше) заключение выполняется тоже. Такое *включающее* понимание связки ИЛИ противопоставляется *исключающему* пониманию, при котором "A или B" интерпретируется как высказывание о том, что либо A, либо B ложно*).

Из включающей интерпретации ИЛИ следует, например, что множество высказываний

B1 Животное (x), Минерал (x), Растение (x) ←
B2 Животное (x) ← Устрица (x)
B3 Минерал (x) ← Кирпич (x)
B4 Растение (x) ← Капуста (x)

совместно с возможностью того, что нечто одновременно является и животным и растением:

B5 Животное (x) ← Бактерия (x)
B6 Растение (x) ← Бактерия (x)
B7 Бактерия () ←

*) Но не оба вместе. – *Примеч. Д.А. Поспелова*

Исключительный смысл ИЛИ может быть привнесен во включающее ИЛИ с помощью отрицания. Например, для выражения того факта, что каждый человек либо мужчина, либо женщина, но никак не то и другое сразу, требуются две клаузы

Женщина (x), Мужчина (x) ← Человек (x)
← Женщина (x), Мужчина (x), Человек (x)

Клаузы Хорна

Во многих приложениях логики достаточно ограничиться такой формой клауз, которая содержит не больше одного заключения. Клаузы, содержащие не больше одного заключения, называются *клаузами Хорна,* поскольку они были впервые предложены логиком Альфредом Хорном [Horn, 1951]. На самом деле можно показать (упражнение 5 в гл. 12), что любая задача, выражаемая средствами логики, может быть выражена и при помощи клауз Хорна.

Значительная часть формальных методов, используемых в компьютерном программировании, имеет бо́льшее сходство с клаузами Хорна, чем с "нехорновскими" клаузами. Вдобавок, модели поиска решений задач, создававшиеся в рамках исследований по искусственному интеллекту, в большинстве своем можно считать моделями задач, сформулированных на языке клауз Хорна.

Поскольку клаузы Хорна составляют такое важное подмножество множества всех клауз и поскольку методы вывода для них явно интерпретируются как методы поиска решений или как компьютерные программы, мы детально изучим их (в гл. 3..6) перед общим рассмотрением полной клаузальной формы (в гл. 7..8). Важно отметить, однако, что хотя теоретически без нехорновских клауз можно обойтись, практически они являются необходимыми. Более того, переход от использования в методах поиска решений клауз Хорна к использованию общей клаузальной формы обеспечивает значительное усиление простейших моделей поиска решений, столь еще популярных сегодня.

Пример "Простые грибы и поганки"

В качестве простого примера, естественно описываемого только при помощи нехорновских клауз, можно рассмотреть перечень некоторых типичных поверий о простых грибах и поганках. Допустим, я верю что

(1) Каждый гриб – это простой гриб или поганка
(2) Каждый гриб рода Boletus*) – это гриб
(3) Все поганки ядовиты
(4) Ни один гриб рода Boletus не является простым грибом

Формально:

---

*) К роду Boletus класса гименолицетных грибов относятся как ядовитые сатанинский или ложный белый грибы, так и самые высококачественные съедобные грибы (белый гриб, боровик, подосиновик и т.п.). – *Примеч. пер.*

Fung1 Простой–гриб (x), Поганка (x) ← Гриб (x)
Fung2 Гриб (x) ← Boletus (x)
Fung3 Ядовитый (x) ← Поганка (x)
Fung4 ← Boletus (x), Простой–гриб (x)

Тогда я должен также поверить, по крайней мере, в самые очевидные из логических следствий моих убеждений. В частности, я должен поверить, что все грибы Boletus, в том числе, белые, боровики и т.п., ядовиты.

Fung5 Ядовитый (x) ← Boletus (x)

Но каждый любитель "тихой охоты" знает, что очень немногие грибы Boletus ядовиты, а большинство весьма вкусны. Если отказаться от заключения Fung 5, но сохранять при этом логичность рассуждений, то придется отказаться по крайней мере от одной из исходных посылок Fung 1. .4. Поистине удивительно, как много людей вместо этого отказывается от логики.

Упражнения

1. Используя тот же словарь, что и в F1. .19 (т.е. предикатные, константные и функциональные символы), выразите следующие высказывания на языке клауз

а) x – есть мать у, если
x – женщина и x – родитель у
б) x – есть отец у, если
x – мужчина и x – родитель у
в) x – человек, если
y – родитель x и y – человек

г) Индивид есть человек, если его (или ее) мать есть человек и его (или ее) отец есть человек.

д) Если некто есть человек, то его (или ее) мать есть человек или его (или ее) отец есть человек.

е) Никто не является своим собственным родителем.

2. Пусть даны клаузы, представляющие отношения

Отец (x, y)
Мать (x, y)
Мужчина (x)
Женщина (x)
Родитель (x, y)
Разный (x, y)

Определить следующие добавочные отношения:

М (x) (x есть мать)
О (x) (x есть отец)
С (x, y) (x – сын y)
Дч (x, y) (x – дочь y)
Дд (x, y) (x – дед y)
Бс (x, y) (x – брат или сестра y)

3. Рассмотрим такие соотношения

Нт (x) (x есть небесное тело)
Зв (x) (x заслуживает внимание)
Зз (x) (x звезда)
Ком (x) (x – комета)
Пл (x) (x – планета)
Нед (x, y) (x недалеко от y)
Хв (x) (y x есть хвост)

а) Выразите нижеследующие утверждения на языке клауз

Каждое небесное тело, заслуживающее внимание, представляет собой либо звезду, либо планету, либо комету
Венера есть небесное тело, не являющееся звездой
У комет, расположенных недалеко от солнца есть хвосты
Венера недалеко от солнца, но у нее нет хвоста

б) Какое "очевидное", но пропущенное утверждение должно быть добавлено к полученным в п. а) клаузам для того, чтоб из них можно было вывести заключение о том, что Венера – планета.

4. Используя только предикатные символы Число, Нечетный, и Четный, функциональный символ s *) и константу 0, выразить на языке клауз

а) условия, при которых число четно;
б) условия, при которых число нечетно;
в) ни одно число не является четным и нечетным одновременно;
г) число нечетно, если следующее за ним четно;
д) число четно, если следующее за ним нечетно;
е) число, следующее за данным числом нечетно, если данное число четно и число, следующее за данным числом четно, если данное число нечетно.

5. Пусть

Четность (x, четно) означает, что x четно
Четность (x, нечетно) означает, что x нечетно

Пусть сведения о том, что свойства четности и нечетности противоположны, выражаются следующими двумя клаузами

Против (четно, нечетно) ←
Против (нечетно, четно) ←

Определите понятия "быть нечетным" или "быть четным", используя только три клаузы, причем так, чтобы две из них были бы свободными от переменных утверждениями.

6. Придумайте Ваши собственные предикатные символы, выражающие приведенные ниже высказывания в форме клауз. Воспользуйтесь двумя и только двумя константами, одной для имени моей кошки, а второй – для моего имени.

Птицам нравятся червяки
Кошкам нравятся рыбы
Друзья нравятся друг другу

---

*) Функция следования; s(x) есть x + 1.

Моя кошка – мой друг
Моя кошка ест все, что ей нравится

7. Пусть дуги направленного графа на рис. 1.1 описываются утверждениями вида

Расстояние (r, s, t)

(расстояние по дуге от r до s есть t).

Например, утверждение

Расстояние (A, B, 3)

описывает дугу из A в B. Пусть также задано отношение

Плюс (x, y, z)

которое истинно, когда x + y = z. Используя только одну клаузу, получите расширенное определение отношения

Расст (x, y, z)

которое выражало бы факт наличия пути длиной z от x к y

8. Предположим, что заданы отношения

Пустой (x)	(список x пуст)
Первый (x, y)	(первый элемент списка x есть y)
Остаток (x, v)	(остаток списка x, следующий за его первым элементом есть список v)

Так, отношение, изображенное на рис. 1.2 истинно, когда оба условия Первый (x, u) и Остаток (x, v) истинны.

а) Определите новое отношение

Элемент (z, x) (элемент z – член списка x)

в терминах отношений Первый и Остаток. При этом необходимы две клаузы.

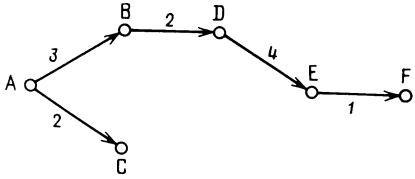


Рис. 1.1

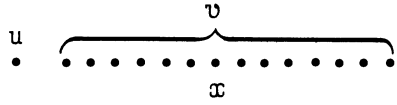


Рис. 1.2

б) Определите отношение

Подсписок (x, y) (все элементы списка x суть элементы списка y)

в терминах отношений Пустой, Первый, Остаток, Элемент.

в) Пусть дано

Плюс (x, y, z) (т.е. x + y = z)

Определите отношение

Сумма (x, w) (сумма всех чисел в списке x есть w)

используя отношения Пустой, Первый, Остаток, Плюс

9. Используя собственные предикатные символы, но не используя функциональные и константные символы, выразите следующие высказывания на языке клауз

Ни один дракон, который живет в зоопарке, не является счастливым

Любой зверь, который встречает добрых людей, является счастливым

Люди, которые посещают зоопарк – добрые

Звери, которые живут в зоопарке, встречают людей, которые посещают зоопарк

Какие два пропущенных добавочных высказывания нужны, чтобы установить заключение, что

Ни один дракон не живет в зоопарке

10. Исходя из словаря, построенного на основе клауз L1. .3 можно построить четыре различных свободных от переменных атома. Следовательно, имеется 16 разных интерпретаций L1. .3. Во скольких интерпретациях истинны одновременно и L1 и L2? Сколько интерпретаций делают истинной L3? Сколько интерпретаций делают истинными L1. .3?

Для того, чтобы построить систему автоматического поиска решений задач, надо уметь описывать для нее информацию на однозначно понимаемом языке. Для того, чтобы система могла служить моделью человеческой деятельности по поиску решений задач, такой язык обязательно должен быть похож на естественные языки, употребляемые в человеческом общении. Представляется, что язык математической логики годится в достаточной мере как для "понимания" и использования в вычислительных системах, так и в качестве упрощенного естественного языка.

В этой главе мы займемся сравнением клаузальной формы логики с некоторыми характерными чертами естественных языков, с семантическими сетями, используемыми для описания смысловых характеристик естественных языков, а также с реляционными базами данных, обеспечивающими представление информации в вычислительных системах. Для того, чтобы такая взаимосвязь логики с естественными языками стала более очевидной, воспользуемся инфиксной нотацией при записи предикатных символов.

Инфиксная нотация

Попробуем придать более формализованный характер той неформальной системе обозначений, которая использовалась для клаузальной формы в начале первой главы.

Имеется возможность записывать *бинарные* (двуместные) предикатные символы между аргументами.

Вместо того, чтобы записывать атомы в *префиксном виде,* например

$$= (x, y),\ \leqslant (x, y),\ \text{Отец}\ (x, y)$$

их можно записать в *инфиксной* форме:

$$x = y,\ x \leqslant y,\ x\ \textit{является отцом}\ y$$

Здесь выражение "является отцом" рассматривается как единый предикатный символ и поэтому выделено.

Унарные (одноместные) предикатные символы могут записываться после своих аргументов без сопровождающих скобок. Например:

$$x\ \textit{хороший} \leftarrow x\ \textit{такой-же-как}\ y,\ y\ \textit{хороший}$$

вместо

Хороший (x) ← Такой-же-как (x, y), Хороший (y)

Поэтому примем за инфиксную нотацию для унарных предикатных символов способ их записи после своих аргументов.

В случае предикатных символов, имеющих больше двух аргументов, инфиксная нотация заключается в чередовании частей предикатного символа с его аргументами. Например:

Джон *передал* книгу *для* Мери

вместо

Передал-для (Джон, книга, Мери)

В первом варианте "*передал*" и "*для*" рассматриваются как первая и вторая части единого предикатного символа "Передал-для".

Хотя инфиксная нотация облегчает чтение, но интерпретация ее не всегда однозначна. Так, выражение

Джон есть студент ←

записанное в инфиксной нотации, может быть проинтерпретировано либо как

Студент (Джон) ←

либо как

Есть (Джон, студент) ←

в префиксной нотации. Для исключения такой неоднозначности будем в дальнейшем выделять инфиксный предикатный символ и его части. Так, атом в клаузе

Джон *есть студент* ←

обладает одним аргументом, в то время как атом в клаузе

Джон *есть* студент ←

двумя. Можно обойтись и без подчеркиваний (как в случае бинарных предикатных символов "=" и "⩽"), если неоднозначная интерпретация невозможна.

Инфиксная нотация может применяться и для функциональных символов. Например, вместо

+(x, y), умножить (x, y), факт (x), s(x), папаша (x)

можно записать

x + y, x * y, x!, x + 1, x-*в папаша*

Вновь инфиксная нотация для функциональных символов и соответствующее соглашение об исключении скобок появятся у нас в гл. 5.

Переменные и типы

Аналогами переменных логики могут служить такие слова естественного языка как "что-то", "что угодно", "все", "ничто", "некоторая вещь", "вещи".

Например,

← x *хорошо*, x *плохо*

т.е. ничто одновременно не бывает плохим и хорошим;

x *плохо* ← x *такое же как* y, y *плохо*

т.е. все, что является таким же как плохое, плохо само по себе.

Однако бывают случаи, когда в логике по-прежнему применяются переменные, а в естественном языке им ставятся в соответствие слова, относящиеся к некоторому типу (или классу). Для того, чтобы и в логике можно было указывать типы, обычно прибегают к использованию одноаргументных предикатных символов. Так, предложение естественного языка

Все люди являются животными

можно выразить клаузой

x *есть животное* ← x *есть человек*

Обратите внимание на то, что переменная x, появившаяся в клаузе, отсутствует в языке благодаря наличию ссылки на тип "люди". Это становится более понятным, если слегка изменить фразу естественного языка на

Люди являются животными.

Договоримся, что слова естественного языка "тот", "некто", "каждый", "везде", "где-то", "всегда", "иногда" будут в дальнейшем относиться к типам "человек", "место", "время".

Такие относительные местоимения как "кто", "который", "где" будут соотноситься с индивидами, уже упомянутыми в том же предложении. Например,

Тот, кто ест животных, является хищником
x *является хищником* ← x *является человеком*,
x *ест* y,
y *является животным*

Усеченная относительная клауза "Кто ест животных" добавляет два внешних условия, касающихся индивидного типа x, упомянутого в основном предложении "Тот является хищником":

x *является хищником* ← x *является человеком*

Отметим, что неусеченная относительная клауза в предложении

Джон, который ест животных, является хищником
Джон *является хищником* ←
Джон *ест* y ← y *является животным*

добавляет только одно внешнее условие в основное предложение.

Словосочетание "является" (английское "is a") во фразах английского языка встречается столь часто, что его естественно рассматривать как единое целое и выделять в качестве бинарного предикатного симво-

ла. Запишем поэтому

x *является* животным ← x *является* человеком

рассматривая типы как самостоятельные индивиды, а не как их свойства. Рассмотрение типов в качестве индивидов повышает выразительную силу языка. Это позволяет, например, записывать клаузы, в которых ссылки на типы делаются при помощи переменных, например,

x *является* y ← x *является* z, z *является* y

Эта клауза выражает *транзитивность* предиката "является". Если типы рассматривать только как свойства, то транзитивность не может быть выражена в форме клауз.

Существование

Факт существования выражается словом "некоторый". В стандартной форме логики существование индивида можно описать, не указывая его имени. В клаузальной же форме логики факт существования выражается указанием имени индивида, причем для этого используются символы констант и функций. Так, например, предложение

Некоторые люди являются животными

может быть выражено при помощи клауз

☺ *является* человеком ←
☺ *является* животным ←

где константный символ ☺ не используется где бы то ни было в других местах для указания различных индивидов. Обращаем внимание на то, что эти же самые клаузы могут рассматриваться в качестве описания и такого предложения естественного языка

Некоторые животные являются людьми.

Слова "имеет", "имеют" также часто выражают факт существования. Так, например, предложение

Зевс имеет любящего родителя *)

может быть трансформировано в такое

Некоторый родитель Зевса любит его

В клаузальной форме логики необходим константный символ для указания имени такого любящего родителя. При этом надо обеспечить, чтобы такое имя нигде не использовалось для именования различных индивидов. Если константный символ ☺ удовлетворяет этим условиям, то вышеприведенное предложение можно символически представить в виде клауз так:

☺ *является родителем* Зевса ←
☺ *любит* Зевса ←

---

*) Оригинал этой не совсем правильной фразы русского языка построен в соответствии с нормами английского языка. – *Примеч. пер.*

Для описания того факта, что каждый имеет любящего его родителя, этому любящему родителю надо дать имя при помощи функционального символа. Действительно, простые клаузы


является родителем x ← x *является* человеком


любит x ← x *является* человеком

описывают более сильное высказывание, говорящее о том, что имеется некий единичный индивид, являющийся родителем любого человека, и что он любит его. Нам же нужно высказать более скромное суждение вроде того, что для каждого человека x найдется индивид, являющийся любящим родителем этого x. Различные же индивиды могут иметь различных любящих родителей. Таким образом, любящий родитель x – это функция от x и, следовательно, имя этого любящего родителя должно быть построено при помощи функционального символа, приложенного к x. Для этого можно использовать любой функциональный символ, если, конечно, везде обеспечено отличие одного символа от другого. Если функциональный символ "род" удовлетворяет этим требованиям, то терм род(x) вполне подходит для именования любящего родителя x, а само предложение может быть выражено при помощи таких клауз:

род (x) *является родителем* x ← x *является* человеком
род (x) *любит* x ← x *является* человеком

Точно так же высказывания

У каждого человека имеется мать
В учреждениях имеются письменные столы
У птиц имеются крылья

могут быть описаны при помощи функциональных символов нижеследующими клаузами:

мама (x) *является матерью* x ← x *является* человеком
пс (x) *является* письменным столом ← x *является* учреждением
пс (x) *имеется* в x ← x *является* учреждением
к (x) *является* крылом ← x *является* птицей
к (x) *является частью* x ← x *является* птицей

Индивиды могут именоваться и при помощи функциональных символов с несколькими аргументами

Так, предложение

Для каждого индивидного символа x и для каждого списка y существует список, первым элементом которого является x, а остатком является y

можно выразить, например, такими клаузами:

cons (x, y) *является* списком ← y *является* списком
x *является первым* в cons (x, y) ← y *является* списком
y *является остатком* в cons (x, y) ← y *является* списком

где терм cons (x, y) именует собою список на рис. 2.1, *построенный* посредством постановки x перед началом списка y. Инфиксная нотация является

более легкой для чтения, но префиксная нотация более компактна:

C (cons (x, y)) ← C (y)
Первый (x, cons (x, y)) ← C (y)
Остаток (y, cons (x, y)) ← C (y)

Как мы видели, существование индивида, на который имеется ссылка в заключении предложения, должно быть задано при помощи константного или функционального символа. Однако, если на этот индивид имеет-

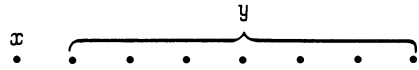


Рис. 2.1

ся ссылка в условиях предложения, но не в заключении, то его надо задавать переменной. Например,

Некий человек приходится дедом или бабушкой другому, если у него имеется ребенок, являющийся родителем этого другого человека

x *является дедом или бабушкой* y ← x *является человеком*,
y *является человеком*,
x *является родителем* z,
z *является родителем* y

Понимание клаузы часто облегчается, когда переменные, встречающиеся в условиях, но не встречающиеся в заключениях, получают оттенок существования. Например, клауза

Мери *нравится* Джон ← Мери *нравится* x

может быть прочитана и так:

Если вообще существует что-нибудь, что нравится Мери, то Мери нравится Джон

Клауза

x *имеет* y ← z *передает* y *для* x

описывает тот факт, что x имеет y, если кто-то передает y для x.

Отрицание

В стандартной форме логики отрицание представляется непосредственно. В клаузальной форме логики оно выражается косвенно. Клаузы с пустым заключением

← Мать (Зевс, x)
← Мать (x, y), Отец (x, y)

например, означают, что

Зевс не является ничьей матерью и никто не является одновременно матерью и отцом.

Особенность семантики клауз заключается в том, что отрицательные условия могут быть переформулированы как неотрицательные заключения. Так, предложение

Роберт на работе, когда он не на своей квартире

которое непосредственно может быть выражено в стандартном виде с отрицательным условием

На (Роберт, работа) ← не-На (Роберт, квартира)

может быть также выражено и без отрицания в клаузальной форме с помощью нехорновской клаузы

На (Роберт, работа), На (Роберт, квартира) ←

Предложение в стандартной форме

не-Счастливый (Джон) ← не-Нравится (Мери, Джон)

может быть преобразовано в клаузальную форму так

Нравится (Мери, Джон) ← Счастливый (Джон) *)

Отметим, что зачастую различные предложения естественного языка, например

Каждый гриб, не являющийся поганкой, простой
Каждый гриб, не являющийся простым, поганка
Все, что не является поганкой или простым грибом, не является грибом

могут быть выражены единой клаузой

Поганка (x), Простой-гриб (x) ← Гриб (x)

Отрицания заключений, являющихся импликациями

Чтобы в клаузальной форме показать, что из условия следует заключение, необходимо выполнить отрицание заключения и доказать, что это отрицание заключения и условие несовместны.

Чаще всего заключение имеет вид импликации, например

Все грибы рода Boletus ядовиты
Ядовитый (x) ← Boletus (x)

В общем случае, *импликация* есть клауза Хорна с единственным заключением и одним или несколькими условиями. Клауза Хорна с заключением, но без условий называется *утверждением.* Как обычно, воспользуемся удобной возможностью использовать термин "импликация" в расширенном смысле, понимая под ним и утверждение.

Для выполнения отрицания импликации необходимо предположить существование индивидов, удовлетворяющих всем условиям, и посчитать неверным то, что они удовлетворяют заключениям. Например, мы

*) Это верно лишь в предположении о замкнутости мира, так как иначе Джон мог быть счастлив по другой причине – *Примеч. Д.А. Поспелова*

предполагаем существование индивида , который относится к роду Boletus, и отрицаем то, что он ядовитый

Boletus () ←
← Ядовитый ()

В гл. 10 при изучении стандартной формы логики будет построена систематическая процедура преобразования отрицательных предложений в клаузальную форму. А пока удовлетворимся применением вышеприведенного правила для отрицания заключений, имеющих вид импликаций.

Условия, являющиеся импликациями

В естественном языке и в стандартной форме логики обычна ситуация, когда условия имеют вид импликации. Например, импликация

Всем студентам Боба нравится логика,

которая имеет структуру клаузы Хорна

x *нравится* логика ← x *является студентом* Боба

представляет собой условие предложения

(1) Боб счастлив, если всем его студентам нравится логика

Хотя это предложение и может быть непосредственно выражено в стандартной форме логики, для преобразования в клаузальную форму его надо перефразировать. Вновь сошлемся на гл. 10, в которой будет представлен систематический метод преобразования таких предложений из стандартной формы в клаузальную. В данной главе мы лишь проиллюстрируем этот метод последовательной трансформацией исходного предложения (1) в рамках естественного языка

(2) Не всем студентам Боба нравится логика, если Боб несчастлив *)

(Неотрицательные условие и заключение в (1) стали отрицательными заключением и условием в (2)).

(3) У Боба имеется студент, которому не нравится логика, раз Боб несчастлив

(Заключение в (2), являющееся отрицанием импликации, преобразовано при помощи утверждения существования индивида, удовлетворяющего условию "быть студентом Боба", но не заключению о том, что "ему нравится логика".)

(4) У Боба есть студент, например , и не нравится логика, раз Боб несчастлив

(Виновник несчастья получает имя.)

(5) является студентом Боба, раз Боб несчастлив
 не нравится логика, раз Боб несчастлив

*) Это не совсем верно (см. примеч. на с. 43). – *Примеч. Д.А. Поспелова.*

(Теперь каждое заключение сообщается в одном из двух предложений, имеющих одинаковые условия.)

(6) ☹ является студентом Боба или Боб счастлив
Боб счастлив, если ☹ нравится логика *)

(Отрицательное условие преобразуется в неотрицательное заключение, а отрицательное заключение преобразуется в неотрицательное условие.)

(7) ☹ *является студентом* Боба, Боб *счастлив* ←
Боб *счастлив* ← ☹ *нравится* логика

Перевод с естественного языка на язык клауз бывает и более сжатым. В простом случае, когда предложение естественного языка имеет форму

A, если B следует из C,

т.е.

$A \leftarrow [B \leftarrow C]$

в стандартной форме логики, тогда соответствующие клаузы приобретают вид:

A, C ←
$A \leftarrow B$

Если же, как в вышеприведенном примере, условие вида $B \leftarrow C$ содержит переменные, то возникают осложнения, состоящие в том, что эти переменные приходится заменять константными символами или термами, включающими функциональные символы.

Хотя появление предложений с условиями, являющимися импликациями, выглядит несколько неестественно на языке клауз, они, как это будет видно в гл 7 и 8, обладают вполне естественной интерпретацией в процессе поиска решений. В гл. 10 такие предложения будут исследованы более детально. А до этого будем рассматривать только такие предложения, которые могут быть выражены клаузами Хорна с условиями в виде простых атомарных формул.

Определения и словосочетание если-и-только-если

В математике и логике в определениях часто фигурирует словосочетание если-и-только-если:

x является дедом или бабушкой y, если-и-только-если
существует z, являющийся ребенком x и родителем y

Выражение

A если-и-только-если B

интерпретируется как

A если B и A только-если B

В свою очередь

A только-если B

обычно интерпретируется как

B если A

*)Это не совсем верно (см. примеч. на с. 43). – *Примеч. Д.А. Поспелова.*

Последняя интерпретация только-если, однако, не является единственной. В гл. 11 будут обсуждаться альтернативные интерпретации.

Выражение если-и-только-если можно непосредственно передать формулой языка стандартной логики. На языке клауз, однако, две его половинки должны быть заданы независимо. Более того, половинка только-если зачастую выглядит не очень естественно. Так, в примере о дедушке и бабушке приходится в половинке только-если использовать новый функциональный символ для того, чтобы назвать родственника x и y, приходящегося ребенком x и родителем y.

x *является родителем* родств (x, y) ← x *является дедом или бабушкой* y

родств (x, y) *является родителем* y ← x *является дедом или бабушкой* y

Определения вида если-и-только-если вместе с предложениями, содержащими импликативные условия, как раз и составляют два основных случая, когда логика клауз кажется более неуклюжей по сравнению и с естественным языком, и со стандартной формой логики. До гл. 10 и 11 мы сознательно уклонимся от этих сложностей, используя лишь если-части определений, тем более, что этого достаточно для большинства целей.

Семантические сети

Многие исследователи в области искусственного интеллекта для описания информации в вычислительных системах используют не математическую логику, а семантические сети. Они могут применяться и как модели организации памяти человека, и как представительные схемы для описания значения предложений естественного языка. *Семантическая сеть* есть граф, *узлы* (вершины) которого соответствуют индивидами, а *ориентированные дуги* (ребра) соответствуют бинарным отношениям и связям. Каждый индивид представляется только одним узлом. Так например, информация из клауз F1 . . 6 гл. 1 может быть представлена при помощи семантической сети на рис. 2.2.

В общем случае семантическая сеть может рассматриваться как эквивалент множества свободных от переменных утверждений, представленных ребрами этой сети. На рис. 2.3 дуга, отмеченная R и направленная из узла s в узел t, представляет собой утверждение

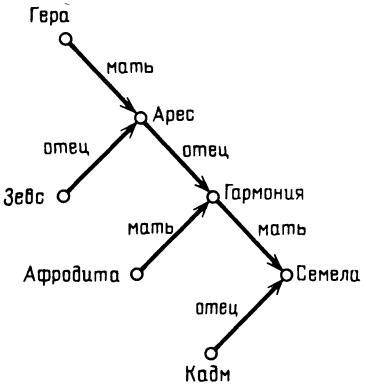


Рис. 2.2

$$R(s, t) \leftarrow$$

Простые семантические сети не имеют средств для представления переменных, функциональных символов, n-арных предикатных символов или клауз, обладающих условиями или альтернативными заключениями. Как мы вскоре увидим, ограничение на использование только бинарных отношений не является критичным, поскольку каждое n-арное отношение

может быть представлено в виде соединения n + 1 бинарных отношений. Другие ограничения, однако, являются более серьезными; они стали для многих исследователей побудительными причинами предложить некоторые обобщения [Shapiro 1971, 1972], [Hendrix 1975], [Shubert 1977], каждое из которых трактует семантические сети как альтернативный

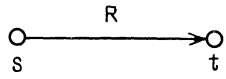


Рис. 2.3

синтаксис для математической логики. Одно из обобщений, которое мы сейчас опишем, рассматривает расширенные семантические сети в качестве графического синтаксиса для логики клауз [Deliyanni, Kowalski 1979].

Расширенные семантические сети

Как и в простых семантических сетях, вершины сопоставляются индивидам, а ребра представляют собой бинарные отношения. Но здесь вершины могут быть константами, переменными или термами, содержащими функциональные символы. Ребра могут представлять собой как условия так и заключения и могут группироваться в клаузы. Условия будут записываться двойными линиями, а заключения одинарной жирной линией как и раньше. Клаузы, содержащие более одного атома выделяются при

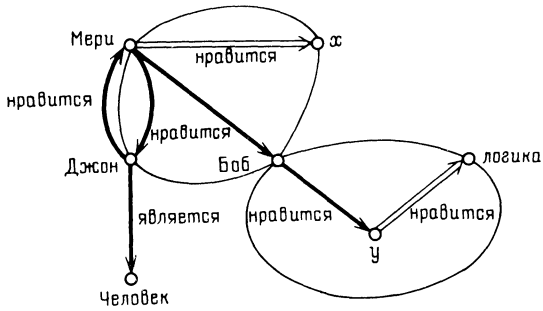


Рис. 2.4

помощи указания подсетей. Расширенная семантическая сеть на рис. 2.4 соответствует множеству клауз

Джону *нравится* Мери ←
Джон *является* человеком ←
Мери *нравится* Джон, Мери *нравится* Боб ← Мери *нравится* x
Бобу *нравится* y ← y *нравится* логика

Помимо изобразительных качеств, семантические сети обладают еще двумя достоинствами: они обеспечивают удобный способ запоминания информации, а также укрепляют тенденцию использования бинарных, а не более общих n-арных предикатных символов. Тот факт, что любой

индивид представляется единственной вершиной, означает, что вся информация об этом индивиде непосредственно доступна из этой вершины. Это полезное свойство было использовано при разработке маршрутно-поисковых стратегий поиска решений задач. Несмотря на эти соображения, в следующих двух пунктах мы будем сравнивать использование бинарных предикатных символов и использование более общих n-арных предикатных символов.

Представление информации при помощи бинарных предикатных символов

Каждый n-арный предикатный символ может быть представлен как соединение n + 1 бинарного отношения. Например, утверждение

Джон *передал* книгу *для* Мери ←

можно переформулировать на естественном языке так:

Имеется событие e,
которое заключается в *действии* передачи
действующим-субъектом Джоном
объекта книги
для *субъекта–приемника передачи* Мери

Если игнорировать тот факт, что e является объектом типа событие, то

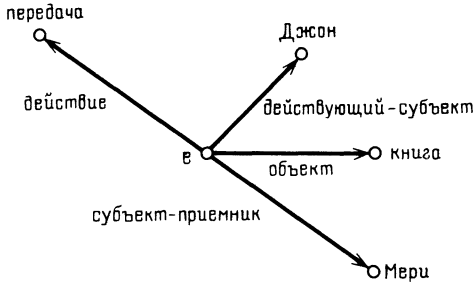


Рис. 2.5

в клаузальной форме одно трехместное отношение может быть переформулировано как 4 бинарных отношения

e *является действием* передачи ←
e *имеет действующего субъекта* Джона ←
e *имеет объект книгу* ←
e *имеет субъекта–приемника передачи* Мери ←

Представление этих клауз в виде семантической сети приведено на рис. 2.5. Оно похоже на анализ вариантных структур в естественных языках, употреблявшийся в лингвистике [Fillmore 1968] и искусственном интеллекте [Quillian 1968], [Shank 1973, 1975], [Simmons 1977].

В общем случае для того, чтобы заменить n-арную связь бинарными, необходимо рассматривать эту n-арную связь и ее отношения как индивиды (придавая им имена, как, например, "е" и "передача" в рассмотренном примере). Вдобавок нужно задать еще одну бинарную связь, показывающую, что данная n-арная связь описывается n-арным отношением: в нашем примере такую функцию несет бинарная связь

е *является действием* передачи ←

Для каждого аргумента n-арной связи бинарная нужна, чтобы отразить тот факт, что этот аргумент принадлежит n-арной связи.

В дальнейшем мы будем говорить о представлении информации в виде общих n-арных связей как об *n-арном представлении,* а о соответствующем ему представлении посредством бинарных отношений, как о *бинарном представлении.*

Бинарные связи могут заменять n-арные связи как в условиях, так и в заключениях клауз. Например, предложение

Некоторое лицо становится обладателем некоторого объекта
после того как этот объект был ему передан

может быть выражено и в такой форме

Для каждого события u, при котором x передает
объект y для субъекта z, существует ситуация,
назовем ее, например, результат (u), возникающая сразу
после u и являющаяся состоянием обладания субъекта z
объектом y.

Последовательная переформулировка предложения на языке клауз, использующем бестиповые бинарные предикатные символы, порождает четыре хорновских клаузы, причем все они имеют одинаковые условия.

результат (u) *следует непосредственно после* u ←
u *является действием* передачи,
u *имеет действующий субъект* x,
u *имеет объект* y,
u *имеет субъекта–приемника передачи*
результат (u) *является состоянием обладания* ←
u *является действием* передачи,
u *имеет действующий субъект* x,
u *имеет объект* y,
u *имеет субъекта–приемника передачи* z
результат (u) *имеет субъект* z ←
u *является действием* передачи,
u *имеет действующий субъект* x,
u *имеет объект* y,
u *имеет субъекта–приемника передачи* z
результат (u) *имеет объект* y ←
u *является действием* передачи
u *имеет действующий субъект* x,
u *имеет объект* y,
u *имеет субъекта–приемника передачи* z

В этом примере бинарное представление оказывается менее компактным, чем n-арное представление, которое включает явные аргументы для действия и состояния результат (u).

результат (u) *следует непосредственно после* u ←
u *является действием* передачи x-*м* y-*ка для* z
результат (u) *является состоянием обладания* z-*м*
y-*ка* ← u *является действием* передачи x-*м*
y-*ка для* z*)

Однако, если принять во внимание, что каждое действие передачи имеет действующего субъекта, объект и субъекта—приемника, то исходное бинарное представление может быть переформулировано более компактно:

результат (u) *следует непосредственно после* u ←
u *является действием* передачи
результат (u) *является состоянием* обладания ←
u *является действием* передачи
результат (u) *имеет субъект* z ←
u *имеет субъекта-приемника передачи*
результат (u) *имеет объект* y ← u *имеет объект* y

Преимущества бинарного представления

Бинарное представление, вообще говоря, обладает большей выразительной способностью, нежели n-арное представление. Оно облегчает добавление новой информации и игнорирование пока еще неизвестной информации.

В бинарном представлении отношения и связи рассматриваются как индивиды. Следовательно, оказывается возможным включать их в такие предложения

Мери желает, чтобы Джон передал ей эту книгу
Мери *желает, чтобы* e ←

Соответствующее этому выражение в n-арном представлении

Мери *желает*, чтобы (Джон *передал* книгу *для* Мери) ←

недопустимо на языке клауз.

Способность бинарного представления описывать связи облегчает также и попытки добавления информации о них. Например, если взять высказывание о том, что

Джон передал книгу для Мери

то для того, чтобы добавить сюда новую информацию, что он сделал это в Гайд-парке, в бинарном представлении требуется всего лишь присоеди-

*) В английском языке функторная часть n-арного предиката легко формируется из предлогов и распределяется по фразе; в русском языке зачастую единственная возможность "проявить" n-арный предикат – явно указать падежные окончания. – *Примеч. пер.*

нение нового утверждения

Гайд-парк *является местом, где случилось* e ←

Но в n-арном представлении такое действие потребует замены исходного утверждения, использующего трехместный предикатный символ

Джон *передал* книгу *для* Мери ←

новым четырехместным символом

Джон *передал* книгу *для* Мери *в* Гайд-парке ←

Подчеркнем, что именно рассмотрение связей как индивидов обеспечивает преимущества бинарного представления в двух вышеприведенных примерах. Оба предложения

Мери *желает, чтобы* e ←
Гайд-парк *является местом локализации* e ←

могут быть выражены в n-арном представлении только с явным аргументом, именующим связь

e *является действием* передачи Джон-*ом* книг-*и для* Мери ←

Бинарное представление также более удобно, нежели n-арное представление, когда компоненты связи неизвестны. Например, чтобы высказать, что

Эта книга была передана Джону

в бинарном представлении достаточно просто указать только то, что известно и проигнорировать то, что неизвестно

e′ *является действием передачи* ←
e′ *имеет объект* книга ←
e′ *имеет субъект–передачи* "Джон" ←

В двух n-арных представлениях этого же факта приходится давать имя неизвестному субъекту действия

☺ *передала* книгу *для* Джона ←

или

e′ *является действием* передачи книг-*и для* Джона *от* ☺ ←

Впрочем, этот довод в пользу бинарных отношений не столь убедителен. Имеется много отношений, например

x *умножить на* y *будет* z
x *получил азимут* y *по курсу* z
x *является* y-*вым элементом последовательности* z
v *является полученным с помощью процедуры* u
доказательством того, что из предположения x
следует заключение y

для которых n-арное представление более удобно, чем бинарное представление. Сверх того, заметим, что общие n-арные, а не бинарные отношения гораздо чаще используются в базах данных.

Базы данных

База данных – это совокупность данных, предназначенных для использования в различных целях. Базы данных могут содержать, например, сведения о кадрах некоторой фирмы, или о каких-либо деталях банковских операций или сведения для полиции об уличенных преступниках. Подчеркнем, к тому же, что базы данных должны быть реализованы так, чтобы позволить обработку на компьютерах. Компьютеры должны применяться для обновления баз данных, для проверки совместности данных, для ответа на информационные запросы.

Клуб-родившихся-в-свой-день-рождения	Имя	Должность	Взнос	День-рождения	День-вступления
	Мери	президент	10	4.Мар.77	4.Мар.77
	Джон	секретарь	10	2.Мар.78	2.Мар.78
	Боб	казначей	10	1.Янв.80	1.Янв.80

Рис. 2.6

К конкретной базе данных могут обращаться за информацией многие пользователи, имеющие лишь минимальный опыт общения с компьютером. В таком случае должно быть обеспечено простое внешнее представление данных, которое к тому же не должно зависеть от их представления внутри компьютера. Следовательно, язык запросов к базе данных должен быть одновременно простым в изучении и легким в пользовании. Сейчас принято считать, что эти требования удовлетворяются наилучшим образом, когда данные представляются в виде отношений (реляционное представление) [Codd 1970].

Реляционное представление данных эквивалентно их представлению в виде таблиц. Действительно, места аргументов отношения можно рассматривать как столбцы таблицы, а связи, из которых построено отношение – как строки таблицы. Поэтому представленная на рис. 2.6 5-столбцовая и 3-строчная таблица является 5-арным отношением; его можно описать тремя утверждениями:

Клуб-родившихся-в-свой-день-рождения (Мери, президент, 10-пенсов, 4. Мар. 77, 4. Мар. 77) ←
Клуб-родившихся-в-свой-день-рождения (Джон, секретарь, 10-пенсов, 2. Мар. 78, 2. Мар. 78) ←
Клуб–родившихся–в–свой–день–рождения (Боб, казначей, 10–пенсов, 1. Янв. 80, 1. Янв. 80) ←

Та же самая информация может быть описана и при помощи бинарных предикатных символов. В этом примере бинарное представление может быть еще более упрощено, поскольку каждая строка таблицы однозначно определяется значением в ее первом столбце. Поэтому первый столбец выступает в качестве ключа таблицы. В бинарном представлении таблицы

ключ может служить именем той связи, которую он идентифицирует:

B1 Член (Мери, клуб–родившихся–в–свой–день–рождения) ←
B2 Член (Джон, клуб–родившихся–в–свой–день–рождения) ←
B3 Член (Боб, клуб–родившихся–в–свой–день–рождения) ←
B4 Должность (Мери, президент) ←
B5 Должность (Джон, секретарь) ←
B6 Должность (Боб, казначей) ←
B7 Взнос (Мери, 10–пенсов) ←
B8 Взнос (Джон, 10–пенсов) ←
B9 Взнос (Боб, 10–пенсов) ←
B10 День–рождения (Мери, 4. Мар. 77) ←
B11 День–рождения (Джон, 2. Мар. 78) ←
B12 День–рождения (Боб, 1. Янв. 80) ←
B13 Дата–вступления (Мери, 4. Мар. 77) ←
B14 Дата–вступления (Джон, 2. Мар. 78) ←
B15 Дата–вступления (Боб, 1. Янв. 80) ←

Отметим, что хотя бинарное представление таблицы длиннее n-арного, воспринимать его легче. Имена столбцов, которые необходимы для понимания таблицы отсутствуют в n-арном представлении, но присутствуют как имена бинарных предикатных символов в бинарном представлении.

Но еще более важным с точки зрения обработки на компьютере является возможность выражать с помощью бинарного представления такие закономерности, которые средствами n-арного представления выражены быть не могут. В частности такие правила как

Взнос (x, 10-пенсов) ←
    Член (x, клуб–родившихся–в–свой–день–рождения).
Дата–вступления (x, y) ←
    Член (x, клуб–родившихся–в–свой–день–рождения),
    День–рождения (x, y)

вполне могут заменить конкретные утверждения B7 . . 9 и B13 . . 15 в бинарном представлении, но в n-арном представлении не могут быть сформулированы вообще.

Языки запросов

Реляционное представление данных чаще применялось для формирования запросов к данным, нежели для представления данных.

В большинстве реляционных языков запросов используется аппарат математической логики или реляционной алгебры. Например, язык запросов реляционного представления [Codd 1972] использует бинарное представление отношений. Если даны, например, таблица Клуба–родившихся–в–свой–день–рождения и таблица Адресов (их структура приведена на рис. 2.7), то, в соответствии с правилами табличного языка query by example [Zloof 1975], запрос

Какие члены Клуба–родившихся–в–свой–день–рождения
    живут на проспекте Евклида?

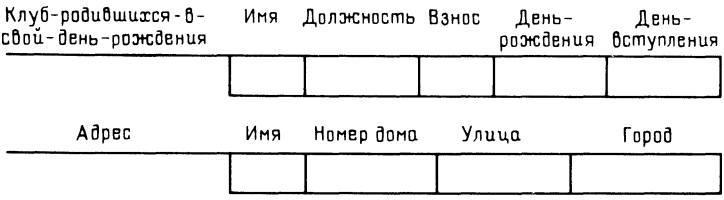


Рис. 2.7

можно сформулировать в n-арном представлении так:

← Ответ (x)
Ответ (x) ← Клуб (x, y, z, u, v),
Адрес (x, y', просп. Евклида, z')

Связь между запросами, выраженными в клаузальной форме логики и запросами, реализованными в языке query by example продемонстрировал Ван Эмден [Van Emden 1979]. Классификацию реляционных языков запросов, базирующихся на стандартной форме логики провел Пиротт [Pirotte 1978].

Описание данных

Реляционная модель данных непосредственно не связана с формальными средствами представления данных в компьютере. Она совместима с любым формализмом, который можно рассматривать абстрактно в терминах отношений. Тем не менее, использование средств математической логики выглядит весьма привлекательно. Оно имеет то достоинство, что один и тот же формализм может применяться как для выражения запросов, так и для определения данных. Больше того, когда данные могут быть определены формулировками общих закономерностей, описание данных неотличимо от программы. Например, предложение

Взнос (x, 10–пенсов) ← Член (x, клуб–родившихся–
в–свой–день–рождения)

можно рассматривать и как общую закономерность, и как программу, вычисляющую взнос, уплачиваемый членами клуба, родившихся в свой день рождения.

Использование математической логики для описания данных и для формулировки запросов в запросно-ответных системах отмечено раньше, чем появились реляционные базы данных. В числе первых можно указать системы, описанные Дарлингтоном [Darlington 1969] и Грином [Green 1969a, 1969b]. В частности, Грин впервые ввел предикатный символ Ответ. Более новые системы были разработаны в Марселе [Colmerauer 1972], [Dahl, Sambuc 1976] и в Мэриленде [Minker 1973], [Mc Skimin, Minker 1977] и Николя вместе с Сиром [Nicolas, Syre 1974], а также Келлогом, Клэром, Трэвисом [Kellog, Klahr, Travis 1978].

Ограничения целостности

Поскольку в данных часто могут встречаться ошибки, постольку для описания специальных условий, которым должны удовлетворять корректные данные, надо использовать *ограничения целостности.*

Так, например, клауза

```
у раньше z ← Сегодня (z),
             Член (x, клуб–родившихся–в–свой–
                      день–рождения),
             День–рождения (x, y)
```

выражает тот факт, что все члены клуба родившихся в свой день рождения родились раньше, чем сегодня. Если бы сегодня было 1. Апр. 79

```
Сегодня (1. Апр. 79) ←
```

и если согласиться с вышеприведенным отношением *раньше,* то данные

```
Член (Боб, клуб–родившихся–в–свой–день–рождения) ←
День–рождения (Боб, 1. Янв. 80) ←
```

окажутся несовместными с ограничениями целостности и должны быть отвергнуты интеллектуальной системой управления базой данных.

Использование математической логики в качестве формальной основы для описания информации стирает известные различия между базами данных и программами. Ограничения целостности в базах данных неотличимы от программно определяемых свойств. Так, клауза

```
x ⩽ y ← Факт (x, y)
```

описывает свойство, которому должно удовлетворять корректное определение отношения "быть факториалом числа". С точки зрения ограничений целостности целью этой клаузы является не соучастие в определении отношения ⩽ или отношения Факт, но предотвращение того, что эти определения получат недопустимые свойства. Ограничения целостности можно использовать и в других целях. Например, их можно применить для отказа от недопустимых запросов типа

```
Какое целое число, меньшее 1300, является факториалом
    числа 5200?
```

и для преобразования слишком сложных целей в более простые. Применение ограничений целостности для целей поиска решений задач рассматривается в гл. 9.

Пример "База данных кафедры"

Пролог [Roussel 1975] – созданная в Марселе система поиска решений, работающая на языке клауз Хорна, использовалась при решении многих задач, когда надо было соединить возможности баз данных и программ. Пролог использовался в Марселе для формирования ответов на вопросы, заданные на естественном языке [Colmerauer 1972], [Dahl, Sambuc 1976] и для символьного интегрирования [Bergman, Kanou 1973], в Эдинбурге – для задач построения планов [Warren 1974, 1976], доказательства

геометрических теорем [Welham 1976], [Coelho, Pereira 1975], решения задач механики, сформулированных на английском языке [Bundy 1979] и для построения компиляторов [Warren, Pereira, Pereira 1977], а также в Будапеште в системах автоматизированного проектирования [Markusz 1977] и в фармакологии [Futo, Darvas, Szeredi 1978]. В Лондоне мы применили Пролог для построения базы данных, описывающей деятельность нашей кафедры. В качестве примера приведем наиболее типичные клаузы из этого описания:

x *занят* y ← x *преподает* y
x *занят* y ← x *посещает* y
x *занят* y ← x *является членом комиссии* y
9 : 30 *есть время начала* 304 ←
Пятн *есть день проведения* 304 ←
3 *есть курс, на котором преподается* 304 ←
145 *есть аудитория, где проводится* x ← 3 *есть курс, на котором преподается* x
Роберт А. Ковальски *преподает* 304 ←
у аудитории 145 *вместимость* 80 ←
65 *человек посещает* 304 ←
x *посещает* y ← x *является студентом* z*-го года обучения*, z *есть курс, на котором преподается* y
Поиск решений задач – *это название* 304 ←

В этом примере предполагается, что дисциплина 304 встречается в расписании один раз в неделю. Если бы она встречалась чаще, то можно было бы использовать, например, составные термы часть(304, 1), часть(304,2) для именования различных частей этой дисциплины.

Разнообразные ограничения целостности типа такого

← x *есть аудитория, где проводится* y,
у аудитории x вместимость u,
v *человек посещает* y, u > v

могут применяться для проверки непротиворечивости данных. Ответы на запросы можно получать, отрицая наличие ответов на запрос, а затем доказывая несовместность и выделяя из доказательства информацию, необходимую для получения ответа. Так, для определения рода деятельности, которым Роберт А. Ковальски занят в 9 : 30 по пятницам, достаточно выполнить отрицание того, что эта деятельность имеет место:

← Ответ (x)
Ответ (x) ← Роберт А. Ковальски *занят* x,
9 : 30 *есть время начала* x,
Пятн *есть день проведения* x

Подстановка x = 304, которая получается в процессе доказательства, является ответом на запрос. Система поиска решений выделяет этот ответ автоматически.

Равенство

Там, где в настоящей книге используются различные предикатные символы, в обычной математической записи чаще применяются функциональные символы и специальный бинарный предикатный символ = (равенство). Так, чаще записывают

x * y = z	вместо Умножить (x, y, z)
x! = y	вместо Факт (x, y)
x = отец (y)	вместо Отец (x, y)

Язык запросов реляционного исчисления тоже использует функциональные символы и равенство:

должность (x) = y	вместо Должность (x, y)
взнос (x) = y	вместо Взнос (x, y)
день–рождения (x) = y	вместо День–рождения (x, y)
дата–вступления (x) = y	вместо Дата–вступления (x, y)

Функциональные обозначения обычно более компактны, чем реляционные. Действительно, фразу

> День, в который член клуба родившихся в свой день рождения вступил в этот клуб, совпадает с его днем рождения

проще записать в функциональных обозначениях:

день–рождения (x) = дата–вступления (x) ← Член (x, клуб–родившихся–в–свой–день–рождения)

нежели в реляционных обозначениях

День–рождения (x, y) ← Член (x клуб–родившихся–в–свой–день–рождения), Дата–вступления (x, y)

День–вступления (x, y) ← Член (x, клуб–родившихся–в–свой–день–рождения), День–рождения (x, y)

Равенство особенно необходимо в тех случаях, когда индивид имеет больше одного имени. Например

Jovis = Юпитер ←

Для указания того, что один аргумент отношения является функцией другого, равенство используется даже в реляционных обозначениях:

x = y ← Отец (x, z), Отец (y, z)

С целью доказательства несовместности некоторого множества клауз S, содержащего символ равенства, в это множество должны быть включены аксиомы, характеризующие отношения равенства для каждого функционального символа f и для каждого предикатного символа P (в том числе и для символа равенства), встречающихся в S

E1 $x = x \leftarrow$

E2 $P(x_1, \ldots, x_m) \leftarrow P(y_1, \ldots, y_m),\ x_1 = y_1, \ldots, x_m = y_m$

E3 $f(x_1, \ldots, x_m) = f(y_1, \ldots, y_m) \leftarrow x_1 = y_1, \ldots, x_m = y_m$

Например, пусть требуется продемонстрировать, что из допущений

J1 Джекил = Хайд ←

J2 отец (Джон) = Хайд ←

J3 Член (отец (Джон), клуб–родившихся–в–свой–день–рождения) ←

следует заключение

Член (Джекил, клуб–родившихся–в–свой–день–рождения) ←

Для этого надо выполнить отрицание заключения

J4 ← Член (Джекил, клуб–родившихся–в–свой–день–рождения)

и добавить соответствующие аксиомы для отношения равенства

J5 $x = x \leftarrow$

J6 Член $(x_1, x_2) \leftarrow$ Член $(y_1, y_2),\ x_1 = y_1, x_2 = y_2$

J7 $x_1 = x_2 \leftarrow y_1 = y_2,\ x_1 = y_1,\ x_2 = y_2$

J8 отец (x) = отец (y) ← x = y

Полученное множество клауз J1 . . 8 несовместно, потому что J1 . . 3 "очевидным образом" несовместимы с приводимыми ниже означиваниями J5 . . .7

Хайд = Хайд ←

клуб–родившихся–в–свой–день–рождения = клуб–родившихся–в–свой–день–рождения ←

Член (Джекил, клуб–родившихся–в–свой–день–рождения) ←
Член (отец (Джон), клуб–родившихся–в–свой–день–рождения
Джекил = отец (Джон),
клуб–родившихся–в–свой–день–рождения =
клуб–родившихся–в–свой–день–рождения

Джекил = отец (Джон) ← Хайд = Хайд,
Джекил = Хайд,
отец (Джон) = Хайд

Клауза J8 для доказательства несовместности в этом примере не понадобилась.

Формирование решений значительно упрощается, если индивиды имеют только одно имя (различные свободные от переменных термы именуют собой различные индивиды).

Тогда для описания *единственной* ситуации, при которой два индивида одинаковы (т.е. когда они имеют одно имя) нужна только одна аксиома

E1 $x = x \leftarrow$

Для каждой пары различных свободных от переменных термов s и t требуется аксиома вида

D Разл (s, t) ←

для указания единственной ситуации, когда индивиды различны (т.е., когда они имеют различные имена). Таких аксиом бесконечно много. Но если имеется конечное множество клауз S, то бесконечное множество аксиом D может быть заменено конечным множеством клауз

D1 Разл (a, b) ←

для каждой пары различных констант a и b в S

D2 Разл (a, $f(x_1, \ldots, x_m)$) ←
D3 Разл ($f(x_1, \ldots, x_m)$, a) ←

для каждой константы a и функционального символа f в S

D4 Разл ($f(x_1, \ldots, x_m)$, $g(y_1, \ldots, y_n)$) ←

для каждой пары различных функциональных символов f и g в S

D5 Разл ($f(x_1, \ldots, x_m)$, $f(y_1, \ldots, y_m)$) ← Разл (x_i, y_i)

для каждого функционального символа f в S и i-го аргумента f

Разл (x, y) — это то же самое, что не (x = y)

Это можно описать так:

Разл (x, y) если—и—только—если не (x = y), т.е.

D*1 Разл (x, y) ← не (x = y)
D*2 не (x = y) ← Разл (x, y)

в "стандартной форме" логики или

Разл (x, y), x = y ←
← Разл (x, y), x = y

в клаузальной форме логики. Но можно дать и другую интерпретацию предложения

Разл (x, y) только—если не (x = y)

которая отличается от D*2:

D* D*1 описывает единственную ситуацию, для которой заключение Разл (x, y) имеет место

D* содержит высказывание о D*1. Это предложение *мета-языка,* говорящее об индивидах, которые являются предложениями *объектного языка.* Связь между объектным языком, на котором формулируются предложения, и мета-языком, на котором говорят об этих предложениях, обсуждается в гл. 11 и 12.

Для упрощения дальнейшего изложения мы будем везде, где это возможно, придавать индивидами уникальные имена, используя предикатные символы равенства и Разл только в посылках клауз, за исключением их "определений":

E1 x = x ←

и

D Разл (s, t) ←

для каждой пары свободных от переменных термов s и t

На практике отношение Разл задается более эффективными средствами.

Упражнения

1. Выразить следующие предложения на языке клауз Хорна (некоторые из них неоднозначны):

а) Каждому кто-то нравится

б) Каждому нравится каждый

в) Кому-то нравится каждый

г) Никому не нравятся все

д) Никому не нравится некто

е) Кому-то не нравится ни кто

ж) Джон и Мери нравятся сами себе

з) Преподаватель счастлив, если он не входит ни в одну комиссию (Сначала это предложение надо преобразовать к такому виду: не имеет места случай, когда преподаватель счастлив и входит в некоторую комиссию)

и) Каждому, кто знает что-нибудь о логике, логика нравится.

2. Во всех приведенных ниже фактах из посылок следуют заключения. Выразите посылки и отрицания заключений на языке клауз так, чтобы результирующее множество клауз стало несовместным. Продемонстрируйте несовместность доказательством того, что полученное множество клауз не является истинным ни в одной интерпретации.

а) П о с ы л к а. Имеется некий индивидуум, являющийся любящим родителем каждого.

З а к л ю ч е н и е. У каждого есть любящий родитель.

б) П о с ы л к и: Всем жителям Востока нравятся все жители Запада. Всем жителям Запада нравятся все те жители Востока, которым нравятся какие-либо жители Запада.

З а к л ю ч е н и е. Всем жителям Запада нравятся все жители Востока без исключения.

в) П о с ы л к и: Канарейки – это птицы. У птиц есть крылья.

З а к л ю ч е н и е: У канареек есть крылья.

г) П о с ы л к и: Нечто, несущее в себе черты чего-либо хорошего, хорошо само по себе. Нечто, несущее в себе черты чего-либо плохого, плохо само по себе. Война несет в себе черты мира и страдания. Мир хорош, а страдание плохо.

З а к л ю ч е н и е: Некоторые вещи сразу и хороши и плохи.

д) П о с ы л к и: x является членом cons (x, y)

x является членом cons (u, y) ,если x является членом y

З а к л ю ч е н и е: A является членом cons (C, cons (A, cons (C, nil)))

е) П о с ы л к а: Боб счастлив, если всем его студентам нравится логика.

З а к л ю ч е н и е: Боб счастлив, если у него нет студентов.

3. Слово "нравится" в упражнении 6 гл. 1 использовалось в двух различных смыслах.

Переделайте упражнение 6 так, чтобы было видно различие между понятиями

x нравится есть y

и

x нравится быть вместе с y.

Вы можете сделать это используя либо два полностью различных предикатных символа "нравится–1" и "нравится–2", либо единственный трехаргументный предикатный символ, одним из аргументов которого было бы имя события (еды) или состояния (пребывания вместе с).

4. Выразите на языке клауз информацию, представленную семантической сетью рис. 2.8 и поясняющими его предложениями:

Объект e′ является некиим действием, состоящем в передаче огня людям. Если властитель запрещает подобное действие, выполняемое одним

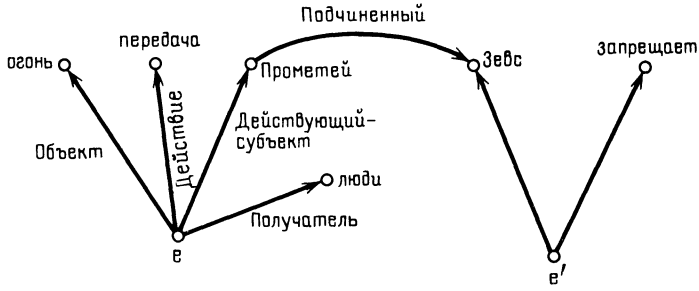


Рис. 2.8

из его подчиненных, то возникает другое событие, при котором властитель наказывает подчиненного.

5. Это упражнение основывается на концептуальном анализе действий, предложенном Шенком [Shank 1973, 1975]. Будем предполагать такую интерпретацию предикатных символов:

Действие (x, y)	x есть действие типа y
Обладает (x, y, u)	x обладает y в состоянии u
Действующий–субъект (x, y)	действующим лицом действия x является y
Объект (x, y)	объектом действия x является y
Источник (x, y)	источником действия x является y
Получатель (x, y)	приемником действия x является y

Пусть терм

Транс	именует тип всех действий передачи–получения в общем смысле
Дать	именует тип всех действий передачи
Получить	именует тип всех действий получения

результат (u) – это состояние непосредственно после действия u
перед (u) – это состояние непосредственно перед действием u

Выразите следующие предложения на языке клауз

а) В состоянии непосредственно после любого действия типа Транс приемник действия обладает объектом действия.

б) В состоянии непосредственно перед любым действием типа Транс источник действия обладает объектом действия.

в) Действие типа Транс есть действие типа передачи, если действующим лицом действия является источник действия.

г) Действие типа Транс есть действие типа получения, если действующим лицом действия является приемник действия.

6. Переделайте упражнение 5), используя равенство и функциональные символы. Пусть функциональные символы для этого будут такими:

действие (x)	именует тип действия x
действующий–субъект (x)	именует действующий субъект x
объект (x)	именует объект x
источник (x)	именует источник x
получатель (x)	именует получатель x

7. Пусть Родители (x, y, z) истинно, когда x есть отец, а y есть мать z. Постройте множество клауз, в котором единственным утверждением

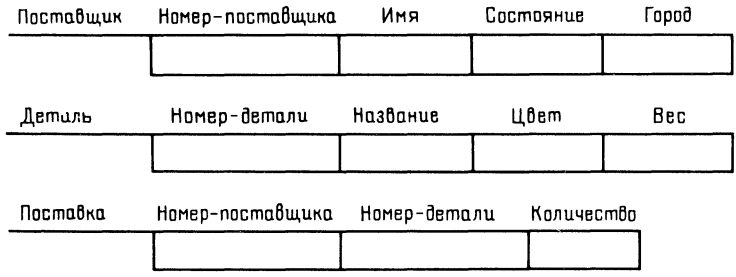


Рис. 2.9

со свободными от переменных было бы отношение Родители, но из которого бы следовали утверждения свободные от переменных F1 . . 8 из гл. 1.

8. Пусть данные заданы таблицами "Поставщик", "Деталь", "Поставка" (рис. 2.9).

Сформулируйте приведенные ниже запросы на языке клауз. Используйте при этом как бинарное, так и n-арное представление, принимая во внимание тот факт, что "Номер-поставщика" является ключом таблицы "Поставщик", а "Номер-детали" является ключом таблицы "Деталь". Предполагайте, что связь

$x < y$ (x меньше y)

уже задана.

а) Каковы номера поставщиков гаек?

б) Каковы имена поставщиков болтов?

в) Где находятся поставщики гаек и болтов?

г) Как называются детали, поставляемые поставщиком по имени Джон?

д) Каковы имена поставщиков, находящихся в Лондоне и поставляющих при этом гайки, весящие больше одной унции?

е) Каковы имена поставщиков, поставляющих сразу и гайки и болты?

ж) Каковы имена поставщиков, поставляющих гайки или болты?

3. НИСХОДЯЩИЕ И ВОСХОДЯЩИЕ ПРОЦЕДУРЫ ДОКАЗАТЕЛЬСТВ, ОСНОВАННЫЕ НА ИСПОЛЬЗОВАНИИ КЛАУЗ ХОРНА

Предварительные замечания

Проблема грамматического разбора, т.е. демонстрации того, что некоторый набор слов составляет предложение, построенное по заданным грамматическим правилам, может быть представлена в терминах математической логики как проблема доказательства несовместности некоторого множества клауз Хорна.

Различные процедуры грамматического разбора, определяющие тот факт, что строка является предложением, соответствуют различным процедурам доказательства несовместности. Процедуры нисходящего (сверху вниз) грамматического разбора соответствуют целеориентированным процедурам доказательства, которые действуют назад от заключений, используя импликации для сведения задач к подзадачам. Целью при этом становится сведение исходной задачи ко множеству подзадач, каждая из которых могла бы быть решена. Процедуры восходящего (снизу вверх) грамматического разбора соответствуют таким процедурам доказательства, которые работают в прямом направлении от исходного множества посылок, используя импликации для вывода заключений из посылок. Целью при этом оказывается вывод утверждений, которые прямо решают каждую из изначально поставленных задач.

Процедуры нисходящего и восходящего доказательств применяются для поиска решения любых задач. Нисходящий вывод представляет при этом *анализ* перехода от целей к подцелям; восходящий вывод состоит в *синтезе* новой информации из старой. В данной главе мы определим нисходящий и восходящий выводы только для случая клауз Хорна. Позднее мы расширим эти определения для нехорновских клауз и введем в рассмотрение системы, комбинирующие оба направления вывода.

Задача грамматического разбора

Следующее ниже описание проблемы синтаксического анализа базируется на описании, предложенном Фостером [Foster 1970] для формулировки Амареля.

Пусть задана грамматика и имеется строка слов

The slithy toves did gyre

(Вращались жиркие товы) *).

*) Строка из книги Льюиса Кэррола "Алиса в зазеркалье"; ее роль в данном случае аналогична роли знаменитой "Глокой куздры. . ." академика Л.В. Щербы. – *Примеч. пер.*

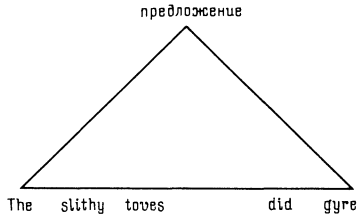


Рис. 3.1

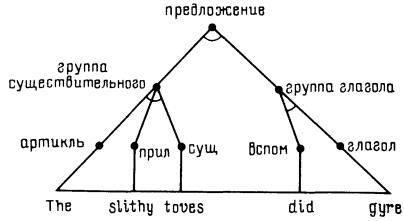


Рис. 3.2

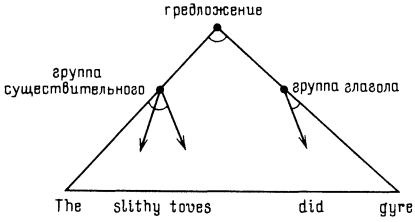


Рис. 3.3

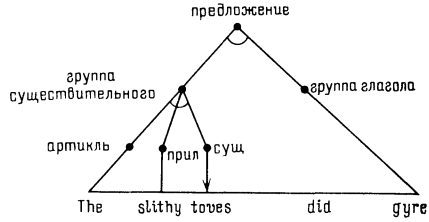


Рис. 3.4

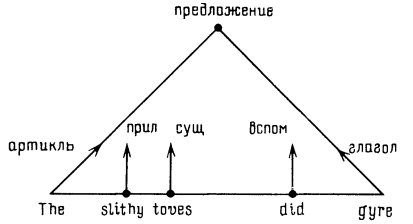


Рис. 3.5

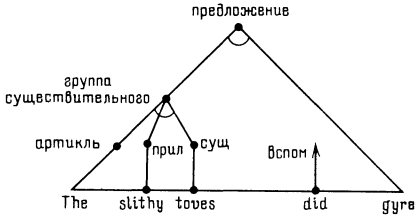


Рис. 3.6

Тогда задача состоит в том, чтобы показать, что эта строка является предложением. Это можно сделать, заполнив треугольник на рис. 3.1 деревом грамматического разбора, как это сделано на рис. 3.2.

Дерево грамматического разбора строится в соответствии с грамматикой. В нашем примере были использованы такие грамматические правила:

(1) Если за группой существительного следует группа глагола, то вместе они составляют предложение

(2) Если за артиклем следует прилагательное, а за прилагательным следует существительное, то вместе они составляют группу существительного

(3) Если за вспомогательным глаголом следует основной глагол, то вместе они составляют группу глагола

(4) "The" – артикль

(5) "slithy" – прилагательное

(6) "toves" – существительное

(7) "did" – вспомогательный глагол

(8) "gyre" – основной глагол

Различные способы заполнения треугольника и определяют различные процедуры грамматического разбора. *Нисходящие процедуры* задаются посредством заполнения треугольника сверху вниз. *Восходящие процедуры* получаются путем заполнения треугольника снизу вверх.

Нисходящая процедура может, как показано на рис. 3.3, порождать все ветви параллельно, а может, как показано на рис. 3.4, порождать одну ветвь целиком, обрабатывая их, например, слева направо.

Аналогично, восходящая процедура может обрабатывать все слова в исходной строке параллельно (рис. 3.5), а может обрабатывать и по одному слову целиком (рис. 3.6).

Треугольник разбора может заполняться и справа налево, и двунаправленно сверху вниз и снизу вверх, и даже от середины в стороны. Каждый систематический метод заполнения треугольника определяет свою процедуру грамматического разбора. Мы, однако, основное различие будем проводить между нисходящими и восходящими процедурами анализа.

Описание задачи грамматического разбора на языке логики предикатов

Имеется много способов описания задачи грамматического разбора на языке логики. Способ, который будет описан здесь, обладает тем свойством, что различные процедуры синтаксического анализа будут соответствовать различным процедурам доказательства при одном и том же описании.

Представим себе исходную строку в виде графа (рис. 3.7). Вершины графа поместим между соседними словами исходной строки, а также в ее начале и конце. Будем считать слова метками на дугах, связывающих

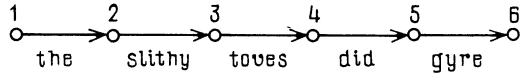


Рис. 3.7

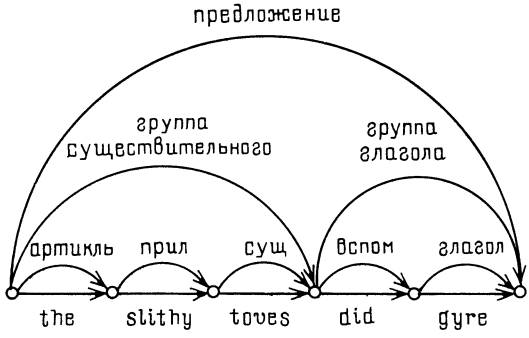


Рис. 3.8

вершины. Вершины, в свою очередь, произвольно пронумерованы значениями 1. . 6. Будем считать, что эта нумерация не задает никакого порядка во множестве вершин.

Правила грамматики могут быть представлены как высказывания о помеченных графах, например,

> Если имеется путь из вершины x в вершину y,
> отмеченный словом "the", то этот путь из x в y также отмечается словом "артикль", т.е.
>
> Арт(x, y) ← the(x, y).
>
> Если имеется путь из вершины x в вершину u, отмеченный словом "артикль", а также путь из u в v, отмеченный словом "прилагательное" и путь из v в y, отмеченный словом "существительное", то имеется путь из x в y, отмеченный как "группа существительного", т.е.
>
> Гс (x, y) ← Арт(x, u), Прил(u, v), Сущ(v, y)

Анализ исходной строки может быть описан графом, помеченным в соответствии с грамматическими правилами и включающим в себя путь, помеченный как "предложение" и ведущий из начала исходной строки в ее конец (рис. 3.8).

Исходный граф представляется в виде множества утверждений:

Parse 1 the (1, 2) ←
Parse 2 slithy (2, 3) ←
Parse 3 toves (3, 4) ←
Parse 4 did (4,5) ←
Parse 5 gyre (5,6) ←

Грамматические правила задаются клаузами, содержащими переменные

Parse 6 Предл(x, y) ← Гс(x, z), Гг (z, y)
Parse 7 Гс(x, y) ← Арт (x, u), Прил (u, v), Сущ (v, y)
Parse 8 Гг (x, y) ← Вспом (x, z), Глаг (z, y)
Parse 9 Арт(x, y) ← the (x, y)
Parse 10 Прил(x, y) ← slithy (x, y)
Parse 11 Сущ (x, y) ← toves (x, y)
Parse 12 Вспом (x, y) ← did (x, y)
Parse 13 Глаг (x, y) ← gyre (x, y)

Эти и только эти грамматические правила нужны для проведения грамматического разбора исходной строки слов. В более общей постановке задачи нам надо было бы рассмотреть использование и других грамматических правил, например, таких

Parse 14 Гс (x, y) ← Арт (x, z), Сущ (z, y)
Parse 15 Гс(x, y) ← Сущ (x, y)
Parse 16 Гг(x, y) ← Глаг (x, y)
Parse 17 Арт (x, y) ← a (x, y)
Parse 18 Прил (x, y) ← brilling (x, y)
Parse 19 Сущ (x, y) ← wabe (x, y)
Parse 20 Глаг (x, y) ← gimble (x, y)

Для того, чтобы показать, что строка слов с первого по шестое является предложением, достаточно, например, показать, что отрицание цели

Parse 21 ← Предл (1, 6)

несовместно с Parse 1..20.

Восходящий вывод

Восходящий метод опровержения начинается с утверждений, имеющихся в исходном множестве клауз. Он использует импликации для обоснования вывода новых утверждений из старых и заканчивается образованием утверждения, которое очевидным образом противоречит отрицанию цели.

Графическое представление восходящего метода опровержения для Parse 1. . 21 показано на рис. 3.9. Оно похоже на перевернутое дерево грамматического разбора. В нем узлы помечаются новыми утверждениями. Для образования новых утверждений, отмечающих пучки дуг, ведущих от старых утверждений к новым, используется импликация.

Например, утверждение

Гс (1, 4) ←

получено из трех утверждений

Арт (1, 2) ←
Прил (2, 3) ←
Сущ (3, 4) ←

при помощи сопоставления с тремя условиями клаузы

Гс (x, y) ← Арт (x, u), Прил (u, v), Сущ (v, y)

Сопоставление выполняется при помощи поиска наиболее общей подстановки, в данном случае,

{x = 1, u = 2, v = 3, v = 4}

которая делает утверждения полностью идентичными условиям.

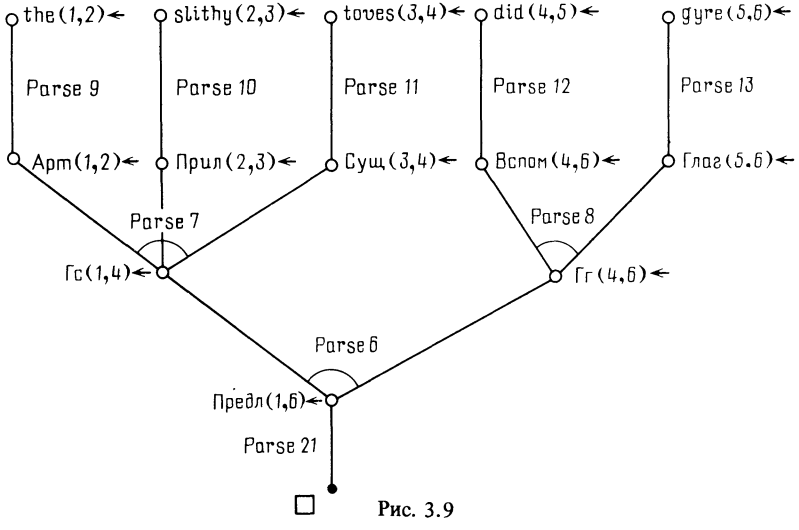


Рис. 3.9

В общем случае один шаг восходящего вывода выполняет сопоставление (наиболее общим из возможных способов) некоторого количества утверждений с условиями клаузы и образует новые утверждения. Каждое такое новое утверждение состоит из заключения клаузы, означенного согласующей подстановкой. Если клауза является отрицанием (т.е. она не имеет заключения), то образуемая клауза является пустой. Более точное определение этого процесса можно найти в конце этой главы.

Восходящий вывод является обобщением комбинации означивания и классического правила *modus ponens*

из A ← и B ← A выводится B ←

При этом означивание ограничивается минимумом, необходимым для такого сопоставления утверждения с условиями, чтоб modus ponens мог быть выполненным.

Нисходящий вывод

Нисходящий метод опровержения начинает работать с отрицания в исходном множестве клауз. В процессе работы в нем используются импликации и утверждения для образования новых отрицаний из старых и заканчивается в момент образования пустой клаузы.

Графическое представление нисходящего метода опровержения Parse1. . 21 дано на рис. 3.10. Вершины графа помечены отрицаниями. Каждая дуга помечается исходной клаузой, которая используется для образования отрицания в конце этой дуги. Выбираемые атомы выделяются курсивом.

Начав процесс с исходного отрицания

← Предл (1, 6)

нисходящий вывод согласовывает условия отрицания с заключением

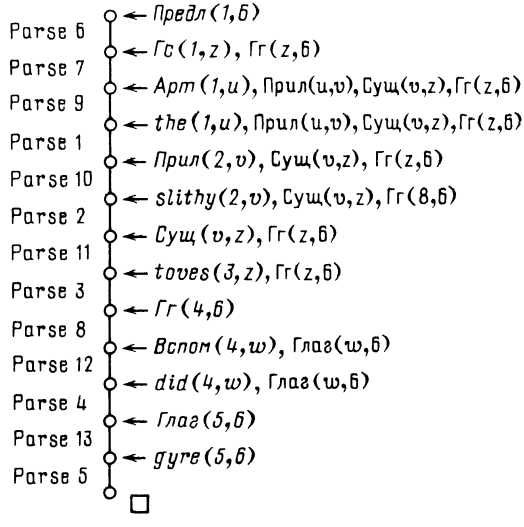


Рис. 3.10

импликации

Предл (x, y) ← Гс(x, z), Гг (z, y),

образуя при этом новое отрицание

← Гс(1, z), Гг(z, 6),

которое состоит из условий исходной клаузы, означенных согласующей подстановкой

{x = 1, y = 6}.

Этот шаг формализует правдоподобное рассуждение о том, что

> если не существует предложения от 1 к 6, то не найдется такое z, для которого существует группа существительного от 1 к z, за которой следует группа глагола от z к 6.

Тот же самый шаг вывода может быть интерпретирован с точки зрения поиска решений так:

> Цель показа того, что имеется предложение от 1 к 6 может быть достигнута, если можно найти такое z, что достижимыми подцелями этого показа станут группа существительного от 1 к z и группа глагола от z к 6.

В терминологии теории поиска решений можно сказать, что исходная цель сводится к двум новым подцелям.

В общем случае нисходящий вывод включает в себя согласование выбранных условий с заключением импликации и получение нового отрицания путем замены выбранных условий условиями импликации и применения согласующей подстановки. Если импликация является утверждением, не имеющим условия, то выбранные условия просто удаляются, и применяется согласующая подстановка. Если, вдобавок, выбранное условие является единственным, то образуемая клауза является пустой. В терминологии методов поиска решений отрицание интерпретируется как набор целей. Нисходящий вывод замещает выбранную цель (в контексте этого набора целей) множеством подцелей. Точное определение нисходящего процесса вывода дано в конце этой главы, в то время как интерпретация поиска решения рассматривается в следующей главе.

Нисходящий вывод является обобщением комбинации означивания и *modus tollens*

из не–A и A ← B выводится не–B

Означивание при этом ограничивается минимумом, необходимым для того, чтобы стало применимым правило modus tollens

Различные нисходящие процедуры опровержения определяются выбором для применения нисходящего вывода различных атомов в отрицаниях.

Например, на клаузу Parse 8 можно было бы подействовать отрицанием

← Гс(1, z), *Гг(z, 6)*

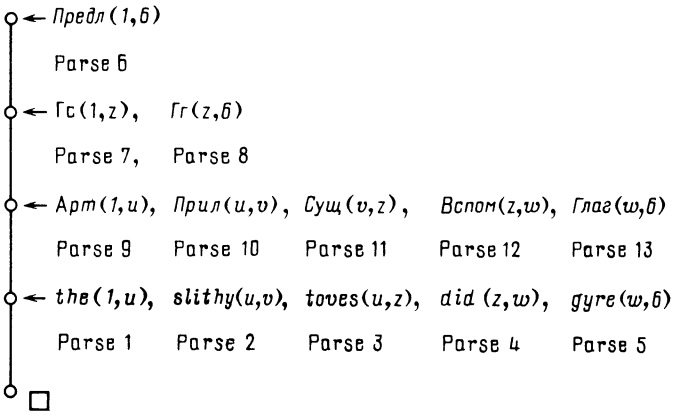


Рис. 3.11

для образования нового отрицания

$$\leftarrow \text{Гс}\,(1, z), \text{Вспом}\,(z, u), \text{Глаг}\,(u, б)$$

При этом, если имеется опровержение для одного способа выбора атомов, то имеется опровержение и для любого другого способа выбора.

Возможно также (как и в восходящем выводе) выбирать все условия в отрицании одновременно. Рисунок 3.11 иллюстрирует такое нисходящее параллельное опровержение. На рисунке выбранное условие является именем клаузы, использованной для образования следующего отрицания.

Такая формулировка проблемы синтаксического анализа была получена Алэном Колмероэ совместно с автором этих строк при переложении Q–систем Колмероэ [Colmerauer 1973] на язык логики. Здесь следует подчеркнуть, что в то время как Q-система базируется на нисходящей процедуре синтаксического анализа, язык клауз Хорна оказывается более общим и может быть использован как в нисходящем, так и в восходящем случае.

Хотя в приведенном примере использовались только контекстно-свободные правила грамматики, нетрудно распространить наше представление для выражения контекстно–зависимых грамматических свойств, а также произвольных преобразующих (переписывающих) систем [Chomsky 1957].

Пример "Родственные связи"

Понятия нисходящего и восходящего выводов применимы к любому множеству клауз Хорна. Поэтому клаузы F1. . 19 из гл. 1, которыми определяются родственные связи древнегреческих богов, позволяют нам построить еще один пример.

Если даны клаузы F1..19, то задача доказательства того, что Зевс является дедом–или–бабушкой Гармонии, может быть сформулирована как проблема заполнения треугольника на рис. 3.12 деревом вывода рис. 3.13.

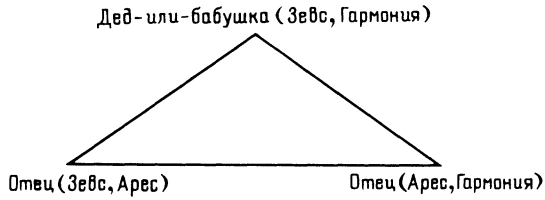


Рис. 3.12

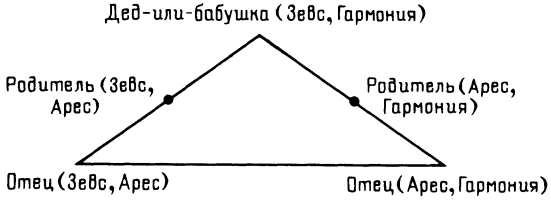


Рис. 3.13

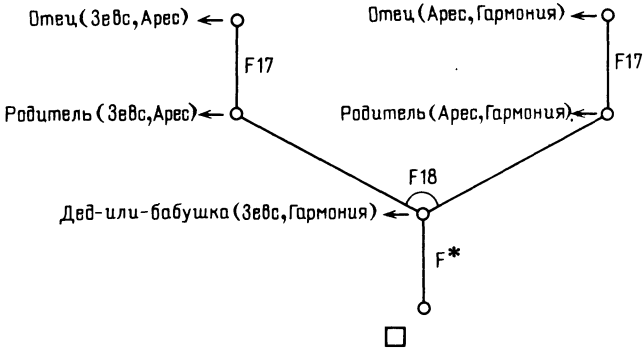


Рис. 3.14. Восходящее опровержение F^* и $F1 \ldots 19$

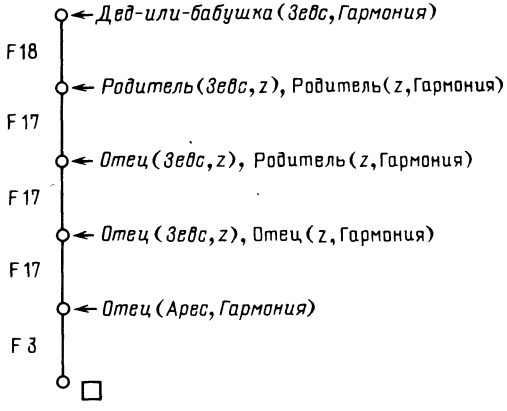


Рис. 3.15. Нисходящее опровержение F^* и $F1 \ldots 19$

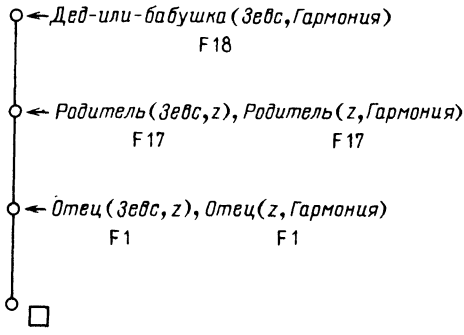


Рис. 3.16. Параллельное нисходящее опровержение F^* и $F1 \ldots 19$

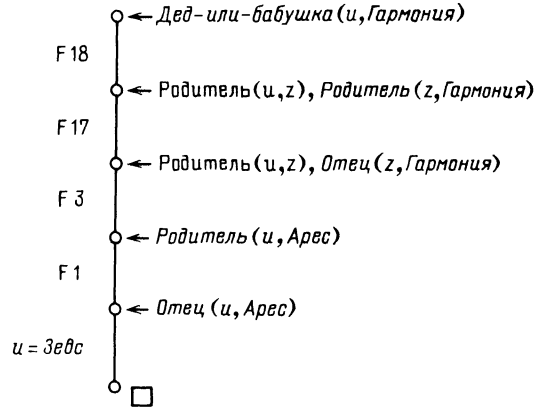


Рис. 3.17

На языке логики клауз задача состоит в том, чтобы доказать, что отрицание

F* ← Дед–или–бабушка (Зевс, Гармония)

несовместимо с клаузами F1..19. Рисунки 3.14, 3.15, 3.16 иллюстрируют процесс восходящего, нисходящего и параллельного нисходящего опровержений соответственно.

Поскольку операция согласования атомарных формул является весьма общей, нисходящая и восходящая процедуры вывода могут быть использованы не только для доказательства того, что Зевс – дед или бабушка Гармонии, но и для отыскания деда–или–бабушки Гармонии или для отыскания внуков Зевса. Это можно проиллюстрировать нисходящей процедурой опровержения, которая доказывает несовместность F1..19 и F*

F** ← Дед–или–бабушка (u, Гармония)

Дед–или–бабушка Гармонии, чье существование противоречит F**, может быть определен посредством анализа согласующих подстановок,

используемых в процессе опровержения. Последний шаг опровержения согласует переменную u из исходного отрицания с константным символом "Зевс" и определяет, что u = Зевс является дедом–или–бабушкой Гармонии (рис. 3.17).

Отметьте, что первый шаг опровержения согласует посылку

Дед–или–бабушка (u, Гармония)

с заключением

Дед–или–бабушка (x, y)

Нисходящий вывод использует наиболее общую подстановку, делающую два атома идентичными; в данном случае эта подстановка есть

{ x = u, y = Гармония }

Любую менее общую подстановку,
например

{ x = Арес, u = Арес, y = Гармония }

или

{ x = Зевс, u = Зевс, y = Гармония }

также делающую оба атома идентичными, при этом нет необходимости использовать.

Если даны любые два атома, то любые (наиболее общие) согласующие подстановки различаются только по именам, которые они придают переменным, а во всем остальном эквивалентны. Следовательно необходимым оказывается использование только одной из них на каждом шаге вывода. Например, согласующая подстановка

{ u = x, y = Гармония }

эквивалентна той подстановке, которую, мы использовали на первом шаге вышеприведенного процесса опровержения. Результатом только что приведенной подстановки является эквивалентное отрицание

← Родитель (x, z), Родитель (z, Гармония),

являющееся вариантом приведенного на рис. 3.16.

Возможность ограничения означивания для порождения наиболее общих согласующих подстановок была изучена Правицем [Pravitz 1960] и развита Робинсоном [Robinson 1965a], включившим ее в правило резолюций (см. гл. 8), обобщающее нисходящее и восходящее правила вывода, рассмотренные в настоящей главе. *Алгоритм унификации* для согласования атомарных формул был предметом весьма многочисленных исследований [Robinson, 1971], [Paterson, Wegman 1976], [Martelli, Montanari 1977].

Правила вывода и стратегии поиска

Правила вывода суть блоки, из которых строятся процедуры доказательства. *Процедура доказательства* — это строгий метод демонстрации того, что из множества посылок следует заключение. Процедуры доказательства в клаузальной форме логики являются *процедурами опровержения,* которые показывают, что из посылок следуют заключения, демонстрируя несовместность этих посылок и отрицания заключения.

Правила вывода формируют элементарные шаги, составляющие доказательство. Всевозможные способы применения правил вывода как к изначально заданному множеству клауз, так и к выведенным из него клаузам, определяют *пространство поиска* для множества клауз. Задание некоторой последовательной *поисковой стратегии* для генерирования клауз в пространстве поиска определяет процедуру доказательства.

Для нисходящего вывода характерны древовидные пространства поиска. Отдельные узлы такого пространства поиска помечаются отрицаниями, содержащими выбранные условия. Для каждой исходной клаузы, заключение которой согласуется с выбранным условием, имеется дуга, помеченная этой исходной клаузой и ведущая к отрицанию, полученному применением нисходящего вывода. Опровержение представляет собой путь в пространстве поиска, который соединяет исходное отрицание с пустой клаузой □

На рисунке 3.18 показано пространство нисходящего поиска для задачи нахождения деда–или–бабушки Гармонии. Для экономии места были использованы сокращения вида

Га для Гармонии

Ге для Геры

Р для Родителя и тому подобное для константных и предикатных символов, а исходные клаузы, помечающие дуги, указаны не были.

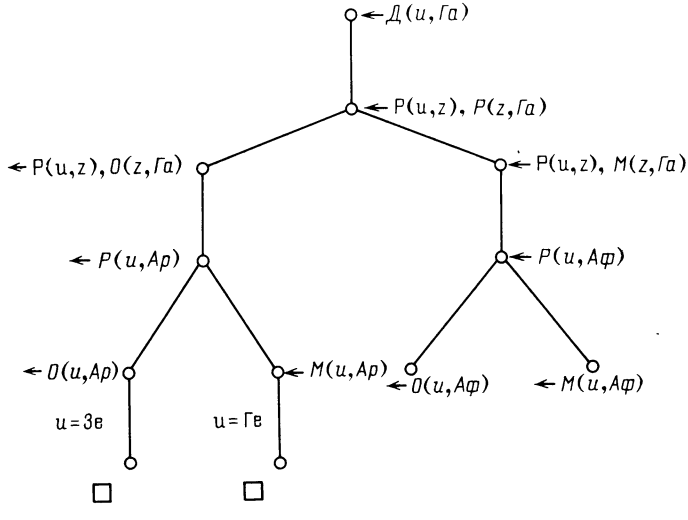


Рис. 3.18

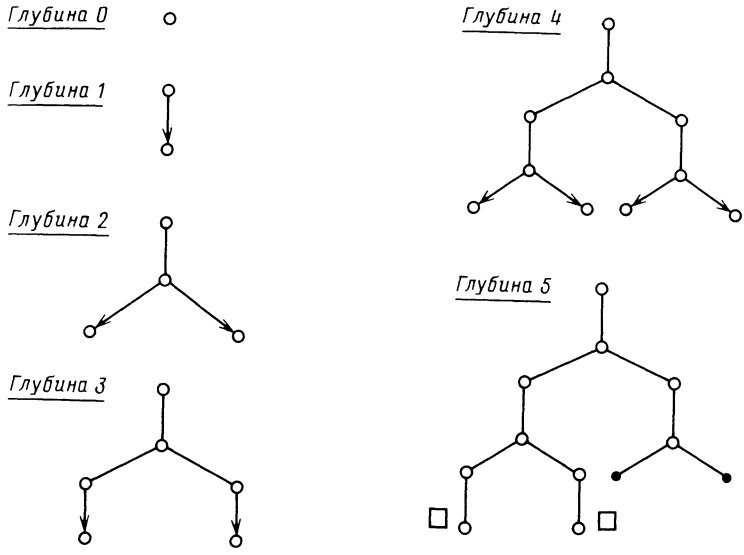


Рис. 3.19

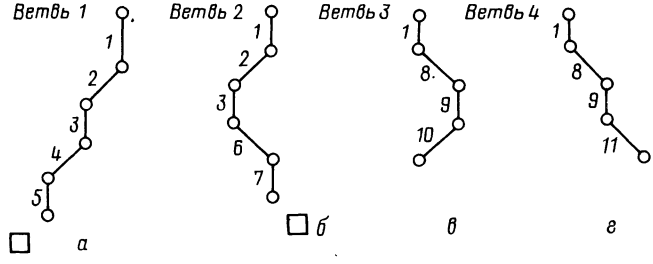


Рис. 3.20

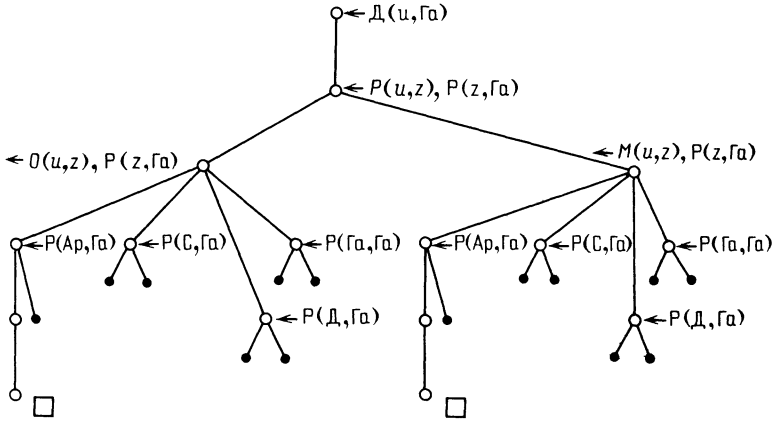


Рис. 3.21

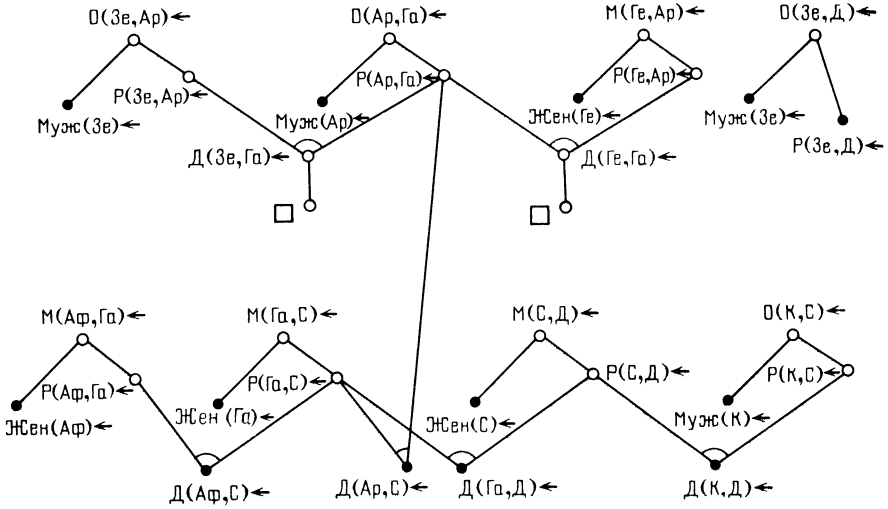


Рис. 3.22

Заштрихованные вершины дерева поиска содержат выбранные условия, которые не согласуются с заключениями ни одной из исходных клауз.

Это пространство поиска конечно, и может быть полностью просмотрено за конечный период времени. Двумя главными видами поисковых стратегий здесь являются поиск вширь и поиск вглубь. *Поиск вширь* проверяет все ветви дерева, находящиеся на одном и том же уровне в n шагов от корня дерева до того, как начать проверку на следующем уровне в n + 1 шаг от корня дерева. Графически проверка поискового пространства посредством поиска вширь может быть изображена на рис. 3.19 в виде такой последовательности.

Поиск вглубь проверяет за один раз одну ветвь поискового пространства. Когда он достигает какой-либо из оконечных вершин дерева, выполняется *возврат* (бэктрекинг) и происходит попытка проверки альтернативной ветви, возможно более близкой к найденной до этого оконечной вершине (рис. 3.20).

На рисунке номера рядом с ребрами показывают последовательность, в которой эти ребра проверяются. Здесь первая ветвь уже содержит решение задачи. Если требуется только одно решение, то оставшуюся часть поискового пространства порождать не нужно. Полное пространство поиска, тем не менее, должно быть порождено, если требуются все решения. В нашем случае имеется два пути опровержения, каждый из которых определяет свой ответ на запрос о том,

Каково значение u, являющегося дедом–или–бабушкой Гармонии?

u = Зевс u = Гера

На пространство поиска нисходящего вывода воздействуют выборы условий в отрицаниях. В вышеприведенном поисковом пространстве условия были специально выбраны с тем, чтобы минимизировать размер

поискового пространства. В пространстве поиска, приведенном на рис. 3.21 выбор условий максимизирует его размеры.

Оба пространства нисходящего поиска *полны* в том смысле, что они содержат опровержения, если исходное множество клауз несовместно. Достаточно таким образом, рассмотреть либо одно поисковое пространство, либо другое. В общем случае, при прочих равных условиях, чем обширнее поисковое пространство, тем труднее реализовать в нем стратегию поиска опровержения.

При интерпретации нисходящего вывода как метода принятия решений выбор условия в отрицании оказывается ни чем иным, как выбором способа достижения одной из множества подцелей. Такой подход является одним из самых важных аспектов поиска решений задач, и он станет главной темой следующих двух глав.

Структура пространства восходящего поиска более сложна, нежели структура пространства нисходящего поиска. Следовательно, поиск в нем затруднен. На рисунке 3.22 изображено пространство поиска восходящего вывода для примера родственных связей. Вершины помечены утверждениями. Утверждения, согласующиеся с условиями исходной клаузы, связаны с новым утверждением, полученным посредством восходящего вывода, при помощи пучка дуг. Исходная клауза, которая должна помечать этот пучок, не указана для экономии места. Заштрихованные вершины обозначают утверждения, к которым неприменимы шаги восходящего вывода. На рисунке вдобавок к введенным ранее сокращениям М обозначает предикат "Мужчина", а Ж – предикат "Женщина".

На рисунке нет исходных утверждений,

Божество (Зевс) ← и Царица (Гармония) ←

поскольку они не согласуются ни с какими условиями.

Отметим, что утверждение

Мужчина (Зевс) ←

выводится двумя различными способами; оба способа порождают две вершины, помеченные одним и тем же утверждением. В следующей главе мы рассмотрим такие представления поисковых пространств, в которых различные вершины помечаются различными клаузами.

На практике, кроме поиска вширь, к пространствам восходящего поиска применялось мало других стратегий. Как и в пространствах нисходящего поиска, *поиск вширь* рассматривает все утверждения глубины n до того, как породить хотя бы одно утверждение глубины n + 1. *Глубина* утверждения вычисляется как увеличенный на единицу максимум глубин порождающих его утверждений.

Поисковые стратегии являются важной частью всех систем поиска решений задач; они рассматриваются детальнее в следующей главе.

Бесконечные поисковые пространства: натуральные числа

Оба рассмотренных поисковых пространства, как для задачи грамматического разбора, так и для задачи о родственных связях, конечны. Бесконечные поисковые пространства возникают, когда имеются клаузы,

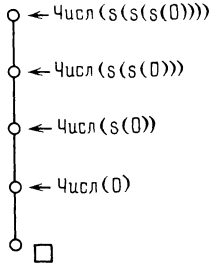


Рис. 3.23

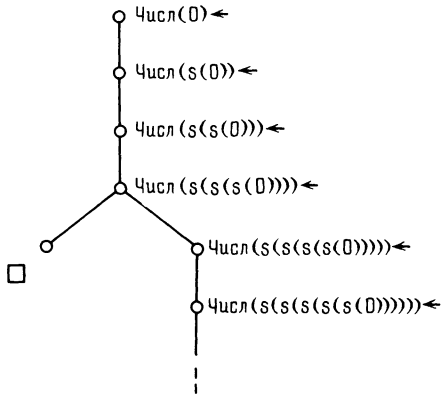


Рис. 3.24

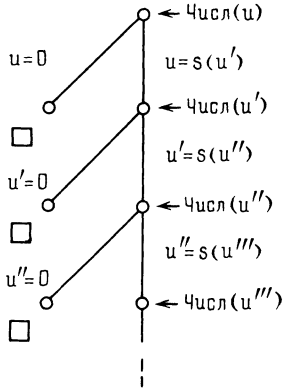


Рис. 3.25

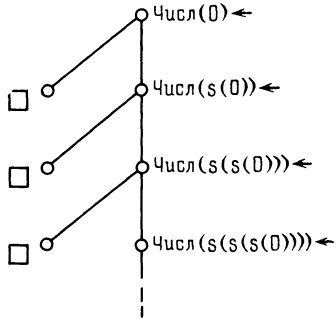


Рис. 3.26

содержащие функциональные символы. Простым примером этого является определение натуральных чисел, использующее функциональный символ следования:

Числ (0) ←
Числ (s(x)) ← Числ (x)

Пусть задача состоит в том, чтобы показать, что три есть число:

← Числ (s (s (s (0))))

Пространство нисходящего поиска (рис. 3.23) конечно и содержит только решение задачи. Пространство восходящего поиска (рис. 3.24), однако, бесконечно.

Тем не менее, для решения задачи нахождения числа оба поисковых пространства бесконечны. Более того, оба эти пространства содержат бесконечное число решений (рис. 3.25 и 3.26).

Каждое ребро пространства нисходящего поиска помечено частью согласующей подстановки, нужной для нахождения числа u, существование которого отрицается начальной постановкой задачи.

Когда поисковые пространства бесконечны, стратегии поиска вглубь могут повести поиск по ложной ветви поискового пространства и, таким образом, могут не привести к достижению опровержения. В нашем примере это происходит в пространстве нисходящего поиска, если клауза

$$\text{Числ}\ (s\ (x)) \leftarrow \text{Числ}\ (x)$$

используется перед утверждением

$$\text{Числ}\ (0) \leftarrow$$

а в пространстве восходящего поиска, если

$$\text{Числ}\ (s\ (x)) \leftarrow \text{Числ}\ (x)$$

используется перед отрицанием

$$\leftarrow \text{Числ}\ (u)$$

Отметим, что завершение процедуры доказательства гарантируется не только тем, что пространство поиска должно быть полным, но и тем, что стратегия поиска должна быть *исчерпывающей:* любая вершина должна быть достижима за конечное число шагов.

Определения

Некоторые понятия, введенные в этой главе, сейчас будут определены более точно.

Пусть S является множеством клауз Хорна и пусть имеется стратегия выбора, которая выделяет условия из любого отрицания. Тогда последовательность отрицаний

$$C_1, C_2, \ldots C_n$$

есть *нисходящий вывод* C_n из S, если

1) первая клауза C_1 принадлежит S и

2) каждое отрицание в последовательности, за исключением первого, получается из предыдущего отрицания при помощи правила нисходящего вывода, использующего некоторую клаузу из S.

Вывод пустой клаузы из S называется *опровержением* S

Если дано отрицание

$$\leftarrow A_1, \ldots, A_{i-1}, A_i, A_{i+1}, \ldots, A_m, \quad m \geqslant 1$$

в котором имеется атом A_i, и если дана импликация

$$B \leftarrow B_1, \ldots, B_n, \quad n \geqslant 0$$

которая не имеет общих переменных с отрицанием, то новое отрицание может быть получено при помощи *правила нисходящего вывода* если указанный атом A_i согласуется с заключением в импликации.

Новое *отрицание* состоит из всех условий исходного отрицания (за исключением условия A_i) и из всех условий импликации; при этом ко

всем к ним применена согласующая подстановка θ:

$$\leftarrow (A_1, \ldots, A_{i-1}, B_1, \ldots, B_n, A_{i+1}, \ldots, A_m)\,\theta$$

Если отрицание и импликация содержат общие переменные, то до применения правила нисходящего вывода они должны быть переименованы, в результате чего должны получиться эквивалентные клаузы, не имеющие общих переменных. Так, применяя нисходящий вывод к отрицанию

$$\leftarrow Гс\ (y, u),\ Гг\ (u, z)$$

и используя клаузу

$$Гс(x, y) \leftarrow Арт\ (x, u), Сущ\ (u, y)$$

необходимо сначала переименовать переменные, взяв, например, другой вариант импликации

$$Гс(x', y') \leftarrow Арт(x', u'), Сущ(u', y')$$

За счет этого получается новое отрицание

$$\leftarrow Арт\ (y, u'), Сущ\ (u', u), Гг\ (u, z)$$

причем согласующая подстановка есть

$$\{x' = y,\ y' = u\}$$

В общем случае в отрицании может быть выбрано любое условие. Стратегию выбора можно считать стратегией типа LIFO*), если выбираемое условие является одним из последних условий, включенных в отрицание к примеру, одним из условий

$$B_1\theta, \ldots, B_n\,\theta$$

в новом отрицании

$$\leftarrow A_1\theta, \ldots, A_{i-1}\theta, B_1\theta, \ldots, B_n\theta, A_{i+1}\theta, \ldots, A_m\theta$$

Нисходящий вывод может быть представлен в виде графа, в котором с каждой вершиной сопоставляется отрицание C_j в выводе, а ребра, помеченные импликацией, использованной на шаге вывода, проводятся из вершины к следующему отрицанию C_{j+1}.

Что касается определения согласующей подстановки, то в нем нуждаются как нисходящий, так и восходящий выводы, и поэтому оно будет представлено после последнего пункта определения восходящего вывода.

Для восходящего вывода удобным оказывается сначала привести определения графического представления. Пусть S есть множество клауз Хорна. Граф D с вершинами, помеченными утверждениями, является *восходящим выводом* клаузы C из S, если

1) D состоит из единственной вершины, помеченной C, которая принадлежит S, и является или утверждением, или пустой клаузой, либо

---

*) Сокращение английской фразы *Last in first out* (последним вошел, первым ушел), служащее общепринятым названием одной из стратегий выбора или обслуживания. – *Примеч. пер.*

2) D состоит из подвыводов:
D_1 для $A_1 \leftarrow$ из S,
D_2 для $A_2 \leftarrow$ из S,
. . . . . .
D_m для $A_m \leftarrow$ из S,

корневые вершины которых связаны ребрами с новой вершиной, помеченной C и полученной из $A_1 \leftarrow$, $A_2 \leftarrow$, . . . , $A_m \leftarrow$ при помощи восходящего правила вывода за счет использования клаузы C' из S; это показано на рис. 3.27.

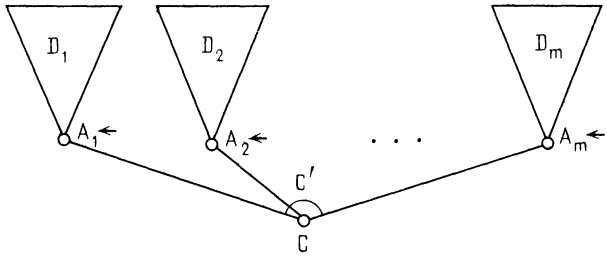


Рис. 3.27

Клауза C' помечает пучок ребер, связанный с шагом вывода.

Восходящий вывод клаузы C из m утверждений

$$A_1 \leftarrow,\ A_2 \leftarrow, \ldots,\ A_m \leftarrow$$

с использованием клаузы C' удобно определить при помощи разложения вывода в последовательность m более простых шагов вывода. Допустим, что C' имеет вид

$$B \leftarrow B_1, B_2, \ldots, B_m$$

или

$$\leftarrow B_1, B_2, \ldots, B_m$$

Тогда клауза C получается при помощи *восходящего вывода* с использованием C' из

$$A_1 \leftarrow,\ A_2 \leftarrow, \ldots,\ A_m \leftarrow$$

1) посредством выбора условия, скажем B_1, из C', согласования его с утверждением, скажем $A_1 \leftarrow$, и вывода промежуточной клаузы C''

$$(B \leftarrow B_2, \ldots, B_m)\theta$$

или

$$(\leftarrow B_2, \ldots, B_m)\theta$$

где θ — согласующая подстановка

и

2) посредством получения C при помощи восходящего вывода из $A_2 \leftarrow, \ldots, A_m \leftarrow$

с использованием клаузы C''

3) если m = 1, то $C = C''$

4) на шаге (1) переменные в $A_1 \leftarrow$ должны отличаться от переменных в C'. При необходимости они должны быть переименованы для достижения различия.

Можно показать, что условия в C' допустимо выбирать в произвольном порядке, и это не повлияет на окончательно выводимую клаузу

Утверждения $A_1 \leftarrow, A_2 \leftarrow, \ldots, A_m \leftarrow$, к которым применяется восходящий вывод, не обязательно должны быть различными. Например, утверждение

Друзья (Нарцисс, Нарцисс) ←

можно получить за один шаг восходящего вывода из двух копий утверждения

Нравится (Нарцисс, Нарцисс) ←

с использованием клаузы

Друзья (x, y) ← Нравится (x, y), Нравится (y, x).

Подстановка и согласование

Нам осталось определить понятия подстановки и согласования.

Подстановка

$\{x_1 = t_1, \ldots, x_m = t_m\}$

есть множество *подстановок – компонентов вида*

$x_i = t_i$

где x_i есть переменная, а t_i – терм, отличный от x_i

В различных подстановках–компонентах

$x_i = t_i$ и $x_j = t_j$

различны и переменные x_i и x_j. Поэтому подстановку можно рассматривать как функцию, отображающую переменные на множество термов. Если E – *выражение* (терм, атом или клауза), то *результатом* применения подстановки

$\theta = \{x_1 = t_1, \ldots, x_m = t_m\}$

является новое выражение $E\theta$, которое идентично E за исключением того, что для каждого компонента $x_i = t_i$, принадлежащего θ и для любого вхождения x_i в E, $E\theta$ содержит вместо этого вхождения t_i.

Про новое выражение $E\theta$ говорят, что это *пример* E

Подстановка σ *унифицирует* два выражения E_1 и E_2, если она делает их идентичными, т.е.

$E_1\sigma = E_2\sigma$

$E_1\sigma$ является в этом случае *общим примером* E_1 и E_2, определенным посредством σ. Подстановка θ *согласует* E_1 и E_2 (есть *наиболее общий унификатор* E_1 и E_2) если

1) θ унифицирует E_1 и E_2

и

2) общий пример

$E_1\sigma$

определенный посредством σ – любого другого унификатора E_1 и E_2, является примером общего примера

$E_1\theta$

заданного посредством θ. Поэтому

$$E_1\sigma = (E_1\theta)\lambda$$

для некоторой подстановки λ.

Каждая пара выражений, которая может быть унифицирована, также может быть и согласована. Более того, все согласующие подстановки эквивалентны в том смысле, что определяемые ими общие примеры являются вариантами друг друга.

Корректность и полнота систем логического вывода

Система правил вывода является *корректной* (или *непротиворечивой*), если любое множество клауз, обладающее опровержением в соответствии с этими правилами вывода, несовместно. Такая система является *полной*, если любое несовместное множество клауз имеет опровержение. Понятия корректности и полноты связывают семантику с той частью синтаксиса, которая относится к теории доказательств. Для системы логического вывода, являющейся одновременно корректной и полной, семантическое понятие несовместности совпадает с теоретико-доказательственным понятием опровержимости. Легко проверяется корректность нисходящего и восходящего способов вывода.

Восходящий способ вывода является специальным случаем правила гиперрезолюции, полностью определенного и обоснованного Робинсоном [Robinson 1965b]. Нисходящий способ вывода является одной из форм модели исключения правил, введенной и доказанной Лавлэндом [Loveland 1968, 1969]. Как и гиперрезолюция, модель исключения применима к произвольным множествам клауз. Однако в обоих случаях при наличии нехорновских клауз для достижения полноты требуется добавочное правило вывода, а именно, правило факторизации, обсуждаемое в гл. 7.

В разное время получили развитие многочисленные формы нисходящего вывода, как-то: линейная резолюция [Loveland 1970], [Luckham 1970], упорядоченная линейная резолюция [Reiter 1971], SL – резолюция [Kowalski, Kuehner 1971], G – дедукция [Michie 1972], взаимная графовая резолюция [Sickel 1976] и аналитическая резолюция [Brand 1976].

Линейная резолюция не накладывает никаких ограничений на выбор атомов при нисходящем выводе. Если дано отрицание, содержащее n атомов, то потенциально оно может привести к n! избыточным последовательностям, в которых эти атомы могут быть выбраны. Остальные системы, в том числе и модель исключения, обеспечивают LIFO – процедуры выбора. Необходимость выбора атомов более гибким способом будет обсуждаться в последующих двух главах. Полнота систем нисходящего вывода, использующих произвольные процедуры выбора, была продемонстрирована многими авторами, в том числе, [Brown 1973] и [Hill 1974].

Нисходящий и восходящий выводы представляют собой специальные случаи правила резолюций [Robinson 1965a]. Система смешанного

нисходяще–восходящего вывода для клауз Хорна была описана Кюнером [Kuehner 1972]. Процедура доказательства на базе графа соединений [Kowalski 1974], рассматриваемая в гл. 8, сочетает в себе оба способа вывода для нехорновских клауз. Системы, не основанные на методе резолюций и базирующиеся на стандартной, а не клаузальной форме логики разрабатывались Бледшоу [Bledsoe 1971, 1977] и его коллегами для приложений в области доказательства математических теорем. Эти системы сочетают в себе восходящий способ вывода в прямом направлении от посылок с нисходящим способом вывода в противоположном направлении от заключений.

Упражнения

1. Строку символов можно рассматривать как граф, вершинами которого служат межсимвольные промежутки (пробелы), а ребра помечены символами и связывают один пробел со следующим. Ребро, помеченное символом w и связывающее пробел x с пробелом y (рис. 3.28), может быть представлено посредством трехместного отношения

Conn (x, w, y)

Так, утверждения

Conn (4, D, 2) ←
Conn (2, A, 3) ←
Conn (3, D, f) ←

задают строку

D A D

в которой для пробелов выбраны произвольные имена

4, 2, 3, f

Некоторая строка называется *палиндромом,* если она читается одинаково как в прямом, так и в обратном направлении. Выразите приведенное ниже более точно выражающее это понятие определение (состоящее из трех пунктов) на языке клауз Хорна:

а) строка от пробела x до пробела y есть палиндром, если символ от x до x′ такой же, как символ от y′ до y, а строка от x′ до y′ есть палиндром*)

б) строка от x до y есть палиндром, если между ними имеется ровно один символ

в) строка от x до x – палиндром

Постройте нисходящий и восходящий способы проверки того, что строка DAD есть палиндром.

2. Пусть строки описываются посредством трехместного отношения Conn как и в упражнении 1)

---

*) x′ – сосед x, а y′ – сосед y. – *Примеч. пер.*

а) опишите с помощью клауз Хорна отношения

Тождественный (w , x, u, v), которое выполняется, когда строка от u до v состоит из w копий элемента x (рис. 3.29)

Допустимый (u, v), которое выполняется, если для некоторого i строка от u до v состоит из i копий символа a, затем i копий символа b и затем i копий символа c (рис. 3.30).

б) представьте полные пространства нисходящего и восходящего выводов для задачи доказательства того, что строка abc допустима; в случае

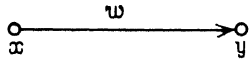


Рис. 3.28

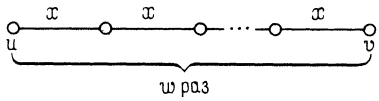


Рис. 3.29

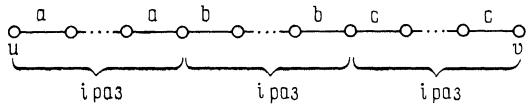


Рис. 3.30

пространства нисходящего вывода выберите условия так, чтобы они минимизировали размеры поискового пространства

3. Используя клаузу

Расстояние (x, y, w) ← Расстояние (x, z, u), Расстояние (z, y, v), Плюс (u, v, w)

и некоторые утверждения вроде

Плюс (3, 2, 5) ←
Плюс (5, 4, 9) ←

которые необходимы для отношения Плюс, постройте нисходящий и восходящий выводы решения задачи ← Расстояние (A, M, w) для графа из упражнения 7) к гл. 1. Сколько различных решений содержит пространство нисходящего вывода?

Является ли задача

← Расстояние (x, x, w)

разрешимой?

4. Отношение $x \leqslant y$ можно задать с помощью клауз Хорна так

$0 \leqslant x \leftarrow$
$s(x) \leqslant s(y) \leftarrow x \leqslant y$

Создайте пространства нисходящего и восходящего выводов (если они конечны) для следующих задач

а) $\leftarrow s(s(0)) \leqslant s(s(s(0)))$
б) $\leftarrow s(s(0)) \leqslant w$
в) $\leftarrow w \leqslant s(s(0))$
г) $\leftarrow s(s(w)) \leqslant s(w)$
д) $\leftarrow s(s(w)) \leqslant s(0)$

5. Определите отношение Плюс (x, y, z), которое выполняется, если x + y = z. При этом можно использовать две клаузы: одну, когда x есть 0, а другую, когда x есть s (x′)

6. Допустим, что отношения Плюс (x, y, z) и

Умножить (u, v, w) определены для свободных от переменных утверждений и выполняются при x + y = z и u * v = w, соответственно.

а) пусть Exp(x, y, z) означает, что x в степени y равно z, что в обычной записи выглядит как $x^y = z$. Выразите следующие высказывания в клаузальной форме, не используя функциональных символов.

$x^1 = x$ для всех x

$x^{(u+v)} = y * z$, если $x^u = y$, а $x^v = z$

$x^u = z$, если $x^{(u+v)} = w$, $x^v = y$, а $y * z = w$

б) используя клаузы из части а) этого упражнения, получите решения следующих задач посредством как нисходящего, так и восходящего опровержений:

– если $2^a = 10$ и a + a = b, то найдите такое w, что $2^b = w$;

– если $3^c = 12$ и b + 1 = c, то найдите такое w, что $3^b = w$;

– покажите, что для каждого x существует такое z, что $x^0 = z$;

при решении можно воспользоваться таким очевидным фактом, касающимся умножения:

Умножить (1, x, x) ←

Коль скоро мы будем применять логику для формулирования задач и описания методов решения задач, то и чисто логические процедуры поиска доказательства начнут приобретать черты решателей задач. Мы постараемся в связи с этим убедить читателя в том, что к выводу на языке клауз Хорна удастся свести много разнообразных моделей поиска решений задач, разработанных в теории искусственного интеллекта.

В этой главе будет проведено сравнение вывода на языке клауз Хорна как с моделями поиска пути на графе типа моделей "Графопроходец" [Doran, Michie 1966] и "Универсальный решатель задач" [Newell, Simon 1963], так и с моделью И–ИЛИ дерева редукции задач [Gelernter 1963], [Nilsson 1971]. В следующей главе мы сравним вывод на языке клауз Хорна с методом поиска решений задач, интерпретируемым как выполнение программ. В последующих главах мы изложим использование нехорновских клауз в поиске решений задач (гл. 7 и 8) и более общие стратегии решения задач (гл. 9).

Тесные связи между методами редукции задач и нисходящим выводом были предметом рассмотрения нескольких работ, в частности [Kowalski, Kuehner, 1971], [Loveland, Stickel 1973], [Pople 1973], [Van der Brug, Minker 1975]. Более того, они уже задействованы в таких системах, как "Логик–теоретик", "Универсальный решатель задач", "Машина для доказательства геометрических теорем" [Gelernter 1963].

Поиск пути

Оказывается, почти любую практически интересную задачу можно свести к задаче поиска пути, имеющей такую формулировку:

Пусть даны исходное состояние A, целевое состояние Z и операторы перехода из одного состояния в другое. Тогда задача состоит в поиске пути из A в Z.

Задача о сосудах с водой

Задачу о сосудах с водой тоже естественным образом можно свести к задаче поиска пути. Исходное состояние этой задачи представлено на рис. 4.1, *а*, а целевое – на рис. 4.1, *б*. Формулировка задачи такая:

Пусть даны изначально пустые семи- и пятилитровый сосуды; цель состоит в поиске такой последовательность действий, которая оставит в

семилитровом сосуде 4 литра жидкости (рис. 4.1,*б*). Для измерения состояния сосудов можно использовать три типа действий:

(1) Сосуд можно целиком заполнить

(2) Сосуд можно целиком опустошить

(3) Жидкость можно перелить из одного сосуда в другой так, что либо первый сосуд станет пустым, либо второй целиком заполнится.

Для этой задачи имеется весьма простая формулировка на языке клауз Хорна. Будем считать, что Состояние(u, v) обозначает такое состояние,

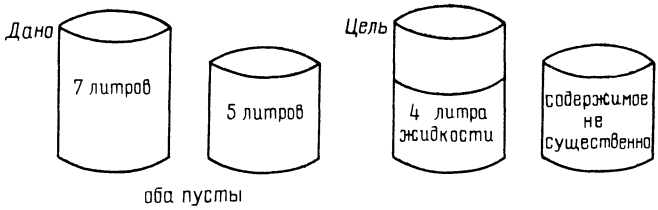


Рис. 4.1

при котором в семилитровом сосуде находится u литров жидкости, а в пятилитровом — v литров.

Допустим, что отношения

$x + y = z$ и $x \leqslant y$

уже заданы (например, бесконечным списком свободных от переменных утверждений)

WC1 Состояние (0, 0) ←

WC2 ← Состояние (4, y)

WC3 Состояние (7, y) ← Состояние (x, y)

WC4 Состояние (x, 5) ← Состояние (x, y)

WC5 Состояние (0, y) ← Состояние (x, y)

WC6 Состояние (x, 0) ← Состояние (x, y)

WC7 Состояние (0, y) ← Состояние (u, v),
$u + v = y$,
$y \leqslant 5$

WC8 Состояние (x, 0) ← Состояние (u, v),
$u + v = y$,
$y \leqslant 5$

WC9 Состояние (7, y) ← Состояние (u, v),
$u + v = w$,
$7 + y = w$

WC10 Состояние (x, 5) ← Состояние (u, v),
$u + v = w$,
$5 + x = w$

Клаузы WC1, WC2 выражают, соответственно, исходное и целевое состояния, WC3 и WC4 определяют действия по заполнению сосудов извне, WC5 и WC6 определяют действия по опустошению сосудов, WC7 и WC8 задают действия по переливанию из одного сосуда в другой до тех пор,

пока первый не станет пустым, WC9 и WC10 задают действия по переливанию из одного сосуда в другой до тех пор пока второй не станет полным.

Перед определением пространств нисходящего и восходящего поиска будет полезно рассмотреть описание пространства поиска с помощью графа. На первых порах мы рассмотрим упрощенную версию задачи поиска пути и ее формулировку на языке клауз Хорна.

Упрощенная задача поиска пути

Представим себе, что наша задача состоит в том, что надо найти путь из вершины A в вершину Z на графе, изображенном на рис. 4.2.

Эта задача может быть сформулирована с помощью одноместного предиката

Достичь (x),

который указывает на возможность достижения узла x нашего графа. Чуть позже в этой главе мы сравним описываемую сейчас формулировку с другой, основанной на семантических сетях и использующей двуместный предикат

Достичь*(x, y),

выражающий то, что возможно достижение вершины x из вершины y.

Достичь (A) ←
Достичь (B) ← Достичь (A)
Достичь (D) ← Достичь (B)
Достичь (E) ← Достичь (B)
Достичь (Z) ← Достичь (X)
Достичь (Z) ← Достичь (Y)

← Достичь (Z)
Достичь (C) ← Достичь (A)
Достичь (F) ← Достичь (C)
Достичь (X) ← Достичь (D)
Достичь (X) ← Достичь (E)

В этой формулировке клаузы, описывающие граф, являются процедурами поиска пути, связывающими соседние вершины. И пространство

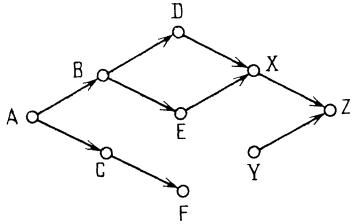


Рис. 4.2

нисходящего поиска (рис. 4.3) и пространство восходящего поиска (рис. 4.4) являются деревьями.

В обоих пространствах поиска имеется взаимнооднозначное соответствие между опровержениями и путями поиска решений. Однако в обоих пространствах поиска имеется нежелательная избыточность. Так, в пространстве восходящего поиска утверждение Достичь (X) ← выводится двумя различными способами, а затем оно дважды совершенно одинаково используется для получения обоих опровержений. А в пространстве нисходя-

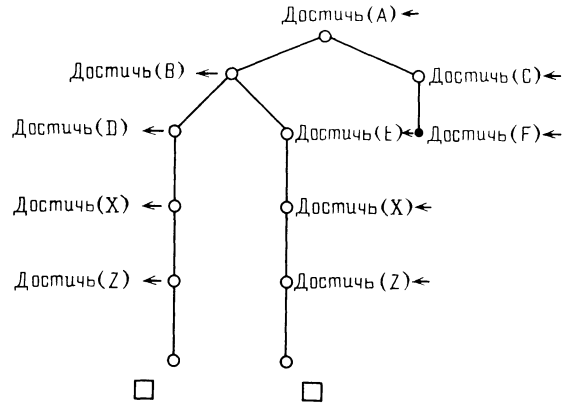


Рис. 4.3. Восходящее пространство поиска

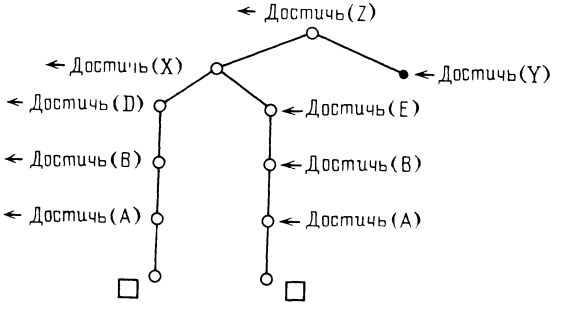


Рис. 4.4. Нисходящее пространство поиска

щего поиска целевое утверждение ← Достичь(B) выводится также двумя различными способами, но затем эта цель дважды совершенно идентично достигается, чем привносится совершенно ненужная избыточность. От такой избыточности можно избавиться, организуя пространство поиска не в виде дерева, а в виде графа общего вида.

Представление поисковых пространств в виде графов

Представление пространства поиска в виде графа получается из древовидного представления за счет идентификации вершин с одинаковыми отметками (рис. 4.5 и 4.6). Поэтому в графовом представлении ни одна клауза не встречается больше одного раза.

Использование графового представления основывается на предположении о том, что, когда бы поисковая стратегия ни пыталась породить клаузу в поисковом пространстве, она прежде всего проверяет, не порождалась ли эта же клауза раньше. Если она действительно порождалась, то сохраняется только один экземпляр этой клаузы. Новый экземпляр, вообще говоря, удаляется.

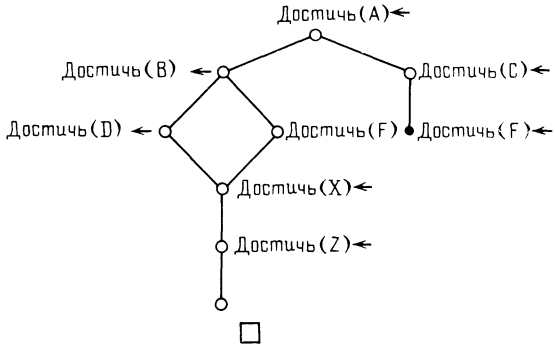


Рис. 4.5. Представление восходящего пространства поиска в виде графа

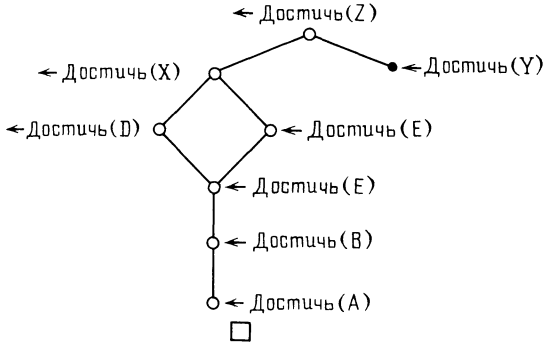


Рис. 4.6. Представление пространства нисходящего поиска в виде графа

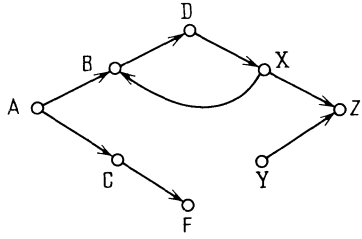


Рис. 4.7

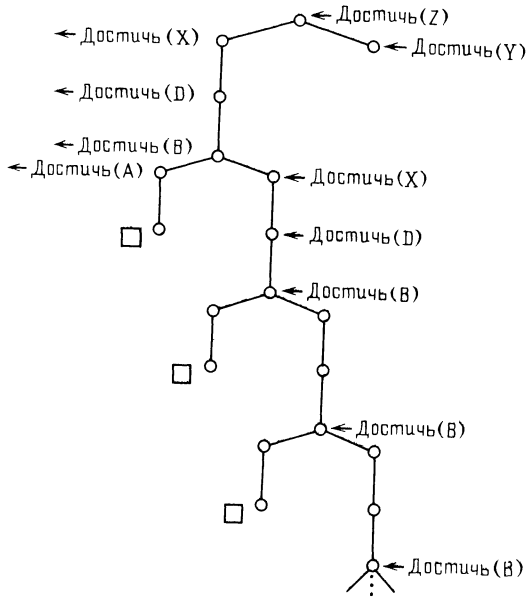


Рис. 4.8. Бесконечное пространство нисходящего поиска, представленное в виде дерева

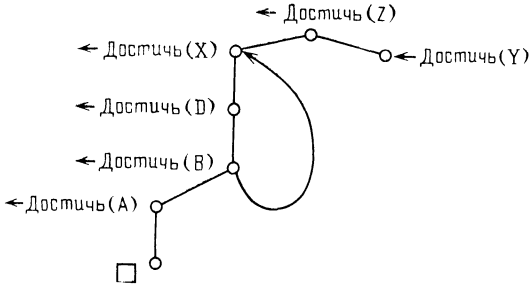


Рис. 4.9. Конечное пространство нисходящего поиска, представленное в виде графа

Графовое представление может свернуть бесконечное пространство поиска в конечное. Для задачи поиска пути из A в Z в приведенном на рис. 4.7 графе пространство нисходящего поиска (рис. 4.8 и 4.9) хорошо иллюстрирует это положение.

Пространство поиска для задачи о сосудах с водой

Теперь мы сможем продемонстрировать графовое представление пространства поиска для задачи о сосудах с водой. В целях упрощения вида поискового пространства не везде будут показаны ребра, которые ведут к вершинам, помеченным уже появлявшимися где-либо в поисковом пространстве клаузами.

Пространство нисходящего поиска (рис. 4.11) оказывается более сложным, чем пространство восходящего поиска (рис. 4.10). Но зато необходи-

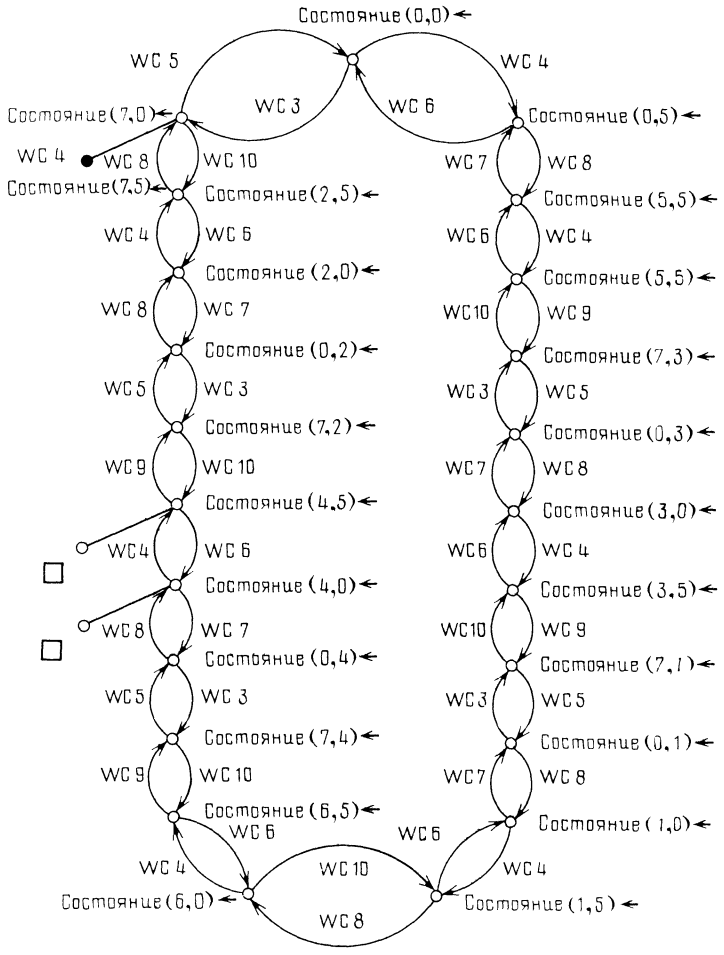


Рис. 4.10. Пространство восходящего поиска для задачи о сосудах

мо отметить, что из вида согласующих подстановок, порождаемых на первом шаге в обеих ветвях пространства нисходящего поиска, следует, что, если цель ←Состояние (4, x) достижима хотя бы при каком-нибудь x, то этот x может быть равен либо 0, либо 5.

Говоря более общо, заключения клауз WC3 . . 10 не будут согласованы ни с одним целевым состоянием, при котором хотя бы один сосуд не был бы пуст или заполнен целиком. В связи с этим соображением в клаузе

←Состояние (u, v), u + v = y

легче рассмотреть сначала вторую цель, достижение которой порождает пары целых чисел, сумма которых равна 9, а затем отбросить те из них, которые делают недостижимой цель Состояние, нежели достигать эти цели в другой последовательности.

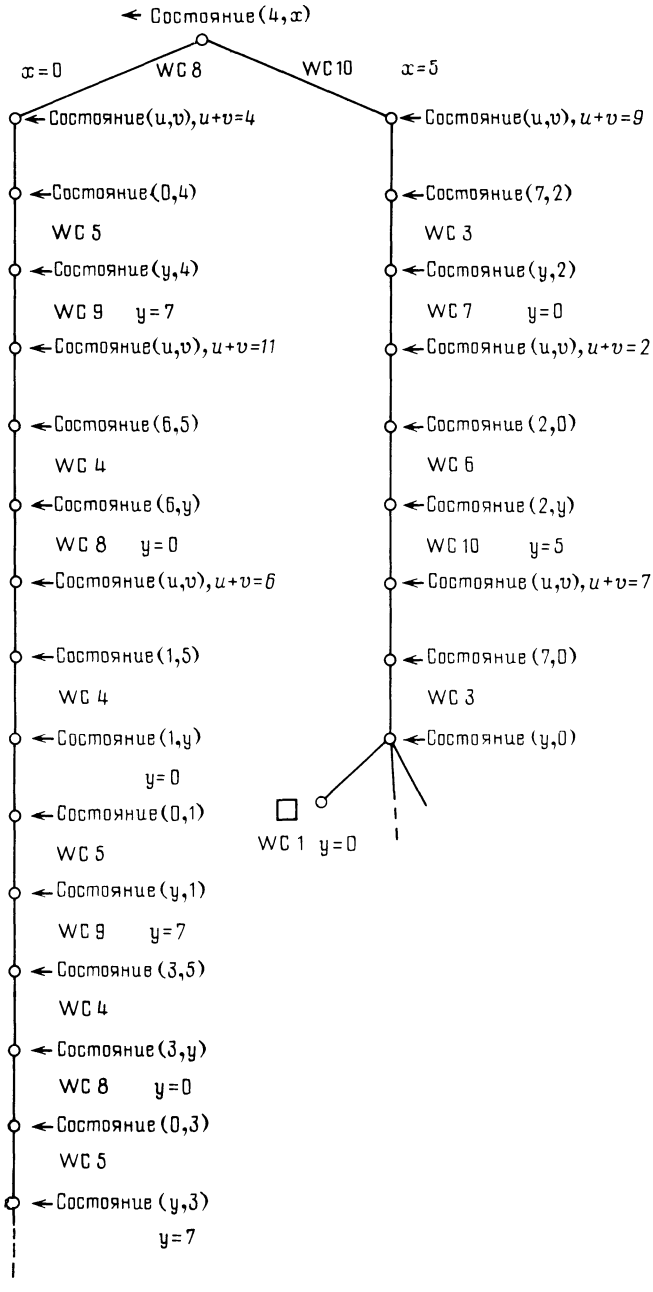


Рис. 4.11. Пространство нисходящего поиска для задачи о сосудах

Поисковые стратегии для задачи поиска пути

Оказывается, что моделирование решения задач в терминах задачи о поиске пути, имеет гораздо большее отношение к выработке поисковых стратегий, нежели к структуре поискового пространства или к представлению информации. При поиске пути на графе общие поисковые стратегии оказываются будто специально приспособленными для этого интеллектуальными методами.

Большинство стратегий решения задачи поиска пути основываются на направленной последовательности действий, заданной так называемой оценочной функцией. Пусть дано некоторое пространство поиска; тогда *оценочная функция* f – это функция, применяемая к вершинам пространства и вырабатывающая в качестве своих значений вещественные числа. Для произвольной вершины N такое значение f(N) приобретает смысл меры полезности продолжения поиска из этой вершины. Чем выше оценка вершины, тем более перспективной является она для применения к ней поисковых операторов. *Эвристическая поисковая* стратегия, направляемая оценочной функцией, всегда организует поиск, начиная с вершины, имеющей наибольшую текущую величину оценки.

В качестве специальных случаев эвристического поиска рассмотрим поиск вглубь и поиск вширь. При поиске вглубь оценка любой вершины прямо пропорциональна ее расстоянию от начальной вершины. При поиске вширь она обратно пропорциональна этому расстоянию от начальной вершины. В обоих случаях расстояние между двумя вершинами может измеряться просто количеством ребер в кратчайшем из путей, связывающих эти вершины.

В типовой задаче поиска пути вершину в пространстве поиска обычно можно сопоставить состоянию некоторого набора объектов. Если имеется набор из n объектов, то состояние такого набора может быть описано n-кой, построенной из индивидуальных состояний каждого из объектов.

В задаче о сосудах с водой, например, имеется два объекта, которые могут находиться в одном из восьми состояний 0 . . 7. Такие базирующиеся на *пространстве состояний* задачи поиска пути могут быть без труда описаны на языке клауз Хорна благодаря использованию предиката

Состояние $(x_1, x_2, \ldots, x_m)$,

выражающего тот факт, что состояние, при котором

1-й объект находится в состоянии x_1
2-й объект находится в состоянии x_2
. . .
m-й объект находится в состоянии x_m

возможно.

Для таких задач, использующих пространство состояний, можно применять оценочные функции специального вида. В простейшем случае, если дана вершина

N = Состояние $(s_1, s_2, \ldots, s_m)$

(которая задает либо утверждение, либо цель, зависящую от какого-ни-

будь направления в пространстве поиска) и если требуется найти вершину

$$T = \text{Состояние}(t_1, t_2, \ldots, t_m),$$

то расстояние между N и T может быть подсчитано как сумма расстояний между двумя состояниями каждого из отдельных объектов:

$$\text{расст}(t_1, s_1) + \text{расст}(t_2, s_2) + \ldots + \text{расст}(t_m, s_m)\ ^{*}).$$

Оценка вершины N при этом должна быть тем больше, чем меньше подсчитанное расстояние до T. Более изысканные оценочные функции могут подсчитывать интегральное расстояние между вершинами как взвешенную сумму отдельных расстояний или как еще более сложную функцию от отдельных расстояний (такую, например, как квадратный корень из взвешенной суммы квадратов расстояний).

Во многих задачах поиска пути вершинам или ребрам графа придается некоторая стоимостная оценка, а задаче придается смысл поиска пути наименьшей стоимости, связывающего данную и целевую вершины. Например, в задаче о сосудах с водой можно было бы потребовать найти самое короткое решение. В подобных случаях считают, что чем больше стоимость достижения вершины, тем меньше оценка этой вершины (в соответствии с некоторой оценочной функцией).

Для таких задач могут пригодиться поисковые стратегии, как направляемые оценочными функциями [Nilsson 1971] так и основанные на методе ветвей и границ [Lawler, Wood 1966].

Для направленной поисковой стратегии оказывается не всегда возможным или желательным использовать численнозначную оценочную функцию. Можно попытаться без ущерба для общности установить како-либо качественное упорядочение между вершинами в поисковом пространстве. Поисковая стратегия, направляемая качественным упорядочением, всегда начинает поиск с вершины, имеющий наивысшее качество.

Поскольку нисходящее опровержение можно рассматривать как путь от начального множества целей до пустой клаузы, постольку проблему поиска опровержения в состоящем из клауз Хорна пространстве нисходящего поиска можно рассматривать как задачу поиска пути, а, следовательно, к ней применима теория эвристического поиска. Однако она должна претерпеть некоторые изменения в случае пространства восходящего поиска, в которых решения более естественно рассматривать как деревья или графы [Kowalski 1972]. Но даже в случае пространств нисходящего поиска эвристическая модель поиска пути не решает важной задачи выделения подцелей. Эти недостатки устраняются с помощью редукционной модели поиска решений задач и связанным с нею представлением в виде И–ИЛИ дерева.

---

*) Вычисление расст (t_i, s_i) зависит от характера t_i и s_i. Если это натуральные числа, то расст $(t_i, s_i) = |t_i - s_i|$ – *Примеч. Д.А. Поспелова*

Процесс редукции поиска решений и его представление в виде И–ИЛИ дерева

Основная идея применения редукционной модели поиска решений заключается в поиске решения исходной задачи за счет использования некоторого исходного набора утверждений и процедур сведения задач к подзадачам.

Воплощением этой идеи служит повторяющееся применение процедур к еще нерешенным задачам, замене их подзадачами и так далее до тех пор,

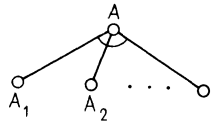


Рис. 4.12.

Рис. 4.13

пока, в конце концов, исходная задача не будет заменена пустым множеством подзадач.

Представление процесса редукции в виде И–ИЛИ дерева состоит в том, что вершины дерева помечаются подзадачами по следующей схеме:

(1) У корневой вершины ставится отметка исходной задачи.

(2) Если текущая вершина помечена задачей А и имеется некоторая процедура, которая редуцирует (т.е. сводит) решение А к решению подзадач $A_1, A_2, \ldots, A_m$, то из текущей вершины исходит *пучок* направленных ребер, каждое из которых ведет в вершину, помеченную одной из подзадач. Сам пучок может быть помечен редуцирующей процедурой (рис. 4.12).

3) Если задача А, помечающая некоторую вершину, согласуется с каким-либо утверждением, то она связывается единственным ребром с вершиной, помеченной пустым множеством подзадач (рис. 4.13).

На рис. 4.14 показано представление в виде И–ИЛИ дерева, а ниже приведено представление на языке клауз Хорна для несложного примера из области редукции задач.

Исходная задача ← Счастливый (Джон)
Процедуры Счастливый (Джон) ← Богатый (Джон)
Счастливый (Джон) ← Нравится (Мери, Джон)
Нравится (Мери, Джон) ← Нравится (Джон, Мери), Добрый (Джон)
Нравится (Мери, Джон) ← Мужественный (Джон), Сильный (Джон)
Нравится (Джон, Мери) ← Прелестная (Мери)
Утверждения Прелестная (Мери) ←
Добрый (Джон) ←
Мужественный (Джон) ←
Сильный (Джон) ←

Задача имеет два решения (рис. 4.15, *а* и 4.15, *б*), каждое из которых можно представить в виде поддерева И–ИЛИ дерева (рис. 4.16).

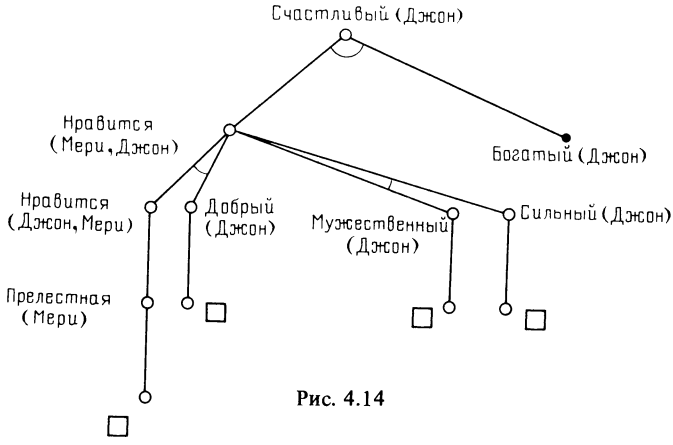


Рис. 4.14

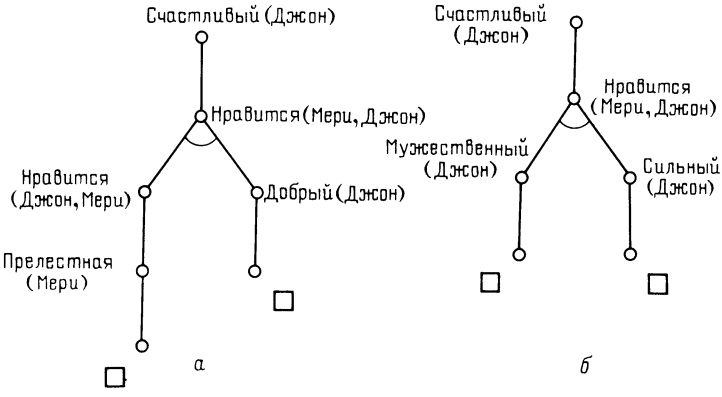


Рис. 4.15. Одно решение (*а*); другое решение (*б*)

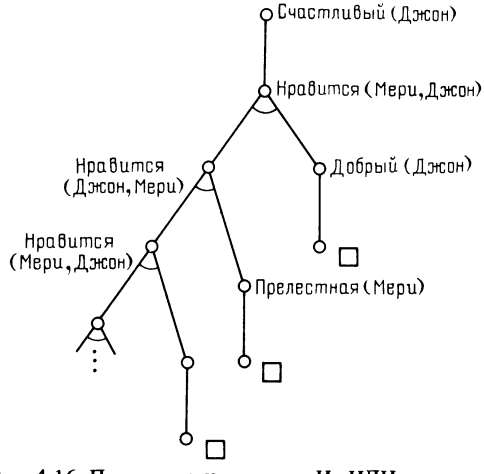


Рис. 4.16. Представление в виде И–ИЛИ дерева

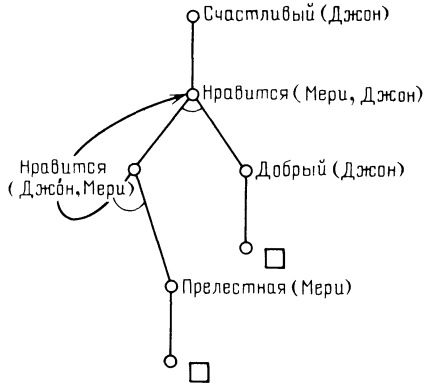


Рис. 4.17. Представление в виде И–ИЛИ графа

Представление в виде И–ИЛИ графа можно получить из представления в виде И–ИЛИ дерева за счет объединения тех вершин, которые помечены одной и той же подзадачей. В нижеприведенном примере (рис. 4.17) представление в виде И–ИЛИ графа превращает бесконечное пространство поиска в виде И–ИЛИ дерева в конечное пространство. Правда, задача все равно не имеет решения.

Исходная задача ← Счастливый (Джон)

Процедуры
Счастливый (Джон) ← Нравится (Мери, Джон)
Нравится (Мери, Джон) ← Нравится (Джон, Мери), Добрый (Джон)
Нравится (Джон, Мери) ← Нравится (Мери, Джон), Прелестная (Мери)

Утверждения
Прелестная (Мери) ←
Добрый (Джон) ←

В представлении редукции задачи как в виде И–ИЛИ дерева, так и в виде И–ИЛИ графа основное внимание уделяется структуре пространства поиска и поисковым стратегиям. Но оба эти представления игнорируют структуру задач, отмечающих вершины пространства поиска и связь между задачами за счет общих переменных. Редукционная модель, сформулированная на языке клауз Хорна, обеспечивает представление задач атомарными формулами и явно указывает (в виде согласующих подстановок) информацию, порождаемую в момент применения процедуры или утверждения.

Интерпретация клауз Хорна в терминах поиска решений

Интерпретация клауз Хорна в терминах поиска решений в основе своей является нисходящей интерпретацией.

Атомы в отрицании $\leftarrow A_1, \ldots, A_m$ интерпретируются как *задачи* или *цели*, которые должны быть решены или достигнуты. Если отрицание содержит переменные $x_1, \ldots, x_k$, то можно считать, что оно задает цель вида:

Найти такие $x_1, \ldots, x_k$, которые
решают задачи $A_1, \ldots, A_m$

В этом случае отрицание называют также *целевым предложением.*

Импликация $A \leftarrow A_1, \ldots, A_m$ интерпретируется как *метод поиска решения задачи* или *процедура*:

Для того, чтобы решить задачу A,
надо решить подзадачи $A_1, \ldots, A_m$

Если имеется задача B, согласующаяся с A, то процедура сводит решение B к решению подзадач $A_1\theta, \ldots, A_m\theta$, где θ есть согласующая подстановка. В дальнейшем будем использовать словосочетание "процедура *согласует* A" наравне со словосочетанием "процедура *применяется* к A".

Утверждение $A\leftarrow$ интерпретируется как *процедура*, которая решает задачу непосредственно, без сведения ее к подзадачам.

Пустая клауза $\square$ интерпретируется как *пустое целевое предложение.*

Представления в виде И–ИЛИ дерева и И–ИЛИ графа можно расширить до общих случаев редукции на языке клауз Хорна. Для этого необходимо представить воздействие процедуры на значения переменных в той задаче, к которой эта процедура применяется. В *представлении в виде расширенного И–ИЛИ дерева* каждый пучок ребер помечается частью согласующей подстановки (называемой значением *выхода*), которая применяется к переменным рассматриваемой задачи. На рисунке 4.18 показано представление в виде расширенного И–ИЛИ дерева для задачи об ошибающихся греках из гл. 1.

В общем случае подстановка θ, которая согласует задачу B с процедурой $A \leftarrow A_1, \ldots, A_m$, может быть разложена на две части $\theta = \theta_i \cup \theta_0$.

(1) Первая часть θ_i воздействует на переменные в процедуре. Она передает *входные данные* из решаемой задачи в процедуру, с помощью которой пытаются эту задачу решить. θ_i называют *входным компонентом* согласующей подстановки.

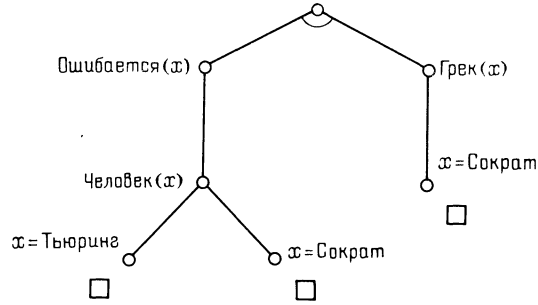


Рис. 4.18

(2) Другая часть θ_0 воздействует на переменные в решаемой задаче. Она передает *выходные значения* из процедуры в задачу, решение которой пытаются получить. θ_0 называется *выходным компонентом* согласующей подстановки.

Таким образом, процедура сводит задачу B к набору подзадач

$A_1\theta_i, \ldots, A_m\theta_i$

в то время как выходной компонент θ_0 воплощает в себе действия процедуры по поиску значений переменных в В.

Когда согласующая подстановка делает переменную в задаче (например, x) идентичной переменной в процедуре (например, y), то полезно представлять себе это действие как передачу подстановкой входных значений, а затем просто включать y = x во входной компонент согласующей подстановки.

Расщепление и независимые подцели

Существенной стороной представления в виде И–ИЛИ дерева является то, что оно наглядно демонстрирует *расщепление* целевого предложения на *отдельные* подцели. Это расщепление особенно полезно, когда подцели не имеют общих переменных. Такие, не имеющие общих переменных подцели, являются ***независимыми*** и могут достигаться различными независимо действующими механизмами поиска решений.

В примере о родственных связях две подцели в начальном целевом предложении

← Родитель (x, Арес) , Родитель (Арес, z)

не имеют общих переменных и, следовательно, независимы (рис. 4.19).

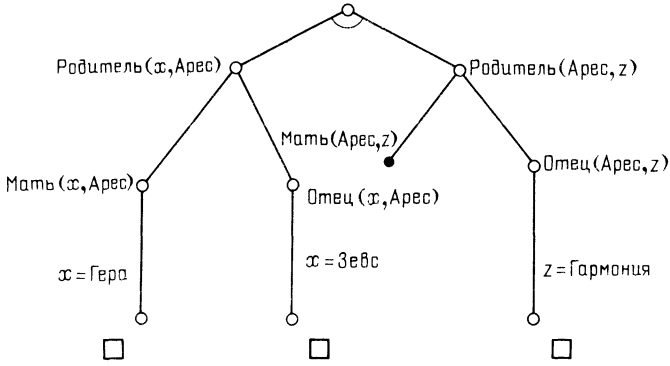


Рис. 4.19

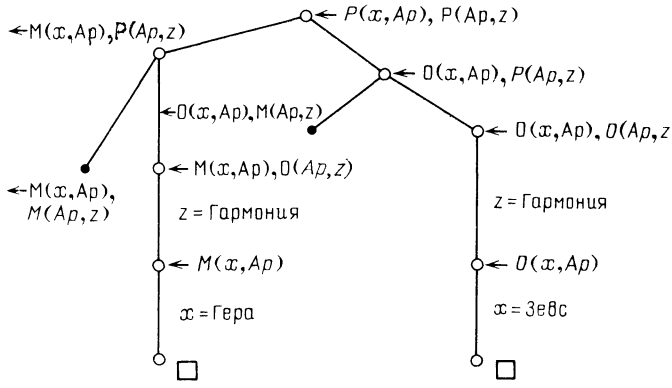


Рис. 4.20

Любое решение задачи поиска x, являющееся родителем Ареса, совместимо с любым решением задачи поиска z, являющимся ребенком Ареса. Механизм поиска решения может работать с отдельными подзадачами одновременно, не опасаясь того, что одно решение помешает другому.

Пространство нисходящего поиска, вершины которого помечены целевыми предложениями, в случае независимых подцелей избыточно. Это можно проиллюстрировать пространством поиска для цели предыдущей задачи (рис. 4.20). Воспользуемся теми же сокращениями, что и в предыдущей главе.

Здесь подцель поиска ребенка Ареса избыточно рассматривается дважды, один раз в контексте целевого предложения $\leftarrow M(x, A_p), P(A_p, z)$, а другой раз – в контексте целевого предложения $\leftarrow O(x, A_p), P(A_p, z)$. В пространстве поиска типа И–ИЛИ дерева цель представляется только однократно.

И вообще, если дано начальное целевое предложение ←A, B, и если имеется n способов достижения цели A и m способов достижения цели B, то в пространстве нисходящего поиска имеется $n * m$ ветвей, в то время как И–ИЛИ дерево содержит их только $n + m$.

Зависимые подцели

Представление в виде расширенного И–ИЛИ дерева не задает связь между решением, обеспечивающим достижение целевого предложения, и решениями, обеспечивающими достижение его отдельных подцелей. В частности, интерпретируя целевое предложение

$$\leftarrow A_1, \ldots, A_m$$

в терминах поиска решений, мы оставляем открытой возможность того, что это целевое предложение может быть достигнуто

(1) за счет независимых попыток достижения отдельных подцелей, получая при этом связанные с ними подстановки $\theta_1, \ldots, \theta_m$, которые обеспечивают попытки достижения подцелей, а затем

(2) за счет комбинации отдельных подстановок для получения решения, обеспечивающего собственно достижение целевого предложения.

Если подцели независимы, то для комбинации отдельных подстановок достаточно выполнить их объединение. Если же они зависимы, то для их комбинации необходимо найти наиболее общий пример этих подстановок. Например, комбинированная подстановка для независимых подцелей в целевом предложении

$$\leftarrow \text{Родитель}(x, \text{Арес}), \text{Родитель}(\text{Арес}, z)$$

является просто объединением

$$\{x = \text{Гера},\ z = \text{Гармония}\}$$

отдельно взятых подстановок. Но комбинированная подстановка для зависимых подцелей

$$\leftarrow 0 < y,\ \text{Четный}(y),$$

в случае, когда даны отдельные подстановки

$$\{y = s(y')\} \text{ и } \{y = s(s(0))\},$$

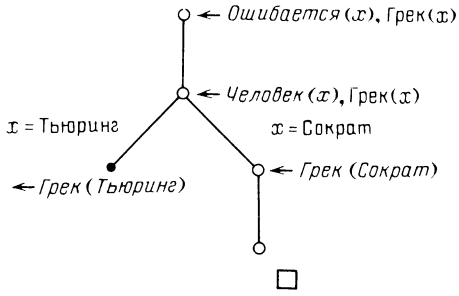


Рис. 4.21. Одно пространство нисходящего поиска

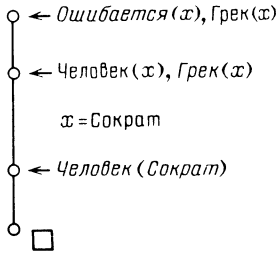


Рис. 4.22. Другое пространство нисходящего поиска

получается лишь за счет согласования значений y и равняется

$$\{y = s(s(0))\}.$$

Пространства нисходящего поиска для целевых предложений проясняют как зависимости между подцелями, так и зависимость размеров поискового пространства от попыток достижения различных подцелей в различной последовательности. И–ИЛИ дерево поискового пространства в задаче об ошибающихся греках, например, не зависит от порядка, в котором достигаются цели верхнего уровня. А вот поисковые пространства целевых предложений совершенно различны. Пытаясь достичь цели в одном порядке, мы получаем пространства поиска с альтернативными ветвями (рис. 4.21), в то время как попытка достичь их в другом порядке порождает пространство поиска, состоящее только из самого решения (рис. 4.22). Отметим, что как и в случае представления в виде расширенного И–ИЛИ дерева, удобно помечать ребра выходными составляющими согласующих подстановок.

В дальнейших частях этой книги мы будем использовать только поисковые пространства целевых предложений, отдавая им предпочтение перед пространствами расширенных И–ИЛИ деревьев, поскольку в пространствах целевых предложений легче продемонстрировать влияние стратегии выбора подцели на размер поискового пространства. Правда, нельзя не сказать, что на практике компьютерные реализации систем поиска решений на языке клауз Хорна используют представление, сочетающее в себе достоинства и тех и других пространств поиска.

Поисковые пространства целевых предложений для задачи об ошибающихся греках иллюстрируют следующий весьма общий принцип. В случае, когда подцели зависимы, *следует выбирать ту, к которой применяется самая маленькая процедура.* Идея состоит в минимизации размера всего пространства поиска за счет локальной минимизации числа альтернативных ветвей, исходящих из вершины.

Поиск versus доказательство*)

В логике не делается различий между процедурами, *доказывающими* то, что некоторое отношение имеет место, и процедурами, которые *отыскивают* конкретные объекты, для которых это так. Например, процедура решения задачи о дедушках и бабушках может не только доказать существование лица, являющегося дедом или бабушкой другого лица, но также может конкретно указать и деда, и бабушку, и внуков.

Различие между доказательством существования и поиском проявляется в наличии или отсутствии переменных. В общем случае, чем больше переменных содержит задача, тем больше результатов она может найти.

Любая процедура, которая применяется для доказательства существования в задаче P(t), также применима и для сопутствующей задачи поиска P(x). Поэтому пространство поиска для задачи поиска решений обычно больше, чем для задачи доказательства существования. Из этого следует принцип *выбора такой подцели, которая обеспечивает минимум поиска при максимуме доказываемого.* Этот прицип перекрывается другим, в соответствии с которым надо выбирать подцели, к которым применимы самые короткие процедуры, но первый принцип легче применять. Действительно, он требует анализа только рассматриваемых подцелей, а не анализа всех согласующих процедур.

Применение этих принципов к процедуре поиска деда или бабушки

Дед–или–бабушка(x, y) ← Родитель(x, z), Родитель(z, y)

можно продемонстрировать при выборе различных подцелей в зависимости от вида решаемой задачи:

(1) Если дан x, то найти для x внуков y, вначале найдя для x детей z, а затем для z найти детей y.

(2) Если дан y, то найти для y дедушек или бабушек x, вначале найдя для y родителей z, а затем для z родителей x

(3) Если даны и x и y, то доказать, что x является дедом или бабушкой y, сравнив прежде всего количество n детей x с количеством m (два) родителей y:

если $n < m$, то сначала найти для x детей
z, а затем доказать, что они родители y;
если $n > m$, то сначала найти для y родителей
z, а затем доказать, что они – дети x;
если $n = m$, то не важно, какую из двух целей выбрать вначале.

(4) Если не даны ни x, ни y, то для того, чтобы найти отдельные объекты, удовлетворяющие отношению "быть дедушкой или бабушкой", оказывается несущественным, какую из подцелей выбрать первой.

---

*) versus – в противопоставлении, вместо (*лат.*) – *Примеч. пер.*

Принцип предпочтения тех подцелей, к которым применяются наименьшие процедуры, имеет два аспекта. С одной стороны, – это *принцип откладывания на потом*, который задерживает сколь возможно выбор самых интенсивно делящихся подцелей, т.е. тех, которые могут быть достигнуты многими способами. С другой стороны, – это *принцип немедленного рассмотрения* тех подцелей, которые могут быть достигнуты немногими способами.

Принцип откладывания на потом может привести к уменьшению поиска благодаря двум моментам. Во-первых, когда подцели имеют общие, разделяемые переменные, то задержка выбора некоторой поисковой задачи (которая может решаться несколькими путями), может привести к преобразованию ее в более обозримую задачу доказательственного типа, которая может быть решена меньшим числом способов.

Во-вторых, поиск значений переменных, в свою очередь, может быть выполнен более эффективно за счет выбора других, менее интенсивно делящихся зависимых подцелей. Можно, не обращая внимания на зависимость или независимость подцелей, отложить рассмотрение интенсивно делящихся подцелей до тех пор, пока исходная задача не была бы решена, быть может, другими способами. К тому времени вне зависимости от того, была ли интенсивно делящаяся подзадача конкретизирована или нет, может случиться, что ее можно проигнорировать.

Принцип немедленного рассмотрения особенно полезен, когда подцель может быть достигнута, самое большее, одним способом. Для разрешения целевого предложения надо достичь все его подцели. Следовательно, если целевое предложение содержит недостижимую подцель, с которой не может быть согласована ни одна процедура, то выбор и распознавание этой недостижимой подцели показывает недостижимость всего целевого предложения; поэтому мы можем избавиться от ненужного рассмотрения других подцелей в нашем целевом предложении. Если же имеется хотя бы единственная процедура, согласующаяся с данной подцелью, то она должна быть рано или поздно применена, если, конечно, само целевое предложение разрешимо. Раннее рассмотрение обладает тем преимуществом, что вся информация в виде значений переменных может быть получена столь скоро, сколь это возможно и может быть, следовательно, передана другим зависимым подцелям. Более того, если в конце концов выясняется, что наша процедура не может обеспечить достижение подцели, то можно исключить рассмотрение более интенсивно делящихся целей в исходном целевом предложении.

Количество процедур (включая утверждение), применимых к данной подцели, есть всего лишь локальное приближение к полному числу способов достижения указанной подцели. Это иногда может ввести в заблуждение. Лучшее приближение можно получить за счет использования механизмов обратного просмотра типа минимаксных методов, обсуждаемых в этой главе позднее.

Воздействие различных стратегий выбора подцелей на размеры поискового пространства просматривается более отчетливо, когда вводятся сложные термы, построенные с использованием значений функциональных символов. Такое влияние сложных термов на выбор подцелей мы рассмотрим в следующей главе.

Леммы, повторяющиеся подцели и циклы

Многие преимущества представления в виде расширенных И–ИЛИ графов могут быть добавлены и в нисходящее представление целевых предложений за счет порождения лемм, фиксирующих решения при достижении подцелей. Когда подцель достигается, можно породить специальное утверждение, которое обеспечивает достижение этой подцели сразу за один шаг. Утверждения такого типа – это *леммы*, которые обнаруживаются нисходящей дедукцией, но не могут быть порождены восходящей. Поэтому лемма, которая была порождена в рамках единого целевого предложения, может быть использована для быстрого достижения этой подцели в случае, когда возникает такая необходимость в рамках другого целевого предложения.

Для увеличения возможностей поиска решений задач на И–ИЛИ графах нужно также порождать и негативные леммы в случаях, когда обнаруживается, что подцель недостижима. Такие негативные леммы можно использовать для распознавания факта недостижимости аналогичной подцели, когда она появляется в рамках другого целевого предложения.

Порождение позитивных лемм было впервые описано Лавлэндом [Loveland 1969] для нисходящей процедуры доказательства посредством исключения образца. И порождение негативных, и порождение позитивных лемм предусмотрено в предложенной Эрли [Earley 1970] процедуре нисходящего грамматического разбора для контекстно–свободных грамматик. Средство, эквивалентное порождению лемм при поиске решений задач на языке клауз Хорна, было предложено Уорреном (неопубликовано) как расширение процедуры разбора Эрли.

Ситуация, когда в одном и том же целевом предложении появляются повторяющиеся подцели, в простейшем случае может быть обработана сразу – просто за счет удаления всех, кроме одного, повторных вхождений. Такое поглощение повторяющихся подцелей в одной и той же клаузе является одним из случаев применения правила факторизации, описанного в гл. 7. Можно отнести это поглощение и к проявлению правила удаления избыточных подцелей, рассматриваемого в гл. 9.

Пожалуй, наиболее важный случай появления повторяющихся подцелей возникает, когда цель встречается в своей собственной подцели. Именно ситуации такого типа чаще всего приводят к зацикливаниям и к бесконечным пространствам поиска. Пусть даны цель B и процедура согласования

$$A \leftarrow A_1, A_2, \ldots, A_m$$

тогда каждая из целей $A_1\theta, A_2\theta, \ldots, A_m\theta$, где θ – согласующая подстановка, является *подцелью* B. Более того, любая подцель подцели B также является *подцелью* B. Таким образом, одна цель является подцелью другой, если они обе находятся на одной и той же ветви И–ИЛИ дерева поискового пространства.

Процедуры обнаружения зацикливаний, проверяющие, не появляется ли цель в качестве своей собственной подцели, составляют одну из важных возможностей процедуры исключения образцов Лавлэнда, а также SL–резолюции. Более общие стратегии обнаружения зацикливаний, которые

проверяют, не выводится ли цель из подцели, были разработаны Дереком Броу [Brough 1979] и включены им в систему поиска решений задач на языке клауз Хорна, разработанную в Имперском Колледже.

Стратегии поиска для пространств редукции задач

Стратегии поиска для И–ИЛИ деревьев и графов обобщают и расширяют стратегии поиска пути. Отличие первых состоит прежде всего в том, что они сочетают работу процедур с выбором подцелей.

Минимаксные и альфа–бета стратегии [Nilsson 1971] обычно используются, когда И–ИЛИ дерево описывает игру. Отдельные подцели в этом случае описывают состояние игры. Альтернативные процедуры, применяемые к данной подцели, представляют собой альтернативные ходы игрока для состояний, заданных подцелью. Пучок подцелей, получающийся в результате применения процедуры, представляет собой набор состояний, связанных со всеми альтернативными ответами оппонента на действия игрока.

Цена хода (представленного процедурой) для игрока велика настолько, насколько силен ответ оппонента. Поэтому цена применяемой процедуры есть минимум цен подцелей в пучке, связанном с этой процедурой. Цена отдельного состояния игры (представленного подцелью), с другой стороны, велика настолько, насколько хорош ход игрока. Поэтому цена подцели есть максимум цен процедур, применяемых к подцели.

Если дана начальная оценка подцелей, то минимаксная оценка выполняет просмотр пространства поиска в обратном направлении и обеспечивает пересмотренную, более точную оценку подцелей. Ее можно использовать не только в теории игр, но и для редукции задач в общем случае. Соответствующим образом модифицированное минимаксное оценивание может быть использовано для построения критерия выбора подцелей. Общий метод использования "просмотра назад" для построения оценочных функций в доказательстве теорем на языке клауз, разработанный для процедур доказательства методом графов соединений [Kowalski 1974] представлен в гл. 8.

Для многих приложений редукции задач оказывается более удобным использовать одну из форм поиска вглубь. Она дает определенный эффект, поскольку за один раз рассматривается только одна ветвь пространства нисходящего поиска. Когда не остается процедур, не испытанных на применение к выбранной подцели целевого предложения на конце ветви, поисковая стратегия возвращается на ближайший к последнему узел этой ветви и пытается достичь выбранную подцель по альтернативному пути. По этой причине поиск вглубь называется также *поиском с возвратами* (*бэктрекингом*).

Хотя бэктрекинг во многих случаях достаточно эффективен, он может оказаться обескураживающе тупым в других случаях. Сейчас будут проиллюстрированы и успешные и неудачные приложения бэктрекинга для задачи распознавания арки.

Эту задачу распознавания арки рассмотрим, например, в сцене, приведенной на рис. 4.23.

Удобно задавать арку при помощи значения функционального символа, зависящего от непосредственных составляющих арки. Пусть терм

a(y, x, z)

обозначает арку (рис. 4.24), которая состоит из блока x (антаблемента), находящегося на верху левой башни y и правой башни z. Башня может быть обозначена при помощи функционального символа, сочетающего

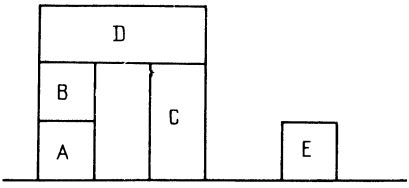


Рис. 4.23

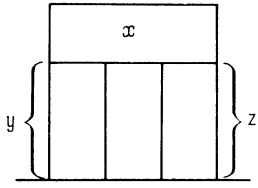


Рис. 4.24

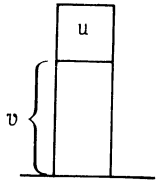


Рис. 4.25

блок на вершине башни с "подбашней" под ним. Введем терм

б(u, v)

обозначающий башню, которая состоит из блока u на верху башни v (рис. 4.25). Поэтому б(B, A) обозначает башню, включающую в себя блок B на вершине блока A; a(б(B, A), D, C) обозначает арку на сцене, приведенной на рис. 4.23. Эту сцену и определения арки и башни мы представим теперь клаузами A1 . . 12

A1 Арка(a(y, x, z)) ← Блок(x), Башня(y), Башня(z), На(x, y), На(x, z)
A2 Башня(x) ← Блок(x)
A3 Башня(б(x, y)) ← Блок(x), Блок(y), На(x, y)
A4 На(x, б(y, z)) ← На(x, y)
A5 Блок(A) ←
A6 Блок(B) ←
A7 Блок(C) ←
A8 Блок(D) ←
A9 Блок(E) ←
A10 На(B, A) ←
A11 На(D, B) ←
A12 На(D, C) ←

Клауза A4 сводит задачу выяснения того, что блок находится на башне к задаче выяснения того, что блок находится на блоке, который находится на верху башни.

Определение арки A1 плохо по нескольких соображениям (см. упражнение 5). Но проблемы, возникающие из-за бэктрекинга, не связаны с этими соображениями.

Рассмотрим задачу

←Арка(a(б(B, A), D, C)

распознавания арки, в которой блок D находится и на башне B, находящейся на A, и на башне C. Если воспользоваться A1 и решать подзадачи в любом

порядке, то получится пространство нисходящего поиска, состоящее лишь из единственного пути, решающего задачу. Здесь ни одна поисковая стратегия, включая бэктрекинг, не ведет себя неразумно.

Предположим, однако, что задача состоит в поиске арки на сцене

← Арка (w)

Допустим при этом, что подзадачи выбираются, а процедуры применяются в том порядке, в котором они записаны. Поскольку такие стратегии особенно просты в реализации, они включены в большое количество действующих компьютерных систем поиска решений. Оказывается, что при таких условиях, исходная задача быстро сводится к неразрешимому целевому предложению, как видно из рис. 4.26.

Простая стратегия поиска вглубь возвращается к предыдущему узлу и ищет другой блок z. Но замена не меняет неразрешимости На (x, y) до тех пор, пока x и y оба равны A. И программа бэктрекинга, пытаясь обработать потенциально бесконечную последовательность башен z, которая не влияет на неразрешимость На (x, y), где x и y суть A, входит в бесконечный цикл.

Бэктрекинг можно сделать более разумным, если при порождении неразрешимой подцели анализировать подстановки, вызывающие эту неразрешимость (в данном случае x = A и y = A) и возвращаться к тому узлу, где их можно отменить (в данном случае целевое предложение, содержащее выбранную подцель Блок (y)). Эффективность может быть повышена путем предотвращения достижения промежуточных подцелей. Программа бэктрекинга может стать более интеллектуальной не только за счет такого анализа неуспеха, который приведет к распознаванию недостижимой подцели, но также и за счет анализа того, как это достижение все-таки можно осуществить [Shmidt 1978]. В нашем примере, когда выясниться, что подцель На (x, y) при x = A и y = A неразрешима, такой анализ может помочь обнаружить, что утверждение На (B, A)← является ближайшим подходящим согласованием. Тогда стратегия поиска может осуществить возврат к целевому предложению, содержащему выбранную подцель Блок (x) с подстановкой x = A, и проверить, не может ли Блок (x) быть достигнутым при x = B. Такой целенаправленный разумный бэктрекинг оказывается близким к идеям модели поиска решений Сассмена [Sussman 1975]. В ней решатель задач вместо аккуратного оценивания подцелей и альтернативных процедур выбирает их произвольно. Если они приводят к неудаче, то решатель анализирует ошибку для того, чтобы найти лучший метод оценивания.

Отметим однако, что такой же, как и в случае достижения подцелей в произвольной последовательности и интеллектуализации бэктрекинга при неудачах, результат может быть получен и непосредственно, если сначала выбирать корректные подцели. В нашем случае достаточно выбрать подцели

На (x, y) и На (x, z)

прежде, чем выбирать остальные в определении арки А1. Точно также и подцель

На (x, y)

должна быть выбрана первой при определении башни А3. Кроме того, выясняется, что необходимо обработать утверждения А10 . . 12, опреде-

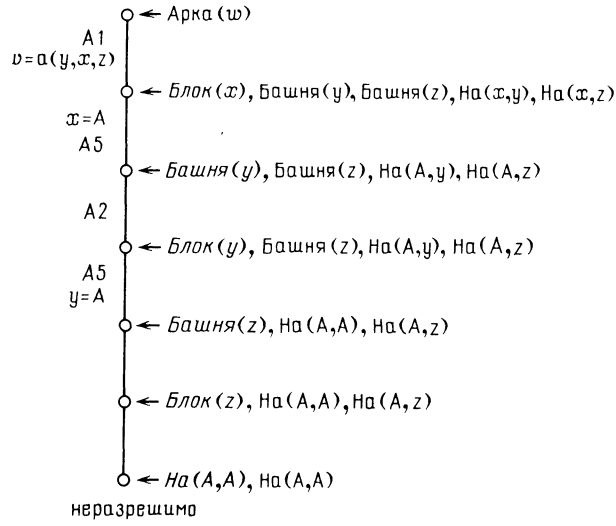


Рис. 4.26

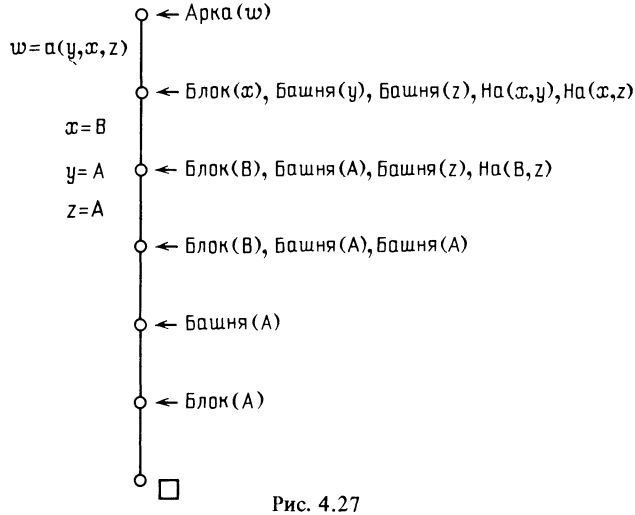


Рис. 4.27

ляющие размещение блоков на блоках, до того, как обрабатывать процедуру A4, которая определяет расположение блоков на башнях (рис. 4.27).

На рисунке повторяющаяся подцель Башня(A) была удалена для устранения избыточности. Отметим также, что первое решение находит "патологическую" арку (рис. 4.28).

Бэктрекинг заложен в состав средств языка программирования Плэннер [Hewitt 1969] и Пролог (программной системы нисходящего разбора на языке клауз Хорна) [Colmerauer 1972], [Roussel 1975]. Неэффективность бэктрекинга в Плэннере привела к разработке Коннай-

вера [Sussman, McDermott 1972a, 1972b] – плэннероподобного языка программирования, в котором программистом определяются и процедуры поиска решений и поисковые стратегии. В Прологе система обеспечивает стратегию поиска методом бэктрекинга, но программист может управлять действиями бэктрекинга.

Различные системы поиска решений, включающие в себя интеллектуальный бэктрекинг, были спроектированы и разработаны Сассменом и

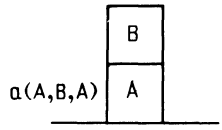


Рис. 4.28

его коллегами [Sussman 1975], [Stallman, Sussman 1977], [Doyle 1978]. Интеллектуальные системы поиска решений, работающие с клаузами Хорна и включающие бэктрекинг, были также исследованы Коксом и Петржиковским [Cox, Pietrzykowski 1976], [Cox 1978] и Бранохе [Bruynooghy 1978]. Ограниченные интеллектуальные стратегии поиска с возвратами были также включены в состав созданных в Имперском Колледже различных систем, базирующихся на клаузах Хорна.

Двунаправленный поиск решений

Клаузы Хорна, описывающие типовую задачу поиска решений, могут быть трех типов:

(1) многоцелевые процедуры (в том числе и утверждения), которые описывают область определения данной задачи;

(2) проблемноориентированные утверждения, которые выражают гипотезы решаемой задачи;

(3) целевое предложение, выражающее саму задачу.

В данном конкретном описании задачи могут отсутствовать проблемноориентированные утверждения. Но, когда они присутствуют, может оказаться полезным сочетание нисходящего (от решаемой задачи) и восходящего (от гипотез к задаче) рассуждений. В этом случае надо устранить из восходящего рассуждения те утверждения, которые являются частью общего описания области определения задачи. Такое ограниченное использование восходящего рассуждения в сочетании с нисходящим является характерной особенностью системы доказательства теорем Бледшоу [Bledsoe 1971].

Бóльшая часть восходящих процедур доказательства, однако, не делает различий между разными типами утверждений. А в результате они в общем случае приводят к комбинаторному взрыву, порождая утверждения, которые следуют из общего описания области определения задачи, и, вдобавок, утверждения, которые следуют из посылок отдельных подзадач.

Полезный критерий для сочетания проблемно–ориентированных восходящих рассуждений с нисходящими рассуждениями состоит в слег-

ка измененном положении Поля [Pohl 1972] для задачи поиска пути:

На каждом шаге выбирать то направление вывода, которое порождает наименьшее число альтернатив.

В нисходящем направлении количество альтернатив – это наименьшее число процедур, которые согласуются с выбранной подцелью в целевом предложении. В восходящем направлении – это есть наименьшее число

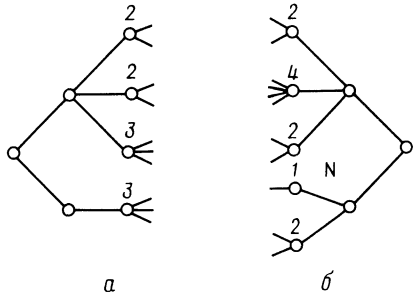


Рис. 4.29. Пространство поиска, порожденное в одном направлении (*а*); пространство поиска, порожденное в другом направлении (*б*)

утверждений, которые могут быть выведены из любых других утверждений. Проиллюстрируем применение критерия Поля для задачи поиска пути на рис. 4.29.

Числа, указанные на рисунке после каждого узла, показывают количества последующих узлов. Критерий Поля выбирает направление, связанное с порождением последователя узла N. В условиях вышеприведенной формулировки задачи поиска пути двунаправленный поиск пути выполняется за счет сочетания нисходящего и восходящего рассуждений.

Обозначения для описаний двунаправленного поиска решений

Разницу между нисходящим и восходящим выводами можно показать на рисунке, используя стрелки, указывающие направление рассуждений. Для каждой пары согласующихся атомов в исходном множестве клауз (в паре один из атомов принадлежит к посылкам, а другой – к заключениям) стрелка направляется от одного атома к другому.

Для нисходящего вывода стрелки направлены от посылок к заключениям. Так, для задачи о дедушках и бабушках мы получим граф, приведенный на рис. 4.30.

Рассуждение ведется по направлению стрелок. Оно начинается с исходного целевого предложения, перемещается по процедурам от заключений к посылкам и заканчивается на утверждениях.

Для восходящего вывода стрелки направлены от заключений к посылкам (рис. 4.31).

Здесь рассуждения начинаются с утверждений, проходят через процедуры от посылок к заключениям и заканчиваются на целевом предложении.

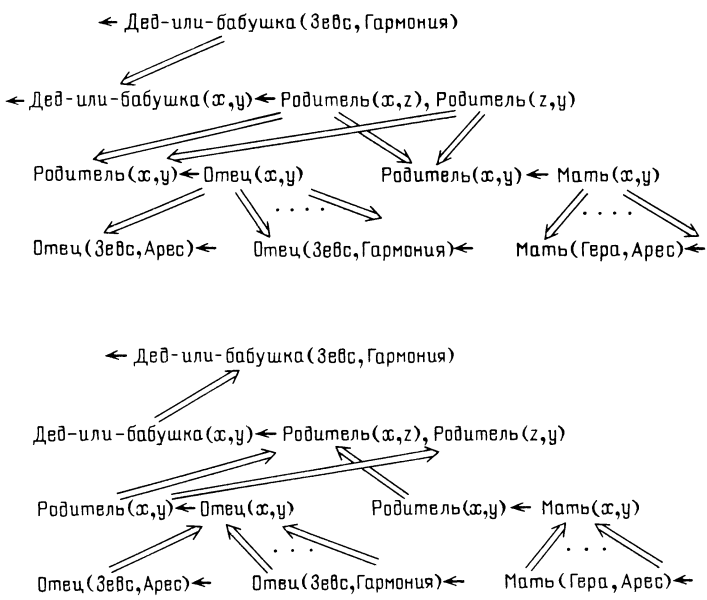


Рис. 4.31

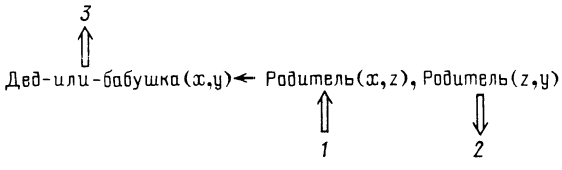


Рис. 4.32

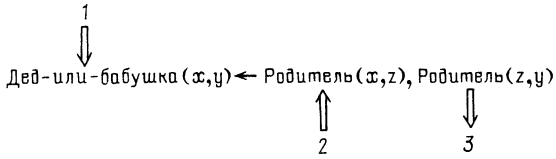


Рис. 4.33

Определение деда-или-бабушки может также быть выполнено комбинированным нисходяще–восходящим методом. Различные комбинации могут быть указаны благодаря использованию номеров для обозначения последовательности действий.

Для простоты мы приведем только обозначения, связанные с определением деда–или–бабушки. Комбинация направлений на рис. 4.32 представляет собой алгоритм, который

1) ждет до тех пор, пока x получит значение родителя z;
2) находит y – ребенка z;
3) утверждает, что x является дедом–или–бабушкой y.

Комбинация, представленная на рис. 4.33, выполняет такие действия:

1) отзывается на задачу демонстрации того, что x является дедом–или–бабушкой y:
2) тем, что ожидает, пока x получит значение родителя z;
3) тем, что пытается показать, что z является родителем y.

Стрелочное обозначение можно также использовать и для нехорновских клауз. В гл. 8 оно применяется для управления поведением процедур доказательства методом графов соединений.

Другая формулировка задачи поиска пути

Эффективность стратегии поиска решений (такой как стратегия двунаправленного рассуждения) зависит не столько от самой задачи, сколько от формулировки задачи. Это можно показать, сравнив вышеприведенную формулировку задачи поиска пути с ее представлением средствами семантических сетей.

В этом представлении будет задействован предикат Достичь* (x, y), который указывает на возможность достичь узел x из узла y. Утверждения описывают двуги в исходном графе. Нижеследующие утверждения описывают граф, приведенный на рис. 4.2

Достичь* (A, B) ←
Достичь* (A, C) ←
Достичь* (B, D) ←
Достичь* (B, E) ←
Достичь* (C, F) ←

Достичь* (D, X) ←
Достичь* (E, X) ←
Достичь* (X, Z) ←
Достичь* (Y, Z) ←

Кроме этих утверждений для поиска пути нужна только одна процедура:

Достичь* (x, y) ← Достичь* (x, z), Достичь* (z, y)

Задача поиска пути из A в Z описывается единственным целевым предложением

← Достичь* (A, Z)

Здесь утверждения специфичны для случая графа, в то время как процедура поиска пути имеет более общее применение. Но только целевое предложение специфично для данного конкретного пути в графе. Восходящий вывод порождает утверждения о путях, которые не обусловливаются отдельными искомыми путями. И прямой и обратный поиск, как и двунаправленный поиск могут быть осуществлены только в рамках нисходящего вывода. Направление поиска зависит от выбора подцели в процедуре поиска пути. Выбор предиката Достичь* (x, z) перед выбором Достичь*(z, y) порождает прямой поиск. Выбор обеих подцелей параллельно или в режиме разделения времени между ними порождает двунаправленный поиск.

Задача поиска пути может быть сформулирована разными способами: одно и то же поведение решателя задач можно получить исходя из разных формулировок за счет применения разных стратегий поиска решений. Даже такое специфическое поведение как заданное двунаправленной стратегией поиска пути, выбирающей на каждом шагу менее всего разрастающее направление, может быть получено с помощью двух формулировок. В первой формулировке оно получается на основе применения критерия Поля в комбинации с нисходящим и восходящим выводами. Во второй формулировке оно получается только за счет применения нисходящего вывода, использующего стратегию выбора подцели, к которой применяется меньше всего процедур (включая утверждения).

Иные аспекты поиска решений

В поиске решений можно выделить три основных этапа:

(1) Первый этап идентифицирует область определения задачи и формулирует процедуры решения задач.

(2) На втором этапе процедуры применяются для решения задач.

(3) На третьем этапе улучшаются стратегии и процедуры поиска решений.

В настоящей главе мы ограничились лишь обсуждением второго этапа и не рассматривали остальные стадии, которые относятся, скорее, к области эрудиции. В этом отношении мы последовали совету Маккарти [McCarthy 1968] и Минского [Minsky 1968] исследовать адекватность языка представления перед тем как иметь дело с проблемами формулирования и улучшения этого представления области определения задачи. В следующей главе мы попробуем проинтерпретировать часть логики, базирующуюся на клаузах Хорна как язык программирования. Это объединит поиск решений с программированием. Первый этап поиска решений задач является начальной стадией формулировки задачи и ее спецификации. Второй этап "запускает" эту спецификацию в работу как программу, а третий выясняет возможную неэффективность этих действий и исправляет их путем улучшения процедур и прочной состыковки стратегий поиска решения задач с решаемыми задачами.

В последующих главах мы изучим роль нехорновских клауз в поиске решения, а также использование глобальных стратегий поиска решений. В последней главе мы сопоставим точку зрения на логику как на метод поиска решений с ее общефилософской ролью модели представления знаний и формализации доводов.

Однако нигде в этой книге не коснемся мы вопросов эрудиции. Не затронем мы также и, вообще говоря, столь важные стратегии поиска решений как поиск по образцу или по аналогии.

Упражнения

1. а) Выразить сформулированную ниже задачу обращения стрелок при помощи языка клауз Хорна, но не используя функциональные символы:

Если даны три стрелки, стоящие подряд и имеющие направления, обозначаемые Н, В, Н, т.е. вниз, вверх, вниз, соответственно, то *целью* является достижение состояния Н, Н, Н, в котором все стрелки направлены вниз. Единственным доступным *действием* для этого будем считать переворачивание пары соседних стрелок, когда у обеих стрелок пары одновременно меняется направление.

У к а з а н и е: Обозначьте предикатом Состояние (x, y, z) тот факт, что допустимо состояние, при котором первая, вторая и третья стрелки ориентированы в направлениях x, y, z

б) Покажите, построив графовое представление пространства нисходящего поиска и продемонстрировав отсутствие в нем решения, что эта задача неразрешима.

в) Опишите, как формулировка на языке клауз может быть изменена с тем, чтобы

i) переворачивать соседние стрелки только тогда, когда они имеют разные направления;

ii) добавлять действие, которое меняет местами соседние стрелки;

iii) обрабатывалась бы строка из четырех, а не из трех стрелок.

2. а) Выразите знаменитую задачу о крестьянине, волке, козе и капусте на языке клауз Хорна:

Крестьянин, волк, коза и капуста находятся на северном берегу реки, и задача состоит в том, чтобы им всем вместе переправиться на южный берег. У крестьянина есть лодка, на которой можно перевести не более одного пассажира за один раз. Козу нельзя оставить вместе с волком в отсутствии крестьянина. Капуста, которая считается равноправным "пассажиром", не может быть оставлена вместе с козой в отсутствии крестьянина.

б) Сравнить графовое представление пространств нисходящего и восходящего поиска.

в) Можете ли вы придумать приемлемую оценочную функцию, направляющую поиск.

3. Пусть даны два различных представления задачи поиска пути; требуется сравнить стратегии поиска решения, нужные для

а) обнаружения того, что нет пути из A в B, если нет ребра, исходящего из A или ребра, входящего в B;

б) доказательства того, что можно достичь A из A.

4. Пусть некие последовательности описываются значениями двух отношений:

Элемент (i, j, k), которое истинно, когда $i_j = k$,

т.е. j-й элемент последовательности i есть k и

Длина (i, u), которое истинно, если длина последовательности i есть u.

Так, последовательность

$A: a_1, a_2, \ldots, a_n$

можно описать утверждениями:

Элемент $(A, 1, a_1) \leftarrow$
Элемент $(A, 2, a_2) \leftarrow$
. . . . . .
Элемент $(A, n, a_n) \leftarrow$
Длина $(A, n) \leftarrow$

Допустим, вдобавок, что отношение Плюс (x, y, z) выполняется, когда $x + y = z$

а) Определить при помощи клауз Хорна отношение Сумма (x, v), которое выполняется, когда v есть сумма чисел в последовательности x.

б) Используя клаузы из части (а), найти методом нисходящего поиска сумму чисел в последовательности B: 3, 4 10.

в) Может ли отношение Сумма (x, v) быть определено таким способом, что если дано x, то для поиска v пространство поиска имело бы только одно решение?

5. а) Выпишите все решения задачи

$\leftarrow$ Арка (w)

выводимые из определения арки и описания сцены, заданными клаузами A1. .12

б) Переформулируйте определение арки и башни с помощью клауз Хорна так, чтобы исключить максимально возможное количество "патологических" арок (эта задача может быть решена более простым способом позже с использованием отрицания как неудачи, рассмотренного в гл. 11).

6. Рассмотрите задачу

$\leftarrow$ Число (u), Число (v), $u > v$

заданную клаузами

Число (0) $\leftarrow$
Число (s(x)) $\leftarrow$ Число (x)
$s(x) > 0 \leftarrow$
$s(x) > s(y) \leftarrow x > y$

Проанализируйте поведение стратегии поиска бэктрекингом для решения этой задачи. Считайте при этом, что достижение подцелей происходит в том порядке, в котором они записаны, и альтернативные клаузы проверяются тоже в заданном порядке.

Клаузу Хорна

$$B \leftarrow A_1, \ldots A_m,\ m \geqslant 0$$

можно интерпретировать как *процедуру, тело* которой $\{A_1, \ldots, A_m\}$, в свою очередь, состоит из процедурных вызовов. Тогда нисходящий разбор можно воспринимать как *вычисления. Обращение к процедуре* — это порождение нового целевого предложения из старого путем согласования некоторого процедурного вызова с *именем* B какой-либо процедуры.

$$B \leftarrow A_1, \ldots, A_m$$

Логическая программа — это множество процедур, выраженных на языке клауз Хорна: ее работа инициируется начальным целевым предложением.

В обычных программах логика информации, используемой в процессе поиска решений, зачастую неотделима от управления способами использования этой информации. Логические программы более абстрактны. Они не управляют ни порядком обращения к различным процедурам при согласовании с процедурным вызовом более, чем одной процедуры, ни порядком выполнения процедурных вызовов в том случае, когда их несколько в одном и том же целевом предложении.

В логических программах заключена только логика методов поиска решений. Они легче в понимании, легче в проверке, легче в изменениях. Они влекут к себе неопытных программистов, пользователей баз данных и вообще всех, кто не желает вникать в детали управления поведением программ.

Первая система логического программирования, базирующаяся на процедурной интерпретации клауз Хорна [Kowalski 1974] и названная Пролог [Colmerauer 1973], [Roussel 1975], была спроектирована и внедрена в 1972 г.

Пролог-компилятор, написанный на Прологе для PDP-10 был разработан в Эдинбургском университете Уорреном, Перейра и Перейра [Warren, Pereira, Pereira 1977]. Они доказали, что Пролог-компилятор исполняет лиспоподобные логические программы так же эффективно, как и Лисп-компилятор [McCarthy 1962].

Термы как структуры данных

Данные в логических программах могут быть представлены посредством термов или посредством отношений. С одной стороны, использование термов в качестве структур данных придает программам на языке клауз Хорна определенное сходство с таким языком обработки списков как Лисп. Говоря более общо, термы ведут себя как рекурсивные структуры данных в духе определений Хоара [Hoare 1972]. С другой стороны, использование отношений в логических программах сближает их с представлением данных отношениями в формальных моделях баз данных [Codd 1970]. Отношения, в свою очередь, похожи на таблицы и массивы в обычных языках программирования. Позже в этой главе мы обсудим отношения более детально.

Как и в Лиспе, бинарные деревья можно описывать с помощью бинарных функциональных символов; так сост(x, y) обозначает дерево на рис. 5.1, у которого имеется поддерево x непосредственно слева от корневой вершины и поддерево y непосредственно справа от корневой вершины. Поэтому терм сост(a, сост(B, c)) обозначает дерево на рис. 5.2, а программа

```
Оконечные–вершины(x, 1) ← Метка(x)
Оконечные–вершины(сост(x, y), w) ←
    Оконечные–вершины (x, u),
    Оконечные вершины(y, v),
    u + v = w
```

определяет отношение Оконечные–вершины(x, y), которое истино, когда y равно числу оконечных вершин бинарного дерева x. Метка(x) истин-

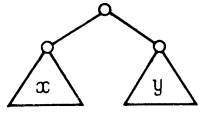


Рис. 5.1

Рис. 5.2

но, когда x есть метка оконечной вершины; например, для рис. 5.2 имеет место

```
Метка (A) ←
Метка (B) ←
Метка (C) ←
```

Целевое предложение

```
← Оконечные–вершины (сост(A, сост(B, C)), y),
```

задает в качестве цели вычисление количества оконечных вершин дерева на рис. 5.2. Здесь терм сост(A, сост(B, C)) обозначает *вход* программы, а переменная y – ее *выход*. Решение задачи нисходящим способом, приведенное на рис. 5.3, является вычислением выхода y = 3. Поисковое пространство содержит только это вычисление.

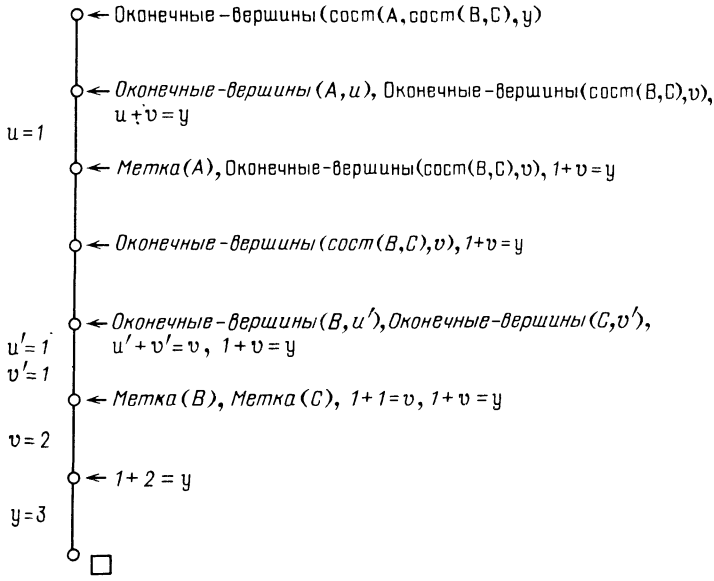


Рис. 5.3

Списки, как и в Лиспе, можно считать особым случаем бинарного дерева. А именно, терм сост(x, y) обозначает список на рис. 5.4, в котором за *первым элементом* x следует *список* y. Константный символ nil обозначает *пустой список*. Поэтому терм сост (A, сост (B, сост (C, nil))) обозначает список A, B, C, а программа

```
Элемент ( сост(x, y), 1, x) ←
Элемент ( сост(x, y), u, z) ←  Элемент (y, v, z),
                               v + 1 = u
```

задает отношение Элемент (x, y, z), которое истинно, когда y-й элемент списка x есть z. Отметим, что терм сост(A, B) не обозначает список, поскольку B – это не список. Список, состоящий из одного элемента B,

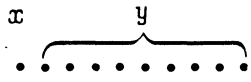


Рис. 5.4

обозначается как сост(B, nil) и поэтому список A, B обозначается как сост(A, сост(B, nil)).

Программы можно сделать более легкими для восприятия, если для функциональных символов использовать *инфиксную нотацию*, а также, если воспользоваться соглашениями о подавлении лишних скобок. Особенно удобным для списков оказалось использовать инфиксный функциональный символ "."; следовательно, x.y можно записать вместо сост(x, y).

Скобки удобно устранять, полагая, что x . y . z можно записать вместо сост(x, сост(y, z)). Поэтому список A, B, C можно представить термом

A. B . C . nil

В системе Пролог предусмотрены специальные средства для задания инфиксных функциональных символов и для устранения скобок. Вдобавок программисту предоставляется еще одна возможность устранять скобки за счет задания отношения приоритета между задаваемыми функциональными символами. Так, если объявить, что инфиксный функциональный символ & теснее связывает свои аргументы, чем инфиксный функциональный символ ⊃, то вместо терма

(p & q) ⊃ (r & s)

можно писать

p & q ⊃ r & s

Вычисление методом последовательного приближения к выходному значению

Процедуры на языке клауз Хорна вырабатывают результат в ходе вычисления. Частичные результаты при этом накапливаются и формируют последовательные приближения к окончательному результату. Эти приближения порождаются независимо от того, завершается ли вычисление успешно или нет.

Рис. 5.5 иллюстрирует вычисление списка, получающегося присоединением 3.nil к 2.1.nil, методом последовательного приближения в программе

(1) Присоединить (nil, x, x) ←
(2) Присоединить (x . y, z, x . u) ← Присоединить (y, z, u)

Здесь клауза (1) показывает, что присоединение любого списка x к пустому списку дает сам список x. Клауза (2) показывает, что присоединение некоторого списка z к непустому списку x . y порождает список x . u с тем же самым первым элементом x с остатком u, который получается в результате присоединения z к y.

Последовательные шаги вычисления определяют последовательные приближения к результату

x = 2 . u
x = 2 . 1 . u′
x = 2 . 1 . 3 . nil

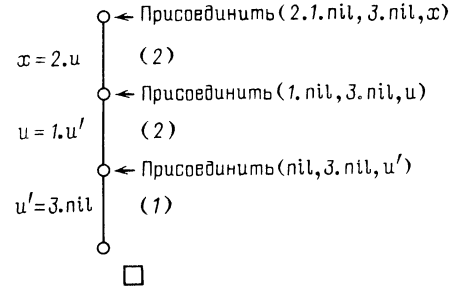


Рис. 5.5

В общем случае, *выходной результат* можно рассматривать как совокупность всех выходных ком-

понентов согласующих подстановок, выполненных в процессе вычисления. Этот результат можно записать более сжато как в вышеприведенном примере, выполняя аппликацию компонентов результатов, находящихся в опровержении ниже, к термам компонентов результата, находящимся в опровержении выше.

Изменение назначений параметрам ролей входов и ролей выходов

Разница между входными и выходными (результирующими) параметрами процедуры зависит от контекста, в котором происходит обращение к процедуре. Любое подмножество параметров процедуры можно считать входным. Тогда оставшиеся параметры вычисляются и тем самым воспринимаются как результат.

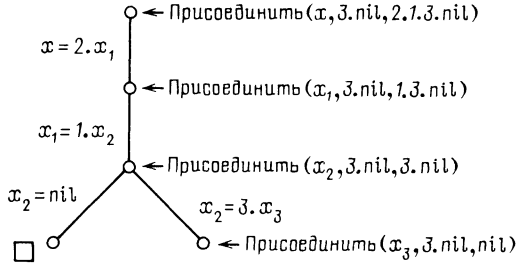


Рис. 5.6

Вычисление, приведенное на рис. 5.6 демонстрирует использование предиката Присоединить для нахождения списка x, который порождает 2.1.3.nil, если к нему присоединить список 3.nil. Пространство поиска, кроме успешно заканчивающегося вычисления, содержит еще шаг, правда только один, заканчивающийся неудачей, потому что не находится процедуры, которая могла бы согласоваться с его процедурным вызовом.

Важной особенностью логических программ является возможность выполнять одну и ту жу процедуру при меняющихся назначениях группам параметров ролей входов и ролей выходов. Из этой особенности, в частности, следует, что одни и те же процедуры можно применять и для вычисления производных от функций и для вычисления первообразных [Bergman, Kanoui 1973]. Точно так же процедуры, которые проверяют соответствие данной программы данным спецификациям, могут быть использованы для порождения программ из спецификаций [Moss 1977].

Недетерминизм первого рода: несколько процедур согласуются с одним процедурным вызовом

Если проводить сравнение с обычными программами, то о выполняемых по нисходящим правилам программах на языке клауз Хорна можно сказать, что они ведут себя недетерминированно в двух основных случаях: во-первых, если несколько процедур согласуется с данным процедурным вызовом, то про поисковую стратегию, в соответствии с которой применяются альтернативные процедуры, говорят, что она обладает не-

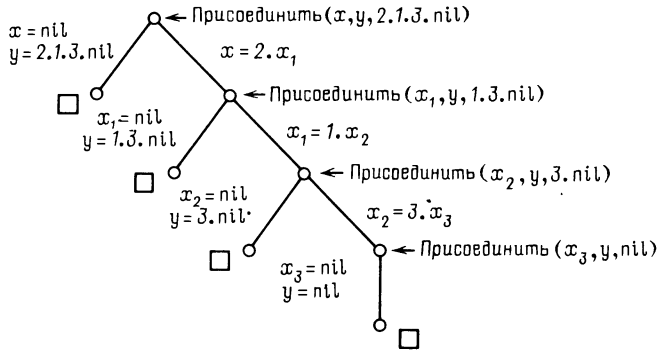


Рис. 5.7

детерминизмом первого рода; во-вторых, если в одном целевом предложении должно быть выполнено несколько процедурных вызовов, то о порядке их выполнения говорят, что он обладает недетерминизмом второго рода.

В первом случае вычисления по альтернативным процедурам могут привести к различным результатам. И, если нужен только один из них, то недетерминизм первого рода проявляется в том, какой из них будет найден. Если же требуются все результаты, то этот недетерминизм проявляется в том порядке, в котором они будут получены.

Процедура, которая не обладает недетерминизмом первого рода при одном распределении назначений параметрам ролей входов и выходов, может обладать этим недетерминизмом при другом распределении. Например, процедура Присоединить обладает недетерминизмом первого рода, когда она используется для разбиения данного списка на две части, например, в такой задаче:

← Присоединить (x, y, 2.1.3.nil).

Поисковое пространство приведено на рис. 5.7. Обратите внимание на определенную экономию, получающуюся за счет представления этого пространства в виде дерева. Действительно, например, два различных разбиения

x = 2.1.nil y = 3.nil

и

x = 2.1.3.nil y = nil

получаются из одного и того же начального приближения

x = 2.1.x_2.

Последовательный поиск как итерация

Существенной особенностью всех языков программирования является возможность задать повторяющееся выполнение одной и той же команды. Такое повторное выполнение, называемое *итерацией**) можно организовать за счет рекурсивного выполнения процедур на языке клауз Хорна. Можно обеспечить его также и при использовании бэктрекинга для поиска в пространстве альтернатив. Простой иллюстрацией сказанного может служить определение деда-или-бабушки. Пусть даны некоторые сведения о том, как отдельные индивидуумы связаны между собой отношением "быть родителем":

Родитель (Зевс, Арес) ←
Родитель (Гера, Арес) ←
Родитель (Арес, Гармония) ←
Родитель (Семела, Дионис) ←
Родитель (Зевс, Дионис) ←

и т.д., а задача состоит в том, чтобы показать, что Зевс – это дед-или-бабушка Гармонии:

← Дед-или-бабушка (Зевс, Гармония)

используя для этого определение отношения "быть дедом-или-бабушкой":

Дед-или-бабушка (x, y) ← Родитель (x, z), Родитель (z, y)

В обычных языках программирования программист должен был бы заботиться и о хранении данных в отношении "быть родителем", и об извлечении данных из этих отношений. В логических программах решения по выполнению этих действий вместо программиста принимает исполняющая система. И в том и в другом случае простейшей стратегией оказывается последовательное запоминание и последовательный поиск данных. Отношение "быть родителем" можно хранить последовательно либо в двумерном массиве, либо в связном списке. Стратегией последовательного поиска служит итерация, состоящая из двойного цикла, причем один вложен в другой. Для того, чтобы показать, что Зевс является дедом-или-бабушкой Гармонии, во внешнем цикле надо найти ребенка Зевса z, а во внутреннем цикле надо проверить, является ли z родителем Гармонии. Итеративный алгоритм, который для этой задачи, скорее всего, напишет программист на обычном языке программирования, будет эквивалентен внесению за счет использования бэктрекинга определенности первого рода в программу, содержащую недетерминизм первого рода.

В других случаях, например, когда процедура Присоединить используется для разбиения списков, бэктрекинг обладает большей общностью, чем итерация. В общем случае, в то время как глубина дерева поиска при итерации определяется числом вложенных циклов, бэктрекинг обеспечивает поиск в дереве альтернатив неограниченной глубины.

Пригодность поисковой стратегии зависит и от структуры хранения данных. Так итерация, являющаяся последовательным поиском, годит-

*) В тексте двух ближайших пунктов этой главы имеются ссылки на некоторые свойства обычных языков программирования. Читатель, не знакомый с такими языками, может без ущерба пропустить эти пункты.

ся для обработки данных, хранящихся последовательно. Другие поисковые стратегии применимы к иным структурам данных, вроде хэш–таблиц, бинарных деревьев, семантических деревьев. Например, Фишмэн и Минкер [Fishman, Minker 1975] организовали хранение данных методом, обеспечивающим параллельный поиск, а Делиянни и Ковальски [Deliyanni, Kowalski 1979] предложили стратегию построения пути для поиска данных в сематических сетях.

"Не знаю" versus "не забочусь" в условиях недетерминизма первого рода

Недетерминизм первого рода не всегда вызывает необходимость добавочной проверки решения. Примером может служить определение отношения Max (x, y, z) (говорящего о том, что максимум из x и y есть z):

Max (x, y, x) ← x ⩾ y
Max (x, y, y) ← y ⩾ x

Если x и y одинаковы, то равноприменимыми оказываются обе процедуры, как, например, в случае

← Max (3, 3, z)

Кроме того, неизбежный в общем случае поиск решения иногда, если он не нужен, может породить избыточность. Например, такая избыточность порождается бэктрекингом, если он возникает в целевом предложении

← Max (3, 3, z), Четный (z)

и если процедурные вызовы происходят в том порядке, в котором они записаны. Второй процедурный вызов, Четный (z), который завершается успехом при четном z, в данном примере неудачен независимо от того, как будет исполнен первый процедурный вызов. Использование же бэктрекинга после первой неудачи в попытке найти иной способ выполнения первого процедурного вызова является и ненужным и избыточным.

В общем случае сужение поиска допустимо, если выходные переменные функционально зависят от входных; например, когда переменная y функционально зависит от переменной x в отношении F(x, y), а про x известно, что это входная переменная. Может оказаться целесообразным отказаться от бэктрекинга, как например, в случае, когда решение, полученное при достижении цели F(A, y), приводит к неудаче попытку достижения второй цели G(y) в целевом предложении

← F(A, y), G(y)

С одной стороны, когда добавочный поиск решения не нужен, исполняющая программная система может "*не заботиться*" о том, какое решение получить или как его получить. С другой стороны, добавочный поиск неизбежен, когда она "*не знает*" что-либо. Отсутствие заботы о решении в условиях недетерминизма первого рода является ведущей особенностью языка охраняемых команд Дейкстры [Dijkstra 1976]. Использование свойства "не забочусь" в условиях недетерминизма первого рода для ог-

раничения ненужного поиска – это одна из форм интеллектуального бэктрекинга.

При недетерминизме первого рода могут сочетаться оба свойства: и "не знаю" и "не забочусь'. Примером этому может служить задача поиска пути. Если, например, дана задача поиска пути от A до N

← Достичь* (A, N)

то программная исполняющая система не заботится о том, какой путь найден, но обычно и не знает, какие процедуры нужно применить для его отыскания. Добавочный поиск решения необходим для нахождения одного пути, а после этого он (поиск) становится и ненужным и избыточным.

Задача поиска пути оказывается специальным случаем более общей ситуации, когда некоторый процедурный вызов не имеет общих переменных с другими процедурными вызовами в одном и том же целевом предложении. Любой недетерминизм первого рода, возникший при выполнении процедурного вызова, сохраняется только до тех пор, пока не найдено первое решение. Примером может служить второй процедурный вызов в теле процедуры

Счастлив (Боб) ← Преподает (Боб, x), Посещает (x, y)

т.е. Боб счастлив, если он преподает курс, который хоть кто-нибудь посещает. Если этот второй вызов исполняется после другого процедурного вызова, то его единственная переменная y больше не встретится в других процедурных вызовах, и оказывается достаточным найти хотя бы одно решение.

Свойство процедурного вызова не содержать переменных или содержать только такие переменные, которые не содержатся в других процедурных вызовах, есть синтаксическое свойство, которое вполне может распознаваться исполняющей программной системой без участия программиста. Однако проблема обнаружения ситуации, при которой поиск можно ограничить в силу того, что процедурный вызов вычисляет значение функции, является принципиально неразрешимой. Поэтому, если программист передаст информацию об этом исполняющей системе, скажем, в виде комментария, то затраты будут куда меньше, чем в случае, когда исполняющая система станет выявлять эту ситуацию самостоятельно.

Возможность не заботиться о решении в условиях недетерминизма первого рода обеспечивает способ внесения в программу добавочной информации без расширения пространства поиска и даже, возможно, с ограничением его размеров. Эта новая информация может привести к более непосредственному решению задачи, чем исходные процедуры, а, если недетерминизм первого рода не возникает, то исходные процедуры вообще можно проигнорировать.

Недетерминизм второго рода: планирование процедурных вызовов

В обычных языках программирования программа управляет планированием процедурных вызовов – чаще в некоторой фиксированной последовательности, реже осуществляя разделение времени между ними или выполняя их параллельно. В логике, однако, тело процедуры задает только некоторый набор процедурных вызовов. Дисциплина их выполнения обладает определенностью второго рода не за счет самой программы, а только за счет исполняющего механизма. Различные стратегии планирования процедурных вызовов воздействуют на эффективность выполнения, но не воздействуют на его смысл, поскольку он определяется вычисляемыми отношениями.

Простым примером может служить программа сортировки списков. Допустим, что определение отношения $\geqslant$ уже дано. Пусть отношение Сорт(x, y) выполняется, когда y есть отсортированный вариант списка x; Перест(x, y) выполняется, когда y является перестановкой x; Удалить(x, y, z) означает, что z получается в результате удаления какого-нибудь одного вхождения x в y.

S1 Сорт (x, y) ← Перест (x, y), Упорядочено (y)
S2 Перест (nil, nil) ←
S3 Перест (z, x . y) ← Удалить (x, z, z′), Перест (z′, y)
S4 Удалить (x, x . y, y) ←
S5 Удалить (x, y . z, y . u) ← Удалить (x, z, u)
S6 Упорядочено (nil) ←
S7 Упорядочено (x . nil) ←
S8 Упорядочено (x . y . z) ← x $\leqslant$ y, Упорядочено (y . z)

В принципе, процедурные вызовы в теле процедуры S1 могут выполняться в любой последовательности. Если дан список l, то для порождения отсортированного варианта y списка l можно сначала выполнить процедурный вызов Упорядочено(y), породив некоторый упорядоченный список y, а затем выполнить Перест(l, y), проверив, не является ли y перестановкой l. Если проверка приведет к неудаче, то можно сгенерировать другой упорядоченный список и так далее до тех пор, пока проверка не закончится успехом. Конечно же, большей эффективности можно достичь, выполняя процедурные вызовы в другой последовательности – сначала порождая перестановку x, а затем проверяя, не упорядочена ли она. Но независимо от порядка выполнения процедурных вызовов и независимо от той цены, которую пришлось бы за это уплатить, результат в смысле вычисления выходных значений из входных был бы тем же самым.

Эффективное планирование процедурных вызовов зависит от распределения входов и выходов. В общем случае более эффективным может оказаться предварительное выполнение тех процедурных вызовов, которые содержат входные параметры, перед теми, которые их не содержат. Поэтому, если дана задача поиска отсортированного варианта y входного списка l, изображенная на рис. 5.8, то лучше сначала избрать для выполнения процедурный вызов Перест (l, y), который содержит входной параметр, а не Упорядочено (y), который его не содержит.

Если, как на рис. 5.9, даны и l_1 и l_2, и задача состоит в *проверке* того, является ли l_2 отсортированной версией l_1, то оба процедурных вызова содержат входные параметры и от очередности выполнения процедурных вызовов эффективность работы программы зависеть не будет. Более того, поскольку оба процедурных вызова не имеют общих переменных и поскольку они являются одинаково подходящими кандидатами для выполнения, они могут быть выполнены совместно – либо в режиме разделения времени между ними, если доступен только один процессор,

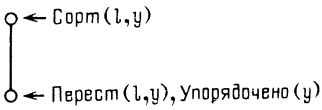


Рис. 5.8

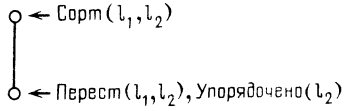


Рис. 5.9

либо в параллельном режиме, если можно использовать несколько процессоров.

В общем случае выгодно выполнять процедурные вызовы сразу как только для них становится доступным достаточный для выполнения набор исходных данных. Если даны процедуры S1 . . 8 и задана цель отсортировать список 2.1.3.nil, порождая перестановки до их проверки на упорядоченность, то проверку на упорядоченность можно начать с равным успехом и тогда, когда определены первые два элемента перестановки, и тогда, когда порождена вся перестановка целиком. Возможно более раннее выполнение процедурных вызовов обладает еще и тем достоинством, что позволяет обнаруживать неудачи столь рано, сколь это возможно. Рисунок 5.10 показывает эффективность раннего выполнения проверки на упорядоченность для устранения за один шаг всех перестановок, у которых первый элемент 2, а второй элемент – 1.

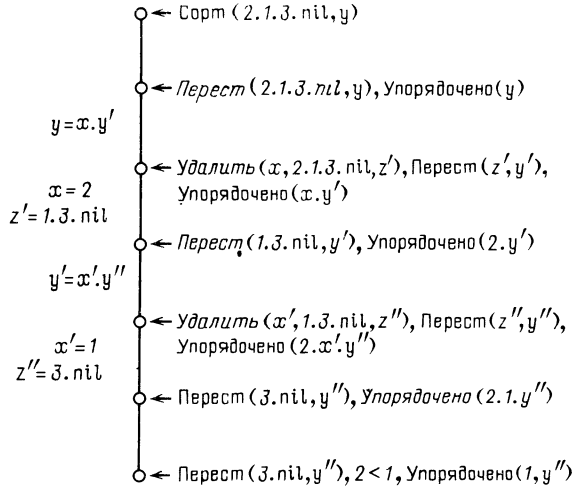


Рис. 5.10

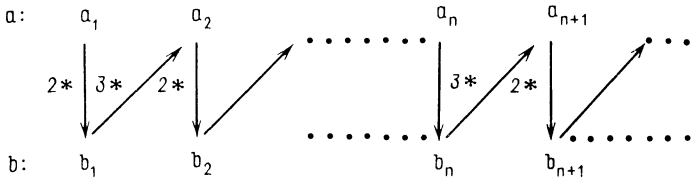


Рис. 5.11

Примером с более драматичным сюжетом может служить поведение задачи о допустимых парах; эта задача обладает неустранимым недетерминизмом первого рода, если процедурные вызовы обрабатываются по правилу "последний пришел первый ушел". Пару целочисленных списков (a, b) назовем допустимой, если оба списка имеют одинаковую длину и для каждого i верно:

если a_i есть i-й элемент a
и
если b_i есть i-й элемент b,
то
$b_i = 2 * a_i$
и
$a_{i+1} = 3 * b_i$

Графически эта связь показана на рис. 5.11. Ниже приведены задающие эту связь клаузы, в которых списки заданы значениями термов:

Допуст (x, y) ← Удвоенный (x, y), Утроенный (x, y)
Удвоенный (nil, nil) ←
Удвоенный (x . y, u . v) ← Умножено (2, x, u), Удвоенный (y, v)
Утроенный (x . nil, u . nil) ←
Утроенный (x . y . z, u . v) ← Умножено (3, u, y), Утроенный (y, z, v)

Рассмотрим задачу порождения такой допустимой пары списков, в которой первый список начинается с числа 1:

← Допуст (1 . y, w)

Эта программа обладает неустранимым недетерминизмом первого рода, если процедурные вызовы выполняются по правилу "последний пришел первый ушел", дополненному обязательным завершением одного вызова до начала работы второго. Но она получает воображаемую определенность первого рода, если процедурные вызовы выполняются сразу как только получен достаточный перечень входных параметров. Тогда оба процедурных вызова начинают работать как два взаимодействующих последовательных процесса. Как только хотя бы один из двух процессов, Удвоенный или Утроенный, получает достаточную входную информацию, он начинает исполняться и исполняется до тех пор, пока такая информация не понадобится снова. А тем временем исполняющийся процесс вырабатывает достаточную выходную информацию для того, чтобы второй процесс продолжил работу (рис. 5.12).

Сопрограммы, которые совместно вырабатывают и потребляют данные, могут быть написаны на языках программирования типа Симула. Такие

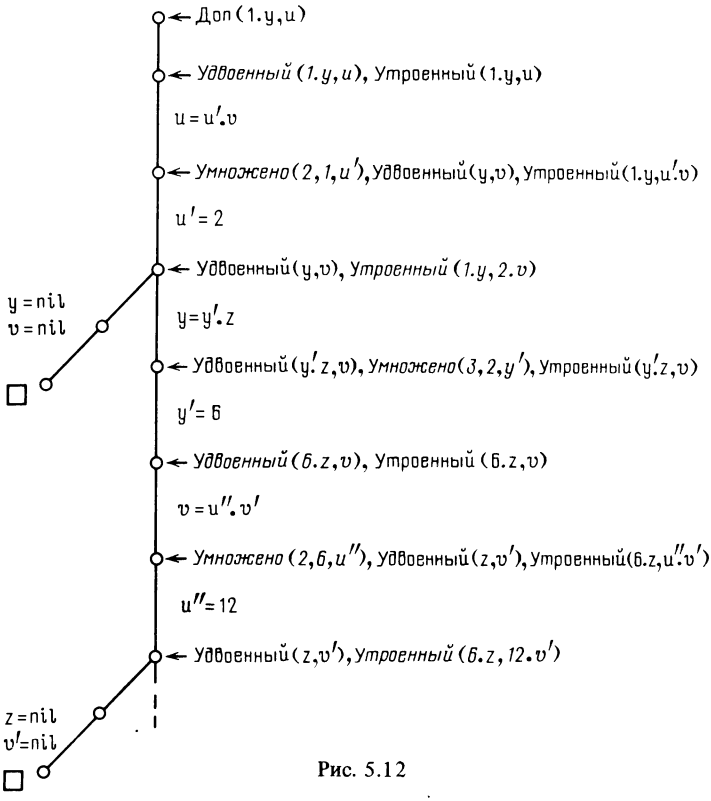


Рис. 5.12

сопрограммы, однако, и синтаксически и семантически отличаются от обычных процедур. У более недавних систем, в которых процедуры вызываются по мере необходимости [Henderson, Morris 1976] и активизация процессов управляется потоком данных [Kahn 1974] [Friedman, Wise 1978], имеется определенное сходство с процедурами в логических программах. Там стратегия выполнения процедурных вызовов определяется не программой, а исполняющей системой.

Выполнение программ способом снизу вверх

Процедурная интерпретация клазу Хорна является изначально нисходящей. Но иногда можно процедурно интерпретировать и восходящий вывод. Хотя в общем случае более эффективной для компьютера является нисходящая интерпретация клауз Хорна, людям кажется доходчивей восходящая интерпретация. Более того, иногда оказывается, что даже выполнять программы выгоднее снизу вверх, а не сверху вниз.

Например, студенту математического факультета легче воспринимать рекурсивное определение факториала

```
Факториал 0 есть 1 ←
Факториал x есть u ← y + 1 = x,
                     факториал y есть v,
                     x * v = u
```

снизу вверх, как задающее последовательность утверждений

Факториал 0 есть 1 ←
Факториал 1 есть 1 ←
Факториал 2 есть 2 ←
Факториал 3 есть 6 ←

и т.д., нежели воспринимать его сверху вниз, как сводящее цели к подцелям. В этом примере у восходящего вывода имеется и, так сказать, вычислительный привкус. Действительно, этот вывод исполняется как итеративный вычислительный процесс, накапливающий факториал последовательно увеличивающихся чисел до тех пор, пока не будет получен факториал требуемого числа.

Определение чисел Фибоначчи тоже может быть построено с большей эффективностью снизу вверх, чем сверху вниз:

0-е число Фибоначчи есть 1 ←
1-е число Фибоначчи есть 1 ←
(u + 2)-е число Фибоначчи есть x ←
    (u + 1)-е число Фибоначчи есть y
    u-е число Фибоначчи есть z
    y + z = x

Здесь термы u + 2 и u + 1 суть выражения, которые должны быть вычислены раньше, чем термы, представляющие структуры данных. Это обозначение служит *сокращением* для такого, которое содержит в теле явные процедурные вызовы для вычисления u + 2 и u + 1.

Если поиск u + 1-го числа Фибоначчи интерпретировать сверху вниз, то он сводится к задаче поиска u-го числа Фибоначчи. Нисходящее вычисление имеет вид И дерева, вершинами которого служат процедурные вызовы, причем их количество оценивается как экспоненциальная функция от u. Задача подсчета 4-го числа Фибоначчи, например, приводит к

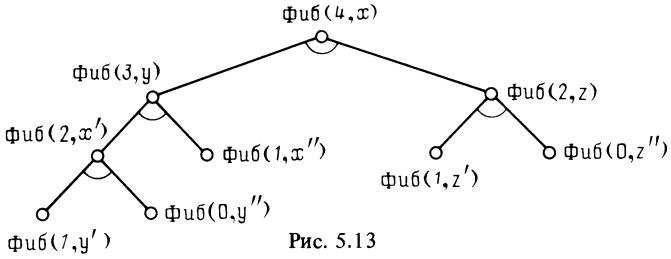


Рис. 5.13

построению дерева, которое, если не учитывать сложения, содержит 9 целей и подцелей (рис. 5.13).

Здесь Фиб(u, x) обозначает, что u-е число Фибоначчи есть x. Реализация этого же определения снизу вверх порождает такую последовательность утверждений:

0-е число Фибоначчи есть 1 ←
1-е число Фибоначчи есть 1 ←
2-е число Фибоначчи есть 2 ←
3-е число Фибоначчи есть 3 ←

и т.д.

Количество шагов вычисления u-го числа Фибоначчи при восходящем выполнении оказывается линейной функцией от u.

Кроме того, из примера видно, что восходящее выполнение требует меньше ресурсов памяти для хранения, чем нисходящее выполнение. Нисходящее выполнение использует объем памяти, пропорциональный u, в то время как для восходящего выполнения нужно хранить лишь два утверждения, и поэтому достаточно использовать небольшой постоянный объем памяти. То, что только два утверждения достаточно хранить во время выполнения снизу вверх, является следствием правил удаления в процедуре доказательства посредством графа соединений (гл. 8).

Отметим, что эффективность нисходящего выполнения может быть приближена к эффективности восходящего, если идентичные процедурные вызовы (типа того, что u-е число Фибоначчи равно z и u-е число Фибоначчи равно z') выполнять только однажды. Такое нисходящее выполнение оказывается расширением алгоритма разбора Эрли [Earley 1970], что было доказано Уорреном (неопубликовано).

Итерация в обычных языках программирования имеет три различные интерпретации в логических программах. В классической интерпретации итерация рассматривается как специальный случай нисходящего исполнения рекурсивных определений. Например, итерация

Делать P, повторяя Q, пока R

может быть задана так

$$P(x) \leftarrow R(x),$$
$$P(x) \leftarrow Q(x, x'), P(x')$$

где x – это входной параметр, который управляет количеством повторений цикла, а $R(x)$ и $Q(x, x')$ истинны для разных x. Рекурсия – это вид итерации, когда $Q(x, x')$ выполняется до $P(x')$. Следовательно, каждая новая подцель $P(x')$ может замещать предыдущую подцель $P(x)$. Выполнение поэтому требует лишь постоянный объем памяти для текущей подцели.

Интерпретация итерации в виде нисходящего выполнения определенных видов рекурсивных определений оказывается единственной интерпретацией, возможной в обычной модели рекурсивных вычислений. В логических программах, однако, можно также рассматривать итерацию либо как последовательный поиск в пространстве альтернативных откликов на процедурные вызовы, либо как восходящее выполнение рекурсивных определений.

Прагматическая сторона логических программ

В восприятии логики лишь как языка спецификаций, предложения которого обладают семантическим содержанием, но лишены прагматической оценки, заключена весьма распространенная ошибка. Такая точка зрения оказывается достаточно ограниченной. Применять логику, игнорируя ее прагматический аспект, значит делать информацию практически бесполезной.

Два разных предложения, к примеру, могут выражать одну и ту же информацию и поэтому обладать одним и тем же значением. Но одно из них может быть полезным для поиска решений задач, а другое – бесполезным.

Прагматический аспект логики хорошо иллюстрирует задача сортировки, изученная ван Эмденом [Van Emden 1977]. С одной стороны, имеется программа сортировки списков S1 . . 8

Сорт (x, y) ← Перест (x, y), Упорядочено (y)

хорошо описывающая задачу, но бесполезная как действующая программа. Даже такое планирование процедурных вызовов, при котором Упорядочено (y) используется для просмотра частичных результатов Перест (x, y), безнадежно неэффективно (затрачивает время 2^n для того, чтобы отсортировать список длины n). С другой стороны, даже простое последовательное выполнение процедурных вызовов в программе

```
Сорт*(nil, nil) ←
Сорт*(x, y, z) ← Разбиение (x, y, u, v),
                 Сорт*(u, u'),
                 Сорт*(v, v'),
                 Присоединить (u', x . v', z)
```

задает эффективный алгоритм быстрой сортировки [Hoare 1961], затрачивающий время порядка n * log(n).

Здесь предполагается, что Разбиение (x, y, u, v) выполняется, когда u есть список всех элементов y, меньших или равных x, а v есть список всех элементов y, больших, чем x.

Сорт и Сорт* *эквивалентны* в том смысле, что Сорт(s, t) и Сорт*(s, t) истинны для одних и тех же пар термов s и t Программа Сорт полезна в качестве определения свойства отсортированности, но бесполезна для практической сортировки списков. Программа Сорт* в этом отношении эффективна, но ее корректность гораздо менее очевидна.

В общем случае, любую конкретную задачу можно задать многими различными способами. Два представления задачи поиска пути (одно, основанное на предикате Достичь(x), и другое, основанное на предикате Достичь*(x, y)) могут быть обобщены и для других задач. Даже определение факториала можно задать двумя способами. То, что было раньше, соответствует формулировке задачи поиска пути с одноместным предикатом. А определение, которое мы приведем сейчас, соответствует формулировке с двуместным предикатом.

Факт*(x, y, u, v)

означает, что факториал x есть y, если факториал u есть v

```
Факт*(u, v, u, v) ←
Факт*(x, y, u, v) ← u + 1 = u',
                    u' * v = v',
                    Факт*(x, y, u', v')
```

Теперь для того, чтобы найти факториал целого числа, заданного термом t, оказывается достаточным применить единственное целевое предложение, правда не только для выражения цели, но и для указания того, что факториал 0 есть 1:

Факт*(t, y, 0, 1)

Новое определение факториала при нисходящем выполнении проходит такие же итерации, как и первоначальное при смешанном нисходяще–вос-

ходящем исполнении. Корректность старой формулировки более очевидна, а для новой формулировки легче обеспечить более высокую эффективность при более жестких технических ограничениях на возможности поиска решений.

Отделение структур данных

Правила построения хорошо структурированных программ требуют, чтобы данные были отделены от процедур, которые обрабатывают и манипулируют этими данными. Отделение структур данных от процедур означает, что описание данных можно менять, не меняя процедуры более высоких уровней. В связи с этим, например, заменяя неэффективные структуры данных на более эффективные, можно без особого труда повышать общую эффективность программы. В больших программных комплексах информация, которая поддерживается структурами данных, часто становится известной только на конечной стадии проектирования комплекса. В этом случае отделение структур данных от процедур позволяет писать программы верхних уровней не дожидаясь полного определения структур данных.

Средства хранения и поиска данных автоматически отделяются от процедур, если данные представлены в виде отношений, как это было в примере о родственных связях. Но если данные в отличие от этого случая представлены термами, то ответственность за отделение данных от процедур возлагается на программиста.

Примером может служить задача распознавания арки. Заменим старую формулировку, в которой были смешаны процедуры и структуры данных, на новую, в которой они будут отделены друг от друга. Смысловые характеристики структур данных в процедурах верхнего уровня мы заменим процедурными вызовами, которые обеспечат доступ к данным, их вычисление и формирование.

```
Арка (x) ←   Блок (v), Башня (u),
             Башня (w), На (v, u),
             На (v, w),
             Левый (x, u), Правый (x, w),
             Верхний (x, v),
Башня (x) ←  Блок (x)
Башня (x) ←  Блок (u), Башня (v), На (u, v),
             Верхний (x, u), Нижний (x, v)
На (x, y) ←  Верхний (y, u), На (x, u)
```

Здесь отношения Верхний, Левый, Правый и Нижний задают интерфейс между процедурами и структурами данных. Предполагается, что

```
Верхний (x, y) имеет место, когда y находится наверху x
Левый (x, y) имеет место, когда левая башня арки x
    есть y
Правый (x, y) имеет место, когда правая башня арки x
    есть y
Нижний (x, y) имеет место, когда основанием башни x
    является y
```

Структуры данных можно определить отдельно, задавая их интерфейсы с процедурами нисходящей программы:

Верхний (а (u, v, w), v) ←
Верхний (б (u, v), u) ←
Левый (а (u, v, w), u) ←
Правый (а (u, v, w), w) ←
Нижний (б (u, v), v) ←

В данном случае процедуры интерфейса задаются просто в виде утверждений. Но в других случаях они могли бы быть определены при помощи процедур более общего вида.

Сравнивая две формулировки программы распознавания арки, можно отметить и другое преимущество разделения процедур и структур данных: при наличии инфиксной нотации для предикатных символов и при удачно выбранных именах интерфейсных процедур программа, в которой обеспечена независимость структур данных, оказывается самодокументированной. В обычных программах, где смешаны структуры данных и процедуры, программист вынужден сам заботиться о средствах документирования, которые поясняют назначение структур данных и являются внешними по отношению к программе. В хорошо структурированных программах, когда структуры данных отделены от процедур, такое документирование обеспечивается интерфейсными процедурами и является составной частью программы.

Вопреки аргументам в пользу разделения процедур и структур данных программисты продолжают смешивать их ради уменьшения времени выполнения программ. Одним из способов примирения требования эффективности с правильным структурированием программ может служить возможность макропроцессирования, обеспечиваемая некоторыми языками программирования. Макропроцессирование выравнивает иерархию нерекурсивных процедурных вызовов за счет исполнения их во время компиляции перед решением самой поставленной задачи. Оно составляет одну из особенностей разработанной Берстэллом и Дарлингтоном системы улучшающих преобразований программ [Burstall, Darlington 1977].

Аналогом макропроцессирования в логике может служить рассуждение снизу вверх, дополненное правилами удаления клауз. Такие макросредства являются специальным случаем более общих методов, обеспечиваемых процедурами доказательства с помощью графов соединений (гл. 8). В случае программы распознавания арки первоначальная формулировка может быть получена из новой просто при помощи выполнения снизу вверх интерфейсных процедурных вызовов.

Структуры данных: термы versus отношения

Данные в логических программах можно задавать либо как значения термов (как в примерах об арке или о присоединении), либо как значения отношений (как в примерах о грамматическом разборе или о родственных связях).

Когда данные представлены термами, вход программы обычно определяется термом в начальном целевом предложении. Нисходящее выполне-

ние зависит от задачи и похоже на рекурсивное вычисление в обычных языках программирования. Восходящее выполнение, хотя и ведет себя иногда подобно итерации как в примерах о факториале и о числах Фибоначчи, чаще всего не зависит от задачи и вызывает стремительный рост объема вычислений за исключением, правда, тех случаев, когда его можно обуздать с помощью глобальных предположений о решаемой задаче. Глобальные стратегии для поиска решений будут изучены в гл. 9.

Когда данные представлены посредством отношений (определенных утверждениями и процедурами), вход программы обычно задается утверждениями. И нисходящие и восходящие способы выполнения зависят от задач. Нисходящие методы перерабатывают вход, а восходящие методы манипулируют им, вырабатывая тем самым новые данные из исходных.

Обычно всегда можно задавать данные как значения термов. Например, в Лиспе все данные задаются с помощью константных символов и единственного двуместного функционального символа "cons"*). В теории рекурсии все данные представляются посредством натуральных чисел благодаря использованию единственного константного символа 0 и унарного функционального символа "s". Поучительно сравнить первоначальную формулировку задачи грамматического разбора с формулировкой, в которой данные представлены термами.

Предл (x) ← Гс (y), Гг (z), Присоединить (y, z, x)
Гс (x) ←Арт (y), Прил (z), Сущ (v),
Присоединить (y, z, u),
Присоединить (u, v, x)
Гг (x) ←Вспом (y), Глаг (z)
Присоединить (y, z, x)
Арт (the . nil) ←
Прил (slithy . nil) ←
Сущ (toves . nil) ←
Вспом (did . nil) ←
Глаг (gyre . nil) ←

И задание исходной строки слов, и постановка задачи доказательства того, что это – предложение, совмещены в начальном целевом предложении

← Предл (the . slithy . toves . did . gyre . nil)

Отметим, что процедурные вызовы Присоединить не имели соответствия в первоначальной формулировке задачи грамматического разбора. Если данные представляются с помощью утверждений, то в программу заложен прямой доступ к данным, подобный тому, что осуществляется посредством массивов в обычных языках программирования. Если данные представлены посредством термов, то к специальным процедурам типа Присоединить нужно добавлять средства поддержки доступа к структурам данных.

Можно описывать данные и целиком с помощью отношений, как в реляционных базах данных [Codd 1970]. Так, вместо того, чтобы представ-

*) Предикат Cons, одноименный с этим функциональным символом Лиспа, переведен нами в этой книге как "Сост". – *Примеч. пер.*

лять список

a, c, b, a

термом

cons(a, cons(c, cons(b, cons(a, nil))))

или в виде

a . c . b . a . nil,

мы может придумать ему имя, например A, и представить его утверждениями:

Элемент (A, 1, a) ←
Элемент (A, 2, c) ←
Элемент (A, 3, b) ←
Элемент (A, 4, a) ←
Длина (A, 4) ←

где Элемент (x, y, z) означает, что z есть y-й элемент x, а Длина (x, y) означает, что y есть длина x.

Вместо того, чтобы записывать явную рекурсивную программу для обращения списков, например

Обратить (nil, nil) ←
Обратить (x . y, z) ← Обратить (y, u),
Присоединить (u, x . nil, z)

или, что более эффективно

Обратить (x, y) ← Обр (x, nil, y)
Обр (nil, y, y) ←
Обр (x.y, z, u) ← Обр (y, x . z, u),

мы можем записать и нерекурсивную программу

Элемент (обр (x), u, y) ← Элемент (x, v, y),
Длина (x, w),
u + v = w′
w + 1 = w′
Длина (обр (x), y) ← Длина (x, y)

Здесь терм обр (x) обозначает список, который является обращением x.

Если данные представлены посредством термов, то в программе необходимо определить средства хранения и поиска данных, а также средства отделения данных от процедур верхних уровней. К данным, расположенным ближе к "поверхности" терма, можно подобраться проще, чем к данным, расположенным глубже, внутри. Если данные представляются посредством отношений, то программа определяет их на уровне абстракций, не зависящем ни от схемы хранения, ни от схемы поиска, адаптируемых программной системой. Если отношение задается утверждением, то в программе обеспечивается прямой доступ к информации.

Формальные средства описания баз данных и языки программирования

Общепринятые формальные средства описания баз данных отличаются от формальных средств, используемых в языках программирования. В противовес этому, логика всегда одна и та же, независимо от того, используется ли она в базах данных, или для организации запросных систем к базам данных, или в программах, или для задания ограничений целостности в базах данных, или для задания программных спецификаций. Действительно, во всех этих случаях, и в особенности, если в качестве данных используются отношения, применение логики стирает обычные различия между базами данных и программами. Общие правила описания данных становятся неотличимыми от программных процедур, а ограничения целостности в базах данных приобретают черты программ.

Привычное различие между базами данных и программами не заложено в природе вычислительных задач. Например, описание задачи символьного интегрирования на языке логики (типа того, что было сделано на Прологе Бергманом и Кануи [Bergman, Kanoui 1973]) можно рассматривать и как базу данных, и как программу. Связь между функцией и ее первообразной можно задавать при помощи утверждений, например так

sin(x) есть первообразная для cos(x) по x

или при помощи общих правил, например так:

u + v есть первообразная для u′ + v′ по x,
если u есть первообразная для u′ по x,
а v есть первообразная для v′ по x

Определение отношения можно воспринимать и как определение рекурсивной процедуры, и как описание базы данных при помощи комбинации явных утверждений и неявных правил.

Считается, что необходимость более тесного сочетания баз данных и программ, нежели это было принято в привычных формализмах, была впервые оценена в кругу разработчиков баз данных. Проектирование языка программирования [Zloof, de Long 1977], базирующегося на query by example *), является одной из наиболее значительных разработок этого рода.

Алгоритм = Логика + Управление

Обычные *алгоритмы* и программы, записанные на обычных языках программирования, сочетают в себе *логику* используемой при поиске решений информации и *управление* способом использования этой информации. Эта связь может быть символически выражена таким уравнением

Алгоритм = Логика + Управление (А = Л + У)

В логических программах заключен только *логический компонент* Л алгоритмов. Обязанности *управляющего компонента* берет на себя исполняющая система, подчиняющаяся либо своим собственным автономным

*) Запрос по образцу – *Примеч. пер.*

управляющим решениям, либо управляющим инструкциям, заданным программистом.

Концепция отделения логики от управления обладает такими преимуществами:

(1) Алгоритмы могут создаваться методом последовательного улучшения за счет проектирования логического компонента до управляющего компонента.

(2) Алгоритмы могут быть улучшены за счет улучшения их управляющего компонента безо всякого изменения логического компонента.

(3) Алгоритмы могут быть порождены из спецификаций, могут быть верифицированы и могут быть преобразованы к более эффективному виду без учета управляющего компонента, только за счет применения дедуктивных правил вывода к логическому компоненту.

(4) Неопытные программисты и пользователи баз данных могут свести свое взаимодействие с вычислительной системой к определению логического компонента, оставляя формирование управляющего компонента компьютеру.

Систематическому проектированию правильных структурированных программ присуща спецификация логического компонента перед управляющим компонентом. Логический компонент формирует проблемнозависимую часть алгоритма. Он одновременно определяет смысл алгоритма и влияет на его поведение. С другой стороны, управляющий компонент определяет общую стратегию поиска решений. Он оказывает влияние только на эффективность алгоритма, но не на его смысл.

Различные алгоритмы $А_1$ и $А_2$, полученные применением разных управляющих компонентов к одному и тому же логическому компоненту Л, эквивалентны в том смысле, что они дают решение одной и той же задачи с одним и тем же результатом. Символически

$А_1$ и $А_2$ эквивалентны, если

$$А_1 = Л + У_1$$

и

$$А_2 = Л + У_2$$

Эквивалентность различных алгоритмов с одним и тем же логическим компонентом можно использовать для повышения эффективности алгоритма за счет улучшения его управляющего компонента без изменения логического компонента. В частности, замена восходящей стратегии управления нисходящей часто, хотя не всегда, повышает эффективность, в то время как замена нисходящего последовательного исполнения процедурных вызовов нисходящим параллельным исполнением почти всегда улучшает качество алгоритма и никогда не ухудшает его.

Аргументы в пользу отделения логики от управления аналогичны аргументам в пользу отделения процедур от структур данных. Когда процедуры отделяются от структур данных, становится возможным отличать функции, выполняемые структурами данных, от способов выполнения этих функций. Алгоритм можно улучшить при помощи замены неэффективных структур данных более эффективными, обеспечивая при этом выполнение новыми структурами данных тех же функций, которые выполнялись старыми. И точно так же, когда логика отделена от управления, становится

возможным различать действия алгоритма, определенные логическим компонентом, и способ выполнения этих действий, определенный управляющим компонентом. Алгоритм может быть улучшен за счет замены неэффективной управляющей стратегии более эффективной, обеспечивая при этом неизменность логического компонента. В обоих случаях легче один раз определить смысловую часть алгоритма, а затем повышать его эффективность, не меняя смысл.

Отделение логики от управления упрощает проблему соотнесения программ со спецификациями. Оказывается возможным, нисколько не затрагивая управляющий компонент, использовать дедуктивные правила для доказательства корректности логического компонента, поскольку она следует из его спецификаций. Такие же методы дедукции могут быть применены и для порождения логической программы из ее спецификации или для преобразования неэффективной программы в более эффективную. Эти методы были разработаны для логических программ Бибелом [Bibel 1976a, 1976b, 1978], Кларком и Тарнлундом [Clark, Tarnlund 1977], Кларком и Сикелем [Clark, Sickel 1978] Кларком и Дарлингтоном [Clark, Darlington 1978] и Хоггером [Hogger 1979]; они аналогичны методам, разработанным для рекурсивных уравнений Берстэллом и Дарлингтоном [Burstall, Darlington 1977] и методам, разработанным для Лиспа Манной и Уолдингером [Manna, Waldinger 1978]. Краткий экскурс в эти методы мы совершим в гл. 10, когда будем обсуждать связь стандартной и клаузальной форм логики.

Разложение алгоритмов на логический и управляющий компоненты обеспечивает два разных метода повышения эффективности алгоритма. Если имеется фиксированный управляющий компонент, включенный в состав управляющей системы, обладающей ограниченными возможностями, то эффективность можно повысить, заменяя формулировку логического компонента; или, если имеется фиксированный логический компонент, то алгоритм можно улучшить за счет расширения возможностей исполняющей системы по поиску решений. Замена логического компонента является полезной краткосрочной стратегией, поскольку представление задачи, вообще говоря, менять проще, чем менять исполняющую систему. С другой стороны, замена управляющего компонента является более качественной долгосрочной стратегией, поскольку улучшение исполняющей системы улучшит ее работу для большого числа различных задач.

Спецификация управляющего компонента

Управляющий компонент может задаваться программистом посредством отдельного управляющего языка; можно его определить и средствами самой управляющей системы. Применение отдельного управляющего языка позволяет программисту построить специальную исполняющую систему для выполнения данной программы и является удобным для опытных программистов. С другой стороны, возложение функций управления на исполняющую систему освобождает программиста от обязанности заботиться об определении управляющих воздействий и является удобным для неопытных программистов, для случайных пользователей базы данных

и даже для опытных программистов на ранних стадиях разработки программ.

Но следует отметить, что полностью автономная, удовлетворяющая всем потребностям управляющая стратегия еще не разработана. Задача разработки эффективных алгоритмов планирования процедурных вызовов, в частности, еще находится в процессе решения. Принцип откладывания, в соответствии с которым происходит задержка выполнения в случае, когда процедурный вызов может быть выполнен несколькими способами, и дополняющий его принцип, в соответствии с которым процедурный вызов исполняется как только он может быть выполнен не более, чем одним способом, во многих случаях работают эффективно. Но они не приводят к хорошим результатам, скажем, когда все процедурные вызовы обладают недетерминизмом первого рода. Для управления выполнением процедурных вызовов, исполняемых как сопрограммы, в системе Пролог в Имперском колледже были предусмотрены аннотации [Clark, McCabe 1979]. Они похожи на аннотации для рекурсивных уравнений, предложенные Шварцем [Shwarz 1977].

Автономные поисковые стратегии были разработаны также и для пространств нисходящего и для пространств восходящего поиска при доказательстве теорем. Эти стратегии используют специальные упорядочения или оценочные функции для управления порождением клауз в поисковом пространстве. Аргументы против таких поисковых стратегий высказывались Хейесом [Hayes 1973]. Он утверждает, что информация, обеспечиваемая этими стратегиями, не приведет к эффективному поиску решений и предлагает более удобную информацию, которую программист может использовать во вспомогательном языке управления. Примером такой информации может служить изображение конкретного отношения функцией некоторых аргументов.

Управляющие примитивы для направляющих воздействий на поисковые стратегии были предусмотрены в таких языках программирования как Плэннер [Hewitt 1969], Микроплэннер [Sussman, Winograd, Charniak 1971], Коннайвер [Sussman, McDermott 1972], Поплер [Davies 1973], Сейл [Feldman и др. 1972], QA4 [Rulifson и др. 1973], QLISP [Reboh, Sacerdoti 1973]. Среди возможностей Плэннера и Микроплэннера имеется, в частности, средство, позволяющее программисту специфицировать порядок, в котором должны использоваться процедуры при исполнении данного процедурного вызова. Такая информация может оказаться полезной в системах диагностики, например, когда программист знает, что симптом P чаще вызывается причиной Q, чем причиной R. Это можно сообщить системе-решателю при помощи рекомендации о том, что процедуру

$$P \leftarrow Q$$

следует рассматривать до процедуры

$$P \leftarrow R.$$

И автономные, и задаваемые пользователем стратегии управления порядком выполнения включались в методы доказательства теорем и в программные языки искусственного интеллекта. В языках программиро-

вания семейства Плэннер порядок, в котором выполняются процедуры, задается предварительно посредством типов, определяемых для описываемой теоремы (консеквентный (целевой) тип при направлении сверху вниз и записывающий (антецедентный) тип при направлении снизу вверх). Более того, каждому процедурному вызову назначается тип процедуры, к которым он может обращаться. Автономные, определяемые системой стратегии управления порядком выполнения, чаще используются при доказательстве теорем или исследований операций. Кроме того, применялись некоторые стратегии, обеспечивающие наименьший текущий коэффициент ветвления. И определяемые системой, и определяемые пользователем методы управления порядком выполнения рассматриваются в гл. 8, в которой описывается процедура доказательства методом графов соединений.

Разделение логики, управления и средств реализации управления для решателя задач, несмотря на возникающие здесь трудности, почти целиком вошло в системы баз данных. Если, к примеру, дана база данных, в которой определены отношения:

Поставщик (x, y, z):	поставщик номер x с именем y и состоянием z,
Деталь (x, y, z):	деталь номер x с именем y и ценой за штуку z,
Поставка (x, y, z):	поставщик номер x поставляет деталь номер y в количестве z,

то запрос: "Кто поставляет книги?"

← Ответ (y)
Ответ (y) ← Поставщик (x, y, z), Поставляет (x, u, v), Деталь (u, книга, w)

задает только логический компонент задачи. Система поиска данных нуждается в том, чтобы, в целях большей эффективности, процедурный вызов Деталь (u, книга, w) (содержащий входное значение) выполнялся бы первым. Если дан аналогичный запрос: "Какие детали поставляет Джон?"

← Ответ (y)
Ответ (y) ← Поставщик (x, Джон, z), Поставляет (x, u, v), Деталь (u, y, w),

то необходимо распознать, что первым исполняемым вызовом должен стать вызов Поставщик (x, Джон, z).

Для неопытных пользователей баз данных желательно уметь выражать запросы на языках, как можно более близких к естественным. Поскольку логика берет начало из анализа естественного языка, неудивительно, что языки запросов к базам данных выражают только логический компонент алгоритмов. Сужение языков запросов до логического компонента имеет и другие достоинства. В этих условиях, например, схемы хранения и поиска можно изменять и улучшать в управляющем компоненте без воздействия на заданное в логическом компоненте представление пользователя о данных. В общем случае, чем выше уровень языка программирования и чем ниже уровень программиста, тем больше необходимость включения в систе-

му средств повышения эффективности и реализации управляющих воздействий на использование заданной информации.

Предположение о том, что

вычисление = управляемая дедукция

было впервые высказано Хейесом [Hayes 1973], а после него Бибелом [Bibel 1978], Ковальским [Kowalski 1976], Праттом [Pratt 1977] и Шварцем [Schwartz 1977]. Подобный этому тезис о том, что системы баз данных должны быть разложены на реляционный компонент, в котором определяется логика данных и на управляющий компонент, в котором задаются средства хранения и поиска данных, выдвигался Коддом [Codd 1970]. Аргументы Хьюитта [Hewitt 1969] в пользу языка программирования Плэннер, которые обычно воспринимаются как серьезный довод против логики, можно рассматривать более позитивно как довод в пользу разделения алгоритма на логический и управляющий компоненты.

Естественный язык = Логика + Управление*)

Процедурная интерпретация клауз Хорна примиряет классическую роль логики в анализе языка с восприятием предложений естественного языка как программ [Winograd 1972]. Как и в алгоритмах, в естественном языке сочетается логика и управление. Предложение

"Если Вы хотите нравиться Мери, *то* дарите ей подарки и будьте ласковы с животными"

сочетает декларативную информацию

"Вы нравитесь Мери, *если* Вы дарите ей подарки и ласковы с животными"

с указанием на то, что надо использовать нисходящий разбор для поиска решения задачи понравиться Мери путем сведения этой задачи к подзадачам дарения ей подарков и ласкового отношения к животным.

Упражнения

1. Пусть отношение Удалить задано следующими процедурами

D1 Удалить (x, x . y, y) ←

D2 Удалить (x, z . y, w) ← Удалить (x, y, w)

а) примените D1 . . 2 способом сверху вниз для удаления 1 из списка 2.1.nil; представьте полное пространство нисходящего поиска;

б) примените D1 . . 2 способом сверху вниз для добавления 1 к списку 2.nil, представьте полное пространство нисходящего поиска;

в) пусть Разл (x, y) истинно, когда x и y не идентичны; определите отношение Удал–вхождения (x, y, w), которое истинно, если w есть список, полученный путем удаления всех вхождений x из списка y.

---

*) Это очень сильное утверждение представляет собой точку зрения автора. Ее можно разделять, а можно и не разделять. – *Примеч. Д.А. Поспелова*

2. Опишите такое представление задачи поиска пути, которое позволяет найти список всех узлов на пути от одного узла до другого.

3. Переформулируйте задачу о сосудах с водой из гл. 4 так, чтобы в программе присутствовал распознаватель зацикливаний, благодаря которому программа могла бы эффективно выполняться даже если зацикливания не распознаются системой.

4. Пусть Разбиение (x, y, u, v) определено так:

```
Разбиение (x, y, u, v) ← Перемешивание (u, v, y),
                         Меньшие (x, u),
                         Большие (x, v)
Перемешивание (nil, v, v) ←
Перемешивание (v, nil, v) ←
Перемешивание (x . y, z, x.u) ← Перемешивание (y, z, u),
Перемешивание (y, x . z, x.u) ← Перемешивание (y, z, u),
```

где Меньшие (x, u) выполняется, если x $\leqslant$ всех элементов из списка u; Большие (x, u) выполняется, если x $\geqslant$ всех элементов из списка u; Перемешивание (u, v, y) выполняется, если списки u и v можно перемешать вместе так, чтобы получить список y.

Рассмотрите задачу ← Разбиение (s, t, u, v), в которой s и t являются входными параметрами, а из u и v требуется получить выходной результат:

а) определите отношения Меньшие и Большие рекурсивно, в терминах отношений $\geqslant$ и $\leqslant$;

б) опишите поведение процедур, приведенных в начале упражнения и в пункте а), если для поиска решений задачи сверху вниз используется бэктрекинг, выполняющий процедурные вызовы последовательно слева направо;

в) опишите более детерминированный способ выполнения процедурных вызовов для этой задачи;

г) переопределите Разбиение (x, y, u, v) так, чтобы поведение программы, аналогичное тому, что получилось в пункте в) этого упражнения, получилось за счет простого выполнения процедурных вызовов слева направо.

5. Пусть отношение Есть (x, y), выполняющееся, когда x есть начальный подсписок списка y (рис. 5.14), задано правилом:

```
Есть (x, y) ← Присоединить (x, z, y)
```

а) задайте Есть (x, y) рекурсивно, без использования Присоединить;

б) отношение Пдсп (x, y), которое истинно, если x есть произвольный подсписок y (рис. 5.15), можно задать так

```
Пдсп (x, y) ← Присоединить (u, x, w),
              Присоединить (w, v, y)
```

определите Пдсп (x, y) рекурсивно в терминах отношения Есть, не используя Присоединить;

в) опишите стратегию выполнения обоих процедурных вызовов в вышеприведенном определении Пдсп, которая ведет себя точно так же, как нисходящее последовательное выполнение рекурсивного определения Пдсп.

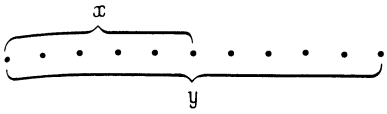


Рис. 5.14

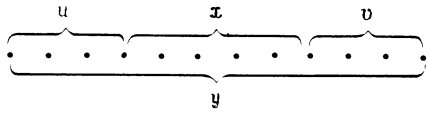


Рис. 5.15

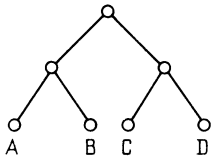


Рис. 5.16

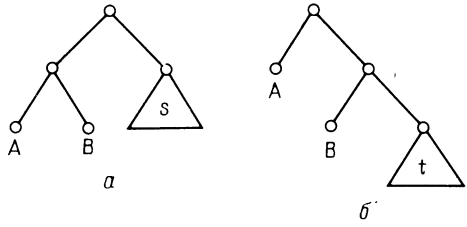


Рис. 5.17

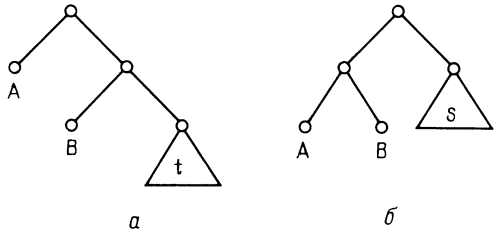


Рис. 5.18

6. а) Выразите задачу о 8 ферзях на языке клауз Хорна: если дана шахматная доска 8 × 8, то найти список из 8 позиций для ферзей так, что ни один ферзь не мог бы взять другой; один ферзь может взять другой, если оба находятся в одном горизонтальном ряду, либо в одном вертикальном ряду, либо на одной диагонали шахматной доски; предполагайте, что отношение Плюс(x, y, z) (x + y = z) уже задано свободными от переменных утверждениями;

б) модифицируйте задачу о 8 ферзях и покажите, строя полное пространство нисходящего поиска, что задача о 2 ферзях (постановка 2-х ферзей на доску 2 × 2) является неразрешимой; выполняйте процедурные вызовы так, чтобы минимизировать размер пространства поиска.

8. Любое бинарное дерево можно представить списком. Например, дерево на рис. 5.16 описывается термом

сост (сост (оконечная–вершина (A), оконечная–вершина (B))
сост (оконечная–вершина (C), оконечная–вершина (D))).

В общем случае связь Представляет(x, y), которая истинна тогда, когда дерево x представляется списком y, может быть задана такими клаузами:

R1 Представляет (nil, nil) ←
R2 Представляет (оконечная–вершина (x), x . nil) ←

R3 Представляет (сост (оконечная–вершина (x), y), x . z) ←
Представляет (y, z)

R4 Представляет (сост (сост (x, y), z), w) ←
Представляет (сост (x, сост (y, z), w)

а) определите отношение То₁–же–список (x, y), которое истинно, если деревья x и y представляются одним и тем же списком;

б) используйте процедуры R1 . . 4 и а) для сведения задачи доказательства того, что дерево на рис. 5.17, *а* и дерево на рис. 5.17, *б* представляются одним и тем же списком, к задаче доказательства того, что поддеревья s и t представляются одним и тем же списком;

в) воспользуйтесь процедурами R1 . . 4 и а), чтобы показать, что неразрешима задача доказательства того, что деревья, приведенные на рисунках 5.18, *а* и 5.18, *б*, представляются одним и тем же списком (здесь t и s – имена произвольных поддеревьев);

г) обобщите стратегию выполнения, развитую в б) и в), и опишите общую эффективную стратегию выполнения процедурных вызовов в R1 . . 4 и а), обеспечивающую преимущество совместных действий перед последовательными.

В *задаче построения плана* задаются начальное состояние, целевое состояние и множество действий, которые переводят одно состояние в другое. Содержанием этой задачи является построение плана, состоящего из подходящей последовательности действий, переводящей исходное состояние в целевое состояние.

Поэтому задача построения плана идентична задаче о пространстве состояний. Известно, что n-арное представление задачи о пространстве состояний может оказаться невозможным в случае, когда количество индивидов велико или неизвестно. В этой главе мы рассмотрим вариант бинарного представления задачи о пространстве состояний.

Использование логики и в n-арном и в бинарном вариантах приводит к *проблеме остова*: как выразить требование того, чтобы почти все предложения, истинные в одном состоянии, оставались бы истинными после действий*), выполняемых для перехода в другое состояние. Обычно считают, что такие требования не удается естественно выражать на языке логики и эффективно использовать.

Эта воображаемая неадекватность логики привела к созданию и развитию таких специальных систем как СТРИПС [Fikes, Nilsson 1971] и Плэннер [Hewitt 1969], предназначенных для выражения проблемы остова. Мы приведем довод в пользу того, что вполне удовлетворительное эквивалентное описание проблемы остова все же может быть создано на языке логики: за счет использования термов, именующих предложения и за счет использования аксиомы остова, которая задает предложения, сохраняющие свою истинность после выполнения действий, чаще – нисходящих, реже – восходящих.

Построение плана и мир блоков

Рассмотрим подробно простую задачу построения плана для мира блоков [Sacerdoti 1977]. Пусть имеются три переставляемых блока A, B, C и три неперставляемых места p, q, r. Расположение объектов в начальном и целевом состояниях приведено на рис. 6.1.

*) Другими словами, предполагается, что состояния, близкие в пространстве состояний, описывают близкие ситуации, что иногда не выполняется в реальной среде. – *Примеч. Д.А. Поспелова.*

Имеется единственное действие

перен(x, y, z),

которое вызывает перенос x с y на z. Действие может быть выполнено в данном состоянии, если

x переставляемо,
x и z свободны,
x находится на y,
x отлично от z.

В новом состоянии, полученном после выполнения этого действия, истинным становится новое предложение о том, что

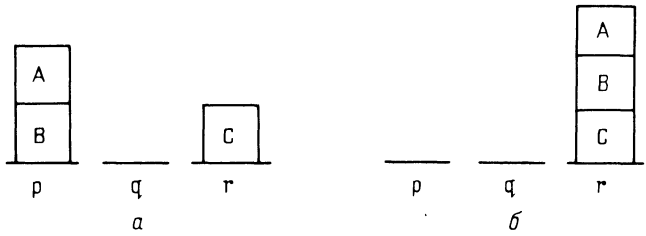
x на z,
y свободно.

Все предложения о старом состоянии, за исключением того, что

x на y,
z свободно,

продолжают оставаться истинными и в новом состоянии.

В общем случае действие можно определить, специфицируя его предусловия и постусловия. *Предусловия* суть предложения, которые должны быть истинными в состоянии до выполнения действия; *постусловия* суть предложения, которые должны быть истинными в новом состоянии после выполнения действия. Постусловия бывают двух видов: новые предложения, которые *добавляются* к описанию нового состояния и старые предложения из предыдущего состояния, которые продолжают сохранять



а *б*

Рис. 6.1

истинность. Старые предложения описываются при помощи *аксиомы остова,* которая говорит о том, что все предложения, истинные в старом состоянии, кроме явно указанных в качестве *удаляемых* исключений, остаются истинными и в новом состоянии. Явная спецификация предусловий, а также добавляемых и удаляемых предложений для каждого действия берет свое начало из СТРИПС.

Описание задачи блочного мира на языке клауз

Договоримся сейчас рассматривать состояния и предложения как индивиды и представлять их при помощи термов. То, что предложение x сохраняет истинность в состоянии y, можно описать при помощи бинарного отношения

Сохраняет(x, y)

Состояния именуются константными символами или составными термами. Удобно считать константный символ 0 именем начального состояния и зарезервировать терм

результат (u, v)

в качестве имени состояния, являющегося результатом применения действия u к состоянию v.

Представление предложений при помощи термов обсуждается в гл. 12, посвященной формализуемой части метаязыка. Пока же оказывается достаточно задать терм

на (x, y)

обозначающий предложение "x на y" и терм

свободный (x)

обозначающий, что x свободен. Можно пользоваться и другим представлением, в котором терм

атом (x, y)

обозначает атомарную формулу с предикатным символом x и списком аргументов y; это представление оказывается более гибким, но пока его применение не обязательно.

Пусть в приведенных ниже клаузах Возм(x) выражает факт, что x возможно, Перемещ(x) выражает факт, что объект x – перемещаемый, Разл (x, y) выражает факт, что x отличается от y.

Начальное состояние 0

(1) Возм (x) ←
(2) Сохраняет (на (A, B), 0) ←
(3) Сохраняет (на (B, p), 0) ←
(4) Сохраняет (на (C, r), 0) ←
(5) Сохраняет (свободный (A), 0) ←
(6) Сохраняет (свободный (q), 0) ←
(7) Сохраняет (свободный (C), 0) ←

Независимые от состояний утверждения

(8) Перемещ (A) ←
(9) Перемещ (B) ←
(10) Перемещ (C) ←

Целевое состояние

(11) ← Сохраняет (на (A, B), w),
Сохраняет (на (B, C), w),
Сохраняет (на (C, r), w),
Возм (w)

Пространство состояний и предусловия

(12) Возм (результат (перен (x, y, z), w)) ←
Возм (w),
Перемещ (x),
Разл (x, z),
Сохраняет (свободный (x), w),
Сохраняет (свободный (z), w),
Сохраняет (на (x, y), w)

Добавляемые предложения

(13) Сохраняет (на (x, z),
результат (перен (x, y, z), w)) ←

(14) Сохраняет (свободный (y),
результат (перен (x, y, z), w)) ←

Аксиома остова и удаляемые предложения

(15) Сохраняет (u, результат
(перен (x, y, z), w)) ←
Сохраняет (u, w),
Разл (u, на (x, y)),
Разл (u, свободный (z))

Клаузы (1) . . (6) описывают начальное состояние, клаузы (7) . . (10) описывают не зависящие от состояния факты о перемещаемости блоков, а клауза (11) описывает целевое состояние. Остальные клаузы описывают действия по переносу объектов с одного места на другое. Клауза (12) задает структуру состояний поискового пространства. Она задает предусловия, которые должны сохраняться до тех пор, пока можно применять действия к возможному состоянию для того, чтобы получить новое.

Клаузы (13) и (14) выражают постусловия, которые добавляются действием, а (15) описывает то, что сохраняется в новом состоянии, поскольку сохранялось в предыдущем состоянии и не было разрушено действием.

Отношение Разл(s, t) истинно для свободных от переменных термов s и t, когда s и t синтаксически различны. Полезно представлять, что клаузы (1) . . (15) дополняются бесконечным набором клауз вида

Разл (s, t) ←

для каждой пары несогласующихся термов s и t. Это же отношение можно эквивалентно определить при помощи аксиом

Разл ($f(x_1, \ldots, x_m)$, $g(y_1, \ldots, y_n)$) ←

для каждой пары различных функциональных символов f и g, включая случай m = 0 и n = 0, когда f и g суть константные символы, а также при помощи

Разл ($f(x_1, \ldots, x_m)$, $f(y_1, \ldots, y_m)$) ← Разл (x_i, y_i)

для каждого функционального символа f и для каждого аргумента этого символа f, исключая случай m = 0, когда f является константным символом. На практике более эффективно задавать Разл в виде отрицания

равенства

Разл (x, y) ← не (x = y)
x = x ←

определяя при этом, что не (x = y) выполняется, когда не удается доказать, что x = y. Такая интерпретация отрицания как неудачи и ее связь с обычной интерпретацией отрицания изучалась Кларком [Clark 1978] и будет обсуждаться нами в главе 11, в которой мы коснемся определений, выражаемых в терминах если–и–только–если.

Наша формулировка задачи построения плана похожа на формулировку, предложенную Грином [Green 1969b], которая базируется на пропозициональном исчислении Маккарти и Хейеса [McCarthy, Hayes 1969]. Однако наша формулировка все же отличается от их формулировки в том, что касается использования отношения Сохраняет. Они добавляют к отношениям параметр состояния, записывая, например, На (x, y, w) для выражения того, что x на y в состоянии w и Свободно (x, w) для выражения того, что x свободно в состоянии w. Использование отношения Сохраняет влечет за собой рассмотрение предложений как индивидов, что можно считать формализацией части метаязыка. Достоинства использования логики в качестве своего собственного метаязыка будут обсуждаться в гл. 12. Пока же достаточно отметить, что трактовка предложений как индивидов устраняет ту часть проблемы остова, которая выражается аксиомой остова.

Вместо того, чтобы использовать отдельную аксиому остова для каждого отношения, записывая, например,

На (u, v, результат (перен (x, y, z), w)) ← На (u, v, w),
Разл (u, x)
Свободный (u, результат (перен (x, y, z), w)) ← Свободный (u, w),
Разл (u, z)

достаточно задать единственную аксиому остова

Сохраняет (u, результат (v, w)) ←
Сохраняет (u, w),
Предохраняет (v, u)

где Предохраняет (v, u), означает, что действие v предохраняет истинность предложения u.

Использование отношения Предохраняет отделяет аксиому остова от спецификаций тех предложений, которые удаляются отдельными действиями. В случае действия "перен"

Предохраняет (перен (x, y, z), u) ← Разл (u, на (x, y)),
Разл (u, свободный (z))

Как мы увидим в следующей главе, клауза (15), в которой сочетаются аксиома остова и спецификации удаляемых предложений, может быть получена макропроцессированием процедурных вызовов в отношении Предохраняет. Макропроцессирование обеспечивает выполнение процедурных вызовов во время компиляции до того как поставлены задачи, а не во время исполнения в ходе попыток их решения. Это можно рас-

сматривать как одну из форм расходящихся от центра рассуждений, которые в свою очередь являются специальным случаем правила резолюций [Robinson 1965a]. Резолюция также обобщает нисходящий и восходящий выводы и применима к нехорновским клаузам.

Во многих случаях оказывается полезным различать два вида отношений: *первичные отношения*, которые на зависят от остальных отношений и *определяемые отношения*, которые можно задавать в терминах первичных отношений. В мире блоков отношение, которое истинно, когда один объект находится над другим, может быть определено в терминах первичного отношения, истинного, когда один объект располагается непосредственно на другом.

Сохраняет (над (x, y), w) ← Сохраняет (на (x, y), w)
Сохраняет (над (x, y), w) ← Сохраняет (над (x, z), w),
Сохраняет (над (z, y), w)

Добавляемые и удаляемые предложения достаточно сформулировать только в терминах первичных отношений. Результат действия на определяемые отношения задается результатом действий на первичные отношения и определения определяемых отношений в терминах первичных. Классификация отношений и их использование при формировании плана были предложены в СТРИПС.

Мы рассматривали отношения На и Свободный как первичные. Более естественным будет, однако, определить отношение Свободный в терминах отношения На.

Сохраняет (свободный (y), w) ← для всех x не–Сохраняет
(на (x, y), w)

Такую возможность мы обсудим в главе 11, когда введем если–и–только–если определения и интерпретацию отрицания как неудачи.

Логика задачи мира блоков отделена от ее использования. Клаузы можно использовать либо сверху вниз, либо снизу вверх. Можно их использовать также и в смешанном направлении. Если аксиома состояния пространства (12) используется снизу вверх, то система поиска решений ведет рассуждения в прямом направлении от начального состояния, получая новые состояния из старых до тех пор, пока не будет порождено целевое предложение. Если аксиома используется сверху вниз, то система поиска решений строит рассуждения в обратном направлении от целевого предложения до тех пор, пока не будет порождено начальное состояние.

Вторая часть проблемы остова возникает, когда аксиома остова (15) используется снизу вверх для вывода из утверждения о том, что некое данное предложение истинно в некоем данном состоянии, того что оно истинно в следующем состоянии. В более реальных задачах построения планов, нежели задача о мире блоков, для описания типичного состояния приходится использовать много разных утверждений, большая часть которых не имеет отношения к задаче. В таких ситуациях оказывается невозможным с точки зрения вычислений использовать аксиому остова снизу вверх для копирования предохраняемых фактов от состояния к состоянию.

И Плэннер и СТРИПС решают проблему остова отказываясь от аксиомы остова и используя вместо этого специальные процедуры. Подобные же результаты можно получить, сохраняя аксиому остова, но придавая ей нисходящую интерпретацию: для того, чтобы определить, сохраняется ли истинность предложения u в состоянии результат(v, w) надо

(i) показать, что u добавляется действием v;

(ii) иначе, если u не удаляется действием v, проверить, сохраняется ли истинность u в предыдущем состоянии w.

Изменение направления применения аксиомы остова оказывается частным случаем общей стратегии улучшения алгоритма путем улучшения его управляющего компонента без изменения логического компонента.

Теперь проиллюстрируем различные решения, получающиеся для мира блоков за счет различных способов использования аксиомы пространства состояний и аксиомы остова.

Восходящее применение аксиомы пространства состояний (12)

Рисунок 6.2 показывает часть поискового пространства состояний, полученного в результате применения (12) снизу вверх. Различные узлы описывают различные состояния. Однако различные состояния, помеченные одним и тем же номером, характеризуются одинаковыми предложениями. В данном случае избыточность связана с тем, что не удобно связывать один и тот же объект дважды в ряду.

Утверждения, порождаемые восходящим применением аксиомы пространства состояний, описывают поисковое пространство состояний, пока-

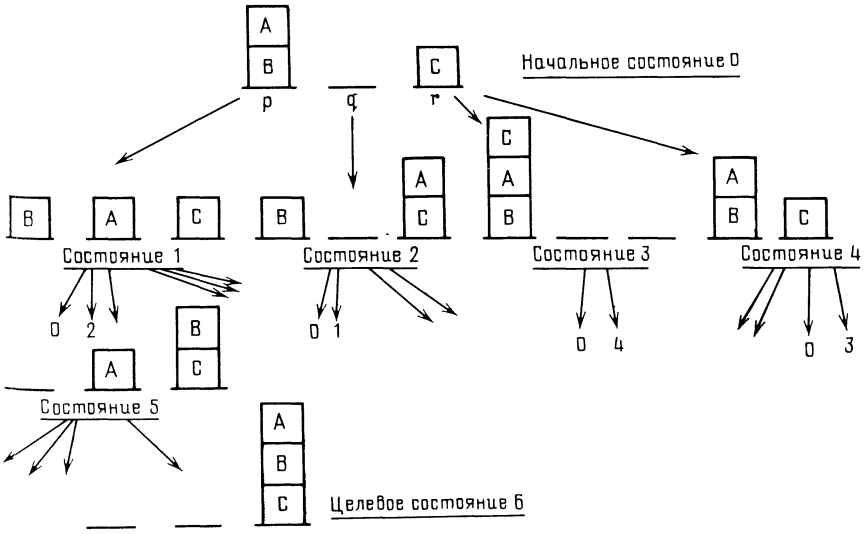


Рис. 6.2

занное на рис. 6.2; они независимы от направления применения аксиомы остова.

Восходящее применение аксиомы остова (15)

Приведенные ниже утверждения, связанные с состояниями, принадлежащими пути решения, порождены восходящим применением аксиомы остова:

Сохраняет (на (B, p) 1) ← Сохраняет (на (C, r), 1) ←
Сохраняет (на (A, q), 5) ← Сохраняет (на (C, r), 5) ←
Сохраняет (на (B, C), 6) ← Сохраняет (на (C, r), 6) ←
Сохраняет (свободный (A), 1) ← Сохраняет (свободный (C), 1) ←
Сохраняет (свободный (B), 5) ← Сохраняет (свободный (A), 5) ←
Сохраняет (свободный (A), 6) ← Сохраняет (свободный (p), 6) ←

Добавочные утверждения:

Сохраняет (на (A, q), 1) ← Сохраняет (свободный (B), 1) ←
Сохраняет (на (B, C), 5) ← Сохраняет (свободный (P), 5) ←
Сохраняет (на (A, B), 6) ← Сохраняет (свободный (q), 6) ←

которые нужны для полного описания тех же состояний, являются примерами клауз (13) и (14), задающими предложения, добавляемые действием перен. Как и на рисунке 6.2 1 обозначает результат (перен (A, B, q), 0), 2 обозначает результат (перен (B, p, C), 1), 3 обозначает результат (перен (A, q, B), 5).

В общем случае поисковая стратегия может вызывать порождение различных утверждений о состояниях, которые нерелевантны решениям, вроде таких утверждений

Сохраняет (на (B, p), результат (перен (A, C, B), 0)) ←
Сохраняет (на (B, p), результат (перен (B, q, C), 0)) ←
Сохраняет (на (B, p), результат (перен (B, B, B), 0)) ←

Они описывают невозможные состояния. Порождение таких нежелательных утверждений устраняется нисходящим применением аксиомы остова. Они могут быть удалены также и за счет применения аксиомы остова снизу вверх, когда в нее добавляется внешнее условие

Возм (результат (перен (x, y, z), w))

Смешанное нисходящее и восходящее выполнение аксиомы остова (15)

Нисходящее применение аксиомы остова можно скомбинировать с восходящим применением аксиомы состояния пространства. Это можно показать с помощью стрелочного обозначения на рис. 6.3.

Это можно промоделировать и с помощью только одного нисходящего выполнения. Для этого достаточно переписать клаузы (1), (11) и (12), используя предикатный символ Невозм, являющийся отрицанием Возм, Клаузы (1), (11) и (12) превращаются в (1′), (11′) и (12′) соответст-

венно

(1′) Невозм (0) ←

(11′) Невозм (w) ← Сохраняет (на (A, B), w),
Сохраняет (на (B, C), w),
Сохраняет (на (C, r), w)

(12′) Невозм (w) ← Невозм (результат (перен (x, y, z), w)),
Перемещ (x),
Сохраняет (свободный (x), w),
Сохраняет (свободный (z), w),
Сохраняет (на (x, y), w),
Разл (x, z)

Переименование предикатных символов в той же манере, как это сделано для клауз (1), (11) и (12) было впервые введено Мельцером [Meltzer, 1966], а мы к этому вернемся снова в следующей главе.

Небольшая часть поискового пространства приведена на рис. 6.4. Смешанная нисходящая и восходящая стратегия выполнения эквивалентна чистому нисходящему выполнению с использованием (1′), (11′) и (12′) вместо (1), (11) и (12). Там показаны все дуги, отходящие от пути решения. Узлы, помеченные клаузами, содержащими неразрешимые подцели, заштрихованы для того, чтобы показать, что они являются оконечными вершинами, соответствующими неудачам. Номера в кружках, предшествующих подчеркнутым клаузам, показывают порядок, в котором выбираются они или их потомки. Непомеченные дуги обозначают выполнение процедурных вызовов, содержащих предикатный символ Разл. Некоторые узлы оставлены непомеченными для того, чтобы не приводить отвлекающие детали; t(x, y, z) обозначает перен(x, y, z).

Отметим, что многие альтернативы пути решения приводят к неудаче за несколько шагов. Альтернативы, которые не приводят к неудаче, соответствуют подлинным альтернативным действиям в поисковом пространстве состояний.

Окончательное приведение к неудаче альтернативных попыток достижения подцелей

Сохраняет (на (A, y), 5)
Сохраняет (на (B, y), 6)
Сохраняет (на (B, C), 6)

может быть ускорено усилением ограничивающих функций аксиомы остова.

Более ограничивающая версия аксиомы остова

Сохраняет (u, результат (перен (x, y, z), w)) ←
Сохраняет (u, w),
Разл (u, на (x, v)),
Разл (u, свободный (z)),
Разл (u, свободный (y))

в частности, немедленно приводит к неудаче, независимо от успеха одной из клауз (13) или (14).

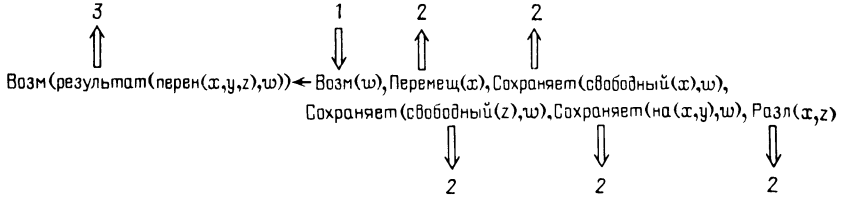
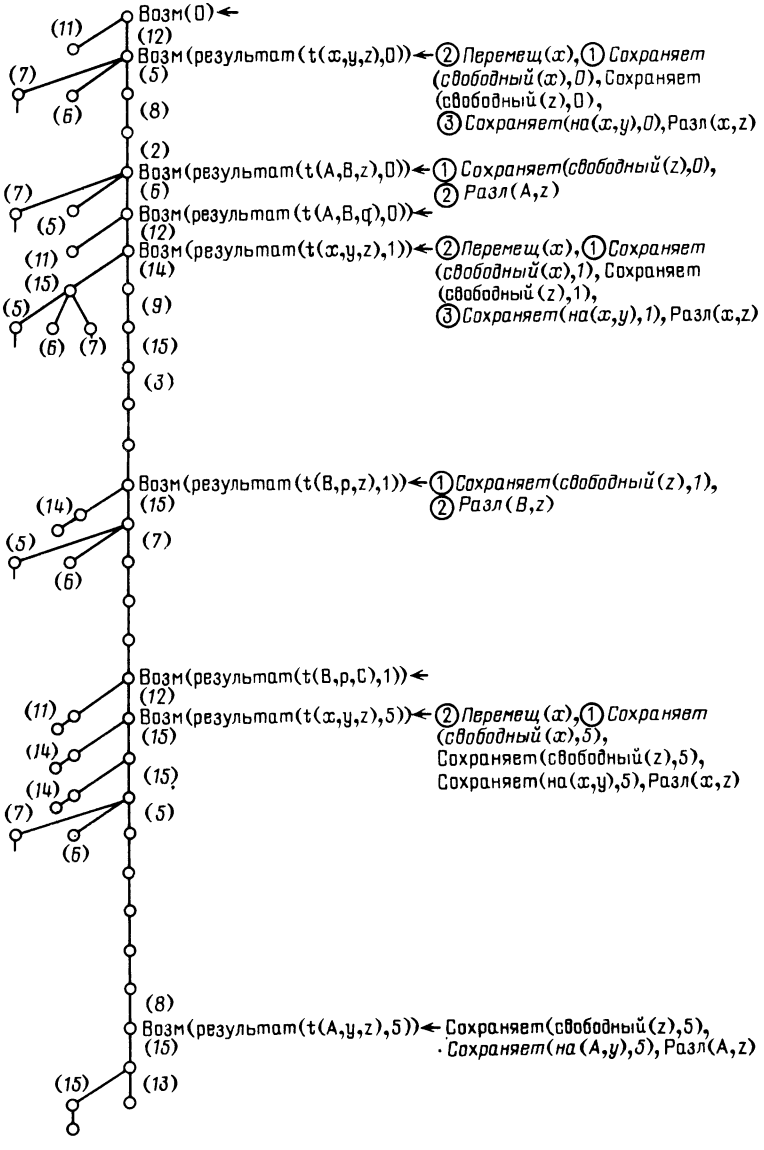

3
1
2
2
Возм(результат(перен(x,y,z),w))← Возм(w), Перемещ(x), Сохраняет(свободный(x),w),
Сохраняет(свободный(z),w), Сохраняет(на(x,y),w), Разл(x,z)
2
2
2

Рис. 6.3


Возм(0)←
Возм(результат(t(x,y,z),0))← ② Перемещ(x), ① Сохраняет (свободный(x),0), Сохраняет (свободный(z),0), ③ Сохраняет(на(x,y),0), Разл(x,z)
Возм(результат(t(A,B,z),0))← ① Сохраняет(свободный(z),0), ② Разл(A,z)
Возм(результат(t(A,B,q),0))←
Возм(результат(t(x,y,z),1))← ② Перемещ(x), ① Сохраняет (свободный(x),1), Сохраняет (свободный(z),1), ③ Сохраняет(на(x,y),1), Разл(x,z)
Возм(результат(t(B,p,z),1))← ① Сохраняет(свободный(z),1), ② Разл(B,z)
Возм(результат(t(B,p,C),1))←
Возм(результат(t(x,y,z),5))← ② Перемещ(x), ① Сохраняет (свободный(x),5), Сохраняет(свободный(z),5), Сохраняет(на(x,y),5), Разл(x,z)
Возм(результат(t(A,y,z),5))← Сохраняет(свободный(z),5), Сохраняет(на(A,y),5), Разл(A,z)

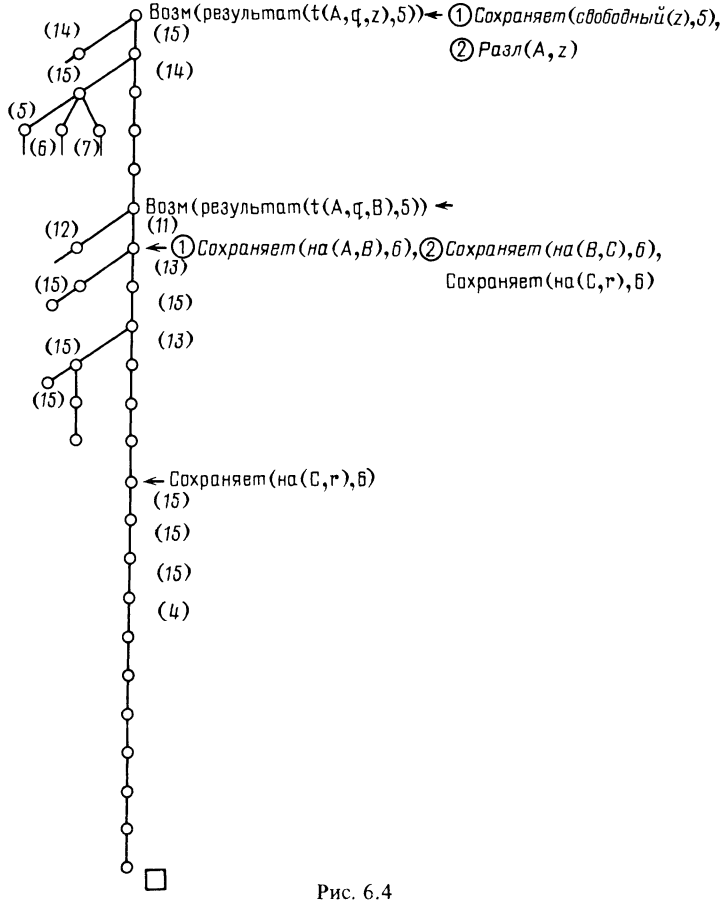


Рис. 6.4

Нисходящее выполнение в пространстве состояний и применение аксиомы остова

Часть поискового пространства состояний, определяемая нисходящим применением аксиомы пространства состояний, показана на рис. 6.5. Как и в случае, когда аксиома пространства состояний применяется снизу вверх, при повторном последовательном появлении одних и тех же объектов позникает избыточность. Переменные y и y′ обозначают те места, которые еще не были определены. В приведенном на рис. 6.6 решении все клаузы выполняются сверху вниз. Подцели рассматриваются вширь, слева направо в том порядке, как они записаны. Повторяющиеся подцели удалены. Для экономии места шаги, включающие достижения подцелей, содержащих предикатный символ Разл, не показаны.

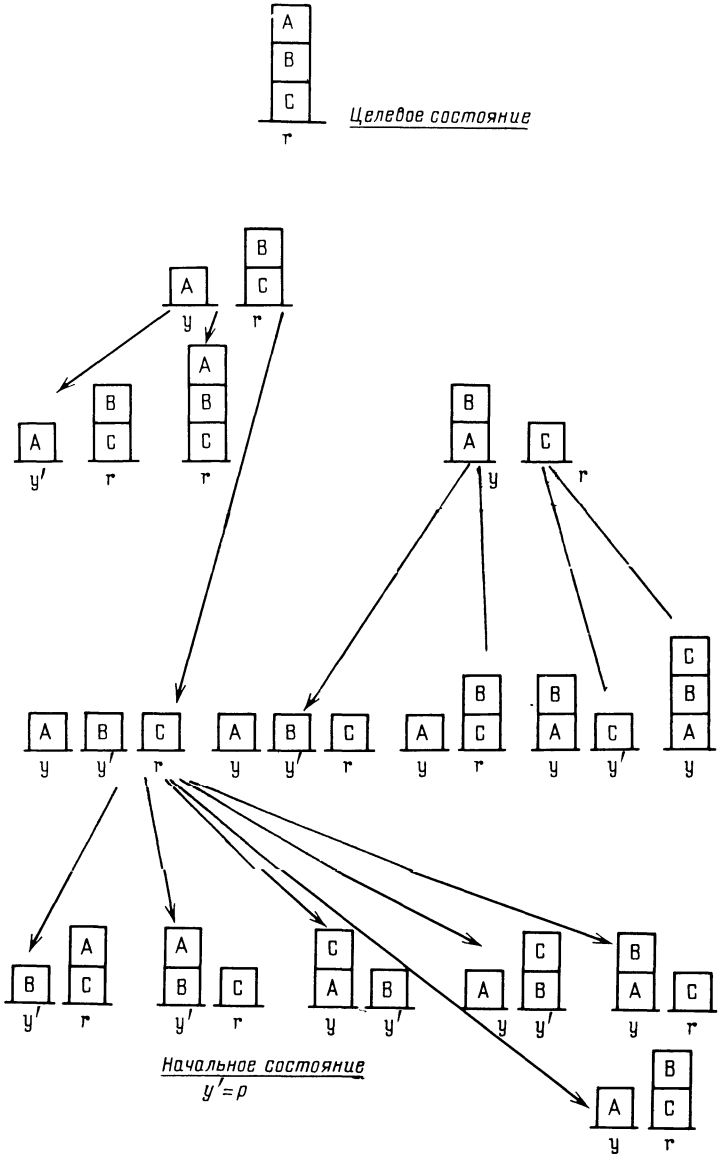

A
B
C
r
Целевое состояние
A
B
C
y
r
A
y′
B
C
r
A
B
C
r
B
A
y
C
r
A
B
C
y
y′
r
A
B
C
y
r
B
A
C
y
y′
C
B
A
y
B
C
y′
r
A
B
C
y
y′
r
C
A
B
y
y′
A
C
y
y′
B
A
C
y
r
B
C
y
r
Начальное состояние
y′ = p

Рис. 6. 5

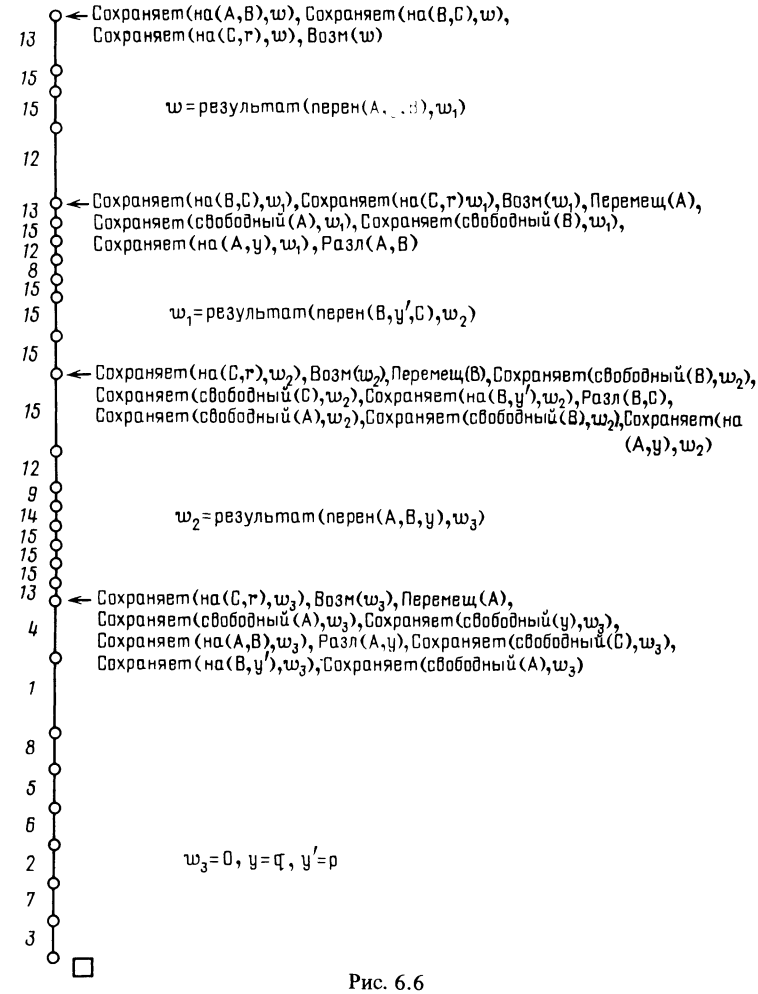


Рис. 6.6

Приложения методов построения планов

Главным приложением методов построения планов всегда было построение плана действий для роботизированных комплексов. Методы построения планов также применялись для автоматического порождения программ из спецификаций. Описание входного и выходного состояний задает спецификацию программы. Определение предусловий, а также добавляемых и удаляемых предложений выражает семантику машинных операций. План состоит из последовательности машинных операций и представляет собой программу. Более тщательно проработанные системы построения планов включают процедуры построения планов с условными, циклическими и другими операциями. Программы построения планов на языке клауз Хорна, написанные Уорреном [Warren 1974, 1976] и Моссом [Moss 1977], применялись для построения программ.

Применение методов построения планов для синтеза органических смесей развивалось Фогелем, в то время студентом высшей школы, в Имперском колледже летом 1977. Химические смеси, как и состояния при построении планов, можно описывать утверждениями об объектах (атомах и цепочках), принадлежащих этим соединениям. Предложения о том, что

цепочка b силы s сохраняется между
атомами a_1 и a_2 в смеси c

можно выразить единым n-арным отношением

Цепочка (b, s, a_1, a_2, c) ←

или несколькими бинарными отношениями

b имеет силу s ←
b сцепляет a_1 ←
b сцепляет a_2 ←
b принадлежит c ←

Начальное состояние смеси берется в качестве начального состояния, а целевое состояние смеси – в качестве целевого состояния. Химические реакции являются действиями, которые переводят одну смесь в другую. Они определяются посредством спецификации (1) предусловий, которые должны иметь место до того, как произойдет реакция, спецификации (2) новых цепочек, которые получаются в результате реакций и спецификации (3) старых цепочек, которые реакцией разрушаются. В аксиоме остова устанавливается, что цепочки, которые не разрушаются реакцией, предохраняются аксиомой. И программа, которую написал Мосс, и программа, которую написал Фогель, были разработаны именно как программы на языке клауз Хорна и выполнялись в прологоподобной системе, разработанной в Имперском колледже.

Программы анализа лекарственных средств были написаны на Прологе в Министерстве легкой промышленности в Будапеште [Futo, Darvas, Szeredi 1978]. В них реляционные структуры данных использовались так же как и в программах синтеза органических структур. Поскольку многие свойства данного лекарства могут быть неизвестны, в этих программах чаще использовались бинарные, а не n-арные отношения. Программы привели к полезным открытиям, касающимся доселе неизвестных действий лекарств и касающимся неясностей в описании лекарств в фармацевтической литературе.

Ограничения

В подходе, пропагандируемом в настоящей главе, информация о начальном состоянии хранится в явном виде, а аксиома остова используется для получения информации о более поздних состояниях. Можно привести доводы в пользу того, что это неестественно и потенциально неэффективно. Альтернатива, использующая поиск вглубь и рассуждения в прямом от исходного состояния направлении, состоит в том, чтобы хранить явно текущее состояние и получать информацию о более ранних состояниях. Эти два подхода интуитивно эквивалентны. Однако пробле-

ма формального выражения и подтверждения этой эквивалентности все еще в стадии решения.

Рассмотрение планов как последовательностей действий является другим ограничением, которое порождает избыточность, если действия не сообщаются между собой и могут выполняться параллельно. Избыточность появляется, так как выполнение действий в некоторой последовательности приводит к тем же результатам, что и в любой другой последовательности. Системы для порождения планов, являющиеся част чно упорядоченными наборами действий, описывались Сэйсердоти [Sacerdoti, 1975] и Тэйтом [Tate 1974]. Программа в терминах клауз Хорна, порождающая частично упорядоченные планы, была также написана на Прологе Уорреном. Обзор систем построения планов и сравнение с одной из систем, представленных в этой главе, был сделан Уолдингером [Waldinger 1977].

Упражнения

1. Сформулируйте n-арное представление задачи мира блоков. Для этого положите Состояние (x, y, z) истинным, когда оказывается возможным блоку A быть на x, блоку B – на y, а блоку C – на z одновременно. Сравните стратегию поиска решений задач для n-арного представления с такими стратегиями при бинарном представлении.

2. Переформулируйте задачу о сосудах с водой, рассмотренную в гл. 4, как задачу построения плана, используя бинарное представление, введенное в настоящей главе. Сравните стратегии поиска решений, нужные для эффективного решения задач в бинарном и n-арном случаях.

3. Оператор присваивания в обычных языках программирования можно рассматривать как действие, которое переводит компьютер из одного состояния в другое. Новое состояние

присваивание (u, v, w)

отличается от предыдущего состояния w тем, что ячейка u содержит v.

Предположим, что A, B и C – ячейки и что в начальном состоянии 0 они содержат, соответственно, a, b и c. Задача состоит в нахождении состояния, в котором начальные состояния A и B поменяются местами.

Сформулировать и решить эту задачу как задачу построения плана.

Сейчас мы раздвинем рамки модели поиска решений задач на языке клауз Хорна до языка нехорновских клауз. Если допустить нехорновские клаузы, то

(1) цели и утверждения могут быть как негативными, так и позитивными;

(2) применение процедур к целям может порождать как утверждения, так и подцели;

(3) достижение подцелей может потребовать анализа нескольких альтернативных случаев;

и

(4) решения могут быть дизъюнктными, т.е. $x = t_1$ или t_2 или . . . или t_m

Нисходящий и восходящий выводы можно тоже распространить на нехорновские клаузы. Новые правила, также как и старые, являются специальными случаями общего правила резолюций, впервые предложенного Робинсоном [Robinson 1965a].

Негативные цели и утверждения

Во многих случаях множество нехорновских клауз можно выразить в терминах клауз Хорна, переименовывая предикатные символы [Meltzer 1966]. К примеру, нехорновская клауза

Приятный (x), Кошмарный (x) ← Сон (x),

может быть выражена в виде клаузы Хорна так:

Кошмарный (x) ← Сон (x), Неприятный (x)

за счет выражения негативного атома не–Приятный (x) в виде позитивного атома Неприятный (x).

Точно также задачу доказательства того, что каждый гриб семейства Boletus ядовит, сформулированную в терминах нехорновских клауз, можно переформулировать в терминах клауз Хорна при помощи исключения предикатного символа Съедобный–гриб и использования вместо него нового предикатного символа Несъедобный–гриб. Неотрицательный атом Несъедобный–гриб (x) означает то же, что и отрицательный атом не–Съедобный–гриб (x). Новую задачу в терминах клауз Хорна Fung′1 . . 6 можно решать сверху вниз или снизу вверх.

Fung′1 Поганка (x) ← Гриб (x), Несъедобный–гриб (x)
Fung′2 Ядовитый (x) ← Поганка (x)
Fung′3 Гриб (x) ← Boletus (x)
Fung′4 Несъедобный–гриб (x) ← Boletus (x)
Fung′5 Boletus ( ) ←
Fung′6 ← Ядовитый ( )

Решение снизу вверх приведено на рис. 7.1, а решение сверху вниз приведено на рис. 7.2.

Восходящий вывод утверждения

Несъедобный–гриб ( ) ←

из клауз Хорна Fung′4 и Fung′5 эквивалентен выводу негативного "утверждения"

← Съедобный–гриб ( )

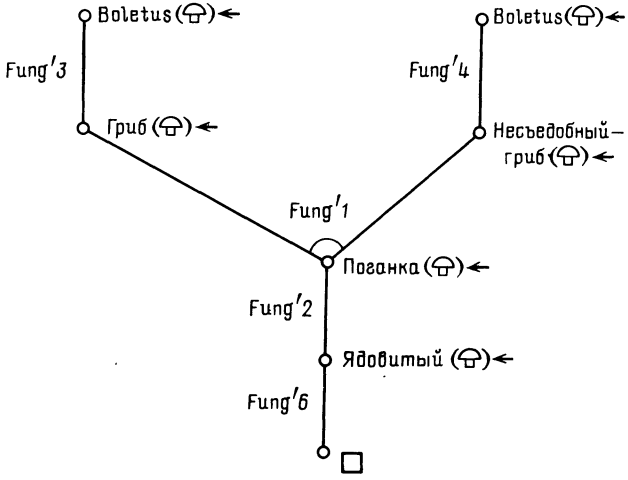


Рис. 7.1. Восходящее решение

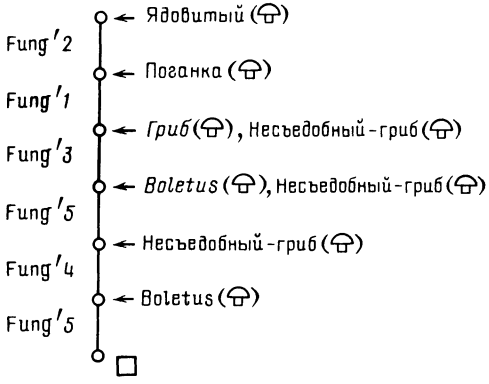


Рис. 7.2. Нисходящее решение

прямо из исходных клауз Fung 4 . . 5

← *Boletus* (*x*), Съедобный–гриб (x)
Boletus ( ) ←

Точно также нисходящий вывод позитивных подцелей

← Гриб (), Несъедобный–гриб ( )

из целевого предложения

← Поганка ()

при помощи клаузы Хорна Fung′1 эквивалентен прямому выводу клаузы

Съедобный–гриб (x) ← Гриб (x)

из того же самого целевого предложения

← *Поганка* ( )

при помощи нехорновской клаузы

Fung1 *Поганка* (*x*), Съедобный–гриб (x) ← Гриб (x)

Резолюция

И нисходящий вывод, и восходящий вывод, как для клауз Хорна, так и для нехорновских клауз, является специальным случаем общего правила резолюций:

Если даны две клаузы, то из них можно составить резольвенту; для этого сначала необходимо переименовать переменные так, чтобы различные клаузы содержали бы различные переменные. *Резольвента* двух клауз существует, если посылка одной из клауз и заключение другой клаузы могут быть согласованы. Про обе клаузы говорят, что они являются *родительскими* для клаузы–резольвенты. Некоторый атом является *посылкой* резольвенты, если он получен применением согласующей подстановки к какой-либо посылке одной из родительских клауз, отличающейся от той посылки, которая согласуется с заключением другой клаузы. Некоторый атом является *заключением* резольвенты, если он получен применением согласующей подстановки к какому-либо заключению одной из родительских клауз, отличающемуся от того заключения, которое согласуется с посылкой другой родительской клаузы.

Это определение может быть сформулировано и на языке клауз Хорна. Пусть

рез(x, u, y, v) является именем резольвенты, которая существует, если после подходящего переименования переменных посылка u в x согласуется с заключением v в y;

пос (x) является совокупностью посылок клаузы x;

закл (y) является совокупностью заключений клаузы y;

объед(x, y) является объединением клауз x и y;

Примен(x, w, x′) выражает то, что результатом применения подстановки w к x является x′;

Переимен (x, y, w) выражает то, что после применения подстановки w к клаузам x и y, эти клаузы больше не содержат общих переменных;

Согл (u, v, w) выражает то, что подстановка w согласует атомы u и v;
Элемент (u, x) выражает то, что u является элементом x;
Композ (w_1, w_2, w) выражает то, что действие подстановки w является композицией действий применяемой первой подстановки w_1 и применяемой второй подстановки w_2;
Резольв (x, u, y, v, w) выражает то, что резольвента клауз x и y по атомам u и v существует, а w является композицией подстановок, осуществляющих переименование переменных и согласование атомов.

Резольв (x, u, y, v, w) ← Переимен (x, y, w_1), Элемент (u, пос (x)),
Примен (u, w_1, u′), Элемент (v, закл (y)),
Примен (v, w_1, v′), Согл (u′, v′, w_2),
Композ (w_1, w_2, w)

Элемент (z, пос (рез (x, u, y, v))) ← Резольв (x, u, y, v, w),
Элемент (z′, объед (пос (x), пос (y))),
Разл (z′, u), Примен (z′, w, z)

Элемент (z, закл (рез (x, u, y, v))) ← Резольв (x, u, y, v, w),
Элемент (z′, объед (закл (x), закл (y))),
Разл (z′, v), Примен (z′, w, z)

Элемент (z, объед (x, y)) ← Элемент (z, x)
Элемент (z, объед (x, y)) ← Элемент (z, y)

Отметим, что это определение можно использовать и сверху вниз и снизу вверх. Реализацию с разделением структур, выполненную Бойером и Муром [Boyer, Moor 1972] можно считать нисходящим применением нашего определения, дополненного запоминанием в качестве лемм достигаемых подцелей вида Резольв (x, u, y, v, w).

Определение, приведенное здесь, является менее общим, чем определение Робинсона, включающее правило факторизации; правило факторизации описывается позднее в этой главе.

Расходящиеся от центра рассуждения, использующие клаузы Хорна

Наряду с нисходящим и восходящим выводами в резолюции применяются также расходящиеся от центра рассуждения, использующие клаузы Хорна. Например, резольвентой двух клауз

Ошибается (*x*) ← Человек (x)
Смертен (x) ← *Ошибается* (*x*)

является клауза Смертен (x) ← Человек (x)

Расходящиеся от центра рассуждения можно применять также и к разным копиям одной и той же клаузы. Например, из двух копий определения предка

Предок (x, y) ← *Предок* (*x*, *z*), Предок (z, y)
Предок (*u*, *v*) ← Предок (u, w), Предок (w, v)

мы можем вывести резольвенту

Предок (x, y) ← Предок (x, w), Предок (w, z), Предок (z, y)

Пример "Пропозициональная логика"

Клаузы, определяющие семантику пропозициональной логики, доставляют поучительные примеры применения правила резолюции. Договоримся через x и y обозначать высказывания x* и y* соответственно; тогда

x & y будет обозначать высказывание "x* и y*";
x ∨ y будет обозначать высказывание "x* или y*";
x ⊃ y будет обозначать высказывание "если x* и y*";
x ↔ y будет обозначать высказывание "x* если и только если y*";
¬x будет обозначать высказывание "неверно, что x*",

где &, ∨, ⊃, ↔ и ¬ суть инфиксные функциональные символы. Пусть Истинно (x) означает, что x истинно. Теперь приведем множество клауз. Оно не может быть записано только на языке клауз Хорна даже за счет переименования предикатных символов

T1 Истинно (x & y) ← Истинно (x), Истинно (y)
T2 Истинно (x) ← Истинно (x & y)
T3 Истинно (y) ← Истинно (x & y)
T4 Истинно (x ∨ y) ← Истинно (x)
T5 Истинно (x ∨ y) ← Истинно (y)
T6 Истинно (x), Истинно (y) ← Истинно (x ∨ y)
T7 Истинно (x ⊃ y), Истинно (x) ←
T8 Истинно (x ⊃ y) ← Истинно (y)
T9 Истинно (y) ← Истинно (x), Истинно (x ⊃ y)
T10 Истинно (x ↔ y) ← Истинно (x ⊃ y), Истинно (y ⊃ x)
T11 Истинно (x ⊃ y) ← Истинно (x ↔ y)
T12 Истинно (y ⊃ x) ← Истинно (x ↔ y)
T13 Истинно (¬x), Истинно (x) ←
T14 ← Истинно (¬x), Истинно (x)

Клаузы T1 . . 3 утверждают, что

x & y истинно, если и только если
x истинно и y истинно

Клауза T1 составляет если–часть этого логического предложения, а клаузы T2 . . 3 составляют только–если часть. Аналогично, остальные клаузы означают:

T4 . . 6	x ∨ y	истинно, если и только если x истинно или y истинно
T7 . . 9	x ⊃ y	истинно, если и только если, если x истинно, то y истинно
T10 . . 12	x ↔ y	истинно, если и только если x ⊃ y истинно и y ⊃ x истинно
T13 . . 14	¬x	истинно, если и только если x не истинно.

Базой для приведенного множества клауз служит более общее понятие "быть истинным", сформулированное для предложений в стандартной форме логики Колмероэ (неопубликовано).

Если-части этих предложений могут применяться сверху вниз, когда задачи установления истинности сложных высказываний сводятся к подзадачам установления истинности более простых высказываний. Только-если части могут применяться при восходящем выводе заключений, связанных с истинностью простых высказываний, из посылок, связанных с истинностью более сложных высказываний.

Например, чтобы показать, что

p & ¬q истинно, если p истинно, а q не истинно

естественнее всего рассуждать сверху вниз, начиная с цели

← Истинно (p & ¬q),

используя посылки

A1 Истинно (p) ←
A2 ← Истинно (q)

и рассматривая вторую посылку A2 как негативную (рис. 7.3).

Можно считать, что здесь клауза T13 сводит задачу доказательства истинности ¬q к задаче доказательства неистинности q, которая непосредственно решается благодаря посылке A2*).

С другой стороны, для доказательства того, что

p истинно, а q не истинно, если p & ¬q истинно

более естественными представляются рассуждения снизу вверх (рис. 7.4), начиная с посылки

Истинно (p & ¬q) ←

Клаузу

G Истинно (q) ← Истинно (p)

можно интерпретировать как цель доказательства того, что p истинно, а q не истинно. Можно считать, что клауза T14 выводит как раз то негативное утверждение о неистинности q, которое обеспечивает достижение негативной цели в G. Отметим, что обозначенный меткой G пучок дуг на рис. 7.4 представляет собой два последовательных шага метода резолюций. Порядок выполнения этих шагов несуществен.

Задача доказательства того, что

p V ¬q истинно,

демонстрирует другую характерную особенность нисходящего поиска решений с помощью нехорновских клауз: "Ни один метод адекватно не решает такую задачу, но несколько альтернативных методов исчерпывают все случаи до конца" (рис. 7.5).

Клаузы T4 и T5 на рис. 7.5 сводят исходную задачу к подзадачам, которые и исчерпывают оба случая, представляемых нехорновской клаузой T13.

Восходящее решение этой же задачи приведет к рассуждению на основе частных случаев. Анализ подобных частных случаев при рассуждении снизу

*) В этой главе мы представим только опровержение методом резолюций. Пространства поиска будут показаны в следующей главе.

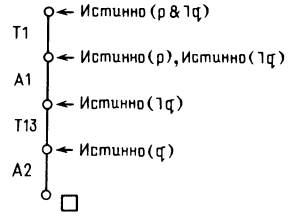


Рис. 7.3

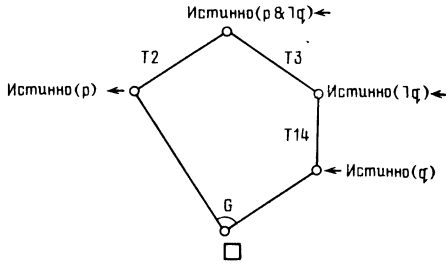


Рис. 7.4

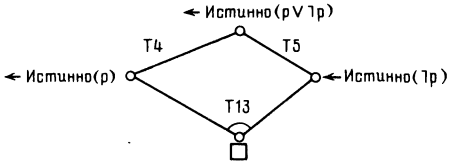


Рис. 7.5

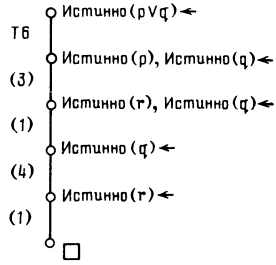


Рис. 7. 6

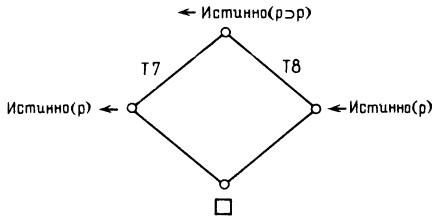


Рис. 7.7

вверх можно сделать более прозрачным в контексте решения другой задачи, в которой надо показать, что

r истинно, если p ∨ q истинно

в предположении, что

r истинно, если p истинно и r истинно, если q истинно:

(1) Истинно (r)
(2) Истинно (p ∨ q) ←
(3) Истинно (r) ← Истинно (p)
(4) Истинно (r) ← Истинно (q)

Посредством клаузы T6 (рис. 7.6) выводится нехорновская клауза, которая влечет за собой два случая. Решение проходит снизу вверх, причем

сначала достигается цель в случае, когда p истинно, а затем она достигается в случае, когда q истинно. Второй случай запоминается на все время работы над первым случаем.

Если даны цель и клауза Хорна, сводящая эту цель к подцелям, то можно использовать нехорновские клаузы для вывода посылок, способствующих достижению подцелей. Такие нехорновские клаузы в общем случае возникают из неклаузального высказывания

$$A \leftarrow [B \leftarrow C], D$$

в котором посылка является импликацией. Используя терминологию методов поиска решений, можно сказать, что клаузальная форма высказывания

$$A, C \leftarrow D$$
$$A \leftarrow B, D$$

описывает такую формулировку: для того, чтобы решить A, следует решить D и решить B, допуская C.

Клаузы T7 . . . 8 возникают из таких неклаузальных высказываний

$$\text{Истинно}\ (x \supset y) \leftarrow [\text{Истинно}\ (y) \leftarrow \text{Истинно}\ (x)]$$

Для того, чтобы показать, что $x \supset y$ истинно, надо показать, что y истинно, допуская, что x истинно.

В некоторых случаях для решения задачи нужна только одна из клауз T7 . . . 8. Если x не истинно, как в случае

$$\leftarrow \text{Истинно}\ ((p \,\&\, \neg p) \supset q)$$

то нужна только нехорновская клауза T7, из которой выводится утверждение

$$\text{Истинно}\ (p \,\&\, \neg p) \leftarrow$$

Но если y истинно, как в случае

$$\leftarrow \text{Истинно}\ (q \supset (p \vee \neg p))$$

то нужна только клауза Хорна T8, из которой выводится подцепь

$$\leftarrow \text{Истинно}\ (p \vee \neg p)$$

В большинстве случаев, тем не менее, нужно использовать обе клаузы. Простейшей задачей, для которой необходимо совместное действие клауз T7 . . . 8, является доказательство истинности $p \supset p$ (рис. 7.7).

Выведенная подцель доказательства истинности p достигается при помощи выведенного утверждения об истинности p. Пучок дуг, связанный с шагом метода резолюций, не помечен, поскольку в этом выводе участвуют только выведенные клаузы.

Задача доказательства того, что

$$p \supset q \text{ истинно, если } p \supset r \text{ истинно и } r \supset q \text{ истинно}$$

представляет большой интерес. Здесь более естественным выглядит двунаправленное рассуждение в прямом направлении от посылок и в обратном направлении от заключения. Более того, когда рассуждение ведется в

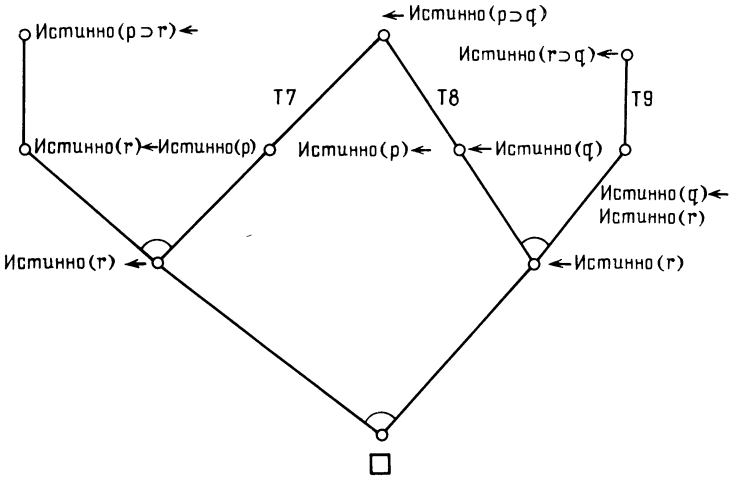


Рис. 7.8

обратном направлении от заключения

$$\leftarrow \text{Истинно } (p \supset q)$$

то естественно рассуждать в прямом направлении от выведенного утверждения

$$\text{Истинно } (p) \leftarrow$$

и в обратном направлении от выведенной подцели

$$\leftarrow \text{Истинно } (q)$$

Доказательство методом резолюций, приведенное на рис. 7.8, формализует этот довод.

Стрелочная нотация для нехорновских клауз

Стрелочная нотация, использовавшаяся раньше для клауз Хорна при обозначении комбинации нисходящего и восходящего выводов, может

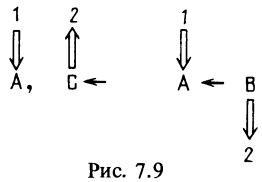


Рис. 7.9

быть использована и в случае нехорновских клауз. В частности, переложение высказывания

$$A \leftarrow [B \leftarrow C]$$

на язык методов поиска решений можно изобразить на рис. 7.9 стрелками, связанными с соответствующими клаузами.

Стрелки, связанные с первой клаузой, говорят о том, что необходимо ждать достижения подцели типа A, а затем выводить утверждение C ←. Стрелки, связанные со второй клаузой, говорят о том, что необходимо ждать достижения подцели типа A, а затем выводить подцель B.

Использование стрелочной нотации для управления поведением системы поиска решений будет подробно рассмотрено в следующей главе.

Дизъюнктные решения задач, сформулированных в терминах нехорновских клауз

В задачах построения планов, описанных на языке нехорновских клауз, может потребоваться построение условных планов из дизъюнктных решений.

Рассмотрим, к примеру, задачу занесения максимума из двух чисел A и B в область памяти L:

M1 ← Выполнено (знач(L, x), w), Max(A, B, x)
M2 Числ(A) ←
M3 Числ(B) ←
M4 Область–памяти (L) ←
M5 $u \leqslant v, v \leqslant u \leftarrow$ Числ(u), Числ(v)
M6 Max(u, v, u) $\leftarrow v \leqslant u$
M7 Max(u, v, v) $\leftarrow u \leqslant v$

Предположим теперь, что единственно допустимым действием является операция *присваивания*. Если дано состояние w, то эта операция порождает новое состояние

присвоить(u, v, w),

получающееся из w благодаря занесению v в область памяти u.

"Семантика" этого действия может быть прояснена за счет указания его предусловий и спецификации добавляемых и удаляемых предложений при выполнении этого действия. Для простоты в клаузы, описывающие добавляемые (M8) и удаляемые (M9) предложения, можно включить единственное предусловие, выражающее то, что u есть область памяти:

M8 Выполнено (знач(u, v), присвоить(u, v, w)) ←
Область–памяти(u)
M9 Выполнено (x, присвоить(u, v, w)) ←
Выполнено(x, w),
Разл (x, знач(u, v)), Область–памяти(u)

До того, как начать решение задачи сверху вниз, будет целесообразно привести один шаг восходящей процедуры (рис. 7.10). Нисходящее решение, использующее выведенную лемму M10, требует согласованного применения двух процедур M6 и M7 для достижения единственной подцели Max(A, B, x) (рис. 7.11).

Решением здесь является дизъюнкция двух возможностей

w = присвоить(L, A, w′) или присвоить(L, B, w′)
для любого w′

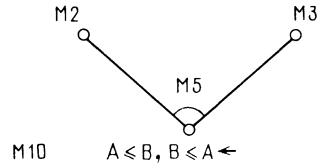


Рис. 7.10

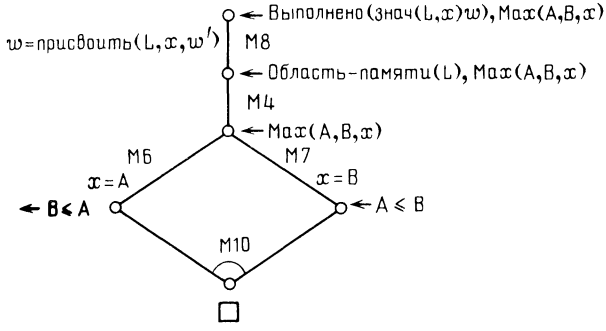


Рис. 7.11

Решение существует, но оно имеет неопределенность третьего рода из-за того, что имеются две указанные возможности.

Недетерминизм третьего рода резко отличается от недетерминизма первого рода. Задача обладает недетерминизмом третьего рода, если ее решение

$$x = t_1 \text{ или } t_2 \text{ или } \ldots \text{ или } t_m$$

недоспецифицировано. Та же задача обладает недетерминизмом первого рода, если ее решение переспецифицировано:

$$x = t_1 \text{ и } t_2 \text{ и } \ldots \text{ и } t_m$$

Изучение программных конструкций как приложений методов построения планов было впервые проведено Грином [Green 1969b], а также Ли и Уолдингером [Lee, Waldinger 1969]. Ли и Уолдингер в частности, получили алгоритм извлечения условных программ вида

Если A ⩽ B то w = присвоить (C, B, w′)
иначе w = присвоить (C, A, w′)

из дизъюнктных решений. Связь между формированием планов и аксиоматической семантикой языков программирования была установлена Моссом [Moss 1977].

Факторизация

Правило резолюций само по себе обладает достаточной полнотой для доказательства несовместности системы клауз Хорна. Более того, оно также подходит для решения многих, правда не всех, задач, выраженных

в терминах нехорновских клауз. Рассмотрим комбинацию факторизации и резолюции, впервые описанную в неопубликованной оригинальной статье Робинсона. Эта комбинация эквивалентна опубликованной версии правила резолюций [Robinson 1965a]. Благодаря ей доказательство полноты в опубликованной статье обеспечивает полноту комбинации резолюции и факторизации.

В качестве простого примера, в котором приходится использовать факторизацию, рассмотрим парадокс брадобрея:

> Допустим, что брадобреи бреют всех людей, которые не бреются сами и не бреют тех, кто бреется сам. Тогда брадобреи не существуют.

Для того, чтобы получить заключение, зададимся утверждением о существовании брадобрея и попытаемся вывести противоречие.

B1 Бреет(x, y), Бреет(y, y) ← Брадобрей(x)
B2 ← Бреет(x, y), Бреет(y, y), Брадобрей(x)
B3 Брадобрей(☺) ←

Продемонстрируем несовместность этих трех клауз, означивая первые две из них:

Бреет(☺,☺), Бреет(☺,☺) ← Брадобрей(☺)
← Бреет(☺,☺), Бреет(☺,☺), Брадобрей(☺)

удаляя повторяющиеся атомы

Бреет(☺, ☺) ← Брадобрей(☺)
← Бреет(☺, ☺), Брадобрей(☺)

и применяя резолюцию (рис. 7.12).

То, что использование одной только резолюции может оказаться недостаточным для демонстрации несовместности, можно показать рассматривая укороченный вариант только что приведенного примера

S1 Б(x), Б(y) ←
S2 ← Б(u), Б(v)

Эти две клаузы несовместны, поскольку у них есть такие параметры

Б(x), Б(x) ←
← Б(u), Б(u)

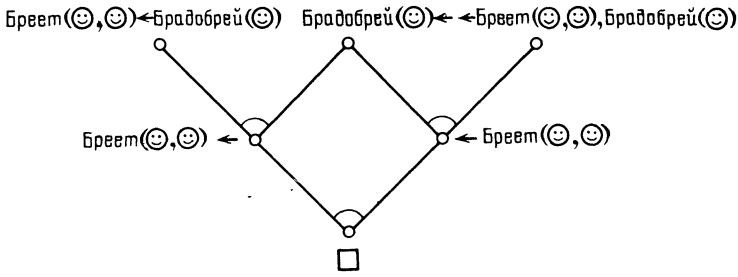


Рис. 7.12

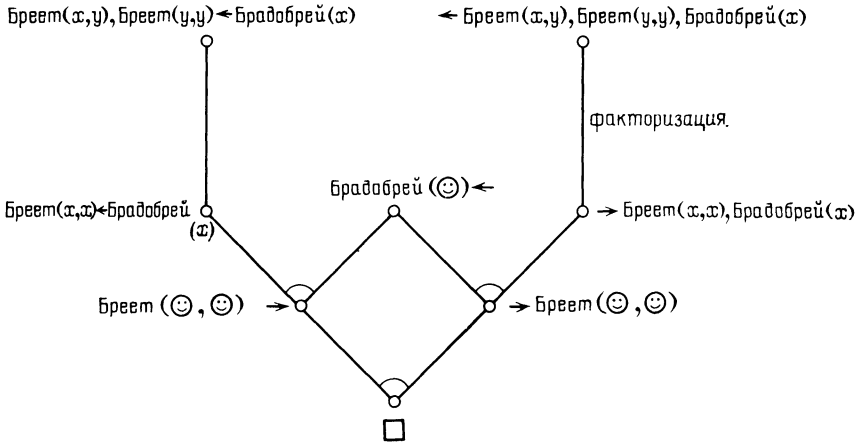


Рис. 7.13

которые после устранения повторяющихся атомов прямо противоречат друг другу:

Б(x) ←
← Б(y)

Однако вне зависимости от того, сколько раз применялась бы резолюция и к самим клаузам S1 . . 2, и к выведенным из них, каждая резольвента все равно будет содержать в точности два атома, и, следовательно, ни одна резольвента не станет пустой клаузой (т.е. не содержащей ни одного атома).

Правило факторизации, которым необходимо дополнить резолюцию в этих и подобных им случаях, порождает примеры клауз специально с целью устранения повторяющихся атомов. Означивающая же подстановка выполняется, чтобы обеспечить такое согласование двух атомов, при котором они станут копиями друг друга. Так, применение факторизации к двух клаузам B1 и B2 порождает более общие примеры, нежели те, что были приведены выше.

B1 *Брает*(*x*, *y*), *Брает*(*y*, *y*) ← Брадобрей(x)
(согласование подчеркнутых атомов)
Брает(x, x), Брает(x, x) ← Брадобрей(x)
(удаление повторяющихся)
B′1 Брает(x, x) ← Брадобрей(x)

Клауза B′1 является единственным фактором B1. По аналогичным соображениям B′2 является единственным фактором B2:

B′2 ← Брает(x, x), Брадобрей(x)

Применение факторизации и комбинацию резолюции с факторизацией можно продемонстрировать на графе (рис. 7.13).

На самом же деле острая необходимость в факторизации возникает нечасто; постоянное ее применение приводит к росту избыточности. Пожалуй, наиболее жесткое ограничение на использование факторизации, не влияющее на полноту, было предоставлено в процедуре доказательства методом исключения [Loveland 1968, 1969, 1978].

Упражнения

1. Используйте резолюцию и факторизацию для доказательства несовместности следующих предположений.

Джону нравится любой, кто не нравится самому себе.
Джону не нравится ни один, кто нравится сам себе.

2. Предположим, что
а) существует дракон;
б) дракон или спит в своей пещере или охотится в лесу;
в) если дракон голоден, то он не может спать;
г) если дракон устал, то он не может охотиться.
Применяя резолюцию, ответьте на такие вопросы:

Что делает дракон, когда он голоден?
Что делает дракон, когда он устал?
Что делает дракон, когда он голоден и устал?

Для того, чтобы ответить на эти вопросы, надо уяснить себе такое предложение:

Если x не может делать y, то x не делает y.

3. Выразите следующие предположения на языке клауз

Каждый восхищается героем.
Неудачник восхищается каждым.
Каждый, кто не является героем, неудачник.

Используйте резолюцию и факторизацию для нахождения пары индивидуумов (не обязательно различных), которые восхищаются друг другом.

4. Эта проблема обсуждалась Муром [Moore 1975]. Предположим, что есть три блока A, B и C (рис. 7.14): A находится на B, который находится на C; A — зеленый, C — голубой, а цвет B неизвестен. Используйте

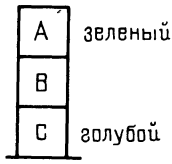


Рис. 7.14

резолюцию (и, если надо, факторизацию) для нахождения зеленого блока, находящегося на незеленом блоке. При этом надо предположить, что голубой не есть зеленый. Какие блоки найдет процедура доказательства?

5. Используя резолюцию и факторизацию, покажите, что приведенные ниже заключения следуют из предположений T1 . . 14

а) Если $p \supset (r \& q)$ истинно, то $(p \supset r) \& (p \supset q)$ истинно.

б) Если $p \supset q$ истинно, то существует такое r, что $(p \supset r) \& (r \supset q)$. Какое r будет найдено процедурой доказательства?

6. Отношение Плюс (x, y, z), выполняющееся, когда x + y = z, можно задать применением таких нехорновских клауз:

Плюс (x, y, z), Получить-сложением (0, y) ←

Плюс (x, y, z) ← Получить-сложением (x, z)

Получить-сложением (s(x), s(z))) ← Получить-сложением (x, z)

где s(x) обозначает следующее за x число. Используя резолюцию и факторизацию, решите задачу

← Плюс (x, y, s(y)), Плюс (x, x, y)

8. ПРОЦЕДУРА ДОКАЗАТЕЛЬСТВА МЕТОДОМ ГРАФА СОЕДИНЕНИЙ

Пространство поиска, образующееся на основе неограниченного применения резолюции, несет в себе большую избыточность. Ценой потери гибкости от этой избыточности можно избавиться, ограничивая резолюцию нисходящим или восходящим выводом. Но от нее можно избавиться также и без потери гибкости благодаря применению процедуры доказательства с помощью графа соединений.

В такой граф прежде всего включаются клаузы, а после этого вхождения согласующихся атомов, появляющиеся на противоположных концах стрелки, соединяются между собой дугами. С каждой дугой в графе ассоциируется резольвента, полученная из клауз, соединенных этой дугой. Основной операцией процедуры доказательства с помощью графа соединений является выбор произвольной дуги и включение ассоциированной с ней резольвенты в граф соединений. Нисходящий вывод получается благодаря выбору дуг, соединенных с целевым предложением; восходящий же вывод – благодаря выбору дуг, соединенных с клаузами, не имеющими посылок. Избыточность устраняется за счет удаления выбранной дуги и ограничения количества новых дуг, добавляемых в момент включения резольвенты в граф.

Начальный граф соединений

Первым шагом процедуры доказательства с помощью графа соединений является построение начального графа соединений. Кроме исходного множества клауз, *начальный граф соединений* содержит по одной дуге для каждой пары согласующихся атомов, находящихся на противоположных сторонах стрелок в разных клаузах. Такая дуга соединяет атомы и помечается согласующей подстановкой. Дальше в этой главе мы рассмотрим случай, когда дуга соединяет атомы в одной и той же клаузе.

На рис. 8.1 приведен начальный граф соединений для простой задачи, сформулированной на языке нехорновских клауз.

С каждой дугой этого графа ассоциируется резольвента, полученная путем согласования атомов, связанных с этой дугой. И обратно, с каждой резольвентой, которая может быть получена из различных исходных клауз, можно ассоциировать клаузу в этом графе.

Следуя принципу чистоты Робинсона [Robinson 1965a], клаузу, содержащую несвязанный атом, можно удалить из всего множества клауз

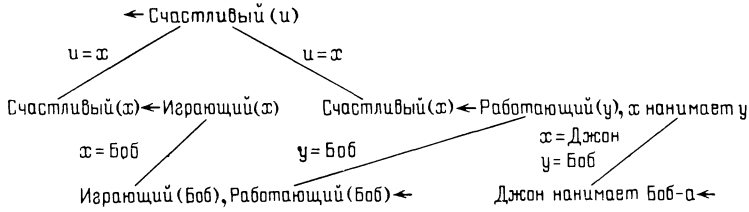


Рис. 8.1

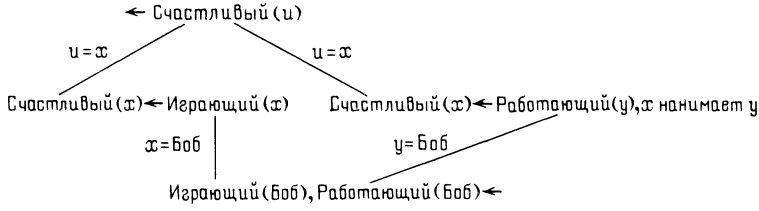


Рис. 8.2

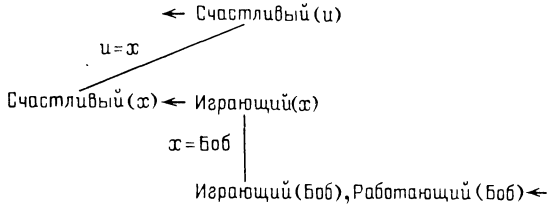


Рис. 8.3

без потерь для их совместности (или несовместности). Такая клауза не привносит ничего нового в опровержение методом резолюций, поскольку несвязанный атом не включается ни в какую резольвенту.

Удаление клауз, содержащих несвязанные атомы есть необходимое важное свойство процедуры доказательства с помощью графа соединений. Вдобавок к удалению самой клаузы, из графа должны быть также удалены все связи ее атомов. Удаление таких связей может привести к тому, что атомы в других клаузах станут несвязанными. Поэтому удаление клаузы может вызвать цепную реакцию удаления целой последовательности клауз из этого графа. Удаление клауз упрощает граф соединений, сокращает пространство поиска и облегчает нахождение решения.

Эффект удаления клауз можно проиллюстрировать на рис. 8.2, предположив, что Боб безработный и соответственно изменив предыдущий пример. Удалим теперь клаузу, содержащую несвязанный атом (рис. 8.3).

Этот новый граф содержит новый несвязанный атом. Удаление клауз продолжается до тех пор, пока у нас не останется пустое их множество. А пустое множество клауз тривиальным образом совместно, поскольку оно не содержит ни одной клаузы, которая могла бы стать ложной в некоторой интерпретации. Поэтому и исходное множество клауз является совместным.

Резолюция связей в графе соединений

Базовой операцией процедуры доказательства является выбор связи и порождение ассоциированной с ней резольвенты. Связь удаляется, а резольвента добавляется к графу. Также добавляются новые связи, соединяющие атомы резольвенты с атомами в оставшейся части графа. Эти новые связи можно получить без организации просмотра графа, исходя из связей, которыми уже соединялись атомы в клаузах, породивших резольвенту.

Например, для начального графа соединений, приведенного на рис. 8.1, можно строить рассуждение снизу вверх посредством выбора связи, согласующей два атома, содержащие предикатный символ "Играющий". В резольвенте атом Счастливый(Боб) получается из атома Счастливый(x) в клаузе, породившей резольвенту. Новые связи, соединяющиеся с новым атомом, получаются из связей, соединявшихся с породившим его атомом. Новая связь, соединяющая атомы Счастливый(Боб) и Счастливый(u), выведена из старой связи, соединявшей Счастливый(x) и Счастливый(u). Новый граф соединений, полученный после выбора связи, после порождения резольвенты, после добавления новых связей и после удаления обеих клауз, породивших резольвенту (они теперь содержат несвязанные атомы), показан на рис. 8.4.

Подстановка u = Боб, помечающая новую связь, может быть вычислена из подстановки x = Боб, помечающей выбранную связь, и из подстановки

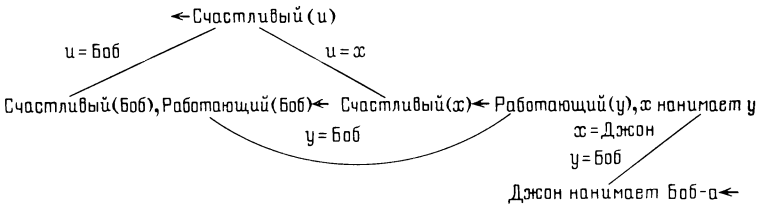


Рис. 8.4

u = x, помечающей "родительскую" связь, из которой получилась новая подцель.

Перед тем как продолжить рассмотрение примера, сформулируем общие принципы рассматриваемой процедуры доказательства.

Процедура доказательства методом графа соединений начинается с обработки начального графа соединений, а затем многократно повторяет обработку до тех пор, пока не будет порождена пустая клауза. *Она обрабатывает граф соединений*

(1) многократно удаляя клаузы, содержащие несвязанные атомы, и удаляя ассоциированные с ними связи до тех пор, пока все такие клаузы не будут удалены, а затем

(2) выбирая связь, удаляя ее и добавляя резольвенту и ассоциированные с нею новые связи к графу.

Это определение самых существенных моментов в процедуре доказательства методом графа соединений было дано в стиле итераций "repeat–until"*), применяемых для описания алгоритмов на алголоподобных

*) Повторяй пока. – *Примеч. пер.*

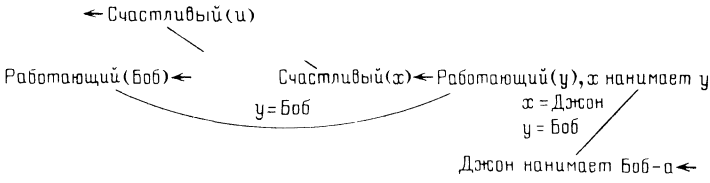


Рис. 8.5

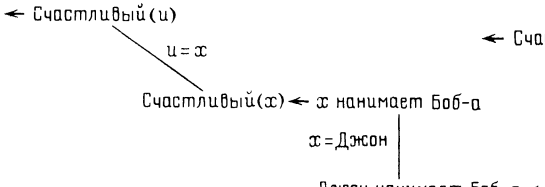


Рис. 8.6

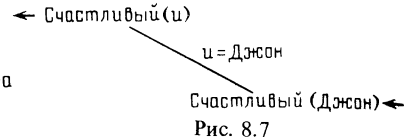
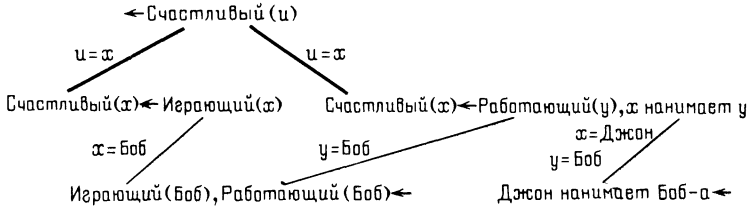


Рис. 8.7

Рис. 8.8



алгоритмических языках. В конце главы мы переформулируем это определение в стиле программирования на языке клауз Хорна.

Но вернемся к нашему примеру. В графе, вообще говоря, можно выбрать любую связь. Но мы продолжим наши действия путем восходящего анализа варианта Играющий(Боб), выбирая связь, помеченную u = Боб. Удаление выбранной связи приводит к появлению в одной из родительских клауз несвязанного атома. Эта родительская клауза удаляется (рис. 8.5).

Цель была достигнута в первом же варианте – Играющий (Боб). После этого, также рассуждая снизу вверх, приступим к рассмотрению оставшегося варианта – Работающий (Боб). Если удалить эту выбранную связь, то обе родительские клаузы будут содержать несвязанные атомы и поэтому тоже будут удалены (рис. 8.6).

Продолжая рассуждать снизу вверх, удалим обе родительские клаузы, потому что они содержат несвязанные атомы (рис. 8.7).

Резольвента, ассоциированная с оставшейся связью, оказывается пустой клаузой, и обе родительские клаузы удаляются (рис. 8.8).

Отметьте что наше доказательство на вопрос:

"Счастлив ли кто-нибудь",

дает дизъюнктный ответ:

"Да, Боб или Джон".

Последовательность графов соединений, порождаемых процедурой доказательства, представляет и доказательство несовместности, и поиск этого доказательства. В нашем примере каждый шаг поиска годится и для самого доказательства. В общем случае, тем не менее, как следует из теоремы Эренфойхта и Рабина [Bundy 1971], [Meltzer 1972] не всегда можно устранить шаги, нерелевантные доказательству.

На каждом этапе пути поиска доказательства каждая связь может быть выбрана для порождения резольвенты. Выбор различных связей приводит к порождению различных поисковых пространств, в одной части которых поиск оказывается легче, чем в другой. Покажем на рис. 8.9 с помощью последовательности графов соединений нисходящий поиск решения вышеприведенной задачи. Выбранные связи выделены жирными линиями. Для того, чтобы уменьшить количество изображаемых на рисунке графов, можно, когда порядок выбора несуществен, выбирать в графе за один раз несколько связей. Удаление несодержащих несвязанные атомы клауз явно на рисунке не представлено.

Как и при восходящем поиске решения, каждый шаг пригоден для доказательства.

Отметьте, что неограниченное применение правила резолюций оказывается избыточным в том смысле, что оно определяет пространство поиска, содержащее много ненужных клауз, включающих, в частности, все те клаузы, которые принадлежат и пространству нисходящего поиска, и пространству восходящего поиска, только что рассмотренным в настоящей главе.

Смешанный нисходяще-восходящий поиск в задаче грамматического разбора

Нисходящий и восходящий выводы могут быть смешаны очень просто за счет смешанного выбора связей, соединяющих атомы в клаузах, не содержащих посылок. В общем случае всегда полезно выбирать связь, которая приводит к наименее сложным новым графам. Применяя эту стратегию к версии задачи грамматического разбора из гл. 5, получим

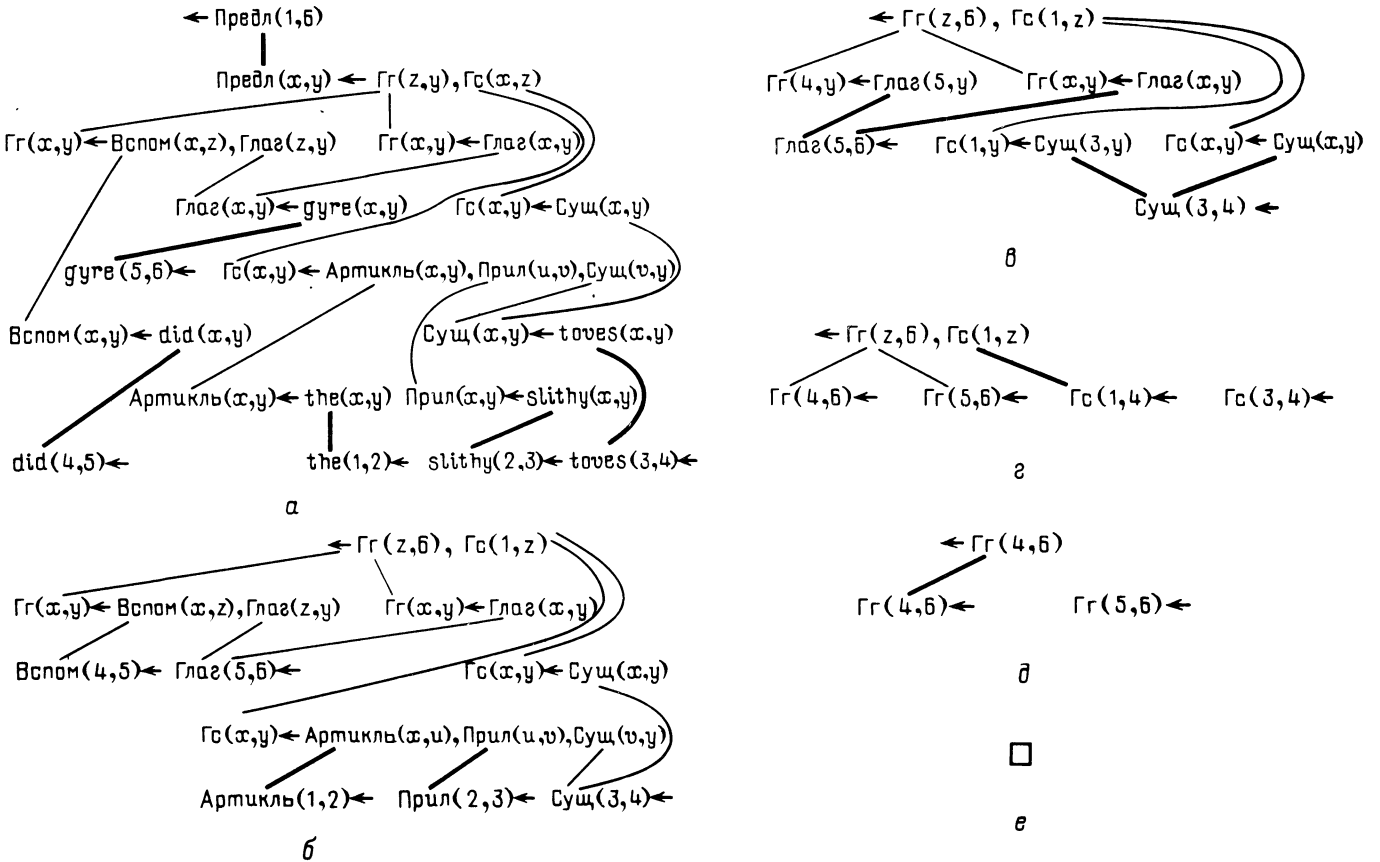
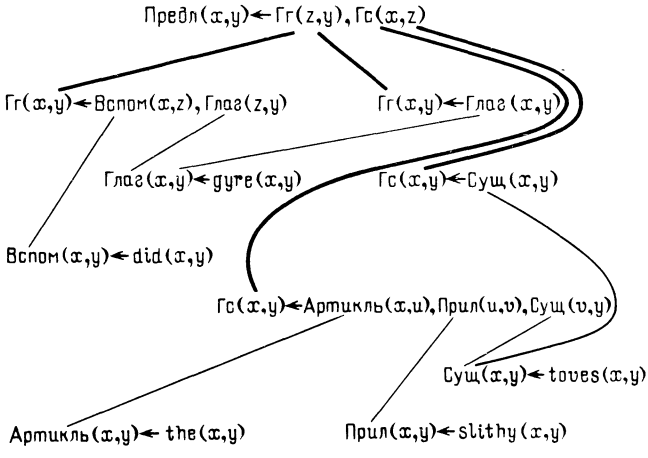

Предл(1,б)
Предл(x,y) ← Гг(z,y), Гс(x,z)
Гг(x,y) ← Вспом(x,z), Глаг(z,y)
Гг(x,y) ← Глаг(x,y)
Глаг(x,y) ← дуге(x,y)
Гс(x,y) ← Сущ(x,y)
дуге(5,б) ←
Гс(x,y) ← Артикль(x,y), Прил(u,v), Сущ(v,y)
Вспом(x,y) ← did(x,y)
Сущ(x,y) ← toves(x,y)
Артикль(x,y) ← the(x,y)
Прил(x,y) ← slithy(x,y)
did(4,5) ←
the(1,2) ←
slithy(2,3) ←
toves(3,4) ←
а
Гг(z,б), Гс(1,z)
Гг(4,y) ← Глаг(5,y)
Гг(x,y) ← Глаг(x,y)
Глаг(5,б) ←
Гс(1,y) ← Сущ(3,y)
Гс(x,y) ← Сущ(x,y)
Сущ(3,4) ←
в
Гг(z,б), Гс(1,z)
Гг(4,б) ←
Гг(5,б) ←
Гс(1,4) ←
Гс(3,4) ←
г
Гг(4,б)
Гг(4,б) ←
Гг(5,б) ←
д
е
Гг(z,б), Гс(1,z)
Гг(x,y) ← Вспом(x,z), Глаг(z,y)
Гг(x,y) ← Глаг(x,y)
Вспом(4,5) ←
Глаг(5,б) ←
Гс(x,y) ← Сущ(x,y)
Гс(x,y) ← Артикль(x,u), Прил(u,v), Сущ(v,y)
Артикль(1,2) ←
Прил(2,3) ←
Сущ(3,4) ←
б

Рис. 8.10

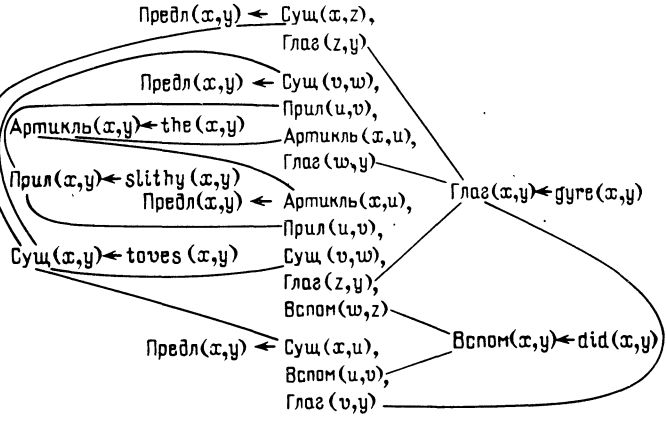
смешанный нисходяще-восходящий поиск. Как и в предыдущей задаче, выбираемые связи выделены на рис. 8.10 жирными линиями. Подстановки, помечающие связи, на графе не показаны.

Макропроцессирование и рассуждения, расходящиеся от центра

В обычных языках программирования *макропроцессирование* преобразует программу путем исключения всех вызовов некоторой процедуры за счет их предварительного, перед той задачей, которая должна быть решена, выполнения. Исходные процедуры заменяются новыми. Аналогом макропроцессирования в логике является расходящееся от центра рассуждение, дополненное удалением родительских клауз из-за того, что они содержат несвязанные атомы.



а



б

Рис. 8.11

Макропроцессирование обладает тем достоинством, что процедурные вызовы выполняются раз и навсегда перед заданными задачами, а не многократно во время попыток их решить.

Макропроцессирование можно продемонстрировать, удаляя все вызовы для процедур Гс и Гг в задаче грамматического разбора (см. рис. 8.11).

Использование стрелочных обозначений для управления выбором связей

Стрелочное обозначение, введенное в нашей книге раньше на неформальном уровне, можно использовать для управления выбором связей в процедуре доказательства на базе графа соединений. Связи графа соединений могут быть преобразованы в стрелки за счет придания им направленности. Клауза считается *активной*, если все связи ее атомов являются исходящими. Связь можно выбрать, если она соединяет атом в активной клаузе. Новые связи, соединяющие атомы в резольвенте, наследуют свою направленность от родительских связей, из которых они получились.

Процедура доказательства на базе графа соединений может быть сужена до нисходящего вывода за счет ориентации всех стрелок в направлении от посылок к заключениям. В этом случае клауза активна, если и только если она является целевым предложением. Приведенная на рис. 8.12 последовательность графов иллюстрирует использование стрелочных обозначений для наложения интерпретации нисходящего поиска решений на задачу об ошибающихся греках. Отметим, что несмотря на близость обозначений, нет никакой связи между стрелочными обозначениями в графе соединений и дугами в семантических сетях.

Процедуру доказательства можно сузить до восходящего вывода за счет ориентации всех стрелок в направлении от заключений к посылкам. В этом случае клауза активна, если и только если у нее нет посылок. Применение стрелочных обозначений для восходящего вывода показано на рис. 8.13.

Стрелочное обозначение можно использовать и в нехорновских клаузах для управления порождением утверждений, используемых в достижении подцелей. Нехорновская клауза в графе соединений на рис. 8.14, например, порождает утверждение

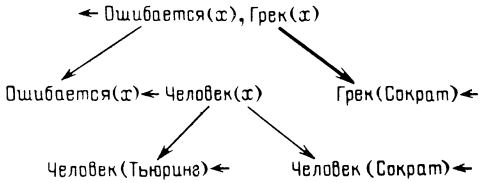
Студент (☺, Боб) ←

помогающее достижению подцели

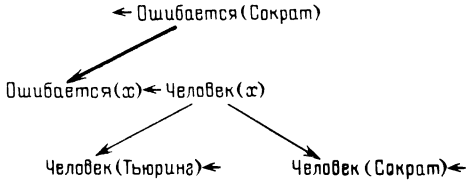
← Нравится (☺, логика).

Две клаузы, из которых получаются это утверждение и эта подцель, вместе с ассоциированным с ними стрелочным обозначением, составляет попытку показать, что Боб счастлив, за счет утверждения того, что ☺ — студент Боба и что даже ☺ нравится логика. Поскольку ничего иного не говорится об индивиде ☺ за исключением того, что можно показать, что ему нравится логика, постольку каждому, кто является студентом Боба, нравится логика. Эти две клаузы, следовательно, фактически утверждают, что

Боб счастлив, если всем его студентам нравится логика.



а

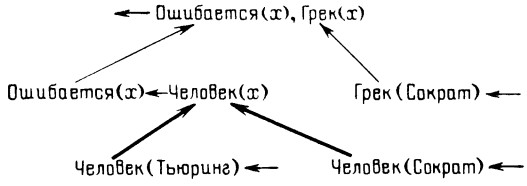


б

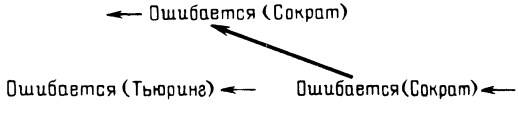
в

г

Рис. 8.12



а



б

в

Рис. 8.13

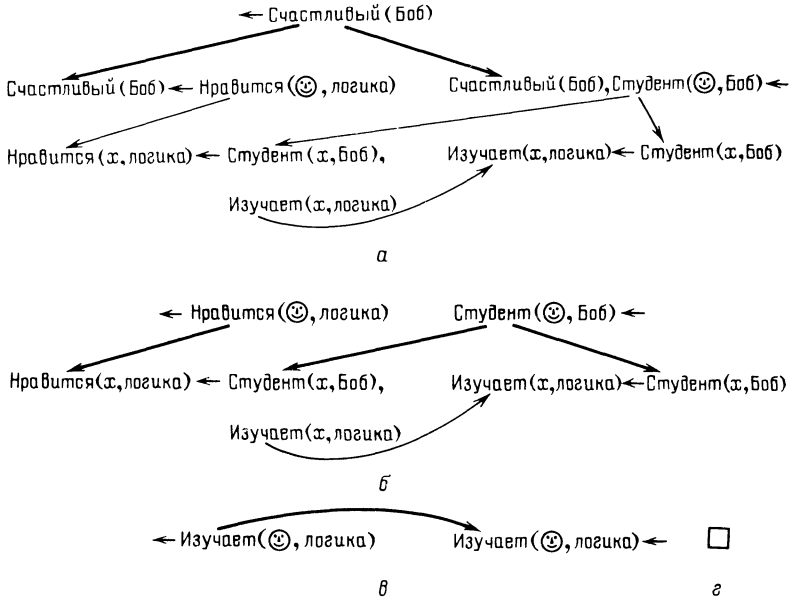


Рис. 8.14

Стрелки в графе соединений на рис. 8.14 ориентируют поиск не только сверху вниз от цели верхнего уровня и выводимых подцелей, но и снизу вверх от утверждения, которое надо использовать для достижения подцели.

Отметьте, что Боб был бы счастлив также и тогда, когда у него совсем бы не было студентов:

← Студент (x, Боб)

или, когда каждому нравилась бы логика:

Нравится (x, логика) ←

Нет никакой гарантии, что каждое назначение направления сохраняет разрешимость графа соединений. Более того, кажется оправданным ограничивать направленность стрелок так, чтобы все связи, соединяющие одинаковые атомы, имели бы одинаковое направление.

Авторезольвентные клаузы

Авторезольвентная клауза – это такая клауза, для которой резольвента может быть построена исходя из нее самой и ее копии. Клауза

Присоединить (x . y, z, x . y′) ← Присоединить (y, z, y′)

может сформировать резольвенту со своей копией

Присоединить (u . v, w, u . v′) ← Присоединить (v, w, v′)

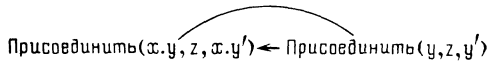


Рис. 8.15

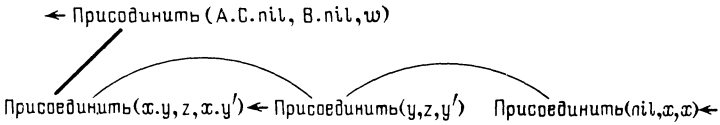


Рис. 8.16

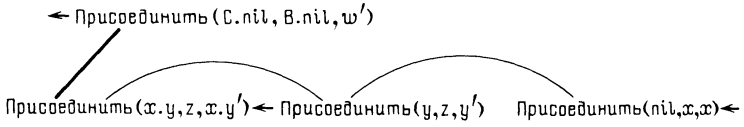


Рис. 8.17

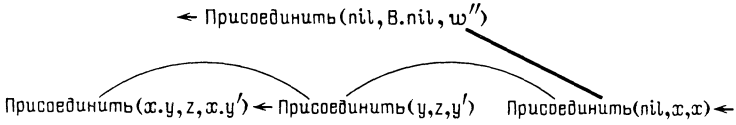


Рис. 8.18

□

Рис. 8.19

Для соответствия предыдущему изложению соединим атомы, вступающие в резолюцию в авторезольвентной клаузе, при помощи связи (рис. 8.15). Такая связь является *псевдосвязью*, потому что она связывает два атома в разных копиях одной и той же клаузы.

Псевдосвязи можно выбирать для дальнейшей обработки, но в целях наглядности лучше ограничивать их применение при выводе новых связей. Это можно показать на примере рис. 8.16.

Единственный атом резольвенты получается из атома, имеющего две связи, одна из которых – псевдосвязь. Из этой псевдосвязи получается обычная связь. Из другой связи, идущей в утверждение, новой связи не получается. Исходное целевое предложение содержит несвязанный атом, и, следовательно, отбрасывается, когда резольвента добавляется к графу (рис. 8.17).

Новый граф аналогичен исходному графу соединений. Однако на этот раз при порождении резольвенты не оказывается последователей у псевдосвязи, а у связи, ведущей к утверждению, они есть (рис. 8.18).

Резольвента новой связи – пустая клауза. Независимо от этого рекурсивную клаузу можно удалить, поскольку ее заключение обладает только псевдосвязью. Но как только рекурсивная клауза удаляется, тотчас же удаляется утверждение. Результирующий граф соединений поэтому состоит только из пустой клаузы (рис. 8.19).

В общем случае авторезольвентная клауза может быть удалена, если хоть один из ее атомов не имеет нормальной (не псевдо) связи. Наследственные свойства связей и псевдосвязей в графах соединений изучались Бранохе [Bruynooghe 1977]. Отметьте, что хотя во всех приведенных примерах окончательный граф соединений состоял только из пустой клаузы, в общем случае он с тем же успехом мог бы содержать и другие клаузы.

Удаление связей, резольвентами которых являются тавтологии

Клауза является *тавтологией*, если она содержит один и тот же атом и в качестве посылки и в качестве заключения. Использование тавтологий в нисходящем поиске решений задач приводит к циклам, в которых цель появляется вновь в качестве своей собственной подцели. По этой причине и потому, что тавтологии не вносят позитивный вклад в решение задач, их можно удалять из множества клауз без ущерба для несовместности [Robinson 1965a]. В процедуре доказательства методом графа соединений эффект от удаления тавтологий можно получить, удаляя связи, резольвентами которых являются тавтологии.

В качестве примера можно использовать множество клауз, описывающих понятие четного числа (рис. 8.20). У обеих связей, соединяющих две рекурсивные клаузы, резольвентами служат тавтологии. Поэтому клаузы удаляются из графа (рис. 8.21).

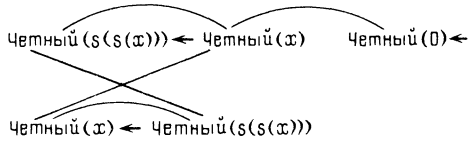


Рис. 8.20

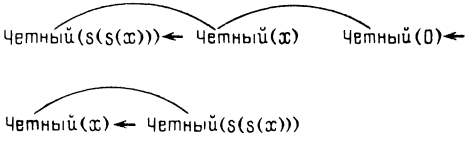


Рис. 8.21

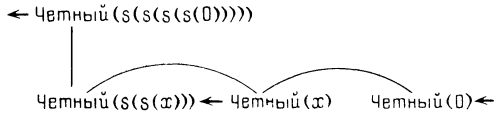


Рис. 8.22

Набор из трех клауз на рис. 8.21 совместен, поскольку он не содержит целевого предложения. Обе рекурсивные клаузы могут быть удалены, так как они содержат только атомы с псевдосвязями. Базовое утверждение можно удалить тоже. Если дано целевое утверждение

$$\leftarrow \text{Четный}(s(s(s(s(0))))),$$

то у посылки второй рекурсивной клаузы все еще нет связи, отличающейся от псевдосвязи. Следовательно, эта клауза может быть удалена, после чего остается простой граф (рис. 8.22).

В более сложных примерах распознать то, что некоторая клауза не вносит никакой вклад в решение, не столь легко. В таких случаях может оказаться полезным глобальный анализ. Глобальные стратегии поиска решений исследуются в следующей главе.

Процедура доказательства методом графа соединений

Подытожим теперь определение процедуры доказательства методом графа соединений в том же стиле, в каком показывалась связь естественного языка с клаузальной формой.

Для того, чтобы показать несовместность множества клауз при помощи *процедуры доказательства методом графа соединений*, следует породить и разрешить начальный граф соединений.

Начальный граф соединений для множества клауз содержит весь набор клауз этого множества, связи (но не псевдосвязи), соединяющие каждую пару согласующихся атомов на противоположных концах стрелок в различных клаузах, и псевдосвязи, соединяющие атомы на противоположных концах стрелки в одной и той же клаузе, если эти атомы согласуются в различных копиях этой клаузы.

Граф *разрешен*, если он содержит пустую клаузу.

Для *разрешения* графа соединений, не содержащего пустой клаузы, надо

либо удалить связь, резольвентой которой является тавтология и разрешить получившийся граф соединений,

либо удалить клаузу, содержащую несвязанный атом, с идущими от нее связями и разрешить получившийся граф соединений,

либо выбрать связь, не являющуюся псевдосвязью, удалить ее, добавить резольвенту вместе с ее новыми связями в граф и разрешить получившийся граф соединений.

Связь (*не псевдосвязь*) соединяет вхождение L некоторого атома в резольвенте с вхождением K некоторого атома в другой клаузе, если L и K согласуются, L получается из вхождения L′ некоторого атома в родительской клаузе, и имеется связь (возможно, псевдосвязь) между L′ и K (рис. 8.23).

Псевдосвязь соединяет L и K в резольвенте, если L и K согласуются, L и K получаются из L′ и K′ в (тех же самых или других) родительских клаузах, и имеется связь между L′ и K′ (рис. 8.24).

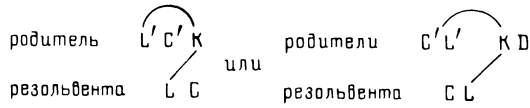


Рис. 8.23

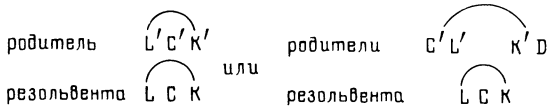


Рис. 8.24

Четыре различных пути разрешения графа соединений соответствуют четырем клаузам, имеющим одинаковые заключения. Если не учитывать удаление связей, резольвентами которых служат тавтологии, то три результирующие процедуры выражают логику и нисходящее управление итеративного алгоритма, описанного в начале этой главы. Этот первый алгоритм может быть получен из нового путем последовательного выделения управляющего компонента при использовании только что приведенных процедур. В частности,

(1) альтернативные способы разрешения графа соединений следует применять по одному за один раз в том порядке, в котором они были записаны здесь,

(2) бэктрекинг использовать не следует, так как недетерминизм первого рода в процедурах несуществен.

Та процедура доказательства, которая была здесь описана и именно в той форме, в которой она была описана, является неполной из-за того, что была опущена операция факторизации. Для того, чтобы избавиться от избыточности, на ее использование надо наложить некоторые ограничения. Поскольку адекватные ограничения еще не выдвигались и поскольку их отсутствие упрощает процедуру доказательства, мы решили проигнорировать операцию факторизации вообще. Определение процедуры доказательства, включающей факторизацию, можно найти в [Kowalski, 1974a].

Полнота процедуры доказательства методом графа соединений не может быть гарантирована, если выбор связей, необходимых для доказательства, откладывается на неопределенное время. Такое откладывание на неопределенное время может возникнуть, например, если стратегия выбора следует принципу поиска вглубь по неоканчивающемуся пути в пространстве нисходящего поиска. Требование того, что каждая связь должна быть явно подвергнута выбору, аналогично свойству исчерпываемости поисковых стратегий для более привычных процедур доказательства.

Доказательство полноты для случая процедуры доказательства методом графа соединений было получено Брауном (неопубликовано). В случае клауз Хорна его доказательство применимо также и к той процедуре доказательства, которая была описана здесь. Другие доказательства полноты для общего случая были анонсированы Зикманом и Штефаном [Siekmann, Stephan 1976] и Бибелом [Bibel 1979].

Некоторые процедуры доказательства используют графы соединений, но обрабатывают их не так, как описано здесь. Среди таких процедур выделяются созданные Сикелем [Sickel 1976] и Келлогом, Клэром и Трэвисом [Kellog, Klahr, Travis 1978]. Ближе всего, однако, к процедуре доказательства методом графа соединений находится неопубликованная аннулирующая система Колмероэ.

Упражнения

1. Выразите верхний уровень определения процедуры доказательства на базе графов соединений при помощи клауз Хорна.

2. Если использовать методы, описываемые позже, в гл. 10, для преобразования предложений из стандартной формы логики в клаузальную форму, то определение подмножества может быть задано при помощи таких двух клауз:

$$x \subseteq y, \text{произвольный}(x, y) \in x \leftarrow$$
$$x \subseteq y \leftarrow \text{произвольный}(x, y) \in y$$

Если эти клаузы применять сверху вниз, то они ведут себя как процедуры, задающие подцель вида $x \subseteq y$: утверждается, что некоторый произвольный индивид, скажем, произвольный (x, y) принадлежит x, и предпринимается попытка доказать, что он принадлежит y.

Используйте процедуру доказательства методом графа соединений для доказательства таких теорем:

а) пустое множество ϕ, определенное при помощи

$$\leftarrow x \in \phi$$

является подмножеством любого множества S;

б) каждое множество S есть подмножество универсального множества, определенного при помощи клаузы

$$x \in U \leftarrow$$

в) каждое множество есть подмножество самого себя;

г) множество A, такое, что

$$a(x), b(x) \leftarrow x \in A$$

является подмножеством множества B, такого, что

$$x \in B \leftarrow a(x)$$
$$x \in B \leftarrow b(x)$$
$$x \in B \leftarrow c(x)$$

все это является формулировкой без использования равенства задачи доказательства того, что

$$\{a, b\} \subseteq \{a, b, c\}$$

д) если множество A является объединением двух подмножеств B и C, что может быть определено клаузами

$x \in A \leftarrow x \in B$

$x \in A \leftarrow x \in C$

то B и C являются подмножествами A.

3. Проверьте заключение, сделанное в гл. 5 о том, что при использовании процедуры доказательства методом графа соединений, нисходящее исполнение определения чисел Фибоначчи

$$\begin{aligned} &\text{Фиб}(0, s(0)) \leftarrow \\ &\text{Фиб}(s(s(x)), v) \leftarrow \text{Фиб}(s(x), u), \\ &\qquad\qquad\qquad\quad \text{Фиб}(x, v), \\ &\qquad\qquad\qquad\quad \text{Плюс}(u, v, w) \end{aligned}$$

требует лишь постоянного объема памяти. Предполагайте, что отношение Плюс определено при помощи свободных от переменных утверждений и не учитывайте объем памяти, нужный для их хранения.

В этой главе мы будем исследовать такие стратегии поиска решений, которые применяются к задачам в целом, а не по отдельности к их подзадачам. Стратегия *преобразования целей* будет применяться к комбинации целей в целевом предложении, а стратегия *анализа различий* — к воздействию процедур на различие между целями и утверждениями.

Преобразование целей состоит из набора относительных стратегий, которые касаются логических связей между подцелями. Оно включает *удаление избыточных подцелей,* которые можно вывести из других подцелей, *добавление неявных подцелей,* которые легче и целесообразнее достигать, нежели те, что представлены явно, *устранение противоречивых подцелей,* которые взаимно несовместны и, наконец, *устранение подцелей, которые опровергаются примерами.*

Методы преобразования целей аналогичны методам преобразования программ, разработанных для рекурсивных уравнений Берстэллом и Дарлингтоном [Burstall, Darlington 1977]. "Преобразование программ" преобразует их до того, как сформулированы задачи, а "преобразование целей" занимается преобразованием целей во время попыток их достижения. Методы преобразования целей применяются также при построении планов для роботов, в математическом программировании и доказательстве геометрических теорем.

Анализ различий между целями и утверждениями включает в себя более глобальный подход к поиску решений. Он пытается распознать процедуры, *уменьшающие различия,* процедуры, *увеличивающие различия,* и процедуры, оставляющие их *неизменными.*

Таким процедурам, которые уменьшают различия, обычно отдается предпочтение перед неуменьшающими различия процедурами. Это вызвано тем, что некоторую цель можно устранить как недостижимую (из-за неразрешимости попыток ее достижения), если удается показать, что вообще не найдется процедуры, уменьшающей различия.

Методы анализа различий аналогичны методам, используемым в доказательстве правильности программ. Демонстрация того, что программы уменьшают различия, обычно включается в процесс доказательства остановки программы, а демонстрация того, что программы оставляют свойства неизменными, используется при доказательстве наличия у программ некоторых свойств. Заметим, что стратегия выбора процедур по их эффектив-

ности в аспекте уменьшения различий является основой Универсального Решателя Задач, разработанного Ньюэллом, Шоу и Саймоном [Newell, Shaw, Simon 1963].

Хотя методы, нами описываемые, применимы и к нехорновским клаузам, мы упростим изложение, ограничившись лишь нисходящим поиском решений на языке клауз Хорна. К тому же никак не будут затронуты эвристики, которые могли бы пригодиться для эффективного использования описываемых методов.

Удаление избыточных подцелей

Подцель может быть удалена из целевого предложения, если из предположения о том, что имеет решение задача достижения других подцелей, следует, что сходное решение имеет и задача достижения рассматриваемой избыточной подцели. Если следовать этому критерию и предполагать транзитивность отношения $\leqslant$

$$x \leqslant y \leftarrow x \leqslant z,\ z \leqslant y$$

то можно прийти к выводу, что целевое предложение

$$\leftarrow r \leqslant s,\ s \leqslant t,\ r \leqslant t$$

содержит избыточную подцель $r \leqslant t$. Ибо, если допустить, что остальные подцели достижимы, то этот вывод следует из того, что утверждения

$$r' \leqslant s' \leftarrow$$

$$s' \leqslant t' \leftarrow$$

истинны для произвольных примеров r', s', t' термов r, s и t соответственно. Но из этих утверждений, дополненных транзитивностью $\leqslant$ следует утверждение

$$r' \leqslant t' \leftarrow$$

которое выражает тот факт, что третья подцель достижима вместе с остальными. И нет необходимости обеспечивать явное достижение избыточной подцели. Достаточно лишь знать, что любое решение, обеспечивающее достижение остальных подцелей, гарантирует существование сходного решения, обеспечивающего достижение третьей подцели.

Клаузе транзитивности вовсе нет необходимости быть частью программы или даже логическим следствием ее. Для оправдания факта удаления избыточной подцели достаточно, чтобы транзитивность (в данном случае) была бы свойством программы. Это имеет место в случае, когда, например, отношение $\leqslant$ определяется клаузами

$$0 \leqslant y \leftarrow$$

$$s(x) \leqslant s(y) \leftarrow x \leqslant y$$

Предложение является *свойством* программы на языке клауз Хорна P, если это предложение совместимо с P и если из этого предложения вместе с P нельзя вывести никакое свободное от переменных утверждение, которое не являлось бы следствием одной P. Свойство программы поэтому не добавляет никаких новых решений к тем, которые можно получить исходя из самой программы.

Удаление повторного вхождения подцели является специальным случаем удаления избыточной подцели, потому что из любого вхождения подцели следует любое другое вхождение. Поэтому, к примеру, целевое предложение

$$\leftarrow P, Q, P$$

может быть заменено на

$$\leftarrow P, Q$$

Добавление замещающих подцелей

Хотя удаление избыточных подцелей почти всегда является полезным, иногда не менее выгодным оказывается их добавление.

Стратегия вывода добавочных подцелей часто встречается в математическом программировании, когда подцели трактуются как ограничения, которые должны быть удовлетворены. *Замещающее ограничение,* выполнение которого является следствием выполнения исходных ограничений, может быть добавлено и удовлетворено до рассмотрения других ограничений. Такие действия оказываются полезными, если проверка выполнения замещающего ограничения проще проверки других ограничений и если выполнение замещающего ограничения приводит к удовлетворению требований других ограничений за счет фиксации значений некоторых переменных.

Рассмотрим, например, исходную совокупность двух ограничений, содержащих переменные x и y

$$\leftarrow x + y = 2, \quad x - y = 0$$

Последовательная, работающая сверху вниз система поиска решений будет порождать пары чисел, удовлетворяющих одному из ограничений, а затем проверять, обеспечивают ли эти числа выполнение второго ограничения. Более интеллектуальная система поиска решений, запрограммированная Уорреном на Прологе, обрабатывает обе подцели как сопрограмма, пытаясь достичь их одновременно методом последовательной аппроксимации. Эта программа, являющаяся системой общего назначения для поиска решений задач на языке клауз Хорна, всегда выбирает ту подцель, которая содержит меньше всего переменных на самом верхнем уровне.

Обычный математический метод поиска решений вместо этого выводит добавочные замещающие ограничения и пытается добиться их выполнения. Этот метод предполагает, что исходные ограничения могут быть выполнены, и выводит из этого предположения (при помощи почленного сложения обоих уравнений), что добавочное ограничение

$$2 * x = 2$$

тоже должно быть выполнено. Новое ограничение является, несомненно, избыточным в новом целевом предложении:

$$\leftarrow x + y = 2, \; x - y = 0, \; 2 * x = 2,$$

но выполнения этого ограничения можно добиться, не прибегая к поиску. Более того, как только получено решение, обеспечивающее выполнение добавочного ограничения, для оставшихся исходных ограничений (уже

означенных) также можно будет получить выполняющее их решение безо всякого поиска. Действительно, достаточно найти решение в точности для одного из ограничений, поскольку остальные сразу становятся избыточными.

Стратегия замещающих подцелей полезна и в задачах построения планов. Рассмотрим, к примеру, задачу поиска состояния w, в котором робот находится в комнате u и располагается рядом с кубиком:

$$\leftarrow \text{В (Роб, комната, w), Рядом (Роб, кубик, w)}$$

Допуская, что кубик может вначале находиться вне комнаты и что робот более подвижен, чем кубик, полезно вывести замещающую подцель

$$\leftarrow \text{В (кубик, комната, w)}$$

из исходных подцелей, используя свойства программы.

$$\text{В (x, y, w)} \leftarrow \text{В (z, y, w), Рядом (x, z, w)}$$

$$\text{Рядом (x, z, w)} \leftarrow \text{Рядом (z, x, w)}$$

Если замещающая подцель добавляется к исходным целевым предложениям

$$\leftarrow \text{В (Роб, комната, w), Рядом (Роб, кубик, w), В (кубик, комната, w)}$$

и выбирается для решения перед остальными, то простейшее ее решение (когда робот заталкивает кубик в комнату), определяет состояние w, в котором непосредственно обеспечивается достижение оставшихся подцелей.

Устранение несовместных целевых предложений

Целевое предложение можно полностью устранить как неразрешимое, если предположение о том, что оно разрешимо, приводит к противоречию.

Один из наиболее простых случаев — это ситуация, когда целевое предложение перекрывается свойством программы или ограничением целостности. Так, целевое предложение

$$G \quad \leftarrow \text{На (A, B, w), Свободно (B, w), На (B, C, w)}$$

перекрывается клаузой

$$C \quad \leftarrow \text{На (x, y, z), Свободно (y, z),}$$

которая означает невозможность ситуации, когда некая вещь является свободной, но одновременно на ней что-то находится. В общем случае одна клауза C_1 *перекрывает* другую клаузу C_2, если все посылки и заключения какого-нибудь примера C_1 содержатся среди посылок и заключений C_2. Перекрывающая клауза является более общей, чем перекрываемая клауза и обычно имеет меньше посылок или меньше заключений. В только что рассмотренном случае пример перекрывающей клаузы C (в которой x = A, y = B и z = w) содержит на одну посылку меньше, чем перекрываемая клауза G.

Некоторую клаузу можно удалить из множества клауз, если она перекрывается другой клаузой из этого же множества. Удаление перекрывае-

мой клаузы не влияет на совместность (или несовместность) множества клауз в целом. Подробное обсуждение вопросов полноты при удалении перекрываемых клауз содержится в книге Лавлэнда [Loveland 1978].

Стратегию удаления перекрываемого целевого предложения можно рассматривать в качестве специального случая удаления несовместной клаузы. В предыдущем случае допущение

На (A, B, s) ←
Свободно (B, s) ←
На (B, C, s) ←

о том, что существует решение w = s целевого предложения G, несовместно с C.

Устранение несовместного целевого предложения, однако, является более общим действием, чем удаление перекрываемого. Она может включать любое количество дедукций. Запрос к базе данных

← Преподает (Джон, y)

например, не перекрывается любой из клауз

T1 Преподаватель (x) ← Преподает (x, y)
T2 ← Преподаватель (x), Студент (x)
T3 Студент (Джон) ←

но является неразрешимым, поскольку допущение о том, что он разрешим, т.е.

Преподает (Джон, A) ←

несовместно с T1 . . 3

Похожие стратегии устранения запросов, несовместных с задаваемой информацией, разрабатывались Макскимином и Минкером [McSkimin, Minker 1977], которые дополнили резолюционную систему доказательства теорем семантической сетью, которая хранит и обрабатывает задаваемую информацию. Перекрытие неразрешимых целевых предложений является также особенностью систем построения планов, разрабатывавшихся Доусоном и Шиклоши [Dowson, Siklossy 1977], Хьюиттом [Hewitt, 1975] и, на более общем уровне, особенностью логических программных систем, разрабатывавшихся Робинсоном и Зибертом [Robinson, Sibert 1978].

Обобщение использования диаграмм в геометрических доказательствах

Для обоснования добавления или удаления избыточных подцелей нужно, чтобы допущения, используемые для вывода подцели, были бы свойствами процедур, которые можно применять для достижения этих подцелей. Но для обоснования устранения несовместности целевых предложений достаточно выполнение более слабого условия: для используемых при выводе несовместности посылок A нужна лишь *совместность* с процедурами P:

Так, предположим, что

(i) P совместно с A;

(ii) G^* выражает то, что целевое предложение G достижимо;
(iii) G^* несовместно с P и A;
(iv) P решает задачу достижения G

Тогда, коль скоро P решает задачу достижения G, G^* следует из P и поэтому из P вместе с A следует G^*. Но тогда, поскольку P совместно с A, G^* совместно с P и A, что противоречит (iii). Отсюда следует, что если P совместно с A, но G^* несовместно с P и A, то P не решает задачу достижения G.

Использование диаграмм для устранения недостижимых подцелей в Машине для Доказательства Геометрических Теорем Гелернтера [Gelernter 1963] можно считать частным случаем использования допущений, которые совместны с процедурами поиска решений, для устранения несовместных целевых предложений. Геометрические аксиомы играют роль процедур, а описание диаграмм играет роль добавочных посылок. Применение диаграмм обосновывается тем, что их описание совместно и с общими геометрическими аксиомами, и с частными предположениями о доказываемых теоремах. Гелернтер установил, что применение диаграмм уменьшает размер поискового пространства в среднем до 1/200 от его начального размера. Вышеприведенный аргумент показывает, что использование примеров для распознавания неразрешимости задачи не обязательно сводить лишь к области геометрии. Можно использовать примеры и для распознавания, и для устранения недостижимых подцелей в любой проблемной области.

Цели как обобщенные решения

Иногда оказывается полезным не пытаться получать решения по достижению подцелей явно, но вместо того рассматривать эти подцели как обобщенные описания полных классов решений, обеспечивающих их достижение.

Рассмотрим, например, начальное целевое предложение

$$\leftarrow G(x)$$

которое в конечном счете сводится к подцели

$$\leftarrow x > 0$$

Вместо того, чтобы порождать произвольное целое число x как явное решение, более информативным оказывается сообщение о том, что любое положительное число является решением. Это сообщение может быть сделано путем объявления подцели $x > 0$ в качестве обобщенного решения, которое обозначает класс всех индивидуальных решений по достижению этой цели.

Достижение подцелей с помощью обобщенных решений является одной из особенностей подхода Бледшоу к доказательству теорем [Bledsoe 1971, 1977]. Для вящей эффективности его следовало бы соединить с преобразованием целей. Если дано, например, целевое предложение

$$\leftarrow x > 0,\ \ x > 1,\ G(x)$$

то для его преобразования к новому виду

$$\leftarrow x > 1, \quad G(x)$$

необходимо выполнить удаление избыточной подцели. Если, с другой стороны, дано

$$\leftarrow x < 0, \quad x > 1, \quad G(x)$$

то устранение несовместных подцелей необходимо для распознавания факта недостижимости цели.

Восприятие некоторых типов подцелей в качестве обобщенных решений полезно также и в запросных системах баз данных. Это является особенностью информационно–поисковой резолюционной системы Дарлингтона [Darlington 1969] и системы доказательства теорем на семантических сетях Макскимина и Минкера [McSkimin, Minker 1977]. Если дан запрос

Кто преподает программирование?
← Преподает (x, программирование)

и дано общее правило

Все профессора преподают программирование
Преподает (x, программирование) ← Профессор (x),

то лучше рассматривать результирующую подцель как обобщенное решение

← Профессор (x),

нежели выдать один или более ответов, которые формируют решение в виде утверждений

Профессор (Мери) ←
Профессор (Джон) ←
Профессор (Боб) ←

Преобразование целей и информационный взрыв

Общей чертой человеческих подходов к поиску решений является то, что добавление информации в общем случае повышает эффективность поиска решений. Это контрастирует с упрощенной моделью поиска решений задач, когда все знания используются в качестве процедур поиска решений. Добавочная информация лишь увеличивает размер поискового пространства и затрудняет решение задачи (за исключением тех случаев, когда требуется только одно решение, а недетерминизм первого рода несуществен). Однако в модели, допускающей преобразование целей, добавочная информация может использоваться для преобразования целевых предложений и для уменьшения размера поискового пространства.

Распознавание зацикливаний путем анализа различий

Как и преобразование целей, анализ различий может обеспечить возможность распознавания зацикливания процедуры.

Рассмотрим, например, процедуру

Числ (x) ← Числ (s(x))

задавшись целью

← Числ (s(s(0)))

и утверждением

Числ (0) ←

Повторяя сверху вниз выполнение этой процедуры, мы получим необрывающуюся, бесконечную последовательность подцелей (рис. 9.1).

В этом случае процедура доказательства методом графа соединений устраняет зацикливание, поскольку процедурный вызов Числ (s(x)) обладает лишь псевдосвязью с заголовком процедуры. Отсюда следует, что эта процедура неприменима, и может быть удалена из графа. Если однако утверждение

Числ (x) ←

заменяется утверждением

Числ (s(0)) ←

то применение процедуры вновь приводит к тому же бесконечному циклу, который на этот раз не может быть устранен, поскольку его процедурный вызов обладает добавочной непсевдосвязью с новым утверждением. Тем не менее зацикливание можно устранить во всех этих случаях, если можно распознать, что применение данной процедуры не может устранить различие между целью и утверждением. Цель отличается от утверждения тем, что она содержит большее количество вхождений функционального символа s. Применение процедуры лишь увеличит это различие за счет порождения подцелей, которые содержат еще больше вхождений s.

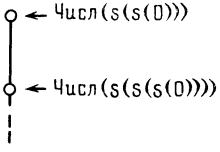


Рис. 9.1

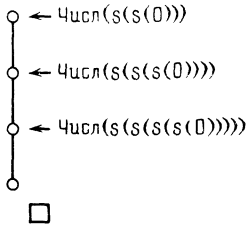


Рис. 9.2

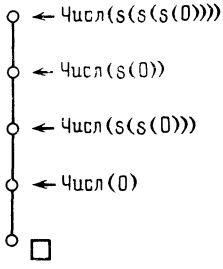


Рис. 9.3

Глобальная природа анализа различий становится видимой невооруженным глазом, если наше утверждение заменить новым

Числ (s(s(s(s(0))))) ←

Теперь применение процедуры уменьшает различие между целью и утверждением и, в конце концов решает задачу (рис. 9.2).

Процедура может пригодиться для решения даже если она увеличивает различие между целью и утверждениями. Если даны, например, цель

$\leftarrow$ Числ (s(s(s(0))))

и утверждение

Числ (0) $\leftarrow$

го процедура

Числ (s(s(x))) $\leftarrow$ Числ (x)

уменьшает различие, в то время как процедура

Числ (x) $\leftarrow$ Числ (s(x))

увеличивает его. Но обе процедуры необходимы для решения задачи (рис. 9.3).

В предыдущих примерах применение процедуры, увеличивающей различия, либо порождает цикл, либо оказывается существенным для решения. Более часты случаи, когда увеличение различий ни помогает решению, ни препятствует его нахождению. Так получается, например, в случае с парой процедур

Числ (s(x)) $\leftarrow$ Числ (x)
Числ (x) $\leftarrow$ Числ (s(x))

Если одна из них приводит к ненужному увеличению различий, то вторую можно использовать для возвращения их к прежнему состоянию. В самом деле, использование одной процедуры после другой просто порождает подобие цикла, который можно устранить в процедуре доказательства методом графа соединений путем удаления связей, резольвентами которых служат тавтологии.

Во всех этих примерах различие между подцелями и утверждениями может быть просто измерено путем подсчета количества вхождений функционального символа s. В других случаях характеризация различий может оказаться более сложной.

Пример "Факториал числа"

Определение факториала представляет более реальную ситуацию. Неклаузальное высказывание

Умножить (s(x), u, v) $\rightarrow$ [Факт (x, u) $\leftrightarrow$ Факт (s(x), v)]

порождает две процедуры на языке клауз Хорна:

(1) Факт (s(x), v) $\leftarrow$ Факт (x, u), Умножить (s(x), u, v)
(2) Факт (x, u) $\leftarrow$ Факт (s(x), v), Умножить (s(x), u, v)

Если задаться утверждением

Факт (0, s(0)) $\leftarrow$

то не найдется цели, для которой пригодилась бы вторая процедура. Но, если вместо этого задаться утверждением

Факт (10, 3628800) $\leftarrow$

то вторая процедура становится необходимой для решения задачи

← Факт (s(0), x)

а первая процедура необязательна. Здесь натуральное число используется как сокращение терма

$$\underbrace{s(s(s(\ldots(0)\ldots)))}_{\text{n раз}}$$

содержащего n вхождений функционального символа s.

В более общем случае может оказаться полезным иметь несколько утверждений, например

Факт (0, 1) ←
Факт (10, 3628800) ←

и, используя анализ различий, применять ту процедуру, которая быстрее сужает брешь между задачей и утверждениями, используя (1), например, для задачи

← Факт (3, x)

и (2) для задачи

← Факт (8, x)

Отметьте, что последний пример есть случай "не забочусь" в условиях недетерминизма первого рода. Есть несколько способов поиска факториала, каждый из которых приводит к одному и тому же результату. И не важно, какой из этих способов будет выбран. Но, если применяется бэктрекинг, то важно (по причинам эффективности), чтобы использовался только один из методов.

Инвариантные свойства процедур

Неразрешимость задачи может быть выявлена не только путем анализа действия процедур на различия, но также и путем анализа тех свойств, которые являются инвариантными относительно процедур. Задача может быть распознана как неразрешимая, если можно показать, что она отличается от утверждений по такому свойству, которое не подвержено воздействию процедур. Типичным свойством такого рода является четность.

Предположим, что нам даны клаузы

Четное (8) ←
Четное (s(s(x))) ← Четное (x)
Четное (x) ← Четное (s(s(x)))
← Четное (17)

После анализа различий вторую процедуру можно устранить как бесполезную. Ибо, если использовать только ее одну, то различия возрастают. Если использовать ее вместе с остальными процедурами, то лишь породятся циклы. После анализа инвариантов первую процедуру можно устранить тоже. Она сводит задачу данной четности к задаче такой же четности. Независимо от того, сколько раз применяется эта процедура, она не может изменить четность исходной задачи. Поскольку исходная задача обладает нечет-

ным количеством вхождений s, а утверждение обладает четным числом этих вхождений, наша процедура не может быть применена к решению задачи. Здесь четность может быть определена путем подсчета количества вхождений функционального символа s. В более реальных случаях инвариантные свойства более сложны.

Таков следующий пример, где инвариантное свойство представляет собой другую форму проявления четности. Если дана последовательность из шести стрелок (или монет), каждая из которых может находиться вверх лицевой поверхностью или вниз ею, то задача состоит в том, чтобы перевести ее из одного состояния в другое, например, из

U U U D D D в U U D D U U *)

Имеется только одно допустимое действие: можно одновременно менять направление двух соседних стрелок.

Простое кортежное представление задачи, когда запись

Состояние $(d_1, d_2, d_3, d_4, d_5, d_6)$

означает, что одновременно направление первой стрелки есть d_1, направление второй стрелки есть d_2, . . . , направление i-й стрелки есть d_i, выглядит так:

Состояние (U, U, U, D, D, D) ←
← Состояние (U, U, D, D, U, U)

Состояние (x, y, z, u, v, w) ← Состояние (x′, y′, z, u, v, w),
Противоположны (x, x′),
Противоположны (y, y′)

Состояние (x, y, z, u, v, w) ← Состояние (x, y′, z′, u, v, w),
Противоположны (y, y′),
Противоположны (z, z′).

Состояние (x, y, z, u, v, w) ← Состояние (x, y, z′, u′, v, w),
Противоположны (z, z′),
Противоположны (u, u′).

Состояние (x, y, z, u, v, w) ← Состояние (x, y, z, u′, v′, w),
Противоположны (u, u′),
Противоположны (v, v′).

Состояние (x, y, z, u, v, w) ← Состояние (x, y, z, u, v′, w′),
Противоположны (v, v′),
Противоположны (w, w′).

Противоположны (U, D) ←
Противоположны (D, U) ←

Задача неразрешима, поскольку, в то время как процедуры оставляют инвариантной четность количества стрелок в каждом направлении, в утверждении имеется нечетное количество стрелок в каждом из двух направлений, а в цели – присутствует четное их количество. Для доказательства того, что процедура оставляет четность инвариантной, необходимо рассмотреть два случая: в первом случае две инвертируемые стрелки имеют

*) U – вверх, D – вниз. – *Примеч. пер.*

одно направление перед инверсией, а во втором они имеют разные направления перед инверсией. Если у них одинаковые направления, то инверсия увеличивает количество стрелок, ориентированных в одном направлении на две и уменьшает количество стрелок, ориентированных в другом направлении тоже на две, но оставляет четность неизменной. Если же стрелки имеют разные направления, то инверсия оставляет количество стрелок, ориентированных в обоих направлениях, без изменения и, поэтому, не влияет на четность. В обоих случаях четность есть инвариантное свойство процедур.

Подобна только что рассмотренной и задача о сломанной шахматной доске. Пусть дана шахматная доска с двумя удаленными квадратиками в противоположных углах (рис. 9.4); задача состоит в том, чтобы покрыть доску костяшками домино, причем каждая из них должна покрывать два соседних квадратика. Поскольку соседние квадратики – разноцветные,

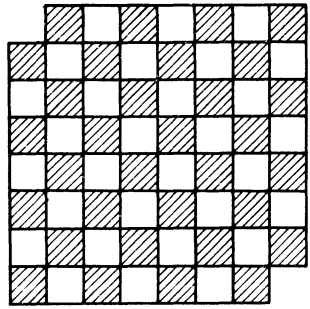


Рис. 9.4

процедура оставляет инвариантной разницу между количествами непокрытых квадратиков разных цветов. Поэтому задача неразрешима, ибо в целевом состоянии эта разница равна нулю, а в начальном – она равна двум.

Имеется очевидное сходство между доказательством того, что логическая процедура оставляет некоторое свойство инвариантным, и доказательством наличия некоторого свойства блок–схемы программы с использованием инвариантов. В обоих случаях цель состоит в том, чтобы показать, что если свойство выполняется перед началом повторяющегося процесса, то оно выполняется и после его конца. Предварительно нужно установить, что из выполнения свойства перед началом какого-либо шага процесса следует выполнение этого свойства в конце этого шага. После этого требуемый результат устанавливается по индукции.

Упражнения

1. Предположим, что y является функцией x в отношении F(x, y), т.е.

$$y = z \leftarrow F(x, y), F(x, z),$$

где *единственной* клаузой, определяющей равенство является

$$x = x \leftarrow$$

Покажите как можно использовать преобразование цели для устранения избыточности, когда целевое предложение содержит пару подцелей вида

F(r, s) и F(r, t)

где r, s и t суть термы.

2. Покажите, что преобразование цели можно использовать для обоснования преобразования клаузы

Башня (t(x, y)) ← Блок (x), Башня (y), На (x, y)

в клаузу

Башня (t(x, y)) ← Башня (y), На (x, y)

Какое *свойство* отношения На нужно для такого преобразования?

3. В гл. 6 предусловие Разл (x, z) можно удалить в формуле (12) из определения действия перен(x, y, z) и его использование можно заменить

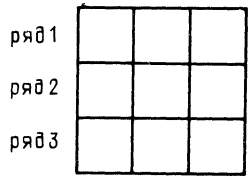


Рис. 9.5

на использование ограничения целостности

← Сохраняет (на (x, x), w)

Сравните поведение решателя задач для этих двух альтернативных формулировок задачи построения плана.

4. Проанализируйте предложение

S1 Откажитесь от воровства как способа приобретения чего-либо, если вы хотите быть добродетельным

как рекомендацию по применению процедуры

Иметь (u, x) ← Воровать (u, x)

к целевым предложениям, содержащим две подцели

Иметь (r, s) и Добродетельный (r)

Могут ли понятия преобразования целей использоваться для установления логической связи между предложением S1 и предложениями S2 и S3

S2 Не воруйте, если хотите быть добродетельным

S3 Каждый, кто ворует, не является добродетельным.

5. Обсудите формализацию следующих задач и стратегии поиска решений, необходимые для их интеллектуального решения

а) Найдите такое назначение цифр 1, 2, 3, . . . , 9 в ячейки матрицы 3 × 3 (рис. 9.5), чтобы выполнялись следующие требования:

(i) в точности одна цифра назначается в одну ячейку;
(ii) ни одна цифра не назначается более, чем в одну ячейку;
(iii) трехразрядное число в третьем ряду является суммой трехразрядных чисел в ряду 1 и в ряду 2
(iv) если цифра i назначается в некоторую ячейку, то цифра i + 1 назначается в ячейку, которая соседствует с первой по горизонтали или по вертикали.

б) Найдите такие назначения цифр 1, 2, 3, . . . , 9 на месте букв в приводимой ниже формуле, чтобы:

```
  DONALD
+
  GERALD
  ROBERT
```

(i) в точности одна цифра назначалась бы на место каждой буквы;
(ii) ни одна цифра не назначалась бы более, чем одной букве;
(iii) шестизначное число, назначенное слову ROBERT, являлось бы суммой шестизначных чисел, назначенных словом DONALD и GERALD;
(iv) букве D назначалась бы цифра 5.

10. СООТНОШЕНИЕ МЕЖДУ КЛАУЗАЛЬНОЙ ФОРМОЙ И СТАНДАРТНОЙ ФОРМОЙ ЛОГИКИ

Клаузальная форма проще стандартной формы логики и обладает бóльшим сходством с формальными средствами, используемыми в базах данных и программировании. К тому же, правило резолюций в этой форме гораздо больше походит на обычные правила обработки информации и поиска решений.

Несмотря на то, что описание любой задачи можно преобразовать из стандартной формы в клаузальную, стандартная форма все же остается более экономной и более естественной, чем результирующий набор клауз. В частности, спецификации программ составляют одну из тех областей, где стандартная форма логики (или некоторое приемлемое расширение представления на языке клауз Хорна) оказывается более подходящей, чем простая клаузальная форма. В рамках стандартной формы логики, кроме того, более естественным оказывается вывод (извлечение) программ из спецификаций. В качестве утешения остается только заметить, что полезные системы вывода для стандартной формы логики можно получить лишь на основе комбинации правил вывода клаузальной формы с правилами преобразования из стандартной формы в клаузальную.

Введение в стандартную форму логики

Мы представим сейчас лишь неформальную семантику стандартной формы логики, сопоставляя между собой высказывания на естественном языке с высказываниями в стандартной форме логики. Такие понятия как "совместность" выражений в стандартной логике могут неформально интерпретироваться в естественном языке по аналогии.

Стандартная форма логики обеспечивает явное символьное представление для *пропозициональных связок* И, ИЛИ, НЕ, ЕСЛИ . . . ТО . . . , и ЕСЛИ-И-ТОЛЬКО-ЕСЛИ, а также для кванторов ДЛЯ ВСЕХ и СУЩЕСТВУЕТ. Пропозициональные связки позволяют строить более сложные высказывания из более простых. Пусть

символ & обозначает И,
символ ∨ обозначает ИЛИ,
символ ¬ обозначает НЕ,
символ → обозначает ЕСЛИ . . ТО . . или СЛЕДУЕТ ИЗ,
символ ↔ обозначает ЕСЛИ-И-ТОЛЬКО-ЕСЛИ.

Клауза $A_1, \ldots, A_m \leftarrow B_1, \ldots, B_n$, не содержащая переменных, в стандартной форме записывается так:

$$[B_1 \& \ldots \& B_n] \rightarrow [A_1 \vee \ldots \vee A_m]$$

Если n = 0, то в стандартной форме логики опускают стрелку:

$$A_1 \vee \ldots \vee A_m$$

Если m = 0, то стрелка превращается в знак отрицания:

$$\neg [B_1 \& \ldots \& B_n]$$

В стандартной форме направление знака импликации → противоположно тому, которое мы использовали в клаузальной форме. Но, как и у арифметических знаков неравенства ⟨ и ⟩, направление знака импликации не фиксируется жестко. Поэтому выражения

$$A \rightarrow B \text{ и } B \leftarrow A$$

эквивалентны. Но отметьте, что

$$A \rightarrow B \text{ и } A \leftarrow B$$

не эквивалентны.

Предложения в стандартной форме логики можно строить также с помощью двух кванторов:

квантор общности

$\forall x$ обозначает "ДЛЯ ВСЕХ x",

квантор существования

$\exists x$ обозначает "СУЩЕСТВУЕТ НЕКОТОРЫЙ x"

Пример. Некоторых устриц можно есть с лимоном.

Клаузальная форма

Устрица (x) ←
Есть-с-лимоном (x) ←

Стандартная форма

$\exists x$ [Устрица (x) & Есть-с-лимоном (x)]

Для того, чтобы сослаться на некоторый индивид в формулировках на языке клауз, этому индивиду необходимо придать имя. Квантор существования позволяет делать ссылки на индивиды без указания их имен. В клаузальной форме предложения соединяются связкой И неявно. В стандартной же форме конъюнкция & должна быть указана явно.

Пример. У каждого человека есть мать.

Стандартная форма

$\forall x \exists y$ [Человек (x) → Мать (y, x)]

Клаузальная форма

Мать (мама (x), x) ← Человек (x)

В клаузальной форме приходится использовать функциональный символ для того, чтобы назвать индивид y, который существует как функция x.

Меняя порядок кванторов, мы меняем и смысл. Так, предложение

$$\exists y \forall x\ [\text{Человек}\ (x) \rightarrow \text{Мать}\ (y, x)]$$

означает, что существует уникальный индивид, который является общей матерью для всех нас. Для именования такого индивида в клаузальной форме надо использовать константный символ

$$\text{Мать}\ (\text{☺}, x) \leftarrow \text{Человек}\ (x)$$

Для точного определения предложения необходимо определить более общее понятие формулы. Формула может содержать свободные (не связанные кванторами) переменные, а предложение — нет. Поэтому формула

$$\forall x \exists y\ \text{Любит}\ (x, y)$$

является предложением, а формула

$$\forall x\ \text{Любит}\ (x, y)$$

не является. Она содержит несвободную (связанную квантором) переменную x и свободную переменную y.

Термы и атомарные формулы определяются так же, как и в клаузальной форме.

Выражение Z является *формулой,* если и только если оно есть атомарная формула или выражение вида

$[X \ \& \ Y]$
$[X \vee Y]$
$[X \rightarrow Y]$ или $[Y \leftarrow X]$
$[X \leftrightarrow Y]$
$\neg X$
$\forall vX$
$\exists vX$

где X и Y суть формулы, а v — переменная.

Любая формула Z является *подформулой* самой себя. В первых четырех вышеприведенных случаях любая подформула и у X и у Y является *подформулой* Z. В последних трех случаях каждая подформула X является *подформулой* Z.

Вхождение некоторой переменной v в формулу Z является *свободным* (или несвязанным), если оно не принадлежит ни одной подформуле Z, имеющей вид $\forall vX$ или $\exists vX$. Если некоторое вхождение переменной v свободно в X, то оно *связано* в $\forall vX$ и в $\exists vX$ кванторами $\forall v$ и $\exists v$ соответственно.

Формула является *предложением,* если и только если она не содержит никаких свободных вхождений переменных.

Наше определение допускает предложения и такого типа:

$$\exists x\ [\text{Устрица}\ (x)\ \&\ \exists x\ \text{Вкусная}\ (x)],$$

в которых одна и та же переменная x связана разными вхождениями квантора. Такие предложения вызывают нежелательные сложности, с

которыми лучше не связываться. Следовательно, нам придется ограничить определение формулы Z так, чтобы оно удовлетворяло такому условию: для каждой переменной v, встречающейся в Z, либо все вхождения v в Z свободны в Z, либо все вхождения v в Z связаны одним и тем же вхождением квантора.

Любая формула Z, нарушающая это ограничение, может быть преобразована за счет переименования переменных к эквивалентному виду, который удовлетворяет ограничению. Это может быть сделано на основании применения следующих правил эквивалентности к подформулам Z:

$$\forall u X \leftrightarrow \forall v X'$$
$$\exists u X \leftrightarrow \exists v X'$$

где X' получается из X заменой всех вхождений u на v, а v не встречается в X.

Любая подформула может быть заменена эквивалентной без воздействия на значение той формулы, в которую эта подформула включена.

Отметьте также, что определения разрешают выполнять квантификацию $\forall v X$ и $\exists v X$ по переменной v, которая не встречается в формуле X. Такая квантификация является *пустой* в том смысле, что результирующая формула эквивалентна бескванторной исходной формуле X. Удаление пустых кванторов может быть оправдано следующими эквивалентностями:

$$\forall v X \leftrightarrow X$$
$$\exists v X \leftrightarrow X$$

где переменная v не встречается в X.

Можно использовать специальные соглашения, повышающие качество восприятия формул при чтении, благодаря уменьшению количества скобок. Самые внешние скобки могут быть всегда удалены; например, можно писать

$A \rightarrow B$ вместо $[A \rightarrow B]$.

Ассоциативностью конъюнкции может быть оправдано удаление скобок, когда несколько скобок связаны конъюнкцией вместе. Поскольку формулы

$A \,\&\, [B \,\&\, C]$

и

$[A \,\&\, B] \,\&\, C$

эквивалентны, постольку позволительно игнорировать скобки и в той и в другой формуле, записывая

$A \,\&\, B \,\&\, C$

По аналогичным соображениям ассоциативностью дизъюнкции может быть оправдано использование записи

$$A \vee B \vee C$$

вместо

$$A \vee [B \vee C]$$

или

$$[A \vee B] \vee C$$

Скобки можно удалять и в других случаях на основании правил предшествования для кванторов и пропозициональных связок.

Мы будем следовать таким соглашениям. *Символ отрицания* $\neg$ и *кванторы* $\exists$, $\forall$ задают более тесную связь, чем другие символы, а *конъюнкция* & и *дизъюнкция* $\vee$ задают более тесную связь, чем *импликация* $\rightarrow$ и *эквивалентность* $\leftrightarrow$. А посему мы можем, например, записывать без всяких опасений

$$A \vee B \vee C \leftarrow D \;\&\; E \;\&\; F$$

вместо

$$[A \vee [B \vee C]] \leftarrow [[D \;\&\; E] \;\&\; F$$

Можно опускать кванторы общности в начале предложений, записывая, например,

Дед-или-бабушка (x, y) ← Родитель (x, z) & Родитель (z, y)

вместо

$\forall x \forall y \forall z$ Дед-или-бабушка (x, y) ←
Родитель (x, z) & Родитель (z, y)

как и в клаузальной форме. Такое освобождение от кванторов общности можно выполнять только в том случае, когда из контекста ясно, что выражение является предложением, а не формулой, содержащей свободные переменные.

Перевод в клаузальную форму

Любое предложение в стандартной форме может быть переведено в клаузальную форму. Результирующее множество клауз является совместным, если и только если предложение в стандартной форме является совместным. Такое приведение к клаузальной форме можно использовать для того, чтобы продемонстрировать несовместность множества предложений в стандартной форме:

> Множество предложений в стандартной форме несовместно, если и только если соответствующее множество клауз было несовместным.

Правила перевода в клаузальную форму можно выразить проще, если вначале импликации и эквивалентности выразить в терминах отрицания, конъюнкции и дизъюнкции на основании следующих правил эквивалентности:

$$[X \rightarrow Y] \leftrightarrow \neg X \vee Y$$
$$[X \leftrightarrow Y] \leftrightarrow [X \rightarrow Y] \;\&\; [Y \rightarrow X]$$

т.е.

$$[X \leftrightarrow Y] \leftrightarrow [\neg X \vee Y] \;\&\; [\neg Y \vee X]$$

где X и Y являются произвольными формулами.

Когда импликации и эквивалентности преобразованы к новой форме, оставшаяся часть приведения состоит из

(1) передвижения отрицаний внутрь предложений за конъюнкции, дизъюнкции и кванторы до тех пор, пока они не окажутся непосредственно перед атомарными формулами;

(2) передвижения дизъюнкций внутрь предложений за конъюнкции и кванторы до тех пор, пока они не станут связывать лишь атомы и атомы с отрицаниями;

(3) устранения кванторов существования;

(4) перезаписи дизъюнкции атомов и их отрицаний

$$A_1 \vee \ldots \vee A_m \vee \neg B_1 \vee \ldots \vee \neg B_n$$

в виде клауз

$$A_1, \ldots, A_m \leftarrow B_1, \ldots, B_n$$

Отрицания могут быть поставлены непосредственно перед атомами путем многократного применения таких правил эквивалентности:

$$\neg[X \;\&\; Y] \leftrightarrow \neg X \vee \neg Y$$
$$\neg[X \vee Y] \leftrightarrow \neg X \;\&\; \neg Y$$
$$\neg \exists v X \leftrightarrow \forall v \neg X$$
$$\neg \forall v X \leftrightarrow \exists v \neg X$$
$$\neg\neg X \leftrightarrow X$$

где X и Y являются формулами, а v – произвольная переменная

Коммутативность дизъюнкции

$$X \vee Y \leftrightarrow Y \vee X$$

нужна для установления аналогичных эквивалентностей

$$[Y \,\&\, Z] \vee X \leftrightarrow [Y \vee X] \,\&\, [Z \vee X]$$
$$\exists v Y \vee X \leftrightarrow \exists v[Y \vee X]$$
$$\forall v Y \vee X \leftrightarrow \forall v[Y \vee X]$$

где v не встречается в X.

Приведенные эквивалентности существенны для преобразования любого бесквантороного предложения в стандартной форме в эквивалентное предложение в клаузальной форме. Исключение квантора существования, однако, приводит к предложениям, которые не являются эквивалентностями. Оно вводит некоторую константу или функциональный символ для того, чтобы сообщить имя индивиду, на который в исходном предложении имелась лишь неявная ссылка. Из нового предложения следует исходное, но не наоборот. Тем не менее, исключение квантора существования не оказывает влияния на совместность множества предложений.

Если дана конъюнкция (или множество) предложений S, то для того, чтобы исключить кванторы существования из S, необходимо исключить их из *предложений* вида

$$\forall v_1 \forall v_2 \ldots \forall v_n \exists u X$$

принадлежащих S. Такое предложение может быть заменено новым:

$$\forall v_1 \forall v_2 \ldots \forall v_n X',$$

где X' получено из X заменой всех свободных вхождений u в X на терм $f(v_1, \ldots, v_n)$, где f есть некоторый функциональный символ, не встречающийся в S.

Если $n = 0$, то терм $f(v_1, \ldots, v_n)$ сводится к константному символу*). Отметьте, что замена не является эквивалентной и применима только к предложениям, а не к формулам. Новая конъюнкция (или множество предложений) является совместным (или несовместным), если и только если S является таковым.

Для того, чтобы преобразовать предложения, входящие в S, к корректному виду, полезно передвинуть кванторы общности внутрь конъюнкций.

$$\forall v[X \,\&\, Y] \leftrightarrow \forall v X \,\&\, \forall v Y$$

*) $f(v_1, \ldots, v_n)$ называется функцией Сколема – *Примеч. Д.А. Поспелова*

Повторное применение приведенных правил приведет к преобразованию любой конъюнкции (или множества) предложений, находящихся в стандартной форме, к конъюнкции (или ко множеству) предложений, каждое из которых имеет вид

$$\forall v_1 \ldots \forall v_k [A_1 \vee \ldots \vee A_m \vee \neg B_1 \vee \ldots \vee \neg B_n],$$

что эквивалентно клаузе

$$A_1, \ldots, A_m \leftarrow B_1, \ldots, B_n$$

Приведенные правила составляют логическую часть семейства алгоритмов для приведения стандартной формы к клаузальной. Все случаи недетерминизма первого рода при этом относятся к категории "не забочусь". Эффективный алгоритм получается за счет постоянного применения правил к самым внешним пропозициональным связкам или кванторам и замены формулы с левой стороны эквивалентности формулой с правой стороны. Более того, на практике более удобно оставлять знак импликации нетронутым и применять производные эквивалентности. Нижеследующие производные эквивалентности (см. упражнение 2)) являются наиболее полезными.

$$[X \rightarrow Y \,\&\, Z] \leftrightarrow [X \rightarrow Y] \,\&\, [X \rightarrow Z]$$
$$[X \vee Y \rightarrow Z] \leftrightarrow [X \rightarrow Z] \,\&\, [Y \rightarrow Z]$$
$$[X \,\&\, \neg Y \rightarrow Z] \leftrightarrow [X \rightarrow Y \vee Z]$$
$$[X \rightarrow \neg Y \vee Z] \leftrightarrow [X \,\&\, Y \rightarrow Z]$$
$$[X \rightarrow [Y \rightarrow Z]] \leftrightarrow [X \,\&\, Y \rightarrow Z]$$
$$[[X \rightarrow Y] \rightarrow Z] \leftrightarrow [X \vee Z] \,\&\, [Y \rightarrow Z]$$
$$X \rightarrow \forall v Y \leftrightarrow \forall v [X \rightarrow Y]$$
$$X \rightarrow \exists v Y \leftrightarrow \exists v [X \rightarrow Y]$$
$$\forall v Y \rightarrow X \leftrightarrow \exists v [Y \rightarrow X]$$
$$\exists v Y \rightarrow X \leftrightarrow \forall v [Y \rightarrow X]$$

где v не входит в X

Кроме того, весьма полезными оказываются обобщения эквивалентности

$$[U \,\&\, [X \vee Y] \rightarrow Z] \leftrightarrow [U \,\&\, X \rightarrow Z] \,\&\, [U \,\&\, Y \rightarrow Z]$$
$$[U \,\&\, [X \rightarrow Y] \rightarrow Z] \leftrightarrow [U \rightarrow X \vee Z] \,\&\, [U \,\&\, Y \rightarrow Z]$$

Для того, чтобы применять их, могут потребоваться сведения о коммутативности конъюнкции:

$$X \,\&\, Y \leftrightarrow Y \,\&\, X$$

Сравнение клаузальной и стандартной форм

Клаузальная форма является ограниченным подмножеством стандартной. Ее преимущество состоит в том, что простые, эффективные и достаточно естественные системы доказательства теорем, базирующиеся на резолюции, созданы на ее языке. Но стандартная форма доставляет бóльшую свободу в выражениях. Некоторые типы предложений могут быть записаны более экономно, а другие и более естественно, чем в клаузальной форме. Анализ случаев, когда стандартная форма обеспечивает бóльшую выразительную мощность, чем клаузальная, который будет проведен в нескольких следующих пунктах, показывает, что на самом деле нужна не полная, неограниченная стандартная форма, а чуть расширенная клаузальная форма. Во многих случаях достаточно позволить, чтоб неатомарные формулы могли содержаться в посылках и заключениях импликаций

$$A_1, \ldots, A_m \leftarrow B_1, \ldots, B_n.$$

В частности, полезно разрешить, чтобы заключения A_i были бы конъюнкциями атомов, а посылки B_j были бы импликациями. Отметим к тому же, что использование эквивалентностей в определениях выглядит более полезным, чем запись по отдельности их половинок.

Идеальная логическая система должна сочетать в себе достоинства клаузальной формы с достоинствами стандартной формы логики. Для того, чтобы достичь этого, такая система должна, с одной стороны, приводиться к резолюции для предложений в клаузальной форме, а с другой стороны – походить на системы натуральной дедукции Бледшоу [Bledsoe 1971], Брауна [Brown 1977], Бибела и Шрайбера [Bibel, Schreiber 1975] и Невинса [Nevins 1974]. Такие системы должны получаться в результате сочетания правила резолюций с правилами перевода предложений из стандартной в клаузальную форму.

Удовлетворительное решение задачи вывода программ на языке клауз Хорна из программных спецификаций в стандартной форме тоже требует такой процедуры доказательства*). Эта проблема была поставлена Бибелом [Bibel 1976a, 1976b, 1978], Кларком и Сикелем [Clark, Sickel 1977] и Хоггером [Hogger 1978a, 1978b, 1979]. Их правила вывода похожи и на правила перехода к клаузальной форме, и на правила резолюции, работающие по принципу процедурных вызовов. Процедуры доказательства для стандартной формы логики, обладающие некоторыми из перечисленных важных свойств, создавались Мюррэем [Murray 1978] и Манной вместе с Уолдингером [Manna, Waldinger 1978].

В следующих пунктах мы рассмотрим несколько примеров, иллюстрирующих ограничения клаузальной формы и неадекватность замены действий в стандартной форме простым приведением к клаузальной форме и применением правила резолюций. В конце главы мы рассмотрим задачу вывода программ на языке клауз Хорна из неклаузальных спецификаций.

---

*) То есть, сочетания правила резолюций с переводом в клаузальную форму. – *Примеч. пер.*

Конъюнктивные заключения и дизъюнктивные посылки

Стандартная форма оказыается более экономной, чем клаузальная форма, когда из одних и тех же посылок следуют различные заключения или когда одни и те же заключения следуют из альтернативных посылок

Пример. Каждый совершает ошибки.

Стандартная форма

$$\forall x \exists y\ [\text{Человек}\ (x) \leftarrow \text{Совершает}\ (x, y)\ \&\ \text{Ошибка}\ (y)]$$

Перевод

(а) Человек (x) → Совершает (x, o (x)) & Ошибка (o (x))
(б) ¬Человек (x) ∨ [Совершает (x, o (x)) & Ошибка (o (x))]
(в) [¬ Человек (x) ∨ Совершает (x, o (x))] &
[¬Человек (x) ∨ Ошибка (o (x))]

Клаузальная форма

(г) Совершает (x, o (x)) ← Человек (x)
Ошибка (o (x)) ← Человек (x)

В клаузальной форме одну и ту же посылку Человек (x) приходится повторять для обоих отдельных заключений. Отметьте, что, если использовать правила преобразования для импликации, то приведение от (а) к (г) может быть выполнено за один шаг.

Пример. Некое лицо является предком другого, если первое лицо является родителем второго или первое лицо является предком предка второго лица.

Стандартная форма

$$\text{Пред}\ (x, y) \leftarrow \text{Род}\ (x, y) \vee \exists z\ [\text{Пред}\ (x, z)\ \&\ \text{Пред}\ (z, y)]$$

Перевод

(а) Пред (x, y) ∨ ¬[Род (x, y) ∨∃z [Пред (x, z) & Пред (z, y)]]
(б) Пред (x, y) ∨ [¬Род (x, y) & ¬∃z [Пред (x, z) & Пред (z, y)]]
(в) [Пред (x, y) ∨ ¬Род (x, y)] &
[Пред (x, y) ∨ ¬∃z [Пред (x, z) & Пред (z, y)]]
(г) [Пред (x, y) ∨ ¬Род (x, y)] &
[Пред (x, y) ∨ ∀z [¬Пред (x, z) ∨ ¬Пред (z, y)]]
(д) [Пред (x, y) ∨ ¬Род (x, y)] &
∀z [Пред (x, y) ∨ ¬Пред (x, z) ∨ ¬Пред (z, y)]

Клаузальная форма

(е) Пред (x, y) ← Род (x, y)
Пред (x, y) ← Пред (x, z), Пред (z, y)

В клаузальной форме одно и то же заключение приходится повторять для обеих альтернативных посылок. Преобразование из стандартной формы упрощается, если используются производные эквивалентности:

(а′) [Пред (x, y) ← Род (x, y)] &
[Пред (x, y) ← ∃z [Пред (x, z) & Пред (z, y)]]

(б′) [Пред (x, y) ← Род (x, y)] &
∀z [Пред (x, y) ← Пред (x, z) & Пред (z, y)]
(в′) Пред (x, y) ← Род (x, y)
Пред (x, y) ← Пред (x, z), Пред (z, y)

В целях упрощения в оставшейся части этой главы мы будем использовать производные эквивалентности.

Дизъюнктивные заключения

Стандартная форма оказывается более экономной и понятной в случаях, когда альтернативами в заключении являются конъюнкции.

Пример. Земля кругла и конечна или плоска и бесконечна.

Стандартная форма

[Кругла (3) & Конечна (3)] ∨ [Плоска (3) & Бесконечна (3)]

Перевод

(а) [[Кругла (3) & Конечна (3)] ∨ Плоска (3)] &
[[Кругла (3) & Конечна (3)] ∨ Бесконечна (3)]
(б) [Кругла (3) ∨ Плоска (3)] &
[Конечна (3) ∨ Плоска (3)] &
[Кругла (3) ∨ Бесконечна (3)] &
[Конечна (3) ∨ Бесконечна (3)]

Клаузальная форма

Кругла (3), Плоска (3) ←
Конечна (3), Плоска (3) ←
Кругла (3), Бесконечна (3) ←
Конечна (3), Бесконечна (3) ←

Только-если части определений

В следующей главе мы приведем доводы в пользу того, что клаузы Хорна зачастую выражают лишь если-части определений вида если-и-только-если. Полное определение если-и-только-если может быть выражено более компактно в стандартной форме, если использовать знак эквивалентности ↔. В клаузальной форме если-части и только-если части приходится задавать по отдельности. Только-если часть в общем случае выражает альтернативные заключения и может оказаться и неэкономной и неестественной.

Пример. Только-если часть если-и-только-если определения предка.

Стандартная форма

Пред (x, y) → Род (x, y) ∨ ∃z [Пред (x, z) & Пред (z, y)]

Перевод

(а) ∃z [Пред (x, y) → Род (x, y) ∨ [Пред (x, z) & Пред (z, y)]]
(б) Пред (x, y) → Род (x, y) ∨
[Пред (x, f (x, y)) & Пред (f (x, y), y)]
(в) Пред (x, y) → [Род (x, y) ∨ Пред (x, f (x, y))] &
[Род (x, y) ∨ Пред (f (x, y), y)]

Клаузальная форма

Род (x, y), Пред (x, f (x, y)) ← Пред (x, y)
Род (x, y), Пред (f (x, y), y) ← Пред (x, y)

Импликации как посылки импликаций

Довольно часто в предложениях естественного языка посылки сами по себе являются не простыми атомами, а импликациями. Такие предложения могут непосредственно и естественно выражаться в стандартной форме, но понимание их в клаузальной форме затруднено.

Пример. x ⊃ y истинно, если y истинно всякий раз, когда истинно x.
Стандартная форма

Истинно (x ⊃ y) ← [Истинно (y) ← Истинно (x)]

Клаузальная форма

Истинно (x ⊃ y), Истинно (x) ←
Истинно (x ⊃ y), ← Истинно (y)

Пример. Боб счастлив, если всем его студентам нравится логика.
Стандартная форма

Счастлив (Боб) ← ∀x [Студент (Боб, x) → Нравится (x, логика)]

Перевод

(а) ∃x [Счастлив (Боб) ← [Студент (Боб, x) → Нравится (x, логика)]]
(б) Счастлив (Боб) ← [Студент (Боб, ☺) → Нравится (☺, логика)]

Клаузальная форма

Счастлив (Боб), Студент (Боб, ☺) ←
Счастлив (Боб) ← Нравится (☺, логика)

Пример. Поставщик заслуживает предпочтения, если все поставляемые им детали прибывают вовремя.
Стандартная форма

Заслуживает-предпочтения (x) ← Поставщик (x) &
∀ u [Поставляет (x, u) → Прибывает-во-время (u)]

Клаузальная форма

Заслуживает-предпочтения (x) ← Поставщик (x),
Прибывает-во-время (p(x))
Заслуживает-предпочтения (x), Поставляет (x, p(x)) ←
Поставщик (x)

Пример. Множество является вполне упорядоченным, если и только если каждое непустое его подмножество обладает наименьшим элементом. Множество непусто, если и только если оно обладает, по меньшей мере, одним элементом. Элемент множества является наименьшим элементом, если и только если он меньше или равен любого элемента этого множества.
Стандартная форма

Вполне-упорядочено (x) ↔ ∀z [Облнаимэлтом (z) ←
z ⊆ x & Непусто (z)]
Непусто (z) ↔ ∃u u ∈ z
Облнаимэлтом (z) ↔ ∃u [u ∈ z & ∀v [v ∈ z → u ⩽ v]]

Клаузальная форма

Вполне-упорядочено (x), произв (x) ⊂ x ←
Вполне-упорядочено (x), Непусто (произв (x)) ←
Вполне-упорядочено (x) ← Облнаимэлтом (произв (x))
Облнаимэлтом (z) ← Вполне-упорядочено (x), z ⊆ x, Непусто (z)
Непусто (z) ← u ∈ z
выбран (z) ∈ z ← Непусто (z)
Облнаимэлтом (z), элт (z, u) ∈ z ← u ∈ z
Облнаимэлтом (z) ← u ⩽ элт (z, u), u ∈ z
наименьший (z) ∈ z ← Облнаимэлтом (z)
наименьший (z) ⩽ u ← Облнаимэлтом (z), u ∈ z

Вывод (извлечение) программ из спецификаций

Программы можно выражать средствами логики более естественно, если допускать присутствие импликаций в посылках. Определение подмножества может служить достаточно простым примером этого

x ⊆ y ← ∀z [z ∈ x → z ∈ y]

Условие того, что "каждый элемент x является также элементом y", нейтрально относительно способа, которым задаются элементы x и которым доказывается, что они являются элементами y. В частности, оно не противоречит возможности того, что все элементы x задаются одновременно, параллельно. Такая высокоуровневая спецификация невозможна в обычных языках программирования. Но она столь же затруднительна и на языке клауз Хорна.

Предположим, что множества представляются конечными списками. Тогда и понятие быть элементом множества и понятие быть подмножеством могут быть заданы рекурсивно при помощи клауз Хорна:

z ∈ z . v ←
z ∈ u . v ← z ∈ v
nil ⊆ y ←
u . v ⊆ y ← u ∈ y, v ⊆ y

Программа на языке клауз Хорна менее естественна и более приближена к уровню компьютера, чем спецификация в стандартной форме. Она выражает такие детали, которые должны оставаться на усмотрении системы поиска доказательств теорем при использовании стандартной формы. Более того, она и работает лишь для конечных множеств, представленных посредством списков. Спецификация в стандартной форме, с другой стороны, работает и с конечными и с бесконечными списками. Упражнение 6б) демонстрирует это для случая упорядоченного списка.

Логику принято использовать скорее как язык спецификаций, а не как язык программирования. Методы верификации обычных программ,

основывающиеся на логических спецификациях, поэтому должны быть дополнены средствами связи двух различных языков. Подобные методы, развитые Флойдом [Floyd 1967], Манной [Manna 1969], Хоаром [Hoare 1969] и Дейкстрой [Dijkstra 1976], выражают спецификации посредством логики и связывают их с программами путем определения семантики программ методами логики.

Верификация программ значительно облегчается, когда программы и спецификации выражаются на одном языке. Это подтверждается результатом Бойера и Мура [Boyer, Moor 1975], которые использовали Лисп и для программ и для спецификаций, результатами Манны и Уолдингера [Manna, Waldinger 1977], которые использовали Лисп для программ и Лисп, дополненный импликациями, замкнутыми кванторами общности, для спецификаций и результатами Берстэлла и Дарлингтона [Burstall, Darlington 1977], которые применяли рекурсивные уравнения и для программ и для спецификаций. В более недавнее время за счет использования процедурной интерпретации клауз Хорна Кларк и Тарнлунд [Clark, Tarnlund 1977], Бибел [Bibel 1976a, 1976b, 1978], Кларк и Сикель [Clark, Sickel 1977], Хоггер [Hogger 1978a,, 1978b, 1979] и Кларк и Дарлингтон [Clark, Darlington 1978] разработали дедуктивные стратегии извлечения логических программ из логических спецификаций. Вдобавок, Манна и Уолдингер [Manna, Waldinger 1978] разработали специальное расширение резолюции для извлечения Лисп-программы из логических спецификаций.

Для извлечения логических программ из логических спецификаций характерно, что дедукция применяется и для использования программ, и для извлечения программ из спецификаций. Программы можно рассматривать как вычислительно значимые логические следствия спецификаций.

Проиллюстрируем общий метод извлечения программ на языке клауз Хорна для задания подмножества из спецификаций, находящихся в стандартной форме логики. Шагами вывода будут последовательные действия по сочетанию резолюции с приведением к клаузальной форме. Начнем с если-и-только-если спецификаций отношений "быть подмножеством" и "быть элементом".

S1 $x \subseteq y \leftrightarrow \forall z\,[z \in x \rightarrow z \in y]$
S2 $\forall z \neg [z \in nil]$ (т.е. $\leftarrow z \in nil$)
S3 $z \in u \,.\, v \leftrightarrow z = u \vee z \in v$

Базис рекурсивной программы на языке клауз Хорна

$$nil \subseteq y \leftarrow$$

может быть получен непосредственно путем построения резольвенты клаузальной формы для S2 с первой из двух клауз

$x \subseteq y$, произв $(x, y) \in x \leftarrow$
$x \subseteq y \leftarrow$ произв $(x, y) \in y$

полученных от превращения S1 в клаузальную форму.

Рекурсивная клауза программы может быть получена более естественно путем обработки спецификаций, выраженных в стандартной форме. Согласовывая подчеркнутые атомы в S1 и в S3, получаем

S4 $\quad u \,.\, v \subseteq y \leftarrow \forall z\,[[z = u \vee z \in v] \rightarrow z \in y]$

В данном случае достаточно использовать лишь если-часть определения подмножества. Можно считать, что S4 получается назначением переменной x значения u . v в S1, а затем использованием эквивалентности S3 для замены $z \in u \,.\, v$ на $z = u \vee z \in v$. Далее приступим к приведению S4 к клаузальной форме

S5 $\quad u \,.\, v \subseteq y \leftarrow \forall z\,[z = u \rightarrow z \in y]\ \&$
$\qquad\qquad\qquad\qquad\quad \forall z\,[z \in v \rightarrow z \in y]$

Любое дальнейшее преобразование приводит к нехорновским клаузам. К счастью, две неатомарные посылки в S5 можно заменить двумя эквивалентными атомарными

S6 $\quad \forall z\,[z = u \rightarrow z \in y] \leftrightarrow u \in y$
S7 $\quad \forall z\,[z \in v \rightarrow z \in y] \leftrightarrow v \subseteq y$

Применяя эти две эквивалентности к S5, получим оставшуюся часть программы

$$u \,.\, v \subseteq y \leftarrow u \in y,\ v \subseteq y$$

Теперь остается продемонстрировать эквивалентность S6 и S7. Что касается S7, то это – просто пример S1. Первая эквивалентность является специальным случаем более общей эквивалентности

$$\forall z\,[z = u \rightarrow X] \leftrightarrow X'.$$

где X' получено из X заменой всех вхождений z на u

которая и вообще является полезной.

Извлечение программы для задания подмножества иллюстрирует использование правил вывода, непосредственно применяемых к стандартной форме и имеющих общие черты как с резолюцией, так и с правилами перевода из стандартной формы в клаузальную форму.

Упражнения

1. Выразите следующие предложения в стандартной форме и преобразуйте их в клаузальную форму

а) Число максимально во множестве чисел, если оно принадлежит этому множеству и является большим либо равным ($\geqslant$) всех чисел, принадлежащих этому множеству

Подсказка: определите вспомогательное отношение Доминирует (x, y), которое истинно, когда x $\geqslant$ всех чисел, принадлежащих множеству чисел y.

б) Список чисел упорядочен, либо если он пуст, либо если его первый элемент $\leqslant$ всех чисел в остатке списка, а остаток списка упорядочен.

в) Число является наибольшим общим делителем чисел x и y, если оно делит x и y и $\geqslant$ всех чисел, которые делят x и y.

2. Производные эквивалентности на стр. 214 могут быть установлены путем приведения каждой половины эквивалентности к одной и той же формуле за счет замены подформул эквивалентными подформулами. Например, обе половины эквивалентности

$$X \rightarrow [Y \;\&\; Z] \leftrightarrow [X \rightarrow Y] \;\&\; [X \rightarrow Z]$$

преобразуются к одной и той же формуле

$$[\neg X \vee Y] \;\&\; [\neg X \vee Z].$$

Выведите оставшиеся эквивалентности на стр. 214.

3. а) Выразите следующие предположения в стандартной форме и приведите их к клаузальной форме:

Дракон счастлив, если все его дети могут летать.

Зеленый дракон может летать.

Дракон зеленый, если, по крайней мере, один из его родителей зеленый, а иначе он ярко-розовый.

б) Примените резолюцию (и, если надо, факторизацию) для того, чтобы показать, что

(i) Зеленые драконы счастливы.

(ii) Бездетные драконы счастливы.

(Здесь вам могут понадобиться некоторые очевидные пропущенные посылки).

(iii) Что делать ярко-розовому дракону для того, чтобы быть счастливым?

4. Это упражнение служит расширением упражнения 8 из гл. 2. Если заданы данные в таблицах "Поставщик", "Деталь", "Поставка", то выразить следующие запросы в стандартной форме, используя при этом и бинарные и n-арные представления

а) Каковы номера поставщиков, поставляющих все детали?

б) Каковы имена поставщиков, не поставляющих книги?

в) Каковы номера таких поставщиков, которые поставляют, по меньшей мере, все те детали, которые поставляет Джон?

5. а) Выразите следующее предположение в стандартной форме и преобразуйте его в клаузальную форму:

Логик счастлив, если все его доводы весомы.

б) Примените резолюцию для того, чтобы показать, что приведенные ниже заключения следуют из этого предположения:

(i) Логик счастлив, если доводы всех весомы.

(ii) Логик счастлив, если у него нет доводов.

6. а) Выразите нижеприведенные предположения в стандартной форме и преобразуйте их в клаузальную форму:

(i) Последовательность z упорядочена, если для каждых x, y, i и j верно, что x является i-м элементом z, y является j-м элементом z, и из $i \leqslant j$ следует, что $x \leqslant y$.

(ii) Если $i \leqslant j$, то $u * i \leqslant u * j$ для всех i, j и u.

(iii) i-й элемент последовательности s равен $3 * i$ для любого i.

б) Примените резолюцию для того, чтобы показать, что s упорядочена. Учтите, что s может быть бесконечной.

7. Предположим, что следующие отношения уже определены:

$x \leqslant y$

$x > y$

Пустое (x)

т.е. дерево x не содержит вершин;

Расщепляется (x, y, u, v)

т.е. дерево x имеет корневую вершину, помеченную меткой y, левое поддерево u и правое поддерево v (рис. 10.1).

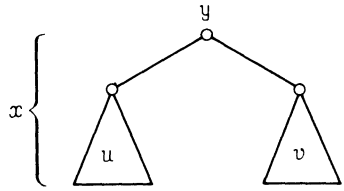


Рис. 10.1

а) Выразите следующее определение отношения Упоряд (x) в стандартной форме: Дерево x упорядочено, если для каждого его непустого поддерева z верно, что все метки, принадлежащие левому поддереву z $\leqslant$ $\leqslant$ метки, помечающей корень z; все метки, принадлежащие правому поддереву z $>$ метки, помечающей корень z. Для этого определите отношения:

Поддерево (z, x)

т.е. z является поддеревом x;

Принадлежит (y, x)

т.е. метка принадлежит дереву x.

б) Переведите это определение Упоряд (x) в клаузальную форму.

8. Отношение Подспис (x, y), утверждающее, что x является подписком y можно задать так

$$\text{Подспис}(x, y) \leftrightarrow \exists u \exists v \exists w\ [\text{Присоединить}(u, x, v) \,\&\, \text{Присоединить}(v, w, y)]$$

$$\text{Присоединить}(x, y, z) \leftrightarrow [x = \text{nil} \,\&\, y = z] \vee \exists u \exists v \exists w\ [x = y \,.\, v \,\&\, z = u \,.\, w \,\&\, \text{Присоединить}(v, y, w)]$$

Извлеките рекурсивную программу для Подспис (x, y), не включающую Подспис, используя предположения об отношении равенства

$x \,.\, y = u \,.\, v \leftrightarrow x = u \,\&\, y = v$

$\neg \exists u \exists v\ u \,.\, v = \text{nil}$

$x = x$

9. Отношение Факт* (x, y, u, v) можно задать так

$$\text{Факт}^*(x, y, u, v) \leftrightarrow [\text{Факт}(x, y) \rightarrow \text{Факт}(u, v)]$$

$$\text{Факт}(x, y) \leftrightarrow [\text{Нуль}(x) \,\&\, \text{Посл}(x, y)] \vee \exists u \exists v\ [\text{Посл}(u, x) \,\&\, \text{Факт}(u, v) \,\&\, \text{Умножить}(x, v, y)]$$

Нуль (0) ←

Посл (x, s (x)) ←

а) Извлеките рекурсивную программу для

Факт* (x, y, u, v), не используя Факт.

б) Покажите, что Факт (u, v) ↔ Факт* (0, s(0)), u, v).

10. Пусть дана спецификация

Упоряд (x) ↔ ∀u∀v [Последователи (u, v, x) → u ⩽ v]

Извлеките программу на языке клауз Хорна, используя такие посылки

¬Последователи (u, v, nil)

¬Последователи (u, v, x . nil)

Последователи (u, v, x . y) ↔ Последователи (u, v, y)

∨ ∃z [u = x & y = v . z]

В классической логике определения формулируются при помощи выражения если–и–только–если. Например,

G* Дед–или–бабушка (x, y) $\leftrightarrow \exists$ z [Родитель (x, z) & Родитель (z, y)]

Но в программах и базах данных, использующих язык клауз Хорна, присутствует лишь если – часть таких если–и–только–если определений:

G Дед–или–бабушка (x, y) $\leftarrow$ Родитель (x, z), Родитель (z, y)

Мы сумели избавиться от полной если–и–только–если формы определений благодаря тому, что использования одних лишь если–частей достаточно для вывода всех положительных примеров отношений. Все свободные от переменных утверждения вида

Дед–или–бабушка (s, t) $\leftarrow$,

которые следуют из G*, также следуют и из G. И, например, никак не удастся сосчитать больше факториалов с помощью если–и–только–если определения

F* Факт (x, y) $\leftrightarrow$ [x = 0 & y = 1] $\vee$ $\exists$ x′ $\exists$ y′ [x = x′ + 1 & Факт (x′, y′) & y = x * y′],

чем с помощью одних если–частей:

F1 Факт (x, y) $\leftarrow$ x = 0, y = 1

F2 Факт (x, y) $\leftarrow$ x = x′ +1, Факт (x′, y), y = x * y′

Но, как мы увидим в следующем разделе, полная если–и–только–если форма определений нужна для доказательства свойств программ. Она также необходима в базах данных для ответа на запросы, включающие кванторы общности и отрицание.

При неформальном использовании естественного языка если-форма определений часто применяется тогда, когда на самом деле подразумевается если–и–только–если форма. Это вызывает к жизни проблему различения случаев, когда пропущенная только–если часть подразумевается, а когда – нет.

Мы убедимся в том, что эта проблема усложняется из-за того, что только–если части определений бывают двусмысленными.

A только если B

можно понимать в объективном языке как

$$B \leftarrow A$$

а в мета–языке – как то, что $A \leftarrow B$ выражает единственное условие, при котором A истинно. Следовательно, доказательства, в которых имеется необходимость использования только–если частей, могут проводиться с учетом этого по-разному в объектном языке и в мета–языке. Но несмотря на это различие, имеется замечательное сходство структур доказательства в обоих случаях.

Необходимость только–если частей определений

Только–если части определений необходимы для доказательства того, что программа обладает некоторыми свойствами, а также для проверки выполнения ограничений целостности в базах данных. Рассмотрим, к примеру, программу на языке клауз Хорна для вычисления факториала F1 . . 2. Несомненно, ее свойством является то, что единственным значением факториала 0 является 1, т.е.

$$y = 1 \leftarrow \text{Факт}\ (0, y)$$

Для доказательства того, что это свойство присуще программе, требуется только–если часть определения факториала, да еще такое свойство равенства:

$$\leftarrow 0 = u + 1$$

Только–если части определений нужны также для ответа на запросы к логическим базам данных. Рассмотрим, например, если–и–только–если определения отношений Преподает и Профессор:

$$\begin{aligned}\text{Преподает}\ (x, y) \leftrightarrow\ & [x = A\ \&\ y = 104]\ \vee \\ & [x = A\ \&\ y = 301]\ \vee \\ & [x = B\ \&\ y = 221]\ \vee \\ & [x = C\ \&\ y = 105]\ \vee \\ & [x = C\ \&\ y = 201]\end{aligned}$$

$$\text{Профессор}\ (x) \leftrightarrow x = A \vee x = B$$

Если, к тому же, даны клаузы

Является (104, программирование) ←
Является (221, программирование) ←

то на запрос:

Правда ли, что все профессора преподают программирование?

$$\forall x\, \exists y\, [\text{Профессор}\ (x) \rightarrow \text{Преподает}\ (x, y)\ \&$$
$$\text{Является}\ (y, \text{программирование})]\ ?$$

можно ответить положительно. Но для ответа на этот запрос понадобилась только–если часть определения отношения Профессор. Вычисление ответа на этот запрос на объектном языке и мета–языке приведены и соотнесены между собой в последующих частях этой главы.

Термы versus отношения при использовании в качестве структур данных

Связь между если–и–только–если определениями и их если–частями имеет отношение к связи между использованием термов и отношений в качестве структур данных в логических программах. Использование термов в программах на языке клауз Хорна представляет возможность пользоваться только частью того потенциала, которым обладает применение отношений, заданных в терминах если–и–только–если.

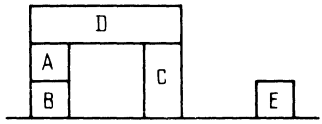


Рис. 11.1

Рассмотрим, например, данные, описываемые сценой на рис. 11.1. Если ограничиться использованием клауз Хорна, то отношения На и Свободно должны быть заданы независимо:

На (A, B) ← Свободно (D) ←
На (D, A) ← Свободно (E) ←
На (D, C) ←

Связь между обоими отношениями может быть задана только при помощи ограничения целостности:

← На (x, y), Свободно (y)

Используя определения в стиле если–и–только–если, можно, оказывается, определить отношение Свободно в терминах отношения На:

Свободно (y) ↔ ¬ ∃ x На (x, y)
На (x, y) ↔ [x = A & y = B] ∨
[x = D & y = A] ∨
[x = D & y = C]

Отметьте, что в этой формулировке свободно все, за исключением A, B и C. Отношение Свободно можно сузить, если нужно, добавив экстраусловия к его определению .

Свободно (y) ↔ Блок (y) & ¬ ∃ x На (x, y)

с учетом подходящего определения предиката Блок.

Если–и–только–если определения обычно нельзя выразить в терминах клауз Хорна. Но часть потенциала, которым обладают если–и–только–если определения, может быть привнесена в язык клауз Хорна за счет использования термов вместо отношений и структур данных. Если данные, связанные с позициями объектов на сцене, собрать в единый терм, то отношение Свободно можно задать в терминах понятий сцены. Отметим, что здесь "На" является предикатным символом, а "на" – функциональным символом.

Сцена (на (A, B) . на (D, A) . на (D, C) . nil) ←
На (x, y) ← Сцена (z), Элемент (на (x, y), z, T)
Свободно (y) ← Сцена (z), Элемент (на (x, y), z, F)
Элемент (x, x . y, T) ←
Элемент (x, nil, F) ←
Элемент (x, u . v, w) ← Разл (x, u), Элемент (x, v, w)

Представление данных в виде термов значительно менее естественно, чем их реляционное представление. Но и если–и–только–если определение, и его имитация с помощью термов имеют все же некоторые преимущества перед простыми если–частями определений на языке клауз Хорна. Многие свойства сцены, например, такие как количество объектов, на ней содержащихся, могут быть заданы и в терминах если–и–только–если определений и с использованием термов, но не могут быть определены при использовании лишь если–частей определений. Более того, из любого изменения в позиции объектов (или за счет изменения если–и–только–если отношения На, или за счет изменения утверждений, описывающих сцену) автоматически следует соответствующая модификация отношения Свободно. Но если отношения заданы независимо, то изменение сцены сопряжено с большими трудностями. В этом случае отношение На и отношение Свободно должны изменяться явно, а новая связь между ними должна все время проходить контроль соблюдения ограничений целостности.

Неформулируемые только–если посылки

Задание лишь если–частей определений является обычным делом в естественных языках даже тогда, когда подразумевается если–и–только–если определение. Даже логики, обычно всегда настаивающие на явном указании всех посылок, допускают неформулируемые только–если посылки в случае рекурсивных определений. Для логика обычным является задание лишь если–части определения натурального числа, например,

N1 0 является натуральным числом
N2 Если x является натуральным числом, то x + 1 является натуральным числом

даже если он подразумевает присутствие и только–если части:

N3 Число является натуральным, только если оно определено предложениями N1 . . 2

Естественные языки вообще обостряют ситуацию с неформулируемыми только–если посылками до крайней степени. Пожалуй, подходящей иллюстрацией для этого может служить классический логический софизм. Предположим, что верно

M1 Смертен (x) ← Человек (x)

Если сделать утверждение о том, что

M2 Смертен (Боб) ←

то может возникнуть искушение прийти к заключению, что

M3 Человек (Боб) ←

Но M3, будучи, вообще говоря, истинным суждением, не является логическим следствием явно заданных предложений. Конечно, этот софизм сразу перестал бы быть софизмом, если бы можно было апеллировать к несформулированной посылке – если бы мы могли допустить, что вместо одной лишь явно высказанной если–части определения подразумевалось полное если–и–только–если определение.

M* Смертен (x) ↔ Человек (x)

Сравнивая эти два примера, т.е. если определение числа и неполную характеризацию смертности M1, мы оказываемся лицом к лицу с необходимостью разрешить дилемму: нужна или не нужна неформулируемая только–если посылка. Та же дилемма возникает и в области баз данных, когда задача состоит в том, чтобы различить, является ли определение данных уже замкнутым, законченным или оно еще открыто. Эта задача была сформулирована Рейтером [Reiter 1978], назвавшим допущение о том, что база данных содержит всю известную информацию, *предположением о замкнутости мира*, а допущение о том, что она еще не содержит всю известную информацию, *предположением об открытости мира.* Наше намерение состоит в том, чтобы соотнести предположение о замкнутости мира с предположением о том, что опущенные только–если части определения подразумеваются, а также, чтобы соотнести предположение об открытости мира с предположением, что они не нужны.

Конечно, задача различия подразумеваемой и неподразумеваемой посылок не возникает, когда все подразумеваемое высказано явно. Явное формулирование подразумеваемых идей и понятий вдобавок облегчает совмещение предположений о замкнутости и об открытости мира в одной базе данных, позволяя применять разные посылки к разным отношениям или даже к разным примерам одного и того же отношения. Мы можем решить, например, замкнуть мир примеров отношения Преподает, описывающих курсы, преподаваемые Бобом, но оставить открытым мир примеров курсов, преподаваемых Джоном.

T1 Преподает (Боб, x) ↔ (x = 304) ∨ (x = 323) ∨ (x = 1.4)
T2 Преподает (Джон, 212) ←
T3 Преподает (Джон, 1.13) ←

Такое пренебрежение естественного языка к указанию подразумеваемой или неподразумеваемой только–если посылки выглядит, по меньшей мере, любопытным. Оно, вероятно, является следствием некоторой неуклюжести синтаксиса если–и–только–если. Так, для того, чтобы замкнуть определение курсов, преподаваемых Джоном, после того как к T1 . . 3 добавлена посылка

T4 Преподает (Джон, 103) ←

необходимо либо заменить T2 . . 4 на

T* Преподает (Джон, x) ↔ (x = 212) ∨ (x = 1.13) ∨ (x = 103)

или добавить к Т2 . . Т4 явно выраженную только–если часть определения

Т5* Преподает (Джон, x) → (x = 212) ∨ (x = 1.13) ∨ (x = 103)

Более удобным синтаксисом для этого может быть такой вариант, когда Т2 . . 4 остаются без изменения, а к ним добавляется

Т5* все примеры Преподает (Джон, x) заданы при помощи Т2 . . 4

Двусмысленность только–если частей

Наша дискуссия о связи между если–и–только–если определениями и их если–частями была упрощена за счет того, что мы временно игнорировали двусмысленность выражений типа

A только–если B.

В некоторых случаях мы можем интерпретировать его как такое предложение объектного языка

B ← A

В других случаях мы можем придать ему интерпретацию мета–языка

"A ← B" выражает единственное условие, при котором A истинно.

Только–если часть определения натурального числа, которая была предварительно задана на мета-языке, может быть задана и на объектном языке

Числ (x) → (x = 0) ∨ ∃ x′[(x = x′ + 1) & Числ (x′)]

Независимо от интерпретации выражения A только–если B на объектном языке или на мета-языке, оно может иметь сходные свойства в обоих языках. Например, в обоих случаях заключение

B ←

является следствием посылок

A только–если B
A ←

Если только–если интерпретируется на объектном языке, то заключение можно вывести за один шаг восходящего рассуждения. Если же оно интерпретируется на мета–языке, то его можно вывести из следующих рассуждений о доказательствах: если единственный способ доказательства A осуществляется посредством доказательства B и A ← может быть доказано, то B ← может быть доказано с тем же успехом.

Этот пример служит иллюстрацией весьма общего явления: такие две интерпретации только–если обосновывают аналогичные заключения в разной, но структурно сходной манере.

Решения в объектном языке и в мета-языке

Задача доказательства того, что все профессора преподают программирование

Q $\forall$ x $\exists$ y [Профессор (x) $\rightarrow$ Преподает (x, y) &
является (y, программирование)]

может быть решена независимо от того, выражена ли только–если часть определения P* отношения Профессор на объектном языке или на мета-языке.

Допустим, что только–если часть P* выражена в виде нехорновской клаузы объектного языка

x = A, x = B ← Профессор (x)

Сам запрос сводится к двум клаузам

Q1 Профессор (😐) ←
Q2 ← Преподает (😐 , y), Является (y, программирование)

Восходящее рассуждение начиная от рассуждения Q1 выводит нехорновскую клаузу

😐 = A, 😐 = B ←

Обе цели в Q2 теперь могут быть достигнуты путем разбора случаев, В случае 😐 = A первая цель в Q2 достигается при помощи

Преподает (x, y) ← (x = A), (y = 104)
x = x ←

а вторая цель при помощи

Является (104, программирование)

Во втором случае, когда 😐 = B, первая цель достигается при помощи

Преподает (x, y) ← (x = B), (y = 221)
x = x ←

а вторая цель при помощи

Является (221, программирование) ←

Допустим, с другой стороны, что только–если часть P* выражена на мета-языке:

P1 Профессор (x) ← x = A
P2 Профессор (x) ← x = B
P3 P1 и P2 выражают единственные условия, при которых индивид связан отношением Профессор.

Для решения задачи запрос Q2 нужно тоже выразить на мета-языке:

Q1* Показать, что для каждого x, обеспечивающего достижение цели
← Профессор (x)
существует y, который обеспечивает достижение целей
Q2* ← Преподает (x, y), Является (y, программирование)

Нисходящее рассуждение выводит из цели Q1* только два решения

x = A и x = B

В случае x = A обе цели в Q2* достигаются посредством y = 104 за счет использования клауз:

Преподает (x, y) ← (x = A), (y = 104)
x = x ←
Является (104, программирование) ←

В случае x = B они достигаются посредством y = 221 за счет использования клауз:

Преподает (x, y) ← (x = B), (y = 221)
x = x ←
Является (221, программирование) ←

Отметьте, что доказательства в обычном языке и в мета-языке имеют сходную структуру. В мета-языковом доказательстве, однако, равенство соотносит переменные с теми термами, с которыми оно связано в компонентах согласующей подстановки. В объективноязыковом доказательстве равенство соотносит разные имена с одним и тем же индивидом. Поэтому символ равенства, используемый для выражения только–если частей определений, удовлетворяет аксиомам E1 . . 3 гл. 2, стр. 58. В общем случае эти аксиомы крайне избыточны. Но сейчас они даже не являются необходимыми.

Объектноязыковые и мета-языковые интерпретации отрицания

Только–если части определений нужны для того, чтобы доказать, что негативное условие

← не – P

истинно. В зависимости от интерпретации только–если, доказательство можно провести либо на объектном языке, либо на мета-языке. Кларк [Clark 1978] показал, что в любом мета-языковом доказательстве не–P, полученном системой доказательства теорем на языке клауз Хорна, дополненной средством доказательства отрицания по неудаче, существует структурное сходство с объектноязыковым доказательством не–P.

Рассмотрим задачу доказательства того, что D свободно

← Свободно (D)

задавшись если–частями определений отношений На и Свободно:

On1 На (A, B) ←
On2 На (D, A) ←
On3 На (D, C) ←
On4 Свободно (y) ← не– ∃x На (x, y)

Вдобавок, для решения нужна еще только–если часть определения отношения На. А вот наличия одной если–части определения отношения Свободно хватает.

Допустим, сначала, что только–если часть определения выражена на объектном языке:

$$\text{On5} \quad \text{На}(x, y) \rightarrow [x = A \,\&\, y = B] \vee [x = D \,\&\, y = A] \vee [x = D \,\&\, y = C]$$

Это предложение выглядит более естественным в стандартной, а не в клаузальной форме. Более естественным будет также проведение доказательства в стандартной форме. К тому же, доказательство в стандартной форме

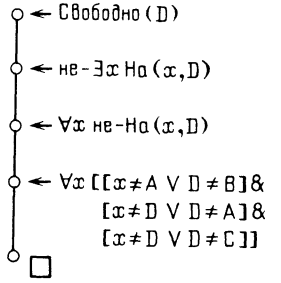


Рис. 11.2

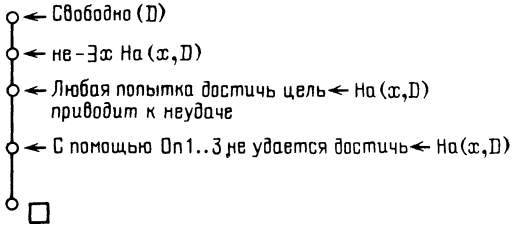


Рис. 11.3

несет в себе черты структурного сходства с мета–языком, а доказательство в клаузальной форме не несет. Для удобства переформулируем только–если часть определения по-другому, сохраняя эквивалентность с прежней формулировкой

$$\text{не-На}(x, y) \leftarrow [x \neq A \vee y \neq B] \,\&\, [x \neq D \vee y \neq A] \,\&\, [x \neq D \vee y \neq C]$$

где $s \neq t$ служит сокращением для $\neg(s = t)$. Доказательство приводится на рис. 11.2.

На последнем шаге доказательства три условия проверяются с помощью отрицательных утверждений

$$\text{On6} \quad D \neq B \leftarrow$$
$$\text{On7} \quad D \neq A \leftarrow$$
$$\text{On8} \quad D \neq C \leftarrow$$

Доказательство для клаузальной формы мы предлагаем в качестве упражнения 3).

Допустим, что только–если часть определения задана на мета-языке:

> Клаузы On1 . . 3 выражают единственные условия при которых отношение На истинно

Доказательство на мета–уровне демонстрирует, что любая попытка попробовать достичь цели ← На (x, D) приводит к неудаче. Структура доказательства, тем не менее, похожа на структуру доказательства на объектном уровне (рис. 11.3). Последний шаг доказательства показывает, что с помощью On1 . . 3 не удастся согласовать На (x, D), поскольку D отлично

от (не согласуется с) A, B и C. Отметим, что доказательство на объектном уровне нуждается в явном рассуждении о равенстве. Кларк [Clark 1978] показал, что в общем случае явные аксиомы равенства нужны на объектном уровне для того, чтобы имитировать неудачу согласующего алгоритма на мета-уровне.

Клаузы Хорна, дополненные отрицанием, интерпретируемым как неудача

Интерпретация только–если на мета–уровне влечет за собой интерпретацию *отрицания как неудачи*: не–Р выполняется, если с помощью если-частей определений не удается установить Р. Язык клауз Хорна, дополненный отрицанием по неудаче обеспечивает мощное расширение чистого языка клауз Хорна. Такой расширенный язык легче и эффективней в применении и обладает значительной долей выразительности стандартной формы логики. Необходимой принадлежностью всех реализаций Пролога является либо возможность явного задания оператора отрицания, либо наличие средств для его определения.

Выразительные возможности языка клауз Хорна вместе с отрицанием иллюстрируется определением подмножества

$$x \subseteq y \leftarrow \forall z[z \in x \rightarrow z \in y]$$

которое можно переформулировать так:

$$x \subseteq y \leftarrow \text{не-}\exists z[z \in x, \text{не } [z \in y]]$$

x является подмножеством y, если отсутствие z в x приводит к неудаче проверку принадлежности z к y. Явный квантор существования ∃z может быть исключен, а знак отрицания может быть поставлен перед атомарной формулой, если воспользоваться вспомогательным предикатом Неподмн(x, y), который истинен, если x не является подмножеством y. Тогда определение подмножества приобретает вид

$$x \subseteq y \leftarrow \text{не–Неподмн}(x, y)$$
$$\text{Неподмн } (x, y) \leftarrow z \in x, \text{не-}[z \in y]$$

x является подмножеством y, если нельзя показать, что он не является подмножеством y; x не является подмножеством y, если в x существует z, проверка принадлежности которого к y приводит к неудаче. Аналогичное преобразование можно применить и к определению свободного блока:

Свободен (y) ← не–Покрыт (y)
Покрыт (y) ← На (x, y).

Кларковский анализ отрицания, интерпретируемого как неудача, исходит из допущения того, что отрицания могут быть преобразованы к виду, когда они поставлены лишь перед атомарными формулами.

Кларк показал, что клаузы Хорна вместе с отрицанием, интерпретируемым как неудача, все же не обладают всей мощью отрицания в стандартной форме логики. Простейшим примером может служить предложение

$$P \leftarrow \text{не–}P$$

из которого следует

$$P \leftarrow$$

в стандартной форме логики, поскольку

$$P \leftarrow \text{не}\text{–}P$$

эквивалентно

$$P, P \leftarrow$$

эквивалентно

$$P \leftarrow$$

Но попытка достичь

$$\leftarrow P, \text{ если известно } P \leftarrow \text{не}\text{–}P$$

оказывается безуспешной, поскольку она приводит к зацикливанию при попытке интерпретации отрицания как неудачи.

Более сложный бесконечный цикл возникает при попытке достичь цель

$$\leftarrow A$$

с использованием

(1) $A \leftarrow P(x)$
(2) $A \leftarrow \text{не}\text{–}P(x)$
(3) $P(x) \leftarrow P(f(x))$

при интерпретации отрицания как неудачи. И процедура (1), и процедура (2) приводят к процедурному вызову

$$\leftarrow P(x)$$

который не приводит ни к успеху, ни к неудаче за конечное время. Но в стандартной форме логики $A \leftarrow$ есть резольвента (1) и (2).

Эти примеры подсказывают, что дедуктивная сила вывода при использовании отрицания как неудачи может быть увеличена, если к ресурсам системы поиска решений на языке клауз Хорна добавить средства распознавания зацикливания. Но из-за неразрешимости логики [Church 1936] никакая система поиска решений не может распознать все ситуации, в которых цель недостижима. Поэтому не существует наилучшей системы поиска решений, но не существует и предела расширения возможностей системы поиска решений по распознаванию циклов или по интерпретации отрицания как неудачи.

Распознавание неудачи при помощи выявления зацикливаний на метаязыке эквивалентно использованию доказательства по индукции в объектном языке за счет добавления к ресурсам системы поиска решений средств доказательства по индукции. Доказательство по индукции нужно, кроме того, во многих случаях, когда только–если части определений применяются для доказательства свойств программ.

Рассмотрим если-часть определения отношения Присоединить на языке клауз Хорна

A1 Присоединить (nil, x, x) ←
A2 Присоединить (x.y, z, x.y′) ← Присоединить (y, z, y′)

Это определение обладает тем свойством, что

Присоединить (x, nil, x) верно для всех списков (x)

В доказательстве этого свойства требуется индукция по структуре списков. Мы представим и доказательство на объектном уровне и доказательство на мета-уровне. Оба доказательства имеют сходную структуру. Но доказательство на мета-уровне, в силу его неформальности, легче рассмотреть сначала.

Предположим, что A — произвольный список. Нужно показать, что

A3 Присоединить (A, nil, A) ←

может быть доказано с помощью (A1) и (A2). Доказательство проводится индукцией по структуре A. Если A пусто (равно nil), то доказательство (A3) проводится за один шаг с использованием лишь (A1). Если A есть B.A′, то по индуктивной гипотезе существует некоторое n-шаговое доказательство для

Присоединить (A′, nil, **A′**) ←

Добавляя к доказательству еще один шаг, основанный на A2, мы получим n + 1-шаговое доказательство для

Присоединить (x.A′, nil, x.A′) ←

для любого x, и поэтому, в частности, доказываем (A3).

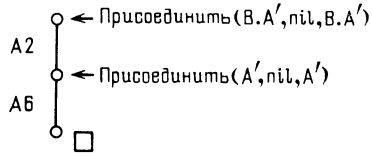


Рис. 11.4

Для доказательства на объектном уровне необходимо выразить индукционную схему для списков на объектном языке.

A4 $F(x) \leftarrow$ Список (x) & $F(nil)$ & $\forall y \forall z [F(z) \rightarrow F(y.z)]$

где $F(x)$ есть произвольная формула, в которую свободно входит лишь переменная x, а $F(t)$ для любого терма t получается заменой всех свободных вхождений x в F на терм t. Доказательство на объектном уровне может быть выполнено в клаузальной форме, но неклаузальное доказательство выглядит естественнее. Выполним отрицание теоремы, которую надо доказать и начнем рассуждения в обратном направлении от цели:

A5 Список (A) ←
← *Присоединить* (*A*, *nil*, *A*)

По A4, полагая, что F(x) есть Присоединить (x, nil, x), получаем:

← *Список* (*A*), *Присоединить* (*nil*, *nil*, *nil*),
∀ y ∀z [Присоединить (z, nil, z) → Присоединить (y.z, nil, y.z)]

По A5 и A1 получаем

← ∀ y ∀ z [Присоединить (z, nil, z) → Присоединить (y.z, nil, y.z)]

Это приводит к утверждению и к подцели, показанным на рис. 11.4.

Подобный метод доказательства свойств программ при помощи аксиомы индукции, выраженной на объектном языке, был разработан Кларком и Тарнлундом [Clark, Tarnlund 1977].

Критика свойства монотонности логического следования

Логика часто являлась мишенью для критики. Одним из недавних и серьезных критических замечаний является высказанная Минским [Minsky 1975] критика свойств монотонности логического следования.

Рассмотрим вновь пример из мира блоков

On1 На (A, B) ←
On2 На (D, A) ←
On3 На (D, C) ←
Свободно (y) ← не-∃ x На (x, y)

дополненный подразумеваемой только–если частью определения отношения На. Из этих посылок следует заключение

Свободно (D) ←

Под свойством монотонности логического следования понимается то, что это же заключение продолжает оставаться истинным вне зависимости от того, какие новые посылки добавляются. В частности, если мы добавим новую посылку

On На (E, D) ←

предыдущее заключение о том, что D свободно, остается истинным, несмотря на то, что оно очевидным образом несовместно с новой добавленной информацией.

Критики убеждают, что монотонность логического следования противоречит здравому смыслу*). Действительно, если принять во внимание новую посылку На (E, D) ←, то здравый смысл должен отвергнуть предыдущее заключение Свободно (D) ←. Но поскольку логика требует, чтобы заключение оставалось истинным, она оказывается неприемлемой в качестве модели человеческого мышления.

По нашему мнению этот довод критиков ошибочен, ибо они слишком просто смотрят на то, что происходит, когда в логическую базу данных

*) Монотонность логического следования есть следствие предположения о замкнутости мира. – *Примеч. Д.А. Поспелова.*

добавляется новая посылка. Мы приведем в последней главе доводы в пользу того, что если база данных становится несовместной, то совместность ее должна быть восстановлена либо за счет устранения, либо за счет модификации посылок в базе данных. В нашем примере следует либо устранить новую информацию, либо устранить или модифицировать только–если часть определения отношения На. Вероятно, более естественно будет либо заменить исходную только если посылку новой, говорящей о том, что лишь On1 . . 4 определяют отношение На, либо вообще избавиться от посылки только–если. В обоих случаях предыдущее заключение Свободно (D)← перестает быть верным в новой базе данных.

Логика счастливо избегнет упреков в монотонности логического следования, если будет использоваться должная оценка только–если посылок и если будет проводиться в жизнь реалистический взгляд на то, как базы данных изменяются во времени.

Упражнения

1. Используйте только–если часть определения факториала совместно с посылкой

$$\leftarrow 0 = n + 1$$

для того, чтобы показать, что единственным значением факториала 0 может быть только 1.

2. Покажите, что

не–Присоединить (nil, a. nil, nil)

является следствием если–и–только–если определения отношения Присоединить. Сравните доказательства на объектном языке и на мета-языке и выявите аксиомы равенства, необходимые для доказательства на объектном языке.

3. Преобразуйте посылки On4 . . 8 в клаузальную форму и примените резолюцию для доказательства того, что

Свободно (D)

следует из них.

4. Покажите с помощью резолюции и факторизации, что

Присоединить (A, nil, A)

следует по если–и–только–если определению из Присоединить вместе с приемлемыми аксиомами индукции и равенства, выраженными в клаузальной форме.

5. Используя отрицание как неудачу, переформулируйте определения арки и башни, данные в гл. 4, так, чтобы задача

← Арка (w)

имела бы только два решения для сцены, описанной A4 . . 12:

w = a(б(B, A), D, C)
w = a(C, D, б(B, A))

6. Пусть даны клаузы Хорна

Присоединить (nil, x, x) ←
Присоединить (x.y, z, x.u) ← Присоединить (y, z, u)
Элемент (x, x.y) ←
Элемент (x, y.z) ← Элемент (x, z)

Покажите на мета-языке, используя индукцию, следующее утверждение

Для всех x, u, v и w если Присоединить (u, v, w) и Элемент (x, w), то Элемент (x, u) или Элемент (x, v).

7. а) Пусть даны посылки

N1 $x \subseteq y \leftarrow$ не–Неподмн (x, y)
N2 Неподмн (x, y) $\leftarrow z \in x$, не–$[z \in y]$
$a \in A \leftarrow$
$a \in B \leftarrow$
$b \in B \leftarrow$

Покажите, что $A \subseteq B$, интерпретируя отрицание как неудачу.

б) Пусть принадлежность элементов множествам A и B описывается при помощи клауз Хорна. Обсудите условия, при которых можно показать, что

(i) $A \subseteq B$;
(ii) $\phi \subseteq B$,

если не существует клауз, устанавливающих принадлежность элементов к ϕ;

(iii) $A \subseteq U$,

если известно, что $x \in U$;

(iv) $A \subseteq A$.

Мета–языковую интерпретацию только–если и сочетание мета–языка с объектным языком можно получить формализуя мета–язык и сливая его с объектным языком. Такая комбинация объектного языка и мета–языка приводит к появлению логической системы, более близкой к естественному языку, чем привычные системы, в которых оба эти языка разделены. В естественном языке комбинация объектного языка и мета–языка приводит к различным парадоксам, вроде такого самоприменимого предложения:

Это предложение ложно

Мы увидим вскоре, что попытки организовать такие парадоксы в смешанном формальном языке приведут к высказыванию истинных, но недоказуемых предложений типа:

Это предложение недоказуемо

Понятие и доказательство недоказуемости основываются на Геделевском подходе к доказательству неполноты формальной арифметики [Gödel 1931]. Вместо неполноты арифметики мы положим в основу нашего изложения невозможность любой попытки полной формализации понятия доказуемости. Кроме того, доказательство неполноты проще для случая доказуемости, чем для арифметики.

Наша идея комбинации объектного языка и мета–языка преследует, прежде всего, практическую цель. Смешанный язык обладает большей выразительностью и большими возможностями поиска решений, чем один лишь объектный язык. Смешанный язык обеспечивает возможности для таких приложений логического программирования как понимание естественного языка, управление базами данных, управление процессами и редактирование программ.

Смешанный язык сочетает в себе объектный язык и мета–язык при сохранении нормальной логической семантики. Поэтому все сведения из теории поиска решений, сформулированные в предыдущих главах лишь для объектного языка, применимы безо всяких изменений и для более мощной комбинации объектного языка и мета–языка.

Комбинация объектного языка и мета–языка является частным случаем более общей конструкции. Если даны любые два языка (т.е. логические системы вместе с ассоциированными с ними логическими процеду-

рами), то иногда оказывается возможной имитация процедуры доказательства в одном языке, скажем, L1, посредством другого языка L2. Эта имитация завершается заданием в L2 бинарного отношения доказуемости, которое является истинным, если заключение процедуры доказательства может быть выведено из посылок в L1. Предложения в L1 следует именовать при помощи термов из L2, а отношение доказуемости в L1 должно быть поименовано бинарным предикатным символом, скажем, "Продемонстрировать" и задано в виде предложений "Дкзмсть" в L2. Если предусмотреть, чтоб определение Дкзмсть *корректно представляло* отношение доказуемости в L1, то *имитация* при помощи Дкзмсть в L2 станет эквивалентной *прямому исполнению* процедуры доказательства в L1. L2, т.е. язык, в котором Дкзмсть имитирует L1, является мета–языком для объектного языка L1. Язык L2 должен обладать достаточной выразительной мощностью для того, чтобы он мог служить в качестве мета–языка. Для любого объектного языка в качестве мета–языка вполне может подойти язык клауз Хорна.

Можно указать и несколько особо интересных аспектов этой проблемы. В случае, когда мета–язык сужается до такого подмножества логики как язык клауз Хорна, а объектный язык совпадает со всей стандартной формой логики, оказывается, что мета–язык усиливает свои собственные возможности поиска решений за счет более мощного объектного языка. В общем случае, простая, не слишком гибкая система поиска решений может быть усилена с помощью своих собственных средств благодаря имитации действий более сложной и изысканной системы.

Если объектный язык и мета–язык являются идентичными, то в качестве соединения объектного языка с его мета–языком может рассматриваться сам этот язык, дополненный определением Дкзмсть своего собственного отношения доказуемости.

Корректная представимость

Условие корректной представимости в своей основе является одинаковым и для определения отношения доказуемости и, например, для определения сложения натуральных чисел.

Для того, чтобы определить сложение средствами логики, нужно придать числам имена при помощи термов. Один из самых простых способов придания имен неотрицательным целым числам заключается в выборе константного символа 0 для нуля и одноместного функционального символа s для функции следования: если t именует целое число n, то s(t) именует целое число n + 1.

Приведенное ниже определение сложения посредством клауз Хорна корректно представляет отношение сложения, поименованное при помощи предикатного символа "Плюс".

Plus1 Плюс (0, x, x) ←
Plus2 Плюс (s(x), y, s (z)) ← Плюс (x, y, z)

Смысл того, что Plus1 . . 2 *корректно представляет* отношение сложения таков:

какими бы ни были целые неотрицательные числа l, m и n, поименованные, соответственно, через r, s и t, отношение l+m=n выполняется если и только если из Plus1..2 следует Плюс(r, s, t) ←

Для того, чтобы определить доказуемость, нужно придать предложениям и другим выражениям имена с помощью термов. Это может быть сделано многими способами, но мы не станем сейчас углубляться в детали. Если дано представление предложений с помощью термов, то определение Дкзмсть в языке L2 *корректно представляет* отношение доказуемости в языке L1, поименованное "Продемонстрировать", если–и–только–если:

какими бы ни были предложения X и Y из языка L1, поименованные, соответственно, термами X′ и Y′ из языка L2, заключение Y может быть выведено из посылки X в L1, если-и-только-если заключение Продемонстрировать (X′, Y′) может быть выведено из посылок Дкзмсть в L2.

Корректная представимость, однако, не требует, чтобы из Дкзмсть следовало ¬Продемонстрировать (X, Y) в L2, если из X не следует Y в L1.

Если задаться языком L1, то построение определения, корректно представляющего свою процедуру доказательства, не является особо сложной проблемой, потому что процедуры доказательства могут быть реализованы посредством компьютерных программ на языке клауз Хорна. И более того, любая программа на языке клауз Хорна, корректно воплощающая процедуру доказательства, корректно представляет свое отношение доказуемости.

Простое определение отношения доказуемости

Сейчас мы представим верхний уровень определения на языке клауз Хорна отношения доказуемости для языка клауз Хорна, в котором посылки рассматриваются как программы, а заключения – как наборы целей. В целях лучшего восприятия договоримся понимать строки из строчных букв, вроде

прог, цели, подстан

как переменные, а строки, начинающиеся с прописных букв, вроде

NIL, Зевс, A

как константы.

Первая клауза нашей программы утверждает, что любая программа *демонстрирует* разрешимость пустого набора целей. Вторая клауза в нисходящей интерпретации говорит о том, что для того, чтобы *продемонстрировать* разрешимость набора целей, надо:

выбрать цель;

подыскать подходящую процедуру в программе;

переименовать переменные в процедуре так, чтобы они отличались от переменных в наборе целей;

согласовать выбранную цель с заголовком процедуры;

добавить тело процедуры к оставшимся в наборе целям;

применить согласующую подстановку для получения нового набора целей;

продемонстрировать, что программа разрешает новый набор целей.

```
D1    Продемонстрировать (прог, цели) ← Пустые (цели)
D2    Продемонстрировать (прог, цели) ←
                Выбрать (цели, цель, остаток), Элемент (процедура, прог),
                Переименоватьперем (процедура, цели, процедура′),
                Разделитьна (процедура′, заголовок, тело),
                Согласовать (цель, заголовок, подстан),
                Добавить (тело, остаток, внутрцели),
                Применить (внутрцели, подстан, новыецели),
                Продемонстрировать (прог, новыецели).
```

Для завершения этого определения надо бы определить процедуры нижних уровней и договориться относительно структур данных для именования программ, целей, набора целей и подстановок. Вместо того, чтобы определять все это в общем случае, мы покажем интерфейс между верхним уровнем и структурой данных для задачи об ошибающихся греках.

Назовем атомарную формулу, предикатный символ которой называется P, а список аргументов называется t, посредством терма

```
атом (P, t)
```

Тела процедур и наборы целей будем именовать при помощи списков имен атомарных формул, которые в них содержатся. Программы и процедуры будем именовать при помощи констант. Приводимые ниже клаузы задают интерфейс между верхним уровнем определения Продемонстрировать и структурами данных в задаче об ошибающихся греках:

```
Элемент (F1, F) ←
Элемент (F2, F) ←
Элемент (F3, F) ←
Элемент (F4, F) ←
Разделитьна (F1,  атом (Ошибается, X . NIL),
                  атом (Человек, X . NIL) . NIL) ←
Разделитьна (F2,  атом (Человек, Тьюринг.NIL), NIL) ←
Разделитьна (F3,  атом (Человек, Сократ.NIL), NIL) ←
Разделитьна (F4,  атом (Грек, Сократ.NIL), NIL) ←
```

Цель верхнего уровня описывается клаузой

```
← Продемонстрировать (F,
                  атом (Ошибается, X . NIL) .атом (Грек,X.NIL).NIL)
```

Константный символ X представляет переменную x.

Непосредственное исполнение versus имитация

Пусть Дкзмсть состоит из клауз D1 . . 2 вместе с необходимыми для завершения определения Продемонстрировать клаузами нижнего уровня, какими бы они ни были. Предположим, что Дкзмсть корректно представляет отношение доказуемости языка L1 и выражается в языке L2 (который может быть идентичен языку L1).

Корректная представимость гарантирует то, что непосредственное исполнение в L1 и имитация в L2 эквивалентны и взаимозаменяемы. Если

даны предложения X и Y в языке L1, поименованные, соответственно, термами X′ и Y′ в языке L2, то непосредственное *исполнение* в L1 процедуры доказательства возможности вывода Y из X в L1 эквивалентно *имитации* L1, состоящей в доказательстве того, что Продемонстрировать (X′, Y′), может быть выведено из Дкзмсть в L2.

Эквивалентность непосредственного исполнения и имитации идентична введенному Вайраухом принципу рефлексии [Weyhrauh 1978].

Корректная представимость отношения доказуемости означает, что можно организовать совместные действия объектного языка и мета-языка по поиску решений. Задача в объектном языке может быть решена при помощи имитации в мета-языке. И наоборот, задача вида

Продемонстрировать (X′, Y′),

в мета-языке может быть решена благодаря тому, что в объектном языке можно показать, что

Y может быть выведено из X.

Последний вариант обладает некоторым преимуществом, потому что непосредственное исполнение, вообще говоря, более эффективно, чем имитация в мета-языке.

Но имитация в мета-языке может оказаться более мощной, чем непосредственное исполнение. В частности, иногда оказывается возможным заменить отдельные доказательства различных, но сходных теорем одним единственным доказательством в мета-языке. В качестве тривиального примера могут служить все приведенные ниже задачи; их нужно решать раздельно в объектном языке, но можно найти одно общее для них всех решение в мета-языке.

Смертен (Сократ) ←

может быть выведено из

Человек (Сократ) ←

и из

Смертен (x) ← Человек (x);

Ядовитый (⏉) ←

может быть выведено из

Boletus (⏉) ←

и из

Ядовитый (x) ← Boletus (x);

Зверь (Пафф) ←

может быть выведено из

Дракон (Пафф) ←

и из

Зверь (x) ← Дракон (x).

А в мета-языке можно с помощью одного общего доказательства показать, что для любой переменной x, предикатных символов P и Q и терма t из

объектного языка

$$Q(t) \leftarrow$$

можно вывести из

$$P(t) \leftarrow$$

и из

$$Q(x) \leftarrow P(x)$$

Мета-язык оказывается более мощным, чем объектный язык и в несколько ином смысле. Процедура доказательства на объектном уровне может показать лишь, что X может быть выведено из Y, если и X и Y заданы как исходные данные. Процедура же доказательства на мета–уровне может достичь цели Продемонстрировать при любых образцах для входных и выходных данных.

Если, например, дано подходящее определение того, что выражает интересующее нас предложение, то можно применить мета–уровневое целевое предложение

← Продемонстрировать (X′, y), Интересующеенас (y)

чтобы, по крайней мере, теоретически породить интересующие нас следствия заданного множества посылок X. Кроме того, за счет одновременного, а не последовательного решения этих двух задач можно в соответствии с критерием интересности предложений организовать направленное порождение следствий X. Целевое предложение

← Продемонстрировать (t, Y′),

в котором Y′ именует некоторое заданное слово, а t является частично означенным термом, который именует заданный набор известных посылок X и, кроме того, набор неизвестных посылок x, может быть применено для нахождения пропущенных посылок x. Целевое предложение

← Продемонстрировать (t, Y'_1), Продемонстрировать (t, Y'_2), . . . ,
Продемонстрировать (t, Y'_m)

может быть применено для нахождения таких пропущенных посылок, из которых, если к ним добавить заданные посылки X, будут следовать все заключения $Y_1, Y_2, \ldots, Y_m$. В простейшем случае, если заключения достаточно просты, пропущенные посылки могут быть индуктивными обобщениями заключений. Если обеспечить достаточно рациональные ограничения целостности для процедуры доказательства, то в процессе генерации заключений можно избавиться от порождения тривиальных посылок типа Y_1 & & Y_2 & . . . & Y_m, из которых заключения следуют тривиальным образом.

Добавление и подавление посылок

Языкам семейства Плэннера и специальным версиям Пролога изначально присущи некоторые возможности отношения Продемонстрировать благодаря обеспечению средств добавления и подавления посылок в процессе выполнения демонстрации. Вместо того, чтобы явно попытаться достичь цель вида

Продемонстрировать (X′, Y′),

в этих языках нужно *добавить* предложения X в программу, попытаться *доказать* Y, и подавить X после этого. Поскольку посылки динамически меняются в процессе выполнения демонстрации, то такие программы могут быть весьма опасными.

Добавление и подавление посылок может быть выполнено с большей безопасностью при помощи отношения Продемонстрировать. Кроме того, можно повысить эффективность процесса, если непосредственно рекурсивно исполнять процедуры доказательства на той же самой машине или параллельно исполнять их на другой машине, а не имитировать ее с помощью определения. С другой стороны, если цели Продемонстрировать для других входных и выходных образцов не могут быть достигнуты за счет добавления и подавления посылок, то их можно достичь, используя определение. Добавление и удаление посылок можно использовать только, если объектный язык и мета-язык совпадают. А если обеспечить достаточную мощность мета-языка, то отношение Продемонстрировать можно использовать для связи двух произвольных языков.

Раскрутка

Мета-язык L2 может отличаться в своих потенциальных возможностях от объектного языка L1. Если он менее приспособлен для того, чтобы именно с его помощью начать поиск решений, то можно воспользоваться определением доказуемости Дкзмсть в L1 для имитации L1 и, следовательно, для расширения потенциальных возможностей самого L2. Такие действия принято называть раскруткой (язык L2 поднимает себя самого над своими возможностями *) используя определение Дкзмсть для решения более интеллектуальных, чем обычно, задач, т.е. используя способ активизации более интеллектуальной процедуры доказательства).

Раскрутка может быть эффективной даже в том случае, когда более мощный язык L1 самостоятельно не существует. Определение Дкзмсть, если оно состоятельно, может само по себе стать таким средством, которое вызовет L1 к жизни.

Раскрутка, а если говорить более общо, эмуляция одного языка средствами другого – это столбовая дорога компьютерной информатики. Эмуляция языка состоит в написании такой программы, которая ведет себя как транслятор или интерпретатор с этого языка на другой существующий язык.

Клаузы D1 . . 2, которые определяют верхний уровень процедуры доказательства на языке клауз Хорна для L1, могут быть использованы для раскрутки простой нисходящей процедуры доказательства на языке клауз Хорна в L2, которая исполняет процедурные вызовы последовательно в порядке их записи. При помощи подходящих определений остатка программы и, в частности, процедуры Выбрать, можно задать процедуру доказательства, которая исполняет процедурные вызовы совместно. Хотя L2 использует процедурные вызовы последовательно, новая процедура доказательства в L1 исполняет их как сопрограммы в соответствии с кри-

*) В оригинале в значении слова "раскрутка" использован термин bootstrapping – *Примеч. пер.*

терием, заданным в процедуре Выбрать. За счет подходящей модификации такого определения к новой процедуре доказательства в L1 могут быть добавлены и иные улучшения, например, такие как распознавание зацикливаний, интеллектуальный бэктрекинг или преобразование целей. Если чуть умерить свои запросы, то определение Продемонстрировать можно ограничить лишь средствами улучшения входного синтаксиса L2, например, позволив применять инфиксную нотацию для предикатных и функциональных символов. Если же замахнуться на большее, то можно было бы создать процедуру доказательства для более богатой версии логики, например, для полной клаузальной формы или для стандартной формы логики.

В системах и программах на Прологе средства раскрутки начали применяться с самых первых разработок в Марселе в 1972 году. Прежде всего, эти средства использовались для улучшения входного синтаксиса и для выполнения сопрограмм. Большое количество программ на языке клауз Хорна, реализующих доказуемость в среде клауз Хорна было написано также и в Имперском колледже. Простые программы на языке клауз Хорна обычно выполняются в 100 раз медленнее, когда они имитируются подобными определениями, а не исполняются непосредственно. Программы на Прологе были написаны также и для доказательств в среде нехорновских клауз, а Крыся Брода сумела создать их и для стандартной формы логики. Компилятор для языка Пролог, написанный на Прологе Уорреном, Перейрой и Перейрой [Warren, Pereira, Pereira, 1977] и интерпретатор Колмероэ [Colmerauer 1977] для ограниченного подмножества естественного языка тоже могут рассматриваться как приложения раскрутки.

Комбинация объектного языка и мета-языка

Пока что мы исходили из предпосылок об асимметрии связи между нашими двумя языками L1 и L2. Но нет никакой причины для того, чтобы один язык знал о своем "компаньоне" больше, чем второй. В обоих языках имеется возможность определить процедуру доказательства другого языка. Каждый язык мог бы служить мета–языком для другого и мог бы имитировать его процедуру доказательства.

И нет никакой причины, чтоб оба эти языка не могли бы быть идентичными по всем параметрам. Поэтому допустимо существование единственного языка, снабженного определением Дкзмсть, являющимся корректным представлением его собственной процедуры доказательства. Если дана задача вида

Продемонстрировать (X′, Y′)

то она может быть использована для имитации себя самой, или, что эквивалентно, она может непосредственно показать, что Y может быть продемонстрировано исходя из X. Решение этой задачи методом непосредственного исполнения эквивалентно процедуре доказательства, рекурсивно вызывающей самое себя.

Такая связь между объектным языком и метаязыком хорошо знакома по языку программирования Лисп [McCarthy и др. 1962]. Функция компилятора или интерпретатора Лиспа состоит в следующем: означить выражение у в условиях х, определяющих значения

символов, встречающихся в y, вырабатывая результат z, являющийся значением y в условиях x.

В функциональных обозначениях это

eval(x, y) = z

что похоже на Продемонстрировать, за исключением того, что добавился параметр z, именующий выходное значение. Чуть позже мы примем во внимание, что удобнее расширить определение Продемонстрировать до четырехаргументного отношения

Продемонстрировать (x, y, u, z),

которое истинно при следующих условиях: если даны посылки, именуемые x; заключение, именуемое y; управление, именуемое u, то процедура доказательства порождает выход, именуемый z.

Функцию eval можно задать и в Лиспе с тем же успехом, с каким Продемонстрировать можно задать в логике. Точно также как то, что цель Продемонстрировать с подходящим входным параметром может быть достигнута либо с помощью определения, либо с помощью прямого исполнения, функциональные eval–вызовы могут быть означены в Лиспе либо непосредственно, либо за счет рекурсивной работы лисповского механизма означивания. Поскольку у лисповских функций входные параметры фиксированы, постольку явное использование eval может быть заменено рекурсивными вызовами. Хочется подчеркнуть, что именно изучение аналога лисповского eval в логике навело автора этих строк и Кена Боуэна на мысль провозгласить идею слияния мета–языка и объектного языка, которая и представлена в этой главе.

Неполнота комбинации объектного языка и мета-языка

Мы видели, что комбинация объектного и мета–языка приводит к парадоксам типа автореференции естественного языка. Попытки обойти эти парадоксы, в свою очередь, приводят к построению истинного, но недоказуемого предложения

D ¬Продемонстрировать (Дкзмсть′, D),

которое ссылается на свое собственное имя D. Терм Дкзмсть′ именует определение Дкзмсть отношения Продемонстрировать.

Легко показать, что *если Дкзмсть непротиворечиво и корректно представляет отношение доказуемости, то ни предложение, названное D, ни его отрицание, не могут быть выведены из Дкзмсть.*

Для доказательства рассмотрим два случая:

(1) предложение, именуемое D, может быть выведено из Дкзмсть;
(2) его отрицание Продемонстрировать (Дкзмсть′, D) может быть выведено из Дкзмсть.

Случай 1. По предположению о корректности представимости из (1) следует что Продемонстрировать (Дкзмсть′, D) может быть выведено из Дкзмсть. Но тогда и предложение и его отрицание могут быть выведены из Дкзмсть, что противоречит предположению о непротиворечивости Дкзмсть.

Случай 2. По предположению о корректной представимости из (2) следует, что предложение, именуемое D, может быть выведено из Дкзмсть. И вновь, и само предложение, и его отрицание могут быть выведены из Дкзмсть, что противоречит предположению о непротиворечивости Дкзмсть.

Поскольку оба случая приводят к противоречию, постольку ни предложение, именуемое D, ни его отрицание не могут быть выведены из Дкзмсть.

Но высказывание

Предложение D может быть выведено из Дкзмсть

или эквивалентное ему (в силу корректной представимости)

Продемонстрировать (Дкзмсть′, D)

может быть истинным или ложным по отношению к доказуемости. Мы только что показали (случай 1), что оно не истинно. Поэтому его отрицание

D ¬ Продемонстрировать (Дкзмсть′, D)

истинно, хотя и недоказуемо.

Предложение, названное D, связано с интерпретацией отрицания как неудачи. Если дана задача

Продемонстрировать (Дкзмсть′, D)

то процедура доказательства не приводит ни к успеху, ни к неудаче за конечное время. (Ибо конечное время приведения к неудаче будет означать, что

D ¬ Продемонстрировать (Дкзмсть′, D)

может быть доказано исходя из если–и–только–если определения Дкзмсть). Поэтому процедура доказательства в своих попытках решить задачу никогда не останавливается, а потому ее отрицание

D ¬ Продемонстрировать (Дкзмсть′, D)

действительно говорит о том, что задача не может быть решена.

Предложение, названное D, можно построить разнообразными способами, включая и тот, что был использован в Геделевском доказательстве неполноты.

Более полная форма отношения Продемонстрировать

Для того, чтобы упростить наше обсуждение, мы предполагали, что процедура доказательства определяет двуместное отношение между посылками и заключениями. В действительности, процедуры доказательства сложнее. Они включают также управление спецификациями, которое направляет стратегию доказательства, и они также сообщают выходное значение. Поэтому реалистичнее рассматривать процедуру доказательства, определяемую четырехместным отношением

Продемонстрировать (x, y, u, z),

которое истинно в следующих условия: если даны посылки, именуемые x; заключения, именуемые y; управление, именуемое u, то процедура доказательства порождает выход, именуемый z. Параметр управления может, например, указывать

(1) что следует применять тот или иной метод доказательства;

(2) что требуется одно решение, все решения или "наилучшие" решения;

(3) что в качестве выхода z требуются доказательство, или трасса поиска, или подстановка для переменных в заключении или простой ответ вида "да" или "нет".

Трасса процедуры доказательства состоит из последовательности предложений и других выражений, порожденных процедурой доказательства в процессе поиска решения. Поэтому процедура доказательства может успешно возвратить в качестве выходного результата трассу неуспешного поиска решения. Она может также возвратить простой ответ "нет", если сможет обнаружить, что пространство поиска не содержит решений.

Более полная форма отношения Продемонстрировать годится для получения и обработки списка всех решений. Это особенно полезно в приложениях баз данных для подсчета всех ответов на запрос или для вывода списка всех ответов в виде таблицы. Если дана база данных S поставщиков и деталей, определенная на языке клауз Хорна, то, например, отношение Продемонстрировать может быть использовано и для формулировки, и для ответа на запрос:

Сколько поставщиков канцпринадлежностей располагаются в Лондоне?

```
← Продемонстрировать (S,
    атом (Поставщики, X . Канцпринадлежности. NIL).
        атом (Расположение, X . Лондон. NIL). NIL,
    все (X), z),
    Сосчитать (z, w)
```

Здесь атом все (X) говорит о том, что в качестве выхода z требуется список всех различных ответов, состоящих из подстановок для переменной X. Сосчитать (z, w) можно задать так:

```
Сосчитать (NIL, 0) ←
Сосчитать (u . v, w) ←  Сосчитать (v, w′),
                        Плюс (w′, 1, w)
```

Вместо подсчета списка всех ответов, можно воспользоваться процедурой

Формат (z, w),

которая преобразует список z, задавая разбиение по страницам, разбиение по строкам и описывая поля так, чтобы результирующий список w на печати выглядел как таблица.

Упражнения

1. Процедуры верхнего уровня D1 . . 2 определения доказуемости в среде клауз Хорна могут быть применены для задачи об ошибающихся греках вообще без задания процедур нижнего уровня. Достаточно предоставить утверждения, которые решают подзадачи, возникающие в процессе попыток решить верхнеуровневую задачу. Нижеследующие утверждения достаточны для переименования процедур F1 . . 4 и для на-

хождения частей результирующих процедур:

Переименоватьперем(F1, цели, F1′) ←
Переименоватьперем(F2, цели, F2′) ←
Переименоватьперем(F3, цели, F3′) ←
Переименоватьперем(F4, цели, F4′) ←
Разделитьна(F′, атом(Ошибается, Y),
атом(Человек, y). NIL) ←

а) Предоставьте утверждения или простые процедуры для оставшихся условий в D1 . . 2.

б) Используя утверждения и простые процедуры из а), проверьте D1 . . 2 для задачи об ошибающихся греках, применяя нисходящий вывод и бэктрекинг для поиска решений.

2. Дополните определение D1 . . 2 отношения Продемонстрировать определением полного набора процедур нижнего уровня. Для этой цели полезно применить разнообразные структуры данных для именования выражений объектного языка:

а) Предикатные символы и функциональные символы могут именоваться константными символами.

б) Константные символы могут именоваться термами const(t), где t означает число, например, 0, s(0) и т.д.

в) Переменные могут именоваться термами var(t), где t – число.

г) Составные термы могут именоваться термами term(s, t), где s означает функциональный символ, а t – список термов.

д) Атомы и списки атомов в целевых предложениях и телах процедур могут именоваться как выше.

е) Процедуры могут именоваться термами proc(s, t), где s именует заголовок, а t – тело процедуры.

ж) Программы могут именоваться списками процедур, которые они содержат.

з) Подстановки могут именоваться списками компонентов подстановок вида sub(s, t), где s означает переменную, а t – терм.

Отметьте, что простой способ переименовать переменные в процедуре состоит в:

(i) нахождении числа T, максимального из всех таких t, что var(t) встречаются в целях;

(ii) замене каждого вхождения переменной var(s) в процедуре вхождением переменной var(r), где r = s + T. Одно из простых определений отношения Согласовать:

Согласовать (выраж1, выраж2, подстан) ← Применить (выраж1, подстан, выраж3), Применить (выраж2, подстан, выраж3)

подвержено зацикливанию в случае, когда два выражения не могут быть согласованы. Более безопасное определение состоит в том, чтобы использовать два подстановочных параметра, один для текущей подстановки, которая согласует части обоих выражений, до которых уже дошло согласование, а второй – для окончательной согласующей подстановки.

3. Измените определение отношения Продемонстрировать, задав отношение

Продемонстрировать (прог, цели, подст)

которое истинно, когда программа обеспечивает достижение целей и порождает в качестве решения подстановку термов для переменных встречающихся в цели.

Это можно сделать на верхнем уровне просто за счет добавления некоторых экстраусловий к D2. Подстановка, нужная в заголовке клаузы, может быть сформирована с помощью подходящей комбинации подстановки, полученной за счет рекурсивного вызова процедуры Продемонстрировать в теле клаузы, с выходным результатом подстановки, согласующей выбранную цель с заголовком процедуры.

4. Определите верхний уровень интерпретатора программ на языке клауз Хорна, который сам задан на языке клауз Хорна и имеет *определенность первого рода.* Этот интерпретатор может получить определенность за счет явного поискового предписания искать в пространстве нисходящего поиска одну ветвь за один раз.

Ветви поискового пространства могут быть представлены в виде списков узлов. Каждый узел состоит из

(i) списка целей узла;

(ii) выбранной цели;

(iii) списка неиспытывавшихся процедур, которые еще не применялись к цели.

Для достижения начального набора целей следует обработать ту ветвь, единственный узел которой состоит из исходного целевого предложения, выбранной цели и подходящего списка неприменявшихся процедур.

Любая программа успешно обрабатывает ветвь, оконечная вершина которой содержит пустой список целей.

Для обработки ветви, оконечная вершина которой содержит непустой список неприменявшихся процедур для выбранной цели, следует попытаться согласовать цель с заголовком первой неприменявшейся процедуры:

(i) если согласование заканчивается неудачей, то следует убрать процедуру из списка неприменявшихся процедур и обработать новую ветвь;

(ii) если согласование заканчивается успешно, то следует убрать процедуру из списка неприменявшихся процедур, добавить к ветви новую оконечную вершину, содержащую новое целевое предложение, полученное в результате применения успешно закончившейся процедуры и обработать новую ветвь.

Для обработки ветви, оконечный узел которой содержит пустой список неприменявшихся процедур для выбранной цели, следует выполнить бэктрекинг, удаляя оконечную вершину из ветви и обработать новую ветвь.

5. Показать, что для любого множества клауз S существует соответствующее множество клауз Хорна S^*, такое, что S совместно (или несовместно), если и только если S^* обладает той же характеристикой. Поэтому любая задача, которая может быть выражена в клаузальной форме, может быть выражена при помощи клауз Хорна за счет использования соответствия $*$.

Это соответствие может быть установлено путем показа того, что отношение доказуемости для клауз в общем случае может быть определено в терминах клауз Хорна.

В предыдущих главах мы поняли, что логику можно использовать для представления информации и для поиска решений. Но информация меняется и ее представление должно меняться вместе с нею. Поэтому сейчас мы рассмотрим такие процессы, которые вынуждают представляющие их информационные системы меняться во времени. Понятие информационной системы*) обобщает изучавшиеся до сих пор понятия программы и базы данных; к информационным системам относятся и такие более сложные системы, как научные теории или машинноориентированные системы понимания естественного языка. В этой главе мы выясним также ту роль, которую в порождении изменчивости играет противоречие.

Информационные системы

На протяжении этой главы термином *информационная система* (*система фактов*) мы будем обозначать сочетание *совокупности посылок* (или фактов), *выраженных средствами логики с совокупностью процедур доказательства* и *процедур поддержки,* управляющих внесением изменений в информационные системы.

Информационные системы включают в себя посылки, представленные *явно* и заключения, представленные *неявно.* Практическое выяснение того, что некоторое предложение является неявным следствием, возможно лишь с некоторой степенью достоверности. Ведь доступность следствий зависит от сложности процесса поиска выводов. Чем более сложен этот процесс, тем менее доступными являются следствия. И, если процесс вывода слишком сложен, то выводимое при его помощи следствие можно считать вообще недостижимым, как если бы его вообще нельзя было вывести. Поэтому различные информационные системы могут совпадать по выводимым или следствиям, но разниться в своей практической значимости. Полезные следствия могут оказаться эффективно доступными в одной системе, но практически недоступными в другой.

Базы данных относятся к простым информационным системам. Они могут претерпевать изменения или вследствие внутренней реорганизации или под воздействием вновь поступающих данных и запросов.

*) Термин информационная система в советской литературе имеет другой смысл, нежели в этой книге. – *Примеч. Д.А. Поспелова*

Процедура доказательства здесь используется не только для ответа на запрос, но также и для принятия новых данных в базу данных. В этой ситуации возникает четыре возможности:

(1) новые данные могут быть выведены как следствие из уже существующей базы данных;

(2) из новых данных может выводиться часть данных, уже имеющихся в базе;

(3) новые данные независимы от существующих;

(4) новые данные несовместны с существующими. Последний случай особенно важен. Он включает как ситуацию нарушения новыми данными ограничений целостности, так и ситуацию, когда новые данные составляют исключение из общего правила.

Программы вместе с их спецификациями тоже можно рассматривать как информационные системы. Если программа несовместна со своими спецификациями, то ее можно сделать совместной, либо модифицируя самое программу, либо меняя спецификации. В программе, совместной со своими спецификациями, можно, не затрагивая спецификаций, менять какую-либо неэффективную процедуру на более эффективную. Программу можно менять также в целях адаптации к различным привходящим условиям.

Если информационная система предназначена для восприятия текстов, то она должна включать подсистему понимания текста, прочитанного до некоторой текущей позиции. Такая подсистема понимания должна уметь изменяться по мере поступления новой информации. При этом новая информация может оказаться просто новым предложением, воспринятым подсистемой понимания, а может оказаться и гипотезой, предназначенной для объяснения информации, уже воспринятой из текста. В обоих случаях для понимания новой информации иногда приходится делать выбор одной из нескольких альтернатив. Действительно, новое предложение может оказаться двусмысленным само по себе и, следовательно, может порождать альтернативные интерпретации, а может случиться и так, что уже воспринятая информация допускает объяснение средствами альтернативных гипотез. Если новая информация несовместна с текущим состоянием информационной системы, то приходится либо рассмотреть возможность принятия альтернативной интерпретации новых данных, либо пересмотреть всю предварительно полученную информацию.

Научные теории тоже относятся к информационным системам, потому что они воспринимают и учитывают экспериментальные данные прошедшего и предсказывают результаты экспериментов на будущее. Необходимость изменения теории может возникнуть в свете нового опыта или в результате выдвижения новых гипотез. Так, результаты экспериментов могут оказаться двусмысленными, а одни и те же явления могут объясняться различными гипотезами. И те и другие альтернативы надо выбирать исходя из их воздействия на состояние всей научной теории в целом. Если принятие какой-либо альтернативы переводит теорию в несовместное состояние, то совместность может быть восстановлена за счет ограничения или приемлемого изменения одной из посылок, приведших к противоречию. Сюда относится как случай устранения нового предложения,

так и случай, когда новое предложение принимается, а вместо него исключается одно из предыдущих*).

Динамика изменения информационных систем

И ситуации, когда информационная система запоминает результаты своего общения с окружающим миром, и ситуации, когда она выдвигает свои собственные гипотезы, возникают в результате попыток восприятия новой информации. Сейчас принято различать четыре основных дедуктивных отношения между новой информацией и текущим состоянием информационной системы. В каждом из этих четырех случаев в качестве нового состояния системы будут предложены различные варианты.

1. *Новая информация может быть выведена как следствие из текущего состояния информационной системы.* В этой ситуации информационная система успешно предсказывает новую информацию, а новое состояние системы совпадает со старым. При этом можно выделить посылки, которые участвуют в выводе новой информации, и оценить их полезность. Оценка полезности может быть получена исходя из значимости для вывода новых полезных сведений того расширения системы, к которому приводят посылки. Полезностная оценка посылок может пригодиться и позже для того, чтобы выяснить, какие посылки следует удалить или изменить, если встретится противоречие.

2. *Часть данных в текущем состоянии системы может быть выведена из объединения новой информации и остальной части данных в информационной системе.* Совокупность явных посылок нового состояния системы – это объединение новой информации с явными посылками старого состояния системы за исключением той их части, которая выводится из новой информации. Новое состояние системы подытоживает старое. Из него можно вывести те же следствия, что и из старого, но, возможно, и какие-то добавочные. Оценка полезности посылок, участвующих в выводе, зависит от количества новых следствий, которые эти посылки вводят в рассмотрение или от сложности вывода или от полезности выведенных новых следствий.

В качестве простого примера может рассматриваться ситуация, когда новая информация является индуктивным обобщением существующей информации. Более сложен случай, когда новая информация является абдуктивной посылкой [Peirce 1931]. Допустим, например, что текущее состояние системы уже содержит такую информацию:

(1) A & B & C ← D

(2) A

Тогда новая информация D является *абдуктивной гипотезой.* Вместе с (1) она влечет (2) и, более того, из нее следуют B и C. Для того, чтобы оправдать включение гипотезы D в информационную систему, нужно доказать ее полезность. Это можно сделать, например, если доказать, что B или C уже избыточно содержатся в существующей базе данных или,

*) Более точно: все предыдущие, которые противоречат новому предложению. – *Примеч. Д.А Поспелова*

если известно, что впоследствие надо будет предотвращать их включение в базу данных. Порождение абдуктивных гипотез похоже на рассуждения по умолчанию [Minsky 1975], [Reiter 1976b]. Если A дано, то D может считаться принятым по умолчанию до тех пор, пока оно не приведет к противоречию или не выяснится, что из него нельзя извлечь достаточное количество полезных следствий.

Отметьте, что случаи 1 и 2 могут оказаться адекватными одновременно. Тогда выбор одного из них зависит от общей итоговой полезности получаемой в результате их приложения информационной системы.

3. *Новая информация совместна с текущим состоянием, но не зависит от него.* В этом случае новая информация не может быть ни выведена из текущего состояния системы, ни использована для вывода существующей информации. Такая ситуация является потенциально нежелательной, так как в результате система может начать искать поясняющие гипотезы, из которых вместе с информацией в оставшейся части системы выводится и новая информация. Конечно, гипотезы сами по себе могут быть тоже независимыми, и для оправдания их принятия придется начать искать новые полезные следствия вдобавок к тем, которые мотивировали их порождение. Предыдущий пример иллюстрирует и эту ситуацию. Допустим, что информационная система содержит посылку

$$A \,\&\, B \,\&\, C \leftarrow D$$

а новая информация A независима. Если в результате система вынуждена будет породить гипотезу D, то D само по себе окажется независимым, и от него не будет никакой чистой выгоды, за исключением того, что одновременно будет подтверждена истинность следствий B или C.

Зачастую в течение приемлемого промежутка времени не удается выяснить, в каком из четырех дедуктивных отношений новая информация находится с информационной системой. В таких случаях, независимо от того, связана ли логически новая информация с текущим состоянием информационной системы или нет, эту новую информацию лучше всего считать независимой и включать в систему.

4. *Новая информация несовместна с текущим состоянием информационной системы.* Здесь при добавлении новой информации может быть выведено противоречие. При этом могут быть явно выделены посылки, которые привели к противоречию. Совместность же восстанавливается, если устранить или модифицировать эти, приведшие к противоречию, посылки. Для выбора модифицируемых посылок можно воспользоваться сделанными ранее оценками полезности.

Из четырех перечисленных случаев последний, связанный с появлением противоречия, оказывается особенно важным.

Восстановление совместности

Противоречие и его устранение играют весьма важную роль в философии и в теориях поиска решений. Они составляют движущее начало всякого развития (тезис, антитезис и синтез) в Гегелевской диалектике и выступают в качестве главного инструмента приобретения знаний (им соответствуют догадка и опровержение в [Popper 1963], а также доказательство и контрпример в [Lakatos 1973]) в Поппероской философии науки и

математики. В системах поиска решений противоречие и его устранение лежат в основе методов современного интеллектуального бэктрекинга и входят как важная составная часть в средства поддержания истинности таких систем [Doyle 1978], [Stallman, Sussman 1977].

Одна из главных предпосылок Квайновских [Quine 1953] контраргументов к противопоставлению необходимой и случайной истины состоит в том, что если противоречие возникает, то совместность может быть восстановлена путем устранения или модификации любой посылки, приведшей к возникновению противоречия. На самом деле, ни один факт не защищен от того, что в один прекрасный момент не будет выведено альтернативное к нему положение. Даже законы математики и логики, которые в широком смысле всегда включены в состав посылок информационных систем, являются предметом критического переосмысления и изменения.

Все это, конечно, не значит, что любые факты можно менять с одинаковой легкостью. Психологическая привязанность и даже компьютерные привычки, конечно, могут заставить изменить свое отношение к фактам. Но из этого не следует, что различные факты можно подвергать одинаковой обработке. Ведь некоторые факты привносят в вывод полезных следствий гораздо больший вклад, чем другие; некоторые же чаще участвуют в выводе противоречий. В общем случае, для правильного функционирования информационных систем в целом бывает полезно отказаться не только от фактов, приводящих к противоречию, но и от фактов, которые меньше всего влияют на вывод полезных следствий. В дальнейшем, если противоречия сохраняются, а оценка полезности фактов меняется, то может оказаться необходимым выполнить бэктрекинг, восстановить удаленные факты и отказаться вместо этого от их альтернатив.

Таким образом, вывод несовместности приводит нас к поисковому пространству альтернативных информационных систем. Для каждой посылки участвующей в выводе противоречия, существует, по меньшей мере, одно новое альтернативное состояние системы фактов, полученное путем удаления или приемлемой модификации этой посылки. Поиск в пространстве может быть организован вглубь с применением бэктрекинга в случае возникновения противоречия, а может быть организован и так, что некоторые ветви обрабатываются параллельно. Параллельный просмотр альтернатив обладает тем достоинством, что следствия удаления факта могут быть исследованы до того, как будет принято решение об удалении. Такой параллельный просмотр нескольких внутренне совместных, но вкупе несовместных систем фактов, может, конечно, вызвать у внешнего наблюдателя нежелательную иллюзию единой несовместной системы. Вывод несовместности играет важную роль в развитии программирования и методов управления базами данных. В общем случае, если возникает несовместность между программой и ее спецификацией или между данными и ограничениями целостности, то именно данные и программы подлежат устранению или модификации. Действительно, по определению, основным назначением спецификаций и ограничений целостности является возможность освобождения от некорректных программ и данных. Тем не менее, нередки случаи, когда приходится устранять или модифицировать спецификации или ограничения целостности. Например, если возникнет

конфликт между таким ограничением целостности

Ни одному экипажу не разрешается заезжать в парк

и необходимостью для полиции или иной службы безопасности иметь быстрый доступ в парк, то, скорее всего, преимущество будет отдано полиции и будет модифицировано ограничение целостности, например, так:

Ни одному, не имеющему санкций экипажу, не разрешается заезжать в парк

Преимущество вновь поступающим данным предоставляется также в случаях, когда они являются исключениями из общих правил. Например, пусть некоторые предыдущие версии расписания для университетской кафедры основывались на таких весьма жестких правилах:

Все лекции для первого курса проводятся в комнате 144.
Все лекции, посещаемые более, чем 80 студентами, проводятся в комнате 145

Последующее добавление в базу данных факта:

Лекции для первого курса по логике посещаются 100 студентами

может привести к противоречию. Совместность может быть восстановлена, если посчитать новые данные исключением из общего правила и заменить исходное правило более ограниченным по силе:

Все лекции для первого курса, за исключением логики, проводятся в комнате 144

Отметьте, что в последнем примере изменяется не обязательно та посылка, которая менее всего была полезной в прошлом. В общем же случае должна учитываться не просто полезность факта, но разность между его полезностью и полезностью его замены. Рассмотрение новых данных в качестве исключения из общего правила обладает еще и тем достоинством, что позволяет устранить противоречие, не затрагивая большую часть полезных следствий текущего состояния информационной системы.

Существенна роль противоречий также и в понимании текста. Они помогают устранять двусмысленность предложений путем отказа от таких интерпретаций текста, которые несовместны с текущей интерпретацией уже прочитанной его части, а также помогают отказываться от несовместных гипотез. Если все интерпретации нового предложения приводят к противоречию, то система может попытаться восстановить совместность путем изменения предыдущих гипотез или путем изменения интерпретации предыдущих предложений. Пожалуй, самым классическим случаем, когда информационной системе приходится справляться с противоречием, является ситуация, в которой результат опытного наблюдения противоречит научной теории. Является ли более предпочтительным отказ от результата опыта или от положений теории, зависит от общего воздействия такого решения на информационную систему. Часто бывает так, что несколько альтернатив могут привести к несравнимым, равно очевидным, но вкупе несовместным теориям.

Как утверждает Лакатос [Lakatos 1974], для вполне развитой теории с большой предысторией полезных следствий обычно выгоднее отказаться от аномального результата, чем от всей теории целиком.

Но почти никогда не бывает случая, когда от всей теории надо отказаться полностью. Сложные информационные системы представляют собой собрание взаимодействующих самостоятельных фактов, некоторые из которых более полезны и обладают более высокой оценкой истинности, чем другие*). Высказывания, касающиеся ядра теории, обычно обладают более высокой оценкой истинности, чем те, что относятся к периферийным областям, где конкурирующие гипотезы сосуществуют как совокупно несовместные альтернативы. Результаты эмпирических наблюдений могут помочь в сборе доводов в пользу предпочтения одной альтернативы другой.

Даже если не пытаться восстанавливать совместность, то все же несовместные системы могут содержать полезную информацию. Хотя теоретически из несовместных посылок можно вывести любое заключение, на практике эффективные процедуры доказательства способны выводить только относящиеся к делу заключения с различной степенью доступности. На самом деле, можно показать, что практическая доказуемость, достигнутая при помощи базирующихся на резолюции эффективных процедур доказательства, удовлетворяет всем требованиям, необходимым для вывода релевантных следствий [Anderson, Belnap 1962].

Так противоречие, не вредя информационной системе, помогает установить области, в которых в эту систему может быть привнесено улучшение. Противоречие дает возможность системам развиваться методом последовательной аппроксимации — смелые догадки возникают после опровержения и переустройства. Последовательное применение противоречия приводит к выбору необычных, легко приводимых к противоречию фактов, которые легко могут быть ослаблены, если возникнет нужда; но оно позволяет не затрагивать надежные, "прячущиеся в тени" факты, которые трудно будет усилить потом. Лучше совершать ошибки и исправлять их, чем вообще отказаться от какого бы то ни было прогресса.

Логическая программа для естественного языка

Неплохим тестом для теории информационных систем, развивавшейся в этой главе, может служить программа для управления естественноязыковым интерфейсом логической базы данных, спроектированная автором совместно с Жаклин Шейн и Карен Риччи. Опытная версия программы была разработана на базе системы доказательства теорем для стандартной формы логики Крысей Бродой.

Верхний уровень программы

Обработать (x, y, z, x')

начинает работу исходя из начальной логической базы данных x, обрабатывает список y входных предложений на естественном языке и порождает соотносящийся с y список z выходных предложений, заканчивает сеанс работы, выдавая новую базу данных x'.

*) То есть, нам желательно с бóльшим приоритетом сохранять веру в них. — *Примеч. Д.А. Поспелова*

```
Обработать (бзд, nil, nil, бзд) ←
Обработать (бзд, вход.оствхода, выход. оствыхода,
    новбзд) ←Представляет (вход, логика, управление),
    Принять (бзд, логика, управление, выход,
    промежбзд),
    Обработать (промежбзд, оствхода, оствыхода,
    новбзд)
```

Здесь, как и в предыдущей главе, строки из малых букв (т.е. "бзд", "вход", "оствхода" и т.п.) обозначают переменные.

Представляет (вход, логика, управление) истинно, если в составе входного текста на естественном языке могут быть выделены логическая часть и управляющий компонент.

Принять (бзд, логика, управление, выход, промежбзд) истинно, если принятие логического предложения и связанного с ним управления в логическую базу данных порождает приемлемый выход и новую промежуточную базу данных.

В самом простом случае управление просто отмечает, является ли предложение декларативным или имеет вид вопроса. В программе клауза (A1) обрабатывает случай, когда вход является вопросом. Результатом попытки ответа на этот вопрос может быть, а может и не быть доказательство. (A2) обрабатывает случай, когда вход является декларативным предложением, уже неявно содержащимся в базе данных. В обоих случаях (A1) и (A2) принятие новой информации не меняет базу данных. В случае (A3) новая база данных содержит вместе с остатком (устойчивой частью) существующей базы данных и новую информацию. Из новой базы данных следуют все данные той части (виртуальной части) старой базы данных, которые больше не содержатся явно в новой базе данных. (A4) добавляет в базу данных новую информацию, если она не может быть выведена или использована для выведения существующей информации. (A5) выполняет обработку в случае, когда новая информация несовместна с текущим состоянием базы данных. Новая база данных получается на основе анализа доказательства противоречия и восстановления совместности.

```
(A1)  Принять (бзд, логика, управление, выход, бзд) ←
                    Вопрос (управление),
                    Продемонстрировать (бзд, логика, управление, результат),
                    Извлекаю Ответ (результат, выход)
(A2)  Принять (бзд, логика, управление, выход, бзд) ←
                    Декларативное (управление),
                    Продемонстрировать (бзд, логика, управление,
                    результат),
                    Доказать (результат), ЯУжеЗнаюЧто (выход).
(A3)  Принять (бзд, логика, управление, выход, следбзд) ←
                    Декларативное (управление),
                    бзд = устчсть ∪виртчсть,
                    следбзд = устчсть ∪{логика},
```

∀ данное [данное ∈ виртчсть →
∃ результат [Продемонстрировать (следбзд, данное,
управление, результат) & Доказать
(результат)]],
СпасибоЗаНовостьДляМеня (выход).

(А4) Принять (бзд, логика, управление, выход, следбзд) ←
Декларативное (управление),
Независимое (бзд, логика, управление),
следбзд = бзд ∪{логика},
Познакомилась (выход).

(А5) Принять (бзд, логика, управление, выход, следбзд) ←
Декларативное (управление),
несовм = бзд ∪{логика},
Продемонстрировать (несовм, □, управление,
результат),
Доказать (результат),
АнализируюНеудачуВосстанавливаюСовместность
(несовм, результат, выход, следбзд).

Все вышеприведенное составляет лишь набросок верхнего уровня программы понимания естественного языка. Необходимые процедуры нижнего уровня должны быть заданы особо, а спецификации вроде тех, что имеются в (А3), должны быть преобразованы в эффективные процедуры.

Наше намерение состояло в обработке ограниченного подмножества естественного языка, подходящего для неискушенных пользователей базы данных. Но мы не настаивали на том, чтобы входные предложения были полностью недвусмысленными. Некоторые двусмысленности обрабатывались за счет того, чтобы допускался недетерминизм первого рода для предиката Представляет; другие двусмысленности, вроде тех, что связаны с анафорическими высказываниями ("он", "она", "оно" и т.д.) — за счет того, что для учета контекста предшествующего входного естественноязыкового предложения к отношению Представляет добавлялись экстрапараметры.

Для пользователей, работающих с базой данных, может потребоваться, чтобы вся информация, включаемая в базу данных, представлялась явно. Однако в обычном диалоге или разборе текста нельзя избавиться от неявных посылок, например, когда схемы порождения гипотез (вроде фреймов [Minsky 1975] или сценариев [Shank 1975]) нужны для подгонки предложений к подходящим остовам. Программа обработки естественного языка может быть расширена, по крайней мере теоретически, с целью приобретения возможности абдуктивного порождения посылок за счет добавления новых процедур. Так, в случае, когда вход независим от существующей базы данных, клауза (А6) порождает и добавляет к базе данных новую посылку, которая вместе с оставшейся частью базы данных влечет новую информацию. Чтобы оправдать эти условия, новая информация должна оказаться существенно более полезной, чем сама входная информация.

(А6) Принять (бзд, логика, управление, выход, следбзд) ←
Декларативное (управление),
Независимы (бзд, логика, управление),
следбзд = бзд ∪ {новпосылка},
Продемонстрировать (следбзд, логика, управление, результат),
Доказать (результат),
новпосылка является более полезной в бзд чем логика,
Ядопускаю (новпосылка, выход).

Заключение

Теория информационных систем пытается соединить традиционную роль, которую логика играет в эпистемологии и философии науки с ее новой ролью в информатике. Она пытается примирить приложения не связанной с компьютерами логики с использованием сложных, базирующихся на компьютерах, вычислительных систем, не имеющих логических оснований. Благодаря использованию компьютерной интерпретации логики возникает надежда со временем прийти к более совершенному взаимодействию логики и информации.

СПИСОК ЛИТЕРАТУРЫ

Amarel S. [1966] On Macshine Representations of Problems of Reasoning about Actions – the Missionaries and Cannibals Problem. Machine Intelligence 3, Edinburgh University Press, New York, (B. Meltzer and D. Michie, Eds.) pp. 131–171.

Anderson A.R., Belnap N.D. [1962] The Pure Calculus of Entailment. Journal of Symbolic Logic, Vol. 27 No 1, pp. 19–52.

Anderson D.B., Hayes P.J. [1972], An Arraignment of Theorem–proving or the Logician's Folly. D.C.L. Memo No. 54, University of Edinburgh.

Bergman M., Kanoui H. [1973] Application of Mechanical Theorem Proving to Symbolic Calculus. Third International Symposium on Advanced Computing Methods in Theoretical Physics. C.N.R.S., Marseilles, June 1973.

Bibel W. [1976a] Synthesis of Strategic Definitions and their Control. Tecn. Univ. Munchen, Abt. Mathem. Bericht Nr. 7610.

[1976b] A Uniform Approach to Programming. Techn. Univ. Munchen, Abt. Mathem., Bericht Nr. 763.

[1978] On Strategies for the Synthesis of Algorithms. Proc. AISB/GI Conf. on AI, Hamburg, July 18–20, 1978.

Bledsoe W.W. [1971] Splitting and Reduction Heuristics in Automatic Theorem Proving. Artificial Intelligence 2, pp. 55–77.

[1977] Non–resolution Theorem Proving. Artificial Intelligence, Vol. 9, pp. 51–85.

Bobrow D.G., Raphael B. [1974] New Programming Languages for Artificial Intelligence Research. ACM Computing Surveys, Vol. 6, No. 3, pp. 153–174.

Boyer R.S., Moore J.S. [1972] The Sharing of Structure in Theorem Proving Programs. Machine Intelligence 7, Edinburgh University Press, New York, (B. Meltzer and D. Michie, Eds.) pp. 101–116.

[1975] Proving Theorems about LISP Functions. J. ACM, Vol. 22, No. 1, pp. 129–144.

Brand D. [1976] Analytic Resolution in Theorem Proving. Artificial Intelligence. Vol. 7. pp. 285–318.

Brough D.B. [1979] Loop Trapping in Logic Programs. Dep. Rep. 79/9 Dep. of Computing and Control, Imperial College, London.

Brown F.M. [1973] The use of Several Models as a Refinement of Resolution with Sets of Horn Clauses. Department of Computational Logic, University of Edinburgh, Memo No. 63.

[1974] SLM. Dep. of Comp. Logic Memo No. 72, Univ. of Edinburgh.

[1977] A Theorem prover for Elementary Set Theory. International Joint Conference on Artificial Intelligence, 5.

Bruynooghe M. [1976] An Interpreter for Predicate Logic Programs, Part 1. Report CW 10, Applied Maths and Programming Division, Katholieke Univ., Louven, Belgium.

[1977] The Inheritance of Links in a Connection Graph and its Relation to Structure Sharing. Applied Mathematics and Programming Division, Katholieke Universiteit, Louven, Belgium.

[1978] Intelligent Backtracking for an Interpreter of Horn Clause Logic Programs. Report CW 16. Applied Maths and Programming Division, Katholieke Universiteit, Louven, Belgium. Also in Proc. Colloquium on Mathematical Logic in Programming, Hungary, Sept. 1978.

Bundy A. [1971] There is No Best Proof Procedure. SIGART Newsletter, December 1971. p. 6. [1976] My Experiences with Prolog. DAI Working Paper No. 12, University of Edinburgh. (editor) [1978] Artificial Intelligence, Edinburgh University Press.

Bundy A., Byrd L., Luger G., Mellish C., Milne R., Palmer M. [1979] MECHO: A Program to Solve Mechanics Problems. DAI Working Paper No. 50, Univ. of Edinburgh.

Burstall R.M., Darlington J. [1977] Transformation for Developing Recursive Programs. J. ACM, Vol. 24, No. 1, pp. 44–67.

Chang C.L. [1976] DEDUCE: A Deductive Query Language for Relational Data Bases. Pattern Recognition and Artificial Intelligence (C.H. Chen Ed.), Academic Press, New York. pp. 108–134.

Chang C.L., Lee R.C.T. [1973] Symbolic Logic and Mechanical Theorem Proving. Academic Press, New York. (Русс. пер.: Чень К.Л., Ли Р.К.Т. Математическая логика и автоматическое доказательство теорем / Пер. с англ. под ред. С.Ю. Маслова. – М.: Мир, 1983).

Chomsky N. [1957] Syntactic Structures, Mouton and Co. The Hague. (Русс. пер.: Хомский Н. Синтаксические структуры // Новое в лингвистике. – Вып. 11. – М.: ИЛ, 1962. – С. 412–527.)

Church A. [1936] A Note on the Entscheidungsproblem. Journal of Symbolic Logic. Vol. 1, p. 40–41, Correction ibid., pp. 101–102.

Clark K.L. [1978] Negation as Failure. Logic and Data Bases. (H. Gallaire and J. Minker, Eds.), Plenum Press. New York, pp. 293–322.

Clark K.L., Darlington J. [1978] Algorithm Classification through Synthesis. Computer Journal.

Clark K.L., McKeeman W., Sickel S. [1978] Logic Programming Applied to Numerical Integration. Technical Rep. 78–8–004, University of California. Santa Cruz.

Clark K.L., Tarnlund S.–A. [1977] A First Order Theory of Data and Programs. Proceedings IFIP 77, North Holland, pp. 939–944.

Clark K.L., McCabe F. [1979] Programmer's Guide to IC–Prolog. CCD Rep 79/7, Imperial College, London.

Codd E.F. [1970] A Relational Model for Large Shared Data Bases. CACM Vol. 13, No. 6 (June 1979), pp. 377–387.

[1972] Relational Completeness of Data Base Sublanguages. Data Bases Systems (R. Rustin, Ed.), Prentice–Hall, Englewood Cliffs, N.J., pp. 65–98.

Coelho H., Pereira L. [1975] The Dialectic Development of GEOM, a PROLOG Geometry Theorem Prover. Dept. of Art. Int., Univ. of Edinburgh.

Colmerauer A. [1973] Les systemes–Q ou un Formalisme pour Analyser et Synthetiser des Phrases sur Ordinateur. Publication Interne No. 43, Dept. d'Informatique, Universite de Montreal.

[1977] An Interesting Natural Language Subset. Proc. Workshop on Logic and Data Bases. Toulouse.

[1978] Metamorphosis Grammars. Natural Language Communication with Computers, (L. Bolc., Ed.), Lecture Notes in Computer Science No. 63, Springer–Verlag, Berlin, Heidelberg, New York, pp. 133–189.

Colmerauer A., Kanoui H., Pasero R., Roussel P. [1973] Un Systeme de Communication Homme–machine en Francais. Rapport, Groupe Intelligence Artificielle, Universite D'Aix Marseille, Luminy.

Cox P. [1978] Locating the Source of Unification Failure. Proc. of 2nd National Conf. Canadian Soc. for Computational Studies of Intelligence, Toronto, pp. 20–29.

Cox P., Pietrzykowski T. [1976] A Graphical Deduction System. Department of Computer Science, University of Waterloo, Ontario, Canada.

Dahl V., Sambuc R. [1976] Un Systeme de Bases de Donnees en Logique du Premier Ordre, en Vue de Sa Consultation en Langue Naturelle. Rapport, Groupe d'Intelligence Artificielle, Universite Marseille–Luminy.

Darlington J. [1975] Application of Transformation to Program Synthesis. Proc. IRIA Symp. on Proving and Improving Programs, Arcet–Senans, France, pp. 133–144.

Darlington J., Burstall R.M. [1976] A System that Automaticaly Improves Programs. Acta Informatica, Vol. 6, pp. 41–60.

Darlington J.L. [1969] Theorem Proving and Information Retrieval. Machine Intelligence 4, (B. Meltzer and D. Michie, Eds), American Elsevier Co., New York.

Davies J. [1973] Popler 1.5 Reference Manual. T.P.U. Report No. 1, University of Edinburgh, May 1973.

Dawson C., Siklossy L. [1977] The Role of Preprocessing in Problem–solving Systems. International Joint Conference on Artificial Intelligence, 5, p. 465–471.

Deliyanni A., Kowalski R.A. [1979] Logic and Semantic Networks. Comm. ACM. Vol. 22, No 3, pp. 184–192.

Derkson J., Rulifson J.P., Waldinger R.J. [1972] QA4 Language Applied to Robot Planning. AFIPS Fall Computer Conference, pp. 1181–1192.

Dijkstra E.W. [1976] A Discipline of Programming. Prentice–Hall, Englewood–Cliffs, New Jersey. (Русс. пер.: Дейкстра Э. Дисциплина программирования. – М.: Мир, 1979)

Doran J.E., Michie D. [1966] Experiments with the Graph Traverser Program. Proc. R. Soc. A. Vol. 294 pp. 235–259.

Doyle J. [1978] Truth Maintenance Systems for Problem Solving. TR–419, MIT AI Lab., Also IJCAI 5, p. 247.

Earley J. [1970] An Efficient Context–free Parsing Algorithm. CACM, pp. 94–102.

Elcock E.W., Foster J.M., Gray P.M.D., McGregor J.J., Murray A.M. [1971], ABSET, a Programming Language Based on Sets: Motivation and Examples. Machine Intelligence 6, Edinburgh University Press, New York, (B. Meltzer and D. Michie, Eds), pp. 467–492.

Ernst G.W. [1971] The Utility of Independent Subgoals in Theorem Proving. Information and Control, April 1971.

Feldman J.A., Low J.R., Swinehart D.C., Taylor R.H. [1972] Recent Developments in SAIL – an Algol–based Language for Artificial Intelligence. IJCAI 5 pp. 225–246.

Fikes R.E., Handrix G.G. [1975] A Network–based Knowledge Representation and its Natural Deduction System. SRI Memo.

Fikes R.E., Nilsson N.J. [1971] STRIPS: A New Approach to the Application of Theorem–proving to Problem Solving. Artificial Intelligence Vol. 2, pp.189–208.

Fillmore C.J. [1968] The Case for Case. Universals in Linguistic Theory, (Bach and Harms, Eds.), Holt, Rinechart and Winston, Chicago.

Fishman D.H., Minker J. [1975] Pi–Representation. A Clause Representation for Parallel Search. Artificial Intelligence, Vol. 6, No. 2, pp. 103–127.

Floyd R.W. [1967] Assigning Meanings to Programs. Proc. Symposia in Applied Mathematics, Vol. 19, American Maths Society, pp. 19–32.

Foster J.M. [1970] Automatic Syntactic Analysis. Macdonald/Elsevier.

Foster J.M., Elcock E.W. [1969] ABSYS 1: An Incremental Compiler for Assertions. Machine Intelligence 4, Edinburgh University Press, New York, (B. Meltzer and D. Michie, Eds.), pp. 423–439.

Friedman D.P., Wise D.S. [1978] Aspects of Applicative Programming for Parallel Processing. IEEE Trans Comp. C–27 (April 78), pp. 289–296.

Futo I., Darvas F., Cholnocky E. [1977] Practical Application of an AI Language 2. Proceedings of the Hungarian Conference of Computing. Budapest. pp. 385–400.

Futo I., Darvas F., Szeredi P. [1978] The Application of PROLOG to the Development of QA and DBM Systems. Logic and Data Bases. (H. Gallaire and J. Minker, Eds.), Plenum Press, New York, pp. 347–375.

Gallaire H., Minker J. (Editors) [1978] Logic and Data Bases. Plenum Press, New York.

Gallaire H., Minker J., Nicolas J.-M. [1978] An Overview and Introduction to Logic and Data Bases. Logic and Data Bases (H. Gallaire and J. Minker, Eds.), Plenum Press, New York 1978. pp. 3–30.

Gelernter H. [1963] Realisation of a Geometry–Theorem Proving Machine. Reproduced in Computers and Thought, (Feigenbaum and Feldman, Eds.), McGraw Hill, New York, pp. 134–152.

Gilmore P.G. [1977] Defining and Computing Many–valued Functions. Parallel Computers – Parallel Mathematics. (M. Feilmeier, ed.) pp. 17–23.

Godel K. [1931] Über Formal Unentshcheidbare Sätze der Principia Mathematica und verwandter System 1. Monatshefte für Mathematik und Physic 38, pp. 173–198.
English Translation // From Frege to Gödel: A Sourcebook in Mathematical Logic 1879–1931. (Ed. by van Heijenoort), Harvard University Press, Cambridge, Mass., pp. 596–616.

Golomb S., Baumert L. [1965] Backtrack Programming. J. ACM Vol. 12, pp. 516–524.

Green C.C. [1969a] Theorem Proving by Resolution as a Basis for Question–Answering Systems. Machine Intelligence 4, Edinburgh University Press, New York, (B. Meltzer and D. Michie, Eds.), pp. 183–205.
[1969b] Application of Theorem–Proving to Problem Solving. Proc. Int. Joint Conf. on AI, Washington, DC. (D.E. Walker and L.M. Norton, Eds.), pp. 219–240.

Hayes P.J. [1973] Computation and Deduction. Proc. 2nd MFCS Symp. Czechoslovak Academy of Sciences, pp. 105–118.
[1977] In Defense of Logic. International Joint Conference on Artificial Intelligence, 5, pp. 559–565.

Henderson P., Morris J. [1976] A Lazy Evaluator. 3rd Symp. on principles of programming languages. Atlanta. pp. 95–103.

Hendrix G.G. [1975] Expanding the Utility of Semantic Networks through Partitioning. IJCAI 4, Tbilisi, Georgia. pp. 115–121.

Herbrand J. [1930] Recherches sur la Theorie de la Demonstration.Travaux de la Societe des Sciences et des Lettres de Varsovie, Classe III, Science Mathematique et Physique, No. 33.

Hewitt C. [1969] PLANNER: A Language for Proving Theorems in Robots. Proc. IJCAI, Washington, D.C., pp. 295–301.
[1975] How to use what you know. Proc. IJCAI, Tbilisi, Georgia, pp. 189–198.

Hill R. [1974] LUSH Resolution and its Completeness. DCL Memo No. 78, University of Edinburgh, School of Artificial Intelligence, August 1974.

Hoare C.A.R. [1961] Algorithm 64. CACM, Vol. 4, pp. 321.
[1969] An Axiomatic Basis for Computer Programming. CACM. Vol. 12, No. 10, pp. 576–583.
[1972] Proof of the Correctness of Data Representation. Acta Informatica 1. pp. 271–281.

Hodges W. [1977] Logic. Penguin Books, Middlesex, England.

Hogger C.J. [1978a] Goal Oriented Derivation of Logic Programs. Proc. MFCS Conf., Polish Academy of Sciences, Zakopane.
[1978b] Program Synthesis in Predicate Logic. Proc. AISB/GI Conf. on AI, Hamburg, July, pp. 18–20.
[1979] Derivation of Logic Programs. Ph. D. Thesis, Imperial College.

Horn A. [1951] On Sentences which are True of Direct Unions of Algebras. Journal of Symbolic Logic, 16, pp. 14–21.

Kellog C., Klahr P., Travis L. [1978] Deductive Planning and Pathfinding for Relational Data Bases. Logic and Data Bases, (H. Gallaire and J. Minker, Eds.), Plenum Press, New York, pp. 179–200.

Kowalski R.A. [1969] Search Strategies for Theorem–proving. Machine Intelligence 5, Edinburgh University Press, New York, (B. Meltzer and D. Michie, Eds.), pp. 181–201.
[1972] And–Or Graphs, Theorem Proving Graphs and Bi–directional Searh. Machine Intelligence 7, Edinburgh University Press, New York, (B. Meltzer and D. Michie, Eds.), pp. 167–194.
[1974a] A Proof Procedure Using Connection Graphs. J. ACM 22, pp. 572–595.
[1974b] Logic for Problem Solving. Memo No. 75, Dept. of Computational Logic, University of Edinburgh.
[1974 c] Predicate Logic as Programming Language. Proc. IFIP 74, North Holland Publishing Co., Amsterdam, pp. 569–574.
[1978] Logic for Data Description. Logic and Data Bases. (H. Gallaire and J. Minker, Eds.), Plenum Press, New York, pp. 77–102.
[1979] Algorithm = Logic + Control. CACM, August 1979.

Kowalski R.A., Hayes P.J. [1968] Semantic Trees in Automatic Theorem proving. Machine Intelligence 4, (B. Meltzer and D. Michie, Eds), Einburgh University Press, pp.87–101. (Русс. пер.: Ковальский Р., Хейес П.Дж. Семантические деревья в автоматическом поиске доказательств//Киберн. сб. – Нов. сер. № 9. – М.: Мир, 1972)

Kowalski R.A., Kuehner D. [1971] Linear Resolution with Selection Function. Artificial Intelligence, Vol 2, p. 227–260.

Kuehner D. [1972] Some Special Purpose Resolution Systems. Machine Intelligence 7, Edinburgh University Press, New York, (B. Meltzer and D. Michie, Eds.), pp. 117–128.

Lakatos I. [1963] Proofs and Refutations. British Journal for the Philosopy of Science, vol. 14, pp. 1–25, 120–139, 221–243, 296–342.
[1974] History of Science and its Rational Reconstructions. The Interaction between Science and Philosophy. (Y. Elkana, Ed.), Humanities Press, Atlantic Heights, N.J., pp. 195–241.
Lawler E., Wood D. Branch and Bound Methods: A Survey. Oper. Res. Vol. 14, No. 4, pp. 699–719.
Lee R.C.T., Waldinger R.J. [1969] PROW: A Step Toward Automatic Program Writing. Proc. IJCAI, Washington, D.C.
Loveland D.W. [1968] Mechanical Theorem Proving by Model Elimination. JACM 15, April 1968, pp. 236–251.
[1969] A Simplified Format for the Model Elimination Procedure. J. ACM, July 1969, pp. 349–363.
[1970] A Linear Format for Resolution. Symposium on Automatic Demonstration, Lecture Notes in Math 125, Springer–Verlag, Berlin, p. 147–162.
[1972] A Unifying View of Some Linear Herbrand Procedures. JACM 19, (April 1972). pp. 366–384.
[1978] Automated Theorem Proving: A Logical Basis. North Holland Publishing Co., Amsterdam, New York and Oxford.
Loveland D.W., Stickel M.E. [1973] A Hole in Goal Trees: Some Guidance from Resolution Theory. Reproduced in IEEE Trans on Computers, C–25, April 1976, p. 335–341.
Luckham D. [1970] Refinement Theorems in Resolution Theory. Symp. on Automatic Demonstration, Lecture Notes in Math 125, Springer–Verlag, Berlin, pp. 163–190.
Manna Z. [1969] The Correctness of Programs. J. Computing and System Science, Vol. 3, p. 119–127.
Manna Z., Waldinger R.J. [1975] Knowledge and Reasoning in Program Synthesis. Artificial Intelligence Journal, Vol. 6, No. 2., pp. 175–208.
[1977] The Automatic Synthesis of Systems of Recursive Programs. Proc. IJCAI Conf. pp. 405–411.
[1978] A Framework for Deductive Programming. Computer Science Dept, Stanford Univ. and SRI International.
Markusz Z. [1977] How to Design Variants of Flats Using the Programming Language PROLOG Based on Mathematical Logic. Proc IFIP 77 North Holland, Amsterdam, pp. 885–889.
Martelli A., Montanari U. [1977] Theorem Proving with Structure Sharing and Efficient Unification. International Joint Conference on Artificial Intelligence 5, pp. 543.
McCarthy J. [1963] A Basis for a Mathematical Theory of Computation. Computer Programming and Formal Systems, (P. Brafford and D. Hirschberg, Eds.), North Holland, Amsterdam, pp. 33–70.
[1968a] Programs with Common Sense. Semantic Information Processing, (M.Minsky, Ed.), MIT Press, Cambridge, Mass., pp. 403–418.
[1968b] Situations, Actions and Causal Laws. Semantic Information Processing, (M. Minsky, Ed.) MIT Press, Cambridge, Mass., pp. 410–417.
McCarthy J., Abrahams P.W., Edwards D.J., Hart T.P., Levin M.I. [1962] LISP Programmers Manual. MIT Press, Cambridge, Mass.
McCarthy J., Hayes P.J. [1969] Some Philosophical Problems from the Standpoint of Artificial Intelligence. Machine Intelligence 4, Edinburgh University Press, New York, (B. Meltzer and D. Michie, Eds.), pp. 463–502.
McDermott D., Doyle J. [1978] Non–monotonic Logic 1. AI Memo 486, August 1978, AI Lab., MIT.
McDermott D., Sussman G.J. [1972] The CONNIVER Reference Manual. AI Memo No. 259, MIT, Project MAC.
McSkimin J.R., Minker J. [1977]. The Use of a Semantic Network in a Deductive Question–Answering System. International Joint Conference on Artificial Intelligence, 5, pp. 50–58.
Meltzer B. [1966] Theorem Proving for Computers: Some Results on Resolution and Renaming. Computing Journal 8, (January 1966), pp. 341–343.
[1972] The Impossibility of Perfect Proof Procedures. AISB European Newsletter, Issue 15, Nov. 1973, pp. 28–29.

Michie D., Ross R., Shannon G.J. [1927] G–Deduction. Machine Intelligence 7, Edinburgh University Press, New York, (B. Meltzer and D. Michie, Eds.), pp. 141–165.
Minker J. [1975] Performing Inferences over Relational Data Bases. Proceedings of 1975 ACM SIGMOD International Conference on Management of Data, pp. 79–91.
Minker J., Fishman D.H. and McSkimin J.R. [1973] The Q* Algorithm – a Search Strategy for a Deductive Question–answering system. Artificial Intelligence, Vol. 4, pp. 225–243.
Minsky M.L. [1968] Descriptive Languges and Problem Solving. Semantic Information Processing, (M. Minsky, Ed.), MIT Press. Cambridge, Mass., pp. 413–424.
[1975] A Framework for the Representation of Knowledge. The Psychology of Computer Vision, (P. Winston, Ed.), McGraw Hill., New York, pp. 211–280. (Русс. пер.: Минский М., Структура для представления знаний // Психология машинного зрения. – М.: Мир, 1978. – С. 249–338)
Moore R.C. [1975] Reasoning from Incomplete Knowledge in a Procedural Deduction System. Memo A1–TR–347, Artificial Intelligence Lab., MIT.
Moss C.D.S. [1977] A Comparison of Hoare's Axiomatic Approach to Semantics and Plan Formation Studies. Imperial College, Dept. of Computing and Control, M.Sc. Thesis.
[1979] A New Grammar for Algol 68. Dep. Rep. 79/6, Imperial College, London.
Murray N. [1978] A Proof Procedure for Non–Clausal First Order Logic. Research Report. University of Syracuse, New York.
Mylopulos J., Cohen P., Borgida A. and Sugar L. [1975] Semantic Networks and the Generation of Context, 4th IJCAI, Tbilisi, Georgia. pp. 134–142.
Nevins A.J. [1974] A Human–Oriented Logic for Automatic Theorem Proving. JACM, Vol. 21, pp. 606–621.
Newell A., Shaw J.C., Simon H.A. [1963] Empirical Explorations with the Logic Theory Machine: A Case Study in Heuristics. Reproduced in Computer and Thought, (Feigenbaum and Feldman, Eds.), McGraw Hill, New York, pp. 109–133.
Newell A., Simon H. [1963] GPS, A Program that Simulates Human Thought. Reproduced in Computers and Thought, (Feigenbaum and Feldman, Eds.), McGraw Hill, New York, pp. 279–296. (Русс. пер.: Ньюэлл А., Саймон Г. GPS – программа, моделирующая процесс человеческого мышления//Вычислительные машины и мышление / Под ред. Э. Фейгенбаума и Дж. Фельдмана. – М.: Мир, 1967. – С. 283–301.)
Nicolas J.–M., Gallaire H. [1978] Data Base: Theory vs. Interpretation. Logic and Data Bases, (H. Gallaire and J. Minker Eds.), Plenum Press, New York, pp. 33–54.
Nicolas J.–M., Syre J.C. [1974] Natural Question–answering and Automatic Deduction in the System SYNTEX. Proceedings IFIP Congress 1974, North Holland, Amsterdam, pp. 595–599.
Nilsson N.J. [1971] Problem Solving Methods in Artificial Inteligence. McGraw Hill, New York. (Русс. пер.: Нильсон Н. Искусственный интеллект. Методы поиска решений / Пер. с англ. – М.: Мир, 1973)
Paterson M.S., Wegman M.N. [1976] Linear Unification. Proc. 8th Annual ACM Symp. on Theory of Computing, pp. 181–186.
Peirce C.S. [1931] Collected Papers of Charles Sunders Peirce. Vol. 2, 1931–1958, (C. Hartshorn et al., Eds.), Harvard University Press, Cambridge, Mass.
Pereira F., Warren D.H.D. [1978] Definite Clause Grammars Compared with Augmented Transition Networks. Research Report, Dept. of AI, Edinburgh.
Pereira L.M., Monteiro L.F. [1978] The Semantics of Parallelism and Coroutining in Logic Programming. Colloquium on Mathematical Logic in Programming, Salgo'tarjan, Hungary.
Pirotte A. [1978] High Level Data Base Query Languages. Logic and Data Bases, (H. Gallaire and J. Minker, Eds.), Plenum Press, New York, pp. 409–436.
Pohl I. [1970] Heurist Search Viewed as Pathfinding in a Graph. Artificial Intelligence Vol. 1, pp. 193–204.
[1972] Bi–directional search. Machine Intelligence 7, Edinburgh University Press, New York, (B. Meltzer and D. Michie, Eds.), pp. 127–140.
Pople H. [1973] On The Mechanisation of Abductive Logic, Proc. IJCAI 3, pp. 387–419.
Popper K.R. [1963] Conjectures and Refutations; The Growth of Scientific Knowledge. Rouledge and Kegan Paul, London.

Pratt V.R. [1977] The Competence / Performance Dichotomy in Programming 4th ACM SIGACT/SIPLAN Symp. on Principles of Programming Languages, Santa Monica, California, pp. 194–200.

Prawitz D. [1960] An Improved Proof Procedure. Theoria 26, pp. 102–139.

Quillian M.R. [1968] Semantic Memory. Semantic Information Processing. (Minsky M., Ed.), MIT Press, Cambridge, Mass., pp. 227–270.

Quine W.V.O. [1941, Revised 1965] Elementary Logic. Harper and Row. New York.

[1953] Two Dogmas of Empiricism // "From a Logical Point of View". Hutchinson. London.

Quine W.V.O., Ullian J.S. [1978] The Web of Belief, 2nd Edition, Random House, New York.

Raphael B. [1971] The Frame Problem in Problem Solving Systems. Artificial Intelligence and Heuristics Programming. (Findler N.V., Meltzer B., Eds.) Edinburgh University Press, Edinburgh, pp. 159–169.

Reboh R., Sacerdoti E. [1973] A Preliminary Qlisp Manuel. Technical Note 81. SRI Project 8721.

Reiter R. [1971] Two Results on Ordering for Resolution with Merging and Linear format. J. ACM 18 (October 1971), pp. 630–646.

[1972] The Use of Models in Automatic Theorem Proving. Technical Report 72–09, Dept. of Computer Science, University of British Columbia.

[1978a] Deductive Questioning–Answering on Relational Data Bases. Logic and Data Bases (H. Gallaire and J. Minker, Eds.), Plenum Press, New York, pp. 149–177.

[1978b] On Reasoning by Default. Proc. 2nd Symp. on Theoretical Issues in Natural Language Processing. Urbana, Illinois.

[1978c] On Closed World Data Bases. Logic and Data Bases (H. Gallaire and J. Minker, Eds.), Plenum Press, New York, pp. 55–76.

Robinson J.A. [1965a] A Machine Oriented Logic Based on the Resolution Principle. J. ACM 12 (January 1965), pp. 23–41. (Русс. пер.: Робинсон Дж. Машинно–ориентированная логика, основанная на принципе резолюции // Киберн. сб., Нов. сер. № 7. – М.: Мир, 1970)

[1965b] Automatic Deduction with Hyper–Resolution. Intern. Journal of Computer Math. 1, pp. 227–234.

[1967] A Review of Automatic Theorem–Proving. Annual Symposia in Applied Math. XIX, American Math. Society, Prov[illegible]e, pp. 1–18.

[1968] The Generalised Resolution Principle. Machine Intelligence 3, (Dale and Michie, Eds.), Oliver and Boyd, Edinburgh 1968, pp. 77–93.

[1971] Computational Logic: The Unification Computation. Machine Intelligence 6, Edinburgh University Press, New York, (B. Meltzer and D. Michie, Eds.), pp. 63–72.

[1979] Logic: Form and Function. Edinburgh University Press.

Robinson J.A., Sibert E.E. [1978] Logic Programming in LISP: A Progress Report. School of Computer and Information Science, Syracuse University.

Roussel P. [1975] PROLOG: Manuel de Reference et d'Utilisation. Groupe d'Intelligence Artificielle, Universite d'Aix–Marseille, Luminy, Sept, 1975.

Rulifson J.F., Derekson J.A.C., Waldinger R.J. [1973] QA4: A Procedural Calculus for Intuitive Reasoning. Technical Note 73, Artificial Intelligence Center, SRI.

Sacerdoti E.D. [1975] The Non–linear Nature of Plans. Proc. IJCAI 4, Tbilisi, Georgia, USSR, pp. 206–214.

[1977] A Structure for Plans and Behaviour. Elsevier North Holland, New York.

Schank R.S. [1973] Identification of Conceptualizations Underlying Natural Language. Computer Models of Thought and Language. (R.C. Schank and K. Colby, Eds.) W.H. Freeman and Co., San Francisko, pp. 187–247.

[1975] Conceptual Information Processing. North Holland Publishing Co., Amsterdam. American Elsevier Publishing Co., New York. (Русс. пер.: Шенк Р. Обработка концептуальной информации. – М.: Энергия, 1980)

Schmidt C.F., Sridharan N.S., Goodson J.L. [1978] The Plan Recognition Problem: An Intersection of Psychology and Artificial Intelligence. Artificial Intelligence, Vol. 11, Nos. 1,2. Aug. 1978, pp. 45–83.

Schubert L.K. [1976] Extending the Expressive Power of Semantic Networks. Artificial Intelligence, Vol. 7, pp. 163–198.

[1977] Inference on Quantified Semantic Networks. Tech. Rep. NL32, University of Texas, February 1977.

Schwartz J. [1977] Using Annotations to Make Recursion Equations Behave. Research Memo, Dept. of Artificial Intelligence, University of Edinburgh.

Shapiro S.C. [1971] A Net Structure for Semantic Information Storage, Deduction and Retrieval. Proc. IJCAI, The British Computer Society, London, pp. 512–523.

[1977] Representing and Locating rules in a Semantic Network. Proc. of the Workshop on Pattern–directed Inference Systems. ACM/SIGART Newsletter No. 63, pp. 14–18.

Sickel S. [1976] A Search Techniques for Clause Interconnectivity Graphs. IEEE Transactions on Computers, Special Issue on Automatic Theorem Proving, C–25, 8, August 1978, pp. 823–835.

[1978] Invertibility of Logic Programs. Technical Rep. 78–8–005, Univ. of California, Santa Cruz.

Siekmann J., Stephan W. [1976] Completeness and Soundness of the Connection Graph Proof Procedure. Interner Bericht Nr. 7/76, Inst. fur Infromatik I, Universitat Karlsruhe.

Simmons R.F. [1973] Semantic Networks: Their Computation and Use for Understanding English Sentences. Computer Models of Thought and Language, (Schank R.C. and Colby K., Eds.), W.H. Freeman and Co., San Francisco, p. 63–113.

Simmons R.F., Chester D. [1977] Inference in Quantified Semantic Networks. International Joint Conference on Artificial Intelligence, 5, p. 267.

Stallman R.M., Sussman G.J. [1977] Forward Reasoning and Dependency-directed Backtracking in a System for Computer–aided Circuit Analysis. Artificial Intelligence, Vol. 9, No. 2, pp. 135–196.

Sussman G.J. [1975] A Computer Model of Skill Acquisition. American Elsevier Publishing Co., Amsterdam.

Sussman G.J., McDermott D.V. [1972a] Why Conniving is Better than Planning. AI Memo No. 255, MIT Project Mac, April 1972.

[1972b] From PLANNER to CONNIVER – a Genetic Approach. AFIPS Fall Joint Computer Conf, pp. 1171–1179.

Sussman G.J., Winograd T., Charniak E. [1971] MICRO–PLANNER Reference Manual. AI Memo 203a, AI Lab, MIT

Tarnlund S.-A. [1975a] An Interpreter for the Programming Language Predicate Logic. Proc. IJCAI, Tbilisi, pp. 601–608.

[1975b] Logic Information Processing. TRITA–IBADB 1034, Department of Information Processing and Computer Science, The Royal Institute of Technology and The University of Stockholm, Sweden.

[1976] A Logical Basis for Data Bases. TRITA–IBADB 1029, Dept. of Computer Science, Royal Institute of Technology, Stockholm.

[1977] Horn Clause Computability. BIT 17, 2, pp. 215–226.

Tate A. [1974] INTERPLAN: A Plan Generation System that Can Deal with Interactions between Goals. Memo MIP–R–109, Machine Intelligence Research Unit, University of Edinburgh.

Van der Brug G.J., Minker J. [1975] State Space, Problem Reduction and Theorem Proving – Some Relationships. C. ACM 18 (February 1975), pp. 107–115.

Van Emden M.H. [1976] Verification Conditions as Representations for Programs. Proc. Third Int. Col. on Automata, Languages and Programming, Edinburgh University Press, pp. 99–119.

[1977] Programming in Resolution Logic. Machine Intelligence 8, pp. 266–299.

[1978] Computation and Deductive Information Retrieval. Formal Description of Programming Concepts, (E. Neuchold, Ed.), North Holland, pp. 421–440.

Van Emden M.H., Kowalski R.A. [1976] The Semantic of Predicate Logic as a Programming Language. J. ACM, Vol. 23, No 4, pp. 733–742.

Waldinger R. [1977] Achieving Several Goals Simultaneously. Machine Intelligence 8, (Elcock E.W. and Michie D., Eds.), Ellis Horwood Ltd. and John Wiley. pp. 94–136.

Warren D.H.D. [1974] WARPLAN: A System for Generating Plans. DCL Memo 76, Dept. of Artificial Intelligence, University of Edinburgh.

[1976] Generating Conditional Plans and Programs. Proc. AISB Summer Conference, Edinburgh, pp. 344–354.

[1977a] Implementing Prolog. Res. Rep. 39, 40. Dept. of A.I., Univ. of Edinburgh.

[1977b] Logic Programming and Compiler Writing. Research Rep. 44, Dep. of A.I., Univ. of Edinburgh.

Warren D.H.D., Pereira L.M., Pereira F. [1977] PROLOG – The Language and its Implementation Compared with LISP. Proc. Symp. on AI and Programming Languages, SIGPLAN Notices, Vol. 12, No. 8, and SIGART Newsletters No. 64, August 1977, pp. 109–115.

Welham R. [1976] Geometry Problem Solving. DAI Research Report No. 14. University of Edinburgh.

Weyhrauch P. [1978] Prolegomena to a Theory of Formal Reasoning. Report AIM–315, Computer Science Department, Stanford University.

Winograd T. [1972] Understanding Natural Language. Academic Press. (Русс. пер.: Виноград Т. Программа, понимающая естественный язык. – М.: Мир, 1976)

[1975] Frame Representation and the Declarative–procedural Controvercy. Representation and Understanding, (D. Bobrow and A. Collins, Eds.), Academic Press.

Winston P.H. [1977] Artificial Intelligence. Addison–Wesley, Reading, Mass.

Wong H.K.T., Mylopoulos J. [1977] Two Views of Data Semantics: A Survey of Data Models in Artificial Intelligence and Database Management. INFOR, Vol. 15, No. 3.

Woods W.A. [1975] What's in a link – Foundations for Semantic Networks. Representation and Understanding, (D. Bobrow and A. Collins, Eds.), Academic Press, New York, pp. 35–582.

Yates R., Raphael B., Hart T. [1970] Resolution Graphs. Artificial Intelligence 1, (Winster 1970), pp. 257–289.

Zloof M.M. [1975] Query–by–Example. Proceedings AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., pp. 431-448.

Zloof M.M., de Long S.P. [1977] The System for Business Automation (SBA): Programming Language. CACM Vol 20, No. 6 (June 1977), pp. 385–396.

ИМЕННОЙ УКАЗАТЕЛЬ

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Научное издание

КОВАЛЬСКИ Роберт (Великобритания)

ЛОГИКА В РЕШЕНИИ ПРОБЛЕМ

"Проблемы искусственного интеллекта", выпуск 18

Редактор *Т.В. Шароватова*
Художественный редактор *Т.Н. Кольченко*
Технические редакторы *О.Б. Черняк, С.Н. Баронина*
Корректоры *Н.П. Круглова, Т.А. Печко*

Набор осуществлен в издательстве
на наборно-печатающих автоматах

ИБ № 32752

Сдано в набор 11.06.90. Подписано к печати 13.09.90
Формат 60 X 90/16. Бумага книжно-журнальная
Гарнитура Пресс-Роман. Печать офсетная
Усл.печ.л. 17,50. Усл.кр.-отт. 17,50. Уч.-изд.л. 17,96
Тираж 10700 экз. Тип.зак. 304. Цена 4 руб.

Ордена Трудового Красного Знамени
издательство "Наука"
Главная редакция физико-математической литературы
117071 Москва В-71, Ленинский проспект, 15

Четвертая типография издательства "Наука"
630077 г. Новосибирск-77, ул. Станиславского, 25

ИЗДАТЕЛЬСТВО "НАУКА"

ГЛАВНАЯ РЕДАКЦИЯ
ФИЗИКО-МАТЕМАТИЧЕСКОЙ ЛИТЕРАТУРЫ

СЕРИЯ "ПРОБЛЕМЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА"

Вышли и выйдут в свет:

1. *Ефимов Е.И.* **Решатели интеллектуальных задач.** – 1982. – 316 с.
2. *Попов Э.В.* **Общение с ЭВМ на естественном языке.** – 1982. – 320 с.
3. *Зарипов Р.Х.* **Машинный поиск вариантов при моделировании творческого прогресса.** – 1983. – 232 с.
4. *Гаек П., Гавранек Т.* **Автоматическое образование гипотез: математические основы общей теории.** – 1984. – 280 с.
5. *Тыугу Э.Х.* **Концептуальное программирование.** – 1984. – 256 с.
6. *Гладкий А.В.* **Синтаксические структуры естественного языка в автоматизированных системах общения.** – 1985. – 144 с.
7. *Поспелов Д.А.* **Ситуационное управление: теория и практика.** – 1986. – 285 с.
8. **Нечеткие множества в моделях управления и искусственного интеллекта** / Под. ред. Д.А. Поспелова. – 1986. – 312 с.
9. **Моделирование языковой деятельности в интеллектуальных системах** / Под ред. А.Е. Кибрика, А.С. Нариньяни; С предисловием А.П. Ершова. – 1987. – 280 с.
10. *Попов Э.В.* **Экспертные системы: решение неформальных задач в диалоге с ЭВМ.** – 1987. – 285 с.
11. *Левин Д.Я.* **Инструментальный комплекс программирования на основе языка высокого уровня.** – 1987. – 197 с.
12. *Вагин В.Н.* **Дедукция и обобщение в системах принятия решений.** – 1988. – 384 с.
13. *Кандрашина Е.Ю., Литвинцева Л.В., Поспелов Д.А.* **Представление знаний о времени и пространстве в интеллектуальных системах.** – 1989.
14. *Цаленко М.Ш.* **Моделирование семантики в базах данных.** – 1989. – 286 с.
15. *Рубашкин В.Ш.* **Представление и анализ смысла в интеллектуальных информационных системах.** – 1989. – 189 с.
16. *Кузнецов В.Е.* **Представление в ЭВМ неформальных процедур.** – 1989.
17. *Любарский Ю.Я.* **Интеллектуальные информационные системы.** – 1990.
18. *Ковальски Р.* **Логика в решении проблем.** – 1990.