МАТЕМАТИЧЕСКОЕ ОБЕСПЕЧЕНИЕ ЭВМ

ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ
Сборник статей

Перевод с английского
и французского

под редакцией
В. Н. Агафонова

Москва «Мир» 1988

ББК 32.973
Л69
УДК 681.3

Логическое программирование: Пер. с англ. и фр.—
Л69 М.: Мир, 1988. — 368 с., ил.
ISBN 5-03-000972-8

Сборник работ зарубежных специалистов, отражающий современное состояние в новом направлении программирования, тесно связанном с математической логикой. Это направление активно развивается как в теоретическом, так и в практическом плане, включая в себя новые языки (Пролог, Логлисп и др.), методы реализации и проекты машинных архитектур. В сборник включен специально написанный обзор литературы по логическому программированию. Среди авторов известные зарубежные специалисты: Б. Домелки, П. Середи (ВНР), А. Колмероэ (Франция), Дж. Робинсон (США), Р. Ковальский (Великобритания).

Для математиков-прикладников, программистов, аспирантов и студентов университетов.

Л $\frac{1702070000-410}{041(01)-88}$ 36—88, ч. 1

ББК 32.973

Редакция литературы по математическим наукам

ISBN 5-03-000972-8

ПРЕДИСЛОВИЕ

В последние годы внимание специалистов в области программирования, информатики и вычислительной техники все больше привлекает новое направление, возникшее на стыке математической логики и программирования и получившее название логического программирования. В русле этого направления появились новые языки, техника реализации и проекты машинных архитектур. Как наиболее известные примеры можно назвать созданный А. Колмероэ во Франции язык Пролог, эдинбургскую реализацию Пролога, выполненную Д. Уорреном, и японский национальный проект вычислительных систем пятого поколения.

Идейные корни логического программирования лежат в математической логике, где усилиями нескольких поколений математиков всего мира были созданы методы формального описания задач с помощью логических формул и методы формальных доказательств (выводов, дедукций), позволяющие достаточно удобно описывать функции и отношения и автоматически получать по этим описаниям решения с приемлемой степенью эффективности. Соответствующая теоретическая работа велась и у нас в СССР, но недостаточно интенсивно и не всегда доводилась до приемлемых для вычислительной практики результатов.

Настоящий сборник призван содействовать осознанию этого нового явления в информатике и освоению накопленного в мире передового опыта. Литература по логическому программированию велика, но плохо доступна и мало известна широким кругам советских специалистов в области программирования и информатики. Сборник задуман как своего рода хрестоматия, в которую отобраны не самые новые, но представительные работы, фиксирующие важные этапы развития логического программирования. Они дают информацию из первых рук о главных аспектах логического программирования — теоретических основах, языках, реализациях, применениях (не представлена только тема машинной архитектуры, которая ближе к вычислительной технике и заслуживает осо-

бого рассмотрения). Половина статей взята из сборника Logic Programming (L.: Academic Press, 1982), который стал, можно сказать, наиболее цитируемым источником и настольной книгой специалистов по логическому программированию.

В открывающих сборник статьях Дж. Робинсона, А. Колмероэ и Р. Ковальского, которых уже считают классиками логического программирования, дается живая картина истории и развития идей логического программирования, включая язык Пролог. Статья Б. Домёлки и П. Середи посвящена применениям Пролога, а статья М. Бранохе — его реализации. В трех из последующих статей рассматриваются другие языки логического программирования, более или менее отличающиеся от Пролога. В статье К. Кларка с соавторами и статье М. Хагия и Т. Сакураи обсуждаются связи логических программ с логическими спецификациями.

Чтобы ориентировать читателей в литературе, оставшейся за пределами сборника, мы включили обзорную статью, в которой систематизируются основные понятия и направления логического программирования в широком смысле, далеко выходящего за рамки Пролога и прологообразных языков, и в которой приводится большой список литературы. По своей направленности этот сборник продолжает линию вышедших ранее в издательстве «Мир» сборников «Данные в языках программирования» (1982 г.) и «Требования и спецификации в разработке программ» (1984 г.). Эта линия заключается в том, чтобы давать в книге материалы, объединяющие теоретиков и практиков, обращая внимание теоретиков на практически важные направления развития теории и показывая практикам прикладную значимость теоретических работ.

А. П. Ершов
В. Н. Агафонов

ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ — ПРОШЛОЕ, НАСТОЯЩЕЕ И БУДУЩЕЕ[1)]

ДЖ. РОБИНСОН

Размышляя об истории и будущем идеи логического программирования, полезно различать следующие периоды:

далекое прошлое (1879—1970)
близкое прошлое (1971—1980)
настоящее (1981—1990)
близкое будущее (1991—2000)
далекое будущее (2001—?)

1. ДАЛЕКОЕ ПРОШЛОЕ (1879—1970)

Далекое прошлое начинается важной вехой в истории идей. В 1879 г. родился Альберт Эйнштейн и в том же году появилось еще кое-что, а именно то исчисление предикатов, каким мы его теперь имеем. Его изобрел один человек, Готтлоб Фреге, математик, цель которого состояла в том, чтобы полностью проанализировать формальную структуру чистого мышления. Фреге назвал свою систему Begriffsschrift — словом, которое ему, по-видимому, тоже пришлось изобрести. Оно, кажется, означает что-то вроде «система обозначений для понятий». Он считал ее универсальным языком, в котором можно было бы представить систематическим и математически точным образом любую возможную форму рационального мышления, которая могла бы стать частью дедуктивного рассуждения.

Так оно и оказалось. Я полагаю, что история системы обозначений Фреге оправдала его веру в нее. Остальная часть далекого прошлого — по существу развитие системы Фреге в одном конкретном направлении. Это направление можно было бы назвать «вычислительным исчислением предикатов»[2)], в котором всегда ищут алгоритмы, систематически

[1)] J. A. Robinson. Logic programming — past, present and future. — New Generation Computing, 1, 1983, p. 107—121. Статья представляет собой сжатое и отредактированное изложение публичной лекции, прочитанной в институте ICOT в Токио 10 февраля 1983 г.

[2)] Computational predicate calculus. — *Прим. ред.*

охватывающие процессы дедукции. Как заметил Фреге, эти процессы нужно представить математически точно, чтобы их можно было изучать формально. Однако вычислительными вопросами как таковыми он непосредственно не занимался.

Таким образом, далекое прошлое логического программирования — это история вычислительной логики. После появления Begriffsschrift в течение долгого периода, около тридцати пяти лет, ничего действительно значительного не произошло. В истории исчисления предикатов модно упоминать знаменитую работу Уайтхеда и Рассела «Principia Mathematica». Однако эта работа является на самом деле частью не вычислительной логики, а другой ветви развития, которой Фреге тоже очень интересовался: речь идет о попытках проанализировать до самых оснований основные идеи математики — понятия функции, бесконечности, множества и т. д. Фреге верил, так же как и Уайтхед с Расселом, что эти понятия можно проанализировать в чисто логических терминах, к чему как раз и стремилось это направление развития — логицизм.

В вычислительной же логике первой после Фреге действительно значительной была написанная в 1915 г. работа Лёвенгейма. Ею начинается плодотворный период исследований и открытий, достигший кульминации около 1930 г. Примерно в это время было окрыто то, что мы теперь считаем фундаментальной теоремой исчисления предикатов. Это сделали независимо друг от друга французский алгебраист и логик Жак Эрбран, который в 1929—1930 гг. в возрасте 21 года писал свою докторскую диссертацию, норвежец Торальф Скулем, зрелый профессиональный математик, работавший в 1920-х годах над той же проблемой, и Курт Гёдель, австрийский математик. Все они пытались доказать тот основной факт об исчислении предикатов, который Фреге, когда он его изобрел, с самого начала принимал на веру, а именно, что это действительно полная система обозначений, т. е. действительно делает все то, для чего она была предназначена.

А предназначение этого языка состояло в том, чтобы обеспечить формальное доказательство каждого записанного на этом языке логически верного предложения, т. е. истинного при всех возможных интерпретациях, и чтобы это доказательство систематически строилось по заданному предложению. Гёдель, Эрбран и Скулем разными способами показали, что так оно и есть. Именно этим людям мы сегодня обязаны процедурами доказательства в исчислении предикатов.

Эрбран дал несколько версий процедуры доказательства, в одной из которых он касается идеи, называемой теперь «унификация».

Конечно, в 1930 г. не было вычислительных машин, на которых можно было бы выполнять эту процедуру, они фактически появились только в начале 1950-х годов. Так что никто и не думал о программировании процедуры доказательства для вычислительной машины примерно до 1955 г., когда датчанин Эверт Бет решил попытаться сделать это. За ним последовали другие, среди которых следует упомянуть Стига Кангера и Дага Правица из Швеции, Поля Гилмора, Хао Вана, Мартина Дэвиса и Хилари Путнама из США, — все они решили испытать процедуру 25-летней давности на вычислительных машинах того времени.

Поступая таким образом, Даг Правиц «оживил» в 1960 г. понятие унификации и использовал его в одной из своих ранних программ.

Опыт этих исследователей показал, что алгоритм, как он сформулирован в 1930 г., был не слишком хорош с вычислительной точки зрения в современном смысле и что в нем встречаются «взрывы», чрезмерные разрастания комбинаторного характера. Исследователи были довольно разочарованы выполнением алгоритма.

Около 1961 г. я также начал изучать эти работы, и мне пришло в голову, что с помощью нескольких приемов можно улучшить выполнение основного алгоритма на вычислительной машине. В ходе моей работы я наткнулся на идею, называемую теперь «резолюция». Это был способ включить концепцию унификации в самую сердцевину той системы доказательства, с которой надо было работать.

Я еще буду говорить о событиях, которые последовали спустя короткое время после того, как резолюция стала способом, с помощью которого люди пытались заставить вычислительную машину выполнять основной процесс доказательства [1]).

Но сначала я хотел бы упомянуть о другой нити в этой краткой истории логического программирования. Она не является непосредственно частью развития собственно логики, но, конечно, имеет отношение к рассказу о логическом программировании. В середине 1960-х годов два джентльмена, чьи имена не так хорошо известны, как того заслуживают, — Майкл Фостер и Тед Элкок — экспериментировали с программным формализмом, который они назвали ABSYS (ABerdeen SYStem). Позднее они переименовали его в ABSET (ABerdeen SETlanguage). Это был язык для вычислений, основанный

[1]) По поводу истории вычислительной логики см. также книгу Чень Ч., Ли Р. «Математическая логика и автоматическое доказательство теорий» (М.: Наука, 1983) и комментарии редактора ее перевода С. Ю. Маслова.— *Прим. перев.*

на идее, что программист будет просто делать утверждения.

Эти утверждения, или предложения, будут, так сказать, аксиомами, которые программист считает истинными. Они должны вводиться в память машины и затем использоваться как посылки дедуктивных выводов, когда системе представлен запрос, как мы бы теперь сказали. Эти утверждения должны затем автоматически вызываться при выводе ответа на запрос.

Это была очень интересная ранняя попытка делать то, что мы теперь представляем себе как логическое программирование. Она предпринималась, однако, не в контексте формального исчисления предикатов, а интуитивно, в более или менее обычных математических обозначениях. Это была довольно сложная система, которая вела себя не так эффективно, как бы хотелось. Но каждый, кто хочет вернуться к истокам логического программирования, должен знать об этой работе.

Далее резолюция получила распространение и многие люди занялись ею, включая одного из первых пионеров резолюционного логического программирования Корделла Грина, который в конце 60-х годов в Станфорде пытался использовать резолюцию по существу современным способом как основу системы логического программирования. Он представлял ее себе как систему, отвечающую на вопросы. И достаточно правильно, это как раз и есть то, в чем фактически состоит логическое программирование. Его работа привлекла большое внимание своими отчасти хорошими, отчасти плохими сторонами. Хорошая сторона состояла в том, что здесь, наконец, был систематический план вычислительной системы, отвечающей на вопросы. Он, казалось, был очень общим и потенциально применимым к широкому кругу задач. Плохая сторона заключалась в том, что конкретный алгоритм резолюции, лежащий в основе системы, был все еще настолько вычислительно сложным, чтобы ограничить применимость системы довольно маленькими задачами. На больших задачах система Грина опять приводит к комбинаторным взрывам, и это разочаровывает.

Так что в конце 60-х начале 70-х годов высказывались отрицательные замечания, особенно со стороны специалистов из Кембриджа (Массачусетс), такого рода, что заниматься вычислениями в области искусственного интеллекта на основе логики и, особенно, резолюции чрезвычайно глупо и что надо идти другим путем. В период этих споров появилась система Плэнер, разрабатываемая в Массачусетском технологическом институте. Это была другая линия развития, которая теперь, как я полагаю, опять сливается с главным потоком.

Несколько человек — Роберт Ковальский, Дональд Кюхнер, Дэвид Лакхэм, Дональд Лавлэнд и другие — упорно атаковали проблему комбинаторной сложности резолюции, и в результате было испытано множество идей. Выигрышной оказалась идея ограничить правило резолюции так, чтобы дедуктивные структуры, порождаемые алгоритмом, были линейными по форме.

Это означает, что каждое доказательство должно иметь древовидную структуру с одной главной ветвью. Каждый выводимый дизъюнкт (предложение специального вида) должен лежать на главной ветви и быть резольвентой предыдущего дизъюнкта на главной ветви и одного из входных дизъюнктов.

Другое направление заключалось в том, чтобы взять процесс унификации и попытаться ускорить его и улучшить его выполнение.

Несколько человек — и я среди них — работали над этим; Боб Бойер и Дж. Мур присоединились к нам в это время. Это было в Эдинбурге в Шотландии. Они придумывали всякие удивительные алгоритмические трюки, включая совместное использование структур [1]), для ускорения резолюционного процесса. По существу достигнутый к этому времени прогресс можно резюмировать словами: все составные части были готовы для Пролога.

2. БЛИЗКОЕ ПРОШЛОЕ (1971—1980)

Нужен был такой человек, как Алэн Колмероэ, чтобы разглядеть все элементы и соединить их в однородную систему. Сначала Колмероэ изобрел систему, которую назвал SYSTEMQ, а потом он назвал ее (или его коллега Филипп Руссель назвал ее, или, возможно, даже жена Русселя) Прологом; мы не можем точно установить, кто придумал это имя.

Во всяком случае Пролог «родился» в Марселе в 1971 г. Попросту говоря, он состоял из системы линейной резолюции, в которой дизъюнкты, описывающие задачу, ограничивались хорновскими дизъюнктами, и предложенной Ковальским интерпретации того, что происходит, когда вы запускаете систему, — интерпретации, непосредственно преобразующей процесс доказательства в более традиционный процесс вычисления. Каждый шаг является вызовом процедуры, который затем возвращает некоторого рода результат тому, кто его вызвал. Все эти вычислительные понятия использовались Ковальским в этой процедурной интерпретации систем, основан-

[1]) Structure sharing. — *Прим. ред.*

ных на хорновских дизъюнктах и линейной резолюции. Так логическое программирование складывалось как концепция.

Ковальский был тем, кто увидел все это в Прологе. Он понял, как его рассматривать с двух точек зрения — во-первых, как логику, во-вторых, как вычисление.

Близкое прошлое начинается, стало быть, с Пролога, который полностью созрел в голове Колмероэ, и с Ковальского, начинающего свою кампанию в качестве неутомимого защитника идеи, эффективно распространяющего новое. Пролог существовал! Люди очень быстро увидели его достоинства и начали пользоваться им. Доклад Ковальского на Конгрессе *IFIP* в 1974 г. стал чрезвычайно убедительным обоснованием логического программирования вообще и Пролога в частности. Ковальский был первым, кто пробудил быстрый рост интереса к логическому программированию.

Это привело к тому, что ряд лучших молодых ученых Европы в области информатики занялись логическим программированием как своим главным делом. Я хотел бы просто упомянуть тех, которых знаю: Стена-Оке Тернлунда из Швеции, Кейта Кларка из Англии, Мартена ван Эмдена из Голландии, Мориса Бранохе из Бельгии, Петера Середи из Венгрии, Эрва Галлэра из Франции, Дэвида Уоррена, нашего теперешнего коллегу, Луиса и Фернандо Перейра из Португалии. Все эти замечательные люди вместе дали громадный импульс логическому программированию.

Довольно быстро происходило развитие и в организационном плане. Это иллюстрируется, например, прекрасной книгой «Логическое программирование» [1]), изданной Кларком и Тернлундом и содержащей труды международного семинара всего сообщества исследователей, работающих в области логического программирования и сообщающих на нем о своих идеях, связанных с развитием и применениями понятия логического программирования. Я думаю, что это очень помогает в работе.

Заседания семинара состоялись в Сиракьюсе (весна 1981 г.) и Лос-Анджелесе (лето 1981 г.), еще один планируется провести летом 1983 г. в Португалии. Летом 1982 г. в Марселе состоялась первая международная конференция по логическому программированию и планируется провести вторую летом 1984 г. в Уппсале, Швеция [2]).

Большую помощь логическому программированию оказали несколько прекрасных книг: «Логика для решения за-

[1]) Пять статей из этой книги представлены в данном сборнике. — *Прим. ред.*

[2]) Перечень других конференций и семинаров, а также их трудов см. в обзоре в конце настоящего сборника. — *Прим. ред.*

дач» Ковальского, руководство по Прологу, написанное Клоксином и Меллишом (Клоксин У., Меллиш К. Программирование на Прологе. — М.: Мир, 1987.), и упомянутая выше книга под редакцией Кларка и Тернлунда [1]).

И конечно же мы должны засвидетельствовать свое уважение удивительной реализации Пролога, последовавшей за первоначальной марсельской реализацией, основанной на Фортране. Логическому программированию чрезвычайно повезло с удивительной эдинбургской реализацией Пролога на DEC-10, выполненной Дэвидом Уорреном, которая, я считаю, действительно выдвинула логическое программирование на первое место и сделала его полезным инструментом для разных применений. Невозможно переоценить воздействие, оказанное Дэвидом благодаря этой реализации.

В 1975 г. Эрни Зиберт и я решили реализовать систему логического программирования на Лиспе в Сиракьюсе. Нашу систему мы назвали Логлиспом (LOGLISP)[2]). Общая идея, лежащая в основе Логлиспа, состоит в том, чтобы как можно деликатнее объединить понятие логического программирования с понятием функционального программирования, примером которого является Лисп. Это самый известный пример, но теперь существуют более чистые и элегантные системы функционального программирования, такие как SASL и KRC Дэвида Тёрнера, LISPKIT Петера Хендерсона и система, о которой говорил Джон Бэкус в своей лекции тьюринговского лауреата 1977 г.[3]).

Внешне функциональное программирование выглядит как понятие, независимое и несколько обособленное от логического программирования. Однако я утверждаю, что и то и другое являются примерами более общей, фундаментальной, единой и простой идеи, которую можно было бы (вспомним идею из Абердина) назвать «утвердительное программирование». Это тип программирования, когда вы занимаетесь тем, что утверждаете о некоторых предложениях, что они истинны, и затем просите вывести другие предложения как их следствия.

В логическом программировании эти утверждаемые предложения имеют вид импликаций, а в функциональном — равенств (уравнений). Но это в действительности только внешнее различие. Я думаю, что главное здесь то, что, когда вы

---

[1]) Библиографические сведения об этих и других книгах и работах по логическому программированию см. в списке литературы к обзору в конце сборника. — *Прим. ред.*

[2]) См. статью Дж. Робинсона и Э. Зиберта в данном сборнике. — *Прим. ред.*

[3]) См. также разд. 2.1 обзора в конце сборника. — *Прим. ред.*

запускаете системы того или другого вида, вы запускаете дедуктивные машины: просите их выполнить для вас нужные дедукции (выводы).

Так что Логлисп, который мы сейчас заканчиваем в Сиракьюсе, — это попытка заключить в единые рамки оба стиля программирования. Другие люди, такие, как Ян Коморовский из Линкопинга в Швеции (сейчас он в Гарвардском университете), тоже пытались объединить Лисп и логическое программирование, и, конечно, нет надобности указывать такой аудитории, что это крупная тема в вашем собственном проекте вычислительных систем пятого поколения.

Начало этого проекта — большое событие, которое отмечает конец близкого прошлого. Мы на Западе совершенно неожиданно получили этот приятный сюрприз: обнаружили, что вы здесь в Японии спокойно изучили эти идеи, неизвестные у нас, и поняли, для чего они предназначены, а именно для красивой, сильной техники, которую можно использовать тем способом, в котором вы сведущи. Для нас в конце 70-х годов это был удивительный способ.

Я думаю, следует сказать, что фактически в качестве центральной идеи проекта вы принимаете не логическое, а утвердительное программирование[1]), потому что, посещая здесь исследовательские группы, я снова и снова слышал ту же самую идею — объединить логическое программирование с функциональным.

Так что мы покидаем близкое прошлое под нарастающие торжественные звуки оркестра и вступаем в настоящее.

3. НАСТОЯЩЕЕ (1981—1990)

Но в этом разговоре мы не можем обсуждать настоящее в историческом плане, так как оно еще не прошло. Вместо этого я хотел бы предложить некоторые наблюдения относительно того, где мы находимся и что нам следует попытаться сделать.

Я думаю, имеются некоторые тенденции, о которых нам следует побеспокоиться. Боюсь, что даже успех Пролога, который так прославился, может иметь некоторые неблагоприятные аспекты. Например, я сожалею (по причинам, сходным с теми, которые Дейкстра высказывал относительно оператора перехода GOTO), что в Прологе есть оператор усечения CUT и что людей, программирующих на Прологе, поощряют проявлять изобретательность в управлении тем конкретным способом, которым в Прологе строится дерево вывода. Там

[1]) Assertional programming. — *Прим. ред.*

это делается поиском в глубину с помощью бектрекинга (отхода в случае необходимости назад), при котором просматриваются все узлы дерева.

Это не является необходимой чертой систем логического программирования, это случайная особенность Пролога. Было бы лучше, если бы такие детали были невидимы для пользователя и не навязывались ему как одна из главных вещей, которые ему надо уметь делать при написании программ.

Таким образом, оператор CUT — нехорошая вещь, но он, по-видимому, уже выходит из употребления, так как это последовательное понятие. По мере того как появляются параллельные реализации Пролога, вы не захотите заниматься бектрекингом изнутри, вы будете строить дерево в целостной, параллельной манере. Тогда интуиция программистов, лежащая в основе разработки алгоритмов на Прологе, изменится и поднимется на более высокий уровень. И это будет хорошо.

Подоплекой конструкции CUT в Прологе является в действительности некоторое средство, позволяющее программисту экономить вычисления при построении дерева вывода. Конечно, хотелось бы, чтобы программист имел возможность дать системе совет, какие элементы дерева игнорировать как ненужные, когда построение дерева в ходе вычисления достигло определенного состояния, которое нельзя установить заранее, до того как вычисление начнется. Очень желательно, мне кажется, обеспечить программиста средствами, влияющими на эффективность построения дерева. Но не за счет ясности программы!

К этому имеет отношение и то обстоятельство, что не следовало бы учитывать порядок, в котором записаны компоненты конъюнкции, потому что логически у них нет никакого особого порядка. Задать для них порядок — значит наложить дополнительное ограничение на то, что мы утверждаем.

Не следовало бы беспокоиться по поводу порядка не только внутри дизъюнкта, но и между самими дизъюнктами.

Короче говоря, мы не должны включать в сами логические обозначения специальные соглашения о том, как управлять деталями поиска вывода. Теми деталями, с которыми процессор сам, возможно, как следует не справится, должен управлять программист с помощью управляющих входов. Но эти входы надо отделить от логических входов.

Надо иметь в виду, что логическое программирование вообще не следует отождествлять с Прологом в частности. Взаимоотношение между ними состоит в том, что Пролог — это одна из реализаций, одно конкретное представление понятия логического программирования. Я бы даже высказал более общее утверждение, что логическое программирование не

следует полностью отождествлять с программированием, основанным на хорновских дизъюнктах и резолюции. Это опять же специальный случай — и очень хороший, как мы видим, — общей идеи дедуктивных вычислений исходя из утверждений.

Вы можете пойти еще дальше и сказать, что эта общая идея не ограничивается исчислением предикатов первого порядка. В конце концов, существуют и другие интересные логики, существует логика высоких порядков, разные оттенки модальной логики и т. д. Существуют богатые формализмы всех сортов, которые когда-нибудь станет мыслимо использовать так, как мы сейчас используем ограниченное исчисление предикатов, занимаясь логическим программированием.

Так что нам надо, как я полагаю, поберечь терминологию и сохранить логическое программирование как особую концепцию, а потом получать отдельные понятия как ее различные специальные случаи.

Я считаю, что нам надо, в том же духе, отделить общую идею системы логического программирования от идеи полной программной обстановки. Мне кажутся странными некоторые вещи, которые вам приходится делать в разных попадавшихся мне реализациях Пролога. Например, вы должны создавать побочные эффекты вроде печатания при попытке доказать предложение, и в ходе поиска доказательства происходят какие-то побочные события. Концептуально это выглядит не очень хорошо. Я думаю, лучше быть честным в том, что касается императивного программирования. Если вы хотите, чтобы что-то происходило, вам надо иметь соответствующие средства, чтобы сказать об этом и сделать это происходящим. Тогда побочные эффекты не приведут всю вашу семантику утверждений в беспорядок.

По поводу Пролога надо, я полагаю, еще отметить, что он преувеличивает роль, которую играют отношения в утвердительном программировании. Они, конечно, играют очень важную роль, но отношения — это еще не всё.

Иногда мне кажется, что мы вернулись к первым дням машинных вычислений, когда, чтобы задать вычисление выражения, нужно было вводить имена промежуточных значений и сохранять их в ячейках с этими именами, а в конце должна была быть ячейка, содержащая ответ. Конечно, промежуточные именования на шагах последовательного вычисления — это нечто такое, чего мы не хотим быть обязанными делать. И наступил очевидный прогресс, когда появился Фортран, который позволил нам просто писать выражения и вычислял их, не требуя от нас именовать все эти промежуточные результаты на языке ассемблера.

Если вы посмотрите на некоторые написанные на Прологе программы, содержащие выражения с большой глубиной вложенности, вы внезапно обнаружите, что вернулись в те самые дни и вынуждены именовать промежуточные шаги последовательного вложенного вычисления для того, чтобы в конце получить значение. Я не думаю, что выражение само по себе не логично — в конце концов это просто терм, — но я предпочел бы поднять функции до того же уровня, на котором находятся отношения, как это делал Фреге в первоначальном замысле исчисления предикатов.

Я знаю азбучную истину, что функция — это просто специальный вид отношения. Но это можно повернуть в обратную сторону и с тем же успехом показать, что отношение — это просто специальный вид функции. На самом деле именно так на это смотрел Фреге. Для него отношение — это функция (отображение) из множества кортежей объектов в множество истинностных значений. И таким образом, вы представляете себе вычисление отношения точно так же, как вычисление любой другой функции. Отношение отличается просто множеством принимаемых им значений.

Далее можно поставить вопрос: что такое наилучшая программная обстановка? Каковы должны быть ее элементы? Если мы хотим, чтобы она содержала редакторы, команды ввода-вывода и другого рода средства, вызывающие побочный эффект, то нам надо все очень тщательно обдумать, чтобы не испортить один из наших самых великолепных инструментов, а именно формализм логического программирования. Он, конечно, должен быть частью этой обстановки, но мы не хотим перегружать его, как мне кажется, всеми этими посторонними обязанностями.

Меня беспокоит и другое. До сих пор все системы логического программирования, с которыми мы имели дело, были маленькими. Что я под этим подразумеваю? Главным образом то, что при их выполнении на машинах все предположения хранились в оперативной памяти и поэтому доступ к ним осуществлялся с помощью хороших методов индексирования и ассоциативного поиска.

Что произойдет, когда мы перейдем к более крупным системам, так что не сможем поместить все в оперативную память? Мы собираемся работать с виртуальной памятью, существенно основанной на использовании дисков. И тогда перед нами встает проблема замедления доступа к предположениям, которая меня несколько беспокоит. Мне не ясно, как при переходе к очень большим системам мы собираемся получить ускорения, о которых говорится в проекте вычислительных систем пятого поколения,

Сегодня системы логического программирования имеют скорости в липсах [1]) порядка 10^4. Если мы собираемся повысить скорость до 10^9, то нам надо подумать, как получить это ускорение. Мне кажется, что если мы сможем остаться в оперативной памяти машины, то вполне можно надеяться, что мы это ускорение получим. Перейдя к параллельным вычислениям, мы, вероятно, получим выигрыш в сто раз. Перейдя к аппаратуре 1990 г., мы, вероятно, умножим скорость еще на сто. Остальные десять раз можно надеяться получить, если мы будем более изобретательны, чем были до сих пор, в организации основных алгоритмов.

Так что я думаю, что ускорение в 10^5 обоснованно при условии, что мы можем заниматься этим в быстрой памяти. Но если нам надо получать наши дизъюнкты из дисковой памяти, то перед нами встает проблема, как достичь этого ускорения. Или как еще мы должны хранить терабайты информации?

Признаюсь, что иногда я испытываю беспокойство из-за того, что вы сделали логическое программирование центральной темой вашего проекта вычислительных систем пятого поколения. Я задаю себе вопрос, оправдывается ли ваше большое доверие к этой идее. Существует, как вы знаете, некоторый риск в том, чтобы сделать эту идею центральной. Фактически это эксперимент. Я думаю, вероятность того, что эксперимент будет успешным, очень велика. Но есть и некоторые опасности. В подходящий момент я еще скажу об этом.

И наконец беспокойство общего характера — что произойдет с логическим программированием как с чисто абстрактной идеей, когда люди разделаются с ним? Сейчас все уделяют большое внимание этой парадигме, изменяют ее, экспериментируют с ней разными способами и с разными мотивами. Что-то произойдет с нею, и меня беспокоит, что это, возможно, не всегда будет к лучшему. Мы должны стараться направлять это развитие в начинающемся сейчас десятилетии, чтобы к концу его иметь такое понятие системы логического программирования, которым мы могли бы гордиться, которое было бы элегантным, мощным, простым и действительно обладало всеми достоинствами, каковыми логическое программирование как идея, по-видимому, сейчас обладает.

Будем надеяться, что, добиваясь практического успеха логического программирования в больших масштабах, мы не пожертвуем этой элегантностью и красотой. Меня нервирует, когда я читаю статьи или слышу дискуссии, в которых логи-

---

[1]) LIPS (Logical Inference Per Second) — количество логических выводов (шагов) в секунду. — *Прим. перев.*

ческое программирование смешивается со всем, что есть под солнцем. Возможно, я напрасно беспокоюсь. Надеюсь, что так.

Наконец, так как я очень люблю Лисп, как и многие другие люди, я надеюсь, что Лисп, эта красивая вещь, не исчезнет. Я не настолько фанатичный сторонник логического программирования, чтобы желать Лиспу поражения и полного вытеснения чем-то вроде Пролога. Как я говорил раньше, правильная линия для них обоих состоит в том, чтобы стать тем, чем они стараются быть, а именно системой утвердительного программирования. Так что я хочу, чтобы Лисп выжил — не обязательно вплоть до мельчайших деталей, но как основная идея формализма, основанного на ламбда-исчислении, с точечной парой в качестве универсальной структуры данных. Это красивая и мощная идея.

Так что не будем разрушать Лисп, обеспечивая успех логическому программированию.

Нам представляется удивительно благоприятная возможность хорошо поработать над парадигмой логического программирования. Посмотрим, что делал с Лиспом Питер Ландин в начале 60-х годов. Он ясно показал то, что, как это ни удивительно, не вполне осознавал Маккарти, изобретая Лисп, а именно что Лисп — это по существу ламбда-исчисление. Он объяснил это с помощью изумительно элегантной абстрактной машины — SECD-машины. Я думаю, что эта работа Ландина чрезвычайно важная и красивая.

Если вы посмотрите на работу современных исследователей в области функционального программирования, таких, как Дэвид Тёрнер и Питер Хендерсон [1]), то и здесь обнаружите такую же жажду элегантности, к которой я лично отношусь очень положительно. Я думаю, что важно стремиться к красоте и элегантности в этих поисках, связанных с приложениями математики. В действительности, если это красиво, то вы не сможете уйти далеко по неверному пути.

Мы не хотим, чтобы логическое программирование плодило «кладжи» [2]). Это было бы ужасно. Одной из неудачных особенностей программирования в области искусственного интеллекта за последние пятнадцать или двадцать лет была тенденция создавать эффективные, но уродливые системы. Давайте попытаемся избежать этого.

---

[1]) П. Хендерсон. Функциональное программирование. Применение и реализация. М.: Мир, 1983. — *Прим. ред.*

[2]) Kludge — словечко из англоязычного программистского жаргона, означающее неуклюжую, уродливую программу, конструкцию и т. п. — *Прим. ред.*

Примером удобной возможности повысить элегантность является унификация. Я верю, и многие другие тоже, что унификация — очень сильная идея, которая может объяснить ряд других идей, возникающих в информатике, очень хорошо, просто и правильно. Я думаю, что это основной механизм для всех процессов передачи параметров как при вызовах функций, так и при их активации.

Так это выглядит в Прологе и в процедурной интерпретации Ковальского. Но этот механизм никогда фактически не испытывался, насколько мне известно, в контексте функционального программирования [1]). В Алголе, Паскале, Лиспе и так далее передача параметра соответствует одностороннему совмещению (отождествлению) формального параметра с фактическим. Фактический параметр при таком сопоставлении не изменяется, изменяется только формальный параметр. Он должен совпасть с фактическим параметром, и после этого выполняется тело процедуры.

Теперь у нас есть шанс посмотреть, что произойдет с функциональным программированием, когда мы обобщим механизм передачи параметров, превратив его в двусторонний обмен информацией. Кеннет Кан и Харви Абрамсон занялись разработкой систем функционального программирования, в которых унификация является ведущим принципом.

Кроме того, сейчас многим людям стало ясно — среди них, разумеется, Колмероэ и ряд лиц, с которыми я разговаривал здесь в Японии, — что можно очень хорошо проводить вычисления со структурами, которые можно описать как бесконечные выражения. В действительности они не бесконечные, они являются представлениями бесконечных структур. Представление строится с помощью указателей, которые вводят циклы в эти структуры. Лисп годами имел дело с такими структурами, но как бы украдкой. Использование функций RPLACA и RPLACD считалось «не совсем респектабельным» и во всяком случае опасным.

Унификацию можно очень хорошо обобщить — и это сейчас сделано во многих системах, — чтобы работать с выражениями такого рода так же, как с более обычными конечными выражениями, которые первоначально имелись в виду.

Если вы это сделаете, то получите возможность представлять потоки, а также вводить в дедуктивные вычисления «ленивые» вычисления [2]) и многие другие хорошие вещи. Неко-

[1]) Сейчас ситуация изменилась, см. обзор: Bellia M., Levi G. The relation between logic and functional languages: a survey. — J. Logic Programming, 3, 1986, No 3, p. 217—236. — *Прим. ред.*

[2]) См. пояснение в разд. 2.1 обзора в конце этой книги. — *Прим. перев.*

торые люди работают над этим, включая многие группы в Японии.

Как показывают ваши планы проекта вычислительных систем пятого поколения, у вас сейчас есть хорошая возможность развивать новые архитектуры, вводить разные виды параллелизма и перейти к применениям очень больших баз данных. Я уже выражал беспокойство по поводу того, сможете ли вы иметь одновременно и огромные совокупности дизъюнктов в терабайтах памяти и гигалипсы скорости. Мне кажется, что перед вами стоит проблема, с которой я лично не знаю как справиться. Однако новая технология заманчива.

Сейчас, я думаю, мы готовы к тому, чтобы появилась книга по методологии логического программирования, подобная известной книге Кнута. Она оказала бы большое влияние на специалистов в области информатики и вычислительных машин, которые, вероятно, еще не слышали о логическом программировании. Возможно, хорошими авторами такой книги были бы Стен-Оке Тернлунд и Кейт Кларк, или Егуд Шапиро, или Мартен ван Эмден. В действительности они уже делали попытки изложить на бумаге, к чему, собственно, стремится логическое программирование. Я знаю, что какие-то книги уже есть, что есть написанное Ковальским введение в идеи логического программирования. Но это не совсем то, что я имею в виду. Я упомянул Кнута, потому что все знают, какую удивительную работу он проделал для, так сказать, фоннеймановского программирования, и теперь логическое программирование нуждается в своем Кнуте. Возможно, этим заинтересуется сам Кнут — кто знает? [1])

Другая вещь, которая меня беспокоит, — это отождествление логического программирования с искусственным интеллектом как движением в истории идей. Мне кажется, что это совсем разные вещи, и мой совет таков: нам надо стараться сохранить логическое программирование достаточно обособленным от искусственного интеллекта, не пытаться сцепить их вместе. Во-первых, я полагаю, что ИИ (искусственному интеллекту) придется, по-видимому, угодить в очередной из своих периодических спадов. Если вы посмотрите на историю ИИ, то увидите, что в нем происходят то подъемы, то спады: чрезмерный энтузиазм сменяется столь же чрезмерным разочарованием. Когда вы видите много поверхностной журналистики, множество телевизионных интервью с известными лицами, вам начинает казаться, что за работу

---

[1]) Здесь стоит упомянуть появившуюся позднее книгу: К. Хоггер. Введение в логическое программирование. — М.: Мир, 1988. — *Прим. перев.*

взялись не те силы. В хорошем научном направлении дело происходит спокойнее и не нуждается в такого рода показе средствами массовой информации, каким пользуется ИИ, если в действительности его и не ищет.

Я считаю, что многие из известных достижений ИИ — это доброкачественные кладжи. Иначе говоря, я не думаю, что вы сможете извлечь из них, даже из успешных, какую-то систематическую, глубокую, фундаментальную науку. Не всегда ясно, почему что-то хорошо работает, если оно хорошо работает. Мне кажется, что ИИ должен пройти большой путь, прежде чем он станет чем-то вроде науки, прежде чем он заслужит это название. Мне кажется, что он состоит главным образом из очень благих устремлений. Много хороших дел затевается, но стремиться — это не то же самое, что достичь. Вы должны делать дело, а не только говорить о том, что вы его делаете.

Например, я полагаю, что какая-то часть пропаганды прессой понятия «экспертной системы» несколько вводит в заблуждение. Если вы посмотрите на хорошо известные примеры, такие как MYCIN, или PROSPECTOR, или MACSYMA (это успешные примеры, поймите меня правильно!), и если вы спросите, почему они успешны, то вы, я думаю, поймете, что дело тут не в методологии, которой следовали при их построении, потому что эта методология относительно тривиальна. В действительности эти системы делает успешными (особенно систему MACSYMA, которая, я думаю, лучше всего это подтверждает) то, что они наполнены специальными знаниями экспертов в конкретной области. MACSYMA — это собрание алгоритмов символьной математики, которые были сведены воедино действительно сильными прикладными математиками, людьми, действительно знающими эту область, а также оказавшимися сведущими в Лиспе. Так что они выразили себя на Лиспе, и результат этого — MACSYMA. Человек, написавший MYCIN, — врач, который сам является хорошим диагностом.

В этих случаях мы имеем дело с людьми, которые, зная свою область, существенно используют преимущества вычислительного формализма, помогающего им сказать то, что они знают. И вполне естественно, если они имеют ясное представление об этом, то могут добиться хороших приложений.

Фейгенбаум убедительно показал, что в экспертных системах имеют значение всегда именно конкретные знания эксперта, а не какая-то общая единообразная техника. Сообществу специалистов по ИИ не очень-то приличествует говорить: «Посмотрите на эти успехи. Технология ИИ просто применялась к таким-то и таким-то проблемным областям, и

мы получили экспертные системы». Дело было не так. Такой вещи как технология ИИ, преимущества которой использовали бы эти люди, не существует. Что они используют, так это вычислительные машины и хороший язык программирования.

Возможно, я спровоцировал вопросы, на которые отвечу после лекции [1]), а сейчас буду продолжать.

Позвольте мне наконец сказать кое-что о проекте вычислительных систем пятого поколения. Будучи очень дружественным наблюдателем, я различаю два класса целей этого проекта. Первый вы могли бы назвать так: «техника программирования и вычислительная техника». Мне кажется, что эти цели реалистичны. Они определенно будут достигнуты, они даже консервативны. Они так хорошо обдуманы и спланированы.

С другой стороны, я думаю, что цели, которые вы бы могли классифицировать как цели в области ИИ, — такие как понимание речи, машинное зрение и перевод с естественных языков, — это очень претенциозные и удивительные устремления. Они имеют другой порядок трудности, потому что в них так много неизвестного, а в нашем багаже не так много приемов, которые помогли бы нам достичь этих целей. Я не решаюсь сказать, что они слишком претенциозны, но они другого рода.

4. БЛИЗКОЕ БУДУЩЕЕ (1991—2000)

В 1990-х годах мы увидим результаты проекта пятого поколения. И можно ожидать, что главным эффектом проекта будет то, чего он пытается достичь, а именно сделать доступными все виды новых приложений и новых способов вычислений.

Я полагаю, что мы можем ожидать широкого распространения экспертных систем. Если инструменты будут доступны, то я не думаю, что понадобятся особого рода эксперты — «инженеры знаний», — чтобы реализовать такие системы. Цель революции пятого поколения исключить, насколько воз-

[1]) Отвечая после этой лекции на вопрос о его отношении к ИИ, Дж. Робинсон сказал: «...у хорошей науки есть определенное достоинство, которое состоит в том, что она верна критерию сдержанности и проверки идей, что она очень систематична в своих объяснениях и т. д., и я ценю это очень высоко... Но насколько я мог следить за областью ИИ, ее литературой и теми, кто в ней практикует, я не обнаружил там большого уважения к этому духу. Там есть бьющая через край энергия и заразительное возбуждение, но там же публикуется много небрежных и незаконченных результатов. Это вибрирующая, юная, хаотическая область». — *Прим. ред.*

можно, роль таких посредников. Сегодняшняя ситуация, когда эксперт-профессионал в какой-то области не всегда в состоянии выразить свои профессиональные знания в подходящей для вычислений форме, не является моделью для будущего. Нам следует ожидать, что распространится «грамотность в области логического программирования»[1]).

Экспертная система близкого будущего только внешне будет «сверхчеловеческой». Она будет просто воплощением существующих профессиональных знаний, находящихся сейчас в самом человеке. Конечно воодушевляющая перспектива состоит в том, чтобы повысить скорость решения и объем задач, похожих на те, с которыми люди могут справляться, но выходящих за границы вычислительных возможностей собственно человеческого инструмента обработки данных.

У нас, у людей, такая маленькая память и такие медленные процессоры, что, даже если они очень параллельные, мы сильно ограничены в том, как можно использовать те знания, которые нам удается приобрести за короткий срок нашей жизни.

Если мы знаем, как это работает, знаем, как это выразить и как это вызвать и привести в действие, тогда у нас есть возможность расширить то, что мы уже понимаем.

Очень хорошим примером этого в сегодняшней технологии может служить сверхъестественная и несколько выводящая нас из себя сила лучших программ для игры в шахматы. Процесс, лежащий в основе игры современных шахматных программ, — элементарный и невоодушевляющий. Это просмотр нескольких ходов вперед в дереве перебора ходов и вычисление по очень понятной схеме оценок признаков позиций, получающихся после этих ходов, а затем обработка вычисленных значений методом минимакса. Это не очень глубокая идея, но оказывается, что она реализуется в таких масштабах, которых уже достаточно, чтобы наступили трудные времена для некоторых людей из числа самых лучших шахматных игроков. Зарегистрированы случаи, когда международные гроссмейстеры обнаруживали, что им трудно избежать поражения в некоторых специфических ситуациях эндшпиля, когда машина просто играет этим открытым легко понимаемым способом, но в гигантских масштабах и с огромной скоростью.

Поэтому бедное человеческое существо сталкивается с чем-то таким, что в принципе оно могло бы делать, но что делается в таких необычных масштабах, при которых воз-

[1]) Сравните с пропагандируемой академиком Л. П. Ершовым «компьютерной грамотностью». — *Прим. ред.*

никает различие в степени выполнения, — «эффект порядка величины».

Все это, мне кажется, можно осуществить, если мы немного экстраполируем текущие тенденции во всех областях. Особенно интересной мне кажется перспектива дешевых персональных рабочих станций с разнообразными возможностями, которых можно ожидать в 1990-х годах. Мне не кажется невероятным, что каждый из нас будет иметь в личном пользовании что-то вроде всемирной библиотеки или Библиотеки Конгресса. Чтобы хранить ее, будет достаточно полочки с оптическими дисками, подобной нынешним собраниям грампластинок.

5. ДАЛЕКОЕ БУДУЩЕЕ (2001—?)

У многих людей 2001-й год ассоциируется с названием популярного фильма Стэнли Кубрика и Артура Кларка, в котором говорящий компьютер HAL вызывает катастрофический невроз и саботирует экспедицию на Юпитер. Такого рода «реалистическая» научная фантастика, по-видимому, не слишком отличается от того вида рациональных догадок, которые нам потребуются, чтобы заглянуть в далекое будущее.

Я уверен, что в этой аудитории есть много людей, которые больше подходят для таких размышлений, чем я. Но мне кажется, что мы можем сейчас различать две долговременные тенденции, которые достигнут некоторой кульминации вскоре после 2001-го года.

Некоторые люди уже исследуют возможности, открываемые генной инженерией, использование которых позволит строить белковые структуры в соответствии с программами, заложенными в ДНК, точно так, как это происходит в природе. Идея состоит в том, что мы тоже могли бы эксплуатировать генетическое кодирование и, используя его как программную среду, строить структуры соответствующего масштаба. Такая интеграция ультравысокого уровня является естественной кульминацией современных тенденций и представляет собой естественную технологию, применяемую самой природой. У природы есть большой опыт по этой части, и наши мозги и нервные системы — это компактные, сложные, мощные устройства, построенные исключительно таким способом.

Я не вижу причин, почему бы нам не поискать способы непосредственного моделирования нейрофизиологических процессов. Скорость, с которой продвигаются экспериментальные работы в медицинских исследовательских центрах, на самом деле просто изумительна. Лауреаты Нобелевской пре-

мии 1982 г. Хьюбел и Визель показали нам кое-что поразительное относительно того, как работает система зрения у животных и, предположительно, у человека.

По-видимому, в ней имеются систематические структуры, которые кажутся очень знакомыми проектировщикам вычислительной аппаратуры. Я думаю, что за два или три десятилетия мы должны уйти очень далеко в понимании настоящих естественных систем и будем в состоянии до некоторой степени воспроизводить их.

Нам следует также ожидать сопряжения искусственных систем с нашими собственными — дополнительных протезирующих устройств для усиления того, что у нас уже есть. Таким образом, мы сможем увидеть дополнительные модули памяти, усиленные зрение и слух, а также вспомогательные обрабатывающие единицы для прямого доступа к внешним информационным ресурсам, словарям и т. д. Сегодня мы видим, что медицинская технология создает много видов механических протезов. Мы начинаем подумывать о построении протезов также и для функций, связанных с обработкой информации.

Наконец давайте попытаемся подумать заранее, что могли бы сделать интеллектуальные вычисления для важных задач, которые чрезвычайно велики или чрезвычайно трудны (или одновременно и то и другое) и с которыми сейчас мы можем сделать не так уж много. Таковы, например, детальные модели мировой экономики или мировой экологии, перевод разговорного естественного языка в реальное время. Очевидно, что долговременные цели проекта вычислительных систем пятого поколения ориентированы на них. Очевидно также, что эти цели будут достигнуты. Вопрос только в том, как скоро. Я думаю, что каждый сам может поразмышлять о том, что все это может означать с точки зрения образа жизни и как это в действительности повлияет на мир и гармонию между разными людьми.

Я хотел бы закончить словами, что, хотя существует некоторый языковой барьер между вами и мной, у меня никогда не было более счастливых и более плодотворных трех недель, чем те, которые сейчас заканчиваются для меня здесь в Японии. Возможно, нам не так уж и нужен автоматический переводчик, как нас уверяют некоторые люди.

Большое вам спасибо. Если есть время для вопросов, я был бы счастлив ответить на них [1]).

[1]) Из-за ограниченности объема сборника вопросы и ответы опущены, кроме приведенного ранее фрагмента. — *Прим. ред.*

ПРОЛОГ — ТЕОРЕТИЧЕСКИЕ ОСНОВЫ И СОВРЕМЕННОЕ РАЗВИТИЕ[1)]

А. КОЛМЕРОЭ, А. КАНУИ, М. ВАН КАНЕГЕМ

СОДЕРЖАНИЕ

Часть первая: язык и его приложения

1. ВОЗНИКНОВЕНИЕ И РАЗВИТИЕ

1.1. Начало

Развитие методов, возникших в области искусственного интеллекта, их применение в более широких областях, таких, как понимание текстов на естественных языках, экспертные системы, банки данных, происходило посредством специализированных языков программирования, которые содержали необходимые средства для аксиоматизации и решения поставленных задач.

Так, в начале шестидесятых годов в США появился язык Лисп, ставший темой углубленных разработок ученых аме-

[1)] Colmerauer A., Kanoui H., van Caneghem M. Prolog, bases théoriques et développements actuels. Technique et Science Informatiques, vol. 2, No 4, 1983, p. 271—311.

риканских университетов, касающихся не только самого языка, но и его окружения (физические устройства и программное обеспечение).

Десятью годами позже А. Колмероэ, занимаясь разработкой нового языка программирования для задач анализа и понимания естественных языков, решил использовать язык логики первого порядка и методику автоматического доказательства теорем. Представленный в виде предложений определенной формы и снабженный удобным правилом вывода, этот язык логики развился в эффективный язык программирования Пролог.

Программа на этом языке состоит из некоторого множества отношений, а ее исполнение сводится к выводу нового отношения из тех, что образуют программу. Таким образом, получается естественный, элегантный и мощный формализм.

Надо отметить, что все классические языки программирования являются языками *процедурного* типа. Это означает, что программист должен указать в явном виде все шаги вычисления один за другим, которые машина должна проделать, чтобы решить поставленную задачу. В противоположность этому в «декларативной» логике, как, например, в Прологе, достаточно описать задачу с помощью правил и утверждений относительно входящих в нее объектов. Если это описание достаточно точно представляет задачу, машина может самостоятельно найти ее решение.

Следовательно, пользователь может направить все свои усилия на описание задачи, оставив второстепенные заботы машине, в данном случае интерпретатору Пролога. Это позволяет работать в принципиально недетерминистском стиле и, кроме того, с объектами частично или полностью неизвестными, как это происходит в математике при решении уравнений.

Вначале Пролог использовался в основном для доказательства теорем и был основан на методе резолюций А. Робинсона [Робинсон 65] с наложенными на него драконовскими ограничениями для уменьшения пространства поиска (линейное доказательство, доступ только к первому литералу в каждом предложении и т. д.). Заслуга Р. Ковальского и М. ван Эмдена [Ковальский 76] заключается в обнаружении того, что эти ограничения эквивалентны использованию так называемых *хорновских дизъюнктов*, т. е. дизъюнктов, содержащих самое большее один положительный литерал. Хорновские дизъюнкты послужили основой для первой теоретической модели Пролога. Затем мы взяли эту модель [Колмероэ 70], чтобы систематизировать использование формальных грамматик, образованных из схем правил, и при-

шли к языку не менее мощному, чем язык Q-систем [Колмероэ 70], который можно рассматривать как предка Пролога.

Первый интерпретатор Пролога был написан в нашей лаборатории П. Русселем в 1973 г. и оказал сильное влияние на своих преемников.

В этом интерпретаторе был применен метод непереписывания термов, и благодаря сделанным в нем сильным ограничениям (линейность доказательства, упорядоченность литералов в предложении, управление недетерминизмом, унификация без проверки на вхождение переменных) стало возможным использование Пролога как языка программирования.

1.2. Успех

Первый интерпретатор, написанный на Фортране, был установлен на бывших тогда в ходу машинах и стал постепенно распространяться (Франция, Англия, Португалия, Испания, США, Канада, Польша, Венгрия, ...).

Среди других реализаций Пролога необходимо отметить компилятор, написанный Д. Уорреном для машины DEC-10. Затем настала очередь микромашин, где нами была сделана версия Пролога для Exorciser (Motorola 6800) и Ф. Макейбом — для Sorcerer.

Пролог был использован в различных приложениях, относящихся к искусственному интеллекту:

— общение с ЭВМ на естественном языке;
— формальные вычисления;
— построение планов действий роботов;
— написание компиляторов;
— базы данных;
— САПРы;
— экспертные системы;
и т. д.

Как это ни парадоксально, но успех Прологу в информатике принесли некоторые добавки к нему, недопустимые с теоретической точки зрения, но дающие возможность запрограммировать необходимые приложения. Это относится в особенности к (знаменитому и оспариваемому многими) оператору «/», который оказался необходимым в тех случаях, когда нужно помочь интерпретатору избежать огромного перебора возможных путей.

Нам кажется, что Прологу суждено сыграть одну из первых ролей в развитии информатики. В частности:

— Пролог является хорошим средством для быстрого построения макетов, что весьма важно в проектах, связанных с искусственным интеллектом;

— при правильном преподавании информатики необходимо использовать такой язык, который помогал бы структурировать мышление. Сначала людей взращивали на Алголе-60, затем на Паскале. Лучше их воспитывать на Лиспе, но еще лучше — на Прологе.

1.3. Пролог II

Очень скоро появились большие и сложные программы. Возможности машины были довольно быстро исчерпаны: память, легкость использования, модульность. Поэтому мы приступили к разработке новой версии, имея в виду следующие цели:

— **переносимость**, которая благодаря виртуальной машине, распространяется на машины всех видов, включая и микромашины;

— **интерактивность**, включая редактор предложений (также написанный на Прологе);

— **модульность**, причем пространство предложений образует древесную иерархию подпространств, которая управляется специальными командами;

— **расширяемость**, с возможностью добавлять подпрограммы в язык включающей машины (арифметика, верстка, управление периферией), причем эти подпрограммы рассматриваются как вычислимые предикаты;

— **надежность**, с ударением на правильную передачу ошибок. Каждая ошибка должна быть доступна пользователю, на каком бы уровне она ни произошла.

Кроме того, во второй версии были введены и некоторые новые концепции.

Был введен предикат *заморозить*, позволяющий отложить исполнение какого-либо процесса до тех пор, пока не станет известна некоторая информация.

Появилась возможность с помощью нового предиката *dif* делать утверждение, что два еще неизвестных дерева являются и будут всегда оставаться различными.

Использование бесконечных деревьев позволило решить некоторым образом проблему проверки на вхождение переменных.

Улучшилось управление исполнением программы благодаря введению понятия блока.

Введена некоторая компактная организация данных, имеющая вид строк и n-ок.

1.4. Замечания о реализации языка

Чтобы обеспечить переносимость языка, мы разработали и реализовали:

— язык высокого уровня Кандид (Candide), который был создан специально для написания интерпретатора Пролога;

— виртуальную машину Микромегас (Micromegas), которая исполняет программу, порожденную компилятором языка Кандид.

Интерпретатор Пролога написан целиком на языке Кандид (приблизительно 70 страниц), так что результирующая программа может исполняться на любой машине, на которой реализована Микромегас.

Вся система была первоначально реализована на машине Apple II, а затем перенесена на машины

— Vax
— Multics
— Solar
— Mitra
— Dec-20

2. ОДИН ПРОСТОЙ ПРИМЕР

Пролог — это язык программирования, предназначенный для представления и использования знаний о некоторой предметной области. Более точно, предметная область — это множество объектов вместе со знанием о них, выраженном в виде множества отношений. Эти отношения описывают свойства объектов и связи между ними.

Некоторое множество правил, задающих объекты и отношения, и образует программу на Прологе.

Например, фраза «Жан является отцом Поля» есть утверждение о том, что некоторое отношение (является отцом) связывает два объекта (обозначенных своими именами — *Жан* и *Поль*); это можно записать в виде

является-отцом (*Жан*, *Поль*)

Подобным же образом вопрос: «Кто является отцом Поля?» сводится к поиску объекта, который отношением *является-отцом* связывается с Полем; этот объект и будет служить ответом на вопрос.

Отметим, что при определении отношения между объектами существен порядок, в котором эти объекты входят в отношение:

является-отцом (*Жан*, *Поль*) отличается от *является-отцом* (*Поль*, *Жан*).

Чтобы выразить отношение из нашего примера, мы использовали идентификаторы для обозначения объектов и отношений, которые их связывают.

Имя отношения *является-отцом* называется «предикатом», а объекты, которые связывает это отношение, называются его «аргументами».

2.1. В ресторане

Для иллюстрации того, как все это происходит в жизни, рассмотрим пример, описывающий меню ресторана.

Объекты, которые нас интересуют, — это блюда, которые можно съесть в ресторане, а первый набор отношений дает классификацию всех блюд на закуски, вторые мясные или рыбные блюда и десерты.

Меню представляет собой небольшую базу данных, которая записывается следующим образом:

закуска (*артишоки-в-белом-соусе*) → ;
закуска (*трюфели-в-шампанском*) → ;
закуска (*салат-с-яйцом*) → ;
мясо (*говяжье-жаркое*) → ;
мясо (*цыпленок-в-липовом-цвете*) → ;
рыба (*окунь-во-фритюре*) → ;
рыба (*фаршированный-судак*) → ;
десерт (*грушевое-мороженое*) → ;
десерт (*земляника-со-взбитыми-сливками*) → ;
десерт (*дыня-сюрприз*) → ;

Заданные выше отношения вводят одновременно и объекты и их классификацию. Например:

закуска (*салат-с-яйцом*) → ;

показывает, что *салат-с-яйцом* является закуской.

Этот первый тип правил употребляется для выражения простых утверждений.

Имея уже некоторый набор утверждений, можно задать такой вопрос:

«Правда ли, что *салат-с-яйцом* является закуской?» Он запишется на Прологе так:

закуска (*салат-с-яйцом*);

Затем нужно проверить, входит ли это утверждение во множество уже известных; ответом будет «да».

Наоборот,

закуска (*салат-из-помидор*);

получит отрицательный ответ, поскольку наш банк не содержит такого утверждения.

Теперь предположим, что мы хотим узнать все закуски. Слишком скучно задавать подряд вопросы:

закуска (*салат-из-помидор*);
...
закуска (*артишоки-в-белом-соусе*);
...
закуска (*земляника-со-взбитыми-сливками*);
...

и ожидать каждый раз ответ да или нет. Мы предпочли бы спросить: «Какие блюда являются закусками?» или еще лучше: «Каковы те объекты *е*, которые являются закусками?», не зная объекта, представленного именем *е*.

Здесь имя *е* обозначает не какой-то конкретный объект, а любой объект, принадлежащий множеству (может быть, пустому) тех объектов, которые обладают свойством быть закуской и которые нужно найти.

В этом случае говорят, что *е* есть «переменная».

Теперь наш вопрос запишется так:

закуска (*е*);

и Пролог ответит:

е = *артишоки-в-белом-соусе*
е = *трюфели-в-шампанском*
е = *салат-с-яйцом*

что является как раз множеством тех значений переменной *е*, при которых утверждение *закуска*(*е*) истинно.

С помощью отношений, которые составляют начальную базу данных, можно конструировать более сложные и более общие отношения. Например, с помощью отношений *мясо*( ) и *рыба*( ), выражающих то, что их аргумент является вторым мясным или рыбным блюдом, можно определить отношение *блюдо*( ): «блюдо — это второе мясное или рыбное блюдо». Это записывается в виде

блюдо (*р*) → *мясо* (*р*);
блюдо (*р*) → *рыба* (*р*);

и читается так:

р является блюдом, если *р* — второе мясное блюдо;
р является блюдом, если *р* — второе рыбное блюдо.

(Последовательность двух правил означает их дизъюнкцию).

Здесь вновь используется переменная p, которая в этих двух правилах обозначает соответственно любое второе мясное блюдо и любое второе рыбное блюдо.

Уточним, что область действия переменной ограничена правилом, в котором она определена. Поэтому переменная p из первого правила никак не связана с переменной p из второго.

В нашем примере вопрос: «Что является блюдом?», выраженный в виде

блюдо(p);

вызовет следующие ответы:

p = *говяжье-жаркое*
p = *цыпленок-в-липовом-цвете*
p = *окунь-во-фритюре*
p = *фаршированный-судак*

Займемся теперь составлением обеда, который, как обычно, состоит из закуски, основного блюда (мясного или рыбного) и десерта.

Обед является, следовательно, тройкой (e, p, d), где e — закуска, p — блюдо и d — десерт. В Прологе это выражается очень естественно в виде правила:

обед (e, p, d) → *закуска* (e) *блюдо* (p) *десерт* (d);

оно читается так:
e, p, d удовлетворяют отношению *обед*, если e удовлетворяет отношению *закуска*, p удовлетворяет отношению *блюдо* и d удовлетворяет отношению *десерт*.

Формально говоря, мы определили новое отношение как конъюнкцию трех других отношений.

На вопрос: «Что является обедом?», т. е.

обед(e, p, d);

Пролог ответит:

e = *артишоки-в-белом-соусе*	p = *говяжье жаркое* d = *грушевое-мороженое*
e = *артишоки-в-белом-соусе*	p = *говяжье-жаркое* d = *земляника-со-сливками*
...	
e = *артишоки-в-белом-соусе*	p = *фаршированный-судак* d = *дыня-сюрприз*
e = *трюфели-в-шампанском*	p = *говяжье жаркое* d = *грушевое-мороженое*

e = *трюфели-в-шампанском* *p* = *фаршированный-судак*
d = *дыня-сюрприз*
e = *салат-с-яйцом* *p* = *говяжье-жаркое*
d = *грушевое-мороженое*
...
e = *салат-с-яйцом* *p* = *фаршированный-судак*
d = *дыня-сюрприз*

(т. е. выдаст список из всех 36 возможных комбинаций).

Уточним вопрос, сохранив то же множество отношений: нас интересуют обеды с главным блюдом из рыбы. Этот вопрос имеет вид:

обед(*e*, *p*, *d*) *рыба*(*p*);

и состоит из конъюнкции двух условий, которые должны быть выполнены. Как обычно, Пролог найдет такие значения переменных *e*, *p* и *d*, что первое условие *обед*(*e*, *p*, *d*) будет удовлетворено. Заметим, что перед этой процедурой переменным *e*, *p* и *d* не было присвоено никакого значения.

Теперь же переменные *e*, *p* и *d* получили определенные значения, например:

e = *артишоки-в-белом-соусе* *p* = *говяжье-жаркое*
d = *грушевое мороженое*

Поэтому при переходе ко второму отношению *рыба*(*p*) со значением, которое приняла переменная *p*, мы получим:

рыба(*говяжье-жаркое*).

Поскольку наша база данных не содержит такого утверждения, то предложенный выбор значений не удовлетворяет нашему запросу — выбор неудачен и нужно пробовать следующее решение.

В результате программа напечатает 18 возможных решений:

e = *артишоки-в-белом-соусе* *p* = *окунь-во-фритюре*
d = *грушевое-мороженое*
e = *артишоки-в-белом-соусе* *p* = *окунь-во-фритюре*
d = *земляника-со-сливками*
e = *артишоки-в-белом-соусе* *p* = *окунь-во-фритюре*
d = *дыня-сюрприз*
e = *артишоки-в-белом-соусе* *p* = *фаршированный-судак*
d = *грушевое-мороженое*
e = *артишоки-в-белом-соусе* *p* = *фаршированный-судак*
d = *земляника-со-сливками*
e = *артишоки-в-белом-соусе* *p* = *фаршированный-судак*
d = *дыня-сюрприз*

e = *трюфели-в-шампанском* *p* = *окунь-во-фритюре*
d = *грушевое-мороженое*

...

e = *трюфели-в-шампанском* *p* = *фаршированный-судак*
d = *дыня-сюрприз*

e = *салат-с-яйцом* *p* = *окунь-во-фритюре*
d-*грушевое-мороженое*

...

e = *салат-с-яйцом* *p* = *фаршированный-судак*
d = *дыня-сюрприз*

Несколько замечаний:

(1) При попытке удовлетворить конъюнкцию отношений их просматривают слева направо.

(2) Во время выполнения программы некоторые переменные могут получить значения. В этом случае все вхождения такой переменной получают одно и то же значение.

(3) В отношении не различаются входные и выходные аргументы (входным называется аргумент, значение которого известно до присвоения значения отношению; выходной аргумент получает значение во время присвоения значения отношению).

С другой стороны, одно и то же отношение может быть использовано несколькими способами: если все его аргументы известны, то проверяется выполнимость отношения на этих аргументах; если же некоторые аргументы неизвестны, то вычисляется множество наборов значений этих переменных, которые удовлетворяют отношению.

(4) Выполнение программы недетерминированно: вычисляются все наборы значений аргументов, которые удовлетворяют отношению.

Пополним слегка наши знания о принятии пищи, введя значение калорийности для каждого кушанья.

«*калорийность одной порции*»

калории(*артишоки-в-белом-соусе*, *150*) →;
калории(*салат-с-яйцом*, *202*) →;
калории(*трюфели-в-шампанском*, *212*) →;
калории(*говяжье-жаркое*, *532*) →;
калории(*цыпленок-в-липовом-цвете*, *400*) →;
калории(*окунь-во-фритюре*, *292*) →;
калории(*фаршированный-судак*, *254*) →;
калории(*грушевое-мороженое*, *233*) →;
калории(*земляника-со-взбитыми-сливками*, *289*) →;
калории(*дыня-сюрприз*, *122*) →;

Утверждение

калории(фаршированный-судак, 254) →;

означает, что одна порция судака содержит 254 калории.

Чтобы узнать калорийность закусок, зададим вопрос: *закуска(e)калории(e, c)*;

Для каждого значения переменной *e*, удовлетворяющего отношению *закуска(e)*, программа сопоставит переменной *c* число, которое удовлетворяет отношению *калории(e,)*.

Мы получим ответы:

e = артишоки-в-белом-соусе c = 150
e = трюфели-в-шампанском c = 212
e = салат-с-яйцом c = 202

Более любопытный посетитель ресторана может захотеть узнать суммарную калорийность обеда. Для этого определим отношение

```
значение(e, p, d, v) →
    калории(e, x)
    калории(p, y)
    калории(d, z)
    сложить(x, y, z, v);
```

Здесь *v* — суммарная калорийность компонентов обеда. Чтобы ее узнать, зададим вопрос:

обед(e, p, d)значение(e, p, d, v); и получим

e = артишоки-в-белом-соусе p = говяжье-жаркое
d = грушевое-мороженое v = 905
e = артишоки-в-белом-соусе p = говяжье-жаркое
d = земляника-со-сливками v = 971
e = артишоки-в-белом-соусе p = говяжье-жаркое
d = дыня-сюрприз v = 804
...
e = артишоки-в-белом-соусе p = фаршированный судак
d = дыня-сюрприз v = 526
e = трюфели-в-шампанском p = говяжье-жаркое
d = грушевое-мороженое v = 967
e = трюфели-в-шампанском p = говяжье-жаркое
d = земляника-со-сливками v = 1033
...
e = салат-с-яйцом p = говяжье-жаркое
d = грушевое-мороженое v = 957
e = салат-с-яйцом p = говяжье-жаркое
d = земляника-со-сливками v = 1023

e = *салат-с-яйцом* *p* = *фаршированный-судак*
d = *дыня-сюрприз* *v* = *578*

Теперь естественно было бы определить сбалансированный обед:

сбалансированный-обед(*e*, *p*, *d*) →
обед(*e*, *p*, *d*)
значение(*e*, *p*, *d*, *v*)
меньше(*v*, *800*);

Это определение задает обед, калорийность которого меньше 800 калорий.

Вопрос:

сбалансированный-обед(*e*, *p*, *d*);

породит список:

e = *артишоки-в-белом-соусе* *p* = *цыпленок-в-липовом-цвете*
d = *грушевое-мороженое*
e = *артишоки-в-белом-соусе* *p* = *цыпленок-в-липовом-цвете*
d = *дыня-сюрприз*
...
e = *трюфели-в-шампанском* *p* = *окунь-во-фритюре*
d = *грушевое-мороженое*
...
e = *салат-с-яйцом* *p* = *фаршированный-судак*
d = *дыня-сюрприз*

Завершим обсуждение нашего примера вопросом:

сбалансированный-обед(*e*, *p*, *d*) *мясо*(*p*);

который позволяет выбрать сбалансированный обед с мясным блюдом. Мы получим ответы:

e = *артишоки-в-белом-соусе* *p* = *цыпленок-в-липовом-цвете*
d = *грушевое-мороженое*
e = *артишоки-в-белом-соусе* *p* = *цыпленок-в-липовом-цвете*
d = *дыня-сюрприз*
e = *трюфели-в-шампанском* *p* = *цыпленок-в-липовом-цвете*
d = *дыня-сюрприз*
e = *салат-с-яйцом* *p* = *цыпленок-в-липовом-цвете*
d = *дыня-сюрприз*

которые не содержат блюд из говядины!

Ниже приводится полная программа «МЕНЮ» (рис. 1).

```
«МЕНЮ»

закуска(артишоки-в-белом-соусе)→;
закуска(трюфели-в-шампанском)→;
закуска(салат-с-яйцом)→;
мясо(говяжье-жаркое)→;
мясо(цыпленок-в-липовом-цвете)→;
рыба(окунь-во-фритюре)→;
рыба(фаршированный-судак)→;
десерт(грушевое-мороженое)→;
десерт(земляника-со-взбитыми-сливками)→;
десерт(дыня-сюрприз)→;

«дежурное блюдо»

блюдо(р) → мясо(р);
блюдо(р) → рыба(р);

«состав обеда»

обед(е,р,d) → закуска(е)блюдо(р)десерт(d);

«калорийность порции»

калории(артишоки-в-белом-соусе, 150)→;
калории(трюфели-в-шампанском, 212)→;
калории(салат-с-яйцом, 202)→;
калории(говяжье-жаркое, 532)→;
калории(цыпленок-в-липовом-цвете, 400)→;
калории(окунь-во-фритюре, 292)→;
калории(фаршированный-судак, 254)→;
калории(грушевое-мороженое, 223)→;
калории(земляника-со-взбитыми-сливками, 289)→;
калории(дыня-сюрприз, 122)→;
«калорийность обеда»
значение(е,р,d,v)→

    калории(е,х)
    калории(р,у)
    калории(d,z)
    сложить(х,у,z,v);

«сбалансированный обед»
сбалансированный-обед(е,р,d)→

   обед(е,р,d)
   значение(е,р,d,v)
   менее(v, 800);

«разное»
сложить(a,b,c,d) — знач(слож(a,слож(b,c)),d);
менее(х,у)→знач(менее(х,у),1);
```

Рис. 1. Программа «МЕНЮ».

2.2. Теперь более подробно

Предыдущий пример позволил нам затронуть несколько важных моментов:

(1) определения отношений между объектами;

(2) запросы к этим отношениям;

(3) понятие переменной;
(4) представление конъюнкции и дизъюнкции отношений;
(5) недетерминизм выполнения программы;
(6) неразличение входных и выходных аргументов.

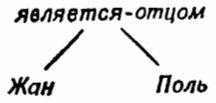
Теперь мы переходим к более детальному обсуждению этих пунктов.

2.2.1. Деревья

В нашем примере единственными объектами, с которыми мы имели дело, были константы, представленные либо их именами (*артишоки-в-белом-соусе*), либо их значениями (254). Кроме того, мы использовали переменные для обозначения неизвестных объектов.

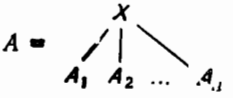
В действительности наиболее общей структурой, которой могут обладать объекты, является структура **дерева**. Например, дерево

изображает арифметическое выражение $23 + 45 * 60$. А дерево



изображает отношение *является-отцом*(*Жан*, *Поль*).

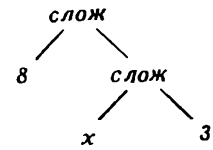
Вообще дерево A состоит из узла X, называемого **корнем**, и упорядоченного множества (может быть, пустого) **элементов** $A_1, \ldots, A_n$, которые сами являются деревьями:



Элементы $A_1, \ldots, A_n$ являются поддеревьями A, и всякое поддерево A_i является также и поддеревом A. Корни поддеревьев $A_1, \ldots, A_n$ являются потомками узла X. Узел, у которого нет потомков, называется *терминалом* или *листом*.

В Прологе дерево может быть частично неизвестно. В этом случае один из его листьев является переменной. Например,

дерево



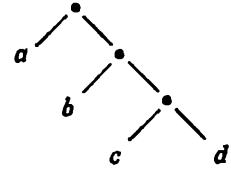
представляет бесконечное множество деревьев, получаемых заменой x на некоторое дерево.

Дерево без переменных может состоять из одного корня; в этом случае изображаемый объект есть константа — идентификатор, цепочка (последовательность) символов или число.

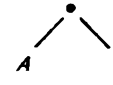
Линейная запись вышеприведенного дерева такова:

слож(8, *слож*(x, 3)), или в виде «n-ки»: ⟨*слож*, 8, ⟨*слож*, x, 3⟩⟩.

Среди других структур данных важную роль играют списки. Списки могут быть построены с помощью инфиксного оператора «.», обладающего ассоциативностью справа; тогда список $a.b.c.d$ представляет дерево



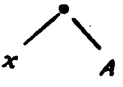
2.2.2. Унификация

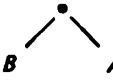
Главной операцией, применяемой к деревьям, является **унификация**. Если даны два дерева, которые, возможно, содержат переменные, то **унификация** состоит в нахождении таких значений этих переменных, при которых данные деревья совпадают. Множество равенств {переменная = значение}, которые делают эти деревья равными, называется подстановкой [1]). Например, деревья  и 

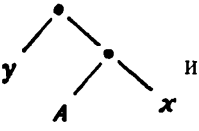
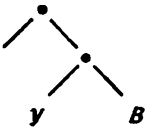
унифицируются подстановкой $\{x = B\}$.

[1]) Предполагается, что в левых частях равенств все переменные различны. — *Прим. ред.*

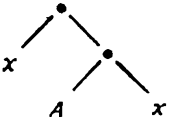
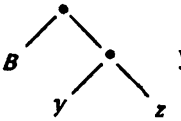
Форма, которая позволяет отождествить эти два дерева, есть

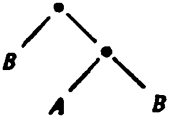
Деревья  и  **унифицируются в форму**

 подстановкой $\{x = B, y = A\}$.

Деревья  и  не унифицируются,

поскольку для этого нужно было бы взять $\{x = y,\ x = B,$

$y = A\}$. Деревья  и  **унифицируются**

в форму 

с помощью $\{x = B, y = A, z = B\}$.

2.3. Структура программы

В общем случае предикат может быть определен посредством некоторой смеси правил и (простых) утверждений и, следовательно, программа на Прологе — это последовательность правил.

Каждое правило состоит из **головы правила** и **хвоста правила** (может быть, пустого), соединенных знаком →.

Голова правила состоит из единственного **литерала** (предиката с его аргументами), а хвост правила — это последовательность литералов.

Программа на Прологе строится с помощью **термов**: правила, литералы, аргументы являются термами. Терм может

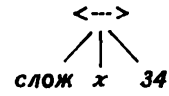
быть **константой** (идентификатором, числом, цепочкой символов), **переменной** или **структурным термом** (*n*-кой).

Понятие *n*-ки является очень полезным, поскольку с его помощью можно сгруппировать несколько термов, чтобы из них построить новый, с которым более удобно работать. Например, в предыдущем разделе мы определили обед, состоящий из трех составляющих: закуски *e*, основного блюда *p* и десерта *d*. Приятно иметь возможность сгруппировать эти составляющие в единый объект — обед, который является тройкой $\langle e, p, d\rangle$.

Этот новый терм имеет структуру дерева

где <---> есть трехместный функциональный символ.

Если первая составляющая *n*-ки (где $n \geqslant 2$) является идентификатором, то можно представить *n*-ку в функциональной форме. Таким образом,



есть то же самое, что и

Естественно, что тождества такого рода учитываются при унификации.

2.4. Основной механизм — стирание

После того как описан мир объектов, с которыми мы имеем дело, и отношений, которые их связывают, можно давать запросы к программе, требуя установить, удовлетворяется ли некоторое отношение (в общем случае — конъюнкция отношений) при заданных значениях аргументов.

Интуитивно говоря, Пролог старается «стереть» последовательность термов, представляющих эти отношения (обрабатывая их в том порядке, в котором они появляются в программе). Он несет с собой и по мере своего продвижения пополняет ту подстановку переменных, которая и делает возможным это стирание. Если это ему (Прологу) удается, то полученная подстановка и является ответом на поставленный вопрос.

В случае, когда несколько подстановок позволяет произвести стирание, т. е. когда существует несколько возможных ответов, все решения вычисляются и выдаются.

Принцип стирания можно объяснить следующим образом. Правило программы, имеющее вид

$$P(\ldots)\rightarrow Q(\ldots)R(\ldots);$$

можно проинтерпретировать так:

«чтобы стереть $P(\ldots)$, сотрите $Q(\ldots)$, а затем $R(\ldots)$».

Правило вида

$$S(\ldots)\rightarrow;$$

интерпретируется так: «$S(\ldots)$ себя стирает».

Запрос вида

$$S_1(\ldots)S_2(\ldots)\ldots S_n(\ldots)$$

означает: «сотрите $S_1(\ldots)$, затем $S_2(\ldots)\ldots$, а затем $S_n(\ldots)$».

Например, если правило

$$S_1(\ldots)\rightarrow Q_1(\ldots)\ldots Q_p(\ldots);$$

выбрано, то стереть $S_1(\ldots)$ и вышеприведенном запросе, наша новая цель — это стереть

$$Q_1(\ldots)\ldots Q_p(\ldots)S_2(\ldots)\ldots S_n(\ldots)$$

произведя необходимые подстановки.

Мы продолжаем в том же духе, учитывая порядок индексов, до тех пор, пока

(1) все будет стерто: это — успех, и решение материализуется с помощью подстановки, которая обеспечила стирание;

(2) мы наталкиваемся на терм, который невозможно стереть: это — неудача.

В обоих случаях на этом вычисления не кончаются. Вспомним, что для стирания терма мы выбирали первое правило, которое позволило это сделать. Может быть, оставались открытыми другие возможности, и по мере нашего продвижения мы их откладывали про запас. Теперь нужно вернуться назад, к последней развилке, и попробовать стереть данный терм другим способом, таким образом пытаясь найти другой ответ на наш запрос.

Мы продолжаем поступать так до тех пор, пока остается в запасе хотя бы один выбор. Важно отметить, что, когда мы делаем новую попытку стереть терм $L(\ldots)$, мы «возвращаем назад» все подстановки переменных, которые имели

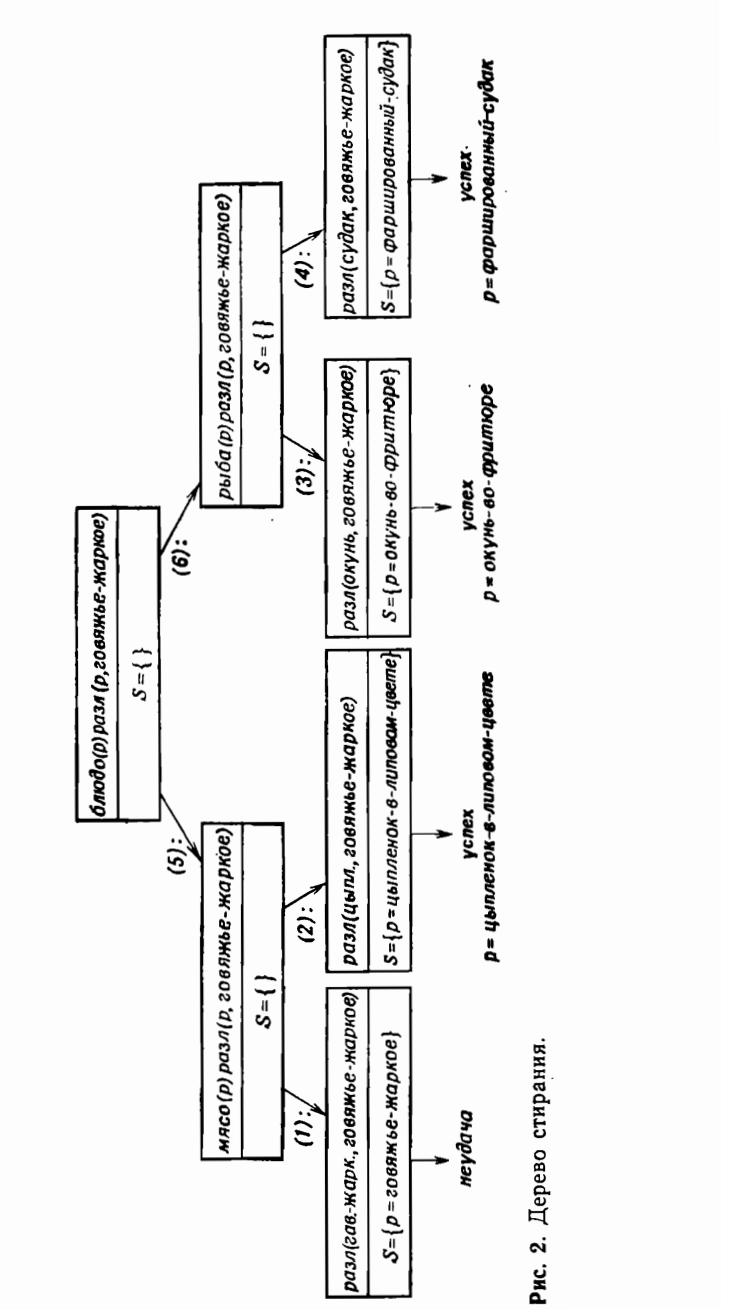


Рис. 2. Дерево стирания.

место на пути от предыдущего выбора к настоящему моменту.

Для иллюстрации этого механизма возьмем начало нашего примера:

(1) *мясо(говяжье-жаркое)* →
(2) *мясо(цыпленок-в-липовом-цвете)* →;
(3) *рыба(окунь-во-фритюре)* →;
(4) *рыба(фаршированный-судак)* →;

«дежурное блюдо»

(5) *блюдо(р)* → *мясо(р)*;
(6) *блюдо(р)* → *рыба(р)*;

и запрос: *блюдо(р)* разл(*р*, *говяжье-жаркое*);

который дает нам начальное множество целей, причем *разл* — это терм, который себя стирает в том и только том случае, когда его аргументы различны.

Мы представим процесс стирания с помощью дерева, в котором:
(1) каждому узлу сопоставлено текущее множество термов для стирания (целей) и текущая подстановка;
(2) каждой дуге сопоставлено правило, выбранное для стирания первого терма;
(3) потомки каждого узла — это новые множества целей, порожденные стиранием первого терма этой вершины.

Дерево, соответствующее предыдущему примеру, приведено на рис. 2.

2.5. Встроенные правила

Некоторые правила изначально известны системе, а не определяются пользователем. Они называются «предопределенными» или «встроенными» правилами и обладают такими возможностями, которые нельзя, вообще говоря, получить с помощью чистого Пролога. В действительности именно существование этих правил и делают Пролог реальным языком программирования.

Эти правила часто имеют и **«побочные эффекты»**, т. е. при их использовании для стирания они могут изменять не только аргументы стираемого терма, но также и некоторое системное окружение.

Встроенные правила касаются главным образом следующих пунктов:
(1) управление процессом вычисления (законы стирания, недетерминированность и т. д.);
(2) ввод и вывод;

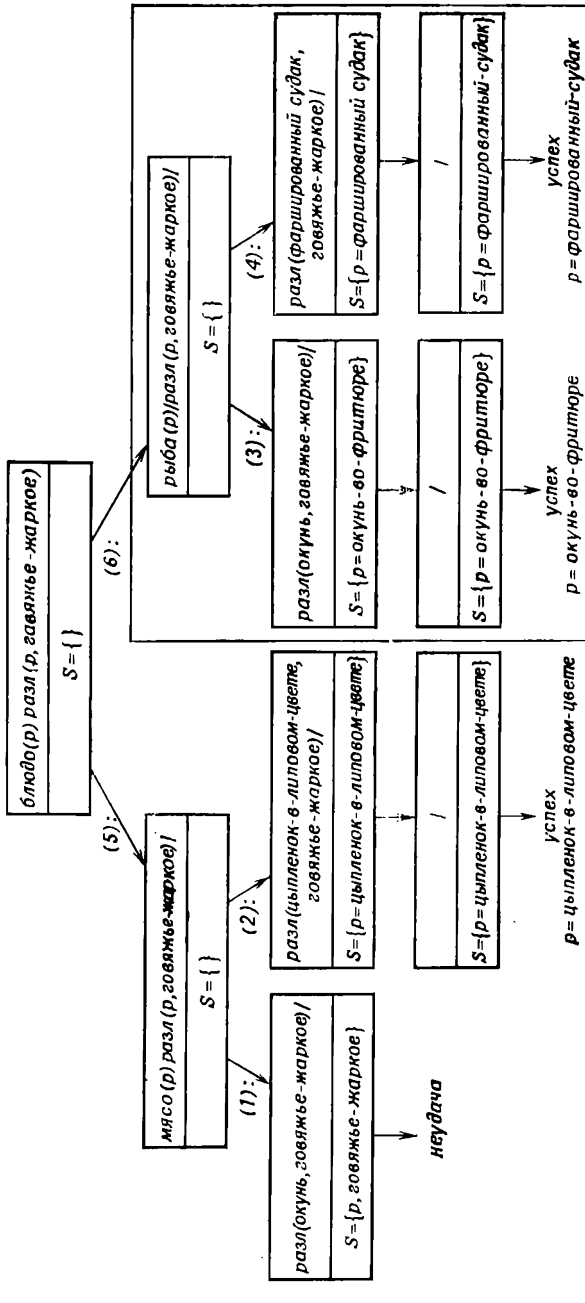


Рис. 3. Дерево стирания с использованием оператора «/».

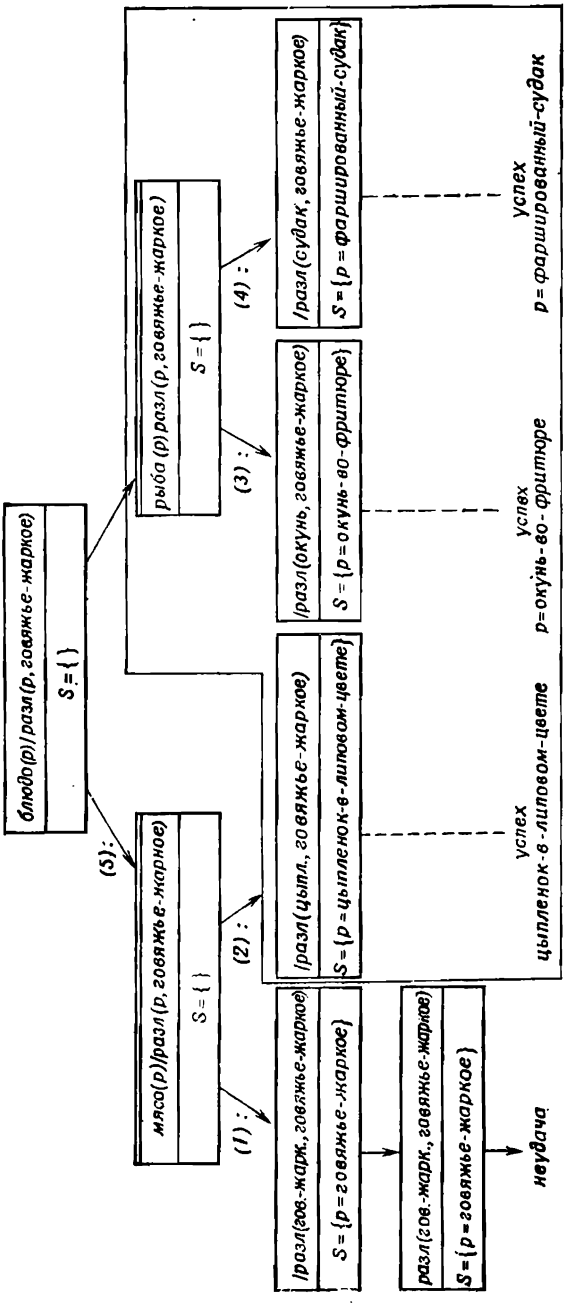


Рис. 4. Другой пример.

(3) проверка типа объекта;
(4) арифметика;
(5) подмиры;
(6) подпрограммы.

Оператор усечения «/»

Мы видели, что стирание терма происходит недетерминированным образом, причем система хранит различные возможности выбора, чтобы к ним вернуться позднее.

Введение оператора «/» в правило позволяет подавить некоторые из таких возможностей, а в отдельных случаях сделать программу полностью детерминированной. Мы получаем тогда «исконный» механизм Пролога. Правило использования «/» можно выразить следующим образом:

> Правило оператора "/"
> Стирание "/" подавляет все отложенные возможности выбора для всех термов, подлежащих стиранию, начиная с того, который активизировал правило, содержащее "/", и кончая тем, который предшествует "/" в хвосте этого правила.

Например, если «/» употребляется в запросе:

блюдо(*p*)*разл*(*p*, *говяжье-жаркое*)/;

то дерево на рис. 2 превратится в дерево на рис. 3.

Стирание оператора/подавит все возможности выбора для *блюдо*(...) — часть дерева, заключенная в рамку, будет игнорироваться, и мы получим единственный ответ на запрос — первый найденный.

Если задать вопрос

блюдо(*p*)/*разл*(*p*, *говяжье-жаркое*);

наше дерево примет вид, изображенный на рис. 4.

Стирание «/» удалит часть дерева, заключенную в рамку, и не останется ни одного ответа, удовлетворяющего запросу.

3. ПРИМЕНЕНИЯ ПРОЛОГА: ПРОГРАММЫ С КОММЕНТАРИЯМИ

Предыдущий пример позволил нам познакомить читателя с общей формой программы на Прологе, а также с главным используемым понятием (деревья) и основным механизмом (стирание).

А теперь с помощью более серьезных примеров мы продемонстрируем разнообразие применений этого языка.

3.1. Искусство спряжения

Вначале мы рассмотрим основные элементы программы, которая способна спрягать все французские глаголы во всех простых временах изъявительного наклонения.

Кроме того, эта же программа может анализировать глагол, т. е. определить форму инфинитива, время и лицо по личной форме этого глагола.

Эта программа обладает практически той же информацией, что и справочник «Бешерель» (Бешерель, «Искусство спряжения», словарь 12000 глаголов), из которого мы выбрали четыре простых времени изъявительного наклонения французских глаголов.

Основное используемое правило таково:

спряжение(v, t p, w) →
    возможные-формы(v, k)
    время(t, k, u)
    лицо(p, u, w);

где
v — глагол в форме инфинитива,
t — время (настоящее, имперфект, простое прошедшее, простое будущее),
p — лицо (единственное или множественное, 1-е, 2-е или 3-е лицо),
w — личная форма глагола v в p-м лице во времени t.

Чтобы получить эту форму, нужно обратиться к таблице типов спряжений глаголов и определить список k личных форм глагола v. Эта таблица представляется множеством правил вида:

возможные-формы(. . . ,
    изъяв-наст.(. . .).
    изъяв-имп(. . .).
    изъяв-прош(...).
    изъяв-буд(...). *nil*) → . . . ;

причем
(1) первый аргумент выявляет окончание, характерное для данного семейства глаголов, и основу;
(2) второй аргумент — список форм спряжения; эти формы используют типы спряжений, которые мы определим несколько позже;
(3) применение правила может быть ограничено условиями на форму основы глагола, которые могут появиться во втором аргументе.

Вернемся к нашей задаче. Выбираем правило, которое лучше всего подходит к глаголу v, который надо проспря-

гать — это правило, содержащее наибольшее общее окончание с v и удовлетворяющее ограничениям на переменные.

Получаем в переменной k возможные формы v в четырех простых временах изъявительного наклонения. Выделяем нужное нам время t с помощью стирания

время (t, k, u)

применением правила

время (t, k, u) — *входит* $(\langle t, r, x, u\rangle, k)\langle t, r, x, u\rangle$.

В терме $\langle t, r, x, u\rangle$

t — элемент множества {*изъяв-наст*, *изъяв-имп*, *изъяв-прош*, *изъяв-буд*},

r — основа нашего глагола v,

x — определяет тип спряжения во временн t, которому подчиняется глагол v,

u — шестерка форм спряжения глагола v во всех лицах во времени t.

Наш поиск оканчивается выделением из u нужного нам лица с помощью стирания терма *лицо*(p, u, w), что и дает в результате форму w.

Остается добавить, что правила, служащие для стирания $\langle t, r, x, u\rangle$ находятся под рубрикой «типы окончаний изъявительного наклонения» в приведенной ниже программе; эти правила разбиты на следующие группы:

4 типа окончаний для единственного числа настоящего времени,

11 — для множественного настоящего,

5 — для имперфекта,

10 — для простого прошедшего,

8 — для простого будущего.

В действительности мы не будем использовать непосредственно отношение *спряжение*, а возьмем вместо этого отношение *красное-спряжение* (v, t, p), которое преобразует v в обратный список v' букв, составляющих v, и, используя *спряжение* (v', t, p, w), вычислит и красиво напечатает w с соответствующим местоимением.

Хотя эта программа была задумана и написана для синтеза, она годится и для анализа, т. е., получив значение w, она вычисляет такие v, t и p, что выполняется отношение *спряжение* (v, t, p, w).

И в этом случае, чтобы получить удобное представление, мы переходим к отношению *красивый-анализ*(w), которое по личной форме w некоторого глагола строит обратный список w' букв формы w, а затем, используя *спряжение* (v, t, p, w'),

вычисляет и красиво печатает найденные значения для *v*, *t* и *p*.

Рассмотрим несколько примеров. Если задать только *v* и *t* для отношения *красивое-спряжение*, то мы получим все формы спряжения *v* во времени *t*; если задать только *v*, то выдаются все формы во всех временах и для всех лиц.

> *красивое-спряжение* («*obtenir*», *изъяв-наст*, *множ*-1-*е-лицо*); nous obtenons

> *красивое-спряжение* («*conquérir*», *изъяв-имп*);

je conquérais
tu conquérais
il conquérait
nous conquérions
vous conquériez
ils conquéraient

> *красивое-спряжение* («*consentir*»);

je consens
tu consens
il consent

А теперь несколько примеров, когда то же отношение *спряжение* используется в обратном направлении, т. е. для анализа.

> *красивый-анализ* («*tenais*»);
tenir изъяв-имп ед-1-е-лицо
tenir изъяв-имп ед-2-е-лицо

> *красивый-анализ* («*finis*»);

finir изъяв-наст ед-1-е-лицо
finir изъяв-наст ед-2-е-лицо
finir изъяв-прош ед-1-е-лицо
finir изъяв-прош ед-2-е-лицо

> *красивый-анализ* («*crumes*»);

croitre изъяв-прош множ-1-е-лицо
croitre изъяв-прош множ-1-е-лицо

> *красивый-анализ* («*mirent*»);
mettre изъяв-прош множ-3-е-лицо
mirer изъяв-наст множ-3-е-лицо

На рис. 5 представлены наиболее важные фрагменты из соответствующей программы.

«Возможные формы глаголов»

```
"вначале — вспомогательные глаголы etre и avoir"
"затем глаголы третьей, второй и первой групп"

возможные-формы ("e"."r"."t"."e".nil,etre,
                 изъяв-наст (nil,type-etre,i-pres).
                 изъяв-имп ("i"."e"."nil",type-1,i-imp).
                 изъяв-прош ("f".nil,type-3,i-ps).
                 изъяв-буд ("e"."s".nil,type-1,i-fs).nil)→;
возможные-формы ("r"."i"."o"."v"."a".nil,avoir),
                 изъяв-наст nil,type-avoir,i-pres).
                 изъяв-имп ("v"."a"."nil",type-1,i-imp).
                 изъяв-прош ("e".nil,type-3,i-ps).
                 изъяв-буд ("u"."a".nil,type-1,i-fs).nil)→;
возможные-формы ("r"."e"."l"."l"."a".nil,aller,
                 изъяв-наст (nil,type-aller,i-pres).
                 изъяв-имп ("l"."l"."a"."nil",type-1,i-imp).
                 изъяв-прош ("l"."l"."a"."nil",type-4,i-ps).
                 изъяв-буд ("i".nil,type-1,i-fs).nil)→;
возможные-формы ("r"."i"."n"."e".x.q,venir.tenir,
                 изъяв-наст (⟨"n"."e"."i".x.q,x.q⟩,
                             ⟨type-s-1,type-p-8⟩,i-pres).
                 изъяв-имп ("n"."e".x.q,type-1,i-imp).
                 изъяв-прош ("n"."i".x.q,type-1,i-ps).
                 изъяв-буд ("d"."n"."e"."i".x.q,type-1,i-fs).nil)
→ входит (x,"t"."v",nil);
возможные-формы ("r"."i"."r"."e"."u"."q".q,acquerir,
                 изъяв-наст ("r"."e"."i"."u"."q".q,"u"."q".q⟩,
                             ⟨type-s-1,type-p-9⟩,i-pres).
                 изъяв-имп ("r"."e"."i"."u"."q".q,type-1,i-imp).
                 изъяв-прош ("u"."q".q,type-2,i-ps).
                 изъяв-буд ("r"."e"."u"."q".q,type-1,i-fs).nil)→;
возможные-формы ("r"."i"."t".x.q,sentir.sortir,
                 изъяв-наст (⟨x.q,"i".x.q⟩,⟨type-s-1,type-p-9⟩,i-pres).
                 изъяв-имп ("t".x.q,type-1,i-imp).
                 изъяв-прош ("t".x.q,type-2,i-ps).
                 изъяв-буд ("t"."t".x.q,type-1,i-fs).nil)
→ входит (x,"n"."r".nil);
возможные-формы ("r"."i".".r".x.q,ouvrir.souffrir,
                 изъяв-наст (⟨"e"."r".x.q,"r".x.q⟩,
                             ⟨type-s-4,type-p-1⟩,i-pres).
                 изъяв-имп ("r".x.q,type-1,i-imp).
                 изъяв-прош ("r".x.q,type-2,i-ps).
                 изъяв-буд ("i"."r".x.q,type-1,i-fs).nil)
→ входит (x,"f"."v".nil);
и т. д.

спряжение (v,t,p,w) → возможные формы (v,e,k) время (t,k,u)
                      лицо (p,u,w);
время (t,k,u) → входит (⟨t,r,x,u⟩,k)⟨t,r,x,u⟩;
лицо (ед-1-е-лицо,⟨j,t,i,n,v,i'⟩,j)→;
лицо (ед-2-е-лицо,⟨j,t,i,n,v,i'⟩,t)→;
лицо (ед-3-е-лицо,⟨j,t,i,n,v,i'⟩,i)→;
лицо (мн-1-е-лицо,⟨j,t,i,n,v,i'⟩,n)→;
лицо (мн-2-е-лицо,⟨j,t,i,n,v,i'⟩,v)→;
лицо (мн-3-е-лицо,⟨j,t,i,n,v,i'⟩,i')→;
```

```
«чтобы было красиво»
красивое-спряжение (v,t,p) → инверсия (v,v') строка
                             спряжение (v',t'p,w')
                             красивый-вывод (p,w') строка;
красивое-спряжение (v,t) → красивое-спряжение (v,t,p);
красивое-спряжение (v) →   красивое-спряжение (v,t,p);
красивый-анализ (v) → инверсия (v,v') строка
                      спряжение (w,t'p,v') красивый-вывод' (w)
                      вывц (" ") выв (t) вывц (" ") выв (p) строка;
красивый-вывод (p,u.nil) → согласовано (p,u,p') вывц (p') вывц (" ")
                           вывц (u);
красивый-вывод (p,v.w) → разл (w,nil) красивый-вывод (p,w) вывц (v);
красивый-вывод' (nil) →;
красивый-вывод (a.b) → красивый-вывод' (b) вывц (a);
согласовано (ед-1-е-лицо,"j'") → входит (u,"a"."e"."i"."o"."u".nil)/;
согласовано (ед-1-е-лицо,u,"je") →;
согласовано (ед-2-е-лицо,u,"tu") →;
согласовано (ед-3-е-лицо,u,"il") →;
согласовано (мн-1-е-лицо,u,u,"nous") →;
согласовано (мн-2-е-лицо,u,"vous") →;

«разное»
входит (a,a.x) →;
входит (a,b.x) → входит (a,x);
инверсия (v,w) → арг (0,v,l) инверсия' (l,v,w);
инверсия' (0,v,nil) →;
инверсия' (l,v,d.w) → разл (l,0) арг (l,v,d) знач (выч (l,1),l')
                      инверсия' (l',v,w);
не-начинается-с (a.l,nil) →;
не-начинается-с (a.l,b.l') → разл (a,b);
не-начинается-с (a.l,a,l') → не-начинается-с (l,l');
не-начинаются-с (nil,q) →;
не-начинаются-с (a.b,q) → не-начинается-с (a,q)
                          не-начинаются-с (b,q);

«типы окончаний изъявительного наклонения»
«настоящее время»

изъяв-наст (nil,type-etre,⟨"s"."i"."u"."s".nil,
                           "s"."e".nil,
                           "t"."s"."e".nil,
                           "s"."e"."m"."m"."o"."s".nil,
                           "s"."e"."t"."e".nil,
                           "t"."n"."o"."s".nil⟩) →;
изъяв-наст (nil,type-avoir,⟨"t"."a".nil,
                            "s"."a".nil,
                            "a"..nil,
                            "s"."n"."o"."v"."a".nil,
                            "z"."e"."v"."a".nil,
                            "t"."n"."o".nil⟩) →;
изъяв-наст (nil,type-aller,⟨"s"."i"."a"."v".nil,
                            "s"."a"."v".nil,
                            "a"."v".nil,
                            "s"."n"."o"."l"."l"."a".nil,
                            "z"."e"."l"."l"."a".nil,
                            "t"."n"."o"."v".nil⟩) →;
```

```
изъяв-наст (⟨s,p⟩,⟨t-s,t-p⟩,⟨j,t,i,n,v,i'⟩)→
изъяв-наст-ед (s,t-s,⟨j,t,i⟩ изъяв-наст-мн (p,t-p,⟨n,v,i'⟩));

«настоящее единственное»

изъяв-наст-ед (p,type-s-1,⟨"s".p."s".p,"t".p⟩)→;
изъяв-наст-ед (p,type-s-2,⟨"s".p."s".p,p⟩)→;
изъяв-наст-ед (p,type-s-3,⟨"x".p,"x".p,"t".p⟩)→;
изъяв-наст-ед (p,type-s-4,⟨p,"s".p,p⟩);

«настоящее множественное»

изъяв-наст-мн (p,type-p-1,⟨"s"."n"."o".p,"z"."e".p,"t"."n"."e".p⟩)→;
изъяв-наст-мн (p,type-p-2,⟨"s"."n"."o"."y".p,
                           "z"."e"."y".p,
                           "t"."n"."e"."i".p,⟩)→;
изъяв-наст-мн (p,type-p-3,⟨"s"."n"."o".x."u"."o".p,"z",
                           "e"."x"."u"."o".p,
                           "t"."n"."e".x."u"."e".p⟩)→;
входит (x,"l"."v".nil);
(и т. д.)
```

Рис. 5. Искусство спряжения.

3.2. Мутанты

Наш второй пример — программа, способная производить «мутантов», т. е. гибридов различных животных.

Животные задаются их названиями в форме цепочки букв. Два животных производят на свет мутанта, если окончание названия первого из них совпадает с началом названия второго.

Интересный аспект этой программы заключается в использовании отношения *конк* двумя различными способами: с одной стороны, для объединения двух списков, а с другой — для разложения списка на два подсписка.

Вот результаты, полученные на множестве животных {*крокодил*, *черепаха*, *карибу*, *лошадь*, *хамелеон*, *буйвол*, *волк*}[1]):

красивые-мутанты;
крокодилошадь
буйволк
карибуйвол
черепахамелеон
волкрокодил
волкарибу
буйволошадь

Эта программа приведена на рис. 6.

[1]) Я позволил себе изменить состав животных, с тем чтобы получить интересные «русские» мутанты. — *Прим. перев.*

```
«МУТАНТЫ»
мутант(z)→
   животное(x)
   животное(y)
   конк(a,b,x)
   разл(b,nil)
   конк(b,c,y)
   разл(c,nil)
   конк(x,c,z);
конк(nil,y,y)→;
конк(e.x,y,e.z) → конк(x,y,z);
красивый-мутант → мутант(z) выр(z);
выр(nil)→;
выр(a.l) → вывц(a) выр(l);
животное("к"."р"."о""к"."о"."д"."и"."л".nil)→;
животное("ч"."е"."р"."е"."п"."а"."х"."а".nil)→;
животное("к"."а"."р"."и"."б"."у".nil)→;
животное("л"."о"."щ"."а"."д"."ь".nil)→;
животное("х"."а"."м"."е"."л"."е"."о"."к".nil)→;)
животное("б"."у"."й"."в"."о"."л".nil)→;
животное("в"."о"."л"."к".nil)→;
```

Рис. 6. Мутанты.

3.3. Запросы к базе данных

Третий пример относится к интересной задаче — приложению математической логики к проблемам, связанным с запросами в базе данных; он показывает, как можно нспользовать Пролог для этих целей.

Предположим, что у нас есть база данных, содержащая для каждого человека его фамилию, возраст, место рождения и сведение о том, носит ли он очки. Все эти данные заключены в утверждении вида

человек (*кандид*, *20*, *константинополь*, *нет*)→;

в котором говорится, что человеку по фамилии *кандид* 20 лет, что он родился в *константинополе* и не носит очков.

Мы имеем в своем распоряжении также элементарные отношения (атомарные формулы), определенные на этих данных.

Программа заключается в вычислении значения логической формулы, построенной из атомарных с использованием связок «и», «или» и кванторов общности и существования на типизированных переменных, т. е. на переменных, для каждой из которых определена ее область значений.

Типичный запрос имеет вид: «каковы значения x, принадлежащие области D, для которых свойство P истинно?»; он запишется так:

элемент (*x*, *множ*(*x*, *D*, *P*)).

«(1) база данных»

человек (кандид, 20, константинополь, нет)→;
человек (кунегонда, 20, константинополь, да)→;
человек (гонтран, 94, ростов-на-дону, нет)→;
человек (казимир, 2, старая-крепость, да)→;
человек (губернатор, 1, глупов, нет)→;
человек (папайа, 99, ростов-на-дону, да)→;
человек (маслина, 99, ростов-на-дону, нет)→;
человек (мимоза, 99, ростов-на-дону, да)→;
человек (бип, 15, тьмутаракань нет)→;
человек (игнатий, 114, лойола, да)→;
человек (бальтазар, 87, иерусалим, нет)→;
человек (гаспар, 96, смирна, да)→;
человек (гаспар, 96, смирна, да)→;
человек (мельхиор, 34, хартум, нет)→;

«(2) определение типов»

тип (x,имя) → имя (x);
тип (x,возраст) → возраст (x);
тип (x,город) → город (x);
тип (x,логический) → логический (x);

имя (кандид)
имя (кунегонда)
имя (гонтран)
имя (казимир)
имя (губернатор)
имя (папайа)
имя (маслина)
имя (мимоза)
имя (бип)
имя (игнатий)
имя (бальтазар)
имя (гаспар)
имя (мельхиор)

лет (20)→;
лет (94)→;
лет (2)→;
лет (1)→;
лет (99)→;
лет (15)→;
лет (114)→;
лет (87)→;
лет (96)→;
лет (34)→;

город (константинополь)→;
город (ростов-на-дону)→;
город (старая-крепость)→;
город (глупов)→;
город (тьмутаракань)→;
город (лойола)→;
город (иерусалим)→;
город (смирна)→;
город (хартум)→;

```
логический (да)→;
логический (нет)→;

                    «(3) список атомарных формул»

атомарная (живет-в (x,y))→;
атомарная (возраст (x,y))→;
атомарная (очки (x,y))→;
атомарная (старше (x,y)→;
атомарная (меньше (x,y))→;
атомарная (различные (x,y))→

                    «(4) значение атомарных формул»

живет-в (x,y)→человек (x,a,y,b);
возраст (x,y)→человек (x,a,v,b);
очки (x,y)→человек (x,a,v,y);
старше (x,y)→
    человек (x,a,v,b)
    человек (x,a',v',b')
    знач (меньше (a',a),1);
меньше (x,y)→знач (меньше (x,y),1);
различные (x,y))→разл (x,y);

                    «(5) значение формул»

истинная (p)→
    атомарная (p)
    p;
истинна (не (p)→
    не (истинна (p));
истинна (и (p,q))→
    истинна (p)
    истинна (q);
истинна (или (p,q))→
    истинна (p);
истинна (или (p,q))→
    истинна (q);
истинна (существует (x,t,p))→
    тип (x,t)
    истинна (p);
истинна (для-всех (x,t,p))→
    не (истинна (существует (x,t,нв (p))));

                    "(6) полезные определения"

не (p)→
    p
    /
    тупик;
не (p)→;

                    "(7) вычисление ответов"

ответы-на-все→
    запрос (i,q)
    элемент (y,q)
    строка;
```

```
    вывц ("-->")
    выв (y)
    строка;
элемент (x, множ (x,t,p))->
    тип (x,t)
    истинна (p);

                    "(8) список запросов"

запрос (1,множ (x,город, живет-в (мимоза,x)))->
    вывц ("(1) в каком городе живет мимоза?");
запрос (2, множ (x, логический, очки (маслина, x)))->
    вывц ("(2) носит ли маслина очки?");
запрос (3, множ (x, город, существует (y, имя, и (живет-в (y,x),
        и (возраст) (y,a),
    и (меньше (a, 20), очки (y, да)))))))->
    вывц ("(3) каковы те города, в которых живет по крайней мере один")
    вывц ("житель моложе 20 лет, носящий очки?");
```

Рис. 7. База данных.

Например, запрос «в каком городе живет мимоза?» задается формулой:

элемент(*x*, *множ*(*x*, *город*, *живет-в*(*мимоза*, *x*))),

а запрос «носит ли маслина очки?» — формулой:

элемент(*x*, *множ*(*x*, *логический*, *очки*(*маслина*, *x*))).

И наконец, запрос «каковы те города, в которых живет по крайней мере один житель моложе 20 лет, носящий очки?» запишется так:

элемент(*x*, *множ*(*x*, *город*, *существует*(*y*, *фамилия*, *и*(*живет-в* (*y*, *x*), *и*(*возраст*(*y*, *a*), *и*(*меньше*(*a*, *20*), *очки*(*y*, *да*))))))).

Эти три запроса были заранее записаны в машину, и их можно вызвать с помощью терма *ответы-на-все*, который выдает целиком запрос, за которым следуют ответы.

Это выглядит так (ответы машины начинаются со знака -->);

```
> ответы-на-все;
в каком городе живет мимоза?
--> ростов на дону
носит ли маслина очки?
--> нет
каковы те города, в которых живет по крайней мере один
житель моложе 20 лет, носящий очки?
--> каменец-подольский
--> ростов на дону
```

Полный текст программы приведен на рис. 7.

3.4. Головоломка

В следующем примере речь идет об известной зашифрованной арифметической задачке, в которой требуется сопо-

```
«РЕШЕНИЕ ВОЛОДЯ + ДАРЬЯ = ЛЮБОВЬ»
решение(в.о.л.д.я.б.а.р.ь.ю)→
    различные(в.о.л.д.я.а.р.ь.ю.nil)
    сумма(r1,в,о,л,о)
    сумма(r2,о,д,ю,r1)
    сумма(r3,л,а,б,r2)
    сумма(r4,о,р,о,r3)
    сумма(r5,л,ь,в,r4)
    сумма(о,я,я,ь,r5);
сумма(x,o,o,x,o) → /перенос(x);
сумма(r,x,y,z,r')→
    перенос(r)
    цифра(x)
    цифра(y)
    знач(слож(r,слож(x,y)),t)
    знач(дел(t,10),r')
    знач(мод(t,10),z);
цифра(0)→;
цифра(1)→;
цифра(2);→
цифра(3)→;
цифра(4)→;
цифра(5)→;
цифра(6)→;
цифра(7)→;
цифра(8)→;
цифра(9)→;
перенос(1)→;
перенос(0)→;
различные(nil)→;
различные(x.l) → вне(x,l) различные(l);
вне(x,nil)→;
вне(x,a.l) → разл(x,a) вне(x,l);
красивое-решение → решение(s) красивый-вывод(s);
красивый-вывод(в.о.л.д.я.б.а.р.ь.ю)→
    вывц(" ") выв(в) выв(о) выв(л) выв(о) выв(д) выв(я) строка
    вывц("+") выв(д) выв(а) выв(р) выв(ь) выв(я) строка
    вывц("----------") строка
    выв(л) выв(ю) выв(б) выв(о) выв(в) выв(ь);
```

Рис. 8. *ВОЛОДЯ + ДАРЬЯ = ЛЮБОВЬ.*

ставить каждой букве *в, о, л, д, я, а, р, ь, ю, б* цифру таким образом, чтобы получить

ВОЛОДЯ + ДАРЬЯ = ЛЮБОВЬ [1])

При традиционном программировании пришлось бы иметь дело одновременно с двумя проблемами: собственно сложе-

[1]) В оригинале — *SEND + MORE = MONEY* (*ШЛИ + БОЛЬШЕ = = ДЕНЕГ* (англ.)). — *Прим. перев.*

ние и условие, требующее заменять разные буквы разными цифрами.

Здесь же из-за отложенного отношения *разл* эти две проблемы легко разделяются: предикат *различные* выставляет все значения *разл* заранее. Остается только описать сложение столбиком, а пары отношения *разл* будут разблокироваться постепенно, по мере решения задачи. Программа становится более прозрачной, а также и более эффективной.

А вот и решение:

красивое-решение

```
  243 458
+  57 968
---------
  301 426
```

Программа приведена на рис. 8.

3.5. Синтаксический анализатор

3.5.1. Описание задачи

Следующий пример — синтаксический анализатор для некоторого языка. Интересная особенность этой программы состоит в том, что она написана в форме, наиболее приближенной к определению грамматики.

```
(0) ⟨выражение⟩ :: = ⟨сумма⟩
(1) ⟨сумма⟩ :: = ⟨произведение⟩ ⟨хвост суммы⟩
(2) ⟨произведение⟩ :: = ⟨сомножитель⟩ ⟨хвост произведения⟩
(3) ⟨сомножитель⟩ :: = ⟨число⟩
(4)                 :: = (⟨выражение⟩)
(5) ⟨хвост суммы⟩ :: = ⟨оп слож⟩ ⟨произведение⟩ ⟨хвост суммы⟩
(6)                 :: = ⟨пусто⟩
(7) ⟨хвост произведения⟩ :: = ⟨оп умнож⟩ ⟨сомножитель⟩ ⟨хвост произведения⟩
(8)                      :: = ⟨пусто⟩
(9) ⟨оп умнож⟩ :: = *
(10) ⟨оп слож⟩ :: = +
(11)           :: = -
(12) ⟨число⟩ :: = 0
             :: = 1
             :: = 2
.....
(13) ⟨пусто⟩ :: =
```

Рис. 9. Грамматика выражений.

Мы используем классическое описание грамматики в форме Бэкуса — Наура (БНФ). Бесконтекстная грамматика задается множеством правил вида

$$A ::= a_1 a_2 \ldots a_n$$

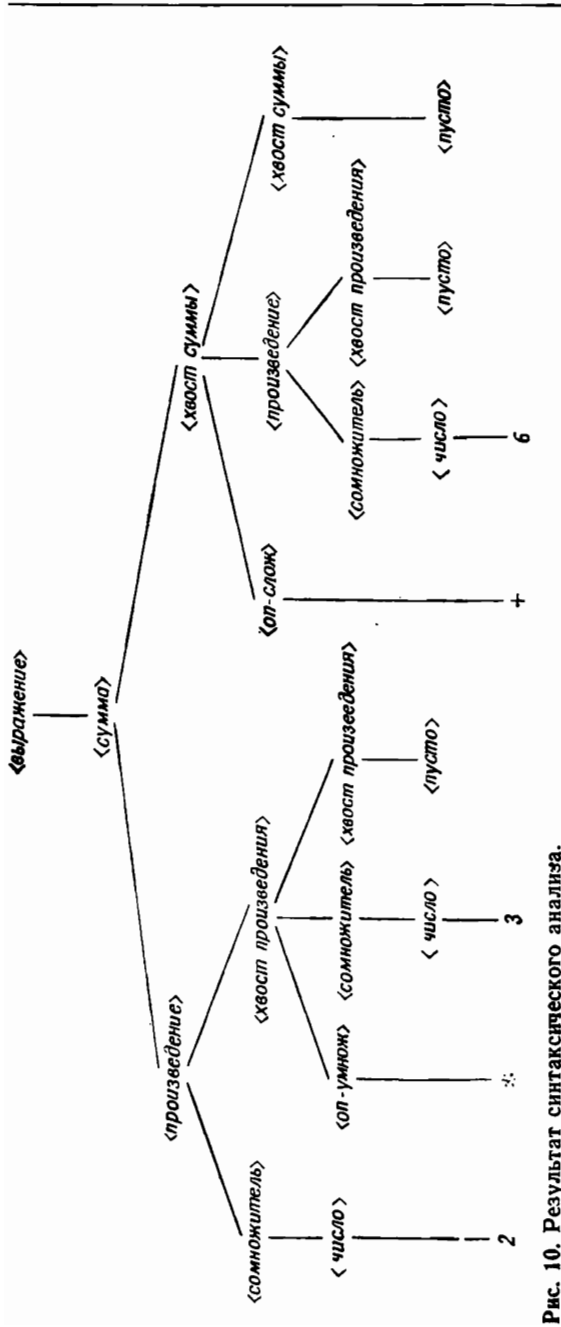


Рис. 10. Результат синтаксического анализа.

где A — нетерминал (нетерминальный символ), а a_1, a_2, ..., a_n — нетерминалы или терминалы (терминальные символы).

Так, приведенная на стр. 61 (рис. 9) грамматика описывает множество всех арифметических выражений, которые можно построить из целых чисел с помощью символов операций $+$, $-$ и $*$, понимаемых в обычном смысле.

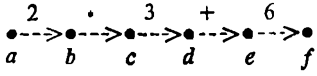
Цель синтаксического анализатора состоит в том, чтобы по «фразе» языка (в данном случае по целочисленному арифметическому выражению) восстановить ее синтаксическую структуру, соответствующую вышеприведенной грамматике.

Пример декомпозиции выражения $2*3+6$ приведен на рис. 10, см. стр. 62.

3.5.2. Представление грамматики

Как писать такой анализатор на Прологе? Представим входную цепочку в виде графа: первый узел графа ставится в начале фразы, а затем после каждого слова добавляется еще один узел. Каждое слово будет использовано как пометка на дуге, соединяющей два узла, окружающих это слово.

В нашем примере мы получим:



Тогда правила грамматики можно рассматривать как инструкции, позволяющие достроить это дерево. Такое правило, как (12), будет интерпретироваться так:

«Если в графе найдется дуга между узлами x и y, помеченная числом 2, то необходимо добавить дугу между x и y с пометкой *число*».

А правило типа (1) дает:

«Если найдутся дуга с пометкой *произведение* между узлами x и y и дуга с пометкой *хвост суммы* между узлами y и z, то необходимо добавить дугу с пометкой *выражение* между x и z».

При этих соглашениях анализ фразы сводится к нахождению дуги с пометкой *выражение* между первым и последним узлов в графе, сопоставленном данной фразе. Граф, построенный для нашего примера, приведен на рис. 11.

Теперь напишем программу на Прологе: каждому нетерминалу N в грамматике мы сопоставим предикат с этим именем. Таким образом, $N(\langle x, y\rangle)$ будет интерпретироваться как «в графе существует дуга с пометкой N между узлами x и y».

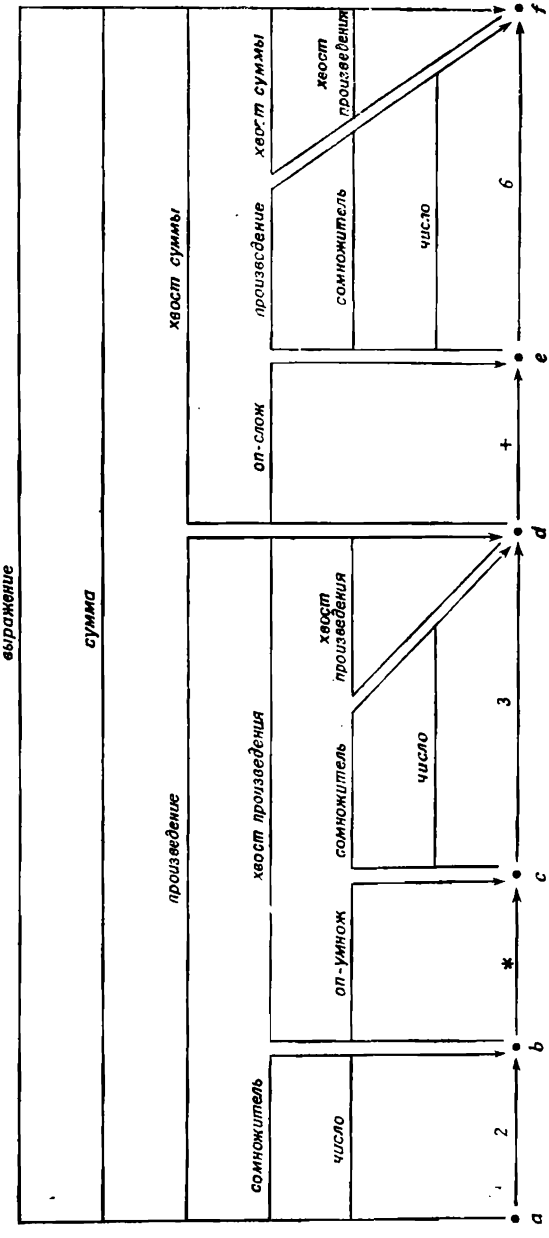


Рис. 11. Граф, сопоставленный арифметическому выражению.

Для правила (1) мы получим:

сумма(⟨ *x*, *y* ⟩) → *произведение* (⟨ *x*, *z* ⟩) *хвост-суммы*(⟨ *z*, *y* ⟩);

Вернемся ненадолго к графу — его легко представить в виде списка слов входной фразы (рис. 12), причем каждой

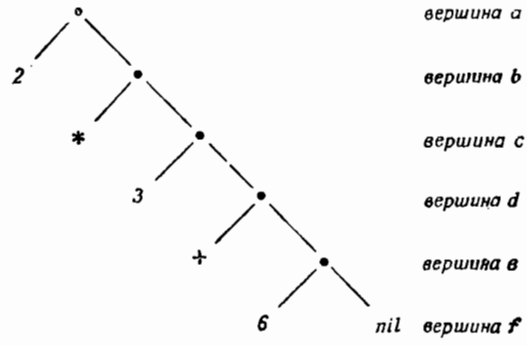


Рис. 12. Список, соответствующий начальному графу.

вершине графа будет соответствовать поддерево в дереве, сопоставленном предыдущему узлу.

Тогда терм $N(\langle x, y\rangle)$ можно интерпретировать так: «x есть входная цепочка до стирания N, а y — остаток цепочки, который нужно проанализировать после стирания N».

Кроме того, такое представление позволяет переписать терминальные правила [1]), такие как (12), в виде:

число (⟨ *x*, *y* ⟩) → *слово* (*n*, ⟨ *x*, *y* ⟩) *целое* (*n*);

где использовано обычное правило

слово (*a*, ⟨*a*. *x*, *x* ⟩) →;

и встроенное правило *целое*.

И наконец, определение того, принадлежит ли некоторая фраза к нашему языку, сводится к стиранию *выражение* (⟨*p*, *nil*⟩), где *p* — представление входной фразы в форме списка.

3.5.3. Выявление структуры

Таким образом, мы уже можем перевести грамматику во множество правил, определяющих принадлежность фразы к языку. Теперь мы можем постараться добиться бóльшего,

[1]) То есть не содержащие нетерминалов в правых частях. — *Прим. перев.*

дополнив наши отношения таким образом, чтобы выдавать дерево анализа

для фразы из предыдущего примера.

Для этого нам нужно лишь добавить аргумент в предикаты, связанные с нетерминальными символами грамматики. Этот аргумент будет показывать, каким образом фраза построена из составляющих ее подфраз.

Мы заменим

число $(\langle x, y\rangle) \rightarrow$ *слово* $(n, \langle x, y\rangle)$;

на

целое $(n, \langle x, y\rangle) \rightarrow$ *слово* $(n, \langle x, y\rangle)$;

и

примитив $(\langle x, y\rangle) \rightarrow$ *число* $(\langle x, y\rangle)$;

на

примитив $(n, \langle x, y\rangle) \rightarrow$ *число* $(n, \langle x, y\rangle)$;

и т. д.

И наконец наше первое правило примет вид:

выражение $(e, \langle x, y\rangle) \rightarrow$ *произведение* $(p, \langle x, z\rangle)$
хвост-суммы $(p, e, \langle z, y\rangle)$;

где e обозначает дерево, сопоставленное проанализированному выражению.

Программа в окончательном виде приведена на рис. 13.

Мы используем предикат *прочесть*, который читает фразу, заканчивающуюся точкой, и переводит ее в список слов.

Например:

прочесть(p);

если мы введем

$12 + (23 - 4) * 5 + 210 - 34 - 43.$

то Пролог ответит:

$p = 12."+"."("."23."-"."4.")"."*"."5."+"."210."-"."34."-"."43."nil$

```
выражение (e,⟨x,y⟩) → сумма (e,⟨x,y⟩);
сумма (e,⟨x,y⟩) → произведение (p,⟨x,z⟩) хвост-суммы (p,e,⟨z,y⟩);
произведение (p,⟨x,y⟩) → сомножитель (f,⟨x,z⟩) хвост-произведения
(f,p,⟨z,y⟩);
сомножитель (n,⟨x,y⟩) → слово (n,⟨x,y⟩) целое (n);
сомножитель (e,⟨x,y⟩) → слово ("(",⟨x,z⟩) выражение (e,⟨z,t⟩)
слово (")",⟨t,y⟩);
хвост-суммы (p,e,⟨x,y⟩) → слово (o,⟨x,z⟩) оп-слож (o,o-a)
  произведение (p',⟨z,t⟩) хвост-суммы (⟨o-a,p,p'⟩,e,⟨t,y⟩);
хвост-суммы (e,e,⟨x,x⟩)→;
хвост-произведения (f,p,⟨x,y⟩) → слово (o,⟨x,z⟩) оп-умнож (o,o-m)
  сомножитель (f',⟨z,t⟩)
  хвост-произведения (⟨o-m,f,f'⟩,p,⟨t,y⟩);
хвост-произведения (f,f,⟨x,x⟩)→;
слово (⟨a.x,x⟩)→;
оп-слож ("+", слож)→;
оп-слож ("—", выч)→;
оп-умнож ("*", умнож)→;
"считывание"
прочесть (nil) → сим-след'(".")/вв-сим'(".");
прочесть (a.b) → вв-ид (a)/прочесть (b);
прочесть (a.b) → вв-целое (a)/прочесть (b);
прочесть (a.b) → вв-ид (a)/прочесть (b);
прочесть (a.b) → вв-целое (a)/прочесть (b);
прочесть (a.b) → вв-сим' (a) прочесть (b);
"запуск"
текущее→
    вывц ("выражение")
    прочесть (p)
    анализ (p,e)
    знач (e,f)
    вывц (" равно") выв (f);
анализ (p,e) → выражение (e,⟨p,nil⟩)/;
анализ (p,e) → вывц ("...является некорректным") тупик;
```

Рис. 13. Анализатор выражений.

вычислит и напечатает его значение. Это делается так:

Предикат *выражение* строит синтаксическое дерево для прочитанной фразы (если она принадлежит языку):

прочесть(*p*) *выражение*(*e*, ⟨*p*, *nil*⟩);

Если мы введем: $12 + (23 - 4) * 5 + 210 - 34 - 43$, то получим:

p=*12."+"."("."23."—"."4.")"."*"."5."+"."210."—"."34."—"."43.*
nil

e=*выч*(*выч*(*слож*(*слож*(*12*, *умнож*(*выч*(*23*, *4*), *5*)), *210*), *34*), *43*)

Теперь можно передать это дерево предикату *знач*, который

```
> текущее;
выражение 12 + (23 — 4) 5 + 210 — 34 — 43.
имеет значение 240
```

3.6. Формальное дифференцирование

В математическом анализе формальное дифференцирование — это операция, которая сопоставляет одному алгебраическому выражению другое, называемое его производной.

Эти выражения построены из чисел, операций $+$, $-$, $*$, $\uparrow$, функциональных символов SIN, COS ... и т. д. и некоторого множества переменных.

В нашем примере используются только целые числа, всего одна переменная x и два унарных функциональных символа *sin* и *cos*.

Первая часть программы (рис. 14) состоит из грамматики для анализа этих выражений — это несколько пополненная грамматика из предыдущего примера.

Таков, например, результат анализа некоторого выражения:
после команды

прочесть(*p*)*выражение*(*e*, ⟨*p*, *nil*⟩);

вводим

$3 * x \uparrow 2 + 6 * x + 5$

и получаем

$p = 3.''*''.\ ''x''.''\uparrow''.2.''+''.6.''*''.''x''.''+''.5.\ nil$
$e = слож(слож(умнож(3,\ степ(''x'',\ 2)),\ умнож(6,\ ''x'')),\ 5)$

Следующий шаг — введение отношения *производная*: *производная* (f, x, f') означает: f' *является производной* от f по x.

Это отношение содержит по одному правилу для каждого оператора или функционального символа и имеет очень естественное описание.

Дифференцирование выражения из предыдущего примера делается так:

> *прочесть*(*p*) *выражение* (*e*, ⟨*p*, *nil*⟩) *производная* (*e*, ''*x*'', *e'*);

вводим

$3 * x \uparrow 2 + 6 * x + 5.$

и получаем

.....
$e' = слож(слож(слож(умнож(3,\ умнож(2,\ умнож(1,\ степ(''x'',$
$выч(2,\ 1))))),\ умнож(степ(''x'',\ 2),\ 0)),\ слож(умнож(6,\ 1),$
$умнож(''x'',\ 0))),\ 0)$

Чтобы сделать эти вещи более наглядными, мы ввели предикат *писать*, задача которого — получить линейную

"Грамматика выражений"

```
сумма (e,⟨x,y⟩) → произведение (p,⟨x,z⟩ хвост-суммы (p,e,⟨z,y⟩));
выражение (e,⟨x,y⟩) → сумма (e,⟨x,y⟩);
произведение (p,⟨x,y⟩) → сомножитель (f,⟨x,z⟩)
                          хвост-произведения (f,p,⟨z,y⟩));
сомножитель (f,⟨x,y⟩) → примитив (p,⟨x,z⟩)
                         хвост-сомножителя (p,f,⟨z,y⟩));
примитив (n,⟨x,y⟩) → слово (n,⟨x,y⟩) целое (n);
примитив ("x",⟨x,y⟩) → слово ("x",⟨x,y⟩);
примитив (⟨o-u,p⟩,⟨x,y⟩) → слово (o,⟨x,z⟩ оп-ун (p,o-u)
                             примитив (p,⟨z,y⟩));
примитив (e,⟨x,y⟩) → слово ("(",⟨x,z⟩) выражение (e,⟨z,t⟩)
                      слово (")",⟨t,y⟩)

хвост-суммы (p,e,⟨z,y⟩)→
   слово (o,⟨x,z⟩)
   оп-сложс (o,o-a)
   произведение (p',⟨z,t⟩)
   хвост-суммы (⟨o-a,p,p'⟩,e,⟨t,y⟩));
хвост-суммы (e,e,⟨x,x⟩)→;

хвост-произведения (f,p,⟨z,y⟩)→
   слово (o,⟨x,z⟩)
   оп-умнож (o,o-m)
   сомножитель (f',⟨x,y⟩)
   хвост-произведения (⟨o-m,f,f'⟩,p,⟨t,y⟩));
хвост-произведения (f,f,⟨x,x⟩)→;

хвост-сомножителя (p,f,⟨z,y⟩)→
   слово (o,⟨x,z⟩)
   оп-степ (o,o-e)
   примитив (p',⟨z,t⟩)
   хвост-сомножителя (⟨o-e,p,p'⟩,f,⟨t,y⟩));
хвост-сомножителя (f,f,⟨x,x⟩)→

слово (a,⟨a.x,x⟩)→;

оп-сложс ("+",add)→;
оп-выч ("—",sub)→;
оп-умнож ("*",mul)→;
оп-степ (" ",exp)→;
оп-ун ("—",sub)→;
оп-ун (sin,sin)→;
оп-ун (cos,cos)→;
```

«Правила дифференцирования»

```
производная (x,x,1)→;
производная (n,x,0) → целое (n);
производная (add(u,v),x,add(u',v'))) → производная (u,x,u')
                                       производная (v,x,v');
производная (sub(u,v),x,sub(u',v')) → производная (u,x,u')
                                      производная (v,x,v');
производная (mul(u,v),x,add(mul(u,v'),mul(v,u'))→
   производная (u,x,u')
   производная (v,x,v');
производная (exp(u,v),x,mul(n,mul(u',sub(n,1)))))→
   производная (u,x,u');
```

```
производная (sub(u),x,sub(u′)) → производная (u,x,u′);
производная (sin(u),x,mul(u′,cos(u))) → производная (u,x,u′);
производная (cos(u),x,sub(mul(u′,sin(u))) → производная (u,x,u′);
```

«Правила упрощения»

```
упростить (⟨o-b,x,y⟩,u)→
    упростить (x,x′)
    упростить (y,y′);
    упр (o-b,x′,y′,u);
упростить (⟨o-u,x⟩,u) → упростить (x, x′) упр (o-u,x′,u);
упростить (x,x)→;

упр (add,0,x,x)→;
упр (add,x,0,x)→;
упр (sub,x,0,x)→;
упр (sub,0,x,y) → упр (sub,x,y);
упр (mul,0,x,0)→;
упр (mul,x,0,0)→;
упр (mul,1,x,x)→;
упр (mul,x,1,x)→;
упр (exp,x,0,1)→;
упр (mul,sub(x),y,u)→
    упр (mul,x,u,v)
    упр (sub,v,u);
упр (mul,x,sub(y),u)→
    упр (mul,x,u,v)
    упр (sub,v,u);
упр (exp,x,1,x)→;
упр (exp,0,x,0)→;
упр (exp,1,x,1)→;
упр (o-b,x,y,u) → различные (o-b,exp) целое (x) целое (y)
                  знач (⟨o-b,x,y⟩,u);
упр (o-b,x,⟨o-b,u,v⟩,t) → упр (o-b,x,u,z) упр (o-b,z,v,t);
упр (o-b,x,y,⟨o-b,x,y⟩)→;
упр (sub,0,0)→;
упр (sub,sub(x),x)→;
упр (sin,0,0)→;
упр (cos,0,1)→;
упр (o-u,x⟨o-u,x⟩)→;
```

«Считывание»

```
прочесть (nil) → сим-след′ (".")/вв-сим′ (".");
прочесть (a.b) → вв-ид (a)/прочесть (b);
прочесть (a.b) → вв-целое (a)/прочесть (b);
прочесть (a.b) → вв-сим′ (a)/прочесть (b);
```

«Печать»

```
писать (⟨o-b,x,y⟩)→
    оп-бин (o,o-b)
    вывц ("(")
    писать (x)
    вывц (o)
    писать (y)
    вывц (")");
писать (sub(x)) → /вывц ("—") писать (x);
писать (⟨o-u.x⟩) → /оп-ун (o,o-u) выв (o) писать (x);
```

```
писать (x) -> цепочка (x)/вывц (x);
писать (x) -> выв (x);
оп-бин (o,o-b) -> оп-слож (o,o-b);
оп-бин (o,o-b) -> оп-умнож (o,o-b);
оп-бин (o,o-b) -> оп-степ (o,o-b);

    «Запуск»

текущее->
    вывц ("выражение")
    прочесть (p)
    анализ (p,e)
    производная (e,"x",e')
    упростить (e',f)
    вывц ("имеет производную")
    писать (f)
    строка
    /;

анализ (p,e) -> выражение (e,⟨p,nil⟩);
анализ (p,e) -> вывц ("... содержит ошибку") тупик;
```

Рис. 14. Дифференцирование выражений (без упрощения).

запись дерева, которое задается его аргументом; допускается появление нескольких лишних скобок.

Вот что мы получим для значения e' из предыдущего примера:

$$((((3 * (2 * (1 * (x \uparrow (2 - 1))))) + ((x \uparrow 2) * 0)) +$$

$$((6 * 1) + (x * 0))) + 0)$$

Как мы видим, результат еще слишком далек от совершенства!

Поэтому мы добавили еще программу (далеко не полную!), упрощающую выражения, которая позволяет получить более сжатую запись.

Все это представлено предикатом *текущее*, и мы приводим два примера его использования.

```
< текущее;
выражение 3 * x ↑ 2 + 6 * x + 5.
имеет производную (6 * x) + 6)
< текущее;
выражение (— 3 * x ↑ 2 + 2).
имеет производную ((6 * x) * sin (— (3 * (x ↑ 2)) + 2))
```

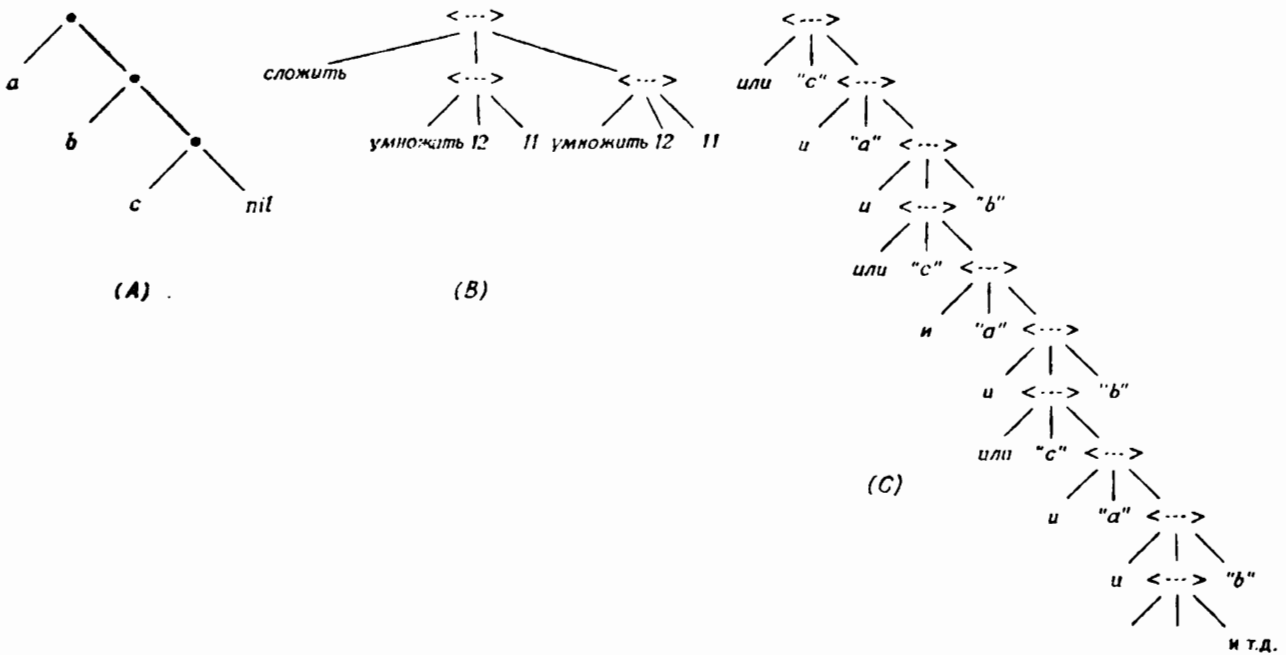


Рис. 15. Три дерева.

Часть вторая: теоретические основы

4. КОНЕЧНЫЕ И БЕСКОНЕЧНЫЕ ДЕРЕВЬЯ

4.1. Основные понятия

4.1.1. Интуитивное понятие дерева

Все данные, с которыми имеет дело Пролог, являются деревьями (возможно, бесконечными), поэтому мы вначале дадим интуитивное описание. Эти деревья (рис. 15 на стр. 72) построены из узлов помеченных:

(1) константой; у таких узлов нет сыновей;
(2) символом *точка*; в этом случае узел имеет двух сыновей;
(3) несколькими символами тире в угловых скобках: «⟨ ⟩», «⟨—⟩», «⟨— —⟩», и т. д.; в этом случае число тире соответствует числу сыновей.

Рис. 15 дает три примера деревьев. Деревья (A) и (B)— конечные, (C)— бесконечное дерево.

4.1.2. Поддеревья

В общем случае дерево может содержать поддеревья; мы не будем строго определять это понятие, но проиллюстрируем его с помощью двух предыдущих примеров (рис. 15(B) и (C)), множества поддеревьев которых таковы:

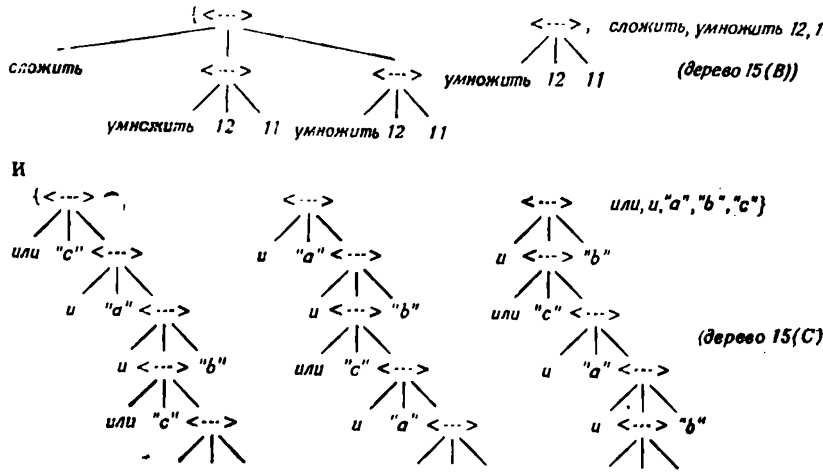


Рис. 16.

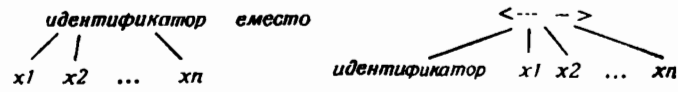

идентификатор
x1 x2 ... xn
вместо
<--- -->
идентификатор x1 x2 ... xn

Рис. 17.

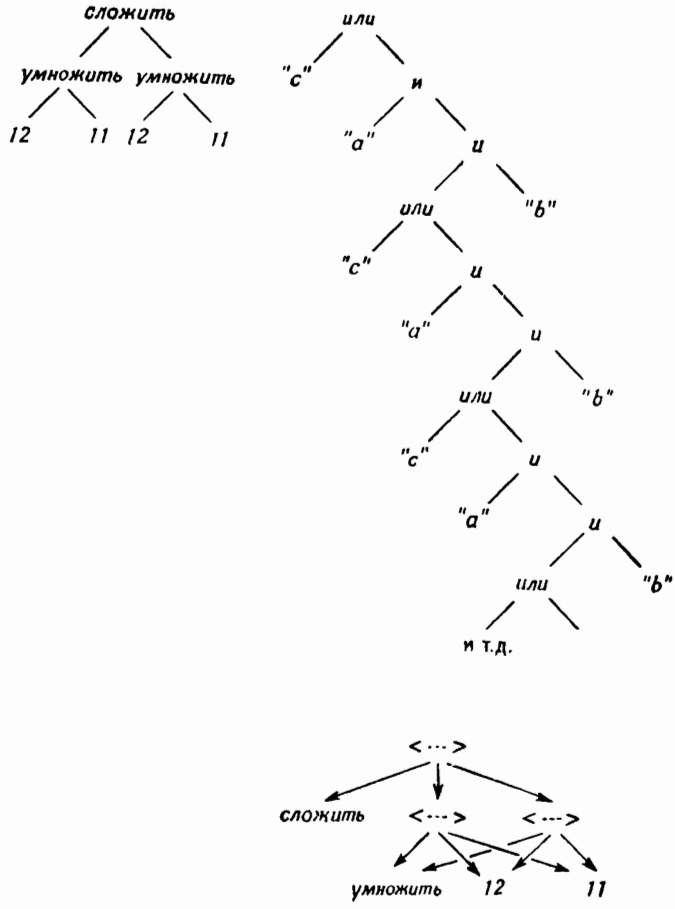

сложить
умножить умножить
12 11 12 11
или
"c"
и
"a"
и
или
"b"
"c"
и
"a"
и
или
"b"
"c"
и
"a"
и
или
"b"
и т.д.
<--->
сложить
<---> <--->
умножить 12 11

Рис. 18.

4.1.3. Упрощенное представление

На рис. 17 (см. стр. 74) приведен упрощенный способ рисования деревьев. Конечно, для такого упрощения требуется, чтобы n не было равно нулю. Два последних дерева можно теперь представить в традиционной форме (рис. 18, см. также стр. 74).

Надо иметь в виду, что различные диаграммы могут изображать одно и то же дерево.

И наконец интересно познакомиться с нерациональным деревом, т. е. с деревом, которое имеет бесконечное число

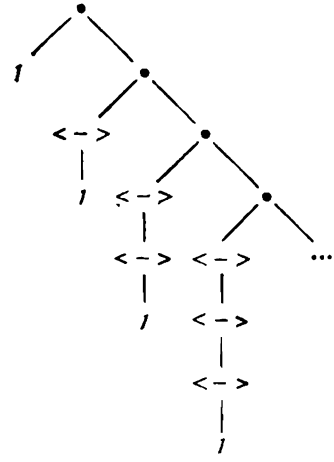


Рис. 19.

(различных) поддеревьев. Наиболее простой из известных нам примеров такого дерева приведен на рис. 19.

4.1.4. Рациональные деревья

Понятие бесконечного дерева несколько непривычно и заслуживает некоторой проработки. Интуитивно, дерево является бесконечным, если оно обладает по крайней мере одной бесконечной ветвью.

Мы ограничимся особым классом бесконечных деревьев — так называемыми рациональными бесконечными деревьями, которые обладают важным свойством: они могут быть заданы с помощью конечного описания.

> Определение
> Дерево называется рациональным, если множество его поддеревьев конечно.

Все конечные деревья являются, таким образом, рациональными; однако таковыми являются и некоторые бесконеч-

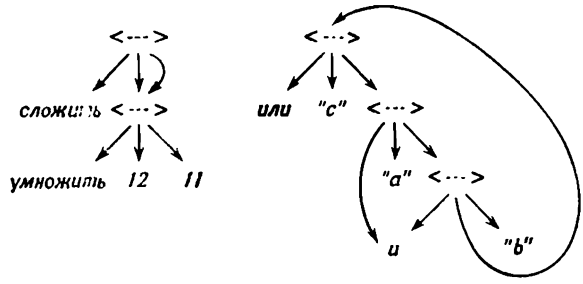


Рис. 20.

ные деревья — это, например, дерево (C) на рис. 15, для которого мы указали конечное множество его поддеревьев (см. рис. 16).

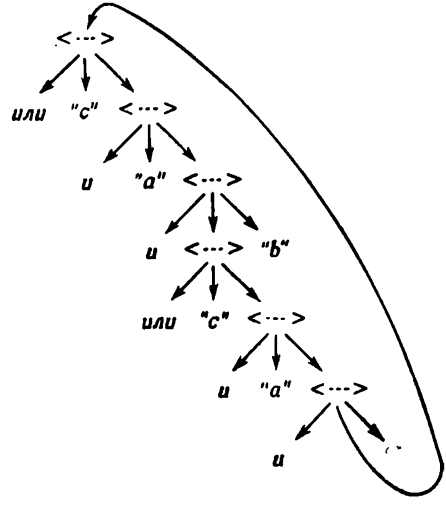


Рис. 21.

Тот факт, что рациональное дерево содержит конечное множество поддеревьев, немедленно дает нам способ представления его с помощью конечной диаграммы: достаточно

слить все узлы, из которых выходят одинаковые поддеревья. На рис. 20 представлены такие диаграммы для примеров на рис. 15(B) и 15(C).

Если не делать всех слияний, то можно получить другую диаграмму (рис. 21).

4.2. Свойства композиции и однозначной декомпозиции

Теперь настало время для более строгого определения деревьев. Легко уподобить конечное дерево формуле определенного вида. Трудности возникают с бесконечным деревом — последовательность символов формулы, представляющей дерево, рискует, в свою очередь, стать бесконечной, а математика знает только последовательности, бесконечные в одном направлении!

Чтобы избежать этих трудностей, приходится определять деревья достаточно сложным способом, вводя последовательности целых чисел для обозначения узлов. Такое определение можно найти в гл. 5 диссертации Г. Юэ [Юэ 76]. Это определение имеет по крайней мере то преимущество, что является строго формальным; однако, как нам кажется, оно очень сильно уступает в прозрачности интуитивному понятию дерева. Мы же будем называть *множеством деревьев* любое множество R, обладающее **тремя характеристическими свойствами**, сформулированными ниже (два первых сразу же, а третье — несколько позже). *Дерево* — это элемент множества R. Мы называем эти свойства *характеристическими*, поскольку они кажутся достаточно ограничивающими, чтобы все множества R, которые им удовлетворяют, были бы изоморфны.

Характеристическое Свойство 1 **(композиция)**
Множество R удовлетворяет свойству композиции, если выполняются следующие три условия:
(1) всякая константа есть элемент R;
(2) если r_1 и r_2 — элементы R, то сущность, обозначаемая $(r_1.r_2)$, также является элементом R;
(3) если $r_1, r_2, \ldots, r_3$ есть последовательность (возможно, пустая) элементов из R, то сущность, обозначаемая $\langle r_1, r_2, \ldots, r_n \rangle$, также принадлежит R.

Характеристическое Свойство 2 (**однозначная декомпозиция**)

Множество R удовлетворяет свойству декомпозиции, если для всякого элемента r из R выполняется одно и только одно из следующих трех утверждений:

(1) существует такая единственная константа k, что $r = k$; в этом случае говорят, что последовательность сыновей r пуста;

(2) существует такая единственная пара r_1, r_2 элементов R, называемая последовательностью сыновей r, что $r = (r_1, r_2)$;

(3) существует такая единственная последовательность (может быть, пустая) $r_1, \ldots, r_n$ элементов R, называемая последовательностью сыновей r, что $r = \langle r_1, r_2, \ldots r_n \rangle$.

Первое свойство дает нам способ конструирования дерева из других деревьев. Оно нас приводит к введению понятия терма, который есть формула, задающая такую конструкцию. Второе свойство имеет два важных следствия:

(1) всякое равенство вида $(r_1 . r_2) = (s_1 . s_2)$ влечет за собой равенства $r_1 = s_1$ и $r_2 = s_2$; аналогично, всякое равенство вида $\langle r_1, \ldots, r_n \rangle = \langle s_1, \ldots, s_n \rangle$ влечет за собой равенства $r_1 = s_1, \ldots$ и $r_n = s_n$;

(2) всякое дерево, имеющее вид константы, отличается от дерева вида $(r_1 . r_2)$ или $\langle r_1, \ldots, r_m \rangle$; всякое дерево вида $(r_1 . r_2)$ отлично от дерева вида $\langle r_1, \ldots, r_m \rangle$; если n и m различны, то любое дерево вида $\langle r_1 \ldots, r_m \rangle$ отлично от дерева вида $\langle s_1, \ldots s_n \rangle$.

Надо заметить также, что свойство 2 вводит формальное понятие *сына*, поэтому теперь мы можем дать строгое определение бесконечного дерева и множества поддеревьев данного дерева.

Определение

Дерево является бесконечным тогда и только тогда, когда существует бесконечная последовательность деревьев $r_0, r_1, r_2, \ldots$, в которой r_{i+1} является сыном r_i.

Определение

Множество поддеревьев дерева r есть наименьшее множество деревьев, содержащее r и содержащее всех сыновей всякого дерева, которое оно содержит.

Рациональное дерево — это, как и раньше, дерево, имеющее конечное множество поддеревьев.

4.3. Термы, системы уравнений и неравенств

Для представления деревьев мы будем использовать формулы, называемые термами. Вначале мы введем понятие *строгого терма*:

⟨ *строгий терм* ⟩
::= ⟨ *переменная* ⟩
::= ⟨ *константа* ⟩
::= (⟨ *строгий терм* ⟩, ⟨ *строгий терм* ⟩)
::= ⟨ ⟩
::= ⟨⟨ *строгий терм* ⟩⟩
::= ⟨⟨ *строгий терм* ⟩, ⟨ *строгий терм* ⟩⟩
::= ⟨⟨ *строгий терм* ⟩, ⟨ *строгий терм* ⟩, ⟨ *строгий терм* ⟩⟩
. . . . . .

Строгие термы — это настоящие термы. Однако из соображений удобства мы расширяем синтаксис строгих термов (не меняя их смысла), разрешая:

(1) добавлять и снимать скобки, придерживаясь соглашения, что $t_1.t_2.\_\_\_.\ t$ изображает $(t_1.(t_2.(\_\_\_.t_n)\_)))$;

(2) писать $id(t_1, t_2, \ldots, t_n)$ вместо $\langle id, t_1, t_2, \ldots, t_n\rangle$ при условии, что id — идентификатор и n отлично от нуля.

Все это приводит к более общему понятию — понятию *терма*:

⟨ *терм* ⟩
::= ⟨ *простой терм* ⟩
::= ⟨ *простой терм* ⟩.⟨ *терм* ⟩
⟨ *простой терм* ⟩
::= (⟨ *терм* ⟩)
::= ⟨ *переменная* ⟩
::= ⟨ *константа* ⟩
::= ⟨ ⟩
::= ⟨⟨ *терм* ⟩⟩
::= ⟨⟨ *терм* ⟩, ⟨ *терм* ⟩⟩
::= ⟨⟨ *терм* ⟩, ⟨ *терм* ⟩, ⟨ *терм* ⟩⟩
. . . . . .
::= ⟨ *идентификатор* ⟩(⟨ *терм* ⟩)
::= ⟨ *идентификатор* ⟩(⟨ *терм* ⟩, ⟨ *терм* ⟩)
::= ⟨ *идентификатор* ⟩(⟨ *терм* ⟩, ⟨ *терм* ⟩, ⟨ *терм* ⟩)
. . . . . .

Вот пример терма:

Соответствующий строгий терм выглядит так:

⟨*любят*, (*Пьер*.(*Поль*.(*Жан*. *nil*))), ⟨*является-отцом x*⟩⟩.

Чтобы преобразовать терм в дерево, необходимо присвоить деревья его переменным, откуда возникает понятие *древесного означивания*[1]).

Определение

Назовем **древесным означиванием** всякое множество, имеющее вид:

$$X = \{x_1 := r_1, \quad x_2 := r_2, \quad \ldots\},$$

где x_i — различные переменные и r_i — деревья.

Дерево, сопоставляемое терму t, обозначается через $t(X)$ и определяется следующим образом.

Определение

Если t — строгий терм, переменные которого образуют подмножество переменных древесного означивания $X = \{x_1 := r_1, x_2 := r_2, \ldots\}$, тогда $t(X)$ обозначает дерево, полученное заменой переменных x_i на соответствующие деревья r_i. Более точно:

(1) $t(X) = r_i$, если $t = x_i$;
(2) $t(X) = k$, если t — константа k;
(3) $t(X) = (t_1(X).t_2(X))$, если $t = (t_1.t_2)$;
(4) $t(X) = \langle t_1(X), \ldots, t_n(X)\rangle$, если $t = \langle t_1, \ldots, t_n\rangle$.

Если t не содержит переменных, то $t(X)$ обозначается также через $'t'$.

Если t_1 и t_2 — термы, тогда формулы $t_1 = t_2$ и $t_1 \neq t_2$ являются соответственно *уравнением* и *неравенством*. Во всех тех случаях, когда не оговорено противное, мы будем использовать слово *система* для обозначения конечной системы. Мы можем теперь дать синтаксическое определение системы:

⟨*система*⟩
  ::= { }
  ::= {⟨*уравнения и неравенства*⟩}
⟨*уравнения и неравенства*⟩
  ::= ⟨*уравнение*⟩
  ::= ⟨*неравенство*⟩

[1]) Означивание — это перевод термина evaluation, который в логике переводится как «присваивание значений переменных». Однако термин «присваивание» широко используется в другом смысле в литературе, посвященной языкам программирования. — *Прим. ред.*

::=⟨*уравнение*⟩, ⟨*уравнения и неравенства*⟩
::=⟨*неравенство*⟩, ⟨*уравнения и неравенства*⟩
⟨*уравнение*⟩
::=⟨*терм*⟩=⟨*терм*⟩
⟨*неравенство*⟩
::=⟨*терм*⟩≠⟨*терм*⟩

Отметим уже сейчас один очень специальный вид системы, который причинил нам много хлопот: *кольцо переменных.*

Определение
Кольцо переменных — это непустая система вида

$$\{x_1 = x_2,\ x_2 = x_3,\ \ldots,\ x_n = x_1\},$$

причем все переменные x_i различны и n может быть равно 1.

Кто сказал «система» (возможно, бесконечная) — должен сказать «решение системы». Понятие решения системы должно быть определено строго, и при этом таким образом, чтобы, как ни парадоксально, множество переменных, встречающихся в решении, не зависело бы от множества переменных, встречающихся в системе. Это существенно, если рассматривать решения объединения нескольких систем.

Определение
Древесное означивание X называется **древесным решением** системы (может быть, бесконечной)

$$\{p_1 = q_1,\ p_2 = q_2,\ \ldots\} \cup \{s_1 \neq t_1,\ s_2 \neq t_2,\ \ldots\},$$

если X есть подмножество древесного означивания Y, такого, что $p_i(Y)$ равны соответственно $q_i(Y)$, и $s_i(Y)$ не равны соответственно $t_i(Y)$.

Заметим, что всякая система (возможно, бесконечная), допускающая по крайней мере одно древесное решение, допускает также и пустое древесное решение { }.

4.4. Свойство единственности решения

Теперь у нас все готово, чтобы сформулировать третье характеристическое свойство множества R деревьев.

Характеристическое свойство 3 (единственность решения)

Множество R удовлетворяет свойству единственности решения, если для любой системы уравнений S, возможно, бесконечной, имеющей вид

$$S = \{x_1 = t_1,\ x_2 = t_2,\ \ldots\},$$

где

x_i — различные переменные,

t_i — термы, не сводящиеся к переменным и не содержащие других переменных, кроме x_i,

существует такая единственная последовательность деревьев $r_1, r_2, \ldots$ в R, что древесное означивание

$$\{x_1 := r_1,\ x_2 := r_2,\ \ldots\}$$

есть решение S.

Пусть дана, например, система

$\{x_1 = gg\,(x_1,\ x_2),\ x_2 = dd\,(x_1,\ x_2)\}$.

Сразу переходим к диаграмме, изображенной на рис. 22, и получаем единственное решение (рис. 23).

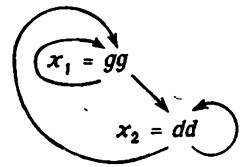


Рис. 22.

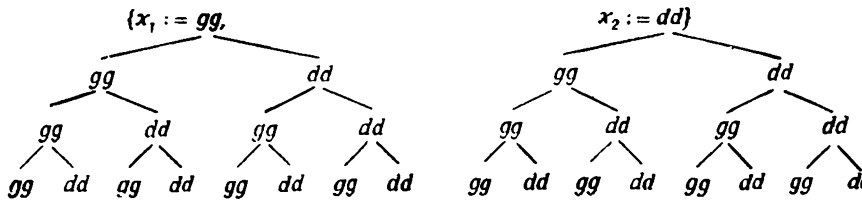


Рис. 23.

Пусть дано множество $M = \{r_1, r_2, \ldots\}$ поддеревьев некоторого множества деревьев. Если сопоставить переменную x_i каждому дереву r_i, то, помня, что сыновья каждого r_i также принадлежат M, можно написать уравненне для каждого r_i

и получить систему уравнений, для которой $\{x_1 := r_1,\ x_2 := r_2,\ \ldots\}$ будет единственным решением. Мы имеет, таким образом:

Свойство ассоциированной системы

Каково бы ни было множество $\{r_1, r_2, \ldots\}$ поддеревьев некоторого множества деревьев, существует система S, возможно бесконечная,

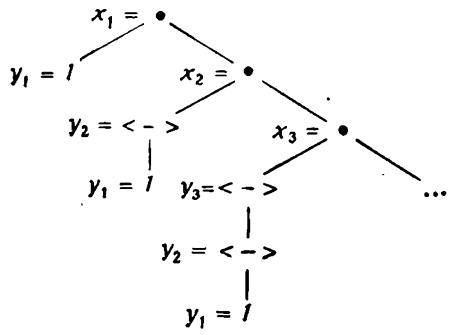
$$S = \{x_1 = t_1,\ x_2 = t_2,\ \ldots\},$$

для которой

$$\{x_1 := r_1,\ x_2 := r_2,\ \ldots\}$$

является древесным решением. Все переменные x_i различны, а термы t_i не являются переменными и не содержат других переменных, кроме x_i.

Вернемся к примеру нерационального дерева, помещенному в конце разд. 4.1 (см. рис. 19), и сопоставим переменную каждому из его поддеревьев:



Это дерево является решением для x_1 системы

$$\{x_1 = (y_1 . x_2),\ y_1 = 1,\ x_2 = (y_2 . x_2),\ y_2 = \langle y_1 \rangle,\ x_3 = (y_3 . x_4),$$
$$y_3 = \langle y_2 \rangle,\ \ldots\}$$

5. РЕШЕНИЕ СИСТЕМЫ УРАВНЕНИЙ И НЕРАВЕНСТВ

5.1. Приведенный вид системы

В этом разделе мы займемся проблемой существования древесного решения для системы. Система, для которой такое решение существует, называется *разрешимой*; в противном случае система называется *неразрешимой*. Вначале мы

ассмотрим системы некоторого специального вида — систе-ы в *приведенной* форме.

Определение
Система S, состоящая из множества E уравнений и множества I неравенств, находится в приведенной форме, если она имеет вид (1) или (2):
(1) I — пусто, а E, возможно пустое, не содержит кольца переменных и имеет вид

$$\{x_1 = t_1, \ \ldots, \ x_n = t_n\},$$

где все x_i — различные переменные, а t_i — произвольные термы.
(2) Множество I не пусто, и каждое неравенство имеет вид

$$\langle y_1, \ \ldots, \ y_m\rangle \neq \langle t_1, \ \ldots, \ t_m\rangle,$$

где m не равно нулю, переменные y_i отличны от x_i, y_i — произвольные термы, и для каждого неравенства ассоциированная система

$$E \cup \{y_1 = t_1, \ \ldots, \ y_m = t_m\}$$

находится в приведенной форме.

Сформулируем первое фундаментальное свойство систем приведенной форме.

Свойство разрешимости
Всякая система в приведенной форме разрешима.

Доказательство приведено в приложении IV. Дадим несколько примеров систем в приведенной форме:

$\{x = 1, \ y = 2\}$,

$\{x = \langle x, \ u\rangle, \ y = \langle x, \ v\rangle, \ \langle u, \ v\rangle \neq \langle x, \ y\rangle, \ \langle u, \ v\rangle \neq \langle u, \ 1\rangle\}$;

несколько примеров систем, не находящихся в приведенной ›орме:

$$\{u = 1, \ v = x, \ \langle x, \ y, \ z\rangle \neq \langle y, \ z, \ x\rangle\},$$
$$\{x = 1, \ \langle u, \ v\rangle \neq \langle x, \ u\rangle, \ \langle x\rangle \neq \langle 1\rangle\}.$$

.2. Эквивалентные системы

Системы в приведенной форме обладают вторым фунда-ентальным свойством, связанным с понятием эквивалент-ости двух систем, к которому мы теперь и переходим.

Определение
Две системы **эквивалентны**, если они имеют одно и то же множество древесных решений.

Обратимся к определению древесного решения и заметим, что для доказательства эквивалентности двух систем достаточно рассматривать только решения, в которых встречаются переменные из объединения этих систем. Приведем 4 важных и интересных частных случая эквивалентности.

Эквивалентность 1: Каковы бы ни были термы t_i и s_i, системы

$$\{(s_1 . s_2) = (t_1 . t_2)\} \text{ и } \{s_1 = t_1,\ s_2 = t_2\},$$

а также системы

$$\{\langle s_1, \ldots, s_n\rangle = \langle t_1, \ldots, t_n\rangle\} \text{ и } \{s_1 = t_1,\ s_n = t_n\}$$

являются эквивалентными.

Эквивалентность 2: Пусть даны две системы S и T в приведенной форме, имеющие вид

$$S = \{x_1 = t_1, \ldots, x_n = t_n\} \text{ и } T = \{x_1 = t_1', \ldots, x_n = t_n'\}.$$

Тогда, если всякое древесное решение S есть также древесное решение T, то S и T эквивалентны.

Эквивалентность 3: Пусть дана система S и термы s, t. Если система $S \cup \{s = t\}$ неразрешима, то системы

$$S \cup \{s \neq t\} \text{ и } S$$

являются эквивалентными.

Эквивалентность 4: Пусть S — система, s, t — термы, $s_1, \ldots, s_n$ и $t_1, \ldots, t_n$ — последовательности термов, возможно пустые. Если эквивалентны системы

$$S \cup \{s = t\} \text{ и } S \cup \{s_1 = t_1, \ldots, s_n = t_n\},$$

то эквивалентны и системы

$$S \cup \{s \neq t\} \text{ и } S \cup \{\langle s_1, \ldots, s_n\rangle \neq \langle t_1, \ldots, t_n\rangle\}.$$

Эквивалентность 1 является непосредственным следствием 2-го характеристического свойства (однозначность декомпо-

зиции). Доказательство эквивалентности 2 приводится в приложении. Эквивалентность 3 тривиальна, так же как и эквивалентность 4, если заметить, что никакое решение системы $\{s_1, \ldots, t_1, \ldots, s_n = t_n\}$, в которое входят все переменные, не является решением системы $\{\langle s_1, \ldots, s_n\rangle \neq \langle t_1, \ldots, t_n\rangle\}$, и наоборот.

После этих отступлений мы можем теперь сформулировать второе фундаментальное свойство систем в приведенной форме.

Свойство нормальной формы
Всякая разрешимая система эквивалентна некоторой системе в приведенной форме.

Для доказательства достаточно продемонстрировать формальный алгоритм, называемый *приведением*, который преобразует всякую конечную систему в эквивалентную, которая либо находится в приведенной форме, либо является тривиально неразрешимой. Одновременно мы решаем и исходную задачу — определение того, является ли данная система S разрешимой. Действительно, достаточно применить к S алгоритм приведения, и если результат находится в приведенной форме, то S разрешима.

5.3. Приведение уравнений

Первый алгоритм, который мы рассматриваем, — это основной алгоритм приведения подсистемы, состоящей из множества уравнений. Поскольку нашей целью было доказательство существования, мы выбрали простой алгоритм из соображений наглядности. Ни в коем случае этот алгоритм нельзя рассматривать как основной эффективный алгоритм, который следовало бы запрограммировать в таком виде, как мы его приводим!

Основной алгоритм приведения
Пусть дана система S. Алгоритм состоит в применении к S, по мере возможности, трансформаций из следующего списка:
T_1 (поглощение): удаляем любое уравнение вида $x = x$ или $k_1 = k_2$, где x — переменная, а k_1 и k_2 — константы с одинаковым значением.
T_2 (исключение переменной): если в системе есть уравнение $x = y$, где x, y — различные переменные, а x имеет и другие вхождения, то заменяем x в этих вхождениях на y.

T_3 (перестановка переменной): заменяем уравнение $t = x$ на уравнение $x = t$, где x — переменная, а t не является переменной.
T_4 (противопоставление): заменяем подсистему $\{x = t_1, x = t_2\}$ на подсистему $\{x = t_1, t_1 = t_2\}$, где x — переменная, t_1, t_2 — термы, не являющиеся переменными, и высота терма t_1 меньше высоты терма t_2. Под высотой терма мы понимаем суммарное число вхождений в него переменных, констант, знаков *точка* и $\langle$.
T_5 (разложение): заменяем подсистему $\{(s_1 . s_2) = (t_1 . t_2)\}$ на $\{s_1 = t_1, s_2 = t_2\}$, а подсистему $\{\langle s_1, \ldots, s_n\rangle = \langle t_1, \ldots, t_n\rangle\}$ на $\{s_1 = t_1, \ldots, s_n = t_n\}$.

Этот алгоритм всегда заканчивается благодаря свойству приведенному ниже (доказательство в приложении).

Свойство остановки

Не существует бесконечной последовательности систем $S_1, S_2, S_3, \ldots$, где S_{i+1} получена из S_i с помощью одной из трансформаций T_1, T_2, T_3, T_4 и T_5

Этот алгоритм действительно является алгоритмом приведения уравнений по двум причинам:

(1) Каждая трансформация сохраняет эквивалентность систем. Для трансформаций T_1, T_2, T_3 и T_4 это следует из свойств равенства, а для T_5 — из эквивалентности 1.
(2) Когда ни одна из трансформаций не применима, окончательная система, если она не находится в приведенной форме, обязательно содержит уравнение, принадлежащее одному из следующих 5 типов:

(1) $k_1 = k_2$, где k_1 и k_2 — константы с различными значениями
(2) $k = (t_1 . t_2)$ или $(t_1 . t_2) = k$, где k — константа;
(3) $k = \langle t_1, \ldots, t_n\rangle$ или $\langle t_1, \ldots, t_n\rangle = k$, где k — константа;
(4) $(s_1 . s_2) = \langle t_1, \ldots, t_n\rangle$ или $\langle t_1, \ldots, t_n\rangle = (s_1 . s_2)$;
(5) $\langle s_1, \ldots, s_m\rangle = \langle t_1, \ldots, t_n\rangle$, где m не равно n.

Такая система тривиальным образом неразрешима по 2-му характеристическому свойству (однозначность декомпозиции).

Ниже следуют три примера работы алгоритма приведения.

Пример 1

$\{\langle\langle x, \textit{жан}\rangle, y\rangle = \langle y, \langle \textit{поль}, \textit{жан}\rangle\rangle\}$ — разрешимая система, поскольку:
$\{\langle\langle x, \textit{жан}\rangle, y\rangle = \langle y, \langle \textit{поль}, \textit{жан}\rangle\rangle\}$ — разложение
$\{\langle x, \textit{жан}\rangle = y, y = \langle \textit{поль}, \textit{жан}\rangle\}$ — перестановка

$\{y = \langle x,\ жан\rangle,\ y = \langle поль,\ жан\rangle\}$ — противопоставление
$\{y = \langle x,\ жан\rangle,\ \langle x,\ жан\rangle = \langle поль,\ жан\rangle\}$ — разложение
$\{y = \langle x,\ жан\rangle,\ x = поль,\ жан = жан\}$ — поглощение
$\{y = \langle x,\ жан\rangle,\ x = поль\}$ — находится в приведенной форме.

Пример 2

$\{x = \langle\langle x\rangle\rangle,\ \langle\langle жан\rangle\rangle = x\}$ — неразрешимая система, поскольку
$\{x = \langle\langle x\rangle\rangle,\ \langle\langle жан\rangle\rangle = x\}$ — перестановка
$\{x = \langle\langle x\rangle\rangle,\ x = \langle\langle жан\rangle\rangle\}$ — противопоставление
$\{x = \langle\langle x\rangle\rangle,\ \langle\langle x\rangle\rangle = \langle\langle жан\rangle\rangle\}$ — разложение
$\{x = \langle\langle x\rangle\rangle,\ \langle x\rangle = \langle жан\rangle\}$ — разложение
$\{x = \langle\langle x\rangle\rangle,\ x = жан\}$ — противопоставление
$\{жан = \langle\langle x\rangle\rangle,\ x = жан\}$ — тривиально неразрешима.

Пример 3

$\{x = y,\ x = \langle\langle x\rangle\rangle,\ y = \langle\langle\langle y\rangle\rangle\rangle\}$ — разрешима, поскольку
$\{x = y,\ y = \langle\langle y\rangle\rangle,\ y = \langle\langle\langle y\rangle\rangle\rangle\}$ — исключение переменной
$\{x = y,\ y = \langle\langle y\rangle\rangle,\ y = \langle\langle\langle y\rangle\rangle\rangle\}$ — противопоставление
$\{x = y,\ y = \langle\langle y\rangle\rangle,\ \langle\langle y\rangle\rangle = \langle\langle\langle y\rangle\rangle\rangle\}$ — разложение
$\{x = y,\ y = \langle\langle y\rangle\rangle,\ \langle y\rangle = \langle\langle y\rangle\rangle\}$ — разложение
$\{x = y,\ y = \langle\langle y\rangle\rangle,\ y = \langle y\rangle\}$ — противопоставление
$\{x = y,\ \langle y\rangle = \langle\langle y\rangle\rangle,\ y = \langle y\rangle\}$ — разложение
$\{x = y,\ y = \langle y\rangle\}$ — находится в приведенной форме.

В последнем примере следует обратить внимание на то, что несоблюдение правила, касающегося высоты терма при второй трансформации типа «противопоставление», приводит к зацикливанию:

$\{x = y,\ y = \langle\langle y\rangle\rangle,\ y = \langle y\rangle\}$ — противопоставление без проверки высоты
$\{x = y,\ y = \langle\langle y\rangle\rangle,\ \langle\langle y\rangle\rangle = \langle y\rangle\}$ — разложение
$\{x = y,\ y = \langle\langle y\rangle\rangle,\ \langle y\rangle = y\}$ — перестановка
$\{x = y,\ y = \langle\langle y\rangle\rangle,\ y = \langle y\rangle\}$ — то же, что и 3 строки выше

Если мы имеем дело с системой уравнений S, содержащей подсистему E уже в приведенной форме, было бы интересно в случае, когда S разрешима, получить такую приведенную форму S, которая содержала бы E. Это возможно благодаря следующему утверждению.

Свойство сохранения

Пусть дана разрешимая система S_1 без неравенств, имеющая вид $S_1 = E_1 \cup F_1$, где E_1 находится в приведенной форме

$$E_1 = \{x_1 = t_1, \ \ldots, \ x_n = t_n\},$$

причем каждая переменная x_i, у которой соответствующий терм t_i есть переменная, имеет единственное вхождение в S_1. Если применить основной алгоритм приведения к S_1, то мы получим приведенную систему S_2 вида $S_2 = E_2 \cup F_2$, где E_2 и F_2 не пересекаются и E_2 имеет вид

$$E_2 = \{x_1 = t_1', \ \ldots, \ x_n = t_n'\}.$$

Более того, система $E_1 \cup F_2$ находится в приведенной форме и эквивалентна исходной системе $E_1 \cup F_1$.

Доказательство (см. приложение) этого свойства использует эквивалентность 2. Рассмотрим, например, систему, которая уже частично приведена:

$$\{x = \langle x, u\rangle, \ y = \langle y, v\rangle, \ v = u\} \cup \{\langle x, z\rangle = \langle y, 5\rangle\}.$$

Заметим, что переменная v имеет всего одно вхождение. Применяя основной алгоритм приведения, мы последовательно получим

$$\{x = \langle x, u\rangle, \ y = \langle y, v\rangle, \ v = u, \ \langle x, z\rangle = \langle y, 5\rangle\},$$
$$\{x = \langle x, u\rangle, \ y = \langle y, v\rangle, \ v = u, \ x = y, \ z = 5\},$$
$$\{x = \langle x, u\rangle, \ y = \langle y, u\rangle, \ v = u, \ x = y, \ z = 5\},$$
$$\{y = \langle y, u\rangle, \ v = u, \ x = x, \ z = 5\}.$$

Таким образом, исходная система одновременно имеет две приведенные формы:

$$\{x = y, \ y = \langle y, u\rangle, \ v = u\} \cup \{z = 5\},$$

и ту, которая сохраняет приведенную подсистему:

$$\{x = \langle x, u\rangle, \ y = \langle y, v\rangle, \ v = u\} \cup \{z = 5\}.$$

5.4. Приведение уравнений и неравенств

Для приведения системы, содержащей уравнения и неравенства, необходимо иметь трансформации, позволяющие их модифицировать и получать при этом эквивалентную систему. Эквивалентности 3 и 4 дают нам две такие трансформации. Используя свойство сохранения основного алгоритма, мы получаем следующий общий алгоритм.

Общий алгоритм приведения

Пусть требуется привести систему S. Применим к S вначале основной алгоритм приведения и получим систему, состоящую из множества E уравнений и множества I неравенств. Если E не находится в приведенной форме, то S неразрешима. В противном случае полагаем

$$E = \{x_1 = t_1, \ \ldots, \ x_n = t_n\}.$$

Теперь для каждого неравенства $s \neq t$ из I образуем систему $E \cup \{s = t\}$ и применяем к ней алгоритм приведения. Возможны два случая:

(1) система $E \cup \{s = t\}$ неразрешима; тогда удаляем неравенство $s \neq t$;

(2) система $E \cup \{s = t\}$ разрешима и имеет приведенную форму

$$\{x_1 = t'_1, \ \ldots, \ x_n = t'_n, \ y_1 = s_1, \ \ldots, \ y_m = s_m\},$$

где m может быть равно нулю; тогда сохраняем неравенство

$$\langle y_1, \ \ldots, \ y_m \rangle \neq \langle s_1, \ \ldots, \ s_m \rangle.$$

Обозначим через J множество сохраненных неравенств. Если J содержит неравенство $\langle\ \rangle \neq \langle\ \rangle$, тогда исходная система S неразрешима; иначе ее приведенная форма есть $E \cup J$ и, разумеется, S разрешима.

Вот несколько примеров для иллюстрации всего этого.

Пример 1

Требуется привести систему

$$\{\langle u, \ v \rangle = \langle \langle x \rangle, \ w \rangle, \ (v \,.\, w) = (w \,.\, v), \ u \neq 1, \ \langle x, \ y, \ z \rangle \neq \langle y, \ z, \ x \rangle\}.$$

Применяем основной алгоритм:

$$\{\langle u, \ v \rangle = \langle \langle x \rangle, \ w \rangle, \ (v \,.\, w) = (w \,.\, v), \ u \neq 1, \ \langle x, \ y, \ z \rangle \neq \langle y, \ z, \ x \rangle\},$$
$$\{u = \langle x \rangle, \ v = w, \ (v \,.\, w) = (w \,.\, v), \ u \neq 1, \ \langle x, \ y, \ z \rangle \neq \langle y, \ z, \ x \rangle\},$$
$$\{u = \langle x \rangle, \ v = w, \ w = v, \ u \neq 1, \ \langle x, \ y, \ z \rangle \neq \langle y, \ z, \ x \rangle\},$$
$$\{u = \langle x \rangle, \ v = w, \ w = w, \ u \neq 1, \ \langle x, \ y, \ z \rangle \neq \langle y, \ z, \ x \rangle\},$$
$$\{u = \langle x \rangle, \ v = w, \ w \neq 1, \ \langle x, \ y, \ z \rangle \neq \langle y, \ z, \ x \rangle\}.$$

Следовательно, исходная система эквивалентна системе

$$\{u = \langle x \rangle, \ v = w\} \cup \{u \neq 1, \ \langle x, \ y, \ z \rangle \neq \langle y, \ z, \ x \rangle\}.$$

Возьмем первое неравенство и, заменив его на уравнение, получим

$$\{u = \langle x\rangle,\ v = w,\ u = 1\},$$
$$\{1 = \langle x\rangle,\ v = w,\ u = 1\}.$$

Поскольку последняя система неразрешима, удаляем неравенство $u \neq 1$, и нам остается привести систему

$$\{u = \langle x\rangle,\ v = w,\ \langle x,\ y,\ z\rangle \neq \langle y,\ z,\ x\rangle\}.$$

Обработка неравенств дает

$$\{u = \langle x\rangle,\ v = w,\ \langle x,\ y,\ z\rangle = \langle y,\ z,\ x\rangle\},$$
$$\{u = \langle x\rangle,\ v = w,\ x = y,\ y = z,\ z = x\},$$
$$\{u = \langle y\rangle,\ v = w,\ x = y,\ y = z,\ z = y\},$$
$$\{u = \langle z\rangle,\ v = w,\ x = z,\ y = z,\ z = z\},$$
$$\{u = \langle z\rangle,\ v = w,\ x = z,\ y = z\}.$$

Следовательно, исходная система эквивалентна следующей системе в приведенной форме:

$$\{u = \langle x\rangle,\ v = w,\ \langle x,\ y\rangle \neq \langle z,\ z\rangle\},$$

и поэтому является разрешимой.

Пример 2
Дана система

$$\{x = \langle x\rangle,\ y = \langle\langle y\rangle\rangle,\ x \neq y\}.$$

Основной алгоритм не меняет эту систему. Поэтому преобразуем неравенства в равенства и получим

$$\{x = \langle x\rangle,\ y = \langle\langle y\rangle\rangle,\ x = y\},$$
$$\{x = y,\ y = \langle y\rangle,\ y = \langle\langle y\rangle\rangle\},$$
$$\{x = y,\ y = \langle y\rangle,\ \langle y\rangle = \langle\langle y\rangle\rangle\},$$
$$\{x = y,\ y = \langle y\rangle\}.$$

Исходная система с неравенствами эквивалентна, таким образом, системе

$$\{x = \langle x\rangle,\ y = \langle\langle y\rangle\rangle,\ \langle\ \rangle \neq \langle\ \rangle\}$$

и является неразрешимой, поскольку невозможно удовлетворить неравенство $\langle\ \rangle \neq \langle\ \rangle$.

Пример 3
Дана система

$$\{x = \langle z,\ x\rangle,\ y = \langle y,\ z\rangle,\ x \neq y\}.$$

Обработка неравенств дает

$$\{x=\langle z,\ x\rangle,\ z=\langle y,\ z\rangle,\ x=y\},$$
$$\{x=y,\ y=\langle z,\ y\rangle,\ y=\langle y,\ z\rangle\},$$
$$\{x=y,\ y=\langle z,\ y\rangle,\ \langle z,\ y\rangle=\langle y,\ z\rangle\},$$
$$\{x=y,\ y=\langle z,\ y\rangle,\ z=y,\ y=z\},$$
$$\{x=z,\ z=\langle z,\ z\rangle,\ z=z,\ y=z\},$$
$$\{x=z,\ y=z,\ z=\langle z,\ z\rangle\}.$$

Исходная система с неравенствами эквивалентна, таким образом, системе в приведенной форме

$$\{x=\langle z,\ x\rangle,\ y=\langle y,\ z\rangle,\ z\neq\langle z,\ z\rangle\}$$

и, следовательно, является разрешимой.

6. УТВЕРЖДЕНИЯ, ОСНОВНЫЕ ЭЛЕМЕНТЫ ПРОГРАММЫ

6.1. Двойное определение

С теоретической точки зрения программа на Прологе служит для определения подмножества A во множестве R наших деревьев. Элементы A называются *утверждениями*, и каждому утверждению может быть, вообще говоря, сопоставлено повествовательное предложение. Несколько примеров такого сопоставления приведено на рис. 24.

Множество A утверждений является, вообще говоря, бесконечным и представляет собой что-то вроде огромной базы данных. Мы увидим несколько позже, что выполнение программы может рассматриваться как обращение к некоторому фрагменту этой базы. Конечно, эту базу нельзя записать в таком виде, как она есть. Она должна быть представлена с помощью конечной информации, которая, однако, достаточна для порождения всей информации, содержащейся в ней.

С этой целью определение множества A утверждений дается с помощью некоторого конечного множества правил, имеющих вид

$$t_0 \rightarrow t_1\ \ldots\ t_n,\ S,$$

где n может быть равно нулю, t_i — термы, а S — система уравнений и неравенств, которая может отсутствовать. В этом случае считаем, что $S=\{\ \}$.

Ниже следует точный синтаксис правил:

⟨*правило*⟩
::=⟨*терм*⟩→⟨*последовательность термов*⟩;
::=⟨*терм*⟩→⟨*последовательность термов*⟩,
⟨*система*⟩;

⟨*последовательность термов*⟩
::=⟨*пусто*⟩
::=⟨*терм*⟩⟨*пробел*⟩⟨*последовательность термов*⟩
::=⟨*паразит*⟩⟨*пробел*⟩⟨*последовательность термов*⟩

Мы сейчас не уточняем понятие ⟨*паразит*⟩, которое, как мы увидим дальше, есть введенное для данного случая средство для вызова программы, написанной не на Прологе.

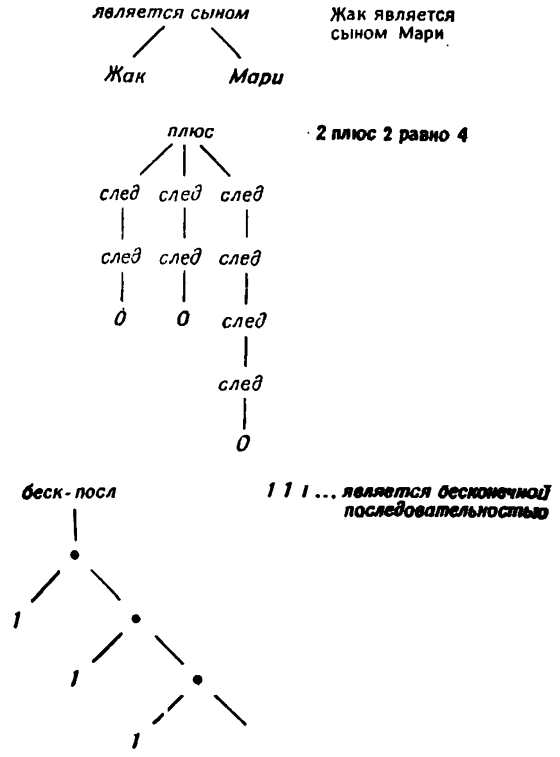


Рис. 24.

Эти правила, имеющие форму

$$t_0 \to t_1 \ \ldots \ t_n,\ S,$$

индуцируют множество, вообще говоря бесконечное, частных правил на деревьях

$$t_0(X) \Rightarrow t_1(X) \ \ldots \ t_n(X),$$

полученных применением всех древесных означиваний

$$X = \{x_1 := s_1,\ \ldots,\ x_m := s_m\},$$

которые являются решениями S и в которых выступают переменные из исходного правила.

Каждое такое частное правило

$$r_0 \Rightarrow r_1 \ \ldots \ r_n$$

может пониматься двумя способами:

(1) Как *правило переписывания*:

вместо r_0 пишем последовательность $r_1 \ \ldots \ r_n$, а в случае $n = 0$

стираем r_0.

(2) Как *логическую импликацию* на подмножестве деревьев A:

из того, что $r_1, r_2, \ldots, r_n$ — элементы A, следует, что r_0 — элемент A.

В случае, когда $n = 0$, эта импликация принимает вид

r_0 — элемент A.

В соответствии с тем, какую из двух интерпретаций мы выбираем, утверждение определяется одним из следующих двух определений.

Определение 1
Утверждения являются деревьями, которые могут быть *стерты* за один или несколько шагов с помощью правил переписывания.

Определение 2
Утверждения образуют наименьшее множество A деревьев, которое удовлетворяет логическим импликациям.

Эти два определения эквивалентны. Чтобы это показать, а также доказать существование наименьшего множества из второго определения, необходимо ввести несколько обозначений.

Слово *пусто* обозначает пустую последовательность; $u \overset{i}{\Rightarrow} v$ означает: последовательность деревьев u переписывается за i шагов в последовательность v с помощью частных правил переписывания. Дадим более строгое определение.

Определение

Пусть u, v — последовательности деревьев, возможно пустые. Тогда $u \overset{i+1}{\Rightarrow} v$ тогда и только тогда, когда существует частное правило $r_0 \Rightarrow r_1 \ldots r_m$ и последовательность, возможно пустая, деревьев $s_1, \ldots, s_n$ такая, что

$$u = r_0 s_1 \ldots s_n \text{ и } r_1 \ldots r_m s_1 \ldots s_n \overset{i}{\Rightarrow} v,$$

$u \overset{0}{\Rightarrow} v$ тогда и только тогда, когда $u = v$.

Легко видеть, что истинность $u \overset{i}{\Rightarrow} v$ означает существование такой последовательности u_i, что

$$u = u_0 \overset{1}{\Rightarrow} u_1 \overset{1}{\Rightarrow} u_2 \overset{1}{\Rightarrow} \ldots \overset{1}{\Rightarrow} u_n \overset{1}{\Rightarrow} v.$$

Двойное определение множества утверждений может теперь быть оправдано следующим свойством.

Свойство двойного определения

Пусть A — множество всех таких деревьев r, что $r \overset{i}{\Rightarrow} пусто$ для некоторого i. Тогда A есть наименьшее множество деревьев, удовлетворяющее логическим импликациям, сопоставленным частным правилам.

Доказательство приводится в приложении. Оно опирается главным образом на принцип независимости стираний, который легко доказывается индукцией по k.

Принцип независимости стирания

Каковы бы ни были деревья $r_1, r_2, \ldots, r_n$, $r_1 \ldots r_n \overset{k}{\Rightarrow} пусто$ тогда и только тогда, когда k есть сумма целых чисел $k = k_1 + \ldots + k_n$ и имеет место

$$r_1 \overset{k_1}{\Rightarrow} пусто,\ r_2 \overset{k_2}{\Rightarrow} пусто,\ \ldots,\ r_n \overset{k_n}{\Rightarrow} пусто.$$

Отсюда следует, что при стирании последовательности деревьев можно в любой момент менять их порядок и, следовательно, игнорировать требование переписывать всегда первое дерево, как того требует определение $\overset{i}{\Rightarrow}$.

6.2. Примеры

Пример 1
Пусть даны правила:

входит (*nil*, *nil*) → ;
входит (*x*, *l*, *y*) → *элемент* (*e*) *входит* (*x*, *y*);
входит (*e*.*x*, *e*.*y*) → *элемент* (*e*) *входит* (*x*, *y*);
элемент ("*a*") → ;
элемент ("*b*") → ;
элемент ("*c*") → ;
элемент ("*d*") → ;

Эти правила индуцируют следующие частные правила:

'*входит* (*nil*, *nil*)' ⇒ *пусто*

. . . . . .

'*входит* ("*d*".*nil*, "*c*"."*d*".*nil*)' ⇒
  '*элемент* ("*c*")' '*входит* ("*d*".*nil*, "*d*".*nil*)'
'*входит* ("*b*"."*d*".*nil*, "*a*"."*b*"."*c*"."*d*".*nil*)' ⇒
  '*элемент* ("*a*")' '*входит* ("*b*"."*d*".*nil*, "*b*"."*c*"."*d*".*nil*)'

. . . . . .

'*входит* ("*d*".*nil*, "*d*".*nil*)' ⇒
  элемент("*d*") '*входит*(*nil*, *nil*)
'*входит*("*b*"."*d*".*nil*, "*b*"."*c*"."*d*".*nil*)' ⇒
  '*элемент*("*b*")' '*входит*("*d*".*nil*, "*c*"."*d*".*nil*)

. . . . . .

'*элемент*("*a*")' ⇒ *пусто*
'*элемент*("*b*")' ⇒ *пусто*
'*элемент*("*c*")' ⇒ *пусто*
'*элемент*("*d*")' ⇒ *пусто*

Имеем:

'*входит*("*b*"."*d*".*nil*, "*a*"."*b*"."*c*"."*d*".*nil*)' $\overset{1}{\Rightarrow}$

'*элемент*("*a*")' '*входит*("*b*"."*d*".*nil*, "*b*"."*c*"."*a*".*nil*)' $\overset{1}{\Rightarrow}$

'*входит*("*b*"."*d*".*nil*, "*b*"."*c*"."*d*".*nil* $\overset{1}{\Rightarrow}$

'*элемент*("*b*")' '*входит*("*d*".*nil*, "*c*"."*d*".*nil*)' $\overset{1}{\Rightarrow}$

'*входит*("*d*".*nil*, "*c*"."*d*".*nil*)' $\overset{1}{\Rightarrow}$

'*элемент*("*c*")' '*входит*("*d*".*nil*."*d*".*nil*)' $\overset{1}{\Rightarrow}$

'*входит*("*d*".*nil*, "*d*".*nil*)' $\overset{1}{\Rightarrow}$

'*элемент*("*d*")' '*входит*(*nil*), *nil*)' $\overset{1}{\Rightarrow}$

'*входит*(*nil*, *nil*)' $\overset{1}{\Rightarrow}$ *пусто*

и, следовательно,

$$'входит("b"."d".nil,"a"."b"."c"."d".nil)' \overset{9}{\Rightarrow} пусто$$

это доказывает, что мы имеем дело с утверждением. Схема, приведенная на рис. 25, хорошо иллюстрирует принцип независимости стираний в этом 9-шаговом выводе.

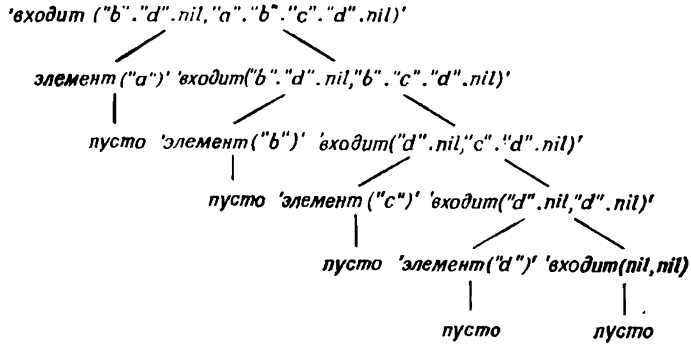


Рис. 25.

Эта древовидная схема, просматриваемая снизу вверх, воспроизводит множество логических импликаций, которые, по определению 2, делают из этого дерева утверждение

$$'входит(b.c.nil, a.b.c.d.nil)'$$

Вообще, легко видеть, что, кроме утверждений *элемент* (*"a"*) ... *элемент* (*"d"*), все утверждения имеют вид *входит* (l_1, l_2), где l_1 — список, полученный вычеркиванием каких-либо элементов из списка l_2.

Пример 2

Даны правила:

$$плюс(0, x, x) \rightarrow;$$
$$плюс(след(x), y, след(z)) \rightarrow плюс(x, y, z);$$

Эти правила индуцируют такие, например, частные правила:

$$'плюс(0, след(след(0)), след(след(0)))' \Rightarrow пусто$$
. . . . . .
$$'плюс(след(0), след(след(0)), (след(след(след(0)))) \Rightarrow$$
$$'плюс(0, след(след(0)), след(след(0)))'$$
$$'плюс(след(след(0)), след(след(0)), след(след(след(след(0))))' \Rightarrow$$
$$'плюс(след(0), след(след(0)), след(след(след(0))))'$$

Имеем:

$'плюс(след(след(0)), след(след(0)), след(след(след(след))))' \overset{1}{\Rightarrow}$

$'плюс(след(0), след(след(0)), след(след(след(0))))' \overset{1}{\Rightarrow}$

$'плюс(0, след(след(0)), след(след\ (0)))' \overset{1}{\Rightarrow} пусто$

и, следовательно,

$'плюс(след(след(0)), след(след(0)), след(след(след(след(0)))))' \overset{3}{\Rightarrow}$
$пусто$

откуда получаем утверждение

$'плюс(след(след(0)), след(след(0)), след(след(след(след(0)))))'$

На первый взгляд кажется, что утверждения этого примера имеют вид *плюс* (x, y, z), где $x + y = z$, причем натуральные числа изображаются в виде *след* от *след* от ... *0*.

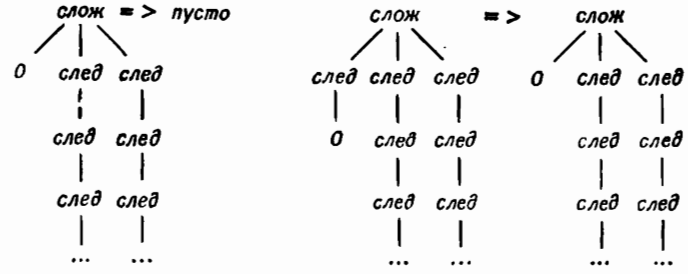


Рис. 26.

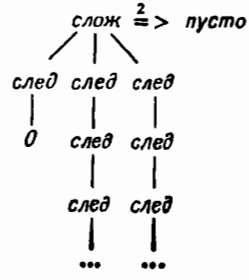


Рис. 27.

Однако это не так, поскольку эти правила порождают частные правила с бесконечными деревьями (рис. 26). Из этих правил выводится утверждение, изображенное на рис. 27, которое можно понимать так: «1 плюс бесконечность равно

бесконечности». Если мы действительно хотим, чтобы *плюс* (x, y, z) соответствовал сложению натуральных чисел, надо изменить правила следующим образом:

плюс(*0*, *x*, *x*) → *целое*(*x*);
плюс(*след*(*x*), *y*, *след*(*z*)) → *плюс*(*x*, *y*, *z*);
целое(*0*) →;
целое(*след*(*x*)) → *целое*(*x*);

Пример 3
В этом последнем примере появляется неравенство:

вне(*x*, *nil*) →;
вне(*x*, *y*.*l*) → *вне*(*x*, *l*), {*x* # *y*};

Эти правила порождают такие частные правила:

$'вне("d".nil)' \Rightarrow пусто$
. . . . . .
$'вне("d","c".nil)' \Rightarrow 'вне("d",nil)'$
$'вне("d","b"."c".nil)' \Rightarrow 'вне("d","c".nil)$
$'вне("d","a"."b"."c".nil)' \Rightarrow 'вне("d","c".nil)$
. . . . . .

которые дают вывод:

$'вне("d","a"."b"."c".nil)' \overset{1}{\Rightarrow}$
$'вне("d","b"."c".nil)' \overset{1}{\Rightarrow}$
$'вне("d","c".nil)' \overset{1}{\Rightarrow}$
$'вне("d",nil)' \overset{1}{\Rightarrow} пусто$

Вообще, все утверждения этого примера имеют вид *вне* (x, l), где l — список, в котором все элементы отличны от x.

7. ИСПОЛНЕНИЕ ПРОГРАММЫ: ВЫЧИСЛЕНИЕ ПОДМНОЖЕСТВА УТВЕРЖДЕНИЙ

7.1. Проблема наложения шаблона

Мы собираемся показать, какая информация неявно содержится в программе на Прологе, однако мы еще не сказали, что такое исполнение программы. Исполнение программы — это решение следующей проблемы, называемой «проблемой наложения шаблона».

Пусть дана программа, которая является рекурсивным описанием некоторого множества A утверждений, и пусть дан *шаблон*, т. е. терм t и множество $\{x_1, \ldots, x_n\}$ переменных; тогда требуется найти все утверждения, на которые накладывается этот *шаблон*, т. е. вычислить все древесные означивания $X = \{x_1 := r_1, \ldots, x_n := r_n\}$, для которых $t(X)$ есть утверждение.

Для решения этой проблемы мы вводим бинарное отношение $\xrightarrow{i}$ на парах (u, S), где u — последовательность термов, возможно пустая, а S — система уравнений и неравенств.

Определение

Пусть u, v — последовательности, возможно пустые, термов, а S, T — системы. Тогда отношение $\xrightarrow{i}$ определяется следующим образом:

(1) $(u, S) \xrightarrow{i+1} (v, T)$ тогда и только тогда, когда существует правило $s_0 \to s_1 \ldots s_m$, U, переменные которого переименованы так, чтобы не совпадать с переменными из (u, S), $u = t_0\ t_1 \ldots t_n$, где t_i — термы, и имеет место

$$(s_1 \ldots s_m\ t_1 \ldots t_n,\ S \cup U \cup \{t_0 = s_0\}) \xrightarrow{i} (v, T);$$

(2) $(u, S) \xrightarrow{0} (v, T)$ тогда и только тогда, когда $u = v$, $S = T$ и S разрешима.

Отношение $\xrightarrow{i}$ имеет несколько свойств, аналогичных свойствам отношения $\overset{i}{\Rightarrow}$.

Во-первых, из определения непосредственно следует, что истинность $(u, S) \xrightarrow{i} (v, T)$ влечет за собой существование последовательности (u_i, S_i) такой, что

$$(u, S) = (u_0, S_0) \xrightarrow{1} (u_1, S_1) \xrightarrow{1} \ldots \xrightarrow{1} (u_n, S_n) = (v, T).$$

Во-вторых, мы встречаем здесь в слегка отличной форме принцип независимости стираний, доказываемый индукцией по k.

Обобщенный принцип независимости стираний

Для любых термов $t_1, \ldots, t_n$ и систем S и T имеет место $(t_1 \ldots t_n, S) \xrightarrow{k} (\textit{пусто}, T)$ тогда и только тогда, когда k есть сумма n чисел $k = k_1 + k_2 + \ldots + k_n$, а T — объединение систем $T = T_1 \cup T_2 \cup \ldots \cup T_n$, не имеющих других общих переменных, кроме тех, которые содержатся в $(t_1 \ldots t_n, S)$, причем $(t_1, S) \xrightarrow{k_1} (\textit{пусто}, T_1) \ldots$ $\ldots (t_n, S) \xrightarrow{k_n} (\textit{пусто}, T_n)$.

Отсюда следует, что при стирании (с помощью $\xrightarrow{i}$) последовательности u термов, входящих в (u, S), эти термы можно переставлять на любом шаге. И последняя аналогия, которая позволяет заключить, что $\xrightarrow{i}$ есть обобщение отношения $\overset{i}{\Rightarrow}$.

Принцип обобщения

Пусть даны i — неотрицательное число, $t_1 \ldots t_n$ — последовательность (возможно пустая) термов, S — система и X — древесное означивание для переменных, содержащихся в $(t_1 \ldots t_n, S)$. Тогда X является древесным решением S и $t_1(X) \ldots t_n(X) \overset{i}{\Rightarrow} \textit{пусто}$ тогда и только тогда, когда существует система T, для которой имеет место

$(t_1 \ldots t_n, S) \xrightarrow{i} (\textit{пусто}, T)$ и X — древесное решение T.

Мы даем доказательство этого принципа в приложении. Используя частный случай этого принципа для $n = 1$, $S = \{\}$ совместно с определением 1 множества утверждений, мы получаем принцип шаблона.

Принцип шаблона

Для всякого терма t и всякого древесного означивания X его переменных $t(X)$ есть утверждение тогда и только тогда, когда существует система S и натуральное i, для которых выполняется

$(t, \{\}) \overset{i}{\Rightarrow} (\textit{пусто}, S)$ и X — древесное решение системы S.

Чтобы решить нашу исходную задачу, достаточно перечислить все последовательности

$$(t, \{\}) = (v_0, S_0) \xrightarrow{1} (u_1, S_1) \xrightarrow{1} (u_2, S_2) \xrightarrow{1} \ldots$$

и попытаться достичь те S_i, у которых соответствующие u_i пусты. Тогда всякое древесное означивание X для переменных шаблона t, которое делает $t(X)$ утверждением, будет решением такого S_i. Конечно, здесь возникают и алгоритмические проблемы из-за того, что некоторые такие последовательности могут быть бесконечными.

Чтобы все это проиллюстрировать, мы вновь обратимся к трем примерам из предыдущего раздела.

7.2. Примеры

Пример 1

Даны правила:

входит(*nil*, *nil*) →;
входит(*x*, *e*.*y*) → *элемент*(*e*)*входит*(*x*, *y*);
входит(*e*.*x*, *e*.*y*) → *элемент*(*e*)*входит*(*x*, *y*);
элемент("*a*") →;
элемент ("*b*") →;
элемент ("*c*") →;
элемент("*d*") →;

Требуется найти все утверждения, на которые накладывается шаблон

$$t = входит(z, "a"."b"."c"."d".nil)$$

т. е. вычислить все означивания

$$X = \{z := r\},$$

для которых $t(X)$ является утверждением. Имеем последовательно (S_i' обозначает приведенную форму для S_i)

$u0 = входит(z, "a"."b"."c"."d".nil)$
$S0 = \{\ \}$
$S0' = \{\ \}$
$u1 = элемент(e)\ входит(x, y)$
$S1 = S0 \cup \{входит(z, "a"."b"."c"."d".nil) = входит(x, e.y)\}$
$S1' = \{z = x, y = "b"."c"."d".nil, e = "a"\}$
$u2 = входит(x, y)$
$S2 = S1 \cup \{элемент(e) = элемент("a")\}$
$S2' = \{z = x, y = "b"."c"."d".nil, e = "a"\}$
$u3 = элемент(e')\ входит(x', y')$
$S3 = S2 \cup \{входит(x, y) = входит(e'.x', e'.y')\}$
$S3' = \{z = x, x = e'.x', y = e'.y', y' = "c"."d".nil, e = "a",$
$e' = "b"\}$
$u4 = входит(x', y')$

$S4 = S3 \cup \{элемент(e') = элемент(''b'')\}$
$S4' = \{z = x, x = e'.x', y = e'.y', y' = ''c''.''d''.nil,$
$e = ''a'', e' = ''b''\}$
$u5 = элемент(e'')\ входит(x'', y'')$
$S5 = S4 \cup \{входит(x', y') = входит(x'', e''.y'')\}$
$S5' = \{z = x, x = e'.x', x' = x'', y = e'.y', y' = e''.y'',$
$y'' = ''d''.nil, e = ''a'', e' = ''b'', e'' = ''c''\}$
$u6 = входит(x'', y'')$
$S6 = S5 \cup \{элемент(e'') = элемент(''c'')\}$
$S6' = \{z = x, x = e'.x', x' = x'', y = e'.y', y' = e''.y'',$
$y'' = ''d''.nil, e = ''a'', e' = ''b'', e'' = ''c''\}$
$u7 = элемент(e''')\ входит(x''', y''')$
$S7 = S6 \cup \{входит(x'', y'') = входит(e'''.x''', e'''.y''')\}$
$S7' = \{z = x, x = e'.x', x' = x'', x'' = e'''.x''', y = e'.y',$
$y' = e''.y'', y'' = e'''.y''', y''' = nil, e = ''a'', e' = ''b'',$
$e'' = ''c'', e''' = ''d''\}$
$u8 = входит(x''', y''')$
$S8 = S7 \cup \{элемент(e''') = элемент(''c'')\}$
$S8' = \{z = x, x = e'.x', x' = x'', x'' = e'''.x''', y = e'.y',$
$y' = e''.y'', y'' = e'''.y''', y''' = nil, e = ''a'', e' = ''b'',$
$e'' = ''c'', e''' = ''d''\}$
$u9 = пусто$
$S9 = S8 \cup \{входит(x''', y''') = входит(nil, nil)\}$
$S9' = \{z = x, x = e'.x', x' = x'', x'' = e'''.x''', x''' = nil,$
$y = e'.y', y' = e''.y'', y'' = e'''.y''', y''' = nil,$
$e = ''a'', e' = ''b'', e'' = ''c'', e''' = ''d''\}$

Итак, мы получили

$$X = \{z := ''b''.''d''.nil\}$$

Таким же образом можно получить еще 15 других подсписков списка $''a''.''b''.''c''.''d''.$ nil, отвечающих всем возможным способам вычеркивания из него каких-либо элементов. Древовидная схема из рис. 28 хорошо иллюстрирует принцип независимости стирания для случая $\xrightarrow{9}$.

Пример 2
Даны правила:

$плюс(0, x, x) \rightarrow;$
$плюс(след(x), y, след(z)) \rightarrow плюс(x, y, z);$

Чтобы вызвать появление бесконечных деревьев, давайте вычислим утверждения, на которые накладывается шаблон

$$t = плюс(след(0), x, x),$$

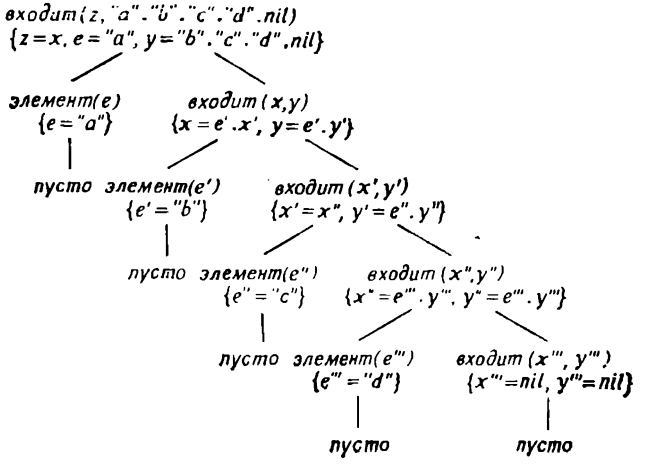


Рис. 28.

т. е. найдем такие древесные означивания

$$X = \{x := r\},$$

при которых $t(X)$ — утверждение.

Имеем последовательно

$u0 = плюс(след(0), x, x)$
$S0 = \{\ \}$
$S0' = \{\ \}$
$u1 = плюс(y', y', z')$
$S1 = S0 \cup \{плюс(след(0), x, x) = плюс(след(x'), y', след(z'))\}$
$S1' = \{x = y', x' = 0, y' = след(z')\}$
$u2 = пусто$
$S2 = S1 \cup \{плюс(x', y', z') = плюс(0, x'', y'')\}$
$S2' = \{x = x'', x' = 0, x'' = след(x''), y' = x'', z' = x''\}$

Единственное решение представлено на рис. 29.

Пример 3

Даны правила:

$вне(x, nil) \rightarrow$;
$вне(x, y.l) \rightarrow вне(x, l), \{x \neq y\}$;
Выберем шаблон:

$$t = вне("c", "a"."b".nil).$$

Требуется, следовательно, проверить, что $"t(\{\ \})"$ является утверждением.

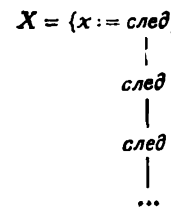


Рис. 29.

Действительно, имеем последовательно

$$
\begin{aligned}
&u0 = плюс(след(0), x, x)\\
&S0 = \{\ \}\\
&S0' = \{\ \}\\
&u1 = вне(x, l)\\
&S1 = S0 \cup \{вне("c", "a"."b".nil) = вне(x, y.l), x \neq y\}\\
&S1' = \{x = "c", l = "b".nil, y = "a"\}\\
&u2 = вне(x', l')\\
&S2 = S1 \cup \{вне(x, l) = вне(x', y'.l), x' \neq l'\}\\
&S2' = \{x = x', x' = "c", l = y'.l', l' = nil, y = "a", y' = "b"\}\\
&u3 = пусто\\
&S3 = S2 \cup \{вне(x', l,) = вне(x'', nil)\}\\
&S3' = \{x = x', x' = x'', x'' = "c", l = y'.l', l' = nil, y = "a",\\
&\qquad y' = "b"\}
\end{aligned}
$$

8. ПРОЛОГ-МАШИНА

8.1. Пролог-часы

Язык Пролог, как мы уже видели, позволяет, с одной стороны, задавать в неявном виде бесконечные множества утверждений; с другой стороны, он позволяет вычислять — иначе говоря, перечислять — некоторые интересующие нас утверждения. Это вычисление происходит как перечисление последовательности пар:

$$(u_0, S_0) \xrightarrow{i} (u_1, S_1) \xrightarrow{i} (u_2, S_2) \xrightarrow{i}$$

До сих пор мы всегда предполагали, что u_0 сведено к одному терму t и S_0 — пустое множество $\{\ \}$. Если пересмотреть принцип обобщения и использовать одновременно 1-е определение утверждения и принцип независимости стираний для $\xrightarrow{i}$, то мы придем к принципу шаблона в более общем виде, где u_0 будет некоторой последовательностью $t_1 \ldots t_n$ и S_0 — некоторой системой S уравнений и неравенств.

Принцип расширенного шаблона

Пусть $\{t_1, \ldots, t_n\}$ — множество термов, S — система, $\{x_1, \ldots, x_m\}$ — множество использованных переменных и пусть X — некоторое древесное означивание вида $X = \{x_1 := r_1, \ldots, x_m := r_m\}$, тогда X является решением S и $\{t_1(X), \ldots, t_n(X)\}$ содержится во множестве утверждений тогда и только тогда, когда существует целое число i и система T, для которой

$(t_1 \ldots t_n, S) \xrightarrow{i}$ (*пусто*, T), причем X является решением T.

Перечисление пар (u_i, S_i) может происходить различными способами, в частности с использованием различных порядков перечисления. Чтобы все это уточнить, а также дать независимое представление описанной до сих пор модели, мы введем операционную семантику Пролога с помощью абстрактной машины, названной Пролог-часами, поскольку время играет в ней важную роль.

Эта машина состоит из следующих частей:

(1) ячейка *последовательность-правил*, содержащая последовательность правил, образующую собственно программу на Прологе;

(2) ячейка t, содержащая время. Время — это не что иное, как текущий индекс i пары (u_i, S_i);

(3) три неограниченных множества ячеек, каждое из которых проиндексировано неотрицательным целым числом i:
   (а) *цели*(i) содержит последовательность термов u_i;
   (б) *система*(i) содержит систему S_i;
   (в) *правила*(i) содержит последовательность термов, *активных* в момент i.

Только ячейка *последовательность-правил* обладает начальным значением, которое является отправной точкой для программиста. Машина считывает команды со своего входного устройства, исполняет их одну за другой и каждый раз печатает вычисленный результат. Синтаксис команд таков:

⟨*команда*⟩
::=⟨*последовательность термов*⟩;
::=⟨*последовательность термов*⟩,⟨*система*⟩;

Отсутствие в команде члена ⟨*система*⟩ означает, естественно, что система пуста. Функционирование машины схематически изображено на рис. 30.

На этом рисунке:

ГОЛОВА(x) означает либо первый элемент последовательности x, либо левую часть правила x.

ХВОСТ(*x*) означает либо последовательность *x* без первого элемента, либо правую часть (без системы) правила *x*.
УСЛОВИЯ(*r*) означает систему уравнений и неравенств правила *r*.

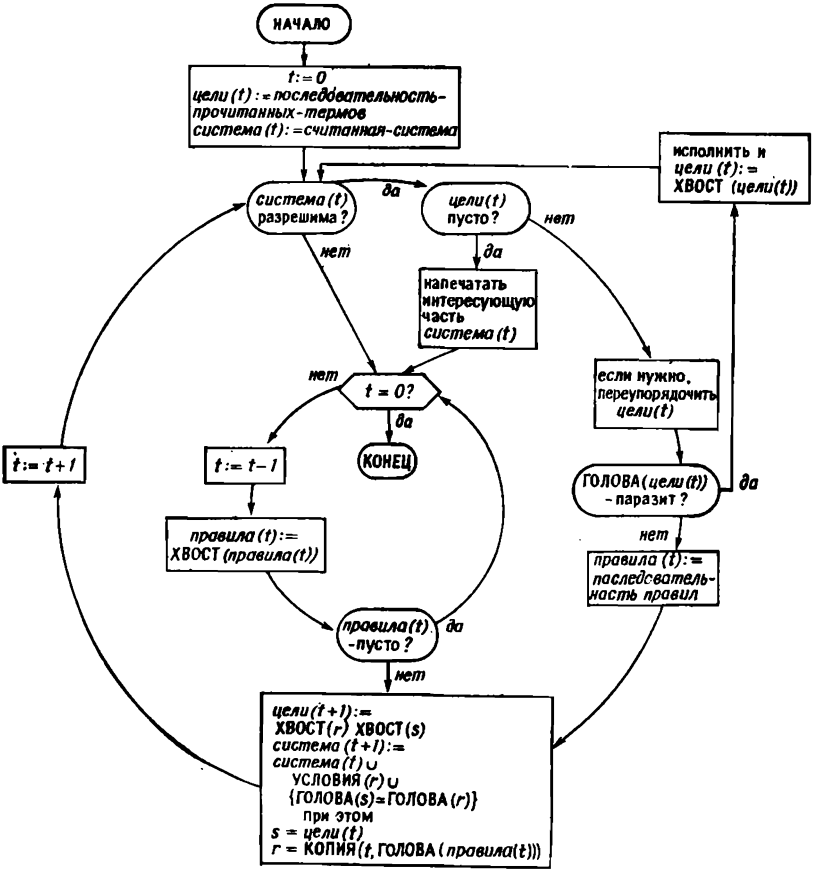


Рис. 30. Пролог-часы.

КОПИЯ(*t*, *r*) означает *t*-ю копию правила *r*. Эта копия не имеет общих переменных ни с ячейкой *цели*(*t*), ни с ячейкой *система*(*t*).

Паразиты — это имена подпрограмм, которые выполняют различные функции.

⟨*паразит*⟩
::=/
::=⟨*синтаксис неизвестен, но отличен от синтаксиса терма*⟩

Лишь один паразит «/», действие которого мы опишем несколько ниже, непосредственно доступен программисту. Остальные паразиты используются внутри правил, составляющих большое множество встроенных правил, которые приводятся в действие в начальный момент в ячейке *последовательность-правил*. Это множество встроенных правил дает весьма полную программную среду, позволяющую

(1) управлять и изменять исполнение программы;
(2) изменять текущее множество правил, содержащихся в ячейке *последовательность-правил* и, таким образом, вводить и изменять программы;
(3) иметь доступ к классическим функциям арифметики и к функциям работы с символьными строками;
(4) поддерживать ввод-вывод.

8.2. Управление

Управление осуществляется с помощью паразита «/» и встроенных правил, которые будут перечислены и объяснены ниже. Чтобы иметь возможность называть паразитов, встречающихся в правилах (а их синтаксис, вообще говоря, неизвестен программисту), мы используем идентификаторы, заключенные в одинарные кавычки.

ПАРАЗИТ «/»

Пролог-часы имитирует недетерминированную машину. Эта имитация проводится с помощью техники бектракинга в накопленные точки разветвления. Это накопление информации имеет пределы, и необходимо суметь подавить все такие точки между моментом в прошлом $t-k$ и настоящим моментом t.

Выполнение паразита «/» вызывает присвоение значения *пусто* всем ячейкам *правило*(i), у которых индекс i является элементом множества $\{t-k, \ldots, t-1, t\}$. Число $t-k$ — это последний такой момент, когда ячейка *цели* $(t-k)$ не содержала данное вхождение «/».

ВСТРОЕННОЕ ПРАВИЛО: *связанная*$(x) \rightarrow$ *'связанная'*;

Исполнение паразита 'связанная' не имеет никакого эффекта в случае, когда x *связана*; в противном случае *система*(t) делается неразрешимой.

Переменная x считается *связанной*, если имеет место один из следующих трех случаев:

(1) существует такое значение k константы, что во всех решениях $\{x := r\}$ *системы*(t) мы имеем $r = k$;
(2) во всех решениях $\{x := r\}$ *системы*(t) r имеет вид $r = r_1.r_2$;

(3) если существует целое n такое, что во всех решениях $\{x := r\}$ *системы*(t) терм r имеет вид $r = \langle r_1, \ldots, r_n \rangle$.

ВСТРОЕННОЕ ПРАВИЛО: свободная$(x) \rightarrow$ *'свободная'*;

Исполнение паразита *'свободная'* не имеет никакого эффекта, когда x *свободна*; в противном случае *система*(t) делается неразрешимой.

Переменная x считается *свободной*, если x не связана.

ВСТРОЕННОЕ ПРАВИЛО: заморозить $(x, p) \rightarrow$ *'в-ожидании'* x p

Обобщенный принцип независимости стираний позволяет в любой момент изменить порядок элементов последовательности *цели*(t) и, в частности, задержать переписывание конкретного элемента p этой последовательности до тех пор, пока переменная x не станет связанной. Данное встроенное правило и призвано решить эту задачу.

Обычно паразит *'в-ожидании'* не выполняется, а лишь учитывается в операции, которая состоит в переупорядочивании последовательности *цели*(t) в Пролог-часах. Это переупорядочивание производится на два счета:

(1) сначала все тройки вида *'в-ожидании'* x p, встречающиеся в *цели*(t), переносятся в конец последовательности;

(2) затем все те p из этих троек, у которых переменная x стала связанной в момент t, переносятся в начало *цели*(t) и все остатки этих троек, т. е. *'в-ожидании'* и x, удаляются.

Все эти перегруппировки учитывают порядок элементов p в *цели*(t). Если же, несмотря на постоянное удаление паразита *'в-ожидании'*, придется все-таки его исполнить, то это приведет к удалению терма x, стоящего за ним.

Вот интересный пример использования правила *заморозить*(x, p). Речь идет о программе, позволяющей проверить тот факт, что терм x является и всегда будет оставаться конечным деревом.

конечное-дерево$(x) \rightarrow$ *конечная-ветвь*(x, nil);
конечная-ветвь$(x, l) \rightarrow$ *заморозить*$(x,$ *конечная-ветвь*$(x, l))$;
конечная-ветвь'$(x, l) \rightarrow$ *вне*(x, l) *доминирует* (x, l')
конечные ветви(l', x, l);
конечные-ветви$(nil, l) \rightarrow$;
конечные-ветви$(x.l', l) \rightarrow$ *конечная-ветвь*(x, l)
конечные-ветви(x, l) *конечные-ветви*(l', l);

вне(*x*, *nil*) →;
вне(*x*, *y*.*l*) → *вне*(*x*, *l*), {*x* ≠ *y*};
доминирует(,) является встроенным предикатом с паразитами, которые имитируют псевдопрограмму;

доминирует(*x*, *l*) → *свободная*(*x*)/*ошибка*("*в доминирует*");
доминирует(*x*, *nil*) → *константа*(*x*);
доминирует(*x1*.*x2*, *x1*.*x2*.*nil*) →;
доминирует(⟨ ⟩, *nil*) →;
доминирует(⟨*x1*⟩, *x1*.*nil*) →;
доминирует(⟨*x1*, *x2*⟩, *x1*.*x2*.*nil*) →;
доминирует(⟨*x1*, *x2*, *x3*⟩, *x1*.*x2*.*x3*.*nil*) →
. . . . . .
константа(*x*) → . . . . .
ошибка(*m*) → . . .

ВСТРОЕННОЕ ПРАВИЛО: *блок*(*n*, *p*) → *p*'*метка*'*n*/;

Это правило позволяет вставить метку *n* в последовательность *цели*(*t*), давая, таким образом, возможность прервать переписывание некоторого числа термов посредством передачи управления на эту метку. Эти метки можно вставлять рекурсивно, что приводит к образованию на последовательности *цели*(*t*) структуры накладывающихся блоков, имеющих в качестве имен соответствующие метки. Передать управление на метку *n* означает мгновенно окончить стирание блока с именем *n*.

Исполнение паразита *'метка'* приводит к удалению из *цели*(*t*) терма, непосредственно следующего за вхождением этого паразита.

ВСТРОЕННОЕ ПРАВИЛО: *конец-блока*(*n*) → *'конец-блока'*;

Это и есть то правило, которое позволяет мгновенно оборвать стирание первого охватывающего блока с именем *n*. При этом нельзя игнорировать некоторые *правила-паразиты*. Они называются *паразитами восстановления* и все равно исполняются. Более точно, исполнение паразита *'конец-блока'* происходит по схеме, изображенной на рис. 31.

ВСТРОЕННОЕ ПРАВИЛО: *p*.*q* → *p q*;

Это правило позволяет слить последовательность целей в единую цель.

ВСТРОЕННОЕ ПРАВИЛО: *рав*(*x*, *x*) →;

Это правило позволяет избежать уравнений в правилах или командах.

ВСТРОЕННОЕ ПРАВИЛО: *разл*(*x*, *y*) →, {*x* # *y*};

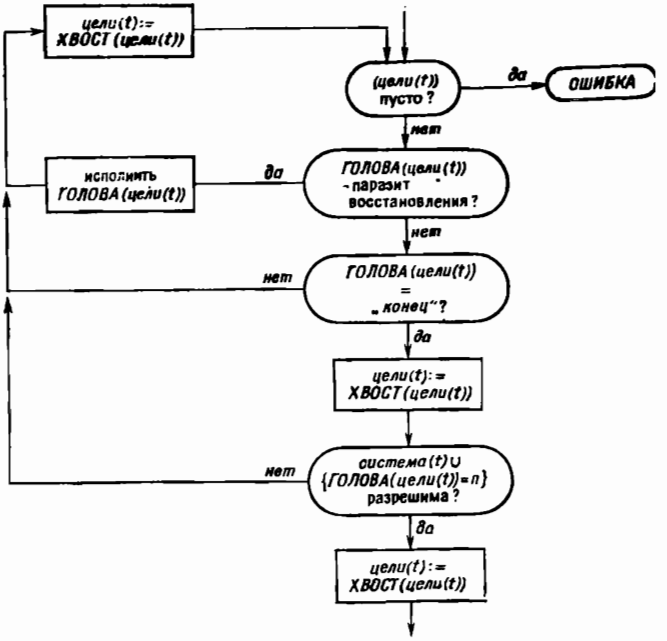


Рис. 31. Исполнение паразита *конец-блока*.

Это правило позволяет избежать неравенств в правилах и командах.

8.3. Кратко об управлении выполнением программ

В любой момент можно изменить содержание ячейки *последовательность-правил*, добавляя или удаляя некоторые правила. Эти изменения производятся обычно относительно некоторого текущего указателя, положение которого можно менять. Наиболее частое изменение — вставление последовательности правил. Эта последовательность является обычно программой с комментариями. Ее синтаксис:

⟨*программа*⟩
  ::=;
  ::=⟨*высказывание*⟩⟨*программа*⟩
⟨*высказывание*⟩
  ::=⟨*комментарий*⟩
  ::=⟨*правило*⟩
⟨*комментарий*⟩
  ::=⟨*цепочка*⟩

Вставка производится с помощью встроенного правила

вставить → *'вставить'*

Паразит *'вставить'* считывает программу с устройства ввода и вставляет ее сразу за текущим указателем. Поскольку комментарии не играют никакой роли при исполнении программы, они запоминаются отдельно и могут быть восстановлены.

Существует одно важное ограничение в синтаксисе правил программы. Цель этого ограничения — оптимизировать поиск во множестве правил, что достигается с помощью прямого доступа к правилам на основе идентификаторов. Это ограничение звучит так:

(1) всякий терм, составляющий левую часть правила, должен содержать по крайней мере одно вхождение идентификатора, которому не предшествуют (слева) ни переменная, ни константа, ни ⟨ ⟩;

(2) терм, образующий левую часть правила, не может иметь вид $t_1 . t_2$. Исключение составляет встроенное правило $p . q — p\,q$;

Для того чтобы организовать удобным образом исполнение больших множеств правил, эти множества разделяются на подмножества, называемые *мирами*. Каждый из таких миров имеет имя M, состоящее из последовательности цепочек:

$$M = c_1,\ c_2, \ldots,\ c_n.$$

Будем говорить, что мир с именем M_2 является *подмиром* мира с именем M_1, если M_1 и M_2 имеют вид

$$M_1 = c_1, \ldots,\ c_m,\ M_2 = c_1, \ldots,\ c_m,\ c_{m+1}, \ldots,\ c_n,$$

где n строго больше m. Если $n = m + 1$, то говорят, что M_2 есть непосредственный подмир мира M_1. Это понятие подмира порождает иерархию миров, в вершине которой находится мир с пустым именем.

Каждому вхождению идентификатора сопоставляется некоторый мир. Сопоставление происходит в момент считывания этого вхождения и зависит от текущего мира M, в котором мы находимся в даный момент. Возможны два случая:

(1) Мир M является подмиром мира *Mbis*, сопоставленного какому-то другому вхождению этого же идентификатора. В этом случае данному вхождению идентификатора сопоставляется мир *Mbis*.

(2) В противном случае считанному вхождению идентификатора сопоставляется текущий мир M.

Два вхождения идентификатора, состоящие из одной и той же последовательности символов, рассматриваются как

равные только в том случае, когда им сопоставлен один и тот же мир. Более точно, значение вхождения идентификатора *id*, которому сопоставлен мир *M*, есть пара (M, id).

Приложения

ПРИЛОЖЕНИЕ I. ПРОЛОГ НА МАШИНЕ APPLE II

П.1.1. Введение

Объясним коротко, почему мы решили реализовать Пролог на машине Apple II.

После того как был описан язык Пролог II, требовалось найти машину для его реализации. У нас не было большой машины, однако у всех нас была Apple II. С другой стороны, мы уже реализовали экспериментальную версию Пролога на микрокомпьютере M6800 (Exorciser), что подтвердило возможность такой его реализации.

Была еще одна важная причина, которая руководила нашим выбором. Так как языки искусственного интеллекта были до этого времени доступны только на больших машинах (DEC-10), мы решили освободить искусственный интеллект из этого гетто и дать возможность всем использовать его средства.

Реализация

Чтобы реализовать Пролог II, нам были необходимы три вещи:

(1) Корректная операционная система, и Паскаль UCSD отвечал этому критерию.

(2) Язык для реализации наших инструментов поддержки, и Паскаль прекрасно для этого подходил.

(3) Много памяти. Здесь проблема осложнялась, так как Apple II располагает небольшой памятью. Поэтому мы решили реализовать виртуальную память на гибких дисках. Надо отметить, что мы были среди первых, кто попытался сделать подобное на гибких дисках. Результат превзошел все ожидания! Все же, несмотря на это, производительность системы сильно снижается.

Эта система реализована на Apple II, имеющей

— 48 K памяти;
— 2 гибких диска по 140 K;
— языковую плату с Паскалем UCSD.

Вот несколько деталей реализации:

(1) Виртуальная машина включена в систему Паскаль и состоит из 10 страниц на Паскале (для ввода/вывода и виртуальной памяти) и 15 страниц языка машины 6502 (для исполнения команд).

(2) Программа Micromegas интерпретатора Пролога занимает 12 K

(3) Виртуальная память реализована на двух гибких дисках, и зона реальных страниц занимает 40 страниц по 256 байтов.

Версия Пролога II на Apple II функционирует с сентября 1981 года

П.1.2. Эффективность

Эта версия, хотя и довольно медленная, тем не менее остается очень полезной из-за ее большой интерактивности. Первое же использование этой системы показало нам, что наличие Пролога на микрокомпьютере сильно стимулирует написание программ на этом языке.

Необходнмо понимать, что Пролог II — не игрушка, а реализация большой системы на микрокомпьютере неизбежно приводит к ухудшению эффективности.

Система функционирует со скоростью 10 липс (логических выводов в секунду) и позволяет исполнять программы на Прологе объемом до 40 страниц.

П.1.3. Использование Пролога II на Apple II

Мы использовали Пролог на Apple II для:

(1) Обучения Прологу в рамках курса по информатике.

(2) Реализации проектов на 3-м курсе (факультет искусственного интеллекта).

(3) Проведения наших исследований. Производительность системы не причиняет неудобств в процессе отладки программ, так как всегда можно выбирать для тестирования короткие примеры. Тем не менее мы без всяких проблем пускали программы в течение более 12 часов.

Эта система использовалась в течение более одного года всеми нашими студентами, что позволило нам получить Пролог, практически лишенный ошибок.

Мы думаем, что для знакомства с Прологом эта версия на Apple II оказалась очень удобным инструментом. Она позволяет также реализовать исследования по искусственному интеллекту в тех случаях, когда нет доступа к большим машинам.

Другое использование нам кажется одинаково важным — обучение информатике в вузах и школах, так как Пролог идеально приспособлен для такого рода деятельности. Исследования, убедительно подтверждающие это, были проведены в Великобритании [Энналс 82].

ПРИЛОЖЕНИЕ II. КРАТКОЕ ИЗЛОЖЕНИЕ СИНТАКСИСА

Ниже приводится список всех бесконтекстных правил, расположенных в алфавитном порядке их левых частей; эти правила используются для определения формул, встречающихся в Прологе. Напомним соглашения:

(1) Знаком переписывания является ::=, причем левая часть правила не повторяется, если она совпадает с левой частью предыдущего.

(2) Терминалы являются символами и представляются изолированными символами, кроме символа пробела, который представляется словом «пробел».

(3) Нетерминалы являются последовательностями слов, ограниченных знаками ⟨и⟩. Мы предупреждаем читателя, что символы ⟨и⟩ встречаются также в качестве терминальных символов. В этом случае они появляются изолированно.

⟨*буква*⟩
  ::= ⟨***строчная***⟩
  ::= ⟨***заглавная***⟩
⟨***высказывание***⟩
  ::= ⟨***комментарий***⟩
  ::= ⟨***правило***⟩
⟨*действительное*⟩
  ::= ⟨*знак*⟩⟨***целое***⟩·⟨***последовательность цифр***⟩⟨***показатель***⟩
⟨*длинное слово*⟩

::= ⟨буква⟩⟨короткое слово⟩
::= ⟨буква⟩⟨длинное слово⟩

⟨заглавная⟩
::= A
::= B
. . . . .
::= Z

⟨знак⟩
::= +
::= −

⟨команда⟩
::= ⟨последовательность термов⟩;
::= ⟨последовательность термов⟩, ⟨система⟩;

⟨комментарий⟩
::= ⟨цепочка⟩

⟨константа⟩
::= ⟨идентификатор⟩
::= ⟨цепочка⟩
::= ⟨целое⟩
::= ⟨действительное⟩

⟨короткое слово⟩
::= ⟨буква⟩⟨последовательность цифр⟩
::= ⟨короткое слово⟩′

⟨неравенство⟩
::= ⟨терм⟩ ≠ ⟨терм⟩

⟨идентификатор⟩
::= ⟨длинное слово⟩
::= ⟨идентификатор⟩ — ⟨слово⟩

⟨паразит⟩
::= /
::= ⟨синтаксис неизвестен, но отличается от синтаксиса терма⟩

⟨переменная⟩
::= ⟨коротков слово⟩
::= ⟨переменная⟩ — ⟨слово⟩

⟨показатель⟩
::= ⟨пусто⟩
::= e ⟨целое⟩
::= e ⟨знак⟩⟨целое⟩

⟨последовательность символов⟩
::= ⟨пусто⟩
::= ⟨символ⟩⟨последовательность символов⟩

⟨последовательность термов⟩
::= ⟨пусто⟩
::= ⟨терм⟩⟨пробел⟩⟨последовательность термов⟩
::= ⟨паразит⟩⟨пробел⟩⟨последовательность термов⟩

⟨*последовательность цифр*⟩
::= ⟨*пусто*⟩
::= ⟨*цифра*⟩⟨*последовательность цифр*⟩
⟨*правило*⟩
::= ⟨*терм*⟩ → ⟨*последовательность термов*⟩;
::= ⟨*терм*⟩ → ⟨*последовательность термов*⟩, ⟨*система*⟩;
⟨*программа*⟩
::=;
::= ⟨*высказывание*⟩⟨*программа*⟩
⟨*простой терм*⟩
::= (⟨*терм*⟩)
::= ⟨*переменная*⟩
::= ⟨*константа*⟩
⟨*пусто*⟩
::=
⟨*символ*⟩
::= ⟨*специальный символ*⟩
::= ⟨*буква*⟩
::= ⟨*цифра*⟩
⟨*система*⟩
::= { }
::= {⟨*уравнения и неравенства*⟩}
⟨*слово*⟩
::= ⟨*короткое слово*⟩
::= ⟨*длинное слово*⟩
⟨*специальный символ*⟩
::= +
::= −
::= ′
::= .
::= ,
::= :
::= =
::= ≠
::= ″
::= /
::= (
::=)
::= ⟨
::= ⟩
::= {
::= }
::= ⟨*пробел*⟩
::= ⟨*любой другой доступный символ*⟩
⟨*строгий терм*⟩
::= ⟨*переменная*⟩

⟨*константа*⟩ ::= ⟨*константа*⟩
⟨*строгий терм*⟩, ⟨*строгий терм*⟩) ::= ((⟨*строгий терм*⟩, ⟨*строгий терм*⟩))
( ) ::= ( )
⟨*строгий терм*⟩, ⟨*строгий терм*⟩) ::= (⟨*строгий терм*⟩, ⟨*строгий терм*⟩)
⟨*строгий терм*⟩, ⟨*строгий терм*⟩, ⟨*строгий терм*⟩) ::= (⟨*строгий терм*⟩, ⟨*строгий терм*⟩, ⟨*строгий терм*⟩)
. . . . . .

⟨*строчная*⟩
a ::= *a*
b ::= *b*
. . . .
z ::= *z*

⟨*терм*⟩
⟨*простой терм*⟩ ::= ⟨*простой терм*⟩
⟨*простой терм*⟩, ⟨*терм*⟩ ::= ⟨*простой терм*⟩, ⟨*терм*⟩
( ) ::= ( )
⟨*терм*⟩) ::= (⟨*терм*⟩)
⟨*терм*⟩, ⟨*терм*⟩) ::= (⟨*терм*⟩, ⟨*терм*⟩)
⟨*терм*⟩, ⟨*терм*⟩, ⟨*терм*⟩) ::= (⟨*терм*⟩, ⟨*терм*⟩, ⟨*терм*⟩)
. . . . . .
⟨*идентификатор*⟩ (⟨*терм*⟩) ::= ⟨*идентификатор*⟩ (⟨*терм*⟩)
⟨*идентификатор*⟩ (⟨*терм*⟩, ⟨*терм*⟩) ::= ⟨*идентификатор*⟩ (⟨*терм*⟩, ⟨*терм*⟩)
⟨*идентификатор*⟩ (⟨*терм*⟩, ⟨*терм*⟩, ⟨*терм*⟩) ::= ⟨*идентификатор*⟩ (⟨*терм*⟩, ⟨*терм*⟩, ⟨*терм*⟩)

⟨*уравнение*⟩
⟨*терм*⟩ = ⟨*терм*⟩ ::= ⟨*терм*⟩ = ⟨*терм*⟩

⟨*уравнения и неравенства*⟩
⟨*уравнение*⟩ ::= ⟨*уравнение*⟩
⟨*неравенство*⟩ ::= ⟨*неравенство*⟩
⟨*уравнение*⟩, ⟨*уравнения и неравенства*⟩ ::= ⟨*уравнение*⟩, ⟨*уравнения и неравенства*⟩
⟨*неравенство*⟩, ⟨*уравнения и неравенства*⟩ ::= ⟨*неравенство*⟩, ⟨*уравнения и неравенства*⟩

⟨*цепочка*⟩
«⟨*последовательность символов*⟩» ::= «⟨*последовательность символов*⟩»

⟨*целое*⟩
⟨*цифра*⟩ ⟨*последовательность цифр*⟩ ::= ⟨*цифра*⟩ ⟨*последовательность цифр*⟩

⟨*цифра*⟩
0 ::= 0
1 ::= 1
. . . .
9 ::= 9

Замечание: символ двойная кавычка должен быть удвоен внутри цепочки.

Без изменения смысла любого написанного выражения можно

(1) вставлять пробелы в любое место, за исключением внутренности констант и переменных;

(2) удалять пробелы в любом месте, кроме внутренности цепочек и за исключением тех случаев, когда это приведет к порождению новых констант или переменных от склеивания старых.

ПРИЛОЖЕНИЕ III. ВСТРОЕННЫЕ ПРАВИЛА

Внимание: все вызовы встроенных правил пишутся строчными буквами.

Перед тем как исполниться, каждое встроенное правило проверяет тип своих аргументов. Для облегчения чтения принимаются следующие соглашения:

v → переменная

t → терм

i → идентификатор

c → цепочка

e → целое

П.3.1. Управление

● >/: Речь идет о "/", хорошо известном пользователям Пролога. Поэтому нужно отметить, что в данной версии «/» обязан присутствовать в том дизъюнкте, в котором он появляется. Его нельзя использовать динамически, подставляя вместо переменной.

● > ***блок***(*v*, *t*), ***конец-блока***(*t*): Речь идет о средстве для того, чтобы резко оборвать выполнение программы. Этот механизм используется главным образом для обработки ошибок, например:

блок(*u*,*pp*(*x*))...
pp(*x*) → ... *конец-блока*(*w*) ...;

блок выполняет нормально *pp*(*x*).

Если во время выполнения *pp*(*x*) встретится *конец-блока*(*u*), управление передается на непосредственно включающий блок, и в случае, когда *u* и *w* унифицируемы, выполнение блока продолжается детерминированным образом.

В противном случае управление передается на высший блок и т. д. Если этот процесс не останавливается, то выдается ошибка *НЕПРАВИЛЬНОЕ ОКОНЧАНИЕ КОНЦА-БЛОКА*.

```
команда →
   строка
   вывц("+")
   вв-сим'(k)
   блок(x, исполнить(k))
   проверить(x);
проверить(x) → свободная(x)/;
проверить(x) → целое(x)/ошибка(x);
проверить(выход) →/;
проверить(x) → конец-блока(x);
```

В этом примере, взятом из супервизора, можно видеть использование блоков. Здесь правило *команда* исполняет одну команду редактора. Правило *проверить* проверяет окончание этой команды, причем возможны четыре случая:

1. Нормальный выход из блока, команда выполнилась без ошибки.

2. Произошла ошибка в процессе выполнения команды либо во встроенном правиле, либо в системе. В этом случае Пролог выдает *конец-блока*(*е*), где *е* — это целое число, означающее номер ошибкн.

3. Чтобы завершить команду редактора *заменить*, порождается *конец-блока*(*выход*), что позволяет выйти из бесконечного цикла.

4. Этот последний случай соответствует правилу *конец-блока*, появившемуся в самой команде (напримсер, в комаиде исполиить); управление передается на включающий блок.

П.3.2. Обработка высказываний

Обработка высказываний производится, вообще говоря, редактором дизъюиктов, однако встроенные правила, которые в этом участвуют, могут употребляться и независимо. Напомним, что существует текущий указатель высказываний (правил или комментариев), и большинство нижеследующих функций ссылаются на этот указатель. Вся работа происходит в текущем мире. Более полное описание этих команд можно найти в [ван Канегем 82].

- > ***низ***: Указатель устанавливается на самый конец мира.
- > ***опустить***(*е*): Указатель опускается вниз на *е* высказываний или на конец мира, если *е* превышает число оставшихся высказываний.
- > ***верх***: Указатель устанавливается на начало мира.
- > ***поднять***(*е*): Указатель поднимается на *е* высказываний или на начало мира, если *е* превышает число оставшихся высказываний.
- > ***голова***(*i*): Указатель устанавлнвается на первом правиле, имеющем идентификатор *i*.
- > ***вставить***: Высказывания, считанные с активного устройства, вставляются перед указателем. Если произошла ошибка, происходит считывание до символа «;» и выполнение заканчивается. Выполнение ***вставить*** заканчивается нормально, когда считывается пустое высказывание, т. е. последовательность «;;». Если происходит EOF [1]) во время исполнения «***вставить***», исполнение заканчивается нормально с сообщением EOF *ВО ВХОДНОМ ФАЙЛЕ*.
- > ***выдать***(*е*): На активное устройство вывода выдается *е* высказываний, начиная с положения указателя. Это правило ие меняет положения указателя. Выдача стандартная: если правило помещается в одной строке, она так и выдается; в противном случае выдается по одному терму в каждой строке.
- > ***удалить***(*е*): Удаляются *е* высказываний, начиная с позицин указателя. Он устанавливается на первое следующее за удаленным высказывание.

П.3.3. Ввод

П.3.3.1. Введение

В системе есть 4 устройства ввода/вывода:

1. буфер ввода и вывода;
2. консоль;
3. входной файл на диске;
4. выходной файл на диске.

В каждый момент имеется одно активное устройство ввода и одно активное устройство вывода. Весь обмен происходит с помощью этих двух активных устройств.

[1]) EOF — конец файла (от англ. End Of File). — *Прим. перев.*

Считывание происходит построчно. При обмене *буфер* ⟨—⟩ *другое устройство* положение указателя символа сохраняется. В случае обмена *диск* ⟨—⟩ *консоль* сохраняется лишь положение указателя строки; т. е. все, что остается на строке, теряется.

Устройство представляет собой одно из устройств ввода/вывода: буфер, консоль или *xxx.xxx*, т. е. обозначение имени файла во включающей системе. Для Apple II речь идет об имени файла без расширения *text*.

П.3.3.2. Несколько замечаний относительно БУФЕРА

Устройство *буфер* соответствует набору файлов виртуальной памяти. Эти «буферные» файлы служат для всех манипуляций ввода/вывода, таких как добавление, и т. д.

Поскольку в каждый момент *буфер* можно использовать одновременно для ввода и для вывода, необходимо внимательно следить за порядком следования запросов на ввод/вывод. Каждый раз при переходе от ввода к выводу требуется сменить строку (например, командой *строка*).

Замечание. Некоторые встроенные правила, например *вывод*, производят возврат на строку.

Несмотря на все эти объяснения, использование *буфера* остается делом весьма деликатным, и поэтому желательно дополнительное ознакомление с супервизором ввода/вывода.

Буфер используется в следующих командах: корректировка, добавление, уплотнение, выдача.

Ниже приводится программа, использующая буфер для конкатенации двух цепочек.

```
конк(x, y, z) → буфер-новый(конк'(x, y, z));
конк'(x, y, z) →
  вывод("буфер")
  вывц("")
  вывц(x)
  вывц(y)
  вывц("")
  строка
  вывод("консоль")
  ввод("буфер")
  ввод-цепочки(z)
  ввод("консоль");
```

П.3.3.3. Встроенные правила для ввода

● > ***ввод***(*c*): Устройство с именем *c* становится активным устройством ввода.

● > ***буфер-новый***(*t*): Вызывает детерминированное исполнение *t*, создавая новый буферный файл. Когда *t* будет стерто, либо его нельзя стереть, созданный файл уничтожается и система возвращается к старому буферному файлу.

Нижеследующие встроенные правила считывают с активного устройства ввода. Под «символом» понимается цепочка, состоящая из одного символа.

● > ***вв-сим***(*с*): С активного устройства считывается символ в *с*.

● > ***вв-сим'***(*с*): С активного устройства считывается следующий непустой символ в *с*.

● > ***сим-след***(*с*): Переход к следующему символу без реального чтения.

● > ***сим-след'***(*с*): Переход к следующему непустому символу без реального чтения. В реализации положение указателя символа меняется после исполнения *сим-след'* и устанавливается на первый не являющийся пробелом символ после последнего считанного.

● > ***вв***(*t*): Считывается по возможности самый большой терм. Синтаксис терма устроен так, что появление одного символа, не принадлежавшего терму, вызывает прекращение считывания. Внимание: этот символ не читается.

```
> вв(t)вв-сим(";");
это.есть.терм;
t = это.есть.терм
```

● > ***вв-целое***(*v*): Делается попытка считать целое число с текущего устройства. Если читаемый объект не обладает синтаксисом числа, то считывания не происходит. В противном случае последний считанный символ — это последний символ целого числа.

● > ***вв-ид***(*v*): Считывается идентификатор способом, аналогичным считыванию *вв-целое*.

● > ***вв-цепочки***(*v*): Считывается цепочка способом, аналогичным предыдущему случаю.

● > ***вв-пред***(*t*): Считывается предложение, оканчивающееся на ".", "?", " "; это предложение затем преобразуется в последовательность атомов, оканчивающуюся на nil. Каждое слово, т. е. каждая последовательность букв, преобразуется в идентификатор *сл-xxx*, где *xxx* есть считанное слово. Каждое целое число, т. е. последовательность цифр, преобразуется в соответствующее число. Всякий другой символ заменяется на соответствующую цепочку.

```
> вв-пред(x);
это есть предложение.
x = сл-это.сл-есть.сл-предложение.".".
> вв-пред(x);
toto := 52345;.
x = сл-toto.":=".52345.";".".".
```

● > ***конец-строки***(*с*): Преобразует возврат каретки в символ *с*. По умолчанию конец строки является пробелом. В редакторе дизъюнктов конец строки есть символ ",".

П.3.4. Вывод

П.3.4.1. Обычный вывод

Вывод происходит на активное выводное устройство.

● > ***вывод***(*с*): Устройство с именем *с* становится активным выводным устройством.

● > ***закрыть-вывод***: Если активным устройством является диск, то происходит закрытие файла на диске и этот файл сохраняется. Это правило выполняется автоматически при выходе из редактора или из Пролога.

● > ***выв***(t): Терм t выводится на активное устройство. При этом атомы не разбиваются — если атом не помещается в строку, он обрезается. Если терм не помещается в строку, то выдается конец строки и происходит переход на следующую строку с отступом в три пробела. Учитывая эти ограничения, всякий терм, выданный посредством правила ***выв***, может быть прочитан правилом ***вв***.

Правило ***выв*** позволяет выводить и бесконечные деревья. При этом $*i$ изображает дерево, равное дереву-предку, находящемуся на i уровней выше. Правило ***выв*** не оптимизирует дерево, т. е. не ищет минимальное представление дерева. Ниже приводится несколько примеров исполнения этого правила.

```
> рав(x,ff(x));
x = ff(*1)
> рав(x,ff(x,x));
x = ff(*1,*1)
> рав(x,x.x.x.x);
x = *1,*2.*3.*4
```

● > ***вывц***(c): На текущее устройство выводится цепочка c без кавычек. При этом, как и в предыдущих случаях, сохраняется требование: нельзя вывести цепочку с длиной большей, чем длина строки.

● > ***строка***: Происходит переход на следующую строку.

● > ***страница***: Происходит переход на следующую страницу. В случае устройства «консоль» экран стирается и курсор устанавливается в верхнем левом углу.

● > ***дл-строки***(e): Длина строки становится равной e. По умолчанию длина строки равна 72 символам. В случае Apple иногда полезно делать строку в 40 символов. Максимальная длина строки — 128 символов.

● > ***курс-ху***($e1,e2$): Устанавливает курсор в позицию $e1,e2$ на экране, аналогично оператору *gotoxy* Паскаля. Точка (0, 0) соответствует верхнему левому углу экрана. Для $e1$ и $e2$ имеет место

$$0 \leqslant e_1 \leqslant 79 \text{ и } 0 \leqslant e_2 \leqslant 23.$$

● > ***поз***(e): Текущий указатель символа устанавливается в позицию e выдаваемой строки. Речь идет о своего рода табуляции. Текущий указатель символа нельзя двигать назад.

● > ***эхо***: Позволяет получить на консоли эхо того, что читается или пишется на дисковый файл или буфер. Внимание: если выводным устройством является дисковый файл, эхо не позволяет выводить то, что набирается на клавиатуре.

● > ***заглушить***: Аннулируется правило *эхо*.

П.3.4.2. Вывод на принтер

Принтер рассматривается как устройство, которое копирует все, что появляется на экране при вводе или выводе.

Можно также копировать содержимое экрана в файл. Для этого достаточно определить этот файл как «принтер.текст» на диске с Прологом и в этом случае копирование пойдет на диск, который заменит, в некотором смысле, принтер.

Внимание: При выборе этой возможности нельзя будет использовать этот файл в качестве выводного устройства в Прологе.

- > ***бумага***: принтер печатает все, что появляется на экране.
- > ***без-бумаги***: Аннулируется действие правила *бумага*.

П.3.5. Проверки

Речь идет о выяснении типа атома.

- > ***свободна***(*t*): Верно, если *t* — свободная переменная (а также когда *t* — свободная замороженная переменная).
- > ***связана***(*t*): Верно, если *t* — не является свободной переменной.
- > ***целое***(*t*): Верно, если *t* — целое число.
- > ***ид***(*t*): Верно, если *t* — идентификатор.
- > ***цепочка***(*t*): Верно, если *t* — цепочка.

П.3.6. Арифметика

Вся арифметика делается посредством встроенного правила *знач*, которое вычисляет значение арифметического выражения. Целые числа расположены в интервале от 0 до 2 097 151 = $2^{21} - 1$. На данный момент в нашей реализации имеются только положительные числа, но в скором времени появятся действительные и отрицательные числа.

- > ***знач***(*t*1,*t*2): Вычисляется *t*1, результат должен быть атомом, и должно выполняться *значение* (*t*1) = *t*2.

Вычисляемый терм составлен из вычисляемых функций, которые применяются только к известным термам. Значение целого или цепочки совпадает с ними самими. В случае идентификатора дело обстоит сложнее. Если вычисляемая функция имеет неправильное число аргументов или какой-либо аргумент имеет неправильный тип, *знач* выдает ошибку. Разумеется, это — восстанавливаемая ошибка.

Идентификатор может обозначать:

1. Набор дизъюнктов. В этом случае значением является сам идентификатор. Например:

toto → ...;

значение(*toto*) = *toto*

2. Вычисляемую функцию, задаваемую внутренним примитивом супервизора. Пример:

значение(*слож*)-*ошибка*

3. Атом. В этом случае значение определяется вычисляемым предикатом присвоить. Например:

присвоить (*toto*,*3*);

значение (*toto*) = 3

Имеется сколько угодно простых переменных (в смысле классических языков программирования).

К настоящему времени имеются следующие вычисляемые функции:

1. ***слож***(*t*1,*t*2): *значение*(***слож***(*t*1,*t*2)) = *значение*(*t*1) + *значение*(*t*2)
2. ***выч***(*t*1,*t*2): *значение*(***выч***(*t*1,*t*2)) = *значение*(*t*1) — *значение*(*t*2)
3. ***умнож***(*t*1,*t*2): *значение*(***умнож***(*t*1,*t*2)) = *значение*(*t*1)**значение*(*t*2)
4. ***дел***(*t*1,*t*2): *значение*(***дел***(*t*1,*t*2)) = *значение*(*t*1)/*значение*(*t*2)
5. ***мод***(*t*1,*t*2): *значение*(***мод***(*t*1,*t*2)) = *значение*(*t*1) *по модулю значение*(*t*2)

6. *меньше*(*t*1,*t*2): *значение*(*меньше*(*t*1,*t*2)) = *если значение*(*t*1) *значение*(*t*2) *то* 1; *иначе* 0
7. *рав*(*t*1,*t*2): *значение*(*рав*(*t*1,*t*2)) = *если значение*(*t*1) = *значение*(*t*2), *то* 1 *иначе* 0
8. *если*(*t*,*t*1,*t*2): *значение*(*если*(*t*,*t*1,*t*2)) = *если значение*(*t*) = 1, *то значение*(*t*1); *иначе значение*(*t*2)

Как правило, аргументы вычисляемой функции имеют тип целое. Аргументами функции *меньше* могут быть и другие атомы. Для целых чисел *меньше* означает отношение меньше на числах. Для цепочек — это отношение лексикографического порядка, для идентификаторов — отношение лексикографического порядка на соответствующих цепочках.

Для булевых функций мы приняли то же соглашение, что и в Бейсике: *истина* = 1, *ложь* = 0. Это позволяет легко выполнять *и* и *или* на булевых выражениях.

● > ***присвоить***(*i*,*t*): Значение *t* присваивается идентификатору *i*. При этом должны выполняться следующие требования:

1. Значение терма *t* должно иметь тип атома.
2. Идентификатор не должен обозначать правило или вычисляемую функцию.
3. Если терм *t* — цепочка, то он должен быть определен в том же подмире, что и идентификатор *i*. В частности, нельзя присвоить идентификатору цепочку, считанную с клавиатуры.

Разумеется, это присваивание сохраняется после бектрекинга. Рекомендуется не злоупотреблять этим правилом, поскольку Пролог — это не Фортран.

● > ***опр-массив***(*i*,*e*): Позволяет определить массив *i* длиной *e*. Доступ к его элементам или присваивание осуществляется так же, как в классических массивах. Рекомендуется использовать только в случае крайней необходимости.

```
"управление стеком"
ув(i) → знач(слож(i,1),x)присвоить(i,x);
ум(i) → знач(выч(i,1),x)присвоить(i,x);
новый-стек → присвоить(счетчик,0)опр-массив(стек,100);
положить(v) →
   ув(счетчик)
   знач(меньше(счетчик,100),1)
   /
   знач(счетчик,p)присвоить(стек(p),v);
положить(v) — вывц("переполнение")строка тупик;
взять →
   знач(рав(счетчик,0),0)
   /
   знач(стек(счетчик),v)ум(счетчик)
взять → вывц("переполнение")строка тупик;
> новый-стек;
> положить(12345);
> положить(12346);
> взять(x)
x = 12346
> взять(x)
x = 12345
> взять(x);
переполнение
```

П.3.7. Миры

Речь идет об организации пространства высказываний в некоторую древесную иерархию. При этом из данного мира доступны все идентификаторы миров-предков. Таким образом, параллельные миры оказываются защищенными друг от друга. На поставляемой виртуальной памяти на диске текущий мир имеет имя «простой» и подчиняется одному предку — миру *начало*. Этот мир имеет доступ к правилам, определенным в супервизоре.

Предыдущее определение является статическим, и оно годится для большинства случаев. Однако, когда миры начинают развиваться во времени, все становится более сложным. Это видно на следующем примере.

```
> toto;
   (toto определяется в "простом" мире)
> спуститься("мир1")
(переходим в мир1)
> вставить;
toto → вывц("toto");
(идентификатор toto взят из мира "простой", правило toto определено в мире1, но доступно в "простом")
lulu → вывц("lulu");
(правило lulu доступно только в мире1)
> подняться("простой");
(возвращаемся в "простой" мир)
> lulu;
(этот идентификатор отличен от lulu, определенного в предыдущем мире)
```

● > ***спуститься***(*с*): Текущим миром становится мир с именем *с*.

1. Если у текущего мира существует сын с именем *с*, этот мир становится текущим.

2. Если не существует такого мира, то у текущего мира создается сын с именем *с* и мы переходим в этот мир, который и становится текущим.

Мир ???? защищен — в него нельзя спуститься.

● > ***подняться***: Переходим в мир-отец текущего мира. Нельзя подняться выше мира *начало*.

● > ***удалить-мир***(*с*): Удаляется мир-сын *с* текущего мира. Отметим, что это единственный способ эффективного восстановления памяти. При удалении мира восстанавливаются значения идентификаторов, с помощью которых осуществлялся доступ к дизъюнктам удаляемого мира. Не проверяется, существовал ли доступ к удаляемому миру в стеках системы.

Внимание: это правило можно применять только к терминальному миру, т. е. миру без сыновей. Если необходимо уничтожить некоторое множество миров, следует использовать команду *очистить*.

● > ***мир***(*t*): Выполняется, если *t* — имя текущего мира. Это имя есть атом типа цепочка.

● > ***под-мир***(*t*): Выполняется, если *t* — список под-миров текущего мира. Например, если мы находимся в мире «начало»:

```
> под-мир(x)
x = "????"."простой".nil
```

Отметим, что команды *новый*, *уплотнить*, *очистить*, *состояние* также имеют дело с мирами.

П.3.8. Координация вычислений

Речь идет о средствах, позволяющих при определенных условиях отложить вычисления. Этот новый для Пролога механизм легок для понимания, но требует большой осторожности при использовании.

● > ***заморозить***(v, t): Цель этого правила — отложить вычисление t до тех пор, пока неизвестно v. Более точно:

1. Если v свободна, правило *заморозить* стирается и вычисление t откладывается до момента, когда v станет связанной.
2. Если v связанна, то t нормально стирается.

Вот знаменитая программа *одинаковые-листья*, которая проверяет, обладают ли два дерева одинаковыми терминальными листьями.

Привлекательность синхронизации этого процесса заключается в возможности установления того, что два дерева не имеют одинаковых листьев, без полного просмотра первого дерева.

```
одинаковые-листья(a,b) →
  листья(a,u)
  листья(b,u)
  список(u);
листья(a,u) — заморозить(u,листья'(a,u)):
листья'(a,a.nil) → терминал(a);
листья'(a.l,a.u) — терминал(a)листья(l,u);
листья'((a.b).l,u) — листья'(a.b.l,u);
список(nil) →;
список(a.u) → список(u);
терминал(a) → ид(a);
```

● > ***разл***$(t1, t2)$: Проверяется, что $t1 \neq t2$. Если $t1$ и $t2$ не содержат свободных переменных, то *разл* сразу вычисляется. Если же вычисление *разл* требует присваивания для свободных переменных в $t1$ или $t2$, то оно откладывается до момента, когда какая-либо из этих переменных получит значение.

Это правило является очень важным и многократно используется в программах. В частности, оно позволяет во многих случаях избавиться от «/».

В следующем примере строятся перестановки посредством описания того, что все цифры в перестановке должны быть различны. Отметим, что правила *различные* и *вне* являются очень употребительными.

```
вне(x,nil) →;
вне(x,a.l) → разл(x,a)вне(x,l);
различные(nil) →;
различные(x.l) → вне(x,l)различные(l);
перестановка(x) →
  рав(x,x1.x2.x3.nil)
  различные(x)
  список-цифр(x);
список-цифр(nil) →;
список-цифр(x.l) →
  цифра(x)
  список-цифр(l);
цифра(1) →;
цифра(2) →;
цифра(3) →;
```

П.3.9. Разное

● > ***арг***(*e*, *t1*,*t2*): Этот примитив вычисляет составляющую с номером *e* в цепочке, списке или *n*-ке.

1. Когда *t1* — цепочка:
   если e = 0, то t2 := *длина(t1)*
   если e ≠ 0, то t2 := *«символ с номером e в t1»*
2. Когда *t1* — последовательность:
   если e = 1, то t2 := *голова(t1)*
   если e = 2, то t2 := *хвост(t1)*
3. Когда *t1* — *n*-ка:
   если e = 0, то t2 := *число аргументов t1*
   если e ≠ 0, то t2 := *аргумент с номером e в t1*

● > ***до-свидания***: Заканчивает исполнение Пролога и сохраняет виртуальную память. Выходить из Пролога нужно всегда с помощью *до-свидания*.

● > ***петля***: Сообщает Прологу, что нужно унифицировать бесконечные деревья. Эта информация хранится в виртуальной памяти. По умолчанию Пролог считает, что бесконечных деревьев нет.

● > ***без-петли***: Стандартная ситуация — нормальная унификация конечных деревьев. Пролог не рискует делать петли, если нет обработки бесконечных деревьев.

● > ***трассировка***: Речь идет о вспомогательной функции, используемой при отладке. Когда это правило активируется, каждый вызов любого правила распечатывается вместе со своими аргументами.

● > ***без трассировки***: Деактивирует предыдущее правило.

● > ***бум***(*i*,*c*): Сопоставляет идентификатору *i* цепочку *c*, которая его представляет, и наоборот. Если *i* — переменная, создается новый идентификатор. Это правило не работает, если оба аргумента являются переменными.

● > ***ном-сам***(*c*,*e*): Сопоставляет цепочке из одного символа его код ASCII, и наоборот.

ПРИЛОЖЕНИЕ IV. ДОКАЗАТЕЛЬСТВА СВОЙСТВ

Мы собрали здесь все важные доказательства, которые не были приведены в основном тексте.

П.4.1. Системы в приведенной форме

Свойство разрешимости: Всякая система в приведенной форме является разрешимой.

Доказательство. Пусть S — система в приведенной форме. Если S содержит одно уравнение вида $v = w$, где v и w — переменные, это уравнение удаляется и каждое вхождение v заменяется на w. Повторяя эту операцию столько раз, сколько потребуется, мы придем к системе T, которая всегда находится в приведенной форме, не содержит больше уравнений вида $v = w$, и, кроме того, из ее разрешимости вытекает разрешимость S. Достаточно, следовательно, показать разрешимость T.

Пусть $T = E \cup I$, где E — множество уравнений и I — множество неравенств. Пусть $\{x_1, \ldots, x_m\}$ — множество переменных, составляющих левые части уравнений из E, и пусть $\{y_1, \ldots, y_n\}$ — множество всех других переменных, встречающихся в T. Рассмотрим систему

$$E \cup \{y_1 = k_1, \ldots, y_n = k_n\},$$

где k_i — константы, имеющие попарно различные значения и отличающиеся от значений констант, встречающихся в T. По характеристическому свойству 3 эта система имеет решение вида

$$X = \{x_1 := r_1, \ldots, x_m := r_m\} \cup \{y_1 := {}'k_1', \ldots, y_n := {}'k_n'\}.$$

Древесное означивание X является решением E; остается показать, что оно является также и решением I.

Будем доказывать от противного: предположим, что X не является решением I. Тогда существует такое неравенство в I

$$\langle u_1, \ldots, u_p\rangle \neq \langle t_1, \ldots, t_p\rangle,$$

что деревья $\langle u_1, \ldots, u_p\rangle(X)$ и $\langle t_1, \ldots, t_p\rangle(X)$ равны и, следовательно, по характеристическому свойству однозначной декомпозиции

$$u_1(X) = t_1(X).$$

Поскольку T находится в приведенной форме и не содержит уравнений вида $v = w$, возможны всего три случая:

(1) u_1 и t_1 — различные переменные из множества $\{y_1, \ldots, y_n\}$. Так как все k_i имеют попарно различные значения, $u_1(X)$ отличается от $t_1(X)$, и мы приходим к противоречию с равенством $u_1(X) = t_1(X)$;

(2) u_2 — переменная из множества $\{y_1, \ldots, y_n\}$, t_1 — переменная из $\{x_1, \ldots, x_m\}$, и в E существует уравнение вида $t_1 = t_1'$, где, разумеется, t' не является переменной. Поскольку k_i имеют значения, не совпадающие со значениями констант, встречающихся в T, значение $u_1(X)$ отличается от значения $t_1'(X)$ и, следовательно, от значения $t_1(X)$. Мы снова приходим к противоречию с равенством $u_1(X) = t_1(X)$;

(3) u_1 — переменная из $\{y_1, \ldots, y_n\}$, и t_1 не является переменной. Поскольку k_i имеют значения, не совпадающие со значениями констант, встречающихся в T, получаем, что $u_1(X)$ отличается от $t_1(X)$, а это снова приводит к противоречию с равенством $u_1(X) = t_1(X)$.

П.4.2. Эквивалентные системы

Эквивалентность 2: Пусть S и T — системы в приведенной форме, имеющие вид

$$S = \{x_1 = t_1, \ldots, x_n = t_n\}, \quad T = \{x_1 = t_1', \ldots, x_n = t_n'\}.$$

Если всякое древесное решение S является древесным решением T, то S и T эквивалентны.

Доказательство. Пусть $\{y_1, \ldots, y_m\}$ — переменные, отличные от x_i, которые встречаются в $S \cup T$. Чтобы доказать эквивалентность S и T, достаточно рассмотреть древесное решение системы T вида

$$X = \{x_1 := a_1, \ldots, x_n := a_n, \ y_1 := b_1, \ldots, y_m := b_m\}$$

и показать, что оно является также решением S. Пусть

$$\{b_1, \ldots, b_m, c_1, c_2, \ldots\}$$

есть множество поддеревьев множества $\{b_1, \ldots, b_m\}$. По свойству связанной системы из разд. 2.4, существует такая система E вида

$$E = \{y_1 = p_1, \ldots, y_m = p_m, \ z_1 = q_1, \ z_2 = q_2, \ldots\},$$

для которой означивание

$$\{y_1 := b_1, \ldots, y_m := b_m, z_1 := c_1, z_2 := c_2, \ldots\}$$

является древесным решением, причем p_i и q_i не являются переменными и в E не содержится никаких других переменных, кроме y_i и z_i. Легко преобразовать $S \cup E$ в эквивалентную систему, к которой применимо 3-е характеристическое свойство. То же верно и относительно системы $T \cup E$. Отсюда следует, что каждая из систем $S \cup E$ и $T \cup E$ имеет единственное древесное решение вида

$$\{x_1 := a'_1, \ldots, x_n := a'_n, y_1 := b'_1, \ldots, y_m := b'_m, z_1 := c'_1, z_2 := c'_2, \ldots\}.$$

Однако, поскольку всякое решение системы $S \cup E$ есть решение системы $T \cup E$, эти решения должны совпадать. Решение системы $T \cup E$

$$\{x_1 := a_1, \ldots, x_n := a_n, y_1 := b_1, \ldots, y_m = p_m, z_1 := c_1,$$
$$z_2 := c_2, \ldots\}$$

является, следовательно, решением системы $S \cup E$. Отсюда следует, что означивание

$$X = \{x_1 := a_1, \ldots, x_n := a_n, y_1 := b_1, \ldots, y_m := b_m\}$$

является также и решением S, что и требовалось доказать.

П.4.3. Приведение уравнений

Свойство остановки: Если S_1 — конечная система уравнений, то не существует бесконечной последовательности $S_1, S_2, S_3, \ldots$ систем, в которой S_{i+1} получается применением к S_i одной из трансформаций T_1, T_2, T_3, T_4 и T_5.

Доказательство. Покажем вначале, что такая последовательность конечна, если не рассматривать трансформацию разложения. Рассмотрим начальную систему S_1. Пусть n — общее число уравнений вида $x = x$ и $x = y$, а m — общее число уравнений вида $x = t$ и $t = x$. Если исключить разложение, то число поглощений и исключений не может превышать n, а число перестановок и противопоставлений не может превышать m.

Рассмотрим теперь отображение f, которое сопоставляет неотрицательные целые числа $f(S)$ и $f(t)$ каждой системе и каждому терму и определяется следующими правилами:

$f(\{s_1 = t_1, \ldots, s_n = t_n\}) = f(\{s_1 = t_1\}) + \ldots + f(\{s_n = t_n\})$,

$f(\{s_i = t_i\}) = k^r$, где $r = \max(f(s_i), f(t_i))$,

$f(t)$ — высота терма t (была определена в T_5),

где $k > 2$ и превышает максимальную длину l конструкции $\langle e_1, \ldots, e_l\rangle$, встречающейся в S_1.

(1) $f(S_i) > f(S_{i+1})$ в случае трансформации разложения, ибо

$f(\{\langle s_1, \ldots, s_n\rangle = \langle t_1, \ldots, t_n\rangle\}) > k^{q+1}$, где q — наименьшая высота s_i и t_i;

$k^{q+1} > nk^q \geqslant f(\{s_1 = t_1\}) + \ldots + f(\{s_n = t_n\}) \geqslant f(\{\langle s_1 = t_1\rangle, \ldots, \langle s_n = t_n\rangle\})$.

Таким же образом

$$f(\{s_1 . s_2 = t_1 . t_2\}) > f(\{s_1 = t_1, s_2 = t_2\}).$$

(2) Очевидно, что $f(S_i) \geqslant f(S_{i+1})$ в случае всех других трансформаций.

Из первой части доказательства следует, что существование бесконечной последовательности систем S_i предполагает применение бесконечного числа разложений. Неравенства из второй части доказательства делают это невозможным, так как $f(S_i)$ стала бы отрицательной, что противоречит ее определению. Доказательство закончено.

Свойство сохранения: Пусть дана разрешимая система S_1 без неравенств, имеющая вид $S_1 = E_1 \cup F_1$, где E_1 находится в приведенной форме:

$$E_1 = \{x_1 = t_1, \ldots, x_n = t_n\},$$

причем каждая переменная x_i, для которой t_i является переменной, имеет ровно одно вхождение в систему $E_1 \cup F_1$.

Если применить основной алгоритм приведения к S_1, мы получим приведенную систему S_2 вида $S_2 = E_2 \cup F_2$, где E_2 и F_2 не пересекаются и E_2 имеет вид

$$E_2 = \{x_1 = t'_1, \ldots, x_n = t'_n\}.$$

При этом система $E_1 \cup F_2$ находится в приведенной форме и эквивалентна исходной системе $E_1 \cup F_1$.

Доказательство. Рассмотрим произвольное уравнение $x_i = t_i$ системы E_1. Если t_i — переменная, то никакая трансформация не удалит единственное вхождение x_i в процессе приведения S_1, и, следовательно, в S_2 будет входить уравнение вида $x_i = t'_i$. Если t_i не является переменной, то в S_2 всегда будет входить уравнение вида $x_i = t'_i$, так как иначе существовало бы по крайней мере одно означивание $\{x_i := r\}$, которое было бы решением S_2, не являясь решением эквивалентной системы S_1. Система S_2 имеет, следовательно, нужный нам вид $S_2 = E_2 \cup F_2$, где $E_2 = \{x_1 = t'_1, \ldots, x_n = t'_n\}$.

Система $E_1 \cup F_2$ не может содержать кольцо переменных, ибо, с одной стороны, F_2 его не содержит, поскольку является приведенной, а, с другой стороны, каждый левый член x_i уравнения из E_1, у которого правый член t_i является переменной, имеет ровно одно вхождение в $E_1 \cup F_2$. Следовательно, $E_1 \cup F_2$ находится в приведенной форме.

Левые члены уравнений двух систем $E_2 \cup F_1$ и $E_1 \cup F_1$ совпадают. Всякое решение системы $E_1 \cup F_1$ является решением системы $E_2 \cup F_2$ и, следовательно, решением системы $E_1 \cup F_2$. По свойству эквивалентности 2 системы $E_1 \cup F_2$ и $E_1 \cup F_1$ являются эквивалентными, что и требовалось доказать.

П.4.4. Двойное определение

Свойство двойного определения: Пусть $A \xrightarrow{t}$ — множество деревьев, для каждого из которых существует такое t, что $r \Rightarrow$ *пусто*. Тогда A есть наименьшее множество деревьев, которые удовлетворяют логическим импликациям, сопоставленным частным правилам.

Доказательство. Первая часть. Покажем, что A удовлетворяет логическим импликациям, сопоставленным частным правилам

$r_0 \Rightarrow r_1 \ldots r_n$.

Если $n = 0$, то

$r_0 \overset{1}{\Rightarrow}$ *пусто* и, следовательно, r_0 принадлежит A. Если $n \neq 0$, тогда

$r_i \in A, \ldots, r_n \in A$

влечет за собой

$r_1 \overset{k_1}{\Rightarrow}$ *пусто*, ..., $r_n \overset{k_n}{\Rightarrow}$ *пусто*,

что влечет за собой по принципу независимости стираний

$r_1 \ldots r_n \overset{k}{\Rightarrow}$ *пусто*, где $k = k_1 + \ldots + k_n$,

откуда $r_0 \in A$, что и требовалось доказать.

Доказательство. Вторая часть. Покажем, что A входит в любое множество E деревьев, которые удовлетворяют логическим импликациям, т. е. для любого i и любого дерева r

$r \overset{i}{\Rightarrow}$ *пусто* влечет за собой $r \in E$.

Это утверждение верно для $i = 1$, ибо из $r \overset{1}{\Rightarrow}$ *пусто* следует, что существует частное правило $r \Rightarrow$ *пусто*, откуда $r \in E$.

Предположим, что это утверждение верно для $j < i$, и покажем, что оно верно для i: из $r \overset{i}{\Rightarrow}$ *пусто* и $i > 1$ следует, что существует частное правило $r = r_1 \ldots r_n$ и $r_1 \ldots r_n \overset{i-1}{\Rightarrow}$ *пусто*. Отсюда по принципу независимости имеем частное правило $r = r_1 \ldots r_n$ и $r_1 \overset{k_1}{\Rightarrow}$ *пусто*, ..., $r_n \overset{k_n}{\Rightarrow}$ *пусто*, где $k_1 < i, \ldots, k_n < i$, что дает по принципу индукции частное правило $r \Rightarrow r_1 \ldots r_n$ и $r_1 \subset E_1, \ldots, r_n \in E$, откуда следует $r \in E$, что и требовалось доказать.

П.4.5. Проблема шаблона

Принцип обобщения: Пусть i — неотрицательное целое число, $t_1, \ldots, t_n$ — последовательность термов (возможно, пустая), S — система и X — древесное означивание переменных, содержащихся в $(t_1, \ldots, t_n, S)$. Тогда X является древесным решением системы S и $t_1(X) \ldots t_n(X) \overset{i}{\Rightarrow}$ *пусто* тогда и только тогда, когда существует такая система, что $(t_1 \ldots t_n, S) \overset{i}{\rightarrow}$ (*пусто*, T) и X — древесное решение T.

Доказательство. Первая часть. Покажем, что принцип обобщения верен для $i = 0$:

$t_1(X) \ldots t_n(X) \overset{0}{\Rightarrow}$ *пусто* и X — древесное решение S; тогда и только тогда, когда $t_1 \ldots t_n \Rightarrow$ *пусто* и X — древесное решение S; тогда и только

тогда, когда: существует такая система T, что $(t_1 \dots t_n, S) \overset{0}{\Rightarrow}$ (*пусто*, T) и X — древесное решение S, что и требовалось доказать.

Доказательство. Вторая часть. Предположим, что принцип обобщения верен для i, и покажем, что он верен для $i+1$: $t_1(X) \dots t_n(X) \overset{i+1}{\Rightarrow}$ *пусто* и X — древесное решение S; тогда и только тогда, когда по определению $\overset{j}{\Rightarrow}$ существует частное правило $r_0 \Rightarrow r_1 \dots r_m$, причем $t_1(X) = r_0$, $r_1 \dots r_m t_2(X) \dots t_n(X) \overset{i}{\Rightarrow}$ *пусто* и X — решение S;
тогда и только тогда, когда по определению частного правила существует правило $s_0 \to s_1 \dots s_m$, U с переменными, переименованными таким образом, чтобы не совпадать ни с одной из тех, которые встречаются в $(t_1 \dots t_n, S)$, и существует такое древесное означивание Y, что:
в Y входят только переменные из $(s_0 \dots s_n, U)$,
Y — древесное решение U,
$t_1(X) = s_0(Y)$,
$s_1(Y) \dots s_m(Y)\, t_2(X) \dots t_n(X) \overset{i}{\Rightarrow}$ *пусто* и X — решение S;
тогда и только тогда, когда, учитывая переменные, входящие в X и Y, существует правило $s_0 \to s_1 \dots s_m$, U
с переменными, переименованными таким образом, чтобы не совпадать ни с одной из $(t_1 \dots t_n, S)$,
и существует такое древесное означивание Y, что:
в Y входят только переменные из $(s_0 \dots s_m, U)$;
$X \cup Y$ — древесное решение U,
$t_1(X \cup Y) = s_0(X \cup Y)$,
$s_1(Y) \dots s_m(Y)\, t_2(X) \dots t_n(X) \overset{i}{\Rightarrow}$ *пусто* и
$X \cup Y$ — решение S;
тогда и только тогда, когда:
существует правило $s_0 \to s_1 \dots s_m$, U с переменными, переименованными таким образом, чтобы не совпадать ни с одной из $(t_1 \dots t_n, S)$, и существует такое древесное означивание Y, что в Y входят только переменные из $(s_0 \dots s_m, U)$, существует такая система T, что
$s_1 \dots s_m t_2 \dots t_n$, $S \cup U \cup \{t_1 = s_0\} \overset{i}{\Rightarrow}$ (*пусто* T), и $X \cup Y$ — древесное решение T,
тогда и только тогда, когда по определению древесного решения существует правило $s_0 \to s_1 \dots s_m$, U с переменными, переименованными так, чтобы не совпадать ни с одной из $(t_1 \dots t_n, S)$, и существует такое древесное означивание Y, что:
в Y входят только переменные из $(s_0 \dots s_m, U)$, существует такая система T, что:
$(s_1 \dots s_m t_2 \dots t_n, S \cup U \cup \{t_1 = s_0\}) \overset{i}{\to}$ (*пусто*, T), и X — решение T;
тогда и только тогда, когда: по определению $\overset{j}{\Rightarrow}$ существует такая система T, что:
$(t_1 \dots t_n, S) \overset{0}{\Rightarrow}$ (*пусто*, T) и X — древесное решение S; что и требовалось доказать.

ЛИТЕРАТУРА

[Баттани 73] Battani G. et Meloni H. Interpréteur du langage de programmation Prolog; rapport intern, Groupe Intelligence Artificielle, Université Aix — Marseille II, septembre 1973.

[ваи Канегем 82] Van Caneghem M. Prolog II, Manuel d'utilisation rapport interne, Groupe Intelligence Artificielle, Université Aix — Marseille II, 1982.

[Кануи 82] Kanoui H. Prolog H. Manuel d'exemples; rapport interne Groupe Intelligence Artificielle, Université Aix — Marceille II.

[Клоксин 81] Clocksin W. F. et Mellish C. S. Programming in Prolog Springer Verlag, 1981. [Русский перевод: Клоксин У., Меллиш К. Программирование на языке Пролог.— М.: Мир, 1987.]

[Ковальский 76]Kowalski R. and Van Eden M. The semantics of predicate logic as programming language, JACM, 23 no 4, october 1976 pp. 733—743.

[Ковальский 79] Kowalski R. Logic for problem solving; North Holland 1979.

[Коелхо 79] Coelho H. How to solve it with Prolog; Laboratorio nacional de engenharia civil, Lisbonne, 1979.

[Колмероэ 70] Colmerauer A. Les systèmes-q ou un formalisme pour analyser et synthétiser des phrases sur ordinateur; Rapport interne 43, Department d'Informatic, Université de Montréal, septembre 1970.

[Колмероэ 73] Colmerauer A., Kanoui H., Pasero R. Un système de communication homme-machine en français; rapport de recherche CRI 72-18 Groupe Intelligence Artificielle, Université Aix — Marseille II, juin 1973

[Колмероэ 78] Colmerauer A. Metamorphosis Grammars, Natural language communication with computers; édité par L. Bolc, Lecture Notes in Computer Science 63, Springer Verlag, pp. 133—189.

[Колмероэ 82] Colmerauer A. Manuel de rérérence et modèle théorique Groupe Intelligence Artificielle, Université Aix — Marseille II.

[Робинсон 65] Robinson J. A. A machine — oriented logic based on the resolution principle; JACM 12 no 1, décembre 1965, pp. 227—234. Имеется перевод: Робинсон Дж. А. Машинно-ориентированная логика, основанная на принципе резолюции. — В кн.: Кибернетич. сборник, вып. 7 1970.]

[Руссель 72] Roussel P. Définition et traitement de l'égalité formelle en démonstration automatique; thèse de 3 cycle, Groupe Intelligence Artificielle, Université Aix — Marseille II, 1972.

[Руссель 75] Roussel P. Prolog: Manuel de référence et d'utilisation Groupe Intelligence Artificielle, Université Aix — Marseille II, 1975.

[Энналс 82] Ennals N. Micro-Prolog; Ellis — Horwood, 1982.

[Юэ 76] Huet G. Résolution d'équation dans les langages d'ordre 1, 2, ..., omega; thèses de doctorat d'état, Université Paris VII, septembre 1976.

ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ[1)]

Р. Ковальский

В этой статье я привожу аргументы в пользу логического программирования, сравнивая его с основанными на правилах языками для задач искусственного интеллекта и с функциональным программированием. Будут также рассмотрены две главные задачи, которые надо решить: во-первых, введение в логическое программирование средств, аналогичных использованию функций высших порядков в языках функционального программирования, во-вторых, проблема фреймов и ее связь с присваиванием. Я покажу, что комбинация объектного языка и метаязыка в духе системы FOL Вейхроуча дает перспективный подход к решению этих проблем.

1. ВВЕДЕНИЕ

В последнее время логическому программированию уделяется большое внимание в связи с тем, что оно было выбрано японцами в качестве основного машинного языка в проекте вычислительных систем пятого поколения. В дальнейшем при обсуждении логического программирования часто допускали двоякого рода путаницу, которую следует отметить.

(1) Так как Пролог (название языка программирования, разработанного в 1972 г. в Марселе А. Колмероэ и его коллегами) есть сокращение для «ПРОграммирование в терминах ЛОГики», Пролог часто путают с логическим программированием. Пролог базируется на логическом программировании почти в таком же смысле, в каком Лисп базируется на ламбда-исчислении. В этой работе я рассмотрю центральные понятия логического программирования и прокомментирую их связь с Прологом.

(2) Сам термин «логическое программирование» дает повод к путанице с языками программирования. Это приводит к недооценке той роли, которую должно сыграть логическое

[1)] Kowalski R. Logic programming. — IFIP'83, North Holland, 1983, p. 133—145.

программирование в областях спецификации программ, описания данных и представления знаний и запросов вообще. Объем статьи не позволяет мне воздать должное этим более широким применениям логического программирования. Хорошие примеры работ в этой области вместе с достаточно полной библиографией до 1981 г. можно найти в [12] [1]). Книга [30] дает более полное представление о логике для решения задач.

Существует еще одна путаница, в которой отчасти виноват и я. Понятие логического программирования само по себе плохо определено. В самой скромной форме оно относится только к программированию на хорновских дизъюнктах (см. разд. 2 ниже). Однако практический опыт показал, что это узкое понятие является слишком ограничительным. Было установлено, что программирование на хорновских дизъюнктах необходимо расширить в нескольких направлениях. Большая часть данной работы как раз и посвящена обсуждению этих расширений.

2. ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ НА ХОРНОВСКИХ ДИЗЪЮНКТАХ

2.1. Утверждения, правила и запросы

Логическая программа на основе хорновских дизъюнктов состоит из правил и утверждений. Каждое *утверждение* выражает простейшее отношение между индивидуумами (отдельными объектами), например:

Джон любит Мэри (1)

Мэри хочет велосипед (2)

Колесо часть велосипеда (3)

Боб достанет велосипед для Мэри (4)

Каждое *правило* имеет вид

A если B и C и ... и D

с единственным заключением A и одной или более посылками B, C, ..., D. Каждое заключение и условие, как и утверждение, выражает простейшее отношение между индивидуумами. Правила, как и утверждения, могут содержать в дополнение к *константам* (таким, как «Джон», «Мэри»)

[1]) Пять статей из [12] переведены в этом сборнике. — *Прим. ред.*

переменные (такие, как u, v, w, x, y), представляющие произвольных индивидуумов, например:

Боб любит x (5)
т. е. Боб любит всё

Мэри любит x, если x любит Мэри (6)
т. е. Мэри любит всех, кто любит ее

x достанет y для Мэри, если x любит Мэри и Мэри хочет y (7)

т. е. любой, кто любит Мэри, достанет ей все, что она хочет

Правила и утверждения называют еще *хорновскими дизъюнктами*. Константы и переменные называют *термами*. Как мы увидим позже, другие термы могут быть построены с использованием функциональных символов.

Логическая программа вызывается посредством предположения, что некоторая конъюнкция условий есть логическое следствие программы. Таким образом, например, если предполагается заключение

Джон достанет велосипед для Мэри (8)

то это приводит к вызову программы (1)—(7), которая дает ответ «да», так как (8) логически следует из дизъюнктов (1), (2) и (7). В более общем случае предполагаемое заключение (называемое также *запросом*) может иметь вид выходного шаблона, содержащего переменные вместе с одним или более условиями.

Например, запрос

Найти x, где Мэри любит x (9)

к программе (1)—(7) дает ответ

x = Джон

потому, что

Мэри любит Джона

является следствием программы (1)—(7). С другой стороны, запрос

Найти (x y), где x любит y и y любит x (10)

дает ответ

(x y) = (Джон Мэри)
(x y) = (Мэри Джон).

Подмножество логики, состоящее из предположений, которые имеют вид правил и утверждений, и из запросов, которые имеют вид конъюнкции условий, эквивалентно подмножеству логики, состоящему из *хорновских дизъюнктов*.

Логическое программирование отражает семантику логической импликации: любой ответ на запрос представляет собой просто сокращение для предложения, которое является логическим следствием программы. В результате логические программы можно понимать *декларативно*, не ссылаясь на поведение компьютера.

2.2. Процедуриая интерпретация хорновских дизъюнктов

Логическое программирование не было бы осуществимо, если бы дедуктивные рассуждения на уровне хорновских дизъюнктов не могли бы быть эффективно реализованы. В самом деле, термин «логическая программа» возник потому, что правила вида

A **если** B **и** C **и** ... **и** D

можно интерпретировать (и реализовать) как *процедуры*, которые сводят проблему A к подпроблемам B и C и ... и D. Утверждения вида A используются для прямого решения проблемы. В этой статье мы будем употреблять термины *цель*, *проблема* и *вызов процедуры* как для условий правил, так и для запросов.

Применение правила или утверждения для решения выбранной проблемы является специальным случаем правила вывода, называемого *правилом резолюции* [34]. Совмещение формы проблемы с формой заключения включает в себя подстановку термов вместо переменных. Например, совмещение заключения из правила

x достанет y для Мэри, **если** x **любит** Мэри
и Мэри хочет y

с проблемой

Джон достанет u для v

достигается подстановкой

Джон **вместо** x, Мэри **вместо** v **и** u **вместо** y
(или, **эквивалентно**, y вместо u)

Подстановка одновременно передает процедуре *вход* проблемы, а проблеме — *выход* процедуры. Процесс совмещения называется *унификацией*.

2.3. Параллелизм

Логическое программирование позволяет реализовать два основных вида параллелизма: *И-параллелизм*, когда несколько подпроблем

B и C и ... и D

могут решаться одновременно, и ***ИЛИ-параллелизм,*** когда несколько альтернативных правил

A если B и ... и C
A если D и ... и E

могут использоваться одновременно для решения одной и той же проблемы. *Унификацию* тоже можно проводить *параллельно.* Более того, *обратные рассуждения* (как в процедурной интерпретации), когда правила

A если B и C и ... и D

используются для сведения проблем к подпроблемам, могут осуществляться *параллельно* с *прямыми рассуждениями,* использующими эти правила для вывода заключения *A* из посылок B, C, ..., D.

Кроме работ по проекту пятого поколения [27] архитектуры ЭВМ, допускающие *ИЛИ*-параллелизм, исследовали Минкер на ZMOB [4], Конери и Киблер [17], Поллард [33] и Дарлингтон, Рив и Грегори на ALICE [18]. *И*-параллелизм в общем случае более труден. Здесь некоторые вопросы изучались Поллардом. Ограниченные, но важные случаи *И*-параллелизма в сочетании с ограниченной формой *ИЛИ*-параллелизма являются характерными чертами языка PARLOG [8, 9].

2.4. Пролог

Язык Пролог — строго *последовательный* язык. Процедуры и вызовы процедур выполняются в нем строго по очереди, в том порядке, в котором они записаны. Если вызов процедуры *завершается успешно*, Пролог «*продвигается вперед*», чтобы попытаться выполнить вызов следующей процедуры. Если вызов *неудачен*, происходит *отход назад* (*бектрекинг*) к ближайшему из предыдущих вызовов процедуры, который имеет неиспытанную соответствующую процедуру. Такое последовательное выполнение процедур и вызовов процедур позволяет программисту, пишущему на Прологе, влиять на эффективность и завершение выполнения программ по-

средством упорядочивания дизъюнктов в программе и условий, входящих в запрос или правила.

Сила и эффективность языка Пролог и его реализаций оказались такими, что он нашел многочисленные и разнообразные применения. Среди наиболее известных — применения в символической математике, планировании, автоматизированном проектировании, построении компиляторов, базах данных, обработке текстов на естественных языках, машинной индукции, отладке и экспертных системах. Главным препятствием для более широкого коммерческого использования является присущая существующим реализациям ограниченность памяти. Это ограничение несвойственно Прологу как таковому, а скорее отражает ограниченность ресурсов ЭВМ, которые до сих пор использовались для его реализации. В книгах [10, 13] приводится хорошее введение в Пролог, работа [21] содержит материалы для обучения 10- и 11-летних детей логическому программированию на подмножестве Пролога, в [13] имеется хорошая библиография по Прологу и его применениям.

2.5. Циклы

Поскольку в Прологе процедуры и вызовы процедур выполняются последовательно, нетрудно написать программы, которые приводят к бесконечным циклам. Большинство таких зацикливаний можно избежать простой переформулировкой программы. Простым примером служит цикл, возникающий в результате расширения отношения «часть» таким образом, чтобы один объект являлся частью другого, если от первого ко второму можно перейти с помощью цепочки связей отношения «часть». Например:

Колесо есть часть велосипеда.
Шина есть часть колеса.
x есть часть y, если x есть часть z и
z есть часть y
Найти x, где x есть часть велосипеда.

Программа зацикливается после получения первого ответа, потому что она пытается снова и снова использовать третий дизъюнкт для сведения проблемы вида

x есть часть z

к подпроблемам

x есть часть z′ и z′ есть часть z.

В этом случае можно избежать зацикливания, используя разные предикаты для одношаговой и многошаговой связей:

Колесо есть часть велосипеда. ($p1$)
Шина есть часть колеса. ($p2$)
x входит в y, если x есть часть y. ($p3$)
x входит в y, если x есть часть z
и z входит в y. ($p4$)

Найти x, где x входит в велосипед.

Программа ($p1 - p4$) останавливается после двух ответов

x = колесо
x = шина

Можно привести много случаев бесконечных циклов, которые устраняются только посредством гораздо более радикальной переформулировки программы. В таких случаях переформулировка равносильна реализации на Прологе «интеллектуального решателя задач» для обнаружения циклов. Более отдаленным решением проблемы могла бы быть замена Пролога на язык логического программирования, который будет включать некоторые формы обнаружения циклов.

2.6. Обратимость

Теоретически логическую программу можно использовать для поиска любой информации, которая из нее логически следует. Например, в предыдущем примере отношения «x входит в y» можно не только запросить информацию о частях велосипеда, т. е.

Найти x, где x входит в велосипед.

но и спросить, частью чего является шина, то есть

Найти x, где шина входит в x.

Это свойство логических программ, т. е. возможность находить любого индивидуума, находящегося в данном отношении с другими индивидуумами, называется *обратимостью.* Свойство обратимости делает логическое программирование похожим на языки запросов для реляционных баз данных.

К сожалению, в Прологе обратимость реализована плохо. Это связано с тем, что в Прологе вызовы процедур выполняются строго в том порядке, в котором они записаны. Например, при заданном x правило ($p4$) будет искать y следующим образом:

вначале ищется z, такой, что x часть z,
затем ищется y, такой, что z входит в y.

Для проблемы нахождения y по x указанная последовательность действий вполне приемлема. Однако для проблемы поиска x по заданному y было бы лучше решать подпроблемы в обратном порядке. В Прологе нельзя сделать это прямо, хотя в нем имеются некоторые внелогические примитивы, которые можно использовать для управления поведением при решении задачи. Более качественным, но отдаленным решением проблемы было бы создание более гибкого, более интеллектуального решателя задач. Такой решатель задач мог бы либо сам принимать решения в соответствии с подходящей стратегией решения подпроблем, либо следовать советам, выраженным в метаязыке.

2.7. Декларативная отладка

Хотя обратимость программ в Прологе ограничена, все же эта возможность представляет практическую ценность. Одним из самых полезных применений обратимости является поиск отношения вход-выход, определяемого заданной программой, например

Найти (x y), где x входит в y.

По заданным ($p1 - p4$) Пролог порождает следующую последовательность ответов:

(x y) = (колесо велосипед)
(x y) = (шина колесо)
(x y) = (шина велосипед)

в том порядке, в котором они найдены. Из-за этого обычный способ отладки программы тестированием на некоторых входах становится не нужным. Запрос в Прологе приводит к порождению всех пар вход — выход. Более того, та же самая техника отладки хорошо работает и на множествах правил, которые больше похожи на спецификации программ, чем собственно на программы.

На практике декларативная отладка в Прологе не может быть реализована полностью из-за ограничений на обратимость некоторых встроенных отношений. Например, в большинстве Пролог-систем наложены ограничения на обратимость арифметических отношений, таких, как

Plus(x y z), которое выполняется, когда $x + y = z$.

При этом соответственно ограничена обратимость любого отношения, определяемого программой, в которой используются такие отношения.

3. ЭКСПЕРТНЫЕ СИСТЕМЫ

Большинство экспертных систем реализовано на языках программирования, основанных на правилах, которые, в свою очередь, реализованы на Лиспе или некоторых более традиционных языках, — таких, как Паскаль. Так как Пролог также является языком, основанным на правилах, хотя и с более простым механизмом решения задач, то он представляет собой отличное средство для реализации экспертных систем.

3.1. Простой пример

Приводимый ниже набор дизъюнктов дает представление об очень упрощенном множестве правил, описывающем причинно-следственные связи между поломками в велосипеде:

шина имеет неисправность, если камера имеет прокол
шина имеет неисправность, если золотник имеет утечку
велосипед имеет неровный ход, если шина имеет неисправность
колесо имеет искривление, если спица имеет разрыв
велосипед имеет неровный ход, если колесо имеет искривление
тормоз имеет повреждение, если трос имеет повреждение
велосипед имеет аварийное состояние, если тормоз имеет повреждение

Чтобы обнаруживать причины неисправности велосипеда, нам нужно находить утверждения, такие, как

трос имеет повреждение.

которые, будучи добавлены к множеству правил, дают утверждение

велосипед имеет аварийное состояние.

На первый взгляд, это задача индукции, а не дедукции. Тем не менее она может быть решена и дедуктивно, если мы предположим, что база данных, состоящая из утверждений, находится вне программы, например в голове пользователя или в датчиках компьютера. Этот подход был элегантно развит Серго в средстве для логических программ, называемом «спроси — пользователя» [36], и в оболочке экспертной системы Хэммонда APES (22), реализованной на Прологе.

3.2. Перестановочность прямых и обратных рассуждений

Вместо использования индукции или средства «спроси — пользователя» такой же эффект можно получить, если переставить утверждения и запросы, условия и заключения. В примере с неисправностями велосипеда для того, чтобы новые правила имели смысл, мы изменим еще и имена предикатов. При этом получим следующее множество правил:

камера может иметь прокол,
если шина может иметь неисправность
золотник может иметь утечку,
если шина может иметь неисправность
шина может иметь неисправность
если велосипед может иметь неровный ход
спица может иметь разрыв,
если колесо может иметь искривление
колесо может иметь искривление,
если велосипед может иметь неровный ход
трос может иметь повреждение,
если тормоз может иметь повреждение
тормоз может иметь повреждение,
если велосипед может иметь аварийное состояние

Для того чтобы установить возможные причины неисправности велосипеда, теперь достаточно добавить утверждение

велосипед может иметь аварийное состояние

и сделать запрос

Найти (ху), где х может иметь у.

Новая формулировка обладает другими интересными свойствами. Прямые рассуждения (в новой формулировке), начинающиеся с утверждения «велосипед может иметь аварийное состояние», имитируют обратные рассуждения в старой формулировке, начинающиеся с запроса. Наоборот, обратные рассуждения в новой формулировке имитируют прямые рассуждения в старой.

В качестве другого примера рассмотрим предложение

если идет дождь и вы хотите остаться сухим, то пользуйтесь зонтиком.

Оно имеет вид

если условия, то действия

и прекрасно подходит для вывода в прямом направлении. Для того чтобы получить тот же эффект обратными рассуждениями, необходимо переформулировать правило:

вы останетесь сухим, если идет дождь и вы пользуетесь зонтиком.

Заметьте, что в первой формулировке опускается очевидная необходимость в модальной логике, чтобы иметь дело со словом «хотеть», и логике команд, чтобы иметь дело с императивом «пользуйтесь», а во второй этого нет.

Приведенные примеры показывают, что для большого класса задач прямые и обратные утверждения могут быть взаимно обратимы, если мы соответствующим образом переформулируем «базу знаний». Это полезно для оценки относительных достоинств прямых и обратных утверждений, так как методы вывода в различных экспертных системах коренным образом зависят от направления поиска, которое в них заложено.

3.3. Более гибкая формула

Для получения эффекта прямых рассуждений в системах с обратными рассуждениями можно также использовать более мощное преобразование, если смотреть на решаемую задачу как на задачу поиска пути. В таком случае правила можно заменить на утверждения. Например, в задаче о неисправностях велосипеда мы получаем:

(камера прокол) влечет (шина неисправность)
(золотник утечка) влечет (шина неисправность)
(шина неисправность) влечет (велосипед неровный ход)
(спица разрыв) влечет (колесо искривление)
(колесо искривление) влечет (велосипед неровный ход)
(трос повреждение) влечет (тормоз повреждение)
(тормоз повреждение) влечет (велосипед аварийное состояние)

Кроме того, нам нужны правила для расширения причинно-следственных отношений на цепочки случаев. Правила

x приводит к y, если x влечет y
x приводит к y, если x влечет z и z приводит к y

позволяют сделать это, избегая бесконечных циклов. В новой формулировке обратные рассуждения могут быть использованы как для диагностики неисправностей, например,

Найти x, где x приводит к (велосипед аварийное состояние)
Ответ: x = (тормоз поломка)
x = (провод поломка)

так и для выявления эффекта неисправностей:

Найти x, где (спица поломка) приводит к x
Ответ: x = (колесо искривление)
x = (велосипед неровный ход)

К сожалению, в Прологе по причине жесткого порядка, в котором выполняются вызовы процедур, запросы по диагностике не могут выполняться эффективно.

Последняя формулировка примера о неисправностях велосипеда более приспособлена для обратных рассуждений, нежели для прямых. Однако можно привести и другие переформулировки, которые будут более эффективны для прямых рассуждений.

Здесь следует заметить, что логическое программирование в широком смысле полностью нейтрально по отношению к методу поиска вывода. До сих пор обратные рассуждения были, по-видимому, единственно эффективным методом выполнения. Однако во многих применениях комбинация прямых и обратных рассуждений, как, например, в методе графа связей [30], заслуживает дальнейшего изучения.

3.4. Рекурсивные структуры данных

Большинство экспертных систем реализовано в системах, основанных на правилах, которые, в свою очередь, встроены в более обычный включающий язык программирования. Включающий язык может использоваться для реализации вспомогательных процедур, которые используются в условиях и действиях правил. Но для логики хорновских дизъюнктов характерно, что она хорошо приспособлена для определения как правил экспертных систем, так и более рутинных процедур включающего языка программирования. В большой степени это связано с тем, что эта логика приспособлена для работы с рекурсивными структурами данных.

Простым примером служит конструктор списков cons(x y), обозначающий список, первым элементом которого является x, а остатком — список y, как в языке программирования Лисп. Для обозначения пустого списка можно использовать символ Nil. Следующая программа определяет отношение Length(x, y), которое выполняется, когда x — список, а y — его длина:

```
Length(Nil 0)
Length(cons(x y) u), если Length(y v)
                      и Plus(v 1 u),
```

Здесь предикаты Length и Plus[1]) записаны в *префиксной форме*, в которой имя предикатов предшествует именам индивидуумов (заключенным в скобки), к которым применяется этот предикат. В качестве имен индивидуумов (называемых также *термами*) могут выступать константы, переменные или выражения, полученные приписыванием функционального символа, как, например, cons, к подходящему числу термов.

Обратные рассуждения с такими предикатами моделируют обычные вычисления определений рекурсивных функций. Прямые рассуждения в таких случаях неэффективны, так как если дано большое число таких определений, то следствия из них порождаются независимо от вида решаемой задачи.

Конструктор списков cons является примером использования функционального символа для именования составного объекта, а не результата вычисления. Использование функциональных символов для именования результатов вычислений является основой функционального программирования.

4. ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ В СРАВНЕНИИ С РЕЛЯЦИОННЫМ

Определение Length выглядит более естественно в функциональных обозначениях:

length(Nil) = 0
length(cons(x y)) = length(y) + 1

Однако такое функциональное определение легко переводится в реляционные обозначения, если использовать эквивалентность

$$f(x) = y \leftrightarrow F(x\ y) \qquad (E1)$$

где F — предикатный символ, соответствующий функциональному символу f, и свойства отношения равенства

$$P(f(x)) \leftrightarrow \text{Для-всех } y\,[P(y) \text{ если } f(x) = y] \qquad (E2)$$
$$P(f(x) \leftrightarrow \text{Существует } y\,[P(y) \text{ и } f(x) = y] \qquad (E3)$$

Здесь P(t) представляет любое выражение, содержащее терм t. Символ $\leftrightarrow$ (читается «тогда и только тогда») выражает эквивалентность. Для-всех (обычно обозначается через $\forall$) и Существует (обычно обозначает через $\exists$) — это соответственно квантор всеобщности и квантор существования. Заметим, что до сих пор мы избегали прямого использования кванторов, потому что неявно предполагалось, что все переменные

[1]) Plus(x y z) выполняется, если x + y = z. — *Прим. ред.*

в хорновских дизъюнктах связаны квантором всеобщности, в то время как переменные в запросах неявно считались связанными квантором существования. Таким образом, правило

Мэри любит x, если x любит Мэри

является сокращением для утверждения

Для-всех x (Мэри любит x, если x любит Мэри).

Применяя (E1 — E3) к определениям length и используя функциональные обозначения, мы получаем эквивалентное определение в реляционных обозначениях:

length(Nil) = 0
↔ Length(Nil 0)
length(cons(x y)) = length(y) + 1
↔ Length(cons(x y) length(y) + 1)
↔ Length(cons(x y) u), если length(y) + 1 = u
↔ Length(cons(x y) u), если Plus(length(y) 1 u)
↔ Length(cons(x y) u), если Plus(v 1 u)
и length(y) = v
↔ Length(cons(x y) u), если Plus(v 1 u)
и Length(y v)

Каждый шаг перевода состоит в применении эквивалентности для замены подвыражения на эквивалентное подвыражение. Аналогичный перевод используется в [20].

Этот метод перевода дает простое доказательство того, что хорновские дизъюнкты могут быть использованы для представления всех вычислимых функций (впервые это было доказано Хиллом [25] более сложным методом).

Набросок доказательства: С помощью проиллюстрированного выше метода перевода любую систему уравнений Эрбрана — Гёделя (представляющую вычислимую функцию) можно перевести в эквивалентное множество хорновских дизъюнктов. Более того, эффект использования уравнения как переписывающего правила для замены термов, которые совмещаются с левыми частями уравнений, на правые части этих уравнений можно промоделировать, используя соответствующие дизъюнкты в обратном направлении — для сведения проблем к подпроблемам. Такое моделирование ставит в соответствие любому вычислению посредством уравнений структурно схожее с ним вычисление на хорновских дизъюнктах, приводящее к тому же результату. Это показывает, что логические программы на хорновских дизъюнктах вычисляют все функции, которые вычислимы посредством уравнений Эрбрана — Гёделя, что и требовалось доказать.

Несмотря на то, что функциональные обозначения больше привычны пользователю, чем реляционные, вычисления посредством переписывающих правил менее универсальны, чем обратные рассуждения. В то время как функциональная программа может использоваться только для определения длины списка, реляционная программа обратима, и к ней можно обращаться с запросами, как к базе данных.

В случае определения Length это обращение работает даже в Прологе:

Найти x, где
Length(cons(M cons(A cons(R cons(Y Nil)))) x)
Ответ: x = 4
Найти x, где Length(x 2)
Ответ: x = (cons(u cons(v Nil))).

В ответе на второй запрос алгоритм унификации дает один результат, обозначающий бесконечно много списков. В языке функционального программирования пришлось бы писать разные программы для разных задач: вычисления длины списка и нахождения списков данной длины. Более того, во втором случае пришлось бы использовать функцию более высокого порядка, выдающую функцию как результат. В других случаях достичь эффекта функций более высоких порядков не так просто. Позднее мы исследуем вопрос о том, как можно получить такой эффект, сочетая объектный язык с метаязыком.

Обозначение cons для списков довольно громоздкое. В последующем мы будем использовать обозначение из Лиспа ($a_1\ a_2\ \ldots\ a_n$) как сокращение для $cons(a_1\ cons(a_2\ \ldots\ \ldots\ cons(a_n\ Nil)\ \ldots))$ и ( ) вместо Nil, как в языке Микро-Пролог [11].

5. РАСШИРЕНИЯ ПРОГРАММИРОВАНИЯ НА ХОРНОВСКИХ ДИЗЪЮНКТАХ

Реляционное программирование на хорновских дизъюнктах имеет еще одно преимущество перед функциональным программированием: оно допускает расширения, которые позволяют более широко использовать всю силу классической логики. Три наиболее очевидных расширения, которые реализованы во многих Пролог-системах с помощью внелогических средств Пролога, это

отрицание по неудаче,
правила как посылки правил и
списки решений как индивидуумы.

Более широкое значение имеют расширения, которые позволяют комбинировать рассуждения объектного уровня с рассуждениями на метауровне.

5.1. Отрицание по неудаче

Использование *отрицания по неудаче*[1]) означает, что негативное утверждение вида

Не(Р)

считается выполненным, если не удается установить, что выполняется Р. Таким образом, запрос

Найти х, где Мэри любит х и Не(х хочет у),
то есть того, кто любит Мэри и ничего не хочет

при заданных предположениях (1)—(7) имеет ответы

х = Боб
х = Джон

Отрицание по неудаче базируется на предположении о замкнутости мира[2]), т. е. считается, что «база знаний» «знает» все, что нужно знать. Впервые отрицание по неудаче было введено Хьюиттом в языке программирования Плэнер [24][3]). Кларк [6] показал, что предположение о замкнутости мира, которое оправдывает отрицание по неудаче, можно выразить в обычной логике посредством переписывания частей определений, начинающихся с «если», в форме «тогда и только тогда». Например, предположение о замкнутости мира, примененное к двум дизъюнктам

Боб достанет велосипед для Мэри (4)
х достанет у для Мэри, если х любит Мэри
и Мэри хочет у (7)

дает предложение

х достанет у для Мэри ←→
х = Боб и у = велосипед
или х любит Мэри и Мэри хочет у

которое выразимо в классической логике, но не в логике хорновских дизъюнктов.

---

[1]) В оригинале — negation by failure. Говорят также «отрицание как неудача» (negation as failure). — *Прим. ред.*

[2]) Автор допускает неточность при формулировке свойства замкнутости мира. Правильная формулировка имеется, например, в Reiter R. On closed world data bases. — In: Logic and Data Bases. — Plenum Press, New York, 1978, p. 55—76. — *Прим. перев.*

[3]) См. также: Пильщиков Г. Н. Язык Плэнер. М.: Наука, 1983. — *Прим. ред.*

К сожалению, наиболее очевидная реализация отрицания по неудаче в Прологе посредством внелогических средств опасно чувствительна к порядку, в котором условия выбираются, чтобы получить решение. Например, выполнение вызовов процедуры в том порядке, в котором они написаны, при запросе

Найти x, где Не (x хочет y) и Мэри любит x

некорректно приводит к неудаче при попытке найти ответ, так как к неудаче приводит попытка решить первую подпроблему «Не (x хочет y)». Это происходит из-за того, что попытка решить подпроблему, стоящую под знаком отрицания, «x хочет y», приводит к удаче, давая ответ (x y) = (Мери велосипед).

Для того чтобы исправить эту ошибку, необходимо рассматривать x и y по-разному в негативном условии «Не (x хочет y)». В обычной логике это можно выразить предложением

Найти x, где Не(Существует y (x хочет y))
и Мэри любит x.

Корректную реализацию можно получить, если считать, что негативное условие

Не(Существует (x_1 x_2 ... x_n)(P))

выполняется тогда и только тогда, когда не удается доказать, что, выполняется условие P и P не содержит переменных, отличных от x_1, ..., x_n.

Если P содержит другие переменные, то простейшей стратегией будет сгенерировать ошибку управляющего характера, подобно той, которая возникает, когда не может выполниться обращение арифметического отношения. Более претенциозная стратегия — порождать соответствующие наборы значений таких переменных и выдавать в качестве решений для запроса

"Не(Существует (x_1 x_2 ... x_n)(P))"

те наборы, при которых P не выполняется. Можно предложить еще одно решение, промежуточное по силе между двумя указанными: откладывать выполнение вызова процедуры до тех пор, пока дополнительные переменные не будут замещены значениями при других вызовах процедуры. Аналогичный подход был реализован в последней марсельской версии Пролога.

Реализация отрицания как отрицания по неудаче весьма эффективна по сравнению с классическими методами обра-

щения с отрицанием. Более того, исследование альтернативных путей решения задачи P при попытке показать, что все они ведут к неудаче, может проводиться по одному, как в Прологе, или *параллельно*. Такое параллельное исследование альтернатив есть *ИЛИ-параллелизм*.

Упомянутые модификации гарантируют корректность отрицания по неудаче. Тем не менее они не обеспечивают ему полной силы классического отрицания. В [6] и [30] приводятся примеры, подтверждающие это. Таким образом, отрицание по неудаче, соответствующим образом модифицированное, является корректным, но неполным.

5.2. Правила как условия правил

Такие предложения, как

Мэри достанет приглашение для x
если Для-всех y[x достанет y для Мэри,
если Мэри хочет y]

т. е. Мэри приглашает всех, кто достанет ей все, что она хочет, можно реализовать посредством их сведéния к логически эквивалентному двойному отрицанию, например,

Мэри достанет приглашение для x,
если Не(Существует y[Не (x достанет y для Мэри)
и Мэри хочет y]).

Эта интерпретация конструкции Для-всех, очевидно, подвержена тем же ограничениям, оговоркам и неполноте, что и интерпретация отрицания как отрицания по неудаче.

Следует заметить, что во многих случаях при использовании конструкции Для-всех удается избежать использования рекурсии. Очевидным примером служит определение подмножества:

x̀ есть подмножество y,
если Для-всех z [z принадлежит y,
если z принадлежит x].

Это определение нельзя перевести в хорновские дизъюнкты без обращения к рекурсии. Например,

Nil есть подмножество y
cons(z u) есть подмножество y, если z принадлежит y и u
есть подмножество y

при условии, что множества представлены списками.

Часто утверждают, что итерация более естественна, чем рекурсия. В защиту этого иногда приводят примеры,

аналогичные определению подмножества с использованием Для-всех. В самом деле, реализация Для-всех посредством двойного отрицания и последовательного исследования альтернатив, как в Прологе, *моделирует итерацию* и, следовательно, как будто поддерживает это утверждение. С другой стороны, исследование альтернатив можно также выполнять *параллельно*. Такой *ИЛИ*-параллелизм можно применить и к реализации Для-всех, и в какой-то мере он даже более естественен, чем итерация. Но это не та итерация, которая более естественная, чем рекурсия. Это именно Для-всех.

5.3. Спецификация программ

Хотя конструкции типа Для-всех все более поддерживаются как средства языков программирования очень высокого уровня, их обычно рассматривают как средства языков спецификаций программ. Простейшим примером может служить спецификация наибольшего общего делителя:

z есть наибольший общий делитель x и y $\leftrightarrow$ z делит x и z делит y и
Для-всех u [z $\geqslant$ u, если u делит x и u делит y].

Очевидно, что такую спецификацию можно выполнять, хотя и очень неэффективно, посредством сведения Для-всех к двойному отрицанию. В то же время алгоритм Евклида, который гораздо эффективней, можно выразить на языке хорновских дизъюнктов:

y есть наибольший общий делитель x и y, если остаток от деления x на y есть 0.
z есть наибольший общий делитель x и y, если остаток от деления x на y есть u и u $>$ 0 и z есть наибольший общий делитель y и u.
z есть наибольший общий делитель x и y, если x $<$ y.
и z есть наибольший общий делитель y и x.

Корректность алгоритма Евклида, рассматриваемого как программа, относительно определения наибольшего общего делителя, рассматриваемого как спецификация программы, сводится к математическому доказательству того, что оба описания дают одно и то же понятие. Таким образом, задача верификации программы выходит за рамки программирования и попадает в область математики. Более того, при этом требуется, чтобы интерпретация Для-всех была вполне классической, а не была более слабой интерпретацией в терминах отрицания по неудаче. Этот подход к верификации программ впервые появился в работах Кларка и Тернлунда [11],

Кларка [5], а также в статье Хоггера [26], основанной на более ранней работе Дарлингтона и Берстолла [2]. Другие ссылки на литературу можно найти в [12].

5.4. Списки решений как индивидуумы

Во многих приложениях недостаточно просто напечатать все ответы на запрос. Часто необходимо ими манипулировать. Для этого хорошо подходят обозначения, принятые в теории множеств.

Условия вида

$$y = \{x : P\}$$

можно реализовать, порождая все ответы на запрос

Найти x, где P

и составляя из них список. Например, задачу определения числа предметов, которые может достать Джон, можно представить запросом

Найти z, где y = {x : Джон достанет x} и Length(y z)

Конструктор множеств $y = \{x : P\}$ можно эффективно реализовать, порождая все альтернативные решения для P. Альтернативы могут порождаться последовательно, как в Прологе, или *параллельно*. Такое параллельное порождение множества решений снова приводит нас к *ИЛИ-параллелизму*. Однако в обоих случаях явное перечисление всех решений, хотя и не противоречит обычному построению множеств, но более ограниченно, чем построение множества в классической теории множеств.

Конструктор множеств характерен для многих языков запросов к базам данных и языков спецификаций программ, а также он является существенной чертой языка программирования СЕТЛ [19][1]) и языков функционального программирования HOPE [3] и KRC [37]. Он был реализован в нескольких Пролог-системах, из которых наиболее заслуживает внимания система Дэвида Уоррена [38].

6. КОМБИНИРОВАНИЕ РАССУЖДЕНИЙ НА ОБЪЕКТНОМ УРОВНЕ С РАССУЖДЕНИЯМИ НА МЕТАУРОВНЕ

В этом разделе мы покажем, как использовать рассуждения на метауровне, чтобы моделировать программирование

[1]) См также: Левин Д. Я. Язык сверхвысокого уровня СЕТЛ и его реализация на машине БЭСМ6. Новосибирск: Наука, 1983. — *Прим. перев.*

с функциями высших типов. Прежде всего нам надо описать предикат доказуемости Demo, который связывает объектный язык с метаязыком. Более детальное изложение можно найти в [1] и [30][1]).

6.1. Предикат доказуемости

Взаимодействие между объектным уровнем, на котором происходит исполнение программ, и метауровнем, на котором манипулируют с программами, можно реализовать посредством предиката доказуемости

Demo(A B C), который истинен, когда попытка показать, что из предположения с именем A вытекает заключение с именем B дает результат с именем C.

Предикат Demo можно использовать внутри программ метауровня, чтобы ссылаться на программы объективного уровня. Его можно реализовать при помощи определения, заданного полностью в метаязыке, или посредством декодирования имен из метауровня в соответствующие выражения объектного уровня. Это аналогично двум способам, которыми программа на Лиспе может, используя eval, вызывать интерпретатор Лиспа: либо выполняя записанное на Лиспе определение интерпретатора, либо вызывая интерпретатор.

Реализация предиката доказуемости с помощью определения в метаязыке — это обычный прием математической логики. Проиллюстрируем, как указанные ниже хорновские дизъюнкты метауровня определяют на самом верхнем уровне «решатель задач» объектного уровня, т. е. выполнение программ на хорновских дизъюнктах. Это подобно определению интерпретатора Лиспа на Лиспе. Для простоты мы опустим третий аргумент предиката Demo. Здесь Demo(A B) истинно, когда заключение с именем B следует из посылок с именем A. Будем считать, что заключение представлено в виде списка целей:

```
Demo(x ( ))
Demo(x y), если Выбрать(y цель остаток)
           и Найти(x цель дизъюнкт)
           и Совместить(цель дизъюнкт)
           и Новые-цели(цель дизъюнкт остаток новые)
           и Demo(x новые)
```

[1]) Заслуживает внимания более поздняя работа: Bowen K. A. Meta-level programming and knowledge representation. — New generation computing, v. 3, No 4, 1985, p. 359—383. — *Прим. перев.*

Для удобства через «цель», «остаток», «дизъюнкт» и «новые» мы обозначили переменные. Первый дизъюнкт утверждает, что пустое множество целей следует из любого множества посылок. Второй утверждает, что множество посылок x является решением для набора целей y, если этот набор содержит дизъюнкт, совместимый с выбранной целью, и, кроме того, то же самое множество посылок является решением для нового набора целей, полученного из старого заменой выбранной цели на новые, взятые из этого дизъюнкта.

Такое определение предиката доказуемости на метауровне имеет много применений. В частности, его можно использовать для реализации более мощного и гибкого механизма решения задач объектного уровня. Например, можно воспользоваться последовательным, как в Прологе, исполнением программ на метауровне, чтобы исполнить определение решателя задач объектного уровня, использующего более тонкий критерий выбора вызовов процедуры.

Определением на метауровне можно пользоваться не только для того, чтобы показать, что из данных посылок следует данное заключение. Его можно также обратить, чтобы порождать посылки или заключения. Таким образом, индукцию на объектном уровне можно осуществить посредством дедукции на метауровне.

Если независимо сам по себе задан объектный язык, то предикат Demo можно реализовать с помощью *правила рефлексии*, как в FOL Вейхрауча [39]:

> Решение для цели на метауровне вида Demo(A B C) можно найти прямо в объектном языке установлением того, что попытка вывести из посылок с именем A заключение с именем B дает результат с именем C.

Такой перевод работы с метауровня на объектный уровень требует, чтобы имена, которые записываются в метаязыке как термы, декодировались в выражения объектного уровня, именами которых они являются. Ниже об этом будет сказано более подробно.

6.2. Ссылки на себя

Сказанное выше вовсе не означает, что объектный язык и метаязык — это один и тот же язык. Такая система будет ближе к естественному языку, чем система, в которой два уровня полностью разделяются по двум отдельным языкам. Она позволяет отличать предложения вида

«Некто невиновен, если не доказано, что он виновен»,

в языке которых метауровень и объектный уровень смешаны, от предложений вида

«Некто невиновен, если он не является виновным»,

выраженных целиком на объектном уровне. Она позволяет также формализовать утверждения, содержащие ссылку на себя, например утверждение

«Это предложение недоказуемо»,

которое истинно, но недоказуемо, как в гёделевой арифметизации математики.

В ситуации, когда объектный язык и метаязык совпадают, некоторое приближение к рефлексии можно формализовать в Прологе, используя его внелогические средства. В частности, правило

Demo(x y), если y

можно использовать в том случае, когда множество посылок идентифицируется с текущим глобальным множеством дизъюнктов. Здесь первое вхождение y надо интерпретировать как имя второго вхождения. Однако это нельзя сделать непротиворечиво, так как возникают проблемы, когда переменным разрешается служить именами для самих себя.

6.3. Имена выражений

В арифметизации метаматематики и в других формализациях предиката доказуемости часто именуют выражения термами, похожими на них по структуре. Например, дизъюнкту

Мэри любит x, если x любит Мэри

можно дать имя, которое является списком списков:

((ИКС)(ЛЮБИТ МЭРИ ИКС)(ЛЮБИТ ИКС МЭРИ)) (L)

где имена предикатных символов, констант и переменных являются константами, а переменные, связанные квантором всеобщности, явно перечисляются.

Так как структурированные термы приводят к довольно громоздким именам, полезно в качестве сокращения использовать запись с кавычками:

«P» в качестве одного из имен для P.

Эта запись будет однозначной, если *явно указаны кванторы для всех переменных объектного уровня.* Таким образом, сокращением для (L) будет

Для-всех x (Мэри любит x, если x любит Мэри)

Вместо структурированных термов для именования выражений часто удобно использовать константы или другие термы, которые лишь частично отражают структуру этих выражений. Такое соответствие между термами метауровня и выражениями объектного уровня можно записывать с помощью неформальных предложений, например,

Пусть М имя для
Для-всех х (Мэри любит х, если х любит Мэри) конец.

6.4. Вычисления с функциями высших порядков

Важной особенностью языков функционального программирования является использование функций высших порядков. Подобные средства можно ввести в логическое программирование, включив кое-что из логики более высокого порядка. Другая альтернатива состоит в моделировании функций высших порядков с помощью метаязыка. Такой подход применил Уоррен в [38]. Мы продемонстрируем это на известном примере, функции Maplist, которая применяет заданную функцию ко всем элементам заданного списка. Допустим, что результат у применения функции к аргументу х задается отношением F(x y). Тогда функцию Maplist можно определить следующим образом:

Пусть Map имя для
Maplist(( ) ( ))
Maplist(cons(x1 x2) cons(y1 y2))
если F(x1 y1)
и Maplist(x2 y2) конец.

Для того чтобы применить Maplist к различным функциям, мы будем объединять определение Map с разными определениями отношения F, например,

Пусть F1 имя для
F(x, y), если Plus(x 2 y) конец.
Пусть F2 имя для
F(x y), если Times(x 2 y) конец.

Map можно понимать как модуль, который импортирует определения отношения F из модулей F1 или F2.

Пусть «+» — инфиксный, ассоциативный, бинарный функциональный символ, служащий именем для результата объединения двух множеств предложений, а Arith — имя для некоторого множества дизъюнктов, которое определяет

предикаты Plus и Times. Для того чтобы применить Maplist к F1 и списку (1), (2), (3), мы делаем запрос

Найти y, где Demo
(Map + F1 + Arith
"Существует x(Maplist((1 2 3) x))"y).
Ответ: y = ((ТРИ ЧЕТЫРЕ ПЯТЬ)).

Чтобы применить Maplist к F2 и тому же списку, делаем запрос

Найти y, где Demo
(Map + F2 + Arith
"Существует x(Maplist((1 2 3) x))"y).
Ответ: y = ((ДВА ЧЕТЫРЕ ШЕСТЬ)).

Здесь мы для простоты предположили, что третий параметр предиката Demo является списком имен всех решений для переменных, связанных квантором существования во втором параметре. Для того чтобы преобразовать имена результатов в результаты, объединяя метауровень и объектный уровень в одном выражении, необходимо ввести предикат именования:

Имя(x y) выполняется, когда x служит именем для y.

Например,

Имя(МЭРИ Мэри)
Имя(Два 2)

Тогда на запрос

Найти y, где Demo
(Map + F1 + Arith
"Существует x(Maplist((1 2 3) x))" z)
и Имя(z y)

получим ответ

y = ((3 4 5)).

Предикат именования необходим и для формализации таких предложений, как

Невиновен(x), если Не(Demo(y "Виновен(x)"))
и Известные-факты(y).

Опять для простоты используется двухместный предикат Demo. Более формальное определение:

Невиновен(x), если Не(Demo(y (( ) (Виновен z))))
и Имя(z x)
и Известные-факты(y).

В дальнейшем мы будем продолжать использовать менее формальные обозначения, хотя это скрывает упомянутые сложности. Заметим здесь же, что ссылка на странице 162 в [1] неправильна, так как в ней не учитывается необходимость введения такого предиката именования.

Определение Maplist с использованием метауровня бесспорно сложнее, чем определение с использованием функций высших порядков. Однако можно привести более сложные примеры, в которых применение метауровня выглядит проще и естественней, чем при использовании функций или логик высших порядков. В любом случае соотношение между метаязыком и логикой высших порядков заслуживает большего внимания, чем ему уделялось до настоящего времени.

7. ПРОБЛЕМА ФРЕЙМОВ И ПРИСВАИВАНИЕ

Отсутствие в логическом программировании оператора присваивания может приводить к серьезной неэффективности вычислений. Это становится особенно очевидным, когда мы пытаемся писать логические программы, работающие с реляционными структурами данных. Появляющаяся при этом неэффективность имеет долгую историю в области искусственного интеллекта и носит название «проблема фреймов». В этом разделе мы покажем, что проблемы фреймов можно избежать, комбинируя языки объектного уровня и метауровня и спрятав присваивание в правило рефлексии. При этом достигается эффективность присваивания и сохраняется чистота логики.

7.1. Отношения как структуры данных

Использование отношений, а не рекурсивных структур данных, обладает тем преимуществом, что программы могут находить элементы явно, не прибегая к рекурсии. Простым примером может служить определение упорядоченной последовательности:

Упорядочен(x), если Для-всех i ($x_i \leqslant x_{i+1}$).

Здесь для простоты использованы функциональные обозначения для индексов. Это определение можно перевести в реляционные обозначения, как описано ранее. Для того чтобы проверить, упорядочена ли последовательность

A : 2 4 1

сначала необходимо описать ее множеством дизъюнктов:

$A_1 = 2$ $A_2 = 4$ $A_3 = 1$.

Такое употребление реляционных структур данных аналогично использованию массивов. Концептуально оно дает доступ к i-му элементу последовательности, в то время как при представлении последовательности в виде списка требуется i шагов. Кроме того, его преимущество над массивами и реляционными базами данных заключается в том, что этот способ позволяет описывать последовательности с помощью как утверждений, так и правил, например,

$B_i = i$, если $1 \leqslant i$ и $i < 100$.

7.2. Проблема фреймов

Проблема возникает, когда нужно получить новые структуры из старых, — например, при перестановке i-го и j-го элементов последовательности. Для именования новой последовательности удобно использовать функциональный символ обмен(x i y). Новая последовательность определяется правилами, описывающими ее элементы:

обмен(x i j)$_i = x_j$
обмен(x i j)$_j = x_i$
обмен(x i j)$_k = x_k$, если $k \neq i$ и $k \neq j$.

Последний дизъюнкт определения как раз и создает *проблему фреймов* — вычислительную неэффективность явного рассуждения о том, что *k*-й элемент новой последовательности совпадает с *k*-м элементом старой. Неэффективность возникает независимо от того, используется ли этот дизъюнкт, называемый также *аксиомой фрейма*, для рассуждений в прямом или обратном направлении.

Проблема фреймов возникает в скрытой форме даже тогда, когда последовательности представлены рекурсивными структурами данных: чтобы переставить i-й и j-й элементы, необходимо для построения нового списка скопировать первые i и j элементов старого.

Проблема фреймов в традиционных языках программирования не возникает благодаря деструктивному присваиванию. Логический статус присваивания станет яснее, если мы рассмотрим операции над структурами данных — такие, как перестановка элементов последовательности — более абстрактно.

7.3. Абстрактная программа сортировки перестановками

Программы высокого уровня, которые обрабатывают последовательности, можно записать в абстрактной форме, не

упоминая конкретных структур данных. Простым примером может служить программа сортировки перестановками [1]):

сорт(x) = x, если Упорядочен(x)
сорт(x) = z, если i < j и $x_i > x_j$
и Перестановка(x i j y)
и сорт(y) = z.

Заметим, что в функциональных обозначениях выражение $x_i > x_j$ является сокращением для

$x_i = u$ и $x_j = v$ и $u > v$.

Предикат «Перестановка(x i j y)», который выполняется в том случае, когда последовательность y является результатом перестановки i-го и j-го элементов последовательности x, связывает абстрактную программу сортировки на верхнем уровне с представлением последовательностей на нижнем уровне. Например, если последовательности представлены отношениями, «Перестановка» определяется утверждением

Перестановка(x i j обмен(x i j))

вместе с тремя уравнениями, которые определяют элементы новой последовательности обмен(x i j).

Сам вид предиката «Перестановка» в абстрактной программе сортировки показывает, что на концептуальном уровне перестановка элементов последовательности приводит к созданию *новой последовательности*. Программы, которые порождают новую последовательность путем необратимого изменения старой, затемняют семантику ради достижения эффективности.

7.4. Присваивание

Присваивание можно промоделировать и в Прологе, используя его внелогические средства, позволяющие удалять и добавлять дизъюнкты. Следующий дизъюнкт является такой реализацией предиката «Перестановка»:

Перестановка(x i j x), если Убрать($x_i = u$)
и Убрать($x_j = v$)
и Добавить($x_i = v$)
и Добавить($x_j = u$)

Здесь из-за добавления и удаления дизъюнктов возникает побочный эффект изменения глобальной базы данных. Имена старой и новой баз данных одни и те же.

[1]) Эту сортировку называют также сортировкой обменами или обменной сортировкой. — *Прим. ред.*

Такое использование присваивания в Пролог-программах значительно повышает их эффективность, но без надлежащей дисциплины оно создает серьезную опасность для их логической чистоты. Это в свою очередь отрицательно влияет на возможность параллельного выполнения программ и осложняет их спецификацию, преобразования и верификацию.

7.5. Представление последовательностей предложениями

С логической точки зрения использование присваивания для добавления и удаления дизъюнктов можно рассматривать как операцию метауровня, связывающую одну базу данных с другой. Ее можно формализовать в метаязыке, где доступ к базе данных задается посредством предиката Demo. Это в свою очередь можно реализовать либо при помощи определения Demo на метауровне, либо при помощи рефлексии. Использование определений на метауровне снова приводит к неэффективности, связанной с проблемой фреймов. Использование рефлексии дает тем не менее возможность использовать присваивание в целях оптимизации процесса декодирования имен метауровня в базы данных объектного уровня. Чтобы конкретизировать это, рассмотрим снова предыдущий пример.

Здесь для определения последовательностей мы будем использовать предложения, но таким образом, что одно и то же предложение можно использовать для определения *различных последовательностей.* По этой причине мы представим последовательность множеством предложений, не содержащим ее имени. Например,

Пусть A имя для
val(1) = 2 val(2) = 4 val(3) = 1 конец.

Теперь предикат «Перестановка» можно определить так:

Перестановка(x i j y),
если Demo(x "val(i) = u")
и Demo(x "val(j) = v")
и Принять(x "val(i) = v" z)
и Принять(z "val(j) = u" y).

Здесь для простоты мы снова использовали вместо трехместного предиката Demo двухместный.

Предикат «Принять» для баз данных, определяемых утверждениями, определяется посредством

Принять(x "val(i) = v" z),
если Demo(x "val(i) = u")
и x — "val(i) = u" = y
и y + "val(i) = v" = z

Здесь x — y есть имя множества предложений, получающихся в результате удаления предложения с именем y из множества предложений с именем x, а x + y является именем множества предложений, получающихся в результате добавления предложения с именем y к множеству предложений с именем x.

Случай, когда базы данных определяются при помощи и правил, и утверждений, не намного сложнее. Если утверждение, которое следует принять в базу данных, противоречит правилу, то с ним нужно обращаться как с исключением: к правилу добавляется дополнительное условие, которое запрещает его применение в этом исключительном случае. Мы не будем приводить детали формализации.

Заметим, что между двумя целями

Demo(x "val(i) = u")
Demo(x "Существует u(val(i) = u)" (z))

имеется тонкое различие. Если предположить, что вместо переменной i уже сделана подстановка конкретного терма, то первая цель содержит во втором параметре переменную, в то время как вторая — нет. Тем не менее эффект обеих целей один и тот же: найти i-е значение в последовательности с именем x. Официально рефлексию можно использовать только для второй цели. Однако, так как вторая цель дает тот же результат, что и первая, мы можем считать, что рефлексию можно использовать и для первой цели.

Заметим также, что для работы с программой сортировки перестановками нам нужно уметь решать цели вида $x_i = u$. Это можно сделать с помощью правила

$x_i = u$, если Demo(x "val(i) = u")

Результирующая программа сортировки может оказаться синтаксически сложной, зато она выражается в чистой логике, без присваивания. С другой стороны, в запросах вида

Найти x где сорт(A) = x

или

Найти y где сорт(A) = x
и y = {(i u) : $x_i = u$}

все цели, содержащие предикат Demo, могут быть выполнены посредством рефлексии. Любое применение рефлексии требует, чтобы последовательность декодировалась в базу данных, определяющую эту последовательность. Метод декодирования, если конечно он корректен, сам по себе не влияет на не связанную с присваиванием логику программы сортировки. Однако самая прямая реализация рефлексии

также страдает от проблемы фреймов: следующие друг за другом базы данных загружаются в решатель задач объектного уровня посредством независимой декодировки их имен. Таким образом, предложения, принадлежащие обеим базам данных, приходится загружать дважды.

Однако логически оправданно использовать работу, выполненную при декодировании имени первой базы данных, для декодирования имени следующей за ней базы. Для того чтобы построить вторую базу данных, достаточно удалить из первой базы данных предложения, не принадлежащие второй, и добавить предложения, которые должны принадлежать второй базе данных, но не принадлежат первой. Это санкционирует использование деструктивного присваивания, выполняемого при реализации правила рефлексии и вне объектного языка и метаязыка (так что здесь не возникает помех для их логической семантики). Такая ситуация похожа на использование деструктивного присваивания в реализации рекурсивных структур данных, при которой распределение памяти зависит от числа ссылок. Более того, здесь действует сходное ограничение, что программа не содержит более ссылок на первую базу данных, так что ее содержимое можно без опасения разрушить.

Предлагаемые здесь решение проблемы фреймов и обоснование деструктивного присваивания неполны и носят характер наброска. К ним лучше относиться как к предварительному обсуждению возникающих проблем, а не как к окончательному решению.

Другой подход к встраиванию присваивания в рефлексию был предложен в беседах с Кейтом Кларком, Кейвом Эшги и Франком Маккейбом. Они предложили использовать присваивание в реализации множеств как абстрактных типов данных. Этот подход заслуживает дальнейшего исследования и сравнения с подходом, предложенным здесь.

8. ЗАКЛЮЧЕНИЕ

Из-за ограниченного объема статьи мне пришлось оставить в стороне много интересных и важных направлений. Я надеюсь, что читатель будет заинтересован в том, чтобы узнать побольше об этой быстро развивающейся области.

Благодарности

Я признателен Кейту Кларку, Мартену ван Эмдену, Стиву Грегори, Пэту Холлу, Питеру Хэммонду, Крису Хоггеру, Фрэнку Маккейбу, Мареку Серго и Владиславу Турскому

за ценные обсуждения и полезные комментарии к предварительным версиям статьи. Особую благодарность выражаю Сандре Эванс за напечатание многочисленных набросков статьи и окончательного экземпляра, пригодного для ксерокопирования.

ЛИТЕРАТУРА

[1] Bowen K. A., Kowalski R. A., Amalgamating language and metalanguage in logic programming//Logic Programming. — New York a. o., 1982. — P. 153—172.
[2] Burstall R. M., Darlington J. Transformation for developing recursive programs//J. ACM. — 1977. — V. 24, no. 1. — P. 44—67.
[3] Burstall R. M., MacQueen D. B., Sanella D. T. HOPE: an experimental applicative language//LISP Conference.— Stanford, 1980.— P. 136—143.
[4] Chakravarthy U. S., Minker J., Tran D. Interfacing predicate logic languages and relational databases//Logic Programming Conf. — Marseille, 1982. — P. 91—98.
[5] Clark K. L., Sickel S. Predicate logic: a calculus for deriving programs//5th Int. Joint Conf. on Artificial Intelligence. — Cambridge, 1977.
[6] Clark K. L. Negation as failure//Logic and Data Bases. — New York a. o., 1978. — P. 293—322.
[7] Clark K. L., McCabe F. G. PROLOG: a language for implementing expert systems//Machine Intelligence 10. — Ellis Horwood, 1980.
[8] Clark K. L., Gregory S. A relational language for parallel programming//Functional Programming and Computer Architecture. — New York, 1981.
[9] Clark K. L., Gregory S. PARLOG: a parallel logic programming language//London, 1985. — (Research Report/Imperial College, Dept. of Computing).
[10] Clark K. L., Ennals J. R., McCabe F. G. A Micro — PROLOG primer.— London: Logic Programming Associates Ltd.
[11] Clark K. L., Tärnlund S. — Å. A first order theory of data and programs//IFIP'77. — Amsterdam a. o., 1977. — P. 939—944.
[12] Logic programming//Ed. by K. L. Clark and S. — Å. Tarnlund. — New York a. o.; Academic Press, 1982.
[13] Clocksin W. F., Mellish C. S. Programming in Prolog. — Berlin a. o.: Springer, 1981. [Имеется перевод: Клоксин У., Меллиш К. Программирование на Прологе. М.: Мир, 1987.]
[14] Colmerauer A., Kanoui H., Pasero R., Roussel P. Une système de communication Homme-machine en Francais — Marseille, 1973. — (Research Report/Universite d'Aix Marseille).
[15] Colmerauer A. An interesting natural language subset//Logic Programming. — New York a. o.: Academic Press, 1982.
[16] Colmerauer A. Metamorphosis grammars//Lecture Notes in Computer Science. — Berlin a. o.: Springer, 1978. — V. 63: Natural Language Communication with Computers. — P. 133—189.
[17] Conery J. S., Kibler D. F. Parallel implementation of logic programs//Functional Programming and Computer Architecture. — New York: ACM, 1981.
[18] Darlington J., Reeve M. ALICE: a multi-processor reduction machine for the parallel evaluation of applicative languages//Functional Programming and Computer Architecture. — New York: ACM, 1981.

[19] Programming by refinement, as exemplified by the SETL representation sublanguage/Dewar R. B. K., Grand A., Liu S.-C., Schwartz J., Schonberg E.//ACM Transactions on Programming Languages and Systems. — 1979. — V. 1, no. 1. — P. 27—49.
[20] Van Emden M. H., Maibaum T. S. E. Equations compared with the clauses for specification of abstract data types//Advances in Database Theory. — New York: Plenum Press, 1981. — P. 159—194.
[21] Ennals J. R. Beginning Micro-PROLOG. — London: Heinemann Computers in Education, 1982.
[22] Hammond P. APES, A user manual. — London, 1982. — (DOC Report/ Imperial College; 82/9).
[23] Hayes P. J. Computation and deduction//Mathematical Foundations of Computer Science. — Prague: Czechoslovak Academy of Sciences, 1973. — P. 105—118.
[24] Hewitt C. PLANNER: a language for manipulating models and proving theorems in a robot//Int. Joint Conf. on Artificial Intelligence. — Washington, 1969.
[25] Hill R. Lush resolution and its completeness. — Edinburgh, 1974. — (DCL Memo./Univ. of Edinburgh; no. 78).
[26] Hogger C. J. Derivation of Logic Programs//J. ACM. — 1981. — V. 28, no. 2. — P. 372—392.
[27] International Conference on Fifth Generation Computer Systems: Proc./Int. Conf., Tokyo, 1979/Amsterdam a. o.: North Holland, 1981.
[28] Kowalski R. A. Predicate logic as a programming language//IFIP'74.— Amsterdam a. o.: North Holland, 1974. — P. 569—574.
[29] Kowalski R. A. Algorithm=Logic + Control//Communications ACM.— 1979.
[30] Kowalski R. A. Logic for problem solving. — New York: North Holland Elsevier, 1979.
[31] Kowalski R. A. Prolog as a logic programming language//AICA Congress, Pavia, Italy, 1981.
[32] Kowalski R. A. Logic as a computer language for children//European Conf. on Artificial Intelligence, Orsay, France, 1982.
[33] Pollard G. H. Parallel execution of Horn clause programs. — Imperial College, London. — 1981.
[34] Robinson J. A. A machine-oriented logic based on the resolution principle//J. ACM. — 1965. — V. 12. — P. 23—41. [Имеется перевод: Робинсон Дж. А. Машинно-ориентированная логика, основанная на принципе резолюции. — В кн.: Кнбернетич. сборник, вып. 7, 1970.]
[35] Roussel P. PROLOG: manuel de reference et d'utilisation. — Marseille: Universite d'Aix Marseille, 1975.
[36] Sergot M. A query-the-user facility for logic programming//ECICS/Ed. by P. Degano and E. Sanderall. — Amsterdam a. o., 1982.
[37] Turner D. The semantic elegance of applicative languages//Functional Programming Languages and Computer Architecture. — New York: ACM, 1981.
[38] Warren D. H. Higher-order extensions to PROLOG — are they needed?//Machine Intelligence 10. — Ellis Horwood, 1981.
[39] Weyhrauch R. Prolegomena to a theory of mechanized formal reasoning//Artificial Intelligence. — 1980. — V. 13. — P. 133—170.

ПРАКТИЧЕСКОЕ ИСПОЛЬЗОВАНИЕ ПРОЛОГА[1])

Б. ДОМЁЛКИ, П. СЕРЕДИ

Резюме. Пролог в последнее время становится все более популярным как многообещающий инструмент для создания систем решения задач. Его можно рассматривать также как практический язык программирования для некоторых классов прикладных задач. Для этих целей нужны промышленные реализации, от которых естественно требовать: эффективного выполнения программ, расширения возможностей языка, создания инструментов, поддерживающих дисциплину программирования (таких, как модульность), а также создания среды разработки программ. Описывается система МПролог, претендующая на то, чтобы удовлетворить большую часть этих требований.

1. ВВЕДЕНИЕ

Пролог появился около десяти лет назад. Его теоретической основой были идеи логического программирования, предложенные Р. Ковальским и П. Хейсом [1], [2], [3]. Первый интерпретатор Пролога был создан в Марселе в 1973 г. А. Колмероэ [4]. Большую часть следующего десятилетия Пролог жил обычной жизнью нового научного изобретения в области, связанной с вычислительными машинами: завоевывал популярность, собирал поклонников и пророков, появился специальный бюллетень [5], выделился ряд центров, где велись активные исследования по развитию и применению Пролога. Однако до самого последнего времени вся эта активность оставалась почти исключительно уделом академических кругов, не оказывая почти никакого влияния на мир «большой информатики», где едва ли кто-нибудь даже слышал слово «Пролог» (а если и слышал, то считал, что это либо пакет программ для математической статистики [6], либо компания, производящая изделия для микропроцессорной техники [7], но не язык логического программирования).

Эта ситуация начала меняться в начале 80-х годов. Самым впечатляющим событием в этом отношении было, конечно, объявление в октябре 1981 г. о японском проекте вычисли-

---

[1]) Dömölki R., Szeredi P. PROLOG in practice. Information Processing 83 (IFIP Congress), 1983, p. 627—636.

тельных систем пятого поколения [8], в котором Пролог (или, скорее, один из его будущих усовершенствованных вариантов) был выбран в качестве базового языка новой системы (связь Пролога с японским проектом анализируется в [9]).

С тех пор Пролог не только не сходит со страниц газетных статей и научных публикаций, относящихся к различным аспектам будущего информатики, но постоянно возрастает интерес и к его практическим применениям в самых разных областях информатики. (Так, например, Пролог вошел в «первую десятку» языков в популярном журнале по микрокомпьютерам [10]).

Эту быстро растущую популярность нельзя, однако, объяснить просто выбором, сделанным авторами японского проекта. Некоторые более глубокие причины, лежащие за этим явлением, можно подытожить следующим образом:

— использование языков сверхвысокого уровня как практических инструментов для создания программного обеспечения стало *необходимым* из-за ограниченных успехов других методов при решении проблем «кризиса программного обеспечения» и стало (практически) *возможным* благодаря быстрому росту эффективности доступных машинных ресурсов. Пролог же можно рассматривать как универсальный язык сверхвысокого уровня с хорошо определенной и математически (логически) обоснованной семантикой;

— исследования в области искусственного интеллекта достигли той стадии, когда их результаты могут широко применяться в решении реальных практических задач. Это стало ясно прежде всего при появлении *экспертных систем*, где основная работа состоит в выполнении логического вывода на основании фактов и правил, относящихся к области данной задачи. Пролог, будучи языком, основанным на логике, может считаться особенно подходящим для написания программ такого типа;

— вообще говоря, вычислительные машины все чаще используются для решения неарифметических задач и наиболее естественные примитивные операции и структуры управления для этого класса применений коренным образом отличаются от операций и структур, созданных главным образом для реализации численных алгоритмов. В этом отношении Пролог может обеспечить переход — в области языков программирования — от традиционного подхода к вычислениям (в смысле фон Неймана) к более общей архитектуре для решения задач.

Эти (и другие) возможности Пролога стали осознаваться широким кругом специалистов по информатике еще за неко-

торое время до объявления о японском проекте: в 1980—1981 гг. состоялись три международные конференции (одна в Венгрии [11] и две в США [12], [13]), и в них участвовали далеко не только поклонники Пролога. Интересно отметить, что на американской сцене Пролог завоевывает популярность в условиях сильной конкуренции с Лиспом, другим языком, имеющим прочный математический фундамент и более чем двадцатилетний опыт реализации и интенсивного использования в самых различных областях, включающих искусственный интеллект (ИИ), но не ограниченных только им. (Интересное сравнение американских и европейских усилий в этой области можно найти в [14].)

Так как Пролог, как говорилось выше, входит сейчас в моду, он может показаться «философским камнем», решающим все настоящие и будущие проблемы вычислений. Поэтому, чтобы пользоваться Прологом как практическим инструментом, нужно иметь ясное представление о его устройстве (разд. 2) и о тех областях применения (или ситуациях), в которых он является наиболее адекватным инструментом (разд. 3). Рассматривая с этой точки зрения достоинства и недостатки Пролога, можно сформулировать требования к его реализации, претендующей на действительно широкие практические приложения (разд. 4). Пример такой системы описан в разд. 5.

2. ЧТО ТАКОЕ ПРОЛОГ?

Чтобы пояснить, чем на самом деле является Пролог, нужно исследовать три проблемы (и три источника недоразумений):

1) Что происходит в Прологе: доказательство теорем или выполнение программы?

2) Является ли Пролог системой для решения задач или просто языком для записи программ (предназначенных для решения этих задач)?

3) Является ли Пролог действительно языком сверхвысокого (или довольно низкого) уровня?

(К сожалению, не существует ясного и однозначного ответа на эти вопросы, так как Пролог сам по себе является результатом компромиссов, позволяющих его использовать на практике.)

Чтобы исследовать связь между выполнением Пролог-программы и доказательством теорем, нужно вернуться к основной идее логического программирования. Идея эта чрезвычайно проста: спецификация задачи записывается как набор утверждений на языке логики первого порядка (точнее, на

некотором ограниченном фрагменте этого языка), а затем система доказательства теорем (эффективная для этого фрагмента) используется для получения (конструктивного) доказательства существования объекта, удовлетворяющего этой спецификации. В действительности текст Пролога

A: B_1, B_2, ..., B_k

где A, B_1, ..., B_k — отношения (возможно, с аргументами), может пониматься двумя способами:

— либо как логическое утверждение, что A истинно, если одновременно истинны утверждения B_1, B_2, ..., B_k;

— либо как определение процедуры (проверяющей правильность) A, утверждающее, что для того чтобы выполнить A, должны быть выполнены все процедуры B_1, B_2, ..., B_k.

Эти две интерпретации можно считать эквивалентными, что дает возможность рассматривать Пролог как чисто алгоритмический язык программирования. Существуют два аспекта, делающих такую алгоритмическую интерпретацию более детерминистской, чем «чистая» система доказательства теорем в логике первого порядка:

— процедуры (условия) B_1, B_2, ..., B_k должны выполняться (проверяться) в определенном порядке;

— в случае, когда существует более чем одно (альтернативное) определение одной и той же процедуры, и выполнение одного из них приводит к неудаче (т. е. одна из процедур, составляющих это определение, вернет значение «ложь»), должен существовать определенный порядок для выбора «следующего» альтернативного определения той же самой процедуры (для попытки его выполнения).

В случае обычного Пролога этот порядок определяется порядком соответствующих конструкций в тексте программы. В некоторых других системах логического программирования (см., например, [15]) применяются другие способы упорядочения, допускающие даже возможность динамического контроля со стороны пользователя. Отказ от этого определенного порядка ведет к параллельным реализациям систем логического программирования.

Механизм выполнения программы Пролога, определенный алгоритмической интерпретацией, может отличаться от того, что требует логика: в некоторых случаях возможность доказать истинность тех или иных утверждений может зависеть от формы их описания (т. е. Пролог может не обеспечить полноты при доказательстве теорем, см., например, [3]). Более того, процедуры могут обладать разного рода побочными эффектами, в этом случае порядок их выполнения (внутри

одного и того же определения) может существенным образом влиять на результат выполнения программы.

Естественно, эти алгоритмические аспекты могут широко использоваться, если при написании программы на Прологе учитываются все детали механизма выполнения программы. Однако хорошая практика программирования требует некоторой сдержанности при использовании таких особенностей (и хорошие реализации помогают этому, упрятывая эти особенности в конструкции с хорошо определенной семантикой).

Таким образом, на первый вопрос можно дать следующий несколько неопределенный ответ:

1) Текст Пролога выполняется как алгоритмическая программа с помощью хорошо определенного механизма, который можно — с некоторыми ограничениями — рассматривать как упрощенную систему доказательства теорем в логике первого порядка.

Это приближает нас и к ответу на второй вопрос: Пролог обеспечивает «встроенный» механизм доказательства теорем для поддержки системы решения задач (требуя от пользователя только определения этой задачи — с помощью логики первого порядка — и позволяя системе вырабатывать решение). Но, чтобы эффективно использовать этот механизм, нужно тщательно делать описание задачи, с учетом алгоритмических аспектов.

2) Пролог не является (автоматической) системой решения задач, а скорее языком прпрограммирования, особенности которого позволяют легко создавать системы такого рода.

Возможность описывать саму задачу, а не шаги ее решения («что» вместо «как»), считается одной из главных характеристик языков сверхвысокого уровня. В этом смысле уровень Пролога можно оценивать как очень высокий (с некоторой предосторожностью из-за того, что возможно, как указывалось выше, небольшое отличие (подразумеваемой) логической интерпретации от (реального) выполнения программы). Некоторые конструкции языка, необходимые для реализации механизма доказательства теорем в Прологе, такие, как сопоставление с образцом (pattern matching) или бектрекинг (backtracking), действительно можно отнести к очень высокому уровню. С другой стороны, однако, Пролог — в его сегодняшнем виде — содержит только такие языковые конструкции, которые необходимы для его работы на базовом уровне; в нем отсутствуют некоторые структуры управления высокого уровня (реализующие, например, универсальную квантификацию или работу с множествами), а также возможность определять сложные структуры данных (отличные

от достаточно изысканных форм записей, реализуемых термами Пролога).

Ситуация несколько усложняется, если мы вспомним, что «уровень» языка иногда измеряется его «расстоянием» от операций, поддерживаемых аппаратурой. В этом смысле Пролог, с одной стороны, определенно является языком очень высокого уровня, так как его примитивы существенно отличаются от операций сегодняшних машин. С другой стороны, проектируемая аппаратная поддержка Пролога как базового языка новой компьютерной архитектуры может сделать его *по определению* языком низкого уровня.

Все это можно сформулировать как ответ на третий вопрос следующим образом:

3) Пролог обладает некоторыми свойствами языков очень высокого уровня, и в то же время уровень его конструкций очень неоднороден.

Чтобы выразить эти несколько неопределенные ответы более простым способом, можно сказать:

1)—3) Пролог — это язык (программирования) совершенно нового типа.

3. ПРИМЕНЕНИЕ ПРОЛОГА

Анализ некоторых особенностей Пролога, данный в разд. 2, должен предостеречь от того, чтобы рассматривать Пролог как средство, одинаково удобное для всех применений. Его можно эффективно использовать для специфических типов задач и ситуаций (перечисленных в этом разделе), если язык и его среда реализованы с расчетом на практическое использование (см. разд. 4 и 5). Применения Пролога даже в данный момент слишком многочисленны, чтобы их можно было более или менее полно описать в этой статье. Сведения обзорного характера — не самые последние — можно найти в работах [16], [17], а также в библиографии сборника [18]. В этом разделе вначале будут перечислены некоторые особенности прикладных задач, делающие использование Пролога естественным, затем некоторые потенциальные области его применения, описанные в достаточно общих терминах.

3.1. Особенности задач

Особенности задач, делающие использование Пролога естественным, можно резюмировать следующим образом:

1) Задачи, в которых требуется некоторый вид *поиска*. Типичная ситуация, подходящая для применения Пролога, — это наличие некоторого набора условий, которым должен

удовлетворять объект. Поиск этого объекта производится с помощью механизма бектрекинга, встроенного в Пролог. Эффективность этого поиска можно увеличить, если выписать этот набор условий (вместе с некоторой управляющей информацией, см. п. 4.2) в подходящей форме, учитывающей механизм исполнения программы, принятый в Прологе.

Другими словами, недетерминированный поиск реализуется механизмом выполнения программы на Прологе, который позволяет уменьшить последствия комбинаторного взрыва, благодаря средствам управления этим поиском (вплоть до возможности написать «прямолинейную» программу такого поиска).

2) *Символьные вычисления* в широком смысле, включая не только операции обработки цепочек символов, но и достаточно сложные структуры вроде графов. Такого рода работа обеспечивается следующими свойствами Пролога:

— *сопоставлением с образцом,* позволяющим идентифицировать элементы структуры с определенными (синтаксическими) свойствами, не используя явно функции выбора или конструирования;

— *рекурсией,* позволяющей лаконично описывать операции на структуре в терминах операций на подструктурах;

— *синтаксисом,* позволяющим описывать *структуры данных* в удобной форме выражений, обладающих свойством операторного предшествования и вводить новые операции с нужными приоритетами (а также иметь — в большинстве реализаций — встроенные операции и специальную нотацию для наиболее часто используемых структур данных типа списков).

Из этого следует [19], что возможности символьных вычислений в Прологе сопоставимы с соответствующими возможностями Лиспа, а в комбинации с другими средствами языка могут и превосходить их.

3) Развитые и гибкие средства обращения к *базе данных.* Программу на Прологе можно рассматривать как (реляционную) базу данных, содержащую отношения, выражающие факты и правила, вместе с удобным способом

— делать *запросы* и

— *добавлять* новую информацию к базе данных.

В обоих случаях обеспечивается большая гибкость: в организации хранимой информации не нужно придерживаться какой-нибудь заранее принятой схемы. Все это, однако, относится только к базе данных (ее части), непосредственно обрабатываемой Пролог-программой (т. е. находящейся в оперативной памяти), в тех случаях, когда нужно обрабатывать

большие объемы данных, обычная система управления базой данных должна быть подходящим образом связана с системой, реализующей Пролог.

3.2. Области применения

Рассмотрев стороны языка, наиболее существенные с точки зрения потенциальных применений, укажем несколько конкретных областей, которые подходят для использования Пролога (заметим, что обычно это связано с теми свойствами языка, которые упомянуты в разд. 3.1):

а) Различные области *автоматизированного проектирования* (computer aided design), например:

— архитектура (здания, планы этажей и т. д.),

— химические соединения (лекарства, удобрения и т. д.),

— схемы (печатные платы, СБИС и т. д.).

Обычно задача здесь формулируется как некоторый вид поиска в смысле 1), включающий иногда сложные структуры данных (например, структуру молекул) и большое количество данных (например, правила, которым должно удовлетворять устройство объекта).

б) *Логический вывод* как одно из наиболее важных применений недетерминированного поиска. Поддержанный логической интерпретацией программ Пролога, он может использоваться во всех областях, включающих доказательства теорем, например синтез и верификацию программ, запросно-ответные системы, исследования по ИИ и т. д.

в) Задачи *управления базами данных*, когда база данных содержит не только простые факты, но также и некоторые правила, формулирующие законы, которым подчиняются эти факты («базы знаний»), и/или где требуется гибкость Пролога, позволяющая делать запросы и изменения в базе данных, не предусмотренные ее создателями (если программа на Прологе рассматривается как база данных, в этом случае утверждение любого типа может быть просто добавлено как новое правило Пролога, и допускаются самые разнообразные запросы, если ответ на них может быть логически выведен Прологом из содержимого базы данных).

г) *Обработка текстов естественного языка*, где могут быть эффективно использованы две особенности Пролога — символьные вычисления и логический вывод. (Первый марсельский интерпретатор был создан главным образом именно для таких применений, и с тех пор было опубликовано много теоретических и практических результатов, полученных в этой области, см., например, [20], [21].)

Пункты (б)—(г) можно рассматривать как главные компоненты *экспертных систем* [22], что подтверждает тезис,

высказанный в разд. 1 о том, что Пролог является «естественным» языком для создания экспертных систем.

д) *Моделирование* очень высокого уровня, когда свойства модели можно описать (логически интерпретируемыми) утверждениями Пролога. Механизм вычисления Пролога (включая бектрекинг и совмещение с образцом) можно расширить некоторыми специальными средствами, позволяющими описывать (квази)параллельные процессы:

— указание длительности тех или иных действий,

— отправление и прием сообщений,

— распределение ресурсов.

Такого рода система моделирования может быть построена на основе Пролога и может описывать некоторые виды параллелизма и понятие времени (T — PROLOG, см. [23]).

(е) *Быстрое прототипирование* как нужный инструмент проектирования и создания программ [24]. В некоторых областях очень важно за короткое время перейти от спецификации задачи к работающей программе. Это поддерживается как очень высоким уровнем механизмов Пролога, так и тем, что он обычно реализуется в качестве интерпретирующей системы (иногда вместе с программной средой, позволяющей писать и отлаживать программу в режиме диалога, см. разд. 4.4 и 5.1). Чтобы использовать «прототип» как основу для создания программы и иметь возможность получать окончательную (эффективную) программу в результате пошагового уточнения, в Пролог должны быть добавлены некоторые дополнительные механизмы, например абстрактные типы данных. Исследования в этом направлении, комбинирующие идеи венского метода создания программ [25] с Прологом, рассмотрены в [26].

4. ТРЕБОВАНИЯ К ПРИМЕНЯЕМОЙ НА ПРАКТИКЕ ПРОЛОГ-СИСТЕМЕ

Как указывалось в предыдущих разделах, Пролог начинает внедряться в область практического программирования. Растет число широко используемых реализаций (см., например, [27], [28], [29], а также разд. 5). Ниже перечисляются свойства, которыми должны обладать Пролог-системы, предназначенные для промышленного использования.

4.1. Эффективная реализация

Один из самых часто употребляемых аргументов против Пролога заключается в том, что он использует намного больше памяти и машинного времени, чем традиционные

алгоритмические языки. Это, безусловно, верно для ранних прямолинейных реализаций и в принципе является следствием того, что Пролог обладает небольшим числом универсальных, чрезвычайно мощных средств управления:

— механизмом (рекурсивного) *вызова процедур*, который одновременно служит средством реализации условий и циклов;

— обобщенным *сопоставлением с образцом* (унификация), которое служит механизмом для передачи параметров, обеспечивая функции выбора и конструирования для декомпозиции и создания объектов со сложной структурой;

— *бектрекингом*, обеспечивающим недетерминированный поиск и вид итерации «очень высокого уровня».

Прямолинейная реализация таких средств управления удивительно проста. Можно использовать, как это сделано в первом марсельском интерпретаторе, единственный стек, выполняющий целый ряд функций типа хранения управляющей информации для организации рекурсии и бектрекинга или резервирования памяти для новых конкретизаций переменных и т. п. Простота реализации — одна из причин, объясняющих, почему Пролог мог распространиться относительно быстро. Для ряда применений даже простая реализация оказалась достаточно мощной [17].

Прямолинейный подход обладал, однако, целым рядом дефектов, связанных с требованиями к пространству и времени. Так, например, при выходе из процедуры нельзя было претендовать на какое-нибудь место в стеке (как в традиционных языках), так как при бектрекинге нужно было исследовать альтернативные ветви данной процедуры. При этом каждый вызов является потенциальной точкой бектрекинга.

Во второй половине 70-х годов был разработан целый ряд более тонких методов реализации. Самой значительной из них была реализация Д. Уоррена [30].

Одним из самых главных улучшений было *управление памятью*. Например, благодаря двум новым механизмам, так называемым *выталкиванию из стека* при успехе (success popping) и *оптимизации хвостовой рекурсии* (tail recursion optimisation), схема управления памятью в Прологе не уступает традиционным языкам программирования, по крайней мере для случая детерминированных процедур[1]). Оба эти механизма основаны на том, что стек разделяется на несколько (под)стеков. В случае успешного выталкивания из стека так называемый главный стек освобождается при выходе из

[1]) См. статью М. Бранохе в настоящем сборнике. — *Прим. ред.*

процедуры, так как внутри ее не осталось больше точек бектрекинга. Оптимизация хвостовой рекурсии позволяет освобождать стековое пространство процедуры даже раньше: во время самого последнего вызова процедуры из ее тела. Это просто означает, что нет издержек, связанных с реализацией циклов посредством рекурсии.

Другим средством повышения эффективности (одновременно пространства и времени) является *индексирование* процедур. Это ускоряет поиск подходящей альтернативы благодаря предварительному изучению значений определенных аргументов и обращению к подпроцедурам, соответствующим этим значениям. Это экономит время, которое пошло бы на неудачные унификации, и, кроме того, в результате может оказаться, что процедура детерминирована (тем самым можно использовать оптимизацию, описанную в предыдущем абзаце). В некоторых Пролог-системах схемы индексирования вырабатываются автоматически для фиксированных аргументов в процедурах со многими альтернативами, в других системах пользователь может указать, для каких аргументов эта схема может применяться.

Пользователь может предоставлять и другие данные для оптимизации: например, в так называемых *декларациях видов* можно выделять некоторые аргументы для использования их только в качестве входных или выходных параметров процедур.

Главным достижением в технике реализации Пролога было создание в 1977 г. Д. Уорреном *компилятора* [30] (для системы DEC-10). Это доказало, что Пролог может быть реализован так же эффективно, как и другие языки ИИ [19].

Основная идея уорреновского метода компиляции заключалась в специализации общего алгоритма унификации для различных типов заголовков правил и в выработке в каждом случае такой последовательности машинных команд для унификации, которую можно предсказать по данному типу заголовка. Конечно, оставались случаи, когда приходилось применять общий алгоритм унификации, однако, как правило, унификацию можно было значительно упростить, например когда она использовалась просто для передачи параметра процедуры (заголовок с единственной переменной в качестве аргумента), в этом случае она компилировалась в единственную команду присваивания.

4.2. Введение средств, лежащих за пределами чистой логики

Одним из основных средств, которые помогли Прологу стать «реальным» языком программирования, было введение в язык элементов, лежащих за пределами *чистой* логики.

Большинство этих элементов в современных Пролог-системах реализуются как *встроенные предикаты.*

Все реализации Пролога предлагают встроенные предикаты в ряде областей: арифметике, обработке цепочек символов, вводе и выводе, модификации программы, управлении ходом вычислений. Некоторые из встроенных предикатов в принципе определяются средствами логики, например *арифметические* или предикаты *обработки цепочек символов.* Однако они делаются встроенными просто потому, что их логическая реализация была бы значительно медленнее, чем машинная. Необходимость в другой группе встроенных предикатов (для *ввода-вывода*) вызвана тем, что программы должны быть связаны с внешним (по отношению к машине) миром.

Практическая реализация Пролога должна обеспечить достаточно богатый набор встроенных предикатов в обеих группах, например, программы символьных вычислений часто пользуются процедурами разрезания цепочек символов и поиска подцепочек, почти все реальные программы требуют процедур вывода самых разных форматов.

Предикаты ввода-вывода неизбежно приводят к *побочным эффектам* и тем самым лежат за пределами логики. Существует, однако, возможность реализовать версии таких предикатов, допускающие бектрекинг, т. е. бектрекинг позволяет «ликвидировать» связанные с этими предикатами побочные эффекты. Это может быть полезным, например, в случае ввода, когда для обработки одного и того же входного потока используется несколько альтернатив процедуры.

Предикаты *модификации программы* служат для добавления правил к программе и удаления их из программы во время ее выполнения. В принципе это дает возможность написать самомодифицирующиеся процедуры, однако практика показывает, что собственно программа четко отделяется от модифицируемых частей, последние в большинстве случаев являются просто базой данных для фактов. С другой стороны, возможность модифицировать любой предикат в программе важна для реализации средств разработки программ в самом Прологе. Во многих случаях использования встроенных предикатов можно избежать изменения программы, выделяя их в предикаты высших типов, например в предикат «множество всех решений» [31].

Больше всего споров вызывает группа встроенных предикатов, относящаяся к *управлению* ходом вычислений. В этой группе существует главный предикат, так называемый оператор *усечения* (*cut* или *slash*), при выполнении которого

отсекаются ветви дерева поиска благодаря тому, что выбрасываются оставшиеся альтернативы в выполняемой в данный момент процедуре. Это можно использовать для повышения эффективности, не меняя смысла программы (когда программист знает, что ни одна из отбрасываемых ветвей не приведет к успеху), но, как правило, оператор усечения сознательно используется для изменения смысла программы. И снова в ряде случаев он может — и должен — быть выделен в некоторые конструкции более высокого уровня типа отрицания или «если-то-иначе» [32]. Предлагалось использовать вместо оператора усечения другие, более декларативные средства, например возможность объявлять, что процедура детерминирована (т. е. после успешного выхода из процедуры не должно быть бектрекинга).

В этой области ведется много исследований, предлагаются специальные средства для управления ходом вычислений (см., например, [33]), строятся специальные метаязыки для этой цели, например [34]. Некоторые средства такого типа уже используются в ряде реализаций Пролога, например оператор *замораживания* (freeze operation), позволяющий отложить вызов процедуры до тех пор, пока переменная не получит значения [35]. В репертуар реализаций Пролога включаются также некоторые традиционные механизмы управления, такие, как обработка исключительных ситуаций (например, ошибок или прерываний).

В ряде ситуаций набор встроенных предикатов должен быть увеличен. Некоторые прикладные программы могут требовать специальных встроенных процедур как для реализации некоторых специфических алгоритмов данной проблемной области, так и для обеспечения доступа к различным машинным ресурсам. Может возникнуть необходимость введения новых предикатов для настройки, т. е. для замены наиболее часто используемых алгоритмов, запрограммированных первоначально на Прологе, специальными встроенными предикатами. Для этого практическая реализация Пролога должна обеспечить удобный способ расширения набора встроенных предикатов.

Другой, более гибкий путь реализации алгоритмов, написанных не на Прологе, состоит в создании *интерфейса* с другими традиционными языками программирования. В этом случае некоторые части программы могут быть написаны на алгоритмическом языке и их скомпилированный код может быть присоединен к части программы, написанной на Прологе.

4.3. Поддержка дисциплины программирования

В 70-е годы для того, чтобы преодолеть кризис в разработке программных средств, был разработан целый ряд приемов структурного программирования. Хотя очень высокий уровень Пролога уменьшает размер программ, все же для реальных задач могут понадобиться программы размером в несколько тысяч строк. Практическая реализация Пролога должна иметь средства для структурирования таких программ, т. е. должна позволять разбивать программы на модули.

Введение в Пролог понятия *модуль* ставит целый ряд задач. Обычные спецификации *экспорта-импорта* могут управлять *видимостью* (visibility) имен тех или иных процедур, но необходимо управлять и видимостью *имен данных*. Эти проблемы не независимы, так как все реализации Пролога обеспечивают возможность преобразования структуры данных в вызов процедуры. В Прологе могут также понадобиться особые средства, чтобы решать, нужно ли хранить то или иное имя в *символьном* виде во время выполнения программы. Обсуждение этих проблем и обзор предлагавшихся решений содержатся в [36].

Удобочитаемость и надежность Пролог-программы можно увеличить также введением различных форм избыточности. Многие реализации, например, вводят *декларации видов* (см. разд. 4.1). Следующим шагом могло бы стать введение *типов данных* и информации о типах для процедур. Подобные средства изучались в некоторых экспериментальных реализациях [37]. Расширения такого рода могли бы также повысить эффективность, так как информация о типах упрощала бы унификацию компилируемой программы [38].

4.4. Среда для разработки программы

Хорошо известно, что наличие развитой *среды для разработки программ* значительно увеличивает практическую пригодность языка программирования. Интерактивная среда используется на разных стадиях разработки программы: при написании, отладке, редактировании модулей программы и их интеграции. Кроме помощи в индивидуальной работе программиста развитая среда поддерживает также организацию процесса разработки программы при разделении труда, коммуникации между участниками и т. д.

Так как Пролог является интерпретируемым и интерактивным (диалоговым) языком, то большинство его реализаций обеспечивают инструменты для интерактивной *отладки*,

а некоторые имеют также средства для *редактирования программ*. Пролог-система с развитой подсистемой разработки программ описана в разд. 5.1, где приведена детальная картина возможностей и использования программной среды для Пролога.

5. СИСТЕМА МПРОЛОГ

Система МПролог [41] является попыткой создать реализацию Пролога, пригодную для практических применений в больших масштабах и по крайней мере частично удовлетворяющую требованиям, перечисленным в предыдущих разделах. МПролог является наследником первой венгерской реализации Пролога, созданной в 1975 г. [42]. «Старый» Пролог опирался в основном на марсельскую модель и был использован в многочисленных экспериментальных, а также нескольких промышленных приложениях [17]. Работа по проекту МПролог началась в 1978 г. с целью создать совершенно новую реализацию, соединяющую опыт эксплуатации с усовершенствованными методами реализации (описанными выше в разд. 4.1).

Система МПролог предназначена для того, чтобы удовлетворить двум противоположным стремлениям: поддержать процесс разработки программы и помочь в создании Пролог-программ, пригодных для промышленной эксплуатации. Чтобы обеспечить дружескую для пользователя среду при создании программ, была построена специальная подсистема разработки программ (PDSS — Program Development SubSystem) [43], которая сама была написана на МПрологе [1]).

5.1. Разработка программы на МПрологе

На рис. 1 приведен образец диалога с помощью которого можно представить основные особенности PDSS. Текст, вводимый пользователем, выделен *курсивом*. Ниже дан краткий комментарий по поводу этого диалога, номера строк соответствуют номерам на рисунке.

Строка 1. Принимается решение начать разработку модуля обработки списков с именем list_handling (если бы эта строка была опущена, все последующие предикаты вводились бы в unnamed_module — безымянный модуль).

---

[1]) МПролог является названием входного языка системы МПролог.— *Прим. ред.*

Строки 2, 3. Вводятся два правила, определяющие предикат length (длина). Система PDSS подтверждает это, выдавая на экран имя/арность для предиката + номер правила. Последнее введенное правило становится текущим «фокусом» внимания. Заметим, что имена команд могут быть сокращены (например, «*e*» вместо «enter»).

```
(1) module list_handling.
    MODULE list_handling.
(2) enter length (nil,0).
    length/2
(3) e length (X,L,N):— lengh (L,N1), N is N1 + 1.
    length/2 + 2
(4) type
    length (X.L,N):—
        lengh (L,N1) N is N1 + 1.
(5) ? length (1.2.nil,N).
    Exception —505: undefined predicate
      In call of lengh (2.nil,_104)
    FUNCTION (h for help)?
(6) b
    lengh (2.nil,_104)
    length (1.2.nil,_93)
    FUNCTION (h for help)?
(7) a
    Execution aborted
(8) edit
     10: length (X.L,N):—
     20:     lengh (L,N1),
     30:     N is N1 + 1
    ***Enter editor commands***
(9) 20:     length (L,N1),
    ***Line 20 replaced***
(10) end
    GLAUSE length/2 + 2 REPLACED.
(11) ? length (1.2.nil,N).
        N=2
    CONTINUE? (y/n)
(12) save myfile
    SAVING list_handlling
           length/2
    SAVED
```

Рис. 1. Диалог с PDSS (первая часть).

Строка 4. Команда type (печатай)[1]) служит для расположения указанной части программы в стандартном «красивом» формате. Если не указан параметр, печатается текущий фокус. Но можно использовать команды type Predicate или type Module, чтобы выдать весь предикат или весь модуль.

Строка 5. «?»— команда для выполнения указанного вызова процедуры и демонстрации результирующих подстано-

[1]) Здесь и ниже слово «печатай» относится к «печати» на дисплее,

вок. В процессе выполнения второго предложения для length встречается обращение к неопределенному предикату lengh (из-за опечатки в строке 3), вызывающее обработку исключительной ситуации. Программа обработки исключительных ситуаций в системе PDSS печатает сообщение об ошибке, ошибочный вызов и спрашивает пользователя, что делать (возможности: отобразить обратное прослеживание — продолжать, признать неудачей, прекратить или войти на другой уровень PDSS). Пользователь может также определить свою собственную процедуру обработки исключительных ситуаций, например newhandler (undefined, fail), в которой случай обращения к неопределенному предикату просто считается неудачей. Запись _104, появившаяся в напечатанной форме ошибочного вызова является внутренним именем (конкретизации) переменной N1. Заметим, что команда type выдает исходные имена переменных (как в строке 4).

Строка 6. Пользователь требует выдать обратное прослеживание (backtrace), т. е. текущего списка вызовов.

Строка 7. Выполнение прекращено (aborted).

Строка 8. Команда редактирования (edit) без параметров требует редактирования предложения, находящегося в текущем фокусе. На экране печатается предложение, разбитое на строки.

Строка 9. Пользователь вводит исправленную строку. Заметим, что нет необходимости перепечатывать всю строку, так как для большинства операционных систем МПролог обеспечивает средства экранного редактирования.

Строка 10. Редактирование завершено.

Строка 11. Следующая попытка запустить предикат length на этот раз успешная. Команда «?» печатает подстановку N = 2 и спрашивает, нужно ли продолжать поиск альтернативных решений. Ввод новой команды (в строке 12) эквивалентен ответу «нет».

Строка 12. Сохранить текущее состояние модуля в файле myfile.

Нужно упомянуть две другие группы команд, не представленные в этом примере: команды *трассировки* и *фокусировки.* С помощью команд трассировки можно предписать (либо отменить) трассировку одного или нескольких предикатов (или всех предикатов модуля). Команда фокусировки просто устанавливает фокус на заданное предложение, например, команда «focus length + 2» сдвинет фокус на второе предложение предиката length.

5.2. Модульность в МПрологе

Модуль МПролога состоит из двух основных частей, спецификации *интерфейса* и *тела*, расположенных следующим образом:

```
module⟨имя модуля⟩.
⟨спецификация интерфейса⟩
body.
⟨предложения и декларации⟩
endmod.
```

Задача спецификации интерфейса — указать *видимость* и *продолжительность жизни* (life time) имен (как для имен процедур, так и для имен структур данных). При отсутствии этой спецификации по умолчанию предполагается, что видимы только имена стандартных процедур и структур данных (вроде «Nil» и «.»). Имя может быть сделано видимым вне модуля с помощью

— спецификации *visible* (для имен данных),

— спецификации *export* (для процедур, определенных в модуле) или

— спецификации *import* (для процедур вызываемых, но не определенных).

Можно также использовать спецификацию *all_visible*, указывающую, что все имена видимы вне модуля.

Продолжительность жизни имен касается того, существенна или нет символическая форма имен (последовательность литер, из которой составлено имя). Для имен, специфицированных указанием *symbolic*, символическая форма существенна, все другие имена могут потерять символическую форму в процессе оптимизации программы. Опять-таки при отсутствии спецификации по умолчанию сохраняют символическую форму только стандартные имена, но существует способ сделать имя символическим «на месте», т. е. в месте употребления, просто заключив его в двойные кавычки. Можно также пользоваться и спецификацией *all_simbolic* («все символические»).

Система PDSS также помогает снабжать модуль всеми необходимыми спецификациями интерфейса (export/import) и различными декларациями (например, декларациями видов).

На рис. 2 приведен второй образец диалога с PDSS, в котором завершается обработка модуля list_handling.

Строка 1. Читается модуль, сохраненный в предыдущем сеансе.

Строка 2. Предикат length снабжается спецификацией export. Заметим, что имя предиката is и имя данных +, встречающиеся в обращении N is N1 + 1 второго предложения

```
(1) read myfile
    myfile READ.
(2) export length/2.
    EXPORT.
(3) mode length (+,—)
    MODE.
(4) type MODULE
    module list_handling.
    export (length/2).
    /*Seject*/
    body.
    mode length (+,—).
    length (nil,0).
    length (X.L,N):—
        length (L,N1), N is N1 + 1.
    endmod /*list_handling*/.
(5) save myfile
    SAVING list_handling
            FACES...
            length/2
    SAVED
```

Рис. 2. Диалог с PDSS (вторая часть).

length, не нуждаются в спецификациях import или visible, так как они стандартны и видимы по умолчанию.

Строка 3. Добавляется декларация вида, предписывающая, что первый аргумент предиката length должен использоваться как входной, а второй — как выходной.

Строка 4. Печатается весь модуль. Заметим, что PDSS располагает всю введенную информацию в стандартном порядке.

Строка 5. Модуль сохраняется в файле myfile.

5.3. Производство программ для эффективного исполнения

До сих пор мы иллюстрировали процесс создания программы в МПрологе. Во время этого процесса некоторое число модулей может быть создано, отлажено, а также снабжено дополнительной информацией, относящейся к видимости и оптимизации, как в нашем маленьком примере.

Эти модули могут теперь обрабатываться другими компонентами МПролога для получения оптимизированной программы. Перечислим основные цели этого процесса:

— Модули подвергаются оптимизации, для чего используется ряд методов, рассмотренных в разд. 4.1. Цель оптими-

зации — получить более эффективный код Пролог-программы с точки зрения объема памяти и времени.

— Результатом этого процесса является Пролог-программа в так называемой *двоичной форме*, т. е. внутренней форме, реально используемой интерпретатором. Это значит, что процесс преобразования исходной Пролог-программы во внутреннюю форму выполняется только один раз, что экономит много времени при загрузке программы.

— Единственная компонента системы, необходимая для выполнения результирующей двоичной программы, — это интерпретатор. Такие *обособленные* (*stand — alone*) программы могут выполняться независимо от самой системы МПролог (без возможности модифицировать исходную программу) аналогично программам, производимым традиционными компиляторами.

Диаграмма производства бинарной МПролог-программы показана на рис. 3. Роль каждой компоненты системы рассмотрена ниже.

Претранслятор читает исходный модуль МПролог-программы, проводит лексический анализ, синтаксический анализ, статическую семантическую проверку, оптимизирует и создает компактную внутреннюю форму модуля (имена, не включенные в связи модуля, кодируются и теряют свою символическую форму, что экономит место в таблице имен). Эта внутренняя форма называется двоичным модулем. Претранслятор также составляет полные словари перекрестных ссылок для документирования модуля.

Консолидатор — это редактор связей на уровне Пролога: он связывает двоичные модули, из которых состоит полная программа, в интерпретируемую двоичную программу, он может также объединять набор модулей в один новый двоичный модуль, имеющий (новую) определенную пользователем спецификацию интерфейса. Консолидатор составляет также словари перекрестных ссылок.

Интерпретатор выполняет одну цель, указанную в главном модуле.

Стандартный двоичный модуль содержит определения стандартных (встроенных) процедур МПролога. Большая часть из этих процедур определена «нелогическим» способом, т. е. выполняется интерпретатором как непрологовый код программы, другие определены в самом МПрологе. Стандартный модуль автоматически присоединяется к каждой двоичной программе.

Система PDSS, как уже раньше говорилось, сама написана на МПрологе — таким образом, выполнение PDSS означает запуск интерпретатора, выполняющего двоичную про-

грамму, представляющую саму PDSS. Более того: PDSS доступна также в форме двоичного модуля. Это означает, что вместо изготовления обособленной программы из набора претранслированных модулей, можно объединять их с PDSS, чтобы получить одну двоичную программу. Эта двоичная

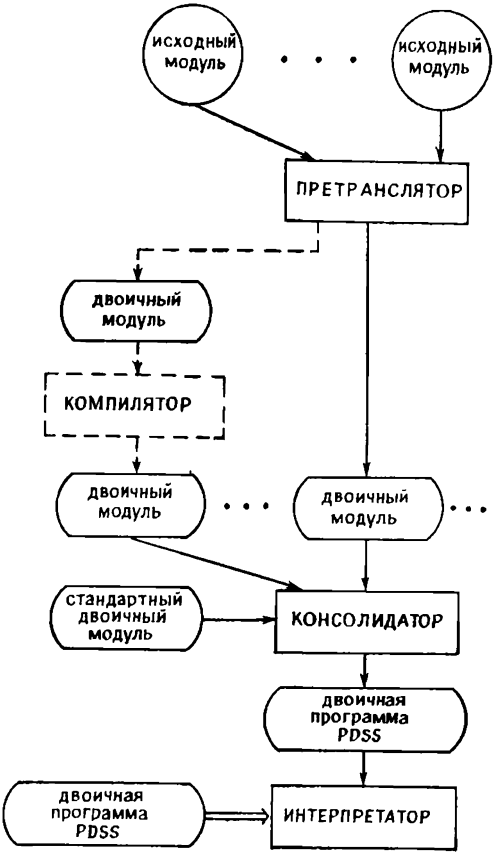


Рис. 3. Структура системы МПролог.

программа ведет себя как обычная PDSS, но уже загруженная всеми модулями пользователя с их интерфейсом экспорта/импорта и готовая для выполнения и трассировки. Чтобы внести изменения в модуль пользователя, можно пользоваться специальным встроенным предикатом для удаления претранслированной формы этого модуля и, следовательно, читать его в исходной форме, что позволяет использовать все средства PDSS для этого модуля, сохраняя возможность проверки всей программы.

Система МПролога в настоящее время расширяется — в нее вводится *компилятор*. Компилятор — это расширение претранслятора, так как он работает с двоичными модулями и его результатами также являются двоичные модули с той же самой структурой и внешним интерфейсом, причем их интерпретируемое внутреннее представление в пределах этого модуля заменяется последовательностью машинных команд. Этот подход позволяет свободно связывать компилируемый и интерпретируемый коды.

5.4. Интерпретатор МПролога

Интерпретатор МПролога — это сердце всей системы МПролог. Он использует алгоритм интерпретации с соответствующей структурой стеков, в основном аналогичный компилятору Д. Уоррена [30]. Предоставляется расширенная схема индексации: большую процедуру можно представить в виде дерева подпроцедур, соответствующих конкретным значениям определенных аргументов, — это задается пользователем в так называемых *декларациях порядка совмещения* (*match-order* declarations).

Интерпретатор обеспечивает два типа обработки исключительных ситуаций. Механизм локальной обработки позволяет пользователю определить процедуру обработки (в самом МПрологе), которая будет подставляться в вызов, порожденный исключительной ситуацией. Кроме того, существует механизм защиты исключительных ситуаций, позволяющий вылавливать все необработанные исключительные ситуации, встречающиеся в ходе выполнения данного вызова. Интерпретатор содержит определения более чем 200 *непрологовских процедур*, включающих практически все независимые от машины предикаты версии Пролога для DEC-10, т. е. все описанные в учебнике по Прологу [44]. Язык МПролог уникален в том смысле, что он обладает тщательно разработанным набором процедур для вывода в различных форматах и для обработки цепочек символов, а кроме того, «допускающими бектрекинг» входными процедурами и процедурами модификации программ. Имеются также специальные непрологовские процедуры (недоступные обычному программисту), разработанные специально для PDSS, например доступ к самому стеку выполнения для показа информации о бектрекинге.

Интерпретатор содержит хорошо определенный *интерфейс* для удобного включения новых непрологовских процедур. Наличие этого интерфейса делает добавление новой непрологовской процедуры (написанной на CDL2, языке реализации

МПролога) делом нескольких часов. Действительно, ряд новых процедур был добавлен в ходе разработки некоторых прикладных программ [45].

Есть и другое расширение, доступное широкому кругу пользователей и позволяющее вызывать подпрограммы, написанные на других традиционных алгоритмических языках (Фортране, Коболе, ассемблере). В настоящее время эта возможность предоставлена только в некоторых операционных системах, так как она использует динамическое связывание модулей на «иностранных языках».

5.5. О реализации системы МПролог

Система МПролог реализована на языке программирования CDL2 [46] и самом МПрологе. Базисная структура управления в CDL2 очень похожа на соответствующую структуру Пролога, и на самом деле этот язык можно рассматривать как упрощенный Пролог. Это очень удобно, когда одна и та же группа лиц использует оба языка.

Важной особенностью CDL2 является то, что, будучи *открытым* (*open-ended*) языком (т. е. дающим прямой доступ к уровню машинного кода), он прекрасно сочетает переносимость с эффективностью. Удобство переноса достигается сравнительно небольшим числом машинно-независимых примитивов, а эффективность — использованием метода настройки (tuning), позволяющим переписать наиболее критические процедуры (в интерпретаторе МПролога относящиеся в основном к унификации) в коде низкого уровня.

Система МПролог в настоящее время доступна для машин VAX-11/VMS, IBM/CMS и SIEMENS/BS 2000. Разрабатывается несколько проектов по переносу МПролога на другие большие машины, а также создаются (ограниченные) версии для микроЭВМ [1]).

6. ЗАКЛЮЧЕНИЕ

Пролог становится практическим языком программирования, пригодным для эффективного использования в целом ряде прикладных областей. Нацеленные на это реализации должны включать средства, обеспечивающие разработку и эффективное выполнение больших и сложных Пролог-программ. Растущее число таких реализаций и их широкое применение готовят почву для будущего развития и использования Пролога, особенно в следующих областях:

[1]) В настоящее время существуют реализации МПролога на машинах серий ЕС и СМ. — *Прим. перев.*

— Реализация на *персональных компьютерах* — все увеличивающийся объем и эффективность (с точки зрения стоимости) аппаратных ресурсов, доступных для этого класса машин, дают возможность создавать мощные реализации Пролога. В этой сфере Пролог можно использовать для создания по-настоящему гибких, удобных прикладных программ для человеко-машинного общения и развитых средств работы с базами данных.

— *Методология программирования* — очень высокий уровень конструкций Пролога может быть полезен для создания инструментов проектирования и разработки программ.

— *Новые типы архитектуры* для логического программирования, поддерживающие реализацию Пролога и его расширения.

Широкое использование Пролога (особенно на персональных машинах) может также изменить общие представления о вычислениях: образ компьютеров как декларативно программируемых разумных машин, обладающих способностью решать задачи, постепенно заменит традиционный образ алгоритмически программируемых вычислительных устройств.

Благодарности

Авторы благодарны П. Кёвишу, Е. Шантане-Тот и другим участникам проекта МПролог.

ЛИТЕРАТУРА

[1] Kowalski R. A., Predicate Logic as Programming Language, Pr. Proc. IFIP 74, North Holland, Amsterdam, 1974, 569—574.
[2] Hayes P. J. Computation and deduction, Proc. 2nd MFCS Symp., Czechoslovak Academy of Sciences, 1973, 105—118.
[3] Kowalski R. A. Logic for problem solving. North Holland Elsevier, 1979.
[4] Colmerauer A., Kanoui H., Pasero R., Roussel P. Un Systeme de Communication Homme-machine en Francais, Research report, Groupe Intelligence Artificielle, Université Aix Marseille, Luminy, 1973.
[5] Pereira L. M. ed., Logic Programming Newsletter, No. 1—4, Universiedade Nova de Lisboa, 1981—1983.
[6] PROLOG Statistikprogramme im Dialog, ISIS Software Report, July 1981.
[7] Microprocessor User's Guide, Pro-Log Corporation, Monterey, Ca., July 1979.
[8] Moto-Oka T. ed., Proc. of International Conference on Fifth Generation Computer Systems, North Holland, 1982.
[9] Warren D. A view o the Fifth Generation and its impact, SRI International Technical Note 265, July 1982.
[10] Practical Computing, April 1983.
[11] Tärnlund S.-Å. ed., Proc. of the Logic Programming Workshop, Debrecen, Hungary, 1980.

[12] Logic Programming Workshop, Syracuse University, USA, 8—10 April, 1981.
[13] Workshop on Logic Programming for Intelligent Systems, Long Beach, California, USA, 19—21, August 1981.
[14] McDermott D. The PROLOG Phenomenon, ACM SIGART Newsletter no. 72, July 1980, 16—20.
[15] Clark K. L., McCabe F. G., Gregory S. IC — PROLOG Language Features, in [18] 253—266. [Имеется перевод: Кларк К., Маккейб Ф., Грегори С. Средства языка IC-PROLOG, см. наст. сборник.]
[16] Coelho H., Cotta J. C., Pereira L. M. How to solve it with PROLOG, 2nd edition, Laboratorio Nacional de Engenharia Civil, Lisboa, Portugal, 1980.
[17] Sántáné-Tóth E., Szeredi P. PROLOG applications in Hungary, in [18], 19—31.
[18] Clark K. L., Tärnlud S.-Å. ed., Logic Programming, Academic Press, London, 1982. [Имеется перевод 5-статей: см. наст. сборник.]
[19] Warren D., Pereira L. M., Pereira F. Prolog — the language and its implementation compared with Lisp, SIGPLAN Notices 12, No. 8, 1977, 109—115.
[20] Dahl V. Quantification in a three-valued Logic for Natural Language Question-Answering Systems, Proc. 6th IJCAI, 1979, 182—187.
[21] Pereira F., Warren D. Definite clause grammars for language analisis — A survey of the formalism and a comparison with augmented transition networks. Artificial Intelligence, 13, 1980, 231—278.
[22] Feigenbaum E. A. Innovation and Symbol Manipulation in the Fifth Generation Computer System, in [8], 223—226.
[23] Futó I., Szeredi J. A discrete simulation system based on artificial intelligence methods, in Discrete Simulation and Related Fields, North Holland, 1982.
[24] Squires S. L., Zelkowitz M., Branstad M. Rapid Prototyping Workshop: An Overview. ACM Software Engineering Notes, 7, 1982. No. 3. 14—15.
[25] Bjorner D., Jones C. B. eds. The Vienna Development Method: The Meta-Language. LNCS 61, 1978.
[26] Farkas Zs., Szeredi P., Sántáné-Tóth E. LDM — A program specification support system. Proc. of the First International Logic Programming Conference, Marseille, 1982, 123—128.
[27] Bowen D. L., DECsystem — 10 Prolog User's Manual. Department of Artifitial Intelligence. University of Edinburgh, 1981.
[28] Clocksin W., Mellish C., Fisher R. The RT — 11 Prolog System, Software Report 5a, Department of Artifitial Intelligence, University of Edinburgh, 1980.
[29] McCabe F. G. micro-PROLOG Programmer's Reference Manual Logic Programming Associates Ltd., London, 1981.
[30] Warren D. Implementing PROLOG — Compiling Logic Programs. vol. I and 2. D. A. I. Research Report No 39—40, University of Edinburgh, 1977.
[31] Warren D. H. Higher-order Extentions to PROLOG — Are they needed?, Machine Intelligence 10, Ellis and Horwood, 1981.
[32] Kowalski R. Logic Programming, IFIP 83, 133—145. [Имеется перевод: Р. Ковальский. Логическое программирование. См. наст. сборник.]
[33] Clark K. L., McCabe F. The control facilities of IC — PROLOG, in Expert Systems in the Micro-Electronic AGe, Edinburgh University Press, 1979.
[34] Gallaire H., Lasserre C. Metalevel Control for Logic Programs, in [18], 173—185.

[35] Colmerauer A., Kahoui H., Caneghem M. Last steps toward an ultimate. Prolog, Proc. 7th IJCAI, Vancouver, 1981.
[36] Szeredi P. Module Concepts for PROLOG, Proc. of the Workshop on PROLOG Programming Environments, Linköping, March 1982.
[37] Shapiro E. Algorithmic Program Debugging, PhD Thesis, Yale University, USA, 1982.
[38] Nillson J. F. On the Compilation of a Domain-based PROLOG, Technical Note, Dept. of Computer Science, Technical University of Denmark, Dec. 1982.
[39] Cheatham T. E. Jr., Comparing Programming Support Environments, in Software Engineering Environments, North Holland, 1981, 97—118.
[40] Bayer M. et al. Software Engineering Environments, North Holland, 1981, 97—118.
[41] MPROLOG Language Reference Manual & MPROLOG Users' Guide, Institute for Co-ordination of Computer Techniques, Budapest, Hungary, 1982.
[42] Szeredi P. PROLOG — a Very High-Level Language Based on Predicate Logic, in Preprints of the Second Hungarian Computer Science Conference, Budapest, 1977, 853—866.
[43] Köves P. The MPROLOG Programming Environment: Today and Tomorow, Workshop on PROLOG Programming Environments, Linköping, March 1982.
[44] Clocksin W. F., Mellish C. S. Programming in PROLOG, Springer — Verlag, 1981. [Имеется перевод: Клоксин У., Меллиш К. Программирование на Прологе. — М.: Мир, 1987.]
[45] Szeredi P. Mixed Language Programming: a Method for Producing Efficient Prolog Programs, in [13].
[46] Koster C. H. A. CDL — A Compiler Implementation Language. in Methods of Algorithmic Language Implementation, LNCS 47, 1976.

УПРАВЛЕНИЕ ПАМЯТЬЮ В РЕАЛИЗАЦИЯХ ПРОЛОГА [1)]

М. БРАНОХЕ

Резюме. В статье описываются исполнение логнческих программ «сверху вниз», понятия стратегии исчисления и стратегии поиска. Показывается, как используемая в Прологе стратегия поиска в глубину позволяет существенно упростить необходимые структуры данных периода исполиения. В общих чертах описываются интерпретатор и его структуры данных, причем это довольно точное описание наиболее известных реализаций. Также в общих чертах, не вдаваясь в действительное представление связываний переменных, обсуждаются различные возможности экономии памяти посредством удаления элементов коитекстного стека периода исполнения. В заключение обсуждаются проблемы экономии памяти, зависящие от представления связываний переменных. Описываются два возможиых способа представления: совместное использование структуры и копирование.

1. ЛОГИЧЕСКИЕ ПРОГРАММЫ И ПРОЛОГ

Логические программы [6]

Логическая программа состоит из набора процедур и целевого утверждения. Процедура (или хорновский дизъюнкт) имеет вид $B \leftarrow A_1, \ldots, A_n$ $(n \geqslant 0)$, где B и A_i — литералы. Литералы имеют вид [2)] $R(t_1, \ldots, t_k)$ $(k > 0)$, где R — это k-местное отношение, а t_i — термы, т. е. константы (имена, начинающиеся с прописной буквы), переменные (имена, начинающиеся со строчной буквы) или выражения вида $f(t_1, \ldots, t_m)$ $(m > 0)$, где f есть m-арный функциональный символ, а t_i — произвольные термы.

Процедуру $B \leftarrow A_1, \ldots, A_n$ можно использовать для решения задачи, если заголовок B процедуры совместим с литералом («вызовом»), представляющим задачу. «Совместимость» означает, что заголовок и вызов должны быть согласованы так, чтобы их можно было рассматривать как одну

[1)] M. Bruynooghe «The memory Management of Prolog Inplementations». Logic programming. — N. Y.: Academic Press, 1982.

[2)] Формулу такого вида обычно называют атомом, а литералом (или литерой) — атом или его отрицание. — *Прим. ред.*

и ту же задачу (литерал), а именно как самый общий случай вызова, для которого можно использовать процедуру. Совмещение («унификация») достигается посредством подстановки $\theta = (x_1 \leftarrow t_1, \ldots, x_k \leftarrow t_k)$ термов t_i вместо переменных x_i.

Чтобы исполнить программу способом «сверху вниз», *стратегия поиска* выбирает исходное целевое утверждение $\leftarrow A_1, \ldots, A_n$, просит *стратегию вычисления* выбрать вызов $A_l = R(t_1, \ldots, t_p)$, применяет процедуру $R(t'_1, \ldots, t'_p) \leftarrow \leftarrow B_1, \ldots, B_m$, совместимую с данным вызовом при помощи подстановки («наиболее общего унификатора») θ, и порождает новое целевое утверждение

$$\leftarrow (A_1, \ldots, A_{l-1}, B_1, \ldots, B_m, A_{l+1}, \ldots, A_n)\theta^1).$$

Стратегия поиска начинает все сначала: выбирает целевое утверждение, возможно, активирует стратегию вычисления и порождает новое целевое утверждение. Новые целевые утверждения являются *активными*. Целевые утверждения становятся *пассивными* после того, как использованы все процедуры, совместимые с выбранным вызовом. Поиск можно представить с помощью ИЛИ-дерева (*дерева поиска*), узлами которого являются целевые утверждения. Потомки узла — альтернативные целевые утверждения, выводимые из целевого утверждения данного узла. Поиск заканчивается после того, как все узлы становятся пассивными. Концевые узлы дерева соответствуют неразрешимым либо пустым целевым утверждениям. Пустые целевые утверждения представляют решения. Композиция подстановок [2]), использованных на пути от корневого узла к пустому, примененная к переменным исходного целевого утверждения, дает требуемый результат. Для завершения поиска необходимы только активные целевые утверждения. Пример дерева поиска приведен на рис. 1.

В данном примере стратегия вычисления выбирает вызов A_2 в исходном целевом утверждении. Стратегия поиска выводит из G_0 целевые утверждения G_1 и G_2, пунктирная линия показывает, что имеется еще одна процедура, которую можно применить к вызову A_2 в G_0. Вновь стратегия вычисления выбирает вызов A_1 в G_2. Кандидатами на совмещение с A_1 являются три процедуры; в данной точке была применена только одна из них.

---

[1]) $(A_1, \ldots, A_n)\theta$ — это результат применения подстановки θ ко всем вхождениям переменных формул $A_1, \ldots, A_n$. — *Прим. ред.*

[2]) Композиция $\theta_2\theta_1$ подстановок θ_1 и θ_2 — это подстановка, которая получится, если сначала применить подстановку θ_1, а затем подстановку θ_2. — *Прим. ред.*

Целевое утверждение можно представить с помощью И-дерева (*дерева доказательства*). Непосредственными потомками корня дерева являются подцели A_i исходного целевого

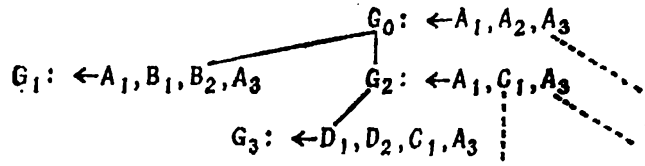


Рис. 1. Набросок дерева поиска (без учета подстановок).

утверждения $\leftarrow A_1, \ldots, A_n$. Подцель A_i, к которой применяется процедура $B \leftarrow B_1, \ldots, B_m$, получает в качестве потомков подцели $B_1, \ldots, B_m$. Унификатор θ далее применяется к каждому из узлов дерева. Целевое утверждение, соответствующее дереву доказательства, определяется его непустыми листьями.

На рис. 2 деревья доказательств G_0 и G_2 образуют поддеревья дерева доказательства G_3. Предшествующие деревья

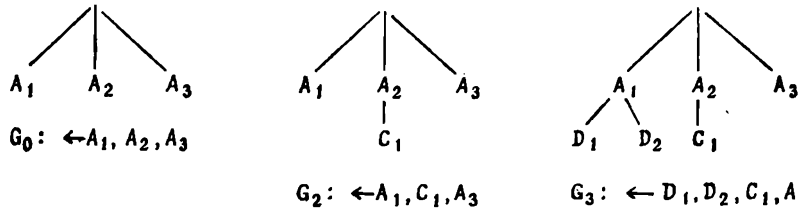


Рис. 2. Последовательность деревьев доказательств (без учета подстановок), соответствующая ветви дерева поиска на рис. 1.

непосредственно восстанавливаются из последнего дерева, если известен порядок, в котором выбираются подцели. Вообще, деревья доказательств, соответствующие последовательным целевым утверждениям, расположенным на одной ветви дерева поиска, образуют последовательность, каждый член которой получается из предшествующего путем подвешивания поддерева к листу, помеченному выбранным вызовом.

Дерево доказательства и последовательность его неконцевых узлов (порядок их выбора) определяют соответствующую последовательность деревьев доказательств и целевых утверждений. Далее часто будет говориться о целевом утверждении, соответствующем какой-либо неконцевой

подцели дерева доказательства. Например, на рис. 2 G_2 соответствует подцели A_1, а G_0 — подцели A_2 в последнем дереве доказательства.

Пролог

Стратегия поиска, используемая в Прологе, задает обход дерева поиска методом «в глубину»: всегда обрабатывается последнее активное целевое утверждение (*текущее целевое утверждение*). Такая стратегия позволяет существенно упростить необходимые структуры данных периода исполнения. В самом деле, все активные целевые утверждения расположены на одной ветви, ведущей от корня к текущему целевому утверждению, а это означает, что нужно хранить только дерево доказательства, соответствующее текущему целевому утверждению. Если последнее становится пассивным, можно восстановить непосредственно предшествующее ему активное целевое утверждение, ликвидировав последние приращения (отход назад, или «бектрекинг»).

Стратегия вычисления Пролога всегда выбирает самую левую подцель в дереве доказательства. Это значит, что подцели решаются последовательно: обработка подцели A_{i+1} начинается только после того, как подцель A_i решена полностью. Узлы достраиваются слева направо, при бектрекинге приращения удаляются справа налево. Узел в дереве доказательства, соответствующий выбранной подцели в активном целевом утверждении (цели, для которой остались неиспользованные процедуры), называется *точкой возврата.* Узлы, добавленные к дереву с момента обработки последней точки возврата, составляют часть *текущего сегмента* дерева доказательства. При бектрекинге текущий сегмент удаляется, а подстановки, примененные к дереву в процессе построения данного сегмента, отменяются. Оставшееся дерево становится текущим целевым утверждением.

2. СТРУКТУРА ДАННЫХ ПЕРИОДА ИСПОЛНЕНИЯ И ИНТЕРПРЕТАТОР

Как обсуждалось выше, структура данных периода исполнения должна представлять дерево доказательства, соответствующее текущему целевому утверждению, и давать возможность восстанавливать деревья, соответствующие точкам возврата.

В дереве доказательства подцели представлены литералами. Обычно эти литералы содержат переменные. Прежде чем применить процедуру, необходимо гарантировать, что целевое утверждение и процедура используют для своих пере-

менных различные имена. Очевидное решение заключается в создании копии процедуры с уникальными именами переменных. Однако это приводит к неэффективному использованию памяти: каждый раз, когда применяется процедура, создается новая копия. Предпочтительнее, подобно тому, как это делается в Алголе, использовать для представления процедур реентерабельный код, так чтобы все частные случаи одного и того же литерала (получаемые подстановками) имели дело с одним и тем же кодом, описывающим данный литерал. Это можно сделать, используя технику *связывающих контекстов*. Доступ к чистому коду всегда происходит в конкретном связывающем контексте. При осуществлении доступа к переменной в реентерабельном коде производится обращение к соответствующему контексту. Таким образом, литералы могут быть представлены посредством указателя на чистый код и указателя на контекст. Поскольку все литералы тела процедуры имеют дело с одним и тем же контекстом, удобно хранить его вместе с узлом, являющимся их общим отцом.

В дереве доказательства можно выделить два типа узлов. Имеются концевые узлы, представляющие нерешенные подцели текущего целевого утверждения, и неконцевые узлы, которые представляют частично решенные задачи. За исключением самого левого концевого узла (*текущей подцели*), выбранного стратегией вычисления, все концевые узлы являются частями экземпляров процедур $B \leftarrow B_1, \ldots, B_m$, в которых первый литерал B_i — либо текущая подцель, либо неконцевой узел. Если предположить, что перемещаемый код позволяет найти правых братьев B_{i+1}, B_{i+2}, ... для любой подцели B_i, и известно, что для их обработки нужен тот же самый контекст, то явного представления концевых узлов дерева доказательства не требуется. Доступ к ним можно осуществить через неконцевые узлы или текущую подцель. Для хранения неконцевых узлов удобно использовать стек («контекстный стек»). Обрабатываемые подцели помещаются в этот стек, а при бектрекинге удаляются из него.

Наконец, следует рассмотреть унификацию вызова и заголовка процедуры. Унификация не только создает связывающий контекст процедуры, но и изменяет (связывая переменные со значениями) контекст вызова и, возможно, другие контексты. Переменные и их связывания удаляются из стека (дерева доказательства) при неудачном завершении унификации, если они входят в текущий сегмент. Остальные переменные не затрагиваются этой операцией. Чтобы иметь возможность восстановить при бектрекинге исходную ситуацию, необходимо фиксировать связанные переменные, которые не

принадлежат текущему сегменту. В абстрактном дереве доказательства (рис. 3а) каждый узел содержит список переменных, значения которых, возможно, понадобится отменить. В конкретной реализации (рис. 3б и 3в) проще всего собирать такие переменные в специальный стек («след») и хра-

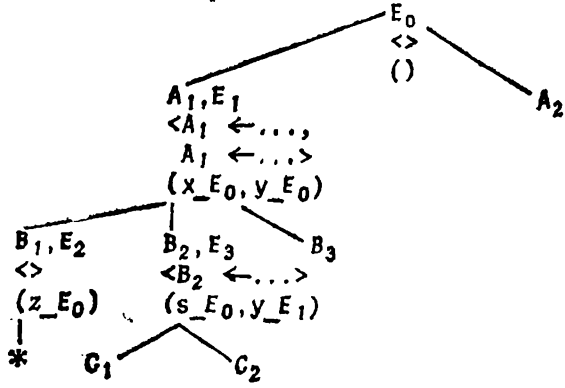


Рис. 3а. Полное абстрактное дерево доказательства. Каждый нелистовой узел содержит подцель, связывающую контекст E_i, список неиспользованных процедур (в угловых скобках) и список переменных (в круглых скобках), значения которых не аннулируются автоматически при обрезании дерева доказательства после неудачи. Рисунок описывает текущее активное целевое утверждение, а также (поскольку стратегия вычисления Пролога известна) последовательность деревьев доказательств (целевых утверждений), образующих полную ветвь в дереве поиска. В соответствии с используемой в Прологе стратегией поиска эта ветвь — единственная, содержащая активные целевые утверждения. Таким образом, рисунок описывает процесс поиска в целом.

нить для каждой точки возврата указатель на соответствующий сегмент следа.

Узлы, изображенные на рис. 3б, помещаются в стек и дополняются управляющей информацией, т. е. полями ОТЕЦ и ВОЗВ.

Состояние вычисления характеризуется следующими компонентами:

— ТЕК-ВЫЗ — указатель на код текущей подцели;

— ТЕК-КОНТ — указатель на узел, содержащий контекст текущего вызова («отца» текущего вызова);

— ТЕК-ПРОЦ — указатель на код процедуры, которая применяется к текущему вызову;

— ПОСЛ-ВОЗВ — указатель на последнюю точку возврата.

Текущее целевое утверждение определяется текущей подцелью, ее правыми братьями и правыми братьями всех ее предков.

При бектрекинге текущий сегмент удаляется из контекстного стека (узел, на который ссылается ПОСЛ-ВОЗВ, ста-

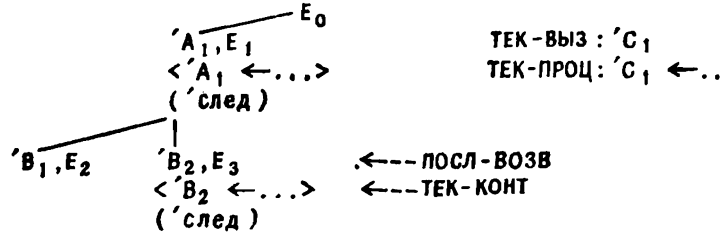


Рис. 3б. Конкретное дерево доказательства и глобальная информация о состоянии вычисления. Остаются лишь неконцевые узлы. Каждый узел содержит указатель (имя, начинающееся с апострофа) на чистый код подцели. Точки возврата содержат также указатель на чистый код первой из неиспользованных процедур и указатель на сегмент следа. Данная информация (след изображен на рис. 3в) полностью определяет абстрактное дерево доказательства, поскольку непустые концевые узлы и непустые списки неиспользованных процедур доступны посредством имеющихся указателей.

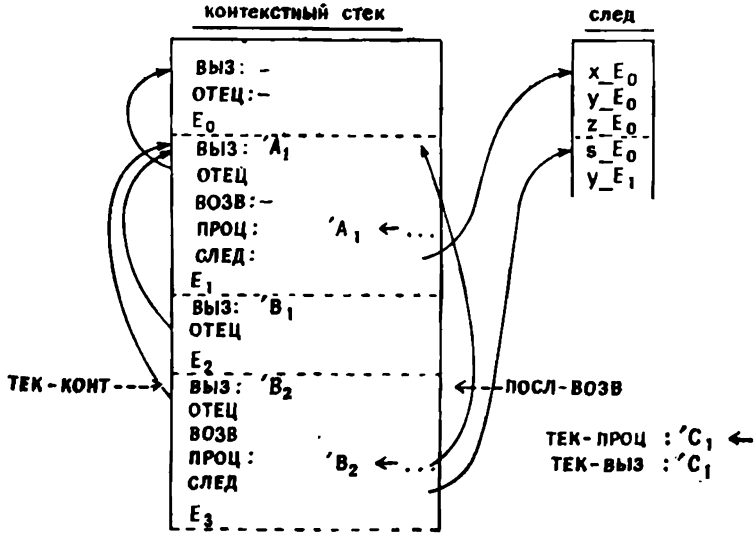


Рис. 3в. Стековое представление.

новится текущей подцелью), отменяются все присваивания, занесенные в текущий сегмент следа, и этот сегмент удаляется.

«Детерминированный» узел содержит:

— ВЫЗ — указатель на код вызова (он обеспечивает также доступ к его правым братьям);

— ОТЕЦ — указатель отца данного узла (узел-отец содержит контекст вызова ВЫЗ, а также обеспечивает доступ к правым братьям всех предков вызова ВЫЗ);

— контекст процедуры, применяемый к ВЫЗ.

Точка возврата содержит также:

— ВОЗВ — указатель на предшествующую точку возврата;

— ПРОЦ — указатель на код следующей процедуры, которую нужно применить к ВЫЗ (он обеспечивает доступ ко всем неиспользованным процедурам);

— СЛЕД — указатель на текущий сегмент следа.

Алгоритм

1. Добавить узел к контекстному стеку, при этом:
— ВЫЗ := ТЕК-ВЫЗ;
— ОТЕЦ := ТЕК-КОНТ.
— Найти следующую за ТЕК-ПРОЦ процедуру, возможно совместимую с ТЕК-ВЫЗ.
— Если такой нет,
то имеем детерминированный узел,
иначе имеем точку возврата (для различения этих случаев можно использовать бит в поле ВЫЗ или ОТЕЦ) и нужны дополнительные действия:
ВОЗВ := ПОСЛ-ВОЗВ, ПОСЛ-ВОЗВ становится указателем на новый узел;
ПРОЦ := следующая, возможно совместимая, процедура;
СЛЕД := указатель на верхушку следа.
— Сформировать связывающий контекст для переменных из ТЕК-ПРОЦ.

2. Унифицировать ТЕК-ВЫЗ (с контекстом ТЕК-КОНТ) и заголовок ТЕК-ПРОЦ (с контекстом нового узла). Все изменения в контекстном стеке, не ограничивающиеся текущим сегментом, записать в след.
ТЕК-ВЫЗ := первый вызов тела ТЕК-ПРОЦ;
ТЕК-КОНТ := новый узел контекстного стека.

3. Если унификация завершилась успешно,
то найти следующую нерешенную подцель (если такой нет, то решение получено): пока ТЕК-ВЫЗ — пустая ссылка
повторять:
ТЕК-ВЫЗ := вызов, следующий за ВЫЗ в ТЕК-КОНТ;
ТЕК-КОНТ := ОТЕЦ в ТЕК-КОНТ,

иначе бектрекинг (если ПОСЛ-ВОЗВ — пустая ссылка, то вычисление закончено):
используя указатель СЛЕД в ПОСЛ-ВОЗВ, ликвидировать изменения, не затрагивающие текущий сегмент контекстного стека, и удалить текущий сегмент следа;
ТЕК-ВЫЗ := ВЫЗ в ПОСЛ-ВОЗВ;
ТЕК-КОНТ := ОТЕЦ в ПОСЛ-ВОЗВ;
ТЕК-ПРОЦ := ПРОЦ в ПОСЛ-ВОЗВ;
ПОСЛ-ВОЗВ := ВОЗВ в ПОСЛ-ВОЗВ и удалить текущий сегмент из стека (все узлы, включая тот, на который указывало прежнее значение ПОСЛ-ВОЗВ).

Замечания

1. В первой реализации Пролога [1, 5] не делалось различия между детерминированными узлами и точками возврата. Многие, включая Уоррена [9], предпочитают иметь дело с фиксированным смещением контекста относительно начала узла. Это особенно желательно из-за частого переключения контекстов в процессе унификации при совместном использовании структуры. В реализации, выполненной автором, такое различие проводится, и для указания типа узла используется бит поля ВЫЗ.

2. Детерминированный узел занимает меньше памяти, и, что более важно, детерминированные узлы играют решающую роль в способах экономии памяти, описанных в следующем разделе. Это оправдывает некоторые затраты (связанные с вычислениями или адекватным представлением) на обнаружение «первой процедуры, возможно совместимой с вызовом». Уоррен [9] использует индексацию по первому аргументу заголовка процедуры. Другие (в том числе автор) пользуются более медленной, но более точной «мягкой» индексацией по всем аргументам: процедура приемлема, если все аргументы заголовка и соответствующие аргументы вызова взаимно приемлемы. Два соответствующих аргумента являются взаимно приемлемыми, если один из них содержит переменную, или оба содержат одинаковую константу, или оба содержат одинаковый функциональный символ и соответствующие аргументы этих функциональных символов взаимно приемлемы.

3. УДАЛЕНИЕ ЭЛЕМЕНТОВ ИЗ СТЕКА

В данный момент мы игнорируем наличие контекстов или, более точно, предполагаем, что ссылки между контекстами, если таковые имеются, ориентированы сверху вниз (от листьев к корню дерева доказательства). Это значит, что

элементы стека можно удалять, не порождая висячих ссылок. Позже мы вернемся к этой проблеме.

а. Обработка детерминированной подцели

На рис. 4 через r обозначен корень дерева доказательства, d_i — детерминированные подцели, b_i — точки возврата, s_i —

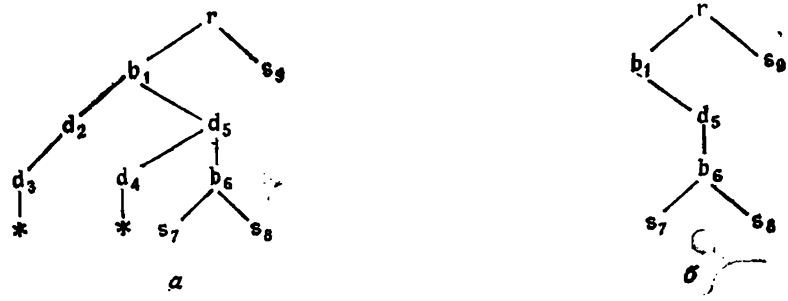


Рис. 4. Отсечение детерминированного поддерева.

нерешенные подцели, а через * — пустые множества подцелей. Подцель, соответствующая d_2, полностью решена, это решение единственно (в поддереве нет точек возврата). Такое завершенное поддерево можно удалить из дерева доказательства (рис. 4, *б*), не повлияв на поведение алгоритма. Действительно, текущее целевое утверждение остается

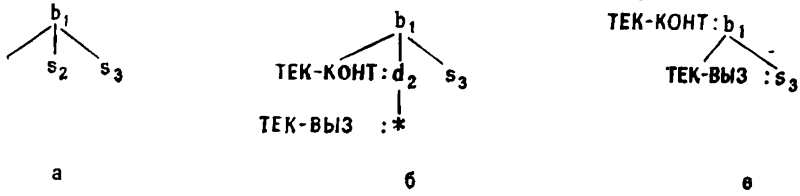


Рис. 5. Обработка вызова s_2, с которым совместима единственная процедура с пустым телом.

прежним, сохраняются также и целевые утверждения, соответствующие точкам возврата (b_1, b_6).

Такой метод соответствует ситуации в традиционных языках программирования: запись активации удаляется из стека при возврате из процедуры.

Если интерпретатор распознает и обрабатывает случай «хвостовой рекурсии» (см. разд. 3, *б*), то применение данного метода требуется только в ситуации, изображенной на рис. 5, где с вызовом совместима единственная процедура с пустым телом.

Интерпретатор распознает эту ситуацию (рис. 5, *б*) с помощью проверки условия: ТЕК-ВЫЗ — пустая ссылка, а ТЕК-КОНТ указывает на детерминированный узел.

При обработке ситуации алгоритм выполняет следующие действия: ТЕК-ВЫЗ := вызов, следующий за ВЫЗ в ТЕК-КОНТ; ТЕК-КОНТ := ОТЕЦ в ТЕК-КОНТ; удаляет верхний

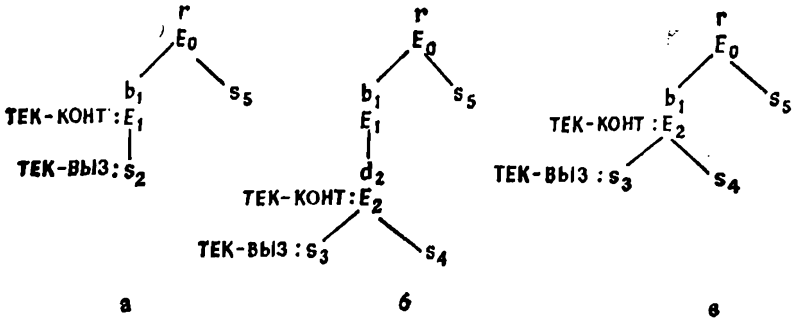


Рис. 6. Замена хвостовой рекурсии итерацией.

узел из стека (новой верхушкой становится ТЕК-КОНТ либо ПОСЛ-ВОЗВ).

б. Хвостовая рекурсия

В ситуации, изображенной на рис. 6, E_1 обозначают контексты, входящие в состав узлов, s_2 (рис. 6, *а*) — последний вызов в теле процедуры, сохранившийся в дереве доказательства (возможные левые братья удалены). С ним совместима единственная процедура (рис. 6, *б*). Поведение интерпретатора на деревьях доказательств рис. 6, *б* и 6, *в* одно и то же. Причины этого таковы: (1) текущие целевые утверждения одинаковы (с одинаковыми контекстами !) и (2) целевые утверждения, соответствующие точкам возврата, также идентичны. Узел, описывающий вызов s_2, был склеен со своим отцом (E_2 заменяет E_1, а узел s_2 удаляется). Этот же узел можно теперь использовать для обработки s_3 (повторное использование памяти). Такая ситуация типична для рекурсивных вызовов в хвостовой рекурсии. Рекурсия заменяется итерацией. Это дает значительную экономию памяти в случае глубокой рекурсии.

Здесь нарушается предположение, сделанное в начале раздела. Мы удаляем контекст E_1, находящийся не в вершине стека. Это может породить висячие ссылки. Чтобы иметь возможность удалить E_1, нужно либо произвести слож-

ное преобразование ссылок, направленных от E_2 к E_1, либо добавить специальный управляющий механизм, гарантирующий, что ссылки, связывающие E_1 и E_2, направлены от E_1 к E_2. Этой проблемы можно избежать, обменяв E_1 и E_2 перед созданием связывающих их ссылок. Это восстанавливает наше предположение.

Интерпретатор распознает описанную ситуацию непосредственно перед унификацией (см. рис. 6, *а*), проверив условие: у ТЕК-ВЫЗ нет следующего за ним вызова, а ТЕК-КОНТ не глубже в стеке, чем ПОСЛ-ВОЗВ (т. е. ТЕК-ВЫЗ является детерминированным, а его левые братья удалены из стека).

При обработке ситуации перед унификацией нужно поменять местами контексты нового узла и узла ТЕК-КОНТ. Перемещение контекста E_1 следует производить осторожно. Не исключено, что E_1 содержит ссылки на себя, необходимо простое преобразование таких указателей (прибавление константы).

После унификации (см. рис. 6, *б*) действия таковы: ТЕК-КОНТ := ОТЕЦ в ТЕК-КОНТ, удалить верхний узел из стека (новой вершиной становится прежнее значение ТЕК-КОНТ).

Замечания

1. При обработке s_3 или s_4 (см. рис. 6, *в*) поле ОТЕЦ соответствующих узлов будет содержать ссылку на узел b_1, хотя b_1 в действительности является не отцом, а более далеким предком. Это важно при исполнении усечения (/), специальной операции в Прологе, которая преобразует все точки возврата поддерева, корнем которого является (подлинный) отец соответствующего узла, в детерминированные узлы. Ложный отец (например, b_1, если s_3 или s_4 — усечение) должен быть защищен от такой участи. Бит одного из полей ОТЕЦ или ВЫЗ (в b_1) можно использовать для указания, является ли узел подлинным отцом своих ближайших потомков (s_3 и s_4) или более далеким предком.

2. Есть случаи, когда оптимизация хвостовой рекурсии не проходит вследствие недетерминизма последнего вызова. Исполнение усечения в теле применяемой процедуры устраняет данный недетерминизм. Тогда становится возможным (задержанное) выполнение оптимизации хвостовой рекурсии. Это сложная задача вследствие необходимости преобразования указателей.

3. Оптимизация хвостовой рекурсии важна, когда доступно небольшое количество рабочей памяти. Она менее важна, когда памяти достаточно, поскольку последняя освобождается в любом случае. О повышении быстродействия с помощью данной оптимизации см. [10].

в. Трудности

До сих пор не учитывалось действительное представление связываний переменных. Описанные выше оптимизационные преобразования основывались на предположении о том, что все ссылки между контекстами ориентированы в сторону дна стека. В первом интерпретаторе для Пролога [1] это условие нарушалось, вследствие чего удаление элементов из стека могло происходить только при бектрекинге. Попытка удаления при обработке детерминированной подцели побудила автора [3, 4] и Дэвида Уоррена [9] разработать иные методы работы с контекстами. Автор первым осуществил оптимизацию хвостовой рекурсии (в 1977 г.) [3], впоследствии реализованную также Уорреном [10].

4. ПРЕДСТАВЛЕНИЕ СВЯЗЫВАЮЩИХ КОНТЕКСТОВ

Совместное использование структуры [1])

В первом интерпретаторе для Пролога [1] представление связывания переменной является вариантом совместного использования структуры Бойера и Мура [2]. Как и подцели, связывание переменной представлено парой указателей — на чистый код и на связывающий контекст. Связывающий контекст в свою очередь содержит связывания переменных, указанных в коде [2]). (В некоторых реализациях пара указателей используется только для переменной, связанной со сложным термом. Для других значений используются поле типа и либо адрес значения (указатель), либо его непосредственное представление, например, для переменной — адрес в стеке.)

При такой схеме представления в ситуации, когда алгоритм унификации должен связать переменную, возможны три случая:

1. Совмещаются переменная x, свободная в контексте E_i, и переменная y, свободная в контексте E_j. Предположим, что контекст E_i более новый, чем E_j (или совпадает с ним). Тогда переменная x в E_i связывается с кодом y и с контекстом E_j. Ссылка от E_i к E_j ориентирована в сторону дна контекстного стека.

2. Совмещаются переменная x, свободная в контексте E_i, и терм t с контекстом E_j, причем контекст E_i более новый, чем E_j. Переменная x связывается с кодом t и с контекстом E_j. Эта ссылка также ориентирована в сторону стека.

---

[1]) В оригинале — structure sharing, что часто неудачно переводят как «разделение структуры», тогда как речь идет об использовании (доступе и т. д.) одной структуры несколькими «потребителями». — *Прим. ред.*

[2]) В коде переменная обычно указывается своим смещением относительно начала связывающего контекста. — *Прим. перев.*

3. То же, что и в (2), но контекст E_j более новый, чем E_i. Здесь также x должна быть связана с кодом t и с контекстом E_j, но ссылка от E_i к E_j ориентирована к вершине контекстного стека. Вследствие этого методы экономии памяти, описанные в предыдущем разделе, неприменимы.

Пример

Вызов представлен чистым кодом P(r, s, t) и связывающим контекстом E_i, который образуют:
— r: 'f(...), E_r — указатель на чистый код, представляющий терм f(...), и указатель на контекст E_r;
— s: свободная переменная;
— t: свободная переменная.
Заголовок представлен чистым кодом P(x, y, g(x, u)).
Создается новый связывающий контекст E_j со свободными переменными x, y и u.

Процесс совмещения дает следующие связывания:
— x из E_j: 'f(...), E_r(случай 2) (значение то же, что у r из E_i);
— y из E_j: 's, E_i(случай 1) (указатель на чистый код переменной s в вызове);
— t из E_i: 'g(x, u), E_j(случай 3) (указатель на чистый код в заголовке).

При решении этой проблемы Уоррен [9] заметил, что трудности причиняют переменные, встречающиеся в термах из чистого кода. Он назвал такие переменные *глобальными*, а остальные — *локальными* и разбил связывающий контекст на глобальную и локальную части. В чистом коде он также провел различие между глобальными и локальными переменными. Локальные контексты поместил в контекстный стек, а глобальные — в специальный *глобальный стек*. Элементы из этого стека удаляются только при бектрекинге. Теперь унификация начинается с двумя литералами, представленными чистым кодом, каждый с локальным и глобальным контекстами. Алгоритм предусматривает, что все ссылки между контекстами ориентированы либо сверху вниз в контекстном стеке, либо от контекстного стека к глобальному. Это возможно, поскольку всякий раз, когда свободная переменная совмещается с термом, все переменные, встречающиеся в терме, находятся по соглашению в глобальном контексте.

В нашем примере y принадлежит $E_{j-лок}$, x и u принадлежат $E_{j-глоб}$, а связывание принимает вид:

t из E_i: 'g(x, u), $E_{j-глоб}$.

Замечание. Пользователь может объявить (с помощью «деклараций вида»), что некоторые термы никогда не будут совмещаться со свободными переменными. Это дает возможность классифицировать большее число переменных как локальные.

Копирование чистого кода [3, 4, 7]

Связывание переменной может быть представлено одним указателем на прямое представление значения. Однако чистый код, содержащий переменные, не может быть частью такого представления. Всякий раз, когда свободная переменная совмещается с термом из чистого кода, содержащим переменные, создается копия терма. В этой копии коды переменных заменяются следующим образом:

— если переменная свободна в соответствующем связывающем контексте, то копия получает новую свободную переменную, с которой связывается данная переменная;

— если переменная в соответствующем связывающем контексте уже связана с частью копии, то устанавливается указатель от новой копии к имеющейся (копируется «связанная» переменная).

Использование специального стека копий, элементы из которого удаляются только при бектрекинге, гарантирует, что ссылки ориентированы либо сверху вниз внутри контекстного стека, либо от контекстного стека к стеку копий, либо в любом направлении в стеке копий. В этом случае опять можно удалять элементы из контекстного стека без угрозы появления висячих ссылок.

Вернемся к нашему примеру. Заданный связывающий контекст E_1 теперь образуют:

— r: 'f(...) — указатель на прямое представление терма f(...);
— s: свободная переменная;
— t: свободная переменная.

Процесс совмещения дает теперь следующие результаты:

— x из E_1: 'f(...) — указатель на существующее прямое представление f(...);
— y из E_1: 's — указатель на переменную s из контекста E_1;
— t из E_1: 'g('f(...), u') — указатель на построенное прямое представление терма g с двумя аргументами: первый аргумент является указателем на существующее прямое представление f(...), второй аргумент — новая свободная переменная с именем u', и мы также получаем:
— u из E_1: 'u' — указатель на новую свободную переменную u'.

Замечания

1. Уоррен указал, что переменные можно удалять из локального контекста, если на них не будет ссылок при последующем исполнении. В частности, локальные переменные с единственным вхождением («пустые» переменные) не требуют места в контекстном стеке, алгоритму унификации известно, что они являются свободными. Локальные переменные, встречающиеся только в заголовке, могут быть удалены после унификации вызова и заголовка. Действительно, на них нет ссылок в теле процедуры. То же самое, но для всех переменных, верно в случае метода копирования.

2. Некоторые участки глобального стека (стека копий) могут становиться недоступными. В этом случае возможны сборка мусора и уплотнение.

5. ОБСУЖДЕНИЕ

Чтобы получить представление об объеме памяти, который требуется обоим методам, можно сравнить потребности в памяти узлов и связывающих контекстов.

При копировании детерминированный узел требует 2 поля (ВЫЗ, ОТЕЦ); при совместном использовании структуры нужны 3 поля (еще указатель на глобальный контекст), обычно используется 6 полей (чтобы не допустить снижения быстродействия из-за частого переключения контекстов).

Точка возврата требует при каждом методе 6 полей: (ВЫЗ, ОТЕЦ, ВОЗВ, ПРОЦ, СЛЕД, указатель на глобальный стек или стек копий).

При совместном использовании структуры связывающий контекст процедуры требует 2 поля для каждой переменной. Разделение на локальные (размещаемые в контекстном стеке) и глобальные (размещаемые в глобальном стеке) переменные определяется описанием процедуры. При копировании для каждой переменной требуется одно поле в контекстном стеке. Труднее оценить память, нужную для копий. Число создаваемых копий зависит от структуры вызова. Требуемый для копии объем памяти зависит от выбранного представления. Приведем два возможных способа:

а) Для копирования терма вида $f(t_1, \ldots, t_n)$, содержащего переменные, можно использовать $n + 1$ поле, одно для указания имени f и по одному для указателей на представление каждого аргумента. Такое представление обеспечивает быстрый доступ к i-му аргументу.

б) При наличии вложенных термов можно избежать указателей на аргументы, разместив сами аргументы друг за другом. Тогда для каждого символа достаточно одного поля,

Такое представление более компактно, но доступ к i-му аргументу медленнее. При обоих способах представления копия переменной является либо свободной переменной, либо указателем на значение переменной.

Для каждого типичного вызова процедур, выбранных для контрольных измерений Уорреном [9], мы вычислили объем памяти, занимаемой связывающими контекстами. Суммируя результаты для 23 типичных вызовов, мы заметили, что общий объем памяти несколько меньше при копировании (184—216) и что при совместном использовании структуры требуются декларации вида для сокращения глобального стека до сравнимых размеров (от 130 до 82). Однако в отдельных случаях копирование может оказаться существенно хуже, чем совместное использование структуры, а именно в тех случаях, когда требуется копирование больших термов. Возможна и обратная ситуация, когда при наличии большого числа переменных копирование не производится, но она менее вероятна на практике.

В работе Меллиша [8] изучается поведение нескольких программ при двух способах представления. При полном сравнении обоих методов необходимо также учитывать проблему скорости выполнения. Мы вынуждены ограничиться замечанием, что совместное использование структуры быстрее при построении новых сложных термов, но медленнее при осуществлении доступа к ним.

Замечания

1. При анализе потребностей в памяти для каждого метода мы рассматривали количество необходимых полей. Машинно-зависимые детали обсуждаются в [7, 9].

2. Меллиш [7] обсуждает некоторый вариант совместного использования структуры. Он замечает, что два поля для представления связывания переменной требуются только в том случае, когда чистый код указывает на структурный объект (а не на переменную или константу). Он предлагает использовать одно поле. Когда нужны оба поля, в глобальном стеке создается «молекула», содержащая два поля, а переменная указывает на эту молекулу. В большинстве случаев такой подход сокращает общий объем памяти (для наших данных он близок к объему, получаемому при копировании), но увеличивает размер глобального стека, особенно в сочетании с объявлениями вида. С точки зрения оптимизации работы с контекстным стеком, особенно в случае хвостовой рекурсии, предпочтительнее выглядит метод, дающий меньший глобальный стек, в особенности если не производится сборка мусора.

ЛИТЕРАТУРА

[1] Battani G., Meloni H. Interpreteur du language de Programmation PROLOG. Groupe Intelligence Artificielle Université Aix — Marseille 11, 1973.

[2] Boyer R. S., Moore J. S. The Sharing of structure in theorem proving programs. In «Machine Intelligence», 7, Edinburgh University Press, 1972.

[3] Bruynooghe M. An Interpreter for Predicate Logic Programs. Part 1. Report CW 10, Applied Mathematics and Programming Division, Katholieke Universiteit, Leuven, Belgium, 1976.

[4] Bruynooghe M. Naar een Betere Beheersing van de Uitvoering van Programma's in de Logika der Horn-uitdrukkingen. (In Dutch) Doctoral Dissertation, Afdeling Toegepaste Wiskunde en Programmatie, K. U. Leuven, Belgium. 1979.

[5] Colmerauer A., Kanoui H., Passero R., Roussel P. Un Système de Communication Homme-machine en Francais. Research report. Groupe Intelligence Artificielle, Université Aix — Marseille 11, 1973.

[6] Kowalski R. A. Predicate logic as programming language. Proc. IFIP-74 Congress. North-Holland, pp. 569—574.

[7] Mellish C. S. An Alternative to Structure-Sharing in the Implementation of a PROLOG Interpreter. Research Paper 150, Department of Artificial Intelligence, University of Edinburgh, 1980.

[8] Mellish C. S. An alternative to structure sharing in the implementation of a PROLOG-interpreter. In «Logic Programming» (Eds. Clark K. L., Tärnlund S.-Å.), Academic Press, 1982, pp. 99—106.

[9] Warren D. Implementing PROLOG — Compiling Logic Programs. 1 and 2. D. A. I. Research Report No 39, 40, University of Edinburgh, 1977.

[10] Warren D. An improved PROLOG implementation which optimises tail recursion. In «Proceedings of the Logic Programming Workshop» (Ed. Tärnlund S.-Å), Debrecen, Hungary, 1980.

СПЕЦИФИКАЦИЯ АЛГОРИТМОВ ЧИСЛЕННОГО ИНТЕГРИРОВАНИЯ НА ЯЗЫКЕ ЛОГИЧЕСКИХ ПРОГРАММ[1])

К. КЛАРК, У. МАККИМАН, Ш. ЗИКЕЛЬ

1. ВВЕДЕНИЕ

В этой статье будет построено несколько алгоритмов численного интегрирования с использованием обозначений и концепций логического программирования. Мы начнем с логической программы, которая представляет собой просто кодировку двух аксиом определенных интегралов в виде отношений. Эта программа является спецификацией отношения вход-выход алгоритма интегрирования. Она также определяет наш первый, в высшей степени недетерминированный алгоритм. Затем для того, чтобы ограничить недетерминированность программы, будут рассмотрены различные стратегии управления. Добавляя к программе по очереди каждую из трех стратегий управления, мы определим три семейства алгоритмов интегрирования, при этом каждый член семейства будет определяться более жесткой стратегией управления. Это — спецификация алгоритма типа «логика + управление» (Л + У), которую пропагандировали Ковальский [1] и другие [2, 3]. После этого шаг за шагом каждая (Л + У)-спецификация будет преобразована в новую (Л + У)-спецификацию, в которой логика более развита, а управление упрощено. При этом на каждом шаге будет сохраняться корректность логики, т. е. гарантируется, что наша программа по-прежнему состоит из множества истинных утверждений о численном интегрировании. На последней стадии преобразования мы усиливаем управление так, чтобы получить детерминированный алгоритм. Таким образом, будут систематически разработаны пять детерминированных алгоритмов интегрирования. Каждый из них будет задан логической программой в паре с последовательным управлением.

2. ОБОЗНАЧЕНИЯ ЛОГИЧЕСКОГО ПРОГРАММИРОВАНИЯ

Мы предполагаем, что вы знакомы с основными положениями логического программирования, изложенными Коваль-

[1]) Keith L. Clark, W. M. McKeeman, S. Sickel. Logic program specification of numerical integration. — In: Logic Programming. — N. Y., Academic Press, 1982, p. 123—139.

ским [4]. Наши логические программы будут множествами импликаций вида

$$B \leftarrow A_1, \ldots, A_m, \quad m \geqslant 0,$$

где B и каждое A_i — атомарные формулы. Эта импликация является процедурой для отношения, определяемого атомарной формулой B. Тело процедуры образуется множеством предусловий/вызовов процедур $A_1, \ldots, A_m$. Для удобства чтения мы будем свободно пользоваться инфиксными обозначениями, например писать $e \leqslant EPS$ вместо $\leqslant(e, EPS)$. Кроме того, мы допускаем использование арифметических выражений в качестве аргументов предикатов. Иначе говоря, для сокращения предусловия $y = e$, которое является именем отношения «y имеет значение e», это равенство будет опущено и все остальные вхождения y будут заменены на e. Так, вместо

$$I(a, b, f, y, e) \leftarrow I(a, m, f, y_1, e_1), I(m, b, f, y_2, e_2), y = y_1 + y_2, e = e_1 + e_2$$

мы будем писать

$$I(a, b, f, y_1 + y_2, e_1 + e_2) \leftarrow I(a, m, f, y_1, e_1), I(m, b, f, y_2, e_2).$$

Логика + управление

Исполнение логической программы — это вычисление выходных значений, связывающих переменные вызова процедуры B′. При поиске ответа на вызов может использоваться любая из процедур программы для вызываемого отношения. Это — один источник недетерминизма вычисления. Правило, которое выбирает при каждом вызове в *точности* одну процедуру для поиска ответа на вызов, мы будем называть *правилом выбора*. Другой источник недетерминизма возникает из-за того, что вызовы $A_1, \ldots, A_m$ процедур могут выполняться в любом порядке, даже как сопрограммы. Правило, которое для каждой вызванной процедуры определяет порядок вычисления ее вызовов или дает критерий выбора альтернатив для вычисления некоторых или всех вызовов, мы назовем *правилом вычисления*. Логическая программа вместе с правилом выбора и правилом вычисления становится *детерминированным алгоритмом* (детерминированным потому, что наше правило выбора отбирает для каждого вызова в точности одну процедуру). Он останавливается, если правило выбора отбирает среди всех возможных путей вычисления успешно завершающийся путь вычисления.

Семейства алгоритмов

Если управление не задано, то логическая программа является недетерминированным алгоритмом. Она представляет семейство всех детерминированных алгоритмов, которые можно получить, добавляя управление с помощью специфических правил вычисления и правил выбора. Частично задавая это управление, мы получаем новый недетерминированный алгоритм, который является подсемейством этого семейства алгоритмов.

3. ОБЗОР ЧИСЛЕННОГО ИНТЕГРИРОВАНИЯ

Численное приближение интегралов восходит к временам древних греков и аппроксимации числа π. Прекрасный обзор проблемы и современных методов решения можно найти в [5].

В конечном счете все численные приближения имеют вид

$$\int_a^b f(x)\,dx = (b-a)\sum_{k=0}^{n-1} w_k f(x_k), \quad \text{где} \quad \sum_{k=0}^{n-1} w_k = 1.$$

То есть приближение есть взвешенная сумма значений функции на интервалах интегрирования. Здесь возникают следующие задачи:

1. Выбор точек (x_k).
2. Выбор весов (w_k).
3. Выяснение, насколько точен получаемый результат.

Адаптивные алгоритмы наиболее плотно выбирают точки в областях, где поведение интегрируемой функции наиболее непредсказуемо. Используемые для этого стратегии, как и результирующие программы, имеют тенденции к возрастанию их сложности и непонятности [6].

Принято представлять только детерминированные численные алгоритмы. Они отличаются друг от друга видом используемого приближения, порядком применения приближений и стратегией завершения. Если расширить нашу точку зрения, чтобы включить в рассмотрение недетерминированные алгоритмы, то можно организовать их иерархически, т. е. алгоритм тем более детерминирован, чем ниже он стоит в иерархии. Такая иерархия определяет семейства алгоритмов и является естественным результатом использования логических программ в качестве спецификаций.

4. ФОРМАЛЬНАЯ СПЕЦИФИКАЦИЯ АДАПТИВНОЙ КВАДРАТУРЫ

Эта спецификация — множество аксиом об определенных интегралах. Первая аксиома — формальное тождество из математического анализа:

$$\int_a^b f(x)\,dx = \int_a^m f(x)\,dx + \int_m^b f(x)\,dx.$$

Отсюда и из элементарной теории чисел можно вывести следующую теорему [1]):

$$\left|\int_a^m f(x)\,dx - y_1\right| \leqslant e_1,\ \left|\int_m^b f(x)\,dx - y_2\right| \leqslant e_2,\quad a < m < b$$

$$\Rightarrow \left|\int_a^b f(x)\,d(x) - (y_1 + y_2)\right| \leqslant e_1 + e_2. \qquad (1)$$

Доказательство: $|s| \leqslant x$, $|t| \leqslant y \Rightarrow |s+t| \leqslant x + y$.

Формула (1) дает базисный шаг адаптивной процедуры, редуцируя общую задачу к двум, предположительно более простым, подзадачам. Процесс разбиения на подзадачи оканчивается, когда установлено, что можно использовать численное приближение. Имеется много способов осуществить приближение, каждый из которых зависит от пары формул, называемых здесь quad и err. Например, если задано приближение методом трапеций

$$\mathrm{trap}(f, a, b) = (f(a) + f(b)) * (b - a)/2,$$

то в качестве определений функций quad и err можно использовать

$$\mathrm{quad}(f, a, b) = \mathrm{trap}(f, a, (a + b)/2) + \mathrm{trap}(f, (a + b)/2, b)$$

и

$$\mathrm{err}(f, a, b) = |\,\mathrm{quad}(f, a, b) - \mathrm{trap}(f, a, b)\,|.$$

Для подходящего класса функций имеет место неравенство

$$\left|\int_a^b f(x)\,dx - \mathrm{quad}(f, a, b)\right| \leqslant \mathrm{err}(f, a, b). \qquad (2)$$

Формулы (1) и (2) представляют собой два предположения, на которых основаны процедуры численного интегриро-

[1]) Здесь и далее $A \Rightarrow B$ означает «если A, то B», «$A \Leftrightarrow B$» означает «A, если и только если B».— *Прим. ред.*

вания, и, следовательно, здесь они имеют статус аксиом. Другими словами, мы имеем дело только с функциями, удовлетворяющими (2).

5. ОТ СПЕЦИФИКАЦИИ К ПРОГРАММЕ

Пусть задан интервал ⟨A, B⟩, функция F и ограничение на погрешность EPS, и требуется найти y, такое, что

$$\left| \int_A^B F(x)\,dx - y \right| \leqslant \mathrm{EPS}. \qquad (3)$$

Введем отношение I(a, b, f, y, e) с помощью определения

$$I(a, b, f, y, e) \Leftrightarrow \left| \int_a^b f(x)\,dx - y \right| \leqslant e. \qquad (4)$$

В терминах логического программирования наша задача интегрирования (3) заключается в поиске ответа на запрос

$$y\colon\ I(A, B, F, y, e),\quad e \leqslant \mathrm{EPS} \qquad (5)$$

(найти y, такое, что I(A, B, f, y, e) для некоторого $e \leqslant \mathrm{EPS}$), используя логическую программу для отношения I. Наши определяющие аксиомы (1) и (2) дают нам такую программу. Используя определение (4), аксиомы можно переформулировать в виде пары процедур логической программы:

$$I(a, b, f, y_1 + y_2, e_1 + e_2) \leftarrow$$
$$a < m < b,\ I(a, m, f, y_1, e_1),\ I(m, b, f, y_2, e_2). \qquad (6)$$

$$I(a, b, f, \mathrm{quad}(f, a, b), \mathrm{err}(f, a, b)). \qquad (7)$$

Добавление управления

Использование этой программы для ответа на запрос (5) крайне недетерминировано. На самом первом шаге мы можем использовать или процедуру приближения (7), или процедуру расщепления (6), и при каждом вызове процедуры расщепления этот выбор возникает снова для каждого из двух рекурсивных вызовов. Слишком раннее использование правила приближения для отдельного интервала может привести к неудаче при проверке выполнимости условия $e \leqslant \mathrm{EPS}$, налагаемого на сумму всех оценок погрешностей. Слишком позднее использование означает излишние вычисления при проверке того, что граница ошибки лежит в пределах EPS,

Кроме того, существует недетерминизм при выполнении вызова $a < m < b$ процедуры расщепления. Мы будем считать, что вызов исполняется с помощью недетерминированной программы, которая связывает с m некоторое значение из интервала.

Правило выбора управляет этим недетерминизмом. Оно выбирает определенный путь исполнения для программы, который отвечает на каждый вызов $a < m < b$. Таким образом, оно определяет промежуточную точку m, а также решает при каждом вызове $I(a, b, f, y, e)$, какую из процедур, (6) или (7), следует использовать, т. е. расщеплять интервал или нет. Общий итог правила выбора заключается в определении множества из $k + 1$ подынтервалов

$$S = \{\langle A, m_1\rangle, \langle m_1, m_2\rangle, \ldots, \langle m_k, B\rangle\}$$

заданного начального интервала $\langle A, B\rangle$. На каждом подынтервале применяется приближение. Для того чтобы определить успешно завершающийся алгоритм, правило выбора должно быть таким, чтобы сумма оценок погрешностей по всем этим подынтервалам не превышала EPS.

Другим источником недетерминизма можно управлять с помощью правила вычисления для вызовов процедуры расщепления. Правило вычисления определяет метод порождения для множества интервалов S. Наиболее очевидным правилом вычисления является правило, исполняющее вызовы последовательно, в том порядке, в котором они заданы. Иначе говоря, мы используем вызов $a < m < b$ для нахождения точки расщепления, а затем интегрируем интервалы один за другим. Это соответствует разбиению интервала слева направо, при котором множество порождается как последовательность

$$\langle A, m_1\rangle, \ldots, \langle m_{l-1}, m_l\rangle, \ldots.$$

Каждый раз, когда правило выбора выбирает процедуру приближения для ответа на вызов $I(m_l, m_{l+1}, f, y, e)$, оно добавляет к последовательности новый интервал $\langle m_l, m_{l+1}\rangle$.

Комбинируя логическую программу, состоящую из (6) и (7), с конкретными правилами вычисления и выбора, мы получаем различные алгоритмы интегрирования. Частично определяя управление с помощью правил вычисления/выбора, мы определяем подсемейство алгоритмов интегрирования, идентифицируемых недетерминированным алгоритмом, который определяет это частичное управление. Рассмотрим три таких семейства, идентифицируемых тремя недетерминированными алгоритмами с номерами 1, 2, 3.

Алгоритм 1

Этот алгоритм использует последовательное правило вычисления. Нам нужно добавить к нему правило выбора, которое гарантирует, что сумма приближений погрешностей для интервалов из множества S лежит в пределах допустимой погрешности EPS. Чтобы гарантировать это, мы можем использовать правило выбора, которое применяет процедуру приближения, соответствующую базисному случаю, для вызова $I(m_i, m_{i+1}, f, y, e)$ только тогда, когда оценка погрешности для интервала $\langle m_i, m_{i+1}\rangle$ меньше, чем $EPS(m_{i+1} - m_i)/(B - A)$, а часть EPS линейно относится к длине интервала. Ограниченное таким образом правило выбора определяет простой недетерминированный алгоритм интегрирования. Он недетерминирован, поскольку способ, по которому правило выбора определяет промежуточные точки при вызовах $a < m < b$, остается неопределенным.

Алгоритм 2

Приведенное выше ограничение на правило выбора содержит строгое условие, гарантирующее, что общая погрешность будет находиться в пределах EPS. Более тонкое правило выбора может учитывать разность, которая накапливается из-за того, что сумма e′ вычисленных оценок погрешностей для интервалов $\langle A, m_1\rangle, \ldots, \langle m_{i-1}, m_i\rangle$ меньше, чем $EPS(m_i - A)/(B - A)$ (доля числа EPS для промежутка, который они покрывают). Для того чтобы получить такое поведение, правило выбора выбирает процедуру приближения для вызова $I(m_i, m_{i+1}, f, y, e)$ только тогда, когда оценка погрешности для интервала $\langle m_i, m_{i+1}\rangle$ меньше, чем $(EPS - e')(m_{i+1} - m_i)/(B - m_i)$. Это — доля допустимого остатка погрешности, распределенного линейно по оставшимся интервалам. Алгоритм 2 определяется последовательным правилом вычисления и этим частично определенным правилом выбора. Но метод определения точки расщепления снова остается неуточненным.

Алгоритм 3

Алгоритм 2 может извлечь выгоду из сделанных ранее приближений, которые позволяют получить лучшее приближение, чем приписанная им доля предела погрешности EPS. Если вблизи A приближения хорошие, это может уменьшить работу при разбиении интервала вблизи B. Однако это означает предпочтение, оказываемое левой части интервала. Если приближения вблизи B очень хороши, мы не используем этого преимущества, чтобы уменьшить объем работы при интегрировании вблизи A.

Чтобы обеспечить симметричность обработки, нам нужно иметь правило вычисления, которое может выбирать между выполнениями рекурсивных вызовов процедуры расщепления. Давайте представим последовательность шагов такого вычисления в режиме сопрограмм, в котором правило выбора всегда выбирает процедуру расщепления. Общее состояние такого вычисления представляется в виде множества вызовов

$$S' = \{I(A, m_1, f, y_1, e_1), I(m_1, m_2, f, y_2, e_2), \ldots, I(m_k, B, f, y_k, e_k)\}.$$

Правило вычисления может выбрать любой из этих вызовов. Если оно выбирает вызов $I(m_l, m_{l+1}, f, y_l, e_l)$ и правило выбора снова обращается к процедуре расщепления, мы эффективно расширяем возможное множество интервалов приближения

$$S = \{\langle A, m_1\rangle, \ldots, \langle m_k, B\rangle\}$$

до большего множества, включающего два новых интервала $\langle m_l, m\rangle$, $\langle m, m_{l+1}\rangle$ вместо $\langle m_l, m_{l+1}\rangle$. Здесь m — промежуточная точка, определяемая недетерминированным исполнением $m_l < m < m_{l+1}$ процедуры расщепления. Очевидно, что правило выбора должно выбирать этот путь вычисления только тогда, когда предельная погрешность EPS не может быть достигнута при исполнении каждого из вызовов S', использующих процедуру приближения. Иначе говоря, процедура расщепления должна использоваться только тогда, когда сумма оценок погрешностей для множества интервалов S больше, чем EPS. Если невозможно уложиться в пределы допустимой погрешности, то требуется большее число расщеплений интервала. При выборе вызова для исполнения правило вычисления выбирает интервал для расщепления. Мы должны выбирать интервал с наибольшей оценкой погрешности. Правило вычисления должно выбирать $I(m_l, m_{l+1}, f, y, e)$ только тогда, когда $err(m_l, m_{l+1}, f)$ является максимальной среди оценок погрешностей для интервалов из S. Эти соображения приводят нас к алгоритму 3. Правило вычисления для этого алгоритма всегда вначале исполняет вызов $a < m < b$ процедуры расщепления, однако затем оно работает в режиме сопрограмм с рекурсивными вызовами. На каждом шаге вычисления оно выбирает вызов с наибольшей оценкой погрешности. Правило выбора раз за разом выбирает процедуру расщепления до тех пор, пока оценка погрешности для всех невыполненных вызовов не станет меньше, чем EPS. Затем оно последовательно выбирает процедуру приближения. За-

метим, что этот алгоритм фокусируется на наиболее плохих точках интервала. Мы полагаем, что он характеризует новое семейство алгоритмов интегрирования.

6. ОБЪЕДИНЕНИЕ ЛОГИКИ С УПРАВЛЕНИЕМ

В предыдущем разделе мы привели неформальное описание управляющих компонент, из которых следуют наши три алгоритма. Если мы хотим иметь исполнимые алгоритмы, эти описания управления надо формализовать. Одна возможность заключается в разработке формальных обозначений для описания управления. Аннотации в языке IC-PROLOG [7] являются одним из таких подходов. Однако они слишком слабы, чтобы выразить те концепции управления, которые нам нужны. Хейс [2] предложил использовать другую логическую программу для спецификации управления, и предложения Галлэра и Лассера [8] — первые шаги в этом направлении. Мы считаем, что этот метод обещает многое как метод формальной спецификации алгоритмов. Остается посмотреть, можно ли реализовать эффективные средства исполнения дуальной логической программы. Альтернативой прямому исполнению (Л + У)-описания алгоритма, особенно когда управление сложное, является логическое преобразование. Это есть переформулировка заданной пары Л + У в новую Л′ + У′, определяющую тот же алгоритм, но имеющую более простое, более обычное управление. Грегори [9] применил этот метод к паре Л + У, заданной аннотированной программой на языке IC-PROLOG. Целью этого преобразования была программа, которую можно было бы исполнять с помощью последовательного правила вычисления и простого правила выбора. Именно такую тактику мы и примем.

Переформулировка алгоритма 1

Усиление

В алгоритме 1 правило выбора выбирает процедуру расщепления

$$I(a, b, f, y_1 + y_2, e_1 + e_2) \leftarrow a < m < b, I(a, m, f, y_1, e_1), I(m, b, f, y_2, e_2)$$

всякий раз, когда вызов I(a, b, f, y, e) таков, что оценка погрешности err(f, a, b) для интервала ⟨a, b⟩ больше, чем ее доля EPS(b — a)/(B — A) в общей оценке погрешности EPS. В противном случае правило выбирает процедуру приближения

$$I(a, b, f, quad(f, a, b), err(f, a, b)).$$

Мы можем включить это условие выбора в каждую из процедур с помощью усиления. Усиление заключается в добавлении дополнительного вызова, или с логической точки зрения в добавлении дополнительного предусловия к процедуре, выраженной логической программой. Эта тактика никогда не влияет на частичную корректность программы, так как если $B \leftarrow A$ — истинное утверждение, то и $B \leftarrow C, A$ тоже.

Усиление даст нам две новые процедуры:

$$I(a, b, f, y_1 + y_2, e_1 + e_2) \leftarrow$$
$$\text{err}(f, a, b) \geqslant \text{EPS}\,(b - a)/(B - A):\ a < m < b,$$
$$I(a, m, f, y_1, e_1),\ I(m, b, f, y_2, e_2). \qquad (8)$$

$$I(a, b, f, \text{quad}(f, a, b), \text{err}(f, a, b)) \leftarrow$$
$$\text{err}(f, a, b) \leqslant \text{EPS}\,(b - a)/(B - A):. \qquad (9)$$

Мы можем достичь эффекта нашего первоначального правила выбора, если понимать новые вызовы как охраны (ср. [10]). Можно использовать более простое правило выбора: выбрать процедуру с истинной охраной, где охраной будет первый вызов процедуры. Мы поставили знак «:» после вызова охраны, чтобы указать его особую роль.

Обобщение

Здесь имеется недостаток. Мы упростили правило выбора ценой введения констант в программу A, B, EPS. У нас есть программа, которая работает только для специфического запроса

$$y:\ I(A, B, F, y, e) \quad e \leqslant \text{EPS}. \qquad (5)$$

При новом запросе с другими A, B и EPS нужно переписывать программу заново. Ранее A, B и EPS были параметрами для правила выбора. Чтобы снова достичь нашей первоначальной общности, мы должны заменить охраны в (8) и (9) на охраны, которые имеют тот же эффект, однако не обращаются к A, B или EPS. Сначала заметим, что при исполнении запроса (5) с использованием (8) и (9) суммирование оценок погрешностей для получения значения e излишне. Мы можем, следовательно, использовать свойство обратимости (см. [4] или [11]) логических программ и использовать аргумент погрешности e не для того, чтобы выдавать оценку погрешности, а для получения предела погрешности EPS. Вместо запроса (5) можно использовать запрос

$$y:\ I(A, B, F, y, \text{EPS}). \qquad (10)$$

Проанализируем использование (8) для ответа на этот новый запрос. Напомним, что $e_1 + e_2$ в заголовке процедуры — это сокращение для

$$I(a, b, f, y_1 + y_2, e) \leftarrow e = e_1 + e_2, \ldots,$$

где равенство появляется в процедуре в явном виде. Если e воспринимать как вход, это равенство можно использовать для расщепления заданного предела погрешности в пределы погрешностей для рекурсивных вызовов. Правильным расщеплением будет $e_1 = e(m-a)/(b-a)$ и $e_2 = e(b-m)/(b-a)$. Так как из этих двух равенств следует $e = e_1 + e_2$, то процедура

$$\begin{aligned} &I(a, b, f, y_1 + y_2, e) \leftarrow \\ &\quad \mathrm{err}(f, a, b) > \mathrm{EPS}(b-a)/(B-A): \quad a < m < b, \\ &\quad e_1 = e(m-a)/(b-a), \quad e_2 = e(b-m)/(b-a), \\ &\quad I(a, m, f, y_1, e_1), \quad I(m, b, f, y_2, e_2) \end{aligned} \tag{11}$$

является частным случаем (8).

Теперь заметим, что при первом использовании процедуры для ответа на запрос (10) $e = \mathrm{EPS}$, $a = A$ и $b = B$. Поэтому при первом использовании охрану можно заменить на проверку $\mathrm{err}(f, a, b) > e$. Точно так же e_1 и e_2, выдаваемые вызовами $e_1 = e(m-a)/(b-a)$ и $e_2 = e(b-m)/(b-a)$, — это в точности правильные доли EPS, необходимые при исполнении охраны для рекурсивных вызовов. Это свойство выполняется, как только e имеет корректное значение на входе процедуры (11). Таким образом, для вычисления запроса (10) следующее обобщение процедуры (11)

$$\begin{aligned} &I(a, b, f, y_1 + y_2, e) \leftarrow \mathrm{err}(f, a, b) > e: \quad a < m < b, \\ &\quad e_1 = e(m-a)/(b-a), \quad e_2 = e(b-m)/(b-a), \\ &\quad I(a, m, f, y_1, e_1), \quad I(m, b, f, y_2, e_2) \end{aligned} \tag{12}$$

приводит к такому же поведению правила выбора, что и (11). Более того, (12) по-прежнему является истинным описанием отношения интегрирования, определяемого (4). Аналогично, обобщение процедуры (9)

$$I(a, b, f, \mathrm{quad}(f, a, b), e) \leftarrow \mathrm{err}(f, a, b) \leqslant e:, \tag{13}$$

используемое в сочетании с (12), будет правилом выбора, эквивалентным процедуре (9). Оно также по-прежнему является истинным утверждением о нашем отношении интегрирования. Обобщенная программа, включающая процедуры (12) и (13), должна использоваться с запросом вида (10).

В совокупности с последовательным правилом вычисления и нашим новым проверяемым охраной правилом выбора она также определяет алгоритм 1.

Детерминированные примеры алгоритма 1

Для получения детерминированного примера алгоритма 1 мы должны ограничить недетерминированный выбор m в вызове $a < m < b$. Этот вызов пытается угадать интервал $\langle a, m\rangle$, который будет проверяться охраной для рекурсивного вызова $I(a, m, f, y_1, e_1)$. Если это предположение относительно m не удовлетворяет охране $\mathrm{err}(f, a, m) \leqslant e_1$, то снова используется процедура расщепления и применяется новый вызов $a < m' < m$ для того, чтобы передвинуть промежуточную точку ближе к a. Такое сужение интервалов продолжается с каждым рекурсивным вызовом процедуры расщепления и оканчивается, когда достигается интервал, на котором может применяться приближение. Таким образом, детерминирование правила выбора промежуточной точки m каждого вызова $a < m < b$ вместе с неограниченным рекурсивным вычислением $I(a, m, f, y_1, e_1)$ образует алгоритм нахождения подынтервала, на котором может применяться приближение.

Иметь одновременно и произвольный выбор промежуточной точки m, и произвольную глубину рекурсии — ненужная роскошь. Мы можем обойтись правилом выбора, которое всегда выдает фиксированную промежуточную точку, скажем $(a + b)/2$. Но мы можем, напротив, настаивать на том, чтобы правило выбора для исполнения $a < m < b$ давало точку m такую, что $I(a, m, f, y_1, e_1)$ всегда можно исполнять, используя правило приближения. Иначе говоря, можно фиксировать правило выбора так, чтобы оно давало m, такую, что $\mathrm{err}(f, a, m) \leqslant e(m - a)/(b - a)$. Эти две альтернативы приводят к двум детерминированным примерам алгоритма 1.

Принятие первой альтернативы дает нам логическую программу

$$I(a, b, f, y_1 + y_2, e) \leftarrow \mathrm{err}(f, a, b) > e:$$
$$I(a, (a + b)/2, f, y_1, e/2),\quad I((a + b)/2, b, f, y_2, e/2), \tag{14}$$
$$I(a, b, f, \mathrm{quad}(f, a, b), e) \leftarrow \mathrm{err}(f, a, b) \leqslant e:. \tag{15}$$

Принятие второй альтернативы дает нам программу, использующую хвостовую рекурсию:

$$I(a, b, f, \mathrm{quad}(f, a, m) + y, e) \leftarrow \mathrm{err}(f, a, b) > e:$$
$$\mathrm{approx\text{-}point}(a, m, b, f, e),$$
$$I(m, b, f, y, e(b - m)/(b - a)). \tag{16}$$
$$I(a, b, f, \mathrm{quad}(f, a, b), e) \leftarrow \mathrm{err}(f, a, b) \leqslant e:. \tag{17}$$

Процедуру (16) можно получить из процедуры (12) заменой вызова $a < m < b$ на новый вызов approx-point(a, m, b, f, e). При условии, что исполнение этого вызова эквивалентно исполнению $a < m < b$ с помощью правила выбора, которое всегда дает m, такое, что $err(f, a, m) \leqslant e(m - a)/(b - a)$, можно заранее вычислять рекурсивный вызов $I(a, m, f, y_1, e_1)$, используя процедуру приближения. Мы можем так преобразовать программу, поскольку знаем, что процедура приближения будет всегда применяться к этому вызову. Ниже приведена программа для approx-point с требуемым свойством:

approx-point(a, (a + b)/2, b, f, e) ← err(f, a, b) ⩽ e/2: (18)

approx-point(a, m, b, f, e) ← err(f, a, b) > e/2:

approx-point(a, m, (a + b)/2, f, e/2). (19)

Алгоритм 4

Этот алгоритм включает процедуры (14), (15), правило последовательного вычисления и проверяемое охраной правило выбора.

Алгоритм 5

Этот алгоритм включает процедуры с (16) по (19) с тем же управлением.

Оба эти детерминированные примеры семейства алгоритмов, определяемых алгоритмом 1, находят первый подынтервал $\langle A, m_1\rangle$ приближения заданного интервала $\langle A, B\rangle$ последовательностью бинарных расщеплений $\langle A, B\rangle$. Алгоритм 4 затем находит следующий подынтервал, расщепляя, если необходимо, интервал $\langle m_1, m\rangle$ той же длины, что и $\langle A, m_1\rangle$ (этот интервал используется при связывании аргументов во втором рекурсивном вызове). Алгоритм 5, напротив, находит следующий подынтервал бинарным расщеплением всего оставшегося интервала $\langle m_1, B\rangle$. Алгоритм 5 делает, таким образом, бо́льшую работу при нахождении последовательности подынтервалов аппроксимации. Его преимущество в том, что он использует хвостовую рекурсию, т. е. является скорее итеративным, чем рекурсивным алгоритмом.

Переформулировка алгоритма 2

Единственное различие между алгоритмами 1 и 2 заключалось в том, что алгоритм 2 принимал в расчет накопленную оценку погрешности e для подынтервала $\langle A, M\rangle$, на котором уже применялось приближение. Предел погрешности для интеграла на $\langle M, B\rangle$ принимался затем равным (EPS — e).

Мы не можем далее непосредственно преобразовывать программы, полученные для алгоритмов 4 и 5, чтобы включить это более сложное правило выбора. Это объясняется тем, что они больше не суммируют оценку погрешности. Нам требуется программа, которая и суммирует оценки погрешности, и распределяет предел погрешности. Нужна программа для расширенного отношения I′, определяемого следующим образом:

$$I'(a,\ b,\ f,\ y,\ e,\ e') \Leftrightarrow \left| \int_a^b f\,dx - y \right| \leqslant e \leqslant e'. \qquad (20)$$

Следующие модификации процедур (14) и (15), которые соединяют эти процедуры с первоначальными процедурами (6) и (7), являются истинными утверждениями об этом новом отношении:

$I'(a,\ b,\ f,\ y_1 + y_2,\ e_1 + e_2,\ e) \leftarrow \mathrm{err}(f,\ a,\ b) > e$:

$I'(a, (a + b)/2,\ f,\ y_1,\ e_1,\ e/2),\ I'((a + b)/2,\ b,\ f,\ y_2,\ e_2,\ e/2).$ (21)

$I'(a,\ b,\ f,\ \mathrm{quad}(f,\ a,\ b),\ \mathrm{err}(f,\ a,\ b),\ e) \leftarrow \mathrm{err}(f,\ a,\ b) \leqslant e$: (22)

При получении ответа на запрос

⟨y, e⟩: I′(A, B, F, y, e, EPS)

процедуры (21) и (22) определяют усовершенствованный вариант алгоритма 4, который выдает тот предел на погрешности, который встретился в действительности. Теперь можно преобразовать (21) и (22), чтобы получить детерминированный пример алгоритма 2. Это преобразование заключается просто в подстановке $e - e_2$ вместо e/2 во втором рекурсивном вызове.

Алгоритм 6

$I'(a,\ b,\ f,\ y_1 + y_2,\ e_1 + e_2,\ e) \leftarrow \mathrm{err}(f,\ a,\ b) > e$:

$I'(a, (a + b)/2,\ f,\ y_1,\ e_1,\ e/2),\ I'((a + b)/2,\ b,\ f,\ y_2,\ e_2,\ e - e_1).$ (23)

$I'(a,\ b,\ f,\ \mathrm{quad}(f,\ a,\ b),\ \mathrm{err}(f,\ a,\ b),\ e) \leftarrow \mathrm{err}(f,\ a,\ b) \leqslant e$:. (24)

Здесь используется последовательное правило вычисления и проверяемое охраной правило выбора. Мы просим читателя проверить, что это последнее преобразование не изменяет того, что эти процедуры являются истинными утверждениями об отношении, определяемом в (20). Более того, так как предел погрешности, установленный для второго рекурсивного вызова, есть $e - e_1$ (что больше или равно, чем e/2), то в этот алгоритм включено более сложное правило выбора из алгоритма 2.

Точно такое же преобразование можно применить к процедурам (16) и (17), чтобы получить второй детерминированный пример алгоритма 2. Как и алгоритм 5, он является итеративным.

Алгоритм 7

$$I'(a, b, f, \text{quad}(f, a, m) + y, \text{err}(f, a, m) + e', e) \leftarrow$$
$$\text{err}(f, a, b) > e: \quad \text{approx-point}(a, m, b, f, e),$$
$$I'(m, b, f, y, e', e - \text{err}(f, a, m)), \tag{25}$$
$$I'(a, b, f, \text{quad}(f, a, b), \text{err}(f, a, b), e) \leftarrow \text{err}(f, a, b) \leqslant e: \tag{26}$$

плюс процедуры для approx-point и обычные правила вычисления и выбора.

Переформулировка алгоритма 3

Преобразование алгоритма 3 в логическую программу, которая могла бы исполняться как последовательная программа с единственным проверяемым охраной правилом выбора, требует гораздо более радикального изменения логики. На каждом шаге такая последовательная программа должна быть в состоянии выбрать правильный вызов из множества вызовов, отложенных при выполнении в режиме сопрограмм. Множество оставшихся вызовов определяет текущее множество интервалов из

$$S = \{\langle A, m_1\rangle, \ldots, \langle m_k, B\rangle\}.$$

Это означает, что мы должны искать новую логическую программу не для отношения I', а для обобщения I'' этого отношения, в котором пара аргументов a, b заменяется на множество подынтервалов из $\langle a, b\rangle$. Новое отношение определяется следующим образом:

$$I''(s, f, y, e) \Leftrightarrow s = \{\langle m_0, m_1\rangle, \ldots, \langle m_k, m_{k+1}\rangle\}, \tag{27}$$

где $m_0 < m_1 < \ldots < m_{k+1}$ и $\left|\sum_{l=0}^{k} \int_{m_l}^{m_{l+1}} f dx - y\right| \leqslant e.$

Алгоритм 3 останавливался, когда сумма оценок погрешностей для интервалов из S была меньше, чем EPS — заданный предел погрешности. Если предположить, что значение EPS задано и передается процедуре аргументом e, то процедура

$$I''(s, f, \text{sumquad}(s, f), e) \leftarrow \text{sumerr}(s, f) \leqslant e \tag{28}$$

соответствует этому условию остановки, при условии, что функции sumquad и sumerr удовлетворяют спецификациям

$$\text{sumquad}(s, f) = \sum_{\langle m, m'\rangle \in s} \text{quad}(f, m, m') \tag{29}$$

и

$$\text{sumerr}(s, f) = \sum_{\langle m, m'\rangle \in s} \text{err}(f, m, m'). \tag{30}$$

Мы оставляем читателю проверку того, что (28) является истинным утверждением об отношении, определяемом в (27).

Когда сумма оценок погрешностей интервалов в S больше, чем заданный предел погрешности EPS, алгоритм 3 выбирает интервал с наибольшей оценкой погрешности и затем расщепляет этот интервал. Следующее рекурсивное описание является процедурой для такого расщепления:

$$I''(s, f, y, e) \leftarrow \text{sumerr}(s, f) \geqslant e:$$
$$\text{max-delete}(s, i, s'), \text{split}(f, i, i_1, i_2),$$
$$I''(\text{union}(s', i_1, i_2), f, y, e). \tag{31}$$

Здесь split(f, i, i_1, i_2) истинно, когда i_1, i_2 есть некоторое расщепление интервала i, а max-delete и union удовлетворяют следующим условиям:

$$\text{max-delete}(f, s, \langle a, b\rangle, s') \Leftrightarrow \langle a, b\rangle \in s \;\&\; s' = s - \{\langle a, b\rangle\}$$
$$\&\; \text{err}(f, a, b) = \max\{\text{err}(f, m, m') \mid \langle m, m'\rangle \in s\}, \tag{32}$$
$$\text{union}(s', \langle a, m\rangle, \langle m, b\rangle) = s' \cup \{\langle a, m\rangle, \langle m, b\rangle\}. \tag{33}$$

Теперь нам нужно найти процедуры для этих дополнительных отношений и функций. Они будут получать доступ к некоторому представлению множества S пар интервалов и преобразовывать его. Как и для всех алгоритмов, здесь существуют некоторые соотношения между сложностью представления множества и эффективностью обращения с ним. Первое приходящее на ум представление — это записывать *множество* интервалов в виде *списка* интервалов из этого множества. Недостаток такого представления заключается в том, что функция sumerr должна вычислять погрешность для каждого интервала при каждом охраняемом вычислении, и отношение max-delete должно так же раз за разом вычислять погрешности для каждого интервала.

Чтобы избежать перевычисления для охраняемого исполнения, мы можем представить множество интервалов как пару ⟨L, E⟩, где L — это множество интервалов, а E — сумма оценок погрешностей для всех интервалов из L. В этом случае sumerr(s, f) становится функцией с простым доступом. Для того чтобы ускорить доступ к интервалу с наибольшей

оценкой погрешности, мы можем упорядочить интервалы из L по убыванию величины оценок погрешности. Добавление в S нового интервала означает теперь вставку его в позиции, определяемой его оценкой погрешности. Для того чтобы избежать перевычисления оценок погрешностей интервалов из L при вставке, мы должны накапливать оценку погрешности вместе с интервалом. Однако вычисление err(f, a, b) для интервала ⟨a, b⟩ включает нахождение значений f(a), f(b) и f((a + b)/2). Так как эти же значения будут необходимы при остановке функции sumquad, и во всяком случае значения f(a) и f(b) будут необходимы для вычисления новых оценок погрешности в случае расщепления интервала, нам следует записывать эти значения вместе с оценкой погрешности. Это гарантирует, что интегрируемая функция вычисляется в точности один раз в каждой точке интервала. Каждый интервал ⟨a, b⟩ из множества *S* теперь представляется шестеркой (a, f(a), f((a + b)/2), f(b), b, err(f, a, b)) в списке L. Список L упорядочен по убыванию значений аргумента, содержащего оценку погрешности. В общем представлении S как пары ⟨L, E⟩ величина E является суммой погрешностей, записанных в кортежах из L.

В соответствии с этим представлением приведенные ниже процедуры определяют отношение max-delete и расширяют отношение «x = y» (x имеет значение y), чтобы работать с errsum, sumquad и union:

$$e = \text{errsum}(\langle l, e\rangle) \tag{34}$$

$$\text{QUAD}(a, fa, fm, fb, b) + \text{sumquad}(f, \langle l, e\rangle) =$$

$$\text{sumquad}(f, \langle (a, fa, fm, fb, b, e'): l, e\rangle) \tag{35}$$

$$0 = \text{sumquad}(f, \langle \text{Nil}, e\rangle) \tag{36}$$

$$\text{max-delete}(f, \langle i : l, e\rangle, i, \langle l, e\rangle) \tag{37}$$

$$\langle \text{ord-insert}(i_2, \text{ord-insert}(i_1, l)), e\rangle = \text{union}(\langle l, e\rangle, i_1, e_l). \tag{38}$$

Здесь QUAD идентична нашей ранее определенной функции quad, за тем исключением, что она использует заранее вычисленные значения функций fa, fm, fb. Функция ord-insert вставляет интервал (a, fa, fm, fb, b, e) в список L, сохраняя порядок оценок погрешностей. Инфиксное двоеточие «:» — это конструктор списков, т. е. i : l — список с первым элементом («головой») i и остатком («хвостом») l, а «Nil» означает пустой список.

Процедура для «split» должна выполнять работу по поддержке представления интервалов шестерками. Если мы всегда используем бинарное расщепление, это можно осуществить в точности двумя новыми вычислениями функций в

промежуточных точках. Можно использовать следующую процедуру:

$$\begin{aligned}\text{split}(f, (a, fa, fm, fb, b, e), (a, fa, fm_1, fm, m, e_1),\\ (m, fm, fm_2, fb, b, e_2)) \leftarrow m = (b + a)/2,\end{aligned} \tag{39}$$

$$\begin{aligned}fm_1 &= f(a + (b - a)/4),\ fm_2 = f(b - (b - a)/4),\\ e_1 &= ERR(a, fa, fm_1, fm, m),\\ e_2 &= ERR(m, fm, fm_2, fb, b).\end{aligned}$$

Функция ERR является модификацией функции err, использующей заранее вычисленные значения интегрируемой функции.

Алгоритм 8

Он определяется процедурами (34)—(39), процедурами

$$I''(s, f, \text{sumquad}(s, f), e) \leftarrow \text{sumerr}(s, f) \leqslant e:$$

$$I''(s, f, y, e) \leftarrow \text{sumerr}(s, f) > e:$$

$$\text{max-delete}(s, i, s'),\ \text{split}(f, i, i_1, i_2),$$

$$I''(\text{union}(s', i_1, i_2), f, y, e)$$

и процедурой взаимодействия

$$I(a, b, f, y, e) \leftarrow fa = f(a),\quad fb = f(b),\quad fm = f((b + a)/2),$$

$$e' = ERR(a, fa, fm, fb, b),$$

$$I''(\langle(a, fa, fm, fb, e'): \text{Nil}, e'\rangle, f, y, e).$$

Они исполняются последовательно, при помощи правил выбора с охранами. Если их использовать для ответа на запрос

$$y: I(A, B, F, y, EPS),$$

они ведут себя в точности как детерминированный пример семейства алгоритмов, определяемых алгоритмом 3. Алгоритм 8 использует подходящее представление множества интервалов для минимизации числа вычислений интегрируемой функции.

7. ЗАКЛЮЧИТЕЛЬНЫЕ ЗАМЕЧАНИЯ

Спецификация нескольких различных алгоритмов заданием разных стратегий управления для фиксированной логической программы дает новое средство для анализа и сравнения алгоритмов. Последующее постепенное преобразование программы для того, чтобы встроить управление в логику,

также представляет собой новый подход к систематическому построению программ из их спецификаций. В настоящей статье мы проиллюстрировали эти идеи на примере из области численного интегрирования.

ЛИТЕРАТУРА

[1] Kowalski R. A. Predicate logic as programming language. — In: Information Processing'74, North Holland, 1974, p. 569—574.
[2] Hayes P. J. Computation and deduction. — Proc. 2nd Symp. on Math. Foundations of Computer Science, Czekoslovak Academy of Sciences, 1973.
[3] Pratt V. T. The competence/perfomance dichotomy in programming. — Proc. 4th ACM Sigart/Sigplan Symp. on principles of programming Languages, 1977, p. 194—200.
[4] Kowalski R. A. Algorithm = logic + control. — Comm. ACM, 1979, v. 22, p. 424—431.
[5] Davis P. J., Robinowitz P. Numerical integration. — Blaisdell Publishing Co., Waltham Mass., 1976.
[6] Lyness J. N. SQUANK (Simpson quadrature used adaptively, noise killed), Algorithm 379, Comm. ACM, 1970, v. 33, No 4.
[7] Clark K. L., McCabe F., Gregory S. IC-PROLOG language features.— In: Logic Programming. N. Y., Academic Press, 1982, p. 253—266. [Имеется перевод: Кларк К., Маккейб Ф., Грегори С. Средства языка IC-PROLOG, см. наст. сборник.]
[8] Gallaire H., Lassere C. Metalevel control for logic programs. — In: Logic Programming, Academic Press: N. Y., 1982, p. 173—188.
[9] Gregory S. Toward the compilation of annotated logic programs. — Research Report 80/16, Department of Computing, Imperial College, London, 1980.
[10] Dijkstra E. The discipline of programming. — Prentice-Hall, Englewood Cliffs, New Jersey, 1976. [Имеется перевод: Дейкстра Э. Дисциплина программирования.— М.: Мир, 1978.]
[11] Sickel S. Invertibility of logic programs. Technical Report 78—8—005. Information Sciences, University of California, Santa Crus, 1978.

СВОЙСТВА ОДНОГО ЯЗЫКА ЛОГИЧЕСКОГО ПРОГРАММИРОВАНИЯ [1])

А. ХАНССОН, С. ХАРИДИ, С.-А. ТЕРНЛУНД

Резюме. Разработана система логического программирования, основанная на натуральном выводе. Она включает в себя класс утверждений, в котором содержится класс хорновских дизъюиктов. Можно исполнять в качестве программ логические утверждеиия, которые прежде рассматривались как спецификации. В частности, язык содержит такие логические констаиты, как отрицанне, эквивалеитиость, квантор всеобщности и тождество. Можно определять фуиции, а также отношения, бесконечиые структуры данных и виртуальные классы. Стратегии вычисления обеспечивают управляющую иформацию. Стратегия вычислеиия, управляемая требоваииями, делает возможными конечные вычисления на бесконечиых структурах данных.

1. ВВЕДЕНИЕ

Логическое программирование, как оно представлено в Пролог-системах [7, 19], основано на хорновских дизъюнктах и процедурном толковании отношений [16]. Системой логического вывода является резолюция [21]. В отличие от этого наш язык основан на системе натурального вывода [20] [2]). Наши процедуры — это специальные случаи взаимодействующих агентов. Язык является языком логики первого порядка и имеет следующие общие с Прологом характерные особенности: (1) древовидные структуры данных; (2) недетерминированные программы, обрабатываемые с использованием автоматического бектрекинга; (3) отсутствие различия между входными и выходными данными; (4) логические переменные [3]), позволяющие программам работать с частично определенными структурами данных; (5) оптимизация хвостовой рекурсии.

---

[1]) A. Hansson, S. Haridi, S.-A. Tärnlund. Properties of a logic programming language. — Logic Programming (eds. Clark K. L., Tärnlund S.-A) N .Y.: Academic Press. 1982, p. 267—280.

[2]) К сожалению, подробиого ее описаиия иа русском языке, по-видимому, нет, очень краткое см. в: Вороиков А. А. Синтез логических программ. — Новосибирск: Институт математики СО АН СССР, Препринт № 24, 1986. — *Прим. ред.*

[3]) Логическими переменными в контексте логического программирования иазывают переменные, связываемые со зиачениями (получающие значения) посредством унификацин. — *Прим. ред.*

Его дополнительные свойства таковы: (1) истинно функциональная семантика для всех утверждений языка; (2) функции, определяемые посредством равенств или условных равенств; (3) определение отношений через эквивалентность; (4) отрицание; (5) виртуальные классы; (6) бесконечные структуры данных; (7) стратегия частичного упорядочения вычисления, при которой порядок обработки целей в утверждении зависит от конкретизации переменных в заголовке предложения; (8) управляемая требованиями стратегия вычисления, позволяющая писать программы, которые при выполнении ведут себя как сеть взаимодействующих агентов (процессов), связанных через потоки (сети могут быть статическими или динамически развертывающимися, циклическими или ациклическими, потоки — конечными или бесконечными, а агенты — детерминированными (функции) или недетерминированными (отношения)); (9) метод завершения программ, работающих с бесконечными структурами данных.

2. СИНТАКСИС ЯЗЫКА ПРОГРАММИРОВАНИЯ

Синтаксис языка будет показан главным образом на простых примерах, которые также проиллюстрируют некоторые понятия языка.

Различные свойства конечных и бесконечных структур данных привели нас к тому, что мы сделали наш язык двухсортным. Имеются канонические переменные, вместо которых могут подставляться конечные структуры данных, и неканонические переменные, вместо которых могут подставляться как конечные, так и бесконечные структуры данных. Значением канонического объекта является его имя (ср. [18]), канонические объекты — это наши структуры данных. В отличие от обычных языков программирования логические языки допускают неопределенные или частично определенные структуры данных, например список A.x, хвостом которого является неопределенный список. Это свойство и понятие тождества могут приводить к бесконечным структурам данных, которые можно представлять циклически. Например, уравнение $x = A.x.x$ имеет решение только в том случае, если допустить, что переменная x может обозначать бесконечный список A.A. ... .

Наш язык программирования строится из шести непересекающихся синтаксических категорий.

1) *Константы.* Константа записывается в виде последовательности букв и цифр, начинающейся с прописной буквы, но при этом число записывается обычным способом. Константы являются каноническими объектами.

2) *Структуры данных.* Структура данных — это составной терм, например Tree(Nil, A, z). Для удобства некоторые структуры данных записываются в инфиксной форме, например список A.B.x. Структуры данных являются каноническими объектами.

3) *Переменные.* Они бывают каноническими или неканоническими. Каноническая переменная записывается в виде последовательности букв и цифр с префиксом $ (знак доллара), например $x1. Неканоническая переменная записывается в виде последовательности букв и цифр, начинающейся со строчной буквы, например xx.

4) *Функции.* Функция определяется с использованием тождества, например conc(x.y, z) = x.conc(y, z).

5) *Отношения.* Они определяются посредством импликаций и эквивалентностей, например rev(x.y, w) ← rev(y, z) & conc(z, x.Nil, w).

6) *Логические константы.* ⊥ (абсурдность или ложность), → (импликация), & (конъюнкция), ↔ (эквивалентность), ∨ (дизъюнкция), ∀, ∃ (кванторы всеобщности и существования), = (тождество), — (отрицание).

Программа — это набор предложений. Предложение имеет вид: 'A', 'A ← B' или 'A ↔ B'. Символ A обозначает литерал, т. е. атомарную формулу или ее отрицание. Атомарная формула может быть равенством. Символ B обозначает произвольную логическую формулу.

3. ВЫЧИСЛЕНИЕ

Для управления построением выводов (вычислений) имеются два типа стратегии вычисления (computation rules).

Стратегии первого типа управляют последовательным вычислением. В этом случае если заданы два условия (процедурных вызова), то для одного из них дерево вывода строится полностью (вызов завершается) до того, как обрабатывается другое условие (инициируется другой вызов).

Стратегии второго типа управляют взаимодействующими агентами, связанными через каналы. Например, для двух заданных литералов деревья вывода строятся псевдопараллельным способом. Чередование зависит от выполняемых конкретизаций их общих переменных.

Для последовательной обработки порядок вычисления может быть определен в период компиляции или в период исполнения. По умолчанию порядок вычисления фиксируется в период компиляции. Динамическое упорядочение периода исполнения должно явно запрашиваться программистом. Для утверждения, целиком построенного из отношений, порядок

вычисления такой же, как в Прологе, — слева направо. Для утверждений, включающих в себя функции, применяется стратегия аппликативного упорядочения, которую иллюстрируют следующие примеры.

Пример 1

Приведенная ниже программа определяет функцию конкатенации списков.

$$\text{conc}(\text{Nil}, u) = u \qquad (1)$$
$$\text{conc}(x.y, u) = x.\text{conc}(y, u)$$

Ее можно использовать для определения функции быстрой сортировки quick-sort.

Пример 2

$$\text{quick-sort}(\text{Nil}) = \text{Nil} \qquad (2)$$
$$\text{quick-sort}(x.y) = \text{conc}(\text{quick-sort}(y_1), x.\text{quick-sort}(y_2)) \leftarrow \text{partition}(x, y, y_1, y_2)$$

Порядок обработки вызовов этого рекурсивного утверждения показан на рис. 1.

Стратегия частичного упорядочения

При динамическом упорядочении порядок вызовов зависит от конкретизации некоторых переменных в заголовке предло-

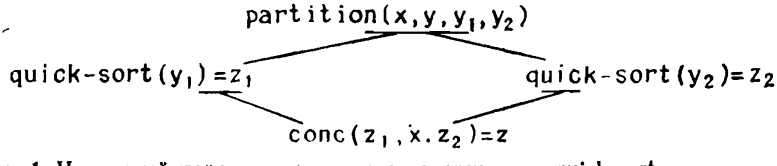


Рис. 1. Частичный порядок вычисления утверждения quick-sort.

жения. Эта конкретизация порождает частичный порядок вычисления на вызовах. Различные обращения к одному и тому же утверждению могут привести к различным частичным порядкам в зависимости от различных конкретизаций. В нашем компиляторе алгоритм динамического упорядочения применяется к утверждению только в том случае, если с утверждением связано метаправило конкретизации. В этом правиле определяется, какие переменные из заголовка утверждения представляют интерес. Тогда алгоритм упорядочения порождает различные частичные порядки, чтобы охватить разные комбинации конкретизированных и неконкретизированных переменных. Хотя это правило требует проверки в период исполнения, оно оказывается полезным в программах,

работающих с базами данных. Аналогичное правило имеется в языке IC-PROLOG, где программист, однако, явно задает разные порядки вычислений [3].

Пример 3

Проиллюстрируем эту идею на примере базы данных, принадлежащем Перейре и Порто [6]:

$$\text{запрос}(\text{студент}, \text{профессор}) \leftarrow \text{слушает}(\text{студент}, \text{курс}_1)\ \&\ \text{курс-в}(\text{курс}_1, \text{день}_1, \text{аудитория})\ \&\ \text{проф-читает}(\text{профессор}, \text{курс}_1)\ \&\ \text{слушает}(\text{студент}, \text{курс}_2)\ \&\ \text{курс-в}(\text{курс}_2, \text{день}_2, \text{аудитория})\ \&\ \text{проф-читает}(\text{профессор}, \text{курс}_2)\ \&\ \text{курс}_1 \neq \text{курс}_2, \quad (3)$$

где стратегия вычисления определяется конкретизацией переменной *профессор.*

Приведенное выше утверждение отвечает на запрос: «Существует ли студент, которому профессор читает два различных курса в одной и той же аудитории?». Ответ извлекается из базы данных, которая содержит информацию о

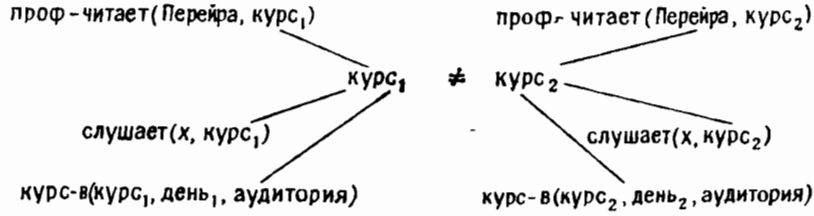


Рис. 2. Частичный порядок, соответствующий конкретизации для запроса (*х*, Перейра).

студентах, слушающих курсы, профессорах, читающих курсы, и расписании курсов по дням недели и аудиториям. Наш компилятор порождает два порядка вычисления в зависимости от того, конкретизируется ли переменная *профессор*. Теперь предположим, что мы хотим найти такого студента х, чтобы удовлетворялся запрос(х, Перейра). Тогда мы получим порядок вычисления, извлекаемый из частичного порядка, изображенного на рис. 2.

Стратегия, управляемая требованиями

Существует также стратегия вычисления, управляемая требованиями. При исполнении утверждения с использованием данной стратегии оно определяет сеть агентов, связан-

ных через иеограниченные буфера. Эти сети близки к потокам (streams) Кана и Маккуина [15], сетям Ландина [17] и потоковым (data flow) вычислениям Денниса [8]. Если процессор один, то агенты будут вести себя подобно сопрограммам. Следующее утверждение

$$n(x) = y \leftarrow p(x) = z \;\&\; c(z) = y \qquad (4)$$

при исполнении в режиме, управляемом требованиями, определяет простую сеть из двух агентов p и c, связанных через канал z. Когда инициируется агент n, создаются два экземпляра агентов для p и для c с переменной z, используемой в качестве связующего звена. Переменная z называется канальной переменной, агент p называется производителем z, а агент c — потребителем z. В каждой сети имеется агент (здесь c), который называется начальным заказчиком и управляет всей сетью. Вычисление производится следующим образом: c исполняется до тех пор, пока не потребуется дальнейшее уточнение z, затем управление передается p до того момента, пока p не произведет такое уточнение, после чего управление возвращается агенту c в точку приостановки. В утверждении, исполнение которого управляется требованиями, начальный заказчик начинает исполнение тела утверждения. Если сеть не содержит циклов, то начальным заказчиком может быть любой непроизводящий потребитель. В циклической сети с одним циклом или перекрывающимися циклами начальным заказчиком может быть любой агент. Если сеть состоит из непересекающихся циклов и один из циклов действует как производитель для одного или более циклов, то должен быть по крайней мере один цикл, действующий как непроизводящий потребитель. Любой агент в таком цикле может быть начальным заказчиком. Единственное ограничение состоит в том, что переменной можно назначить только одного производителя. Это представляется разумным, поскольку в противном случае могла бы возникнуть проблема: как решить, какой производитель должен удовлетворить требование.

В ациклической сети, если агенты являются функциями, вычисление, управляемое требованиями, сводится к эквивалентному виду ленивых вычислений [14], а в циклической сети оно эквивалентно вычислениям, выражаемым на языке потоковой обработки Кана и Маккуина [15]. В контексте логического программирования потоки также рассматривались в работах [1, 3]. Сопрограммы изучались также в работах [3, 10].

Стоит отметить, что наши агенты могут быть и недетерминированными отношениями, что может оказаться полезным

для спецификации задач параллельного программирования. В качестве примера покажем, как специфицировать и решить «задачу ограниченного буфера». Этот пример иллюстрирует, как можно использовать логику, чтобы специфицировать задачу параллельного программирования, и, кроме того, исполнить спецификацию, чтобы понаблюдать ее поведение.

Пример 4

Предположим, имеется ограниченный буфер размера N. Этот буфер можно рассматривать как агента, воспринимающего две последовательности: (1) последовательность команд записи вида «Write(x_1).Write(x_2). ...» (обозначенную через ws); (2) последовательность команд чтения вида «Read.Read. ...» (обозначенную через rs). Агент вырабатывает последовательность ответов (обозначенную через as), соответствующую последовательности команд чтения. Теперь можно формализовать понятие ограниченного буфера следующим образом:

$$\text{bounded-buffer}(ws, rs, as) \leftrightarrow \text{bmerge}(ws, rs, 0, s_1)\ \& \quad \text{buffer}(s_1, \langle \$u, \$u \rangle) = as \tag{5}$$

где предполагается, что вычисление управляется требованиями с использованием s_1 в качестве канальной переменной. Здесь описание bounded-buffer содержит двух агентов: bmerge, который образует из последовательностей команд чтения и записи одну смешанную последовательность, и buffer, который отвечает на операции чтения и записи. Более точно, bmerge — отношение, которое получает последовательность команд записи, последовательность команд чтения и число, указывающее количество элементов в буфере, и выдает смешанную последовательность чтения и записи. Это отношение характеризуется следующими двумя утверждениями:

$$\text{bmerge}(\text{Write}(x).ws, rs, i, \text{Write}(x).as) \leftrightarrow i < N\ \& \quad \text{bmerge}(ws, rs, i+1, as) \tag{6}$$

$$\text{bmerge}(ws, \text{Read}.rs, i, \text{Read}.as) \leftrightarrow i > 0\ \& \quad \text{bmerge}(ws, rs, i-1, as).$$

Функция buffer — это функция, которая получает последовательность команд чтения и записи. Она отвечает на команду Write(x), помещая элемент x в хвост внутренней очереди, представленной в виде списка из двух частей [4].

Она отвечает на команду Read, беря элемент из начала очереди и выдавая его в качестве ответа

$$\text{buffer(Write(x).s,}\langle\$v,x.\$w\rangle) = \text{buffer(s,}\langle\$v,\$w\rangle) \quad (7)$$

$$\text{buffer(Read.s,}\langle x.\$v,\$w\rangle) = x.\text{buffer(s,}\langle\$v,\$w\rangle).$$

Из этого примера видно, что мы получаем недетерминированные последовательности, вырабатываемые отношением bmerge.

Бесконечные структуры данных

Понятие бесконечной структуры данных может упростить написание программы, но существует проблема, как трактовать такие структуры с точки зрения вычислений. Неканонический терм может обозначать, вообще говоря, бесконечную структуру данных. Его нельзя преобразовать в канонические термы — преобразование потребовало бы бесконечно много времени. Но можно преобразовать неканонический терм в такой неканонический терм, что в нем увеличится каноническая часть, которую можно использовать при вычислении.

Пример 5

Можно определить бесконечный список целых чисел, начинающийся с 2:

$$\text{intfrom2}() = \text{inc}(2) \quad (8)$$
$$\text{inc}(x) = x.\text{inc}(x+1)$$

Отсюда следует, что $\text{intfrom2}() = 2.3.\text{inc}(4)$, где этот бесконечный список состоит из канонической части 2.3, за которой следует неканонический терм inc(4). Можно выбрать первые n положительных целых чисел, начиная с 2:

$$\text{n-integers}(n) = y \leftarrow \text{intfrom2}() = x \;\&\; \text{select}(n, x) = y, \quad (9)$$

где

$$\text{select}(0, z) = \text{Nil} \quad (10)$$

$$\text{select}(n, x.y) = x.\text{select}(n-1, y) \leftarrow n > 0$$

Вызов $\text{n-integers,n}) = y$ дает $y = 2.3.\text{Nil}$ при условии, что для предложения (9) используется стратегия вычисления, управляемая требованиями. Более того, вычисление может быть закончено, когда ни у одного потребителя не остается требований. В нашей системе, как будет видно в разд. 4, такое правило имеет теоретическое обоснование.

Когда мы предпочитаем думать о вычислении как об агенте, работающем с бесконечной структурой данных

(потоком), на которой никакой агент не может получить полного результата, вычисление, управляемое требованиями, дает удовлетворительные частичные результаты. Мы проиллюстрируем эту ситуацию далее более сложным примером из [15]. Программа вычисляет простые числа методом решета Эратосфена на потоке положительных целых чисел. Пример показывает, как сеть развертывается динамически в процессе вычисления.

Пример 6

$$\text{prime}() = \text{sift}(\text{intfrom2}()) \qquad (11)$$

где вычисление prime управляется требованиями, и sift — начальный заказчик,

$$\text{sift}(x.y) = x.\text{sift}(\text{filter}(x, y)),$$

где вычисление sift управляется требованиями, и sift — начальный заказчик,

$$\text{filter}(x, y.z) = y.\text{filter}(x, z) \leftarrow \text{mod}(y, x) \neq 0$$
$$\text{filter}(x, y.z) = \text{filter}(x, z) \leftarrow \text{mod}(y, x) = 0$$

Наша цель — вычислить поток простых чисел, т. е. $\text{prime}() = z$. Это вычисление никогда не завершится. Требующий агент sift будет последовательно требовать целое

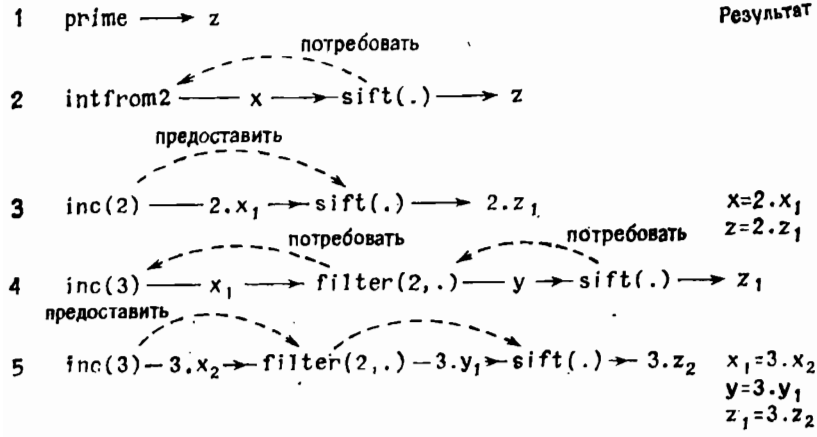


Рис. 3. Развертывание сети для первых простых чисел. Выходным потоком является $z = 2.3.\ z_2$.

число, которое будет отфильтровано, если оно не является простым. Каждая активация агента sift обращается к новому экземпляру агента filter за очередным распознанным простым числом. Здесь sift является примером сети, динамически развертывающейся в процессе вычисления. На рис. 3

приведено несколько шагов вычисления, иллюстрирующих эту ситуацию. Канальная переменная изображена сплошной линией, направленной от агента-производителя к потребителю, а ее позиция как аргумента в агенте-потребителе указана точкой.

Пример 7

Проиллюстрируем циклическую сеть агентов на примере из [9], в котором порождается поток, содержащий элементы

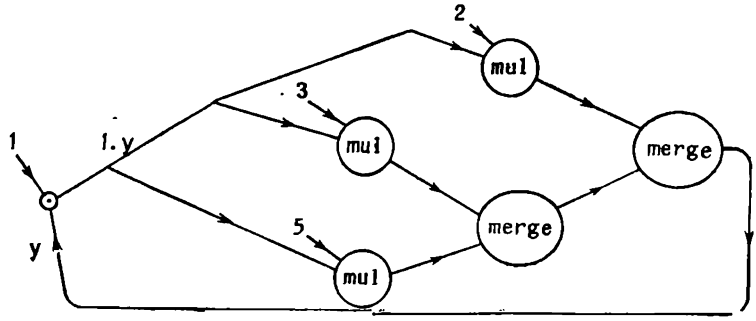


Рис. 4. Циклическая сеть, представляющая утвержденне p( ).

вида $2^a * 3^b * 5^c$, где a, b и c — положительные целые числа, причем элементы должны располагаться по возрастанию без пропусков и повторений. Общая идея решения состоит в том, что каждый порождаемый элемент потока (первый элемент равен 1) умножается на 2, 3 и 5, результаты помещаются в три соответствующих множества потенциальных элементов и наименьший элемент из этих трех множеств выбирается как следующий порождаемый элемент. В программе элементы этих множеств будут порождаться только по требованиям. Имеем:

$$p() = 1.y \leftarrow \text{merge}(\text{mul}(2, 1.y), \text{merge}(\text{mul}(3, 1.y), \text{mul}(5, 1.y))) = y$$

где вычисление p( ) управляется требованиями (12)

$$\begin{aligned} &\text{merge}(x.y, u.v) = x.\text{merge}(y, u.v) \leftarrow x < u \\ &\text{merge}(x.y, u.v) = u.\text{merge}(x.y, v) \leftarrow x > u \\ &\text{merge}(x.y, u.v) = x.\text{merge}(y, v) \leftarrow x = u \\ &\text{mul}(x, y.z) = x * y.\text{mul}(x, z) \end{aligned} \qquad (12)$$

Здесь merge выбирает наименьший элемент из двух возрастающих потоков элементов, а mul умножает первый элемент потока на число. Наша цель — вычислить поток p( ) = z.

Циклическая сеть, определяемая утверждением для p( ), будет последовательно выбирать наименьший элемент из трех подпотоков, порождаемых тремя агентами mul, по требованиям, исходящим от агентов merge. Эту сеть можно проиллюстрировать с помощью рис. 4.

4. ЗАВЕРШЕНИЕ ВЫЧИСЛЕНИЯ

В этом разделе мы покажем, как в режиме, управляемом требованиями, можно достигнуть завершения программ, работающих с бесконечной структурой данных. Для вычисления, управляемого требованиями, у нас есть правило: если производитель p для канальной переменной x является функцией или сводится к функции и все потребители переменной x завершены, то следует завершить p. Данное правило имеет оправдание в теории доказательств, которое мы сейчас поясним. Вернемся к примеру 5. Чтобы вывести равенство n-integers(2) = y, нам потребуется сделать два вывода: вывод intfrom2( ) = x и вывод select(2, x) = y. Последний вывод получается как обычно. Предыдущий можно получить следующим образом:

$$\dfrac{\dfrac{\dfrac{}{\mathrm{inc}(3) = 3.\mathrm{inc}(4)} \quad \dfrac{}{z = 3.w} \quad \dfrac{}{\mathrm{inc}(4) = w}}{\mathrm{inc}(2) = 2.\mathrm{inc}(3)} \quad \dfrac{}{x = 2.z} \quad \dfrac{}{\mathrm{inc}(3) = z}}{\mathrm{intfrom}\,2(\,) = \mathrm{inc}(2)} \quad \dfrac{}{\mathrm{inc}(2) = x}}{\mathrm{intfrom}\,2(\,) = x}$$

Вывод зависит от предположений о переменных x, z и w, но мы легко можем получить и вывод, не зависящий от предположений. Подставим inc(4) вместо w, 3.inc(4) вместо z и 2.3.inc(4) вместо x и получим вывод, зависящий только от утверждений (8), где intfrom2( ) = 2.3.inc(4), а select(2, x) = y теперь выглядит как select(2, 2.3.inc(4)) = y. Эта независимость следует из аксиомы тождества.

Тождество дает несколько упрощенную трактовку завершения по сравнению с ситуацией, когда имеются отношения. Пусть, например, вместо inc(n) = w имеется отношение inc(n, w), тогда, чтобы завершить исполнение, нам потребуется также аксиома $\forall x\, \exists w\, \mathrm{inc}(x, w)$. Для бесконечной структуры данных, например x = A.x, нужна также аксиома $\exists x\, (x = A.x)$.

Пример 8

Вычисление функции prime( ) из примера 6 никогда не завершится, но мы можем использовать ее, чтобы написать

программу — генератор простых чисел, — которая порождает все простые числа от 0 до n:

primegen(n) = y ← prime( ) = x & until(n, x) = y,
где вычисление primegen управляется требованиями, x — канальная переменная, а until — начальный заказчик,
until(n, x.plist) = Nil ← x > n
until(n, x.plist) = x.until(n, plist) ← x ⩽ n

Из тождества primegen(2) = y получаем последовательность утверждений, показанную на рис. 5.

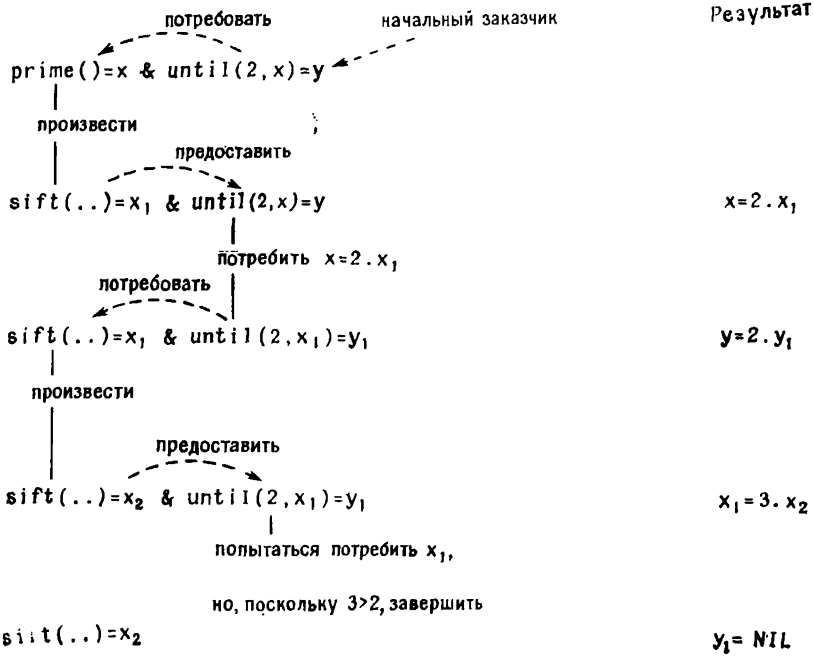


Рис. 5. Порождаются простые числа от 0 до 2, где $y = 2.\mathrm{Nil}$.

Все потребители переменной x_2 завершились, поэтому можно также завершить производителя.

5. ОТРИЦАНИЕ

Предикат A с отрицанием определяется следующим образом:

—A тогда и только тогда, когда $A \rightarrow \perp$ (13)

Правицем было показано [20], что для того, чтобы доказать —A, надо из предположения A получить доказательство абсурдности ($\perp$) и затем воспользоваться введением импликации. Следует заметить, что предположение A может содержать свободные переменные, которые могут быть связаны в процессе вывода, и таким образом мы найдем объекты, удовлетворяющие утверждению с отрицанием.

Чтобы вывести отрицание —A, часто используются определения через эквивалентность. Возьмем пример из [9] и дадим определение отношения maths-major, но сделаем это определение условным и скажем также, что объект, удовлетворяющий отношению maths-major, должен быть студентом. Таким образом, мы исключаем из определения нестудентов

$$\text{student}(x) \rightarrow (\text{maths-major}(x) \leftrightarrow \forall y\,(\text{maths-course}(y) \rightarrow \text{takes}(x, y))) \quad (14)$$

Пусть мы хотим показать, что —maths-major(x) истинно для некоторого конкретного значения x. Тогда предположим, что истинно maths-major(x), и попытаемся из данного предположения вывести абсурдность. Но прежде чем мы сможем воспользоваться определением (14) отношения maths-major, мы должны выбрать студента из базы данных, содержащей информацию о студентах, определяемого, например, утверждением

$$\text{student}(x) \leftarrow x = \text{Brown} \vee x = \text{Smith}.$$ Пусть $x = \text{Brown}$.

Тогда мы получаем доступ к эквивалентности из определения (14) и, в частности, к ее правой части. Если нам удастся установить ложность частей определения, мы выведем абсурдность. Поэтому попытаемся установить истинность maths-course(y) для некоторого конкретного значения y и ложность takes(Brown, y) для данного значения. Заметим, что отношение takes должно быть определено через эквивалентность, чтобы можно было распознать конкретные значения переменных, обращающие данное отношение в ложное.

С логической точки зрения такая трактовка отрицания отличается от прологовой, где, строго говоря, нет отрицания, а также от трактовки отрицания в языке IC-PROLOG, где оно понимается как неудача и нет условных эквивалентностей [5].

6. ВИРТУАЛЬНЫЕ КЛАССЫ

Хотелось бы, чтобы в языке программирования были множества. IC-PROLOG и Пролог имеют встроенные примитивы для работы с «множествами». Мы введем понятие виртуаль-

ного класса, которое слабее понятия множества, но зато его можно определить средствами языка.

$$\text{p-class}(l) \leftrightarrow \forall x(p(x) \leftrightarrow \text{memb}(x, l)), \quad (15$$

где

$$\begin{aligned} &\text{— } \text{memb}(u, \text{Nil}) \\ &\text{memb}(u, x.y) \leftrightarrow u = x \vee \text{memb}(u, y) \end{aligned} \quad (16$$

Недостаток определяемых таким образом p-классов заключается в том, что для каждого p нам нужно писать новое определение.

Пример 9

Определим класс сотрудников учреждения

$$\text{employee-class}(d, l) \leftrightarrow \forall x(\text{employee}(d, x) \leftrightarrow \text{memb}(x, l)) \quad (17$$

далее определим отношение employee:

$$\text{employee}(d, x) \leftrightarrow (d = \text{CS} \,\&\, (x = \text{John} \vee x = \text{Bill})) \vee (d = \text{EE} \,\&\, (x = \text{Mary} \vee x = \text{Jim})$$

Запрос employee-class(CS, l) дает l = John.Bill.Nil.

7. ЗАКЛЮЧИТЕЛЬНЫЕ ЗАМЕЧАНИЯ

Существует компилятор, который переводит наш язык в команды виртуальной логической машины. Компилятор написан на языке C с использованием системы построения трансляторов YACC. Прототипная реализация предназначена для систем PDP-11/UNIX и VAX/UNIX. По сравнению с Прологом язык существенно расширен, поэтому структура генерируемого кода и система команд нашей логической машины отчасти отличаются, а отчасти являются расширениями Пролог-машины, описанной Уорреном [23].

Различные аспекты данного проекта освещались ранее в работах [11, 12, 13, 22].

Благодарности

Мы хотим поблагодарить Кита Кларка за помощь в доработке представленной статьи.

Данная работа поддерживалась Шведским национальным комитетом по техническому развитию (STU).

ЛИТЕРАТУРА

[1] Bellia M., Degano P., Levi G. The call by name semantics of a claus languages with functions. In «Logic Programming» (Eds. Clark K. L Tärnlund S.-A.), Academic Press, 1982, pp. 281—298.

[2] Clark K. L. Negation as failure. In «Logic and Data Bases». (Eds. Gallaire H., Minker J.), Plenum Press, New York, 1978, pp. 293—322.
[3] Clark K. L., McCabe F. The Control facilities of IC-PROLOG. In Expert Systems in the Micro-Electronic Age (Ed. Michie D.), Edinburgh University Press, 1979.
[4] Clark K. L., Tärnlund S.-Å. A first order theory of data and programs. Proc IFIP-77, North Holland, pp. 939—944.
[5] Clark K. L., McCabe F., Gregory S. IC-PROLOG languages features. In «Logic Programming» (Eds. Clark K. L., Tärnlund S.-A.), Academic Press, 1982, pp. 253—266. [Имеется перевод: Кларк К. Л., Маккейб Ф., Грегори С. Средства языка IC-PROLOG, см. наст. сборник.]
[6] Coelho H., Gotta J. C., Pereira L. M. How to Solve it with Prolog. 2nd edition. Laboratorio Nacional de Engenharia Civil, Lisbon Portugal, 1980.
[7] Colmerauer A., Kanoui H., Pasero H., Roussel P. Un Système de Communication Homme-machine en Francais. Research report, 1973. Groupe Intelligence Artificielle, Université Aix — Marseille 11.
[8] Dennis J. On the Design and Specification of a common base language. In «Computers and Automata». Brooklyn Polytechnik Institute, 1971.
[9] Dijkstra E. W. A discipline of programming. Prentice-Hall, Englewood Cliffs, New Jersey, 1976. [Имеется перевод: Дейкстра Э. Дисциплина программирования. — М.: Мир, 1978.]
[10] Gallaire H., Lasserre C. Metalevel control for logic programs. In «Logic Programming» (Eds. Clark K. L., Tärnlund S.-Å.), Academic Press, 1982.
[11] Hansson A., Haridi S. Programming in a natural deduction framework. Proc. Functional Languages and their Implications for Computer Architecture, Göteborg, Sweden, 1981.
[12] Hansson A., Haridi S., Tärnlund S.-A. Some aspects of a logic machine prototype. In «Proceedings of the Logic Programming Workshop», Debrecen, Hungary (Ed. Tärnlund S.-A.), 1980, pp. 53—60.
[13] Haridi S. Logic Programming Based on a Natural Deduction System. Ph. D. Thesis. Department of Telecommunication and Computer Systems, The Royal Institute of Technology, Stockholm, Sweden, 1981.
[14] Henderson P., Morris J. H. A lazy evaluator. Proc. 3rd ACM Symp. on POPL, 1976, pp. 95—103.
[15] Kahn G., McQueen D. B. Coroutines and networks of parallel proceeses. Proc. IFIP-77.
[16] Kowalski R. A. Predicate Logic as programming language. Proc. IFIP-74 Congress. North-Holland, pp. 569—574.
[17] Landin P. The correspondence between ALGOL 60 and Church's lambda notation. Part 1, CACM 8 no. 2, 1965.
[18] Martin-Löf P. Constructive Mathematics and Computer Programming, Department of Mathematics, University of Stockholm, 1979.
[19] Pereira L. M. et al. User's Guide to DECsystem-10 Prolog. Divisao de Informatica, Laboratorio Nacional de Engenharia Civil, Lisbon, 1978.
[20] Prawitz D. Natural Deduction. A Proof-Theoretical Study. Ph. D. Thesis, Almqvist and Wiksell, Stockholm. 1965.
[21] Robinson J. A. A machine-oriented logic based on the resolution principle. Journal of the Association for Computer Machinery, 1965, 12, pp. 23—41.
[22] Tärnlund S.-A. A Programming Language Based on a Natural Deduction System, UPMALL Report, Computing Science Department, Uppsala University, 1981.
[23] Warren D. Implementing PROLOG — Compiling Logic Programs. 1 and 2. D. A. I. Research Report No 39, 40, University of Edinburgh, 1977.

СРЕДСТВА ЯЗЫКА IC-P ROLOG [1)]

К. КЛАРК, Ф. МАККЕЙБ, С. ГРЕГОРИ

Резюме. Эта статья знакомит, в основном на примерах, с базовыми средствами языка IC-PROLOG. Язык IC-PROLOG отличается от обычного Пролога тем, что в нем отсутствуют «внелогические» примитивы, такие, как усечение (/) и isvar, а также не разрешены добавление и удаление аксиом в процессе обработки запроса. С другой стороны, в язык введены в качестве примитивов отрицание и выражения для множеств, а кроме того, имеется богатый набор управляющих средств. В частности, программист может:

(1) сделать порядок исполнения вызовов процедуры зависимым от характера ее использования;

(2) инициировать (псевдо) параллельное исполнение множества процедурных вызовов, при котором совмещенные переменные служат каналами связи (между процессами) с односторонней проводимостью.

Управление задается с помощью аннотаций, не влияющих на декларативную семантику программы. Исполнение программы на языке IC-PROLOG представляет собой управляемый логический вывод в полном смысле слова.

1. ПРОЦЕДУРЫ И ЗАПРОСЫ

Процедура в IC-PROLOG — это импликация вида

$$B \leftarrow L_1 \,\&\, \ldots \,\&\, L_k, \quad k \geqslant 0,$$

где B — атомарная формула (атом), а каждый L_i — литерал, т. е. либо атомарная формула, либо ее отрицание. Атомы имеют вид $R(t_1, \ldots, t_n)$, где R — имя отношения, а $t_1, \ldots, t_n$ — термы.

Так же как и в реализации Пролога для DEC-10 [10], синтаксис можно изменять путем объявления операций. Это позволяет использовать для бинарных отношений инфиксную запись и писать t R t′ вместо R(t, t′).

В декларативной трактовке процедура понимается как импликация, причем все ее переменные считаются связанными кванторами всеобщности [2)]. Процедуры описывают

[1)] K. L. Clark, F. G. McCabe, S. Gregory «IC-PROLOG Language features». Logic Programming Academic Press, p. 253—266, 1982.

[2)] Исключением из этого правила является случай, когда в процедуре встречаются равенства, содержащие множества (см. разд. 2).

отношения над термами, которые являются структурами данных программы.

Запрос имеет одну из трех форм:

(1) : $L_1 \& \ldots \& L_k$
(2) t: $L_1 \& \ldots \& L_k$
(3) {t: $L_1 \& \ldots \& L_k$}

где t — терм, а $L_1, \ldots, L_k$ — литералы. Пусть $x_1, \ldots, x_n$ — все переменные, встречающиеся в конъюнкции $L_1 \& \ldots \& L_k$, а $y_1, \ldots, y_m$ — те из них, которые не входят в t. Тогда приведенные выше запросы нужно понимать следующим образом:

(1) для некоторых $x_1, \ldots, x_n$ имеет место $L_1 \& \ldots \& L_k$;
(2) такое t, что для некоторых $y_1, \ldots, y_m$ имеет место $L_1 \& \ldots \& L_k$;
(3) все t, такие что для некоторых $y_1, \ldots y_m$ имеет место $L_1 \& \ldots \& L_k$.

Обработка запроса. Как запросы, так и процедуры можно снабдить аннотациями (далее это будет показано на примерах), которые задают управление процессом обработки запроса. Если в программе и запросе нет аннотаций, вычисление происходит по правилам стандартного Пролога. Литералы (процедурные вызовы) запроса выбираются по одному слева направо. Неудача в доказательстве литерала L_i при значениях переменных, полученных в результате ранее проведенного доказательства $L_1 \& \ldots \& L_{i-1}$, приводит к бектрекингу (отходу назад). Ответы на множественные запросы вида (3) получаются путем бектрекинга после каждого успешного вычисления (доказательства) $L_1 \& \ldots \& L_k$ до тех пор, пока не будут исчерпаны все возможные пути доказательства. Процедуры выбираются для бектрекинга в том порядке, в котором они записаны в программе.

Негативные атомы. Негативные атомы, т. е. атомы со знаком отрицания, вычисляются по правилу «отрицание как неудача». Это означает, что —A считается доказанным, если все попытки доказать A терпят неудачу. Вызов —A оканчивается неудачей (т. е. соответствующий литерал считается ложным) только тогда, когда существует доказательство A, не связывающее ни одной из переменных атома [1]). Если A может быть доказан только при условии связывания некото-

[1]) Подразумевается связывание (binding) переменной с некоторым значением, т. е. придание ей значения (подстановка значения). При этом значениями служат термы. — *Прим. ред.*

рых из переменных, исполнение завершается с сообщением об ошибке. Как показано в [2], это правило вывода корректно при условии, что отношение R, фигурирующее в атоме A, а также все другие отношения, участвующие в определении R, полностью заданы процедурами программы. Используя негативный атом с отношением R, программист неявно указывает, что это условие выполнено.

Пример использования отрицания. Предположим, что нам надо определить, кто из детей Билла не состоит в браке. Для этого нужно сформулировать следующий запрос:

Все x, такие, что Билл является отцом x и неверно, что для некоторого y x состоит в браке с y.

В IC-PROLOG это можно выразить, только введя дополнительное отношение *в-браке*(x), определяемое условием:

«для некоторого y x состоит в браке с y»,

для которого мы хотим построить отрицание. Таким образом, мы введем процедуру:

в-браке(x) ← x *состоит-в-браке-с* y

Теперь можно сформулировать наш запрос следующим образом:

{x: *Билл отец* x & — *в-браке*(x)}

2. ВСТРОЕННЫЕ ОТНОШЕНИЯ

В языке IC-PROLOG есть встроенные отношения, реализующие арифметику (над натуральными числами), запись в файлы и чтение из них, а также построение списка всех решений, получаемых в ответ на запрос. Работа с файлами будет описана в разд. 4, а здесь мы проиллюстрируем использование арифметических отношений и выражений с множествами.

Арифметические отношения TIMES, PLUS, LESS [1]). Арифметические отношения в языке IC-PROLOG необычны для программиста, поскольку они работают как настоящие отношения без ограничений на характер использования. Например, с помощью встроенного отношения TIMES можно перемножать и делить числа, получать разложения числа на два множителя всеми возможными способами и даже порождать

[1]) TIMES(x, y, z) означает- «x·y = z», PLUS(x, y, z)— «x + y = z» LESS(x, y)— «x < y». — *Прим. ред.*

все тройки натуральных чисел, находящиеся между собой в этом отношении. Такая общность позволяет составлять элегантные арифметические программы, представляющие собой просто очевидные определения тех отношений, которые они вычисляют.

Абстрактно можно представлять себе, что арифметические отношения определены с помощью базы данных, содержащей утверждения, которые позволяют получать все элементы (кортежи) каждого такого отношения. Натуральные числа можно обозначать термами, использующими функцию следования s(x), дающую следующее за x натуральное число, или с помощью обычной десятичной записи. Таким образом, «s(s(0))» и «2» — синонимы, а терм «s(s(x))» обозначает любое число, большее единицы. Ответом на запрос

$$\{\langle s(s(x)), y\rangle\colon \mathrm{TIMES}(s(s(x)), y, 36)\ \&\ \mathrm{LESS}(s(x), y)\}$$

является множество $\{\langle 2, 18\rangle, \langle 3, 12\rangle, \langle 4, 9\rangle, \langle 6, 6\rangle\}$. Процедура

$$\textit{четно}(y) \leftarrow \mathrm{TIMES}(x, 2, y)$$

определяет свойство четности, и ее можно использовать как для проверки четности, так и для порождения четных чисел.

Процедура

$$x\ \textit{делит}\ z \leftarrow \mathrm{TIMES}(x, y, z)$$

задает отношение делимости и позволяет проверять на делимость, находить делители числа́ или чи́сла, кратные данному.

Процедура

$$\textit{делится}(z) \leftarrow s(s(x))\ \textit{делит}\ z\ \&\ -s(s(x)) = z$$

определяет свойство существования у числа хотя бы одного собственного делителя. Если использовать ее для получения ответа на запрос

$$\{z\colon \textit{делится}(z)\}$$

то она будет порождать бесконечное множество чисел {4, 6, 8, 9, 10 ...}, у которых есть собственные делители. (Точнее, числа будут порождаться до тех пор, пока вычисление не будет прервано или не произойдет выход за пределы диапазона чисел, представимых в конкретной ЭВМ.)

И наконец, процедура

$$\textit{простое}(z) \leftarrow \mathrm{LESS}(1, z)\ \&\ -\textit{делится}(z)$$

определяет свойство числа быть простым, а процедура

$$x\ \textit{простой-делитель}\ z \leftarrow x\ \textit{делит}\ z\ \&\ \textit{простое}(x)$$

задает отношение «x — простой делитель z». По запросу

{u: *простое*(u)}

порождаются все простые числа, а запрос

{u: u *простой-делитель* 100}

дает все простые делители числа 100. Определения отношений «*простое*» и «*простой-делитель*» — это по существу спецификации данных отношений. Язык IC-PROLOG позволяет, хотя и не слишком эффективно, вычислять по таким спецификациям элементы определяемых отношений.

Конструктор множества. Конструктор множества представляет собой равенство вида:

$$x = \{t: A\}, \text{ где } t \text{ — терм, а } A \text{ — атом.} \qquad (1)$$

Его вхождение в запрос или в процедуру логически эквивалентно следующему неатомному условию:

$$\forall u\,(u \in x \leftrightarrow \exists y_1, \ldots, y_k\,(u = t \,\&\, A)), \qquad (2)$$

где $y_1, \ldots, y_k$ — «локальные» переменные, встречающиеся только во фрагменте t : A. Например, процедура

p *мать-детей* l ← *женщина* (p) l = {q : q *ребенок* p}

эквивалентна импликации:

мать-детей l ← *женщина* (p) & ∀ u (u ∈ l ↔ ∃ q (u = q & p *ребенок* p))

Переменная p в этом примере не локальна для конструктора множества, так как встречается в остальной части процедуры. Эквивалентности с кванторами всеобщности, подобные рассмотренной, можно выражать непосредственно в логическом языке, описанном в работе [6].

Конструктор множества (1) используется для того, чтобы порождать связывания переменной[1]) x и ни для чего больше. С его помощью нельзя породить связывание никакой другой свободной переменной, входящей в эквивалентность (2), или проверить, что список удовлетворяет этому условию.

Каждый элемент списка, порождаемого для x, — это некоторая конкретизация ts терма t[2]). Через s здесь обозначено множество связываний локальных переменных $y_1, \ldots, y_k$, такое, что As истинно. Список для x получается путем нахождения всех решений для запроса t : A. Каждая конкретизация ts терма t, являющаяся решением, включается в спи-

[1]) См. выше примечание о связывании переменной. — *Прим. ред.*

[2]) То есть результат подстановки в t вместо переменных термов, заданных в s. — *Прим. ред.*

сок x. Так же как и в случае с отрицанием, такой способ вычисления корректен при условии, что отношение A полностью определено программой. Заметим, что этот способ не обязательно порождает наименьший список для x, удовлетворяющий условию (2). Так происходит из-за того, что различные пути при обработке запроса t : A могут приводить к одной и той же конкретизации t′ терма t. Здесь в списке связываемом с x, оказываются несколько экземпляров t′.

Примеры использования

(а) {⟨s, l⟩: студент(s) & l = {u: s посещает u}}. По этому запросу порождается множество пар вида ⟨студент, список посещаемых им курсов⟩. Согласно приведенному выше правилу, это эквивалентно следующему:

{⟨s, l⟩: *студент*(s) & ∀u(u ∈ l ↔ s *посещает* u)}

(б) l *список-простых-делителей* x ←

l = {u: u *простой-делитель* x}

Так определяется отношение между числом x и списком его простых делителей l. Эквивалентное определение:

l *список-простых-делителей* x ←

∀u(u ∈ l ↔ u *простой-делитель* x)

Из-за ограничений на использование конструктора множества эту процедуру можно применять только для получения списка простых делителей некоторого заданного числа x.

3. СВЯЗЬ УПРАВЛЕНИЯ С ИСПОЛЬЗОВАНИЕМ

Как уже упоминалось, стандартная стратегия управления в IC-PROLOG заключается в вычислении конъюнкции условий, образующих запрос, слева направо. Это же правило действует при вычислении условий (вызовов) в правой части какой-либо процедуры. К сожалению, не всегда можно найти порядок вычисления этих условий, подходящий для всех вариантов использования процедуры.

Рассмотрим, например, следующее определение отношения «x *имеет-потомка* y».

x *имеет-потомка* y ← x *родитель* y
x *имеет-потомка* y ← x *родитель* z & z *родитель* y
x *имеет-потомка* y ← x *родитель* z &
z *имеет-потомка* w & w *родитель* y

Если с помощью этих процедур отыскивать потомков по запросу вида

{y: Том *имеет-потомка* y},

то порядок расположения процедур приводит к поиску путем бектрекинга, отправной точкой которого служит заданный объект «Том». Первая процедура находит всех детей Тома, вторая — его внуков, находя детей его детей. И наконец третья отыщет всех остальных потомков путем поиска всех детей каждого потомка каждого из его детей.

Если же эти процедуры применить для поиска предков по запросу вида:

{x: x *имеет-потомка* Билл},

то порядок условий приведет к весьма неэффективному поиску. Так, вторая процедура будет определять дедушек и бабушек Билла не как родителей его родителей, а путем просмотра всех пар, отношения «x *родитель* y» до тех пор, пока не будет обнаружен кто-либо из родителей Билла. Для более эффективного поиска условия в правой части второй процедуры нужно поменять местами. Условия, входящие в третью процедуру, для эффективного поиска предков также следует расположить в обратном порядке.

Один из способов решения этой проблемы в стандартном Прологе заключается в том, чтобы определить обратное отношение «x *имеет-предка* y» при помощи процедур, в которых условия расположены в требуемом порядке. Сделав это, можно использовать новое отношение вместо прежнего в тех случаях, когда требуется поиск предков. Однако таким путем мы вводим логически избыточное отношение, и, кроме того, нарушается свойство обратимости отношений, присущее логическому программированию. Другая возможность в случае стандартного Пролога состоит в использовании метауровневого примитива isvar, который проверяет, связана ли переменная. В этом случае вторая из процедур для отношения «x *имеет-потомка* y» превращается в две процедуры:

x *имеет-потомка* y ←
    isvar(y) & x *родитель* z & z *родитель* y

x *имеет-потомка* y ←
    — isvar(y) & z *родитель* y & x *родитель* z

Серьезный недостаток такого решения, так же как и при использовании других метауровневых примитивов Пролога, в том, что нарушается декларативная трактовка программы. Подобные процедуры уже не являются логическими импликациями, которые можно воспринимать как простые определения отношений.

В IC-PROLOG подобные искажения декларативной семантики не допускаются. Метауровневые условия, такие, как isvar, и все остальные аспекты, относящиеся к управлению, записываются на отдельном языке программных аннотаций. Так, две приведенные выше процедуры превращаются в аннотированные альтернативы управления:

[x *имеет-потомка* y^ ← x *родитель* z & z *родитель* y,
x *имеет-потомка* y? ← z *родитель* y & x *родитель* z]

Аннотирующий знак «^» возле переменной y в первой процедуре выражает управляющее условие, состоящее в том, что эта переменная в момент входа в процедуру не должна быть связанной. Аннотация «?» во второй процедуре означает, что переменная y в момент входа должна быть связана со значением (термом), которое не является переменной. Эти управляющие условия в точности эквивалентны условиям isvar(y) и —isvar(y). И наконец, заключение процедур в скобки говорит вычислителю о том, что это две альтернативы управления, а не логические альтернативы. Аналогичную пару альтернатив управления можно записать и для третьей процедуры отношения «x *имеет-потомка* y». Полученная программа будет эффективно работать при всех вариантах ее использования.

Дальнейшее обсуждение альтернатив управления и семантики аннотаций в заголовках процедур читатель может найти в [3]. Аннотации в заголовках аналогичны объявлениям видов передачи параметров в реализации Пролога для DEC-10. Однако в отличие от объявлений видов аннотации позволяют определять более сложные виды, и, кроме того, аннотация в заголовке процедуры транслируется в проверку периода исполнения. В языке логического программирования, предложенном в [6], для разных видов использования автоматически порождаются разные упорядочения условий (вызовов) в правых частях процедур. Упорядочение основывается на топологической сортировке, при которой приоритет отдается вызовам, имеющим входные аргументы. Это снимает ответственность с программиста, но не всегда приводит к самому эффективному порядку вычисления. Кроме того, это не охватывает случай, когда управление для разных видов использования получается не просто путем разных упорядочений последовательных вычислений, а разными комбинациями последовательных и параллельных вычислений.

4. ПАРАЛЛЕЛЬНОЕ ВЫЧИСЛЕНИЕ

В IC-PROLOG можно задавать различные формы параллельного выполнения. Мы рассмотрим их на примере задачи, хорошо иллюстрирующей выгоду, получаемую от параллель-

ного вычисления. Задача состоит в том, чтобы определить, имеют ли два бинарных дерева одинаковые кроны. В качестве примера на рис. 1 изображены два различных дерева с одинаковыми кронами.

Следующие процедуры образуют логическую программу, которая по существу представляет собой спецификацию отношения *кроны-одинаковы*, определенного на деревьях.

кроны-одинаковы(x, y) ← w *крона* x & w *крона* y
u.Nil *крона* l(u)
u.z *крона* t(l(u), y) ← z *крона* y
w *крона* t(t(x, y), z) ← w *крона* t(x, t(y, z))

Здесь t(x, y) обозначает бинарное дерево с поддеревьями x и y, а l(u) — дерево, состоящее из единственного узла с мет-

Рис. 1.

кой u. Терм u.x обозначает список с головой u и хвостом x, Nil — пустой список.

Рассмотрим теперь использование этой программы для проверки совпадения крон. Последовательное выполнение процедуры *кроны-одинаковы* означает, что сначала порождается (вся) крона x, которая затем сравнивается с кроной y. Таким образом, процедуры, определяющие отношение *крона*, используются как для порождения, так и для сравнения. Если кроны сравниваемых деревьев одинаковы, то надо обойти оба дерева и последовательное выполнение вполне приемлемо. Если же это не так, то порождать крону для x дальше той точки, в которой кроны различаются, слишком расточительно и нужно задать управление таким образом, чтобы вовремя прекратить порождение.

Прежде всего заметим, что при выполнении вызова «w *крона* x» выходное связывание для w будет порождаться в виде последовательности частичных приближений. Возьмем, например, в качестве x дерево t(l(A), t(l(B), l(C))). Тогда к вызову «w *крона* t(l(A), t(l(B), l(C)))» применима только вторая процедура для отношения *крона*, и ее вызов свяжет w с A.z. При этом z обозначает крону поддерева t(l(B), l(C)), которая будет порождена позднее. Следующий шаг вычисления свяжет z с B.z′, и в результате w будет неявно связана со следующим приближением A.B.z′. Таким образом, порождение кроны дерева происходит так, что

метка каждого концевого узла становится «доступной» через связывание переменной w в тот момент, когда этот узел фактически посещается при обходе дерева. Существуют три стратегии выполнения процедуры *кроны-одинаковы*, которые используют это постепенное, метка за меткой, порождение связывания для w.

Параллельное вычисление без синхронизации. Простейшая стратегия заключается в том, чтобы выполнять оба вызова процедуры *крона* параллельно. В IC-PROLOG для этого нужно использовать вместо «&» специальную связку «//»:

кроны-одинаковы(x, y) ← w *крона* x // w *крона* y

Действие связки «//», декларативная семантика которой соответствует конъюнкции, состоит в разветвлении выполнения процедуры на отдельные процессы. Они помещаются в очередь процессов, и вычислитель работает в режиме разделения времени, причем очередной процесс берется из головы очереди, а по окончании кванта вычисления помещается в ее хвост. На каждом кванте вычисления очередному процессу, если только он по каким-либо причинам не приостановлен (см. ниже), предоставляется время, достаточное для выполнения как минимум одного шага резолюции. Заметим, что вследствие этого любую переменную может связать лишь один процесс. Никогда не бывает так, чтобы два процесса одновременно пытались связать одну и ту же переменную. После того как некоторый процесс осуществил связывание, все остальные процессы уже не могут его изменить и должны только «читать» это связывание.

Для нашего примера это означает, что из двух процессов, выполняющих вызовы процедуры *крона*, переменную w первым свяжет тот, который работает с деревом, имеющим более короткий путь до самого левого концевого узла. После этого полученное связывание станет доступно для чтения другому процессу. Далее выполнение процессов продолжается таким образом, что метки узлов добавляются в список для w и читаются из него в порядке, определяемом формой обрабатываемых деревьев. Как только эти процессы обнаружат несовпадающие метки, вычисление прекращается с результатом «неудача».

Параллелизм с направленной связью. Можно ограничить параллельное выполнение таким образом, чтобы только одному процессу разрешалось порождать связывание для переменной w, к которой обращаются два процесса. Для этого нужно снабдить либо вхождение w в процессе-производителе аннотацией «^», либо ее вхождение в процессе-потребителе

аннотацией «?». Так, если переопределить процедуру *кроны одинаковы* следующим образом:

кроны-одинаковы(x, y) ← w *крона* x'// w^ *крона* y

то второй вызов процедуры *крона* становится процессом-про изводителем. Если в ходе параллельного вычисления процесс соответствующий первому вызову (потребитель), пытается добавить к списку w новую метку узла, он приостанавли вается. Связывание переменной w доступно ему только «по чтению». Каждый раз, когда второй процесс находит новый концевой узел, первый процесс снова становится активным и может проверить эту метку.

Сопрограммы, управляемые данными. Приведенное выш ограничение направления связи между процессами позволяе избежать ненужной работы процесса-потребителя по обход узлов кроны после первого несовпадения. Однако при этом не исключается бесполезная работа процесса-производител по дальнейшему порождению меток кроны. Чтобы избежат и этого, нам нужно задать управление таким образом, чтоб процесс-производитель приостанавливался после каждой най денной метки и возобновлялся лишь тогда, когда будет очередной раз приостановлен (в ожидании следующей метки процесс-потребитель. Для достижения этого нужно оставит аннотацию у w в процессе-производителе, но вместо «//: использовать связку «&»:

кроны-одинаковы(x, y) ← w *крона* x & w^ *крона* y

Поскольку здесь используется «&», разветвления вычис лений не происходит. В каждый момент времени активе лишь один процесс, и вызовы в правой части процедуры акти визируются поочередно. При этом аннотация «^» делает вто рой вызов «ленивым производителем» [1]) связывания для об щей переменной w.

Приведенный пример иллюстрирует три вида параллель ного управления, которые могут быть заданы в IC-PROLO с помощью аннотаций. Самая простая стратегия — параллель ное управление без ограничений. При этом моделируется па раллельное вычисление условий-конъюнктов с единственны ограничением: переменную может связывать только один про цесс. Такая форма параллелизма обсуждается в работе Хог гера [7]. Параллелизм с выделением процессов-производи

[1]) По аналогии с «Lazy evalution» (ленивое вычисление) — термино из функционального программирования. В книге П. Хендерсона «Фун циональное программироваиие» (М.: Мир, 1983) он переведен как замед ленное вычисление или вычисление с задержками (см. далее). — *Прим ред.*

телей позволяет моделировать сети параллельных процессов с односторонними каналами связи. Это соответствует модели Кана — Маккуина [8]. Кроме того, в контексте логического программирования этот вариант рассматривается в работах [11, 6]. И наконец, сопрограммный режим с выделением процесса-производителя соответствует «ленивому вычислению» в функциональном языке [5].

Параллельное вычисление с бектрекингом. Поскольку различные формы параллельного управления задаются явно, с помощью аннотаций, их можно смешивать. В комбинации с бектрекингом это дает богатый набор возможных стратегий управления.

В качестве примера мы приведем процедуры верхнего уровня для решения задачи о восьми ферзях. Это решение представляет собой небольшую модификацию чисто сопрограммного решения задачи, описанного в [3]. Там же приведены остальные процедуры программы. В качестве кандидатов на решения задачи служат перестановки списка целых чисел от 1 до 8, где i-й элемент перестановки — это номер вертикали, на которой находится ферзь, стоящий на i-й горизонтали.

Ферзи(x) ← *Безопасно*(x) & x^ *перестановка* 1.2....8.Nil
Безопасно(u.x) ← u *не-бьет* x // *Безопасно*(x)

Аннотация «^» в вызове процедуры *перестановка* делает его ленивым производителем перестановок. Отношение *перестановка* можно определить так, чтобы перестановка порождалась в виде потока частичных приближений, так же как это было с кроной дерева. Каждое новое число, добавляемое к x, соответствует очередному ферзю, для которого немедленно проверяется, не нарушает ли он условие *Безопасно* (x). Проверка условия представляет собой разветвляющееся параллельное вычисление. Для каждого ферзя, появляющегося на доске, порождается новый процесс, который будет проверять, не окажутся ли под ударом этого ферзя какие-либо из ферзей, которые будут поставлены на доску впоследствии. Неудача любой из этих проверок для очередного ферзя приводит к бектрекингу с целью найти иную расстановку. Таким образом, каждое частичное решение-кандидат порождает «шеренгу» параллельных процессов, которая растет и сокращается при выполнении с бектрекингом процедуры *перестановка.* Такое сложное алгоритмическое поведение получается с помощью простых управляющих аннотаций, добавляемых к логической программе, близкой к спецификации решаемой ею задачи.

Другие управляющие аннотации. Существуют два других способа управления параллельными вычислениями. Имеется примитив задержки процесса, записываемый в виде знака «!» после имени переменной в вызове (литерале) внутри процедуры. Если некоторый процесс активизирует эту процедуру и выполнение доходит до вызова с аннотированной таким образом переменной, то процесс приостанавливается до тех пор, пока эту переменную не свяжет какой-нибудь другой процесс. Если же все остальные процессы приостановлены, аннотация игнорируется. Приводимый ниже пример представляет собой аннотированную версию программы, рассматриваемой в [9]. Программа определяет унарное отношение *Допустимый* на списках, состоящих из пар чисел. Список считается допустимым, если в каждой паре второе число вдвое больше первого и первое число каждой следующей пары втрое больше второго числа предыдущей пары.

Допустимый(l) ← *Вдвое*(l) // *Втрое*(l)
Вдвое(Nil)
Вдвое(⟨x, y⟩.l) ← TIMES(x!, 2, y) & *Вдвое*(l)
Втрое(⟨x, y⟩.Nil)
Втрое(⟨x, y⟩.⟨z, w⟩.l) ← TIMES(y!, 3, z) & *Втрое*(⟨z, w⟩.l)

Эта программа может проверять допустимость списка пар или порождать допустимый список путем параллельного вычисления двух условий допустимости. Особенно эффективно ее использование для порождения в случае вызова вида «*Допустимый* (⟨N, x⟩.l)», где N задано. Задержки вызовов TIMES в процедурах *Вдвое* и *Втрое* означают, что остальные элементы списка будут порождаться детерминированной цепочкой умножений, начинающейся с заданного числа N.

Последняя из управляющих аннотаций — это знак «:», позволяющий делать вычисление первого вызова в теле процедуры ее «предохранителем» [4]. Введение такой аннотации в процедуру

$$B \leftarrow G: A_1 \& \ldots \& A_m$$

делает унификацию заголовка B и выполнение атома-предохранителя G неделимой единицей при параллельном или сопрограммном исполнении. Чаще всего это применяется для того, чтобы предотвратить доступ к связываниям переменных, полученных в результате унификации B, до тех пор, пока не будет успешно выполнен предохранитель G. Если вычисление G оканчивается неудачей, эти связывания остаются недоступными другим процессам. Предохранители аналогичны ограничениям, описанным в работе [1]. Различие состоит в том, что в IC-PROLOG удача при вычислении предохрани-

теля не означает, что данная процедура — единственная, которую можно применить к обрабатываемому вызову, не исключено, что для других процедур унификация и вычисление предохранителя окажутся успешными.

5. ПОТОКОВЫЙ ВВОД-ВЫВОД

Еще одна особенность IC-PROLOG — потоковый ввод-вывод. Примитив READ(x) связывает значение переменной-параметра не с одной литерой или одним термом, а со всем потоком литер, которые будут набраны на терминале. Программист обрабатывает этот поток так, как если бы это был список литер. Этот список при выполнении вызова READ(x) порождается в режиме ленивого вычисления. Например, в запросе вида

y: READ(x) & P(x, y)

D должно быть отношением между списком литер и результатом y. Литеры читаются с терминала по мере того, как вычисление P(x, y) требует очередных элементов списка x. Поскольку x — список всех вводимых литер, не возникает проблем с бектрекингом. Прочитываемые литеры явно помещаются в список, с которым связана переменная x, постепенно порождаемый процедурой READ(x). При бектрекинге соответствующие шаги вычисления P(x, y) могут получить литеры из этого частично порожденного списка. Только когда он исчерпывается, с терминала будут считываться новые литеры. Такая обработка потоков, получаемых с терминала, достигается с помощью видоизмененного алгоритма унификации, который «различает» специальные значения — указатели на буфер терминала.

Примитив WRITE(y) выводит на терминал изображение списка, связанного с y. Так же как и в списках, порождаемых процедурой READ(x), здесь используются специальные константы, обозначающие «невидимые» литеры. Так, вывод константы LINE приводит к возврату каретки (переводу курсора в начало следующей строки).

Аргумент для WRITE(y) также может порождаться в виде потока. Рассмотрим запрос вида

: READ(x) & WRITE(y) & R(x, y^),

где R — бинарное отношение над списками литер, которое порождает связывание для y по мере считывания и обработки потока литер x. При вычислении этого запроса ввод входных строк будет перемежаться выводом строк-результатов.

Каждый вводимый символ конца строки служит сигналом для вывода очередной порции результата у вплоть до ближайшей константы LINE.

6. ЗАКЛЮЧИТЕЛЬНЫЕ ЗАМЕЧАНИЯ

IC-PROLOG — язык, основанный на чистой логике, который позволяет проиллюстрировать многие понятия программирования. Используя конструкторы множеств и управляющие альтернативы, можно разрабатывать универсальные дедуктивные базы данных. С помощью потокового ввода-вывода и сопрограммного механизма, управляемого данными, можно иллюстрировать идею «ленивого вычисления». И наконец, параллельные вычисления и различные способы управления ими соответствуют современным идеям в области взаимодействия процессов. Поэтому IC-PROLOG — идеальный язык для обучения этим понятиям, и он успешно используется в этом качестве в Имперском колледже науки и техники и Сиракьюсском университете.

Исследования по языку IC-PROLOG поддерживались Британским советом по научным и инженерным исследованиям (British Science & Engineering Research Council).

ЛИТЕРАТУРА

[1] Bellia M., Degano P., Levi G. The call by name semantics of a clause language with functions. In «Logic Programming» (Eds. Clark K. L., Tärnlund S.-Å.), Academic Press, 1982, pp. 281—298.

[2] Clark K. L. Negation as failure. In «Logic and Data Bases» (Eds. Gallaire H. and Minker J.). Plenum Press, New York, 1978, pp. 293—322.

[3] Clark K. L., McCabe F. The Control facilities of IC-PROLOG. In «Expert Systems in the Micro-Electronic Age» (Ed. Michie D.), Edinburgh University Press, 1979.

[4] Dijkstra E. W. A discipline of programming. Prentice-Hall, Englewood Cliffs, New Jersey, 1976. [Имеется перевод: Дейкстра Э. Дисциплина программирования. — М.: Мир, 1978.]

[5] Friedman D., Wise D. CONS should not evaluate its arguments. In «Automata, Languages and Programming» (Ed. Michaelson S.), Edinburgh University Press, pp. 256—284.

[6] Hansson A., Haridi S., Tärnlund S.-Å. Properties of a logic programming language. In «Logic Programming» (Eds. Clark K. L., Tärnlund S.-Å.), Academic Press, 1982, pp. 267—280. [Имеется перевод: Ханссон А., Харнди С., Тернлунд С.-А. Свойства одного языка логического программирования, см. наст. сборник.]

[7] Hogger C. J. Concurrent logic programming. In «Logic Programming» (Eds. Clark K. L., Tärnlund S.-Å.), Academic Press, 1982, pp. 199—212.

[8] Kahn G., McQueen D. B. Coroutines and networks of parallel processes. Proc. IFIP-77.

[9] Kowalski R. A. Logic for problem solving. Artificial Intelligence Series (Ed. Nilsson N. J.), North Holland, 1979.
[10] Pereira L. M. et al. User's Guide to DEC system-10 Prolog. Divisao de Informatica, Laboratorio Nacional de Engenharia Civil. Lisbon, 1978.
[11] van Emden M. H., de Lucena G. J. Predicate logic as a language for parallel programming. In «Logic Programming» (Eds. Clark K. L., Tärnlund S.-Å.), Academic Press, 1982, p. 189—198.

ЛОГЛИСП: МОТИВИРОВКА, ОСНОВНЫЕ ВОЗМОЖНОСТИ И РЕАЛИЗАЦИЯ[1)]

ДЖ. РОБИНСОН, Э. ЗИБЕРТ

1. МОТИВИРОВКА

Логлисп представляет собой реализацию идеи логического программирования в рамках Лиспа. Одним из мотивов такого расширения Лиспа для нас было желание предоставить себе и другим пользователям Лиспа некоторый эквивалент Пролога, который бы не требовал выхода за пределы богатой, весьма удобной и привычной (для нас) среды программирования Лиспа. Кроме того, мы надеялись, что расширенный таким образом Лисп может оказаться эффективным средством убедить его консервативных приверженцев, что в конце концов благодаря Ковальскому, Колмероэ, ван Эмдену, Кларку и другим пионерам логического программирования на останках старого резолюционного доказательства теорем выросло нечто красивое и полезное.

Мы также хотели в какой-то степени спасти быстро исчезающее различие между логическим программированием в том смысле, как оно было описано Ковальским, и его конкретной реализацией в рамках Пролога. Мы охотно признаем, что быстрый рост интереса к логическому программированию был бы невозможен без наличия ряда реализаций Пролога, и разделяем всеобщее восхищение превосходной эдинбургской Пролог-системой, созданной Уорреном для машин DEC-10. И все же процедурная интерпретация хорновских дизъюнктов в алгоритме доказательства теорем, основанного на LUSH-резолюции [2)], хотя и допускает, но никоим образом не требует привлечения бектрекинга для проверки альтернативных вариантов доказательства. Эти альтернативные варианты могут (как в Логлиспе) развиваться квазипараллельно, так, чтобы не создавалось впечатление, будто процесс продвижения вперед с неудачами, возвратами и новыми

[1)] J. A. Robinson, E. E. Sibert. LOGLISP: Motivation, Design and Implementation — Logic Programming (Eds. Clark K. L., Tärnlund S.-Å.), Academic Press, 1982, p. 293—313.

[2)] Термин LUSH-резолюция — сокращение, обозначающее линейную резолюцию с неограниченным выбором для хорновских дизъюнктов (Linear resolution with Unrestricted Selection for Horn clauses). — *Прим. перев.*

попытками достичь цели (в духе Плэнера) составляет суть логического программирования. В случае Логлиспа логическое программирование выглядит как запись наборов достаточных условий, выражающих истинность некоторых предикатов на определенных кортежах элементов данных. Если есть такой набор (база знаний) D, то для заданной конъюнкции Q атомарных предложений можно вычислить одно, несколько или даже все решения, т. е. такие совокупности E связываний переменных с термами, для которых Q может быть доказана (выведена из D методом LUSH-резолюции).

Логлисп включает в себя Лисп и набор примитивов для логического программирования, получивший название Логика. Мы начнем с того, что кратко напомним основные соглашения и обозначения Лиспа.

2. ЛИСП

В Лиспе всего два типа данных — атомы и точечные пары. *Атомы* — это идентификаторы, строки и числа. Для любых двух объектов данных A и B можно построить *точечную пару* A.B, причем A называется *головой* точечной пары, а B — *хвостом*. Если X = A.B, то мы будем писать

$$A = hX, \quad B = tX.$$

Атом NIL, другое изображение которого — ( ), называется *пустым списком*. В общем случае список — это или ( ), или точечная пара, хвост которой является списком.

Точечные пары вида $A_1.(A_2. \ldots (A_n.A_{n+1}) \ldots)$ можно записывать в виде $(A_1 \ldots A_n.A_{n+1})$ (точечная списочная запись). Если $A_{n+1} = NIL$, можно написать еще короче: $(A_1 \ldots A_n)$ (списочная запись).

3. ЛОГИКА

В рамках Логики утверждения, запросы и все остальные конструкции логического программирования представляются в виде объектов данных Лиспа. Логические переменные обозначаются идентификаторами, начинающимися со строчной буквы. Имена собственные (т. е. имена предикатов, операций и индивидных констант) изображаются идентификаторами, начинающимися с прописной буквы, или специальными литерами, такими, как +, §, $, %, !, =, >, <, *, : или ?. Кроме того, изображениями индивидных констант (но не предикатов или операций) могут служить числа и строки (строка или цепочка — это последовательность литер, заключенная в двойные кавычки). Собственные имена, кроме чисел

и строк, будем называть *собственными идентификаторами.* По сравнению с эдинбургским Прологом (где переменные начинаются с прописной буквы, а константы — со строчной) у нас все наоборот, что соответствует соглашениям, действующим в исчислении предикатов (там принято писать R(x, y), а не r(X, Y)). С другой стороны, нам приходится записывать имена операций не со строчной буквы, как это обычно делается, а с прописной. Кроме того, мы следуем традиции Лиспа, заключая изображение предиката (или операции) в скобки вместе с аргументом (аргументами). Например, мы пишем R x (F x y), а не R(x, F(x, y)), как правило, опуская запятые, хотя, если хочется, их можно оставлять.

Терм — это логическая переменная, константа или список термов. А-предложением [1]) (или *атомарным предложением*) называется список термов, головой которого является собственный идентификатор. Предложения вида

если каждый член A истинен, то истинно B

называются *утверждениями.* В общем случае *заключение* B — это некоторое А-предложение, а *гипотеза* A — это список А-предложений. Утверждение с пустой гипотезой называется *безусловным*, а с непустой — *условным.* Утверждение, содержащее логические переменные, называется *правилом*, а утверждение без переменных — *данным.*

3.1. Утверждения

Мы следуем нотации Ковальского и записываем условные утверждения вида

если каждый член (A_1 ... A_n) истинен, то истинно B

используя обратную стрелку:

$B \leftarrow A_1 \ldots A_n$,

однако для ввода в машину пользователь Логлиспа записывает такое утверждение в виде процедурного вызова Лиспа

(: — B A_1 ... A_n),

который запускает функцию приема утверждения. Эта функция осуществляет запоминание утверждения в системе под первичным индексом P, где P — предикатный символ заключения B. Если P — предикат данных (т. е. предикат, для которого все утверждения являются данными), то для вводи-

[1]) В оригинале — predication. В литературе по математической логике используется термин «атом», но в Лиспе он уже занят. — *Прим. перев.*

мого утверждения производится также вторичное индексирование по каждому из собственных идентификаторов, упоминаемых в заключении. Множество утверждений мы называем *базой знаний*. Базу знаний принято разделять на *процедуры*, каждая из которых соответствует некоторому предикату P и состоит из всех тех утверждений в базе знаний, заключения которых имеют P в качестве заголовка. При этом P считается также и именем процедуры.

По усмотрению пользователя утверждениям могут присваиваться индивидуальные имена. При демонстрации объяснений (логических выводов) удобно иметь возможность ссылаться на утверждение по имени, вместо того чтобы выписывать его целиком.

3.2. Запросы

По существу запрос — это описание множества

$$((x_1 \ldots x_t) : C_1 \,\&\, \ldots \,\&\, C_n),$$

состоящего из всех кортежей длины t, удовлетворяющих некоторому ограничению, записанному в виде конъюнкции A-предложений C_i, $i = 1, \ldots, n$. В логике такой запрос записывается так:

$$(\text{ALL}\ ((x_1 \ldots x_t)\, C_1 \ldots C_n)$$

и фактически представляет собой вызов в Лиспе процедуры ALL (которая относится к виду FEXPR, т. е. не вычисляет своих аргументов). Запрос возвращает в качестве своего значения список всех кортежей, удовлетворяющих заданному ограничению, называемый *ответом* на данный запрос. Кортежи, образующие этот список, получаются с помощью описываемого ниже базового цикла вывода, который составляет центральную часть Логики и инициируется при каждом запросе. К другим формам запроса относится запрос вида

$$(\text{ANY K}\,(x_1 \ldots x_t)\, C_1 \ldots C_n),$$

возвращающий не более чем K кортежей, удовлетворяющих ограничению, запрос вида

$$(\text{THE}\,(x_1 \ldots x_t)\, C_1 \ldots C_n),$$

значением которого является единственный элемент списка (но не сам список):

$$(\text{ANY 1}\,(x_1 \ldots x_t)\, C_1 \ldots C_n),$$

и запрос вида

(SETOF K X C).

Здесь SETOF — это функция вида EXPR (такие функции вычисляют свои аргументы), лежащая в основе FEXPR-функций ALL, ANY и THE. У SETOF три аргумента: K, который должен иметь своим значением неотрицательное целое или атом ALL, X, поставляющий шаблон ответа (x_1 ... x_t), и C, значением которого должен быть список ограничения (C_1 ... ... C_n). Следует особо отметить, что ответ на запрос — это объект данных Лиспа, и его можно подвергнуть дальнейшей программной обработке или вывести на терминал.

4. ЦИКЛ ВЫВОДА

Основной вычислительный процесс в системе Логика выполняется в рамках *цикла вывода* для того, чтобы получить ответ на некоторый запрос.

4.1. Неявные выражения

Для большей ясности, а также из соображений эффективности, представление ограничений в ходе исполнения цикла вывода использует метод Бойера — Мура. Согласно этому методу, выражение C представляется парой (Q E), называемой *неявным выражением*, где Q — некоторое выражение, называемое *скелетом*, а E — множество связываний переменных, именуемое *контекстом* (environment) неявного выражения. Основная идея заключается в том, что неявное выражение (Q E) служит представлением выражения, получаемого из Q путем связывания в нем переменных согласно E.

4.2. Связывания и контексты

В общем случае *связывание* — это точечная пара, головой которой является некоторая логическая переменная, а *контекст* — это такой список связываний, в котором ни у каких двух связываний го́ловы не совпадают. Ниже на довольно абстрактном уровне приводится описание цикла вывода, в котором для большей ясности опущены конкретные детали нашей реализации. Доступ к связываниям переменных в контекстах в действительности осуществляется не путем просмотра списков-контекстов, как можно предположить исходя из нашего описания, а более прямыми методами, характерными для работы с массивами, что позволяет значительно ускорить доступ. Как скоро увидит читатель, наш подход, не

использующий бектрекинга, приводит к деликатной проблеме: как совместить экономию памяти, получаемую от совместного использования структуры по методу Бойера — Мура, с потребностью в практически прямом (т. е. ограниченном по времени константой) доступе к связываниям переменных в процессе унификации. Дальнейшее обсуждение этой проблемы и нашего решения для нее мы продолжим после знакомства с циклом вывода.

4.3. Прямая и конечная ассоциации

Если контекст E содержит точечную пару A.B, мы будем говорить, что A *определено в* E, и записывать это так: (DEF A E). Мы будем также называть B *прямой ассоциацией* A *в* E и писать B = (IMM A E). Как отмечалось выше, вычисление B при заданных A и E в нашей реализации осуществляется практически напрямую. Заметим, что если выражение A определено в контексте E, значит, A — логическая переменная. Если же (DEF A E) = false, то A может как быть, так и не быть логической переменной.

Прямая ассоциация A в E может, в свою очередь, оказаться переменной, которая также определена в E. В этом случае часто требуется пройти по цепочке связываний до конца, чтобы получить *конечную ассоциацию* A в E, которая представляет собой первое из выражений в цепочке

A, (IMM A E), (IMM(IMM A E) E), . . . ,

не определенное в E, и обозначается (ULT A E). Таким образом,

(ULT A E) = если (DEF A E) то (ULT(IMM A E) E) иначе A.

Заметим, что (ULT A E) определена для любых выражений A, а не только для логических переменных.

4.4. Рекурсивные реализации

Теперь мы можем сказать более точно, каким образом выражение неявно представляется парой, состоящей из скелета и контекста. *Рекурсивной реализацией* выражения X в контексте Y называется выражение, обозначаемое (RECREAL X Y) и определяемое так: (RECREAL X Y) =

если X = U.V то (RECREAL U Y).(RECREAL V Y)
иначе если (DEF X Y) то (RECREAL(ULT X Y) Y)
иначе X

Таким образом, пара (Q E), состоящая из скелета и контекста, представляет выражение (RECREAL Q E).

4.5. Унификация

Цикл вывода основывается на многократном применении операции LUSH-резолюции [1]), которая, в свою очередь, включает в себя процесс унификации.

Если даны два выражения A и B, а также некоторый контекст E, мы будем говорить, что A *унифицируется* с B в E тогда и только тогда, когда существует такое расширение E′ контекста E (т. е. контекст, содержащий все связывания из E и, возможно, некоторые другие), что

(RECREAL A E′) = (RECREAL B E′).

Теперь определим функцию UNIFY таким образом, что если A унифицируется с B в E, то контекст (UNIFY A B E) является *самым общим* расширением E′ контекста E, удовлетворяющим приведенному выше равенству. Если же A не унифицируется с B в E, то значением (UNIFY A B E) должно быть сообщение IMPOSSIBLE. Такое определение функции UNIFY выглядит следующим образом:

(UNIFY A B E) = если E = IMPOSSIBLE то IMPOSSIBLE
иначе (EQUATE (ULT A E) (ULT B E) E)

где

(EQUATE A B E) =

если A = B	то E	иначе
если A — переменная	то (A.B).E	иначе
если B — переменная	то (B.A).E	иначе
если A — атом	то IMPOSSIBLE	иначе
если B — атом	то IMPOSSIBLE	иначе

(UNIFY tA tB (UNIFY hA hB E)).

4.6. LUSH-резолюция

Предположим теперь, что у нас есть база знаний D и некоторое ограничение, представленное парой (Q E), состоящей из скелета и контекста.

Пусть (VARIANT Q E D) — некоторый вариант базы D, не имеющий общих переменных ни с Q, ни с E (вариантом D называется объект, идентичный D, за исключением того, что для некоторых или всех логических переменных в нем могут использоваться другие идентификаторы).

Пусть (SELECT Q E D) — некоторое положительное целое число, не превышающее длины списка Q. Предполагается, что

[1]) См.: Hill L. LUSH Resolution and its Completeness. DCL Memo 78. University of Edinburgh. — *Прим. перев.*

для вычисления своего результата SELECT при необходимости может использовать структуру и содержимое Q, E и D.

И наконец, если имеется непустой список X и положительное целое K, не превышающее длины X, можно говорить о *расщеплении* X по его K-му элементу и определить значение выражения (SPLIT X K) как тройку (L A R), такую, что A — это K-й элемент X, и X = L*(A)*R. (L*M обозначает конкатенацию списков L и M, причем *, естественно, обладает свойством ассоциативности.)

Таким образом, если (L A R) = (SPLIT X K), то L — это список из первых K — 1 элементов списка X, а R — список из последних ((LENGTH X) — K) элементов X. В частности, если K = 1, мы имеем

L = ( ), A = hX, R = tX.

Теперь мы можем дать определение *LUSH-резольвент неявного ограничения* (Q E) *относительно базы знаний* D. Ими будут все неявные ограничения вида

(L*H*R(UNIFY A C E))

такие, что:

(1) H — это гипотеза, а C — заключение некоторого утверждения, входящего в (VARIANT Q E D);
(2) (L A R) = (SPLIT Q (SELECT Q E D));
(3) A унифицируется с C в E.

В настоящее время наша реализация использует значение (SELECT Q E D) = 1 для любых Q, E и D. Исследуются также более общие критерии выбора, допускаемые в теории LUSH-резолюции, и в последующих версиях Логлиспа, возможно, будет взят какой-либо более сложный вариант SELECT.

Отделение переменных, которое получается, когда мы вместо D берем (VARIANT Q E D), достаточно эффективно реализуется в Логлиспе, как, впрочем, и во всех известных нам реализациях Пролога. С точки зрения теории брать варианты необходимо, чтобы гарантировать полноту резолюции. Бойер и Мур первыми показали, как можно с малыми затратами представлять варианты с помощью индексирования, о чем более подробно будет сказано ниже.

В Логлиспе множество LUSH-резольвент (Q E) относительно D возвращается в виде списка (REC Q E D), который использует значительную часть своей структуры в духе Бойера — Мура совместно с (Q E).

При вычислении (RES Q E) нередко можно обойтись без просмотра всей процедуры, предикатный символ которой совпадает с предикатным символом выбранного A-предложения.

Ясно, что всегда достаточно просмотреть только эту процедуру (а не всю базу знаний!), но на практике часто бывают ситуации, когда можно ограничиться просмотром лишь некоторого сравнительно небольшого подмножества утверждений процедуры. Например, может оказаться, что ни одно из утверждений процедуры не содержит в своем заключении логических переменных (как в случае процедуры, состоящей из одних данных). В такой ситуации достаточно рассмотреть лишь те утверждения процедуры, заключения которых содержат все собственные идентификаторы, встречающиеся в выбранном А-предложении. К этим утверждениям возможен быстрый доступ с помощью упомянутой ранее системы вторичного индексирования.

4.7. Цикл вывода

Теперь мы можем определить цикл вывода. Если имеется база знаний D, то ответом на запрос (SETOF K X P) будет результат выполнения следующего алгоритма:

```
IN: установить SOLVED равным пустому множеству
    установить WAITING равным множеству, содержа-
    щему единственный элемент (P( ))
RUN: пока WAITING не пусто
        и SOLVED содержит менее K элементов
    цикл 1 удалить из WAITING какое-либо неявное
           ограничение C
           установить (Q E) равным(SIMPLER C D)
         2 если Q=( )
           то добавить E в SOLVED иначе добавить все
           элементы (RES Q E D) в WAITING
OUT: возвратить (SIMPSET X SOLVED)
```

Замечания к алгоритму дедуктивного цикла

(1) Функции SIMPLER и SIMPSET обсуждаются ниже, в пункте, посвященном Лисп-упрощению.

(2) Если K = ALL, то условие цикла сводится к своему первому конъюнктивному члену.

(3) Запросы

(ALL X P_1 ... P_n),
(ANY K X P_1 ... P_n),
(THE X P_1 ... P_n)

эквивалентны соответственно запросам:

(SETOF (QUOTE ALL) (QUOTE X) (QUOTE (P_1 ... P_n))),
(SETOF K (QUOTE X) (QUOTE (P_1 ... P_n))),
h(SETOF 1 (QUOTE X) (QUOTE (P_1 ... P_n))).

(4) *Шаблон ответа* X в запросе может быть логической переменной, собственным именем или списком выражений (и в частности, списком переменных).

(5) (RESQED) может быть пустым. В этом случае ограничение C просто удаляется из WAITING, а к SOLVED ничего не добавляется. Это означает, что C является *явным тупиком* (см. ниже).

(6) Выбор C из WAITING осуществляется среди ограничений с минимальной оценкой *стоимости решения*. Эта оценка представляет собой линейную функцию от длины скелета C и длины логического вывода, в результате которого получено C.

4.8. Дерево вывода. Явные и неизбежные тупики

Исследование алгоритма цикла вывода показывает, что он порождает некоторое дерево (называемое *деревом вывода*), узлами которого являются неявные ограничения. Корнем дерева вывода служит ограничение, содержащееся в первоначальном запросе, неявное представление которого имеет в качестве контекста пустой список. Непосредственными потомками каждого узла в дереве вывода являются LUSH-резольвенты соответствующего неявного ограничения относительно заданной базы знаний, если таковые есть.

В процессе порождения дерева его крона в любой момент состоит из узлов, содержащихся во множестве WAITING, и узлов, контексты которых к этому моменту добавлены в множество SOLVED.

Концевые узлы полностью построенного дерева вывода разделяются на два класса. Те, у которых скелеты — пустые списки, называются *успехами*, и их контексты к этому моменту добавлены в множество SOLVED (построение которого уже также завершено). Второй класс составляют концевые узлы, скелеты которых не являются пустыми списками. Их мы будем называть *явными тупиками*[1]); они не вносят никакого вклада в построение множества SOLVED. Другими словами, явный тупик — это такой узел, для которого выбранное A-предложение не унифицируется ни с каким из заключений утверждений, содержащихся в базе знаний.

В дереве вывода можно также выделить узлы, которые, хотя сами и не являются явными тупиками, не имеют среди своих потомков ни одного успеха. Такие узлы называются неизбежными тупиками[2]). Каждый из них является корнем

[1]) В оригинале — immediate failure. — *Прим. перев.*

[2]) В оригинале — ultimate failure. — *Прим. перев.*

поддерева, у которого все концевые узлы — явные тупики. Было бы замечательно, если бы мы могли распознавать неизбежный тупик до того, как соответствующее поддерево будет построено целиком, и как раз это в некоторых случаях позволяет сделать Лисп-упрощение, к которому мы сейчас и перейдем.

4.9. Лисп-упрощение. Функции SIMPLER, SIMPSET

В алгоритме цикла вывода функции SIMPLER и SIMPSET вызываются для того, чтобы инициировать процесс Лисп-упрощения, на котором основывается одна из наиболее характерных особенностей Логлиспа. Именно здесь проходит интерфейс между Логикой и Лиспом.

Суть Лисп-упрощения заключается в сведéнии (редукции) некоторого выражения (если это возможно) к более простому в соответствии с лисповской семантикой, которой может обладать это выражение и его подвыражения. Например, выражение (+ 3 4) можно свести к выражению 7, а (LESSP (ADD 1 5) (TIMES 2 8)) за три шага сводится к T. Редуцированное выражение получается в результате замены подвыражений выражениями, которые определены как их эквиваленты. Такие замены повторяются до тех пор, пока дальнейшая редукция не станет невозможной. Редукция часто, но не всегда — это то же самое, что вычисление значения выражения. Например, выражение (+ a (+ 2 2)) редуцируется к (+ a 4). Кроме того, не все выражения можно редуцировать, некоторые оказываются нередуцируемыми. Таким образом, в общем случае мы говорим о *Лисп-упрощении* некоторого выражения и определяем его следующим образом. Если выражение нередуцируемо, то его Лисп-упрощением будет оно само, а в противном случае это будет (нередуцируемое) выражение, получаемое из данного путем его максимального редуцирования (иногда — вплоть до «значения», такого, как T или число).

Выражение (SIMPSET X SOLVED) — это просто множество (представленное списком) всех выражений, которые являются Лисп-упрощениями выражений вида (RECREAL X E), где E — некоторый элемент множества SOLVED.

Функция SIMPLER преобразует неявное ограничение C, выбранное на шаге 1 подцикла с меткой RUN в общем цикле вывода. Напомним, что если C — это (Q E), то на самом деле мы имеем дело со списком P = (RECREAL Q E), который C неявно представляет.

Содержательно задача функции SIMPLER заключается в замене A-предложения, которое в списке P имеет порядковый

номер (SELECT Q E D), его Лисп-упрощением. Иногда это А-предложение сводится к T, и в этом случае SIMPLER исключает его из P (поскольку P представляет логическую конъюнкцию своих элементов, результирующий список P′ эквивалентен P). Если Q′ — список, который получается из Q после удаления его элемента с номером, равным (SELECT Q E D), то получается, что P′ — это (RECREAL Q′E). Далее SIMPLER повторяет этот процесс, редуцируя А-предложение с номером (SELECT Q′ E D) в P′, и так далее до тех пор, пока список А-предложений не окажется пустым или не попадется А-предложение, не сводимое к T. Таким образом, результат функции SIMPLER — это неявное ограничение (с контекстом E), которое представляет этот заключительный список А-предложений.

Может получиться так, что выбранное А-предложение сведется к атому NIL (представляющему ложь). Поскольку это означает, что оно не унифицируется ни с каким из заключений утверждений в базе знаний, мы, таким образом, превращаем один из возможных неизбежных тупиков в явный тупик.

Практическое следствие введения преобразования, выполняемого функцией SIMPLER, состоит в том, что пользователь может обращаться почти ко всем средствам Лиспа изнутри гипотез утверждений, задаваемых в рамках Логики.

Такая возможность позволяет обрабатывать запросы, которые в конечном счете просто являются в Лиспе вызовами функций ALL, ANY, THE и SETOF при выводах, выполняемых на более высоких уровнях (на которых они выступают как термы, подвергаемые Лисп-упрощению). Таким образом, в запросах могут содержаться подзапросы и так далее вплоть до любой требуемой глубины.

Простой иллюстрацией этому служит приводимое ниже утверждение, которое придает примитиву NOT (Лиспа) новое качество, соответствующее трактовке отрицания как неудачи доказательства. Мы считаем это не частью семантики NOT, а дополнительным свойством, которое пользователь при желании может задать.

Такое свойство определяется с помощью утверждения

(NOT p)←(NULL(ANY 1 T p))

которое означает, что для того чтобы доказать атомарное предложение (NOT p), достаточно инициировать запрос (ANY 1 T p) и убедиться, что в качестве ответа получится пустой список. Заметим, что у запроса (ANY 1 T p) лишь два возможных ответа: ( ) и T. Ответ T говорит о том, что существует по крайней мере один способ доказать p, и значит, предложение (NOT p) следует считать ложным. А если полу-

чен ответ ( ), значит, несмотря на исчерпывающий поиск, ни одного способа доказать p не найдено.

Таким образом, если какой-то пользователь хочет предполагать относительно своей базы знаний, что неспособность доказать А-предложение равносильна способности доказать его отрицание, он может достичь этого, добавив к базе знаний приведенное утверждение.

4.10. Бесконечный поиск. Окно вывода

Поскольку в общем случае ответ на некоторый запрос может быть бесконечным множеством, следует предусмотреть какой-то способ разумно прекратить бесконечный процесс поиска решений после того, как получено некоторое их конечное число.

В Логике определен набор параметров (значения которых задаются пользователем или устанавливаются по умолчанию), ограничивающих, каждый по-своему, размеры дерева вывода. Например, можно ограничить общее число узлов, максимальную длину ветви или максимальный размер списков ограничений для узлов дерева. Можно также задать максимальное число применений правил на одной ветви и аналогичное ограничение для данных.

Набор этих ограничений называется *окном вывода.*

Стоит отметить, что окно вывода можно устанавливать для каждого запуска цикла вывода, вводя в запрос соответствующие аннотации. Например, при использовании запроса

(ALL X P_1 ... P_n)

окно вывода будет определено по умолчанию, а если добавить к нему аннотации

(ALL X P_1 ... P_n RULES : 5 TREESIZE : 1000)

то указанные ограничения будут соответствующим образом изменены.

Кроме управления формой и размерами дерева вывода пользователь может также задать коэффициенты, которые будут использоваться для вычисления оценки *стоимости решения* каждого из ограничений, добавляемых к множеству WAITING в процессе вывода. Поскольку на шаге 1 подцикла с меткой RUN всегда выбирается ограничение с наименьшей оценкой стоимости решения, пользователь получает возможность в какой-то степени управлять характером порождения дерева вывода.

Можно также задать способ построения дерева, как в Прологе, т. е. использовать поиск в глубину и перебор утверждений точно в том порядке, в каком их первоначально вводил пользователь.

5. ЭФФЕКТИВНОСТЬ РЕАЛИЗАЦИИ

Скорость обработки запросов с помощью Логики не так впечатляет, как, например, скорость работы эдинбургского Пролога при использовании скомпилированных утверждений (последняя, согласно измерениям, составляет около 20 000 узлов в секунду). Когда эдинбургский Пролог работает в чисто интерпретационном режиме, он обрабатывает около 1000 узлов в секунду, и это можно взять за основу для сравнения с Логлиспом, также использующим интерпретацию.

Мы получили для нашей текущей версии Логлиспа цифру порядка 150 узлов в секунду — одну шестую от скорости эдинбургского интерпретатора. Мы хотим достичь несколько лучшего соотношения и сейчас проводим разного рода усовершенствования, направленные на ускорение работы системы. В оставшейся части этого раздела мы рассмотрим ряд аспектов нашей реализации, которые могут представлять интерес.

Мы хотим несколько более детально пояснить, каким именно образом представляются контексты, так, чтобы можно было легко получать варианты утверждений и иметь быстрый доступ к связываниям. Используемый нами метод, как уже было сказано, представляет собой специализированную версию метода совместного использования структуры Бойера — Мура. Неявным представлением выражения в действительности служит тройка (I Q E), где I — неотрицательный целый *индекс*, который приписан каждой переменной, входящей в Q. Контекст E для каждого из индексов, при которых были введены связывания, содержит список этих связываний, причем каждое связывание задается в виде (A J.B), и это значит, что переменная A связана с термом B, причем переменным, входящим в B, приписан индекс J.

Первоначальный запрос получает индекс 0. При вычислении резольвенты некоторого ограничения с каким-либо правилом это правило получает индекс на единицу больше максимального индекса, использованного при выводе ограничения, и тем самым мгновенно формируется вариант правила, переменные которого отличаются от всех переменных, входящих в ограничение. Если же происходит резолюция с данным, нового индекса не заводится.

Контекст, используемый в текущий момент вычисления, представлен массивом ENV, i-й элемент которого является списком связываний для переменных с индексом i. Поскольку все переменные, связанные в одном списке, происходят из одного утверждения, списки не бывают очень длинными. На практике количество связываний в одном списке очень редко превышает шесть, а в среднем оказывается даже меньше.

Другие контексты (кроме текущего, который хранится в массиве ENV) представлены списками, элементы которых являются списками связываний, ранее находившимися в ENV. Когда из множества WAITING выбирается очередное ограничение, его контекстный список нужно загрузить в массив ENV. И наоборот, новые контексты необходимо выгрузить из массива путем построения списка его элементов. Время, требуемое для каждой из этих операций, в худшем случае пропорционально максимальному индексу, для которого есть связывания, и не зависит от числа связанных переменных. Таким образом, доля, которую загрузка и выгрузка могут внести в общее время выполнения, по существу находится в квадратичной зависимости от глубины вывода.

Чтобы снизить потенциальные издержки от этого эффекта, в имеющейся реализации сохраняется информация о том, в какой степени предыдущий список (точнее, некоторый его хвост) отражает текущее содержимое массива ENV. Когда происходит очередная выгрузка из массива, новый список берет все что можно от старого списка. Эта же информация используется для того, чтобы ускорить загрузку массива, когда можно установить, что некоторая его часть уже содержит требуемые связывания. Таким способом мы в значительной степени уменьшаем долю указанной квадратичной составляющей времени вычисления, а в некоторых случаях вовсе избавляемся от нее.

6. ЗАКЛЮЧИТЕЛЬНЫЕ ЗАМЕЧАНИЯ

Наш опыт работы с Логлиспом показывает, что он и в самом деле обеспечивает развитую среду для логического программирования, потребность в которой возникла у нас после первых попыток использовать Пролог. В частности, мы считаем особенно полезной возможность обращения к Логике из Лиспа и наоборот. Весьма удобно получать ответ на запрос в виде объекта данных Лиспа, с тем чтобы при необходимости подвергнуть его анализу и любой требуемой алгоритмической обработке. Пользователь системы Логика не зависит от разработчика системы по части встроенных функций и предикатов — он может сам написать их на Лиспе и затем обращаться к ним из Логики.

В наших общих принципах, видимо, нет ничего такого, что бы мешало реализовать значительно более быстрый алгоритм цикла вывода. Мы думаем, что, позаимствовав (кроме других приемов) методы компиляции Уоррена, сможем ускорить вывод как минимум в 10 раз, чем мы в настоящее время и занимаемся.

ОСНОВЫ ЛОГИЧЕСКОГО ПРОГРАММИРОВАНИЯ, БАЗИРУЮЩИЕСЯ НА ИНДУКТИВНЫХ ОПРЕДЕЛЕНИЯХ [1])

М. ХАГИЯ, Т. САКУРАИ

Резюме. Представлена система логического вывода, предназначенная для того, чтобы служить основой логического программирования. Отличительной чертой принятого здесь подхода является применение теории индуктивных определений, позволяющей единообразно трактовать различные виды схемы индукции, а также рассматривать *отрицание как неудачу* в качестве одного из видов схемы индукции. Этот подход соответствует так называемой семантике наименьшей неподвижной точки. Более того, в нашем формализме логические программы расширены таким образом, что посылкой предложения может быть любая формула первого порядка. Это позволяет писать спецификации, содержащие кванторы, как логические программы, а также значительно расширить класс схем индукции, чтобы включить обычную индукцию по значениям.

1. ВВЕДЕНИЕ

Цель этой статьи [2]) — ввести логическую систему, предназначаемую в качестве базиса для верификации и автоматического построения логических программ. В формулировке правил мы следуем теории итерированных индуктивных определений Мартина-Лёфа [12].

Логическое программирование проводит идею, что семантика языка программирования состоит из описания декларативного (логического) смысла программы и управления ее исполнением. Последнее соответствует операционной семантике, а первое — части денотационной семантики. Преимущество логического программирования состоит в том, что так как программа отображается в формулу логической системы, то ее декларативный смысл, естественно, задается этой формулой и многие ее свойства, такие, как частичная корректность, завершаемость или эквивалентность программ, выразимы в рамках логической системы. Заметим, однако, что полная семантика языка программирования не может быть определена только в логической системе. Требуется описать ее каким-то другим способом и доказать, что выполнение программы в соответствии с полной семантикой не противо-

[1]) M. Hagiya, T. Sakurai, Foundations of Logic Programming Based on Inductive Definitions. — New Generation Computing, 2, 1984, p. 59—77.

[2]) Она объединяет в себе наши предыдущие работы [9] и [16].

речит декларативному смыслу программы, заданному в логической системе. В случае чистого Пролога в качестве логической системы выступает исчисление предикатов первого порядка, а в качестве программы — множество хорновских дизъюнктов, и декларативным смыслом программы является конъюнкция хорновских дизъюнктов, в то время как способ выполнения программ, т. е. операционная семантика чистого Пролога, — это SLD-резолюция с некоторой стратегией, например с поиском в глубину или ширину.

Многие свойства программ можно выразить в рамках обычной логики первого порядка. Кларк и Тернлунд [8] предложили использовать логику предикатов первого порядка (см. также Кларк и Дарлингтон [7], Ханссон и Тернлунд [10]). В их системе структура данных определяется некоторым предикатом, а предикат определяется рекурсивными уравнениями первого порядка. Однако они отмечают, что для полной характеризации структуры данных необходимо ввести схему индукции по этой структуре данных, т. е. добавить предельное предложение к предикатным определениям. Рассмотрим, к примеру, множество натуральных чисел. Для того чтобы охарактеризовать его как структуру данных, вводятся следующие предложения:

1) 0 — натуральное число;

2) x — натуральное число тогда и только тогда, когда то, что следует за x, — натуральное число.

Эти предложения, сформулированные в виде рекурсивных уравнений, не характеризуют множество натуральных чисел полностью. В соответствии с аксиомами Пеано в добавление к этим двум предложениям необходимо ввести схему индукции по натуральным числам. В общем случае, однако, нет никакой гарантии, что предикатное определение и схема индукции совместимы друг с другом. Наиболее значительное отличие нашей системы от системы Кларка и Тернлунда [8] состоит в том, что мы встроили в нашу систему правило, позволяющее порождать такую схему индукции по предикатным определениям. Это правило называется *устранением продукции.*

Когда задано множество рекурсивных уравнений, определяющих предикат, имеется два альтернативных способа определить семантику предиката. Первый — это отождествить предикат с наименьшей неподвижной точкой уравнений, а второй — с наибольшей неподвижной точкой. (См. Апт и ван Эмден [2] и Сато [17] по поводу исследования семантики наибольшей неподвижной точки.) Рекурсивные уравнения сами по себе не определяют, какую семантику следует выбрать, но использование устранения продукции приводит к выбору

семантики наименьшей неподвижной точки: устранение продукции — это синтаксическое представление семантики наименьшей неподвижной точки. Можно также ввести аналогичное правило, представляющее семантику наибольшей неподвижной точки. Однако важное отличие состоит в том, что устранение продукции приводит к широкому спектру схем индукции. В этом и состоит одна из причин, по которой мы выбрали семантику наименьшей неподвижной точки.

Декларативный смысл логической программы можно рассматривать как ее спецификацию, однако в общем случае спецификации программ часто содержат универсальные кванторы и другие логические символы. Используя теорию обобщенных индуктивных определений, мы можем определять предикаты высших *уровней* в терминах произвольных формул первого порядка, построенных из предикатов низших уровней, предикаты более высокого уровня служат спецификациями для предикатов более низкого уровня. Вообще говоря, предикаты более высокого уровня не являются (или не могут быть сделаны) непосредственно выполнимыми, а должны или преобразовываться в предикаты низшего уровня, или использоваться для доказательства некоторых свойств предикатов низшего уровня. Но, разумеется, нет видимых причин для того, чтобы мы *должны* были отказаться от их непосредственного выполнения.

В нашем представленном здесь расширении логических программ условие предложения может содержать любые логические символы. Это позволяет записывать спецификацию, содержащую кванторы, как логическую программу. Удается также значительно расширить за счет этого класс схем индукций, чтобы включить обычную индукцию по значениям.

Так как один из наиболее проблематичных вопросов в логическом программировании состоит в том, как обращаться с отрицанием, мы обсудим эту проблему в нашем формализме. Оказывается, что правило, трактующее *отрицание как неудачу* [5], выводимо в нашей системе.

2. ID

Введем логическую систему **ID**.

2.1. Символы

Символами **ID** являются следующие:

(L1) *Константы*

Счетное число индивидных констант: *последовательность букв латинского алфавита, начинающаяся в прописной буквы, — например* Zero, Nil.

Счетное число n-арных функциональных констант для каждого $n \geqslant 0$: *последовательность строчных букв латинского алфавита, возможно, с индексами, — например* cons, s, fact.

Нульарная предикатная константа уровня 0: $\bot$.

Бинарная предикатная константа уровня 0: $=$.

Счетное число n-арных предикатных констант уровня m для каждого $n \geqslant 0$ и $m \geqslant 1$: *последовательность букв латинского алфавита*[1]), *начинающаяся с прописной буквы, возможно, с индексами, — например* Nat, List, Fib.

(L2) *Переменные*

Счетное число индивидных переменных: *буква латинского алфавита, возможно, с индексами — например* x, y, a, b, l.

(L3) *Логические символы*

$\wedge$, $\vee$, $\supset$, $\forall$, $\exists$.

2.2. Термы, формулы

Термы определяются следующим образом:

(T1) Индивидная константа есть терм.

(T2) Переменная есть терм.

(T3) Если f — n-арная функциональная константа и $t_1, \ldots, t_n$ — термы, то $f(t_1, \ldots, t_n)$ — терм.

Атомарные формулы (*предикаты*) определяются следующим образом:

(A1) Если P — n-арная предикатная константа и $t_1, \ldots, t_n$ — термы, то $P(t_1, \ldots, t_n)$ — атомарная формула.

Число n назовем *арностью* P. Будем считать, что одинаковые по написанию предикатные константы имеют фиксированные арность и уровень. Мы будем вместо $\bot()$, $=(s, t)$, кратко писать $\bot$, $s = t$, и если s и t являются последовательностями термов длины n, т. е. s есть $s_1, \ldots, s_n$ и t есть $t_1, \ldots, t_n$, то вместо $s_1 = t_1, \ldots, s_n = t_n$ пишем $s = t$.

Формулы определяются как обычно, причем $\neg A$, $A \leftrightarrow B$ и $A_1, \ldots, A_n \supset B$ считаются сокращениями для $A \supset \bot$, $(A \supset B) \wedge (B \supset A)$ и $A_1 \supset (\ldots (A_n \supset B) \ldots)$ соответственно.

Свободные и *связанные вхождения* переменной в формулу определяются как обычно.

Для того чтобы определить *P-формы*, введем счетное число символов, скажем, $*_1$, $*_2$, ..., не встречающихся в **ID**. P-формы определяются следующим образом:

(P1) $*_i$ есть P-форма.

---

[1]) Для наглядности мы иногда будем использовать вместо латинского алфавита русский — например Нат, Список, Фиб. — *Прим. перев.*

(P2) Формула есть P-форма.

(P3) Если $\mathbf{P}$ и $\mathbf{Q}$ — P-формы, то $\mathbf{P} \wedge \mathbf{Q}$, $\mathbf{P} \vee \mathbf{Q}$, $\forall x\mathbf{P}$ и $\exists x\mathbf{P}$ — P-формы.

(P4) Если F — формула и $\mathbf{P}$ — P-форма, то $F \supset \mathbf{P}$ — P-форма.

Степень P-формы $\mathbf{P}$ есть наибольшее n, такое, что $*_n$ входит в $\mathbf{P}$. Если $\mathbf{P}$ есть P-форма степени n, и $F_1, \ldots, F_n$ — формулы, то $\mathbf{P}[F_1, \ldots, F_n]$ обозначает результат замены символов $*_1, \ldots, *_n$ в $\mathbf{P}$ на $F_1, \ldots, F_n$ соответственно.

Для синтаксических переменных мы будем использовать следующие обозначения:

x, y, z, a	для переменных,
$\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$	для последовательностей переменных,
r, s, t, u	для термов,
$\boldsymbol{r}, \boldsymbol{s}, \boldsymbol{t}, \boldsymbol{u}$	для последовательностей термов,
P, Q, R	для предикатных констант,
A, B, C, F, G, H	для формул,
$\mathbf{P}, \mathbf{Q}$	для P-форм,
Γ, Δ	для последовательностей формул.

Переменные могут также иметь индексы.

Секвенции определяются следующим образом:

(S1) $\Gamma \rightarrow F$ есть секвенция.

Обозначения для *подстановки* определяются следующим образом:

(ST) Пусть $\boldsymbol{x}$ есть $x_1, \ldots, x_n$ и $\boldsymbol{t}$ есть $t_1, \ldots, t_n$. Тогда $s_{\boldsymbol{x}}(\boldsymbol{t})$ обозначает терм, полученный одновременной подстановкой t_i вместо всех вхождений x_i.

(SF) Пусть $\boldsymbol{x}$ есть $x_1, \ldots, x_n$ и $\boldsymbol{t}$ есть $t_1, \ldots, t_n$. Тогда $A_{\boldsymbol{x}}(\boldsymbol{t})$ обозначает формулу, полученную одновременной подстановкой t_i вместо всех вхождений x_i с переименованием связанных переменных в A, если переменные из $\boldsymbol{t}$ оказываются связанными при подстановке в A.

Если A и B — формулы, такие, что A есть $B_{\boldsymbol{x}}(\boldsymbol{t})$, то для выражения этого факта мы будем писать $B(\boldsymbol{t})$ вместо A и $B(\boldsymbol{x})$ вместо B.

Мы будем говорить, что d' есть *вариант* d *относительно* e (d, e являются последовательностями термов или формул), если d' получается из d переименованием переменных и d' и e не имеют общих свободных переменных.

2.3. Правила вывода

Правило вывода имеет вид

$$\frac{S_1 \ \ldots \ S_n}{S} \quad n \geqslant 0,$$

где S_i, S — секвенции.

Мы назовем $S_1, \ldots, S_n$ *посылками* и S *заключением* правила вывода.

Доказательство определяется как обычно.

Правила вывода состоят из следующих трех групп:

(1) интуиционистская логика,

(2) правила вывода для равенства,

(3) введение продукции и устранение продукции.

⟨1⟩ *Логика*

Логической частью системы **ID** является интуиционистская логика.

Правила вывода здесь таковы:

$$(Ax)\quad \frac{}{\Gamma, A, \Delta \to A}$$

$$(Wk)\quad \frac{\Gamma \to A}{B, \Gamma \to A}$$

$$(Cntr)\quad \frac{A, A, \Gamma \to B}{A, \Gamma \to B}$$

$$(Ex)\quad \frac{\Gamma, A, B, \Delta \to C}{\Gamma, B, A, \Delta \to C}$$

$$(Cut)\quad \frac{\Gamma \to A \quad A, \Delta \to B}{\Gamma, \Delta \to B}$$

$$(\wedge I)\quad \frac{\Gamma \to A \quad \Gamma \to B}{\Gamma \to A \wedge B} \qquad (\wedge E)\quad \frac{\Gamma \to A \wedge B}{\Gamma \to A} \quad \frac{\Gamma \to A \wedge B}{\Gamma \to B}$$

$$(\vee I)\quad \frac{\Gamma \to A}{\Gamma \to A \vee B} \qquad \frac{\Gamma \to B}{\Gamma \to A \vee B}$$

$$(\vee E)\quad \frac{\Gamma \to A \vee B \qquad A, \Gamma \to C \qquad B, \Gamma \to C}{\Gamma \to C}$$

$$(\supset I)\quad \frac{A, \Gamma \to B}{\Gamma \to A \supset B} \qquad (\supset E)\quad \frac{\Gamma \to A \supset B \quad \Gamma \to A}{\Gamma \to B}$$

$$(\forall I)\quad \frac{\Gamma \to A(a)}{\Gamma \to \forall x\, A(x)} \qquad (\forall E)\quad \frac{\Gamma \to \forall x\, A(x)}{\Gamma \to A(t)}$$

$$(\exists I)\quad \frac{\Gamma \to A(t)}{\Gamma \to \exists x\, A(x)} \qquad (\exists E)\quad \frac{\Gamma \to \exists x A(x) \quad A(a), \Gamma \to B}{\Gamma \to B}$$

$$(\bot E)\quad \frac{\Gamma \to \bot}{\Gamma \to A}$$

где в $(\forall I)$ a не входит свободно в Γ и в $(\exists I)$ a не входит свободно в B и Γ.

⟨2⟩ *Правила вывода для равенства*

Подразумеваемым значением предикатной константы «=» является, естественно, равенство. Так как некоторые правила для характеризации равенства нам необходимы, введем

базисные правила

$$\frac{}{\to t = t} \qquad \frac{}{s = t \to t = s} \qquad \frac{}{r = s,\ s = t \to r = t}$$

$$\frac{}{s = t \to r_x(s) = r_x(t)} \qquad \frac{}{A(s),\ s = t \to A(t)}$$

где s и t имеют одинаковую длину.

Более того, что касается равенства и ложности, мы можем ввести любые правила, при условии, что они не нарушают ограничений на уровни (см. 2.3. ⟨3⟩). Например, мы можем ввести некоторые правила Кларка [5]:

$\dfrac{c = c'}{\bot}$ *для различных индивидных констант* c, c'

$\dfrac{f(x_1, \ldots, x_n) = g(y_1, \ldots, y_m)}{\bot}$ *для различных функциональных констант* f, g

$\dfrac{f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)}{x_i = y_i}$ *для любой функциональной константы* f

$\dfrac{f(x_1, \ldots, x_n) = c}{\bot}$ *для любой функциональной константы* f *и индивидной константы* c

$\dfrac{t = x}{\bot}$ *для любого терма* t, *содержащего* x

Они сформулированы в виде, который мы называем продукциями (см. 2.3 ⟨3⟩). С помощью этих правил мы можем объяснить механизм унификации, который используется в Прологе (см. 5.1). Назовем эти правила Peq.

С другой стороны, с помощью некоторых правил можно задавать интерпретацию функциональных констант, — например,

$$\frac{}{\mathrm{fact}(0) = \mathrm{s}(0)} \qquad \frac{}{\mathrm{fact}(\mathrm{s}(\mathrm{x})) = \mathrm{times}(\mathrm{s}(\mathrm{x}), \mathrm{fact}(\mathrm{x}))}$$

...

⟨3⟩ *Введение продукции и устранение продукции*

Эти правила наиболее важны и полезны при доказательстве формул в **ID**. Они развивают правила, введенные Мартин-Лёфом [12]. Определим вначале продукцию.

(а) *Продукция*

Продукции — это схемы для индуктивного определения предикатов. Они имеют вид фигур

$$\frac{F_1 \ldots F_n}{P(t)} \quad n \geqslant 0 \qquad \text{(p)}$$

где t — последовательность термов, P — **предикатная** константа, F_i — формула

$$\mathbf{P}_i[Q_{i_1}(t_{i_1}), \ldots, Q_{ik_i}(t_{ik_i})],$$

$\mathbf{P}_i$ — P-форма степени k_i, Q_{ij} — предикатная константа, при чем выполняется условие на уровни:

уровни Q_{ij} должны быть меньше или равны уровню P, и уровни предикатов в $\mathbf{P}_i$ должны быть меньше уровня P Назовем $F_1, \ldots, F_n$ *условиями* продукции (p).

Пример

$$\frac{}{\text{Список(Nil)}} \qquad \frac{\text{Список(l)}}{\text{Список(cons (x, l))}}$$

(б) *Введение продукции*

Правило *введения продукции* для продукции (p) имеет вид

$$\frac{\Gamma,\ s = t',\ \Delta \to F_1',\ \ldots,\ \Gamma,\ s = t',\ \Delta \to F_n'}{\Gamma,\ s = t',\ \Delta \to P(s)} \qquad (\text{pI})$$

где t', F_1', ..., F_n' есть вариант t, $F_1, \ldots, F_n$ относительно s.

Пример

$$\frac{\text{cons(u,cons(v,w)) = cons(x,1)} \to \text{Список(1)}}{\text{cons(u,cons(v,w)) = cons(x,1)} \to \textbf{Список(cons(u,cons(v,w)))}}$$

(в) *Устранение продукции*

Для того чтобы определить *устранение продукции*, введем определение *связи*, которое является отношением между предикатными константами.

(1) Любая предикатная константа связана сама с собой.

(2) Если предикатная константа P входит в заключение следующей продукции

$$\frac{\ldots\ \mathbf{P}_i[\ldots, Q_{ij}(s_{ij}), \ldots]\ \ldots}{P(s)},$$

то P связана с любой предикатной константой, связанной с Q_{ij}.

Устранение продукции для индуктивно определенной предикатной константы имеет вид

$$\frac{\Gamma \to P(t)\ \textit{малые посылки}}{\Gamma \to F} \qquad (\text{pE})$$

Для того чтобы объяснить, каким образом строить малые посылки, выберем сначала произвольное множество **Ps** предикатных констант, содержащее P и такое, что все его члены связаны с P. С каждой предикатной константой из **Ps** ассоциируем формулу и последовательность термов следующим образом.

(1) С P ассоциируем F и t.

(2) С любой отличной от P предикатной константой Q ассоциируем произвольную формулу и последовательность термов, длина которой равна арности Q.

Малая посылка строится для каждой пары, состоящей из предикатной константы Q из **Ps** и продукции, заключение которой содержит Q. Пусть продукция имеет вид

$$\frac{\ldots\ \mathbf{Q}_i[\ldots,\ R_{ij}(\boldsymbol{s}_{ij}),\ \ldots]\ \ldots}{Q(\boldsymbol{s})}$$

и пусть

$\boldsymbol{r}'$, G', $\boldsymbol{r}'_{ij}$, G'_{ij} — вариант $\boldsymbol{r}$, G, $\boldsymbol{r}_{ij}$, G_{ij} относительно Γ,

$\boldsymbol{z}_{ij}$ — последовательность всех переменных из $\boldsymbol{r}_{ij}$,

$\boldsymbol{s}'$, F_i, H_i — вариант $\boldsymbol{s}$, $\mathbf{Q}_i[\ldots,\ R_{ij}(\boldsymbol{s}_{ij}),\ \ldots]$,

$\mathbf{Q}_i[\ldots,\ H_{ij},\ \ldots]$ относительно $\boldsymbol{r}'$, G', $\boldsymbol{r}'_{ij}$, G'_{ij}, Γ,

где

G и $\boldsymbol{r}$ ассоциированы с Q,

$$H_{ij}=\begin{cases}\forall \boldsymbol{z}_{ij}\,(\boldsymbol{r}'_{ij}=\mathbf{s}_{ij}\supset G'_{ij}), & \text{если } G_{ij} \text{ и } \boldsymbol{r}_{ij} \text{ ассоциированы с } R_{ij}\\ R_{ij}(\boldsymbol{s}_{ij}) & \text{в противном случае.}\end{cases}$$

Соответствующая малая посылка имеет вид

$$\boldsymbol{r}'=\boldsymbol{s}',\ \ldots,\ F_i,\ H_i,\ \ldots,\ \Gamma\to G' \qquad \text{(PpE)}$$

Для заданных P, t и F может найтись несколько устранений продукции в зависимости от **Ps** и установленной ассоциации.

Пример

Для определенной выше предикатной константы «Список» устранение продукции имеет вид

$$\frac{\Gamma\to \text{Список}(t)\ t_y(z)=\text{Nil},\Gamma\to \text{F}(z)\ t_y(z)=\text{cons}(x,l),\ \text{Список}(l),\ \forall z(t_y(z)=l\supset \text{F}_y(z)),\ \Gamma\to \text{F}(z)}{\Gamma\to \text{F}(y)}$$

где y — единственная переменная в t, а x и l не входят в Γ свободно. Чтобы легче было понять, мы укажем, как конструкции второй малой посылки в вышеприведенной схеме соответствуют конструкциям из (PpE):

t ..... $\boldsymbol{r}$

$t_y(z)$ ..... $\boldsymbol{r}'$

z ..... $\boldsymbol{z}_{ij}$

cons(x, l) ..... $\boldsymbol{s}'$

F(y)..... G

F(z) ..... G'

Список ... Q

Перечисленные правила составляют наше определение **ID**. Если нам понадобится явно указать, что в качестве множества продукций **ID** выступает I, будем использовать обозначение $\mathbf{ID}(I)$. Символ доказуемости $\vdash_{\mathbf{ID}(I)}$ будет использоваться как обычно.

3. ПРОДУКЦИЯ

3.1. Смысл устранения продукции

Устранение продукции может показаться весьма сложным, поэтому поясним его смысл на примере:

$$(n1)\ \frac{}{\mathbf{Нат}(0)} \qquad (n2)\ \frac{\mathbf{Нат}(x)}{\mathbf{Нат}(s(x))}$$

где 0 — сокращение для индивидной константы «Нуль». Подразумеваемый смысл записи Нат(x) в том, что x — натуральное число. Одно из правил устранения продукции для Нат есть

$$\frac{\Gamma \to \mathbf{Нат}(x)\ y = 0, \Gamma \to F(y)\ y = s(z), \mathbf{Нат}(z)\ \forall y(y = z \supset F(y)), \Gamma \to F(y)}{\Gamma \to F(x)}$$

В качестве производного правила получаем

$$\frac{\Gamma \to \mathbf{Нат}(x)\ \Gamma \to F(0)\ \mathbf{Нат}(z), F(z), \Gamma \to F(s(z))}{\Gamma \to F(x)}$$

что является схемой индукции для натуральных чисел. Это также означает, что Нат является минимальным решением уравнения [1])

$$X(0) \wedge \forall x(X(x) \supset X(s(x))),$$

в котором предикатная переменная X выступает в качестве неизвестной.

Аналогично из устранения продукции

$$\frac{\Gamma \to P(x)\ \textit{малые посылки}}{\Gamma \to F}$$

мы можем вывести схему индукции, означающую, что P есть минимальное решение.

---

[1]) Здесь авторы употребляют термин «уравнение» довольно необычным образом: под уравнением понимается произвольная формула, содержащая неизвестную предикатную переменную. Более похожая на уравнение запись получится, если эту формулу переписать в виде

$$X(0) \wedge \forall x(X(x) \supset X(s(x))) \equiv \text{Истина}. \text{ — } \textit{Прим. перев.}$$

Грубо говоря, малые посылки устранения продукции получаются заменой предиката в продукциях ассоциированной с ним формулой, причем замена допускается только тогда, когда аргумент предиката принадлежит некоторой области, определяемой последовательностью термов, ассоциированной с предикатной константой (в (PpE) эту область определяют $r' = s'$ и $r'_{ij} = s'_{ij}$). Это означает, что устранение продукции выражает минимальность ограниченного предиката.

3.2. Преимущества устранения продукции

Причина, по которой мы приняли устранение продукции, состоит в том, что оно приводит к широкому спектру схем индукции. То что устранение продукции имеет вид индукции, вполне естественно, потому что наименьшая неподвижная точка преобразования, ассоциированного с продукциями (см. Апт и ван Эмден [2]), равна объединению конечных степеней преобразования, примененного к наименьшему элементу (наименьший элемент соответствует базису, а преобразование — шагу индукции).

Другая причина, по которой мы используем продукцию, состоит в том, что определение предиката с помощью «тогда и только тогда» не подразумевает минимальности определяемого предиката. В качестве простого примера, показывающего, что определение с помощью «тогда и только тогда» не означает минимальности, возьмем определение предикатной константы «Ложь»:

$$\text{Ложь}(\,) \leftrightarrow \text{Ложь}(\,),$$

которое не сообщает ничего определенного о константе «Ложь», в то время как определение с помощью продукции вместе с устранением продукции дает $\text{Ложь}() \leftrightarrow \bot$, как будет видно из примера 3 в разд. 3.3.

3.3. Примеры устранения продукции для простой продукции

Приведем теперь несколько примеров продукций, в которых предикатные константы имеют уровень не больше 1 и все условия атомарные. Мы будем называть такие продукции простыми продукциями.

Пример 1

Устранение продукции не обязательно приводит к обычной схеме индукции. Пусть Нат — предикатная константа, определенная в разд. 3.1. Другим устранением продукции для

Нат может быть

$$\frac{\Gamma \to \text{Нат}(s(x))\ s(y) = 0,\ \Gamma \to F(y)\ s(y) = s(z), \text{Нат}(z), \forall y(s(y) = z \supset F(y)),\ \Gamma \to F(y)}{\Gamma \to F(x)}$$

Так как подразумеваемый смысл для Нат($s(x)$) в том, что $s(x)$ — натуральное число, т. е. x — натуральное число, то доказательства Нат $(s(x)) \to F(x)$ и Нат$(x) \to F(x)$ будут взаимосвязаны. Если введены еще и правила

$$\frac{s(x) = s(y)}{x = y} \qquad \frac{s(x) = 0}{\bot}$$

(некоторые из аксиом Пеано), то правило

$$\frac{\Gamma \to \text{Нат}(s(x))\quad \text{Нат}(z), \Gamma \to F(z)}{\Gamma \to F(x)}$$

является производным.

Пример 2

Если имеются продукции

$$\frac{}{\text{Естьблок(A)}} \qquad \frac{}{\text{Естьблок(B)}} \qquad \frac{}{\text{Естьблок(C)}},$$

то

$$\frac{\Gamma \to \text{Естьблок}(x)\ y = A, \Gamma \to F(y)\ y = B, \Gamma \to F(y)\ y = C, \Gamma \to F(y)}{\Gamma \to F(x)}$$

есть устранение продукции, позволяющее проводить рассуждения разбором случаев. В этом устранении продукции x может входить в Γ, а y нет. Однако, если x входит в Γ, желательно, чтобы малыми посылками были $x = A$, $\Gamma \to F(x)$ и т. д. Тогда мы сможем вывести следующее правило

$$\frac{\Gamma(x) \to \text{Естьблок}(x) \quad \dfrac{y = A, \Gamma(x), \Gamma(y) \to F(y) \quad y = A, \Gamma(x) \to \Gamma(y) \supset F(y)}{\Gamma(x) \to \Gamma(x) \supset F(x)} \quad \dfrac{y = B, \Gamma(x), \Gamma(y) \to F(y) \quad y = B, \Gamma(x) \to \Gamma(y) \supset F(y)}{\Gamma(x) \to \Gamma(x) \supset F(x)} \quad \dfrac{y = C, \Gamma(x), \Gamma(y) \to F(y) \quad y = C, \Gamma(x) \to \Gamma(y) \supset F(y)}{\Gamma(x) \to \Gamma(x) \supset F(x)}}{\Gamma(x) \to F(x)}$$

Пример 3

Определим теперь нульместный предикат Ложь() следующим образом

$$\frac{\text{Ложь}(\,)}{\text{Ложь}(\,)}$$

Используя устранение продукции для Ложь, мы можем доказать ложь$() \to F$ для любой формулы F. Доказательство следующее:

$$\frac{\dfrac{}{\text{Ложь}(\,) \to \text{Ложь}(\,)} \quad \dfrac{}{\text{Ложь}(\,), F, \text{Ложь}(\,) \to F}}{\text{Ложь}(\,) \to F}$$

Таким образом, мы можем доказать

Ложь( ) $\leftrightarrow \perp$

Как будет объяснено в разд. 4, эти продукции можно рассматривать как программы на языке Пролог. Если мы начнем выполнять Ложь( ), вычисление не закончится из-за бесконечной рекурсии. Однако, так как значением Ложь( ) является $\perp$, желательно, чтобы выполнение заканчивалось неудачей. Здесь мы даем представление Пролога, которое содержит средства, делающие это возможным. Заметим, что эти средства разумны, так как мы приняли семантику наименьшей неподвижной точки.

4. ПРОЛОГ И ЕГО ОСНОВЫ

4.1. Чистый Пролог

Так как мы будем иметь дело с программой на Прологе, которая соответствует множеству продукций из **ID**, мы предполагаем, что

(1) любой предикат имеет фиксированную арность,
(2) аргумент является термом **ID**.

Например, если мы выполняем следующую цель

$\leftarrow$ Сумма(s(s(0)), s(0), z)

при заданной программе

Сумма(0, y, y) $\leftarrow$
Сумма(s(x), y, s(z)) $\leftarrow$ Сумма(x, y, z),

то получаем ответ

z = s(s(s(0))).

Когда предикат A выполняется при заданной программе P и выполнение оканчивается удачно, мы будем писать

$P \vdash_{\text{Prolog}} A$,

а когда предикат A выполняется при заданной программе P и выполнение оканчивается неудачно, мы будем писать

$P \nvdash_{\text{Prolog}} A$.

Программа на языке Пролог может рассматриваться как множество простых продукций, в которых уровни предикатных констант равны 1. Приведенная выше программа превра-

щается в

$$\frac{}{\text{Сумма}(0,y,y)} \quad \frac{\text{Сумма}(x,y,z)}{\text{Сумма}(s(x),y,s(z))}.$$

Выполнение предиката в Прологе соответствует его доказательству в ID. Для приведенного выше примера порождается следующее доказательство

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \to \text{Сумма}(x_2, y_2, z_2)}}{\Gamma \to \text{Сумма}(x_1, y_1, z_1)}}{\Gamma \to \text{Сумма}(s(s(0)), s(0), z)}}{\Gamma \to \exists z\ \text{Сумма}(s(s(0)), s(0), z)}$$

где Γ есть

$$s(s(0)) = s(x_1), \quad s(0) = y_1, \quad z = s(z_1), \quad x_1 = s(x_2),$$
$$y_1 = y_2, \quad z_1 = s(z_2), \quad x_2 = 0, \quad y_2 = y_3, \quad z_2 = y_3.$$

Сравним это доказательство с *процедурной интерпретацией* [11]. Применение правила вывода соответствует вызову процедуры, а Γ — унификации. Анализируя Γ, получаем ответ $z = s(s(s(0)))$.

Заметим, что это доказательство нормальное (определение нормального доказательства см. в [12, 14]).

Назовем Γ *и-последовательностью*, если Γ является последовательностью формул вида $s = t$ и Γ непротиворечиво. Следовательно, является ли Γ и-последовательностью или нет, это определяется правилами для равенства.

4.2. Основы Пролога

Обычно объясняют, что Пролог базируется на SLD-резолюции [2, 11]. Однако более естественно рассматривать программу на Прологе и ее исполнение как множество продукций и порождение нормального вывода, чем как множество хорновских дизъюнктов и SLD-резолюцию, так как первое вернее отражает процедурную интерпретацию логики предикатов и, используя теорему о нормализации, легче и естественнее доказать теорему полноты (см. 4.4). Отношение между естественным выводом и резолюцией исследуется, например, в [1] и в [3]. Некоторые процедуры, основанные на резолюции, лучше понимать в терминах вывода, даже если вывод и опровержение эквивалентны. В нашем случае эквивалентность почти тривиальна, однако, когда мы расширяем хорновские дизъюнкты до произвольных формул первого порядка, мы получаем еще одно преимущество (см. разд. 5.2 и 5.3).

4.3. Правильность Пролога

По программе P.p на языке Пролог мы можем построить множество продукций P.

В соответствии с (довольно неформальным) объяснением Пролога в разд. 4.1,

$P.\mathrm{p} \vdash_{\mathrm{Prolog}} A \Rightarrow \vdash_{\mathrm{ID}(P \cup \mathrm{Peq})} \Gamma \to \exists x A$ *для некоторой и-последовательности* Γ,

где A — предикат, определяемый в P, и x — последовательность свободных переменных из A, из чего следует *правильность* [1]). Более того, для Пролога с контролем вхождений

$P.\mathrm{p}' \vdash_{\mathrm{Prolog}} A \Rightarrow \vdash_{\mathrm{ID}(P \cup \mathrm{Peq})} \to \exists x A.$

4.4. Полнота Пролога

Полноту можно описать следующим образом:

$\vdash_{\mathrm{ID}(P \cup \mathrm{Peq})} \to \exists x A \Rightarrow P.\mathrm{p} \vdash_{\mathrm{Prolog}} A.$

Однако по поводу того, имеет это место в действительности или нет, возникают две проблемы.

Первая проблема состоит в том, что доказательство $\to \exists x A$ в $\mathbf{ID}(P \cup \mathrm{Peq})$ должно быть нормальным, для того чтобы при заданной программе P.p вызов A оканчивался успешно. В общем случае доказательство $\to \exists x A$ не обязательно нормальное. Однако по теореме нормализации [12], если существует доказательство формулы F, то существует и нормальное доказательство F. В этом случае, так как A — предикат, то нормальное доказательство $\to \exists x A$ состоит только из введений продукции, за исключением последнего правила $\exists I$.

Вторая проблема следующая. Как объясняется в разд. 4.1, Пролог ищет нормальное доказательство A. В случае идеального поиска, если A выполняется, Пролог найдет нормальное доказательство A. Однако обычно Пролог использует поиск в глубину, который может привести к бесконечному поиску. Если же ввести поиск в ширину, то Пролог отыщет нормальное доказательство, если оно существует, т. е. имеет место полнота.

Итак, если $\vdash_{\mathrm{ID}(P \cup \mathrm{Peq})} \to \exists x A$, то или имеет место $P.\mathrm{p} \vdash_{\mathrm{Prolog}} A$, или вычисление не заканчивается, однако не-

[1]) В оригинале soundness (по-русски это называют еще *корректностью* или *здравостью*). Вообще формальную систему называют *правильной* (*корректной*, *здравой*), если выводимые в ней формулы подкрепляются некоторой разумной содержательной интерпретацией (например, выводимые формулы исчисления высказываний тождественно истинны). В данном случае выводимость A из P (отношение $P \vdash_{\mathrm{Prolog}} A$) влечет выводимость $\Gamma \to \exists x A$ в $\mathbf{ID}(P \cup \mathrm{Peq})$. — *Прим. ред.*

возможно, чтобы вычисление A завершалось неудачно, несмотря на то, что $\vdash_{\mathbf{ID}(P \cup \mathrm{Peq})} \exists x A$.

Сравним нашу трактовку полноты с приведенной в [2] для SLD-резолюции. Так как мы придерживаемся теоретико-доказательственного подхода, нам нет нужды обращаться к теории моделей. Как было указано выше, если Пролог идеальный (т. е. использует стратегию поиска в ширину), то он полон, что соответствует полноте SLD-резолюции. Когда Пролог ищет доказательство, возможны два выбора:

(1) выбор порядка доказательства посылок,

(2) выбор продукций.

Наш подход дает интуитивное представление о том, что (1) не влияет на полноту, и это действительно так. Соответствующий результат доказан и для SLD-резолюции [2]. Если строго описать наше доказательство того, что (1) можно не учитывать, то оно будет не слишком отличаться от соответствующего доказательства для SLD-резолюции, хотя преимущество нашего подхода в том, что понять его гораздо легче.

5. РАСШИРЕНИЯ

5.1. Отрицание как неудача

Отрицание как неудачу можно проинтерпретировать как ряд применений устранений продукции, что видно из следующего примера.

Пример

Рассмотрим следующую программу на Прологе

(N1) Нат(0) ←

(N2) Нат(s(x)) ← Нат(x)

(N3) ← Нат(s(t(0))).

Так как выполнение (N3) заканчивается неудачно, то предполагается, что имеет место $\neg$ Нат(s(t(0))), потому что мы используем *отрицание как неудачу*. (N1) и (N2) переводятся в продукции (n1) и (n2) из разд. 3.1 и $\neg$ Нат(s(t(0))) доказывается в $\mathbf{ID}(\{n1), (n2)\} \cup \mathrm{Peq})$ следующим образом:

$$\dfrac{\dfrac{\mathrm{Нат}(s(t(0))), \Gamma \to \mathrm{Нат}(s(t(0)))\quad s(t(0)) = 0, \Gamma' \to \bot\quad s(t(0)) = s(x), \mathrm{Нат}(x), s(t(0)) = x \supset \bot, \Gamma'' \to \bot}{\mathrm{Нат}(s(t(0))), \Gamma \to \bot}\,\Pi}{\Gamma \to \neg\,\mathrm{Нат}(s(t(0)))}$$

где Γ — $\mathbf{u}$-последовательность, в которую не входят свободно x и y, Γ' есть Нат(s(t(0))), Γ и Π имеет вид

$$\dfrac{\mathrm{Нат}(t(0)), \Gamma' \to \mathrm{Нат}(t(0))\quad t(0) = 0, \Gamma'' \to \bot\quad t(0) = s(y), \mathrm{Нат}(y), t(0) = y \supset \bot, \Gamma'' \to \bot}{\mathrm{Нат}(t(0)), \Gamma' \to \bot}$$

где Γ'' есть Нат(t(0)), Γ'.

Заметим, что $s(t(0)) = 0 \to \bot$, $t(0) = 0 \to \bot$, $t(0) = s(y) \to \bot$ соответствуют трем неудачным попыткам выполнения (N3), так как s(t(0)) не унифицируемо с 0, t(0) с 0, t(0) с s(y).

В общем случае мы можем описать *отрицание как неудачу* следующим образом:

$$P.\mathrm{p} \nvdash_{\mathrm{Prolog}} A \Rightarrow \vdash_{\mathbf{ID}(P \cup \mathrm{Peq})} \to \forall x \neg A$$

и доказать это путем перевода неудачного поиска в смысле Кларка [5] в доказательство в **ID**. Так как сделать это достаточно просто, мы оставляем доказательство читателю.

Как видно из приведенных обсуждений, унификацию в Прологе можно рассматривать как встроенную процедуру проверки равенства в **ID**(Peq). Следовательно, если мы имеем другие правила для равенства и процедуру для проверки равенства и используем ее вместо унификации, мы получаем видоизменение Пролога, т. е. Пролог с функциями в качестве термов.

5.2. Введение предикатов высших уровней

⟨1⟩ *Условие на уровни*

До сих пор мы рассматривали только такие продукции, которые называли простыми. Для так называемого чистого Пролога достаточно простых продукций, однако, как обсуждалось в разд. 5.1, обычный интерпретатор Пролога может обрабатывать отрицания предикатов с помощью *отрицания как неудачи*, и фактически предикатная константа может определяться через отрицания других предикатов. Допущение произвольных продукций, вообще говоря, не удовлетворяющих условию на уровни, определенному в п. 2.3 ⟨3⟩ (а), может быстро привести к противоречию. В качестве простейшего примера рассмотрим следующую продукцию:

$$\frac{\neg\, \text{Лжец}(\,)}{\text{Лжец}(\,)}$$

Рассматривая Лжец( ) как минимальное решение для

$$\neg X \supset X$$

заключаем, что Лжец( ) истинно. Но применяя следующее устранение продукции

$$\frac{\text{Лжец}(\,) \to \text{Лжец}(\,)\ \neg\, \text{Лжец}(\,),\ \neg\,\neg\, \text{Лжец}(\,), \text{Лжец}(\,) \to \neg\, \text{Лжец}(\,)}{\text{Лжец}(\,) \to \neg\, \text{Лжец}(\,)}$$

и предполагая, что Лжец( ) истинно, немедленно получаем противоречие. Это означает, что при нарушении условия на уровни устранение продукции и наивная семантика наимень-

шей неподвижной точки несовместимы. Так получается из-за того, что (недопустимые) продукции не вводят монотонное преобразование в смысле Апта и ван Эмдена [2], как обсуждалось в разд. 3.2.

Условие на уровни требует, чтобы предикатные константы низших уровней были полностью определены до начала процесса определения предикатных констант высших уровней. Таким образом, гарантируется монотонность соответствующего преобразования на каждом уровне определения, так что устранение продукции оказывается верным по отношению к семантике наименьшей неподвижной точки правилом.

⟨2⟩ *Расширение с помощью отрицания*

Рассмотрим программы на языке Пролог, в которых отрицание предиката может появляться в качестве посылки дизъюнкта. При таком расширении, преобразуя дизъюнкты в продукции, мы должны явно проверять условие на уровни.

Пример

Элемент и Добавить

$$\frac{}{\text{Элемент}(x,\text{cons}(x,l))} \qquad \frac{\text{Элемент}(x,l)}{\text{Элемент}(x,\text{cons}(y,l))}$$

$$\frac{\text{Элемент}(x,l)}{\text{Добавить}(x,l,l)} \qquad \frac{\neg\,\text{Элемент}(x,l)}{\text{Добавить}(x,l,\text{cons}(x,l))}$$

Если присвоить уровень 1 предикатной константе Элемент и уровень 2 предикатной константе Добавить, то эти продукции удовлетворяют условию на уровни.

Анализируя существующие программы на языке Пролог, мы убедились, что практически все *логические* программы на Прологе удовлетворяют условию на уровни при подходящем присваивании уровней. Для расширенных программ *отрицание как неудачу* можно обосновать, преобразуя выполнение, использующее *отрицание как неудачу*, в последовательность применений введений и устранений продукции, в точности как в разд. 5.1.

5.3. К верификации и синтезу

⟨1⟩ *Понятие верификации и синтеза*

Как обсуждается в [8] и отмечается во введении, поскольку программу на Прологе можно отобразить в логическую формулу (в нашем случае в множество правил вывода), многие свойства программы можно естественно формализовать и доказать в рамках чистой логики, т. е. не прибегая ни к каким посторонним средствам вроде аксиом и правил вывода Хоара. В случае логического программирования

проблема корректности программы формулируется скорее как проблема эквивалентности двух программ. Типичным примером могут служить два определения последовательности Фибоначчи:

$$\frac{}{\text{Фнб}_1(0,1)} \qquad \frac{}{\text{Фнб}_1(1,1)} \qquad \frac{\text{Фиб}_1(x,y)\ \ \text{Фиб}_1(x+1,z)}{\text{Фнб}_1(x+2,y+z)}$$

$$\frac{}{G(0,1,1)} \qquad \frac{G(x,y,z)}{G(x+1,z,y+z)} \qquad \frac{G(x,y,z)}{\text{Фнб}_2(x,y)},$$

где Фиб_2 можно рассматривать как реализацию Фиб_1 (а Фиб_1 как спецификацию Фиб_2), или, выражаясь более сдержанно, Фиб_2 является оптимизацией Фиб_1. Корректно Фиб_2 относительно Фиб_1 можно выразить так:

$$\forall xy(\text{Фиб}_1(x,y) \leftrightarrow \text{Фиб}_2(x,y)).$$

Верификация Фиб_2 относительно Фиб_1 заключается в доказательстве этой формулы. Порождение (автоматическое) определения Фиб_2 из определения Фиб_1, возможно, вместе с доказательством этой формулы является синтезом Фиб_2 из Фиб_1, или преобразованием Фиб_1 в Фиб_2.

⟨2⟩ *Спецификации, содержащие кванторы*

В общем случае спецификации программ часто содержат универсальные кванторы и другие логические символы. Рассмотрим следующую продукцию

$$\frac{\text{Элемент}\ (y,1)\ \forall x(\text{Элемент}(x,1) \supset \text{Менрав}(x,y))}{\text{Макс}(y,1)}$$

Эта продукция будет удовлетворять условию на уровни, если мы присвоим уровень 2 предикатной константе Макс и уровень 1 другим предикатным константам. Так как в обычном Прологе нельзя использовать кванторы и импликации, эта продукция приводит к расширению Пролога. Предикат высшего уровня Макс, который должен использоваться для спецификации программы, определяется продукцией. (Мы можем выполнять Макс, вызывая универсальный доказыватель теорем нашей логической системы.) Одним из возможных реализаций Макс является следующее множество продукций:

$$\frac{}{\text{Макс}_1(x,\text{cons}(x,\text{Nil}))} \qquad \frac{\text{Макс}_1(x,1)\ \ \text{Менрав}(x,y)}{\text{Макс}_1(y,\text{cons}(y,1))} \qquad \frac{\text{Макс}_1(x,1)\text{Менрав}(y,x)}{\text{Макс}_1(x,\text{cons}\,(y,1))}$$

Мы можем добавить формулу

$$\forall y1(\text{Макс}(y,1) \leftrightarrow \text{Элемент}(y,1) \wedge \forall x(\text{Элемент}(x,1) \supset \supset \text{Менрав}(x,y)))$$

в качестве аксиомы для Макс, ввиду ее эквивалентности введению и устранению продукции. Однако в нашей системе вся

нелогическая информация формулируется в виде продукций, так чтобы условие на уровни гарантировало непротиворечивость системы.

⟨3⟩ *Индукция по значениям*[1])

При доказательстве свойств логической программы нам требуются в зависимости от рекурсивной структуры программы различные виды схемы индукции. В нашей системе любая схема индукции может быть выведена как устранение продукции для соответствующего множества (возможно, не только простых) продукций.

Пример. Индукция по значениям для натуральных чисел

$$\frac{\text{Нат}(y)\ \forall x(\text{Нат}(x) \wedge \text{Меньше}(x,y) \supset \text{Натз}(x))}{\text{Натз}(y)}.$$

Что касается условия на уровни, уровень Натз должен быть больше, чем у остальных предикатных констант. Из правила устранения этой продукции мы можем вывести следующую схему:

$$\frac{\Gamma \to \text{Натз}(t)\ \forall x(\text{Нат}(x) \wedge \text{Меньше}(x,y) \supset F(x)), \Gamma \to F(y)}{\Gamma \to F(t)}.$$

Если мы доказали

$$\forall x(\text{Нат}(x) \leftrightarrow \text{Натз}(x)),$$

то можно заменить в схеме Натз(t) на Нат(t), получая таким образом обычную схему индукции по значениям для натуральных чисел.

Схемы такого рода обычно формализуются как метасхемы, и для их обоснования на основе примитивных схем требуется проводить рассуждения на метауровне. Однако в нашей системе такое обоснование имеет вид обычной формулы, как показано выше, и не нуждается в привлечении конструкций метауровня. Этим мы обязаны общности (и сложности) системы продукций.

6. ЗАКЛЮЧИТЕЛЬНЫЕ ЗАМЕЧАНИЯ

Сначала дадим краткое резюме теории моделей нашей системы. Напомним, что конкретная система в **ID** определяется множеством правил, которые имеют дело с равенством и ложью, а также с множеством продукций, в заключении которых стоят предикатные константы уровня $\geqslant 1$. Для того чтобы ввести в систему функции, нам не надо заранее

[1]) В оригинале course-of-value induction. — *Прим. ред.*

определять правила для равенства, а можно вводить в каждую систему свое собственное множество правил. Правила для равенства формулируются в виде продукций. При построении модели конкретной системы мы должны вначале определить область индивидуумов и приписать интерпретацию каждой функциональной константе. Символ равенства будет интерпретироваться тогда как равенство в этой области.

Как было отмечено в разд. 5.2, как только построена модель для равенства, модель для всей системы строится путем взятия наименьшей неподвижной точки на каждом уровне, начиная с множества продукций, заключение которых содержит предикатную константу уровня 1, переходя по уровням в порядке возрастания и используя модели для предикатов низших уровней. Поскольку условия продукций могут содержать кванторы, соответствующее преобразование не обязательно должно быть непрерывным.

Ввиду того, что схема устранения продукции соответствует семантике наименьшей неподвижной точки, она тесно связана с формализацией немонотонной логики. Фактически «ограничение» (circumscription) Маккарти [13] — это почти то же самое, что и схема устранения из работы Мартин-Лёфа [12]. Разница состоит в использовании иерархии уровней, гарантирующей существование минимальной модели. Идея связать с логической программой (а именно с множеством хорновских дизъюнктов) схему индукции для верификации появилась также у Кларка [6].

Боуэн [4] предложил программирование в терминах полной логики предикатов первого порядка, связав с логическим программированием исчисление секвенций. Его формализм практически применим и в нашем случае, так как для интуиционистской логики секвенциальное исчисление и естественный вывод отличаются не очень существенно.

ЛИТЕРАТУРА

[1] Andrew P. B. Transforming matings into natural deduction proofs.— Lecture Notes in Computer Science, v. 87, 1980, p. 281—292.
[2] Apt R. K., van Emden M. H. Contributions to the theory of logic programming. J — ACM, v. 29, no. 3, 1982, p. 841—862.
[3] Bibel W. A syntactic connection between proof procedures and refutation procedures. — Lecture Notes in Computer Science v. 48, 1978, p. 215—224.
[4] Bowen K. A. Programming with full first-order logic. Machine Intelligence 10, 1982, p. 421—440.
[5] Clark K. L. Negation as failure. — Logic and Data Bases, Plenum Press, New York, 1978, p. 293—324.
[6] Clark K. L. Predicate logic as a computational formalism. — Research Monograph 79/59 TOC, Imperial College, London, 1979.

[7] Clark K. L., Darlington J. Algorithmic classification through synthesis. — Computer J., v. 23, no. 1, 1980, p. 61—65.
[8] Clark K. L., Tärnlund S.-Å. A first order theory of data and programs. — Proc. IFIP-77 Congress, North Holland, 1977, p. 939—944.
[9] Hagiya M. Logic programming and inductive definition.— Preprint 420, Research Institute for Mathematical Science, Kyoto Univ., 1983.
[10] Hansson A., Tärnlund S.-Å. A Natural programming calculus. — Proc. 6th Int. Joint Conf. on Artificial Intelligence, 1979, p. 348—355.
[11] Kowalski R. A. Predicate logic as a programming language. — Information Processing, 74, North Holland, 1974, p. 569—574.
[12] Martin-Löf P. Haupsatz for intuitionistic theory of interacted inductive definitions. — Proc. 2nd Scandinavian Logic Symposium, North Holland, 1970, p. 179—216.
[13] McCarthy J. Circumscription — A form of non-monotonic reasoning. — Artificial Intelligence, v. 13, 1980, p. 27—39.
[14] Prawitz D. Natural Deduction. — Almqvist and Wiksell, Stockholm, 1965.
[15] Prawitz D. Ideas and results in proof theory. — Proc. 2nd Scandinavian logic Symposium, North Holland, 1970, p. 235—307.
[16] Sakurai T. Prolog and Inductive definition. — Technical Report 83—10, Univ. of Tokyo, 1983.
[17] Sato T. Negation and semantics of Prolog programs. — Proc. 1st Int. Logic Programming Conf., 1982, p. 169—174.

ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ В ШИРОКОМ СМЫСЛЕ (ОБЗОР)

В. Н. АГАФОНОВ, В. Б. БОРЩЕВ, А. А. ВОРОНКОВ

Резюме. Дается обзор понятий, конструкций языков и направлений логического программирования в широком смысле, к которому кроме Пролога и подобных ему языков и систем программирования («хорновское» и «резолюционное» программирование) относятся программирование с помощью равенств и подстановок (эквациональное и функциональное программирование), логические спецификации и методы автоматического вывода, которые могут найти применение в системах программирования для решения практических задач.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Логическое программирование — это сравнительно новое направление в программировании и информатике, основанное на идеях и методах, идущих из математической логики. Сам термин «логическое программирование», появившийся примерно в 1975 г., толкуют по-разному. При более узком толковании его связывают прежде всего с системами программирования, основанными на использовании специальных классов логических формул (хорновских дизъюнктов) в качестве логических программ и специальных методов логического вы-

вода (вариантов метода резолюций) в качестве логической модели вычислений или способа исполнения логических программ. Самыми известными системами такого рода являются реализации языка Пролог («Программирование в терминах логики» — Programming in logic) и его вариантов и непосредственных расширений. Поэтому логическое программирование в узком смысле иногда называют хорновским, резолюционным или «прологообразным» программированием, хотя каждый из этих эпитетов требует оговорок, так как схватывает только часть предмета (это будет пояснено далее в разд. 1).

При более широком толковании в логическое программирование включают гораздо больший круг понятий, методов, языков и систем, в основе которого лежит идея описания задачи совокупностью утверждений в некотором формальном логическом языке и получение решения построением вывода в некоторой формальной (дедуктивной) системе[1]). Классы формул, используемые для описания задач, методы определения их семантики, модели вычислений, основанные на тех или иных системах вывода или преобразования формул, очень разнообразны. Они образуют специфические «стили» или «виды» программирования, которые существенно отличаются от традиционных, основанных на моделях вычислений, которые воплощены в традиционных языках программирования и в архитектуре типичных вычислительных машин первых четырех поколений (ее иногда называют фоннеймановской[2]). Кроме хорновского (и/или резолюционного) программирования можно говорить об эквациональном программировании, функциональном (аппликативном) программировании и других «программированиях», а также о разных их комбинациях, объединяемых термином «логическое программирование».

Логическое программирование тесно связано с областью автоматической дедукции (автоматического доказательства теорем, поиска вывода и т. д.). Оно появилось вместе с появлением в этой области достаточно эффективных, практически реализуемых и полезных на практике методов (таких, как SLD-резолюция для хорновских дизъюнктов). Эта *практичность* и *эффективная реализуемость* являются характерными чертами логического программирования, которое помимо «чистых» логических и дедуктивных методов использует и способы управления вычислениями, роднящие его с

[1]) Такое толкование поддерживается во введении к первому номеру журнала Journal of Logic Programming (1984 г.) его редактором Дж. Робинсоном.

[2]) По имени наиболее известного из идеологов этой архитектуры Дж. фон Неймана.

«обычным» программированием. Логические описания, еще не достигающие практически приемлемой «степени реализуемости», уместно называть пока логическими спецификациями, а не программами. По мере развития вычислительной техники, обычного программирования, методов автоматической дедукции и самого логического программирования какие-то логические спецификации будут переходить в разряд логических программ.

В этом обзоре речь идет о логическом программировании в широком смысле. Упомянутое выше логическое программирование в узком смысле и непосредственно примыкающие к нему вопросы рассматриваются в разд. 1. Это ядро логического программирования, и следующие разделы являются его расширениями. Раздел 2 посвящен программированию, основанному на равенствах и системах подстановок (эквациональное и функциональное программирование), а также расширению хорновского и резолюционного программирования с помощью равенств. В разд. 3 обсуждаются логические средства, обогащающие выразительные возможности логических описаний, классы формул, формальных систем и логических вычислений, выходящие за границы, очерченные в разд. 1 и 2. Так как расширение средств может приводить к существенной потере эффективности, здесь рассматриваются логические спецификации и их связь с логическими программами.

Следует отметить, что логическое программирование переживает период интенсивного развития, особенно благодаря значению, которое ему стали придавать после того, как оно было положено в основу японского проекта вычислительных систем пятого поколения. Создано 3 новых международных журнала, целиком или в значительной степени посвященных логическому программированию, (Journal of Logic Programming, New Generation Computing, Journal of Automated Reasoning), и вышло несколько книг: [Клоксин 81, ЛП 82, Ллойд 84а, Кларк 84б, Хоггер 84]. Регулярно проводятся международные и национальные конференции и семинары [Конференция МЛП 82, 84, 86; Симпозиум 84; Семинар 80, 83, Конференция ЯЛП 85].

Поток литературы велик и постоянно растет.[1]) Поэтому наш обзор не может претендовать на полноту. Основное его назначение — помочь ориентироваться в этой большой области, идентифицировать значительные ее подобласти, направления, понятия и связи между ними, послужить кратким пу-

[1]) В библиографии [По 84] упоминается 716 работ. Сейчас их гораздо больше.

теводителем по литературе. В обзоре больше внимания уделяется понятийной стороне логического программирования и значительно меньше — программной и аппаратной реализации логических языков и моделей вычислений. За более полной информацией по этим вопросам мы отсылаем к книгам [Реализации 84, Пролог 87] [1]).

Начальные варианты введения и разд. 2 были написаны В. Н. Агафоновым, разд. 1 — В. Б. Борщевым, разд. 3 — А. А. Воронковым. Затем они обсуждались и корректировались всеми авторами. Поэтому каждый из авторов, отвечая прежде всего за свой раздел, несет ответственность и за текст в целом.

Мы благодарим всех, кто нашел время прочесть и прокомментировать всю рукопись обзора или некоторые ее части: М. К. Валиева, А. И. Дегтярева, Г. А. Кучерова, В. А. Непомнящего, В. Ю. Сазонова, Д. И. Свириденко, М. В. Хомякова.

1. ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ В УЗКОМ СМЫСЛЕ

Основная идея. Программа всегда составляется для решения некоторой задачи. Логическая программа описывает *мир этой задачи* — множество объектов, устроенных тем или иным образом, а также функции и отношения на этих объектах. Логическая программа строится как набор утверждений, описывающих эти объекты, а также определяемые на них функции и отношения. Это описание статическое, само по себе оно никакого процесса вычисления не задает. Удобно считать, что оно определяет базу данных, в которой как бы хранятся соответствующие объекты и заданные на них функции и отношения, например, в виде графиков.

С точки зрения логики, программа — это теория, а само множество объектов с заданными на них функциями и отношениями — модель этой теории. Поэтому основная семантика, определяющая стиль мышления в логическом программировании, — это теоретико-модельная семантика.

Конкретному применению программы в логическом программировании соответствует понятие запроса (цели),— например, каково значение функции (заданной программой)

[1]) Из-за ограниченности объема обзора в нем не обсуждаются приложения логического программирования. Им посвящена статья [Домёлки 83] в этом сборнике. Отметим также приложения в области экспертных систем [Кларк 82в, Парсэй 83, Росси 86, Братко 86] и систем обработки текстов на естественных языках [Колмероэ 78, 82б; Перейра 80, Даль 83, Джанессини 84, Маккорд 86].

при данном значении аргумента. По своей семантике понятие запроса в точности соответствует этому понятию в базах данных — в нашем примере нужно как бы найти в базе данных, хранящей эту функцию, «строчку» для данного аргумента и выдать в качестве ответа значение функции. Реально вычисление ответа на запрос соответствует доказательству существования такого объекта (ответа), доказательству, в процессе которого этот объект строится. Правила, по которым проводятся вычисления, образуют процедурную (или операционную) семантику логических программ. Существование такой семантики, собственно, и делает логические программы программами.

1.1. Основные конструкции и механизмы — классическая версия

Начнем с описания основных конструкций и механизмов логического программирования [1]) в их самой простой, «классической» версии [2]). Объектами тут служат термы. На них программа определяет отношения (функции также представляются в виде отношений). Утверждения программы, описывающей такой мир, называются правилами.

Термы и атомы, программа и запросы. Терм — это либо *переменная*, либо *константа*, либо *составной терм* вида $f(t_1, \ldots, t_n)$, где f — n-арный *функциональный символ*, а $t_1, \ldots, t_n$ — термы. Константы удобно понимать как нульарные функциональные символы. *Атомарная формула* (или для краткости *атом*) имеет вид $\omega(t_1, \ldots, t_n)$, где ω — n-арный предикатный символ (имя отношения) и $t_1, \ldots, t_n$ — термы. Термы и атомы, не содержащие переменных, называются *основными* (ground) или *константными*.

Программа — это множество правил. *Правило* имеет вид $A_0 \leftarrow A_1, \ldots, A_m (m \geqslant 0)$, где $A_0, A_1, \ldots, A_m$ — атомы. Атом A_0 называют *заголовком*, а $A_1, \ldots, A_m$ — *телом* правила. Тело может быть пустым (при $m = 0$). Такие правила называются *фактами*. Если атом A_0 в заголовке правила имеет вид $\omega(t_1, \ldots, t_n)$, то говорят, что это правило для предикатного символа ω. *Запрос* (*цель*) имеет вид $\leftarrow C_1, \ldots, C_r$, где $r \geqslant 0$

---

[1]) Более подробное изложение этих идей и конструкций можно найти в монографиях [Ковальский 79б, Ллойд 84а], на русском языке — в монографии [Хоггер 84] и в статьях [Белов 86, Борщев 86а, 86б, Ковальский 83].

[2]) Существуют разные модификации этих конструкций, не всегда совпадающие со стандартными логическими понятиями.

и $C_1, \ldots, C_r$ — атомы. Если $r = 0$, запрос называется пустым [1]).

Итак, все нужные ему объекты программист представляет в виде термов. Например, с помощью константы 0 и унарного символа s (successor) можно представить натуральные числа: нуль — 0, $1 - s(0)$, $2 - s(s(0))$ и т. д. С помощью константы nil (пустой список) и бинарного символа «.» можно представлять списки, составленные из разного рода объектов. Так, $0.s(0).0.\mathrm{nil}$ — это список, в более привычной записи имеющий вид (0, 1, 0) (точка «.» по традиции употребляется инфиксно, некоторые скобки опускаются — в полной префиксной форме этот список имел бы вид $.(0,.(s(0),.(0,\mathrm{nil})))$.

Заметим, что функциональные символы используются здесь как *конструкторы*, позволяющие строить сложные объекты из более простых. При этом разные термы обозначают разные объекты. Функции, задаваемые программой, представляются в виде отношений (n-местная функция $r(x_1, \ldots, x_n) = y$ в виде $(n+1)$-местного отношения $F(x_1, \ldots, x_n, y)$).

На множестве таких объектов программа определяет отношения. Так, программа

$$\mathbf{Plus}(0, x, x) \leftarrow$$

$$\mathrm{Plus}(s(x), y, s(z)) \leftarrow \mathrm{Plus}(x, y, z)$$

определяет сложение (отношение $\mathrm{Plus}(x, y, z)$, означающее, что $x + y = z$) на натуральных числах, а программа

$$\mathrm{App}(\mathrm{nil}, x, x) \leftarrow$$

$$\mathrm{App}(u.x, y, u, .z) \leftarrow \mathrm{App}(x, y, z)$$

— отношение App на множестве списков (соответствующее лисповской функции append — «склеиванию» списков). Эти программы очень похожи на определения рекурсивных функций, по сути дела они служат для определения рекурсивных отношений на множестве термов. Формально они мало отличаются от давно известных конструкций — канонических исчислений Поста [Пост 43] и особенно их модификации — формальных систем Смальяна [Смальян 62] (см. также [Мальцев 65]).

Теоретико-модельная семантика. Какие же отношения задает программа на множестве всех объектов? Чтобы ответить на этот вопрос, фиксируем для программы P две *сигнатуры* Σ и Ω — наборы употребляемых в ней и в запросах функциональных и соответственно предикатных символов. Фиксируем

[1]) Пользуются и другими видами записи. Так, правило иногда записывается в виде $A_0 : - A_1, \ldots, A_m$, а запрос — $? C_1, \ldots, C_r$.

также множество переменных X. Тогда определено множество всех термов $F_\Sigma(X)$ и множество I_Σ всех основных термов.

Подстановкой [1]) называется произвольное отображение переменных в термы θ: $X \to F_\Sigma(X)$. Это отображение естественным образом продолжается на произвольные выражения — так, для терма $t = \sigma(t_1, \ldots, t_n)$ и атома $A = \omega(t_1, \ldots, t_n)$ имеет место $\theta t = \sigma(\theta t_1, \ldots, \theta t_n)$ и $\theta A = \omega(\theta t_1, \ldots, \theta t_n)$. Содержательно, подстановка θ заменяет в каждом выражении e каждое вхождение переменной $x \in X$ на терм θx. Для подстановок естественным образом определяются их композиции: $(\theta_1\theta_2)e = \theta_1(\theta_2 e)$.

Правило $\theta A_0 \leftarrow \theta A_1, \ldots, \theta A_m$, получаемое из правила $A_0 \leftarrow A_1, \ldots, A_m$ с помощью подстаиовки θ, называется *частным случаем* последнего. Если при этом θ является перестановкой множества X, то частный случай правила называется его *вариантом*. Для дальнейшего нам удобно сопоставить программе P бесконечную программу P^*, состоящую из всех частных случаев правил из P, не содержащих переменных (т. е. полученных с помощью подстановок типа θ: $X \to I_\Sigma$).

Произвольное множество основных атомов называется *интерпретацией* (интерпретация Int сопоставляет каждому предикатному символу ω отношение на I_Σ — кортеж $\langle t_1, \ldots, t_n\rangle$ входит в это отношение, если и только если атом $\omega(t_1, \ldots, t_n)$ принадлежит Int). Интерпретация Int называется *моделью программы* P, если для каждого правила $A_0 \leftarrow A_1, \ldots, A_m$ из P^* имеет место

$$\{A_1, \ldots, A_m\} \subseteq \mathrm{Int} \Rightarrow A_0 \in \mathrm{Int}\ ^2). \qquad (1)$$

Для фактов это условие вырождается в требование $A_0 \in \mathrm{Int}$.

У программы может быть много моделей. Обозначим через $\mathrm{Mod}(P)$ класс всех моделей программы P. Нетрудио видеть, что $\mathrm{Int}_P = \bigcap \mathrm{Mod}(P)$ — пересечение всех моделей программы также является моделью P. Модель Int_P называют *главной моделью* программы P.

Пусть $x_1, \ldots, x_k$ — все переменные запроса $\leftarrow C_1, \ldots, C_r$ к программе P и для некоторой подстановки θ имеет место $\{\theta C_1, \ldots, \theta C_r\} \subseteq \mathrm{Int}_P$. Тогда кортеж термов $\langle t_1, \ldots, t_k\rangle$, где $t_i = \theta x_i$, называется *ответом* на запрос. Если в запросе нет

---

[1]) Термии «подстановка» употребляется в разд. 2 совсем в другом смысле.

[2]) В этом разделе символы $\Rightarrow$ и $\Leftrightarrow$ употребляются как сокращения для слов «влечет» и «если и только если». В разд. 2 символ $\Rightarrow$ употребляется в другом смысле.

переменных, то ответом будет «да», если $\{C_1, \ldots, C_r\} \subseteq \mathrm{Int}_P$, и «нет» в противном случае.

Описанная выше семантика называется *«если»-семантикой* логических программ. В этом случае правило $A_0 \leftarrow A_1, \ldots, A_m$ рассматривается как формула логики предикатов [1] $\forall x_1 \ldots \ldots \forall x_l (A_1 \wedge \ldots \wedge A_m \rightarrow A_0)$, эквивалентная формуле $\forall x_1 \ldots \ldots \forall x_l (\neg A_1 \vee \ldots \vee \neg A_m \vee A_0)$ (в этих формулах $x_1, \ldots, x_l$ — все переменные правила). Формулы вида $\neg A_1 \vee \ldots \vee \neg A_m \vee A_0$ $(m \geqslant 0)$ называются *хорновскими дизъюнктами*, и поэтому рассматриваемый в этом разделе стиль программирования называют иногда *хорновским программированием*.

Кроме «если»-семантики логическим программам сопоставляют *«если и только если»-семантику*, при которой интерпретация Int называется моделью программы P, если для каждого основного атома A_0 имеет место

$$A_0 \in \mathrm{Int} \Leftrightarrow \text{ в } P^* \text{ существует правило } A_0 \leftarrow A_1, \ldots, A_m \text{ и } \{A_1, \ldots, A_m\} \subseteq \mathrm{Int}.$$

Очевидно, что каждая модель программы, соответствующая «если и только если»-семантике, является моделью для «если»-семантики. Обратное, вообще говоря неверно. Для нас существенно, что главная модель Int_P является моделью программы для обеих семантик. Различия в семантиках сказываются при некоторых расширениях логических программ — см. ниже разд. 1.3.

Формулы, сопоставленные программе при «если и только если»-семантике, уже не являются хорновскими (так же как и формулы для многих расширений логических программ — см. в разд. 1.3.1 формулы трехзначной логики для конструкций с отрицанием: используемая там связка $\cong$ отличается от «обычной» эквивалентности, выражаемой стандартным образом через дизъюнкцию и отрицание). Поэтому термин «хорновское программирование» к такой ситуации не очень подходит.

Логические программы являются универсальным алгоритмическим средством — можно показать, что любое перечислимое отношение на множестве I_Σ всех основных термов может быть задано логической программой (как отношение в ее главной модели) и что логические программы задают только такие отношения.

Семантика неподвижной точки. Каждое правило $A_0 \leftarrow \leftarrow A_1, \ldots, A_m$ программы P можно рассматривать как

[1] $\forall$ — квантор всеобщности («для всех»), $\rightarrow$ — импликация («влечет»), $\wedge$ — конъюнкция («и»), $\vee$ — дизъюнкция («или»), $\neg$ — отрицание («не»).

m-арную частичную операцию на множестве всех основных атомов: если правило $\theta A_0 \leftarrow \theta A_1, \ldots, \theta A_m$ принадлежит программе P^*, то эта операция применима к кортежу $\theta A_1, \ldots, \theta A_m$ и θA_0 — ее результат. Тем самым на множестве основных атомов задается алгебра (с не всюду определенными и, вообще говоря, многозначными операциями). Операции этой алгебры задают оператор Φ_P, переводящий каждую интерпретацию Int в интерпретацию

$$\Phi_P(\mathrm{Int}) = \{\theta A_0 \mid \text{в } P^* \text{ существует правило } \theta A_0 \leftarrow \theta A_1, \ldots, \theta A_m \text{ и } \{\theta A_1, \ldots, \theta A_m\} \subseteq \mathrm{Int}\}.$$

Можно показать, что множество неподвижных точек этого оператора (т. е. множество таких интерпретаций, что $\Phi_P(\mathrm{Int}) = \mathrm{Int}$) совпадает с множеством моделей программы P для «если и только если»-семантики. Главная модель Int_P программы P — это *наименьшая неподвижная точка* оператора Φ_P, т. е. наименьшая подалгебра описанной выше алгебры. Для этой модели имеет место

$$\mathrm{Int}_P = \bigcup_{n=1}^{\infty} \Phi_P^n(\varnothing). \qquad (2)$$

Эта алгебраическая структура проясняет «устройство» моделей программы. Соотношение (2) говорит о том, что каждый атом главной модели «получается» за конечное число шагов с помощью операций данной алгебры и тем самым принадлежность его к основной модели может быть установлена за конечное число шагов. На этом основана процедурная семантика программы.

Семантика неподвижной точки для логических программ аналогична денотационной семантике для «обычных» программ. Семантика неподвижной точки для логических программ и ее связь с теоретико-модельной семантикой была рассмотрена в работах [Борщев 72] (для так называемой окрестностной версии логического программирования), [ван Эмден 76] и [Апт 82] (для классической версии).

Процедурная (операционная) семантика. Пусть Q — запрос к программе P. Как вычислить все ответы на этот запрос? Правила таких вычислений и образуют *процедурную семантику.* Рассмотрим метод вычислений «сверху-вниз», являющийся основой большинства систем логического программирования.

Унификация. Подстановка θ называется *унификатором выражений* e и e', если $\theta e = \theta e'$. Унификатор θ называют *наиболее общим* для e и e', если для любого другого унификатора θ' этих выражений найдется подстановка θ'', такая, что

$\theta' = \theta''\theta$. Нахождение (наиболее общего) унификатора называется унификацией.[1])

Вычисление. Фиксируется *правило вычисления* (computation rule), определяющее в каждом запросе *выделенный* атом. Пусть Q — запрос вида $\leftarrow C_1, \ldots, C_{i-1}, C_i, C_{i+1}, \ldots, C_r$, в котором выделен атом C_i и $A_0 \leftarrow A_1, \ldots, A_m$ — вариант некоторого правила программы P, такой, что все его переменные отличны от переменных запроса Q. Если θ — наиболее общий унификатор для C_i и A_0, то говорят, что запрос Q' вида

$$\leftarrow \theta C_1, \ldots, \theta C_{i-1}, \theta A_1, \ldots, \theta A_m, \theta C_{i+1}, \ldots, \theta C_r$$

выводим из запроса Q с помощью правила $A_0 \leftarrow A_1, \ldots, A_m$ и подстановки θ.

Получение запроса Q' из Q (когда выбрано то или иное правило программы) — это *элементарный шаг вычисления* в логическом программировании. Он может завершиться неудачей, так как может не существовать унификатора для C_i и A_0. Если производительность обычных машин измеряют количеством элементарных операций в секунду, то производительность систем логического программирования измеряют количеством таких шагов в секунду (LIPS — Logic Inference Per Second).

Пространством вычислений для программы и данного правила вычисления называется множество всех возможных запросов с заданным на нем отношением выводимости. *Вычислением запроса* Q называется путь в таком пространстве, начинающийся с Q, т. е. последовательность $\alpha = Q_1, Q_2, \ldots$, такая, что $Q_1 = Q$ и Q_{i+1} выводим из Q_i. Вычисление может завершиться на запросе Q_n в двух случаях:

1) Q_n — пустой запрос. Тогда вычисление называется *успешным* и с ним связывается подстановка $\theta_\alpha = \theta_{n-1}\theta_{n-2} \ldots \theta_1$, где θ_i — подстановка, с помощью которой запрос Q_{i+1} выводим из запроса Q_i. Если $x_1, \ldots, x_k$ — переменные запроса Q, то кортеж $\langle t_1, \ldots, t_k \rangle$, где $t_i = \theta_\alpha x_i$, называется *ответом, выдаваемым этим вычислением* (если запрос не содержит переменных, то успешное вычисление выдает ответ «да»).

2) Q_n не пусто и из него не выводим ни один запрос. В этом случае вычисление называется *тупиковым*.

---

[1]) Алгоритм унификации линейной сложности описан в [Патерсон 78]. Известны и другие алгоритмы. Вообще унификация, ее обобщения и связанные с нею методы и проблемы играют важную роль в логическом программировании и связанной с ним области автоматического поиска вывода (см. обзор [Зикман 84], а также работу [Нейман 86], посвященную проблеме быстрого поиска в множестве термов).

Рассмотренная процедура вычисления согласуется с теоретико-модельной семантикой. Она *корректна* в том смысле, что при любом правиле вычисления каждый не содержащий переменных частный случай ответа, выдаваемый успешным вычислением, является ответом на этот запрос в теоретико-модельной семантике. В то же время она *полна* — каждый ответ на запрос в теоретико-модельной семантике является частным случаем ответа, выдаваемого некоторым успешным вычислением.

Описанная выше процедурная семантика была предложена для языка Пролог [Колмероэ 73] с уточнениями, которые будут даны в разд. 1.2, и формально описана в [Ковальский 74] как частный случай метода резолюций [Робинсон 65] (так называемая SLD-резолюция — разновидность линейной резолюции, предложенной в работе [Ковальский 71]). Использование в процедурной семантике метода резолюций объясняет употребление термина *резолюционное программирование* и связывает логическое программирование с давно разрабатываемой областью автоматической дедукции или автоматического доказательства теорем (см. [Чень 73, Воронков 86г, 87а, Стикель 86] и разд. 3.2).

1.2. Пролог и другие языки логического программирования

Первым языком логического программирования был Пролог, разработанный в начале 70-х годов А. Колмероэ [Колмероэ 73]. По сути дела большинство языков логического программирования, особенно реально используемых, это те или иные модификации и расширения Пролога.

Мы начнем с того, что отличает Пролог от описанных в разд. 1.1 механизмов, а затем кратко рассмотрим другие языки и их основные особенности. Предлагавшиеся в этих языках (а также в теоретических работах) расширения основных механизмов и средств логического программирования будут рассмотрены в разд. 1.3.

1.2.1. Пролог и его особенности

Стратегии вычисления. Описанное выше пространство вычислений содержит пути всех возможных вычислений (для данного правила вычисления). Это пространство естественно представить в виде графа, вершинам которого сопоставлены запросы, а дуги соответствуют отношению выводимости, причем каждой дуге сопоставлены примененное правило и унифицирующая подстановка. Каждому запросу соответствует часть пространства вычислений, содержащая все пути (вы-

числения), ведущие из вершины этого запроса. Эти пути изображают обычно в виде дерева, которое называется *деревом вычислений.*

Правило вычислений Пролога выделяет самый левый атом запроса. Упорядочиваются «сверху-вниз» и правила программы. Пролог фиксирует некоторую стратегию последовательного обхода дерева вычислений. На каждом шаге вычисления к самому левому атому запроса применяется первое (из еще не использованных) правило программы для данного предикатного символа. При этом в стеке запоминаются возможные разветвления (следующее правило для того же предикатного символа). Если вычисление заходит в тупик, происходит возврат к месту последнего разветвления — «бектрекинг» (backtracking). При этом отменяются результаты всех подстановок, сделанных после последнего разветвления. Такой же возврат осуществляется и после завершения успешного вычисления — для получения других ответов. Интерпретатор заканчивает работу, когда стек разветвлений пуст.

Основным достоинством этой стратегии является простота ее реализации (на фоннеймановских машинах), недостатком — большой перебор для некоторых программ и запросов. Кроме того, если дерево вычислений содержит бесконечную ветвь, то, попав на нее, интерпретатор с нее не выберется, и поэтому правильные ответы, лежащие «правее» этой ветви на дереве вычисления, не будут получены (т. е. эта стратегия не полна с точки зрения теоретико-модельной семантики).

Оператор усечения. Одним из средств для преодоления указанных выше недостатков является оператор усечения (cut). Рассмотрим пример [1]) (из работы [Мосс 86]):

$a(x) \leftarrow b(x),\ !,\ c(x)$ $\quad b(f) \leftarrow$

$a(x) \leftarrow d(x)$ $\quad c(x) \leftarrow$

$b(e) \leftarrow$ $\quad d(g) \leftarrow$

Знак ! обозначает встроенный оператор усечения, смысл которого заключается в изменении стандартного порядка вычислений. При его «выполнении» (когда он становится самым левым в запросе) забываются (выбрасываются из стека) все разветвления, запомненные с момента обращения к правилу, содержащему этот оператор, и в дальнейшем осуществляется возврат к предыдущему разветвлению. Тем самым «усекается», «обрезается» (и иногда весьма существенно) дерево

[1]) Этот пример, как и большинство других, заимствован из цитированных работ. При этом примеры иногда упрощались и слегка модифицировались — мы стремились к единообразию в синтаксисе.

разветвлений. Так, в данном примере без оператора «|» на запрос $\leftarrow a(z)$ было бы выдано три ответа: $z = e$, f, g. Из-за оператора «|» программа выдает единственный ответ $z = e$, так как она не будет уже возвращаться к развилкам, возникшим до обращения к «|» (при обработке атомов $a(z)$ и $b(z)$).

Оператор усечения — остроумное и довольно мощное средство. Пользуясь им, программист может составлять довольно эффективные программы. Однако этот оператор в какой-то степени подрывает основную идею логического программирования. Программа лишается прозрачной логической семантики, остается лишь процедурная семантика, соответствующая данной стратегии обхода дерева вычислений. Кроме «теоретических» обвинений в неизящности такого решения к нему можно предъявлять и «практические» претензии. Ведь возможны и другие стратегии обхода дерева, в частности параллельные вычисления, и в идеале хотелось бы, чтобы программа была инвариантна относительно стратегии вычислений, а оптимальная стратегия подбиралась бы интерпретатором (может быть, «по подсказке» программиста). Предлагавшиеся решения этой проблемы обсуждаются в разд. 1.3.

Отрицание. В правилах Пролога и в запросах перед некоторыми атомами может стоять символ not, понимаемый, грубо говоря, как логическое отрицание. Точной логической семантики у таких конструкций не было. При их вычислении используется правило «отрицание как неудача» (negation as failure). Так, если запрос начинается с выражения not A, где A — основной атом, то интерпретатор пытается вычислить запрос A. Если все пути вычисления этого запроса заходят в тупик, то выражение not A считается доказанным и интерпретатор переходит к следующему выражению исходного запроса. Если же хотя бы один из путей вычисления A успешен, то вычисление not A считается тупиком и происходит возврат. Как всегда, есть и третья возможность — «зацикливание» (попадание на бесконечную ветвь дерева вычислений для A).

Встроенные отношения. Обычно в реализованных версиях Пролога кроме отношений, определяемых программистом, т. е. задаваемых правилами программы, применяются встроенные (системные) отношения. Соответствующие им атомы могут использоваться в правилах или запросах и понимаются интерпретатором, хотя программа не содержит правил, их определяющих. Обычно, это отношения, которые либо невозможно определить стандартными средствами логического

программирования, либо по соображениям удобства или эффективности желательно предоставить программисту в готовом виде. Это прежде всего средства, связанные с арифметическими действиями над числами, средства ввода-вывода, изменения программы (добавление и исключение правил)[1]) и т. д. Для встроенных средств обычно описывается лишь их процедурная семантика.

«Чистый» и «нечистый» Пролог. Средства Пролога делят на «чистые» и «нечистые» (по аналогии с чистым Лиспом). К чистому Прологу относят обычно те его конструкции, которые имеют четкую теоретико-модельную семантику. Стремление к «чистоте» понятно. Именно потому, что основные конструкции языков логического программирования столь прозрачны теоретически (в отличие от традиционных языков программирования), хотелось бы, чтобы и все конструкции были таковыми.

В то же время имеющиеся «нечистые» конструкции говорят лишь о том, что практика логического программирования обгоняет теорию. Нужно сказать, что в последнее время появилось много работ, предлагающих точную логическую семантику для конструкций, которые раньше такого описания не имели[2]). Некоторые направления таких работ обсуждаются ниже в разд. 1.3.

1.2.2. Другие языки логического программирования

Таких языков уже очень много — пять представлены в статьях настоящего сборника. Мы кратко остановимся лишь на некоторых. Очень интересен Пролог-II [Колмероэ 83]. Отметим лишь некоторые его черты. Не различаются функциональные и предикатные символы — одни и те же символы могут использоваться для построения объектов и отношений между этими объектами. Заметим, что такого различия не делалось и в работе [Борщев 76], где описывались *клубные системы.* Частный случай клубных систем — так называемая *однородная версия* логического программирования [Борщев 86б] — особенно близка в этом смысле к конструкциям Пролога-II. В Прологе-II можно работать с так называемыми *бесконечными деревьями* (бесконечными термами). Более

---

[1]) Заметим, что в Прологе программа иногда называется «базой данных» (программа — то, что хранится), в отличие от нашего сравнения с базой данных главной модели Int_P, задаваемой программой.

[2]) Так, в [Мииц 86] строится логическое исчисление для чистого Пролога с оператором усечения.

подробно такие конструкции рассматриваются в работе [Колмероэ 82а] (см. также [Ллойд 84а]).

Язык IC-PROLOG [Кларк 82а] интересен прежде всего тем, что его авторы стремились пользоваться только чистыми конструкциями. Отметим также предоставляемые программисту средства управления ходом вычислений.

Разработанный в Венгрии язык МПролог [Домёлки 83] — одна из первых промышленных систем логического программирования. Его основной особенностью является *модульность*. Заметим, что существуют реализации МПролога на машинах серий ЕС и СМ. Надо сказать, что логическое программирование очень популярно в Венгрии. Укажем еще три «венгерских» языка. Это Т-Пролог, вариант МПролога для моделирования развивающихся во времени процессов [Футо 83], а также LOBO [Шётц 82] и LQF [Гергей 84]. Два последних очень близки, их особенностью является использование ограниченных кванторов.

Ряд других языков также использует более мощные, чем обычный Пролог, логические средства — это N-PROLOG [Габбай 84, 85], NU-PROLOG [Нэш 86а]. Некоторые языки комбинируют средства логического и функционального программирования. Они упоминаются в разд. 2.3. Параллельные языки логического программирования рассматриваются в разд. 1.3.3.

1.3. Основные проблемы и направления развития

Логическое программирование — бурно развивающаяся область. Ведутся теоретические исследования его оснований, поиск и осмысливание выразительных средств, добавляемых к описанным выше конструкциям, образующим ядро логического программирования. Одна из главных проблем — поиск эффективных методов организации вычислений для логических программ. Важное направление здесь — методы параллельных вычислений, языки и средства, позволяющие организовать такие вычисления. Существует много реализаций языков логического программирования как последовательных, так и параллельных. Естественно, что методы, используемые в таких реализациях, тесно связаны с перечисленными выше направлениями исследований (влияние тут взаимное; как уже указывалось, практика нередко опережает теорию). Много внимания уделяется архитектуре машин, приспособленных для реализации логических программ.

Ниже рассматриваются некоторые из этих направлений, Связь логического программирования с другими не фоннеймановскими стилями программирования (функциональным, эквациональным и т. д.) рассматривается в разд. 2 и 3.

1.3.1. Поиск выразительных средств и изучение оснований

Семантика конструкций с отрицанием. Как уже указывалось, конструкции с отрицанием с самого начала использовались в Прологе. Первой работой по описанию семантики таких конструкций была статья Кларка [Кларк 78] (см. также [Ллойд 84а]). В ней рассматриваются программы с правилами вида

$$A \leftarrow L_1, \ldots, L_m, \tag{3}$$

где A — атом, а $L_1, \ldots, L_m$ — *литералы* (называемые также *литерами*), т. е. либо атомы, либо выражения типа $\neg C$, где C — атом. Пусть A имеет вид $\omega(t_1, \ldots, t_m)$, а $x_1, \ldots, x_k$ — все переменные правила (3), а $y_1, \ldots, y_n$ — переменные, не встречающиеся в программе. Тогда этому правилу сопоставляется формула

$$\varphi = \exists x_1 \ldots \exists x_k((y_1 = t_1) \wedge \ldots \wedge (y_n = t_n) \wedge L_1 \wedge \ldots \wedge L_m). \tag{4}$$

Рассматриваются все правила программы для символа ω и им сопоставляется формула («$\sim$» — знак логической эквивалентности)

$$\forall y_1 \ldots \forall y_n (\omega(y_1, \ldots, y_n) \sim (\varphi_1 \vee \ldots \vee \varphi_s)), \tag{5}$$

где $\varphi_1, \ldots, \varphi_s$ формулы типа (4), сопоставленные этим правилам. Если для символа ω, содержащегося в правых частях каких-нибудь правил, в программе нет ни одного правила, то ему сопоставляется формула $\forall y_1 \ldots y_n \neg \omega(y_1, \ldots, y_n)$. Нетрудно видеть, что для обычных программ (без отрицания) эти формулы в точности соответствуют рассмотренной выше (в разд. 1.1) «если и только если»-семантике. К этим формулам добавляются аксиомы равенства (см. разд. 2.3), и полученная таким образом совокупность формул для программы P называется *замыканием* $P(\mathrm{Comp}(P))$.

Основной результат заключается в следующем. Если для запроса $\leftarrow C_1, \ldots, C_k$ существует успешное вычисление (в ходе которого применялось правило «отрицание как неудача») и θ — подстановка, соответствующая этому вычислению, то формула $\theta C_1 \wedge \ldots \wedge \theta C_k$ является логическим следствием $\mathrm{Comp}(P)$. Тем самым доказывается корректность вычислений с помощью правила «отрицание как неудача».

Работа Кларка, а также последовавшие за ней работы [Апт 82, Шепердсон 84, 85; Ллойд 84а, б] внесли некоторую ясность, но все же оставляли ощущение неудовлетворенности. Самым существенным было то, что для программ с отрицанием не было семантики неподвижной точки, связывающей вроде бы построенную теоретико-модельную семантику с

процедурной семантикой. И в этом плане значительным достижением стала работа [Фиттинг 85].

Фиттинг утверждал, что обычная логика не дает адекватного описания логических программ [1]) и предложил использовать для этой цели трехзначную логику Клини [Клини 52] с истинностными значениями И (истина), Л (ложь), Н (неопределенность). (Трехзначная логика для описания семантики логических программ предлагалась также в работе [Микрофт 84].)

Правило программы, как и раньше, имеет вид

$$A \leftarrow B_1, \ldots, B_m, \qquad (6)$$

но здесь B_i — произвольная бескванторная формула, не содержащая равенств (A, как обычно, атом).

Определяются «трехзначные» интерпретации. Как и в двузначной логике, отношение — это отображение декартова произведения (соответствующей арности) в множество истинностных значений. Если там его можно отождествить с множеством кортежей, отображенных в И, и тем самым интерпретацию — с множеством основных атомов, то в трехзначной логике отношение можно отождествлять с двумя множествами кортежей, отображенных соответственно в И и Л, а интерпретацию — с множеством основных литералов.

Основной литерал B истинен в интерпретации Int, если и только если $B \in \mathrm{Int}$. Значения остальных формул определяются по таблицам истинности:

A	B	¬A	A ∨ B	A ∧ B
Н	И	Н	И	Н
Н	Н	Н	Н	Н
Н	Л	Н	Н	Л

(для «нормальных» значений — И и Л — связки определены стандартно, а $\vee$ и $\wedge$, как обычно, коммутативны).

Как и в разд. 1.1, сопоставим программе P программу P^*, состоящую из всех частных случаев правил из P, не содержащих переменных.

Интерпретация Int называется моделью программы P, если

[1]) Заметим, что эта неадекватность проявляется главным образом для программ с отрицанием. Для программ (и запросов) без отрицания разница между семантикой Фиттинга и рассмотренной выше «если и только если»-семантикой не столь существенна.

1) для каждого атома A:

$A \in \mathrm{Int} \Leftrightarrow$ в P^* существует правило $A \leftarrow B_1, \ldots, B_m$ и формула $B_1 \wedge \ldots \wedge B_m$ истинна в Int;

2) для каждого литерала вида $\neg A$:

$\neg A \in \mathrm{Int} \Leftrightarrow$ для каждого правила из P^* вида $A \leftarrow B_1, \ldots, B_m$ формула $B_1 \wedge \ldots \wedge B_m$ ложна в Int [1]).

Первый пункт этого определения аналогичен «если и только если»-семантике «обычных» программ. Вся суть в пункте 2): «отрицательный» литерал $\neg A$ принадлежит модели, если каждое правило с заголовком A из P^* его «отвергает». Этот пункт в точности соответствует правилу «отрицание как неудача».

Приведенное определение позволяет построить оператор Φ_P на множестве интерпретаций:

1) $A \in \Phi_P(\mathrm{Int}) \Leftrightarrow$ в P^* существует правило $A \leftarrow B_1, \ldots, B_m$ и формула $B_1 \wedge \ldots \wedge B_m$ истинна в Int;

2) $\neg A \in \Phi_P(\mathrm{Int}) \Leftrightarrow$ для каждого правила из P^* вида $A_1 \leftarrow B_1, \ldots, B_m$ формула $B_1 \wedge \ldots \wedge B_m$ ложна в Int.

Нетрудно видеть, что каждая неподвижная точка Φ_P является моделью P, и наоборот. Главная модель Int_P программы P — наименьшая неподвижная точка — является пределом последовательности $\Phi_P \uparrow^{\alpha}(\varnothing)$ (здесь α — ординалы, $\Phi \uparrow^{\alpha+1}(\varnothing) = \Phi(\Phi \uparrow^{\alpha}(\varnothing))$ и $\Phi \uparrow^{\lambda}(\varnothing) = \bigcup\limits_{\alpha<\lambda} \Phi \uparrow^{\alpha}(\varnothing)$ для предельных ординалов). Из пункта 2) следует, что такого рода программы в каком-то смысле чересчур мощны — они могут задавать не только перечислимые отношения. Программы, задающие перечислимые отношения (точнее, перечислимые множества основных литералов), называются *допустимыми*. Очевидно, что только такие программы оправданны. Однако не ясно, существует ли процедура, устанавливающая по тексту программы, является ли она допустимой.

Рассмотренная семантика логических программ сравнивается в цитируемой работе с семантикой, предложенной Крипке для анализа парадоксов [Крипке 75, Фиттинг 86]. Показано, что каждая программа с правилами типа (6) может быть преобразована в эквивалентную ей программу с правилами типа (3). В работе [Ллойд 84б] рассматривались

[1]) В указанной статье Фиттинга программе сопоставлены формулы типа (5), в которых вместо эквивалентности $\sim$ употреблена связка $\cong$: замкнутая формула $\varphi_1 \cong \varphi_2$ истинна, если истинностные значения φ_1 и φ_2 совпадают. Тогда интерпретация Int является моделью программы P, если все формулы вида (5), сопоставленные P, истинны для Int. Нетрудно видеть, что приведенные определения эквивалентны.

правила типа $A \leftarrow W$, где A — атом, а W — произвольная формула языка первого порядка, и приводились преобразования таких программ в программы из правил типа (3). Эти преобразования корректны и для семантики Фиттинга.

В целом работа Фиттинга существенным образом прояснила механизмы, связанные с использованием отрицания. Однако остается проблема исследования методов вычисления, являющихся полными для допустимых программ (правило «отрицание как неудача» не обеспечивает полноты — оно «не работает» на литералах, содержащих переменные). Не ясно, существуют ли достаточно эффективные универсальные методы такого типа.

Отметим еще ряд работ, посвященных использованию отрицания: [Габбай 86, Фланнаган 86, Ван Ту Ле 85а, б, Нэш 86а, б, Гёбел 86].

Метасредства и параметрические конструкции. В работах [Ковальский 82, Боуэн 82, 85] для повышения выразительности программ предлагается использовать средства метаязыка. Основная конструкция, которая при этом рассматривается, — это метапредикат Demo. Интуитивный смысл записи Demo(P, Q, X) — «X — ответ, выдаваемый программой P на запрос Q». По сути дела символ Demo именует отношение, задаваемое интерпретатором логических программ на программах, запросах и ответах. Этот «универсальный» предикат нетрудно описать с помощью логической метапрограммы, играющей ту же роль, какую играет универсальная машина Тьюринга (или универсальный алгоритм Маркова). Для этого нужно закодировать некоторым образом (в виде термов) программы, запросы и ответы. Схема такой метапрограммы приводится в указанных работах.

Такого рода метасредства комбинируют с обычными средствами. Можно, например, задавать «параметрические модули» типа:

M: Maplist(nil, nil) $\leftarrow$
Maplist($x1 . x2, y1 . y2$) $\leftarrow F(x1, y1)$, Maplist($x2, y2$)
$F1$: $F(x, y) \leftarrow$ Plus($x, 2, y$)
$F2$: $F(x, y) \leftarrow$ Times($x, 2, y$)

В этом примере модуль M задает отношение Maplist ($l1$, $l2$) на списках: каждый элемент списка $l2$ есть результат применения функции F к соответствующему элементу списка $l1$. Эта функция, представляемая отношением $F(x, y)$, задается в двух вариантах: модулями $F1$ и $F2$; первый прибавляет 2 к x, второй умножает x на 2. Теперь можно задать запрос

$\leftarrow$ Demo($M + F2$, Maplist(2 . 1 . nil), y),

в котором Demo «собирает» программу из модулей M и $F2$. Ответом будет «удвоенный» список 4.2. nil. А правило

$$\text{Double}(x, y) \leftarrow \text{Demo}(M + F2, \text{Maplist}(x, y), y)$$

задает отношение Double между списком x натуральных чисел и списком y удвоенных элементов списка x.

Другое предлагавшееся решение для параметризации — программы «высших типов» (см., например, [Галлэр 83, Уоррен 82а]):

$$\text{Maplist}(\text{nil}, \text{nil}, F) \leftarrow$$
$$\text{Maplist}(x1 . x2, y1 . y2, F) \leftarrow F(x1, y1), \text{Maplist}(x2, y2, F)$$

Здесь F — предикатная переменная, являющаяся параметром этой программы. Теоретико-модельная и процедурная семантики таких программ точно не описывались. В цитированной выше работе Уоррена предлагалось рассматривать такие конструкции, как «синтаксический сахар», для обычных программ типа

$$\text{Maplist}(\text{nil}, \text{nil}, F) \leftarrow$$
$$\text{Maplist}(x1 . x2, y1 . y2, F) \leftarrow$$
$$\leftarrow \text{apply}(F, x1, y1), \text{Maplist}(x2, y2, F)$$

в которых F — уже обычная переменная. Однако и в этом случае возникали некоторые трудности.

В работах [Борщев 85, 86б] для этой цели вводятся конструкции несколько иного вида:

$$\text{Maplist}[F](\text{nil}, \text{nil}) \leftarrow$$
$$\text{Maplist}[F](x1 . x2, y1 . y2) \leftarrow F(x1, y1), \text{Maplist}[F](x2, y2)$$
$$\text{Plus2}(x, y) \leftarrow \text{Plus}(x, 2, y)$$
$$\text{Twice}(x, y) \leftarrow \text{Times}(x, 2, y)$$
$$\text{Double}(x, y) \leftarrow \text{Maplist}[\text{Twice}](x, y)$$

Идея состоит в использовании «сложных» предикатных символов типа Maplist[F], из которых путем подстановки можно получать символы Maplist[Plus 2], Maplist[Twice] и т. д.

Основным результатом этих работ является построение простой теоретико-модельной семантики для таких конструкций, естественным образом связанной с их процедурной семантикой.

Предлагаемые конструкции легко могут быть перекодированы в обычные логические программы (подобная кодировка используется также в работе [Колмероэ 82а]).

1.3.2. Управление ходом вычислений

Отделение логики программы от управления ходом вычислений — одна из основных идей логического программирования. Она ярко выражена в «уравнении» Ковальского: Алгоритм = Логика + Управление [Ковальский 79а]. Рассматриваемые здесь средства управления вычислениями предлагаются в основном для последовательных реализаций, хотя некоторые идеи применяются и при параллельных вычислениях.

Начнем с уже упоминавшегося в разд. 1.2 оператора усечения cut, применение которого как раз нарушает только что продекларированный принцип разделения логики и управления. В работе [Мосс 86] указывается, что, несмотря на кажущуюся простоту процедурной семантики для этого оператора, в разных системах он реализован по-разному. Тем самым подчеркивается необходимость декларативного описания такого рода конструкций и в этой работе предлагается такое описание.

В работе [Дебрей 86] предлагается вместо оператора cut использовать операторы savecp и cutto, сохраняя стратегию вычислений Пролога. Так, программу

$p(x) \leftarrow q(y), !, r(z)$
$p(x) \leftarrow s(u)$

можно заменить программой

$p(x) \leftarrow \mathrm{savecp}(w), p1(x, w)$
$p1(x, w) \leftarrow q(y), \mathrm{cutto}(w), r(z)$
$p1(x, v) \leftarrow s(u)$

В процессе вычисления по последней программе при обработке «атома» savecp(w) отмечается место в стеке разветвлений (с помощью переменной w), и если затем интерпретатор приходит к обработке атома cutto(w), все запомненные после данной отметки разветвления выбрасываются из стека. Тем самым выполняются все функции оператора cut. Разница заключается в том, что оператор savecp, делающий начальную пометку в стеке, может помещаться в произвольное место программы (в то время как для оператора cut место соответствующей отметки фиксировано). Но главная идея этой работы состоит в том, что соответствующие операторы вставляются не программистом, а автоматически, в процессе предварительной обработки программы. При этом выделяются «функциональные куски» программы, для которых все разветвления, запомненные в процессе их выполнения, зашли бы в тупик, и эти куски «обрамляются» операторами savecp

и cutto. В работе приводятся алгоритмы выделения функциональных кусков программы. Заметим, что эта идея сокращения перебора аналогична идее сокращения перебора при синтаксическом анализе, предлагавшейся, например, в работе [Борщев 67]. Вообще, неоднократно отмечалась глубокая аналогия между грамматиками и методами синтаксического анализа, с одной стороны, и логическими программами и методами вычислений для них, с другой (она проявляется также в методах «снизу вверх» и «сверху вниз» для вычислений и анализа).

В работе [Нэш 86а] рассматривается правило вычисления, применяемое в языке MU-PROLOG и позволяющее повысить эффективность вычислений, а также избегать «зацикливаний». Основная идея — откладывать обработку атомов запроса в тех случаях, когда это бессмысленно или может привести к большим затратам времени. Например, нет смысла обрабатывать атом App(x, t, y) (см. в разд. 1.1 соответствующую программу), до тех пор пока переменная x (или y) не получит значения. Поэтому обработка таких атомов откладывается, а переменные x и y помечаются. В общем случае атомы запроса делятся на обрабатываемые и отложенные[1]). Так, пусть в запросе $\leftarrow P, Q, R \,\&\, S, T$ это атомы P, Q, R и соответственно S, T, разделенные знаком &. Для обработки всегда выбирается самый левый атом (из обрабатываемых), в данном случае — P. Пусть по правилу вычисления этот атом нужно отложить, тогда он переносится в отложенную часть, и новый запрос имеет вид $\leftarrow Q, R \,\&\, S, T, P$. Пусть теперь после обработки атома Q получены атомы U и V и при этом присвоены значения отмеченным переменным в атомах S и P, после чего они готовы для обработки. Тогда они переносятся в обрабатываемую часть запроса, и новый запрос имеет вид $\leftarrow S, P, U, V, R \,\&\, T$.

Для реализации такого правила вычисления предикатным символам сопоставляются специальные декларации (они используются и в других языках, в частности в упоминавшихся выше языках IC-PROLOG и МПролог). Для «рекурсивных» символов (типа App) такие декларации указывают, при каком наборе аргументов имеет смысл обрабатывать данный атом. Для правил типа «фактов в базе данных» применяются другие декларации, учитывающие количество однородных фактов и тем самым позволяющие выбирать в запросе

---

[1]) Эта же идея используется и при параллельных вычислениях, рассматриваемых ниже. Там описываются различные типы аннотаций к программе, пользуясь которыми, можно принять решение — готов атом к обработке или нет.

«выгодные» атомы и откладывать «невыгодные» и тем самым ускорять вычисления.

В цитируемой работе предлагаются алгоритмы, которые вычисляют соответствующие декларации по «обычной» программе, составленной программистом.

Средства управления ходом вычислений рассматриваются также в работах [Колмероэ 83, Кан 84а, Васак 85].

1.3.3. Параллельные вычисления

Одним из основных достоинств логического программирования является то, что по сути своей оно параллельно. Программа, вообще говоря, не определяет порядка вычислений. Поэтому при обработке запроса можно, во-первых, параллельно обрабатывать разные его атомы — так называемый И-*параллелизм* (AND-parallelism), и во-вторых, обрабатывая данный атом, параллельно применять к нему все правила программы с данным предикатным символом (ИЛИ-*параллелизм* — OR-parallelism). Предложено несколько языков, позволяющих организовать параллельные процессы вычислений. Мы рассмотрим три таких языка — PARLOG [Кларк 84а], Concurrent Prolog [Шапиро 83] и GHC [Уеда 86а]. Языки эти очень похожи, все они основаны на идеях, предложенных в работе [Кларк 81].

Программа в таких языках — это конечное множество охраняемых правил (guarded clauses). *Охраняемое правило* имеет вид:

$$H \leftarrow G_1, \ldots, G_n \mid B_1, \ldots, B_m \quad n, m \geqslant 0.$$

Атомы $G_1, \ldots, G_n$ называются *охраной* (guard), а $B_1, \ldots, B_m$ — телом правила. Черта «|» называется *оператором привязки* (commitment operator или commit).

Провозглашается, что теоретико-модельная семантика таких языков не отличается от такого рода семантики для «обычных» языков логического программирования. Так, в рамках «если»-семантики такое правило утверждает: H истинно, если как $G_1, \ldots, G_n$, так и $B_1, \ldots, B_n$ истинны.

Опишем в общих чертах процедурную семантику таких программ. Вводится понятие *процесса*. Рассматриваются два типа процессов — И-процессы и ИЛИ-процессы. Пусть $\leftarrow C_1, \ldots, C_r$ — запрос. Тогда для каждого подзапроса C_i создается И-процесс, и все эти процессы выполняются параллельно. Когда все эти процессы успешно закончатся, закончится вычисление исходного запроса. Пусть ω — предикатный символ подзапроса C_i и в программе существует несколько правил для этого предикатного символа. Тогда для каждого

такого правила создается ИЛИ-процесс. Каждый такой процесс как бы «примеряет» соответствующее правило к подзапросу C_i — производится унификация и проверка выполнимости всех атомов охранной части правила, играющих роль условий на применимость правила. Если все эти условия выполнены и унификация возможна, то ИЛИ-процесс успешно завершается. Если один из ИЛИ-процессов успешно завершен, соответствующее правило «привязывается» к подзапросу C_i, а остальные ИЛИ-процессы прерываются. Для каждого B_j из тела «привязанного» правила (модифицированного унифицирующей подстановкой) заводится новый И-процесс.

На унификацию, осуществляемую ИЛИ-процессом, накладываются дополнительные ограничения, связанные с так называемыми декларациями видов (mode declarations). Суть этих ограничений та же, что и у аннотаций, используемых в языках IC-PROLOG или MU-PROLOG (см. выше разд. 1.2), — унификация атома запроса с заголовком правила возможна только в том случае, когда соответствующие аргументы атома запроса «готовы для вычисления» (например, при вычислении функции места аргументов не должны быть заняты переменными). Соответствующие декларации различны в различных языках.

Так, например, на языке Парлог программа для «слияния» двух списков X и Y в третий список Z имеет вид:

```
mode merge(?,?,^)
merge(A.X,Y,A.Z) ← true | merge(X,Y,Z)
merge(X,A.Y,A.Z) ← true | merge(X,Y,Z)
merge(nil,Y,Y) ← true | true
merge(X,nil X) ← true | true
```

Здесь true обозначает «пустую» охранную часть или пустое тело соответствующих правил. Декларация mode merge (?,?,^) указывает, что два первых аргумента в атомах типа merge являются входными, а третий — выходным. Если в запросе на месте входного аргумента стоит переменная, она может унифицироваться только с переменной, унификация с другими термами запрещается. Унификация выходного аргумента проводится только после того, как произошла привязка правила.

Аналогичная программа на языке Concurrent Prolog имеет вид:

```
merge(A.X,Y,A.Z) ← true | merge(X?,Y,Z)
merge(X,A.Y,A.Z) ← true |merge(X,Y?,Z)
merge(nil,Y,Y) ← true | true
merge(X,nil,X) ← true | true
```

Здесь пометка «?» объявляет соответствующий аргумент входным (read-only variable в терминологии Concurrent Prolog) не заранее, как в языке Парлог, а только после выполнения соответствующего правила, но, как и там, переменная на «входном» месте запроса может унифицироваться только с переменной.

В языке GHC (Guard Horn Clauses — охраняемые хорновские дизъюнкты), принятом сейчас в качестве основного языка в японском проекте машин пятого поколения, деклараций видов нет. Соответствующая программа имеет вид:

```
merge(A.X, Y, Oz) ← true | Oz = A.Z, merge(X, Y, Z)
merge(X, A.Y, Oz) ← true | Oz = A.Z, merge(X, Y, Z)
merge(nil, Y, Oz) ← true | Oz = Y
merge(X, nil, Oz) ← true | Oz = X
```

Здесь каждая переменная атома запроса может быть унифицирована только с переменной, стоящей на соответствующем месте в заголовке правила.

Для иллюстрации приведем еще один пример программы на языке GHC.

```
oneDigitCounter(nil, Os, State) ← true | Os = nil
oneDigitCounter(up.Is, Os, State) ← State ⩽ 8 |
        NewState := State + 1, oneDigitCounter(Is, Os,
        NewState)
oneDigitCounter(up.Is, Os, State) ← State = 9 |
        Os = up.NewOs, oneDigitCounter(Is, NewOs, 0)
oneDigitCounter(clear.Is, Os, State) ← true |
        Os = clear.01s, oneDigitCounter(Is, 01s, 0)
twoDigitCounter(Is, Os) ← true |
        oneDigitCounter(Is, 01s, 0), oneDigitCounter
        (01s, Os, 0)
```

Эта программа определяет отношение twoDigitCounter (Is, Os) (двухразрядный счетчик) между произвольным «входным» списком Is (последовательностью «команд» up и clear) и «выходным» списком Os (последовательностью таких же команд): число команд clear в выходной последовательности должно быть равно числу этих команд во входной последовательности, и каждой сотне команд up между командами clear на входе не должна соответствовать одна команда на выходе (перенос в следующий разряд).

Это отношение определяется с помощью отношения «одноразрядный счетчик» — oneDigitCounter (Is, Os, State), в третьем аргументе которого (State) подсчитывается (по модулю 10) число команд up, не прерываемых командой clear, во входной последовательности Is. Второй аргумент — выходная по-

следовательность Os — это число «переносов» в следующий разряд (каждая команда up соответствует десятку таких команд в Is, не прерываемых командой clear).

Программа устроена так, что при обработке подзапроса с предикатным символом oneDigitCounter успешно завершиться может только один из четырех ИЛИ-процессов. И-процессы, порождаемые телом последнего правила, работают, вообще говоря, параллельно, хотя второй из этих процессов время от времени «зависает», ожидая очередное значение аргумента [1]).

Заметим, что для параллельных языков логического программирования связь между теоретико-модельной и процедурной семантикой устанавливается не слишком четко, да и эта процедуриая семантика не привязывается к каким-либо абстрактным машинам параллельного типа.

Из работ, посвященных параллельным языкам логического программирования, отметим обзорный доклад [Такеути 86], статьи [Уеда 86б, Перейра 86, Кащук 84, Стерлинг 86, Хеллерштейн 86, Кодиш 86, Кларк 85, Визе 82, Клещев 80, 81], а также работу [Фучи 86], посвященную роли таких языков в японском проекте машин пятого поколения. Интересная работа [ван Эмден 82] связывает идеи логического программирования с машинами потоков данных (data flow).

1.3.4. Реализации и машинные архитектуры

Проблемы реализации и создания адекватной машинной архитектуры — безусловно, важнейшие направления работ в области логического программирования. Этим темам посвящена большая литература, но, как часто бывает при описании реальных разработок, важные детали остаются «за бортом» публикаций. Авторы обзора не считают себя специалистами в этой области и потому ограничиваются здесь только грубой «разметкой территории». Концентрированное изложение вопросов реализации можно найти в книгах [Реализации 84, Пролог 88, Хоггер 84].

Первый интерпретатор Пролога был разработан в Марселе группой А. Колмероэ [Баттани 83]. Значительным шагом вперед в смысле эффективности стала «эдинбургская» реализация Д. Уоррена [Уоррен 77]. Принципы этой реали-

[1]) Отметим аналогию между использованием деклараций видов, приводящих к зависанию некоторых процессов в параллельных языках (либо к тому, что обработка некоторых подзапросов откладывается в последовательных языках типа IC-PROLOG или MU-PROLOG, и идеей смешанных вычислений [Ершов А. 78, 80] (см. по этому поводу работу [Степанов 81а, б]).

зации остаются во многих отношениях образцом и в настоящее время. В реализациях Пролога очень важную роль играют механизмы управления памятью, им посвящена работа [Бранохе 82].

Публикации об отечественных реализациях Пролога весьма скудны [Белов 86, Кондратьев 86, Пролог-СМ 85]. Зарубежные работы хорошо представлены в сборнике [Реализации 84]. Статья [ван Эмден 84] содержит четкое описание механизма интерпретации Пролог-программы. Работа [Фогельхольм 84] посвящена реализации Пролога с помощью Лиспа, а работа [Кан 84а, б] — реализации Пролога на Лисп-машине. В реализации, описанной в [Накамура 84], вместо используемых обычно стеков применяется хэширование (метод расстановки).

Другие работы упомянутого сборника посвящены отдельным проблемам реализации. Статья [Перейра 84] обсуждает методы управления ходом вычислений. Работы [Бранохе 84а, Кокс 84] посвящены уточненным методам бектрекинга (intelligent backtracking), статьи [Хариди 84, Фильгуэрос 84] — способам работы с упоминавшимися в разд. 1.2 бесконечными деревьями, а статья [Бранохе 84б] — сборке мусора. Проблемы реализации параллельных систем рассматриваются в статьях [Монтейро 84, Визе 84].

Немало работ ([Уоррен 83, Тик 84, Боуэн 83, Курсаве 86] и др.) связано с построением так называемых абстрактных Пролог-машин. При реализации исходная программа переводится в «коды» такой машины (этап компиляции), а сама машина может быть реализована как программно, так и аппаратно. Отметим, что в процессе такой компиляции иногда используются идеи и методы смешанных вычислений [Ершов 78, 80; Венкен 84, Курсаве 86].

Разрабатываются абстрактные машины и для параллельных вычислений [Леви 86а]. В работе [Херменегилдо 86] описывается управление бектрекингом при И-параллельном исполнении логических программ. Ряд работ посвящен реализации параллельных языков на существующей в настоящее время параллельной архитектуре [Робинсон Я. 86, Леви 86б].

В рамках японского проекта вычислительных систем пятого поколения описано несколько машин, предлагаемых для реализации логических программ. Это последовательные машины PSI [Йокота 83] и PEK [Канеда 86], параллельные машины PIE [Гото 84] и PIM-R [Онаи 85]. Упомянем также работу [Халим 86], в которой описывается управляемая данными (data driven) машина, реализующая ИЛИ-параллелизм.

2. ПРОГРАММИРОВАНИЕ С ПОМОЩЬЮ РАВЕНСТВ И ПОДСТАНОВОК

В этом разделе рассматриваются формулы специального вида

$$p = q, \tag{1}$$

где p и q — *выражения* (*термы*), построенные естественным образом из символов (имен) функций, переменных и констант (например, $f(x) = x^2 + 1$ или $a(x, 0) = x$, $a(x, s(y)) = s(a(x, y))$[1]). Такие формулы называются *равенствами*. Логика, в которой формулами являются только равенства, называется *логикой равенств* или *эквациональной логикой*, а программирование с помощью одних равенств иногда называют *эквациональным программированием*. Законы (аксиомы и правила вывода) эквациональной логики выражают естественные свойства равенства — рефлексивность, симметричность, транзитивность и подстановочность:

$$1.\ p = p \quad 2.\ \frac{p = q}{q = p} \quad 3.\ \frac{p = q,\ q = t}{p = t}$$

$$4.\ \frac{p = q}{\theta p = \theta q} \quad 5.\ \frac{p_1 = q_1, \ldots, p_n = q_n}{f(p_1, \ldots, p_n) = f(q_1, \ldots, q_n)} \tag{2}$$

где p, q, t, p_i, q_i — выражения, f — символ функции, а θt — выражение, полученное из выражения t подстановкой θ (см. разд. 1.1) выражений вместо переменных (над чертой — посылка правила вывода, а под чертой — его заключение).

Равенства — удобный и естественный аппарат, применяемый для определения функций, отношений, множеств и т. д (см. [Агафонов 82, 84]). Однако надо иметь в виду, что их *семантика* (математический смысл) может определяться поразному. При этом одни определения могут быть более *операционными*, *процедурными*, а другие более *декларативными*, *непроцедурными*. Это аналогично различиям между теоретико-модельным, деиотационным и процедурным определениями логических программ в разд. 1.1. В контексте логического программирования мы хотим привлечь внимание к вариантам операционного определения, которые позволяют рассматривать равенство или систему равенств как «логическую программу», определяющую «логические вычисления», не похожие на вычисления в традиционных языках программирования или в машинах традиционной архитектуры.

---

[1]) Определение терма было дано в разд. 1.1. Здесь в термы могут входить функции, принимающие логические значения. Они представляют отношения. Выраженнями в разд. 1.1 назывались термы и атомы.

2.1. Определение функций равенствами и функциональное программирование

Рассмотрим определение функции равенством вида

$$f(x_1, \ldots, x_n) = t, \tag{3}$$

где f — символ (имя) определяемой функции, $x_1, \ldots, x_n$ — переменные, t — выражение, построенное из символов базисных (исходных, заданных) функций, констант и переменных $x_1, \ldots, x_n$. В t может входить символ f, и тогда равенство и определение с его помощью называется *рекурсивным*. В общем случае k функций $f_1, \ldots, f_k$ определяются k равенствами $f_1(x_1, \ldots, x_{n_1}) = t_1, \ldots, f_k(x_1, \ldots, x_{n_k}) = t_k$, но мы ограничимся одним равенством (3). Равенства вида (3) лежат в основе *языков функционального программирования*[1]), самым старым из которых является «чистый» Лисп [Лавров 78] (см. также его вариант — «строго функциональный язык» из книги [Хендерсон 80]), а примерами более современных языков могут служить Рефал [Базисный Рефал 77], FP [Бэкус 78], HOPE [Мур 82], ML [Гордон 79], SASL [Тёрнер 79], KRC [Мейра 84], MIRANDA [Тёрнер 84], TEL [Леви 83], HASL [Абрамсон 84], FP2 [Жорран 86], Amber [Карделли 86] (см. также [Функциональные языки 85]).

Один из вариантов операционной семантики равенства (3) определяется следующим образом. Пусть $t^{x_1 \ldots x_n}_{t_1 \ldots t_n}$ — выражение, получающееся из t заменой переменных $x_1, \ldots, x_n$ на выражения $t_1, \ldots, t_n$ (замена переменных на выражения называлась в разд. 1.1 подстановкой). От равенства (3) перейдем к правилам

$$f(t_1, \ldots, t_n) \to t^{x_1 \ldots x_n}_{t_1 \ldots t_n}, \tag{4}$$

которые будут работать в определении семантики как *правила подстановки* правой части правила вместо левой (или *правила замены* левой части на правую). Чтобы получить значение функции $f(d_1, \ldots, d_n)$ для набора значений аргументов (констант) $d_1, \ldots, d_n$, применим к $f(d_1, \ldots, d_n)$ правило подстановки $f(d_1, \ldots, d_n) \to t^{x_1 \ldots x_n}_{d_1 \ldots d_n}$, т. е. заменим $f(d_1, \ldots, d_n)$ на $t^{x_1 \ldots x_n}_{d_1 \ldots d_n}$. В полученном выражении $t^{x_1 \ldots x_n}_{d_1 \ldots d_n}$ заменим каждое выражение вида $g(c_1, \ldots, c_l)$, где g — базисная функция, а $c_1, \ldots, c_l$ — выражения, его значением (когда оно определено). Если в оставшемся выражении есть

[1]) *Функциональные языки* называют также *аппликативными языками* (слово «аппликация» означает здесь применение функции к аргументу).

вхождение выражения вида $f(t_1, \ldots, t_n)$, то применим к нему правило подстановки (4), т. е. заменим его на $t^{x_1 \ldots x_n}_{t_1 \ldots t_n}$. Выражение может содержать несколько вхождений такого вида. Поэтому, если мы хотим, чтобы процесс вычисления был детерминированным, надо фиксировать *правило выбора* одного такого вхождения. Например, будем всегда выбирать *самое левое внутреннее вхождение*, т. е. не содержащее внутри себя вхождений вида $f(t_1, \ldots, t_n)$. (Аналогичное правило выбора подвыражения будем применять и при вычислении базисных функций.)

Таким образом, мы определяем вычисление значения $f(d_1, \ldots, d_n)$ как процесс *преобразования выражений*, один шаг которого состоит в выполнении подстановки (4) или в замене выражения с базисной функцией $g(c_1, \ldots, c_n)$ его значением. Например, для функции «91» Маккарти, определяемой равенством

$$f(x) = \textit{если } x > 100 \textit{ то } x - 10 \textit{ иначе } f(f(x + 11))$$

при вычислении значения $f(99)$ выражение $f(99)$ будет преобразовываться следующим образом [1]):

$$f(99) \Rightarrow \textit{если } 99 > 100 \textit{ то } 99-100 \textit{ иначе } f(f(99 + 11)) \Rightarrow$$
$$f(f(110)) \Rightarrow f(\textit{если } 110 > 100 \textit{ то } 110-10 \textit{ иначе } f(f(110 + 11)))$$
$$\Rightarrow f(100) \Rightarrow \ldots \Rightarrow 91$$

Кроме семантики «самой левой внутренней подстановки», фактически используемой в языках Лисп и Рефал, рассматривались и другие стратегии преобразования выражений (см. [Манна 74, Дауни 76]), хотя в книге [Хендерсон 80, с. 192] утверждается, что «для функционального программирования этот механизм намного полезнее других и применяется в большинстве случаев». Описанная здесь семантика занимает промежуточное положение между более декларативной *денотационной семантикой* (см. [Манна 74, Агафонов 82]) и более операционной (процедурной) семантикой в терминах «SECD-машины» [Хендерсон 80] или «языка описания реализации» [Лавров 78] (см. также [О'Донелл 77]). Первая рассматривает равенство (1) как *функциональное уравнение* относительно функциональной переменной f и определяет соответствующую функцию не процессом ее вычисления, а как

---

[1]) Символ $\Rightarrow$ обозначает шаг преобразования

$$\textit{если } x \textit{ то } y \textit{ иначе } z = \begin{cases} y, & \text{если } x \text{ истинно} \\ z, & \text{если } y \text{ ложно} \end{cases}$$

решение этого уравнения (если решений несколько, то выбирается одно — «наименьшее»)[1]). Вторая включает в себя конкретный способ представления, хранения и обработки выражений с использованием магазинов (стеков). Она ближе к традиционной реализации функционального языка на машинах традиционной архитектуры, тогда как «подстановочная» семантика ближе к реализации на так называемых редукционных машинах (см. далее). Подстановочная семантика ближе по духу к логическому программированию в том отношении, что *преобразования выражений с помощью систем подстановок* представляют собой один из видов *формальных исчислений*, традиционно изучаемых в логике (мы вернемся к этому в разд. 2.2).

Идея редукционной машины в свою очередь уходит корнями в разделы логики, называемые *ламбда-исчислением* (λ-исчислением) и *комбинáторной логикой*, зародившиеся задолго до появления электронных вычислительных машин (см. [Барендрегт 81]). Здесь одно из основных понятий — это λ-выражение $\lambda x_1, \ldots, x_n.t$, где $x_1, \ldots, x_n$ — переменные и t — выражение[2]). Его значением является функция переменных $x_1, \ldots, x_n$, определяемая выражением t (значение этой функции для $x_1 = a_1, \ldots, x_n = a_n$ есть значение выражения t при $x_1 = a_1, \ldots, x_n = a_n$). Например, значением λ-выражения $\lambda x.x^2 + 1$ является функция, значение которой для $x = 3$ равно $3^2 + 1 = 10$. Применение (аппликация) λ-выражения к аргументу дает соответствующее значение функции, которую оно обозначает (например, $(\lambda x.x^2 + 1)(3) = 10$).

Используя λ-выражение, равенство (3) можно записать

$$f = \lambda x_1, \ldots, x_n.t. \qquad (5)$$

Но здесь λ-выражение, стоящее в правой части равенства, может содержать символ определяемой функции f[1]). Поэтому, чтобы определить его семантику, придется воспользоваться техникой, о которой шла речь выше. В Лиспе равенству (5) соответствует специальное *S*-выражение

SEXPR f (LAMBDA ($x_1\ x_2\ \ldots\ x_n$) t'),

где t' — перевод в синтаксис Лиспа выражения t, а в строго функциональном языке книги [Хеидерсон 80] рекурсивное

[1]) Семантика наименьшего решения (или наименьшей неподвижной точки, ср. разд. 1.1) определяет для равенства вида (1) не обязательно ту же функцию, что описанная выше семантика. Но в языке Рефал каждое равенство можно преобразовать в эквивалентное, для которого обе семантики дают одно и то же [Дехтярь 85].

[2]) Другой вариант синтаксиса λ-выражений — $\lambda(x_1, \ldots, x_n)t$.

[3]) Заметим, что обычное обозначение функции $f(x)$ двусмысленно: иногда им обозначают саму функцию f, а иногда значение функции (результат ее применения к аргументу x).

равенство (5) можно записать как $f(x_1, \ldots, x_n) \equiv t$ или $\{пустьрек\ f = \lambda(x_1, \ldots, x_n)t\ f(z_1, \ldots, z_n)\}$.

Функции являются столь же равноправными значениями в функциональных языках, как числа, логические значения, списки и т. д. Они могут быть аргументами и значениями функций, и равенства (3) и (5) могут, вообще говоря, содержать функциональные переменные. Например, функцию *ред*(g, x), аргументами которой являются двуместная функция g и список $x = (x_1 \ldots x_n)$, а значением — $g(x_1, g(x_2, \ldots \ldots, g(x_n, c) \ldots))$, можно определить следующим образом: [1])

ред$(g, x) =$ *если равно*$(x,$ nil$)$ *то* c
иначе $g($car(x), *ред*$(g,$ cdr$(x)))$

Функции, аргументами и/или значениями которых являются функции, называют *функциями высших порядков, операторами* или *функционалами*. Выражение t в правой части равенства (3) можно рассматривать как определение функционала Φ с функциональной переменной f (т. е. $\Phi = \lambda f.t$), а решение f_0 этого уравнения как «неподвижную точку» этого функционала: $\Phi(f_0) = f_0$ (ср. с неподвижной точкой в разд. 1.1).

Давно замечено, что функции нескольких переменных можно свести к одноместным функциям (например, $f(x, y) = = f_x(y)$, где $f_x = \lambda y.f(x, y)$). В языке FP [Бэкус 78] и языке функциональных схем из [Кутепов 75] переход к одноместным функциям осуществляется с помощью списков и операций над ними. Далее (нефункциональные, предметные) переменные можно ликвидировать вообще, перейдя к выражениям, построенным из символов функций и операций над функциями (функций от функций), называемых *комбинаторами*.

Примерами комбинаторов могут служить композиция одноместных функций $\circ$, которая по функциям f и g дает функцию $f \circ g = \lambda x.f(g(x))$, или упомянутая выше редукция *ред*(g, x), рассматриваемая как операция над функцией g (обозначим ее /, как в АПЛ): $/g = \lambda x.$*ред*(g, x). Используя комбинаторы языка FP (там они называются функциональными формами), определение функции факториал вида (3)

$$\text{fact}(n) = \textit{если}\ n = 0\ \textit{то}\ 1\ \textit{иначе}\ n \cdot \text{fact}(n - 1)$$

можно переписать так:

$$\text{fact} = \text{eq} \circ [\text{id}, 0] \rightarrow 1;\ \times \circ [\text{id}, \text{fact} \circ - \circ [\text{id}, 1]],$$

[1]) Это по существу операция редукции / из языка АПЛ, в определении которой используются предикат равенства *равно*(x, y) и лисповские функции car(x) и cdr(x), дающие «голову» и соответственно «хвост» списка x.

где [ ], ∘, → — комбинаторы, а eq, ×, —, id — базисные функции.

Комбинáторная логика — это эквациональная логика комбинаторных выражений, т. е. выражений, составленных из комбинаторов, а также базисных функций и (функциональных) переменных. Свойства комбинаторов определяются равенствами, которые, в свою очередь, при их операционном понимании приводят к соответствующим правилам подстановки, называемым *редукциями*[1]). Модель вычисления, основанную на редукциях, называют *редукционной машиной* (она может быть реализована программно и/или аппаратно).

В реализации функционального языка SASL [Тёрнер 79] исходные равенства переводятся в комбинаторные выражения (включающие, в частности, классические комбинаторы *S*, *K*, *I*), которые «исполняются» редукционной SK-машиной, преобразующей комбинаторные выражения с помощью редукций. В [Мейра 84] аналогичная техника используется для языка KRC. Реализация функционального языка HOPE на редукционной машине ALICE описывается в [Мур 82]. В SK-машине и в ALICE зависимость функций (комбинаторов) от их аргументов представляется в виде графов. Эти графы являются одним из вариантов понятия *графа потоков данных*, а машины, архитектура которых основывается на использовании таких графов, называются *машинами потоков данных*. Они предназначаются для реализации не только функциональных языков, но и языков логического программирования [Брэйлсфорд 85, ван Эмден 82].

В [Джонсон 84] техника комбинаторов используется для реализации метода выполнения функциональных программ, называемого *ленивым вычислением* (в русском переводе книги [Хендерсон 80] употребляется термин *вычисление с задержками* (lazy evaluation)). Суть его в том, чтобы не вычислять те подвыражения (не вызывать «фактические параметры» функций), которые в действительности не понадобятся, и «лениво» откладывать некоторые вычисления до того момента, когда появится необходимость в их результатах (это называют еще «вызовом по необходимости»). В книге [Хендерсон 80] ленивые вычисления рассматриваются как «функциональный подход к параллелизму». Они используются и в ALICE, которая является параллельной многопроцес-

[1]) В классической бестиповой комбинаторной логике комбинаторы *S*, *K*, *I* — это по существу особые одноместные функции. Их свойства и связь с λ-исчислением определяются равенствами $Sfgh = fh(gh)$, $S = \lambda x, y, z.xz(yz)$, $Ix = x$, $I = \lambda x.x$ и т. д. ((gh) — применение функции g к аргументу h, $Sfgh$ — сокращенная запись выражения $(((Sf)g)h)$) (см. [Барендрегт 81, Тёрнер 79]).

сорной машиной и предлагается также как средство реализации параллельного языка хорновского логического программирования Парлог [Дарлингтон 83].

2.2. Равенства и системы подстановок термов

Теперь, не ограничиваясь равенствами вида (3), рассмотрим вариант подстановочной семантики системы равенств

$$E = \{l_1 = r_1, \ldots, l_n = r_n\} \qquad (6)$$

общего вида (1), потребовав только, чтобы все переменные, входящие в правую часть r_i равенства $l_i = r_i$, содержались среди переменных, входящих в левую часть l_i. Перейдем от равенств E к *системе подстановок термов* [1]) (выражений)

$$S = \{l_1 \to r_1, \ldots, l_n \to r_n\}. \qquad (7)$$

Расширим множество S до множества S^*, содержащего S и все подстановки, получаемые следующими действиями: если подстановка $l \to r$ уже включена в расширение и t — выражение (терм), а v — переменная, то подстановка $l_t^v \to r_t^v$ включается в расширение (S^* — «замыкание» S относительно этого действия). Тогда один шаг преобразования выражения p в выражение q с помощью подстановки $l \to r$ из S^* состоит в замене некоторого вхождения l в p на r. Обозначим это $p \Rightarrow q$. Если q получается из p за какое-нибудь число $k \geqslant 0$ таких шагов, то обозначим это $p \Rightarrow^* q$.

В отличие от преобразования, определенного в разд. 2.1, это преобразование происходит, вообще говоря, недетерминированно. Может оказаться, что выражение t можно преобразовать ($t \Rightarrow^* p$ и $t \Rightarrow^* q$) в разные выражения p и q («пути» преобразования t, так сказать, «расходятся»). Если все расходящиеся пути обязательно сходятся, т. е. всегда существует r такое, что $p \Rightarrow^* r$ и $q \Rightarrow^* r$, то система подстановок называется *конфлюэнтной* (свойство, эквивалентное этому, известно под названием *свойства Черча — Россера*). Выражение называется *тупиковым* (или выражением *в нормальной форме*), если к t не применимо ни одно правило подстановки из S^*. Система называется *конечно завершаемой* (или *нётеровой*), если не существует бесконечной последовательности шагов преобразования $t_1 \Rightarrow t_2 \Rightarrow t_3 \Rightarrow \ldots$, т. е. для каждого выражения t начинающаяся с него последовательность преобразуемых выражений должна закончиться тупиковым выражением.

---

[1]) Другие употребляемые названия — *система переписывания термов* (term rewriting system) и *редукционная система* (reduction system).

Конфлюэнтную и конечно завершаемую систему подстановок термов называют *канонической* (или *полной*). В ней все вычисления (преобразования выражений), начинающиеся произвольным выражением t, завершаются и дают один и тот же результат — тупиковое выражение (обозначим его $S(t)$), называемое нормальной формой выражения t. Определение $S(t)$ завершает определение операционной (подстановочной) семантики системы равенств (6), для которой система подстановок (7) каноническая.

Систему подстановок S можно, разумеется, рассматривать и независимо от равенств как способ определения и вычисления функций. Существуют и другие (возможно, более естественные) способы определения семантики равенств (см. [Юэт 80, О'Доннелл 77, Агафонов 84]), и представляет интерес связь между различными определениями. В частности, система равенств E задает множество $T(E)$ равенств, выводимых из E с помощью правил вывода (2) эквациональной логики. Множество равенств $T(E)$ называют *эквациональной теорией*, определяемой E, и два выражения t и s равны с точки зрения этой теории, если $t = s$ принадлежит $T(E)$ (обозначим это $t =_E s$). Каноническую систему подстановок K назовем *эквивалентной* системе равенств E, если для любых выражений p и q равенство $K(p) = K(q)$ справедливо тогда и только тогда, когда $p =_E q$. Система S, построенная выше по E простым «ориентированием» равенств слева направо, скорее всего не будет эквивалентна E, так как равенство $l = r$ по существу равносильно двум подстановкам $l \to r$ и $r \to l$. Техника получения по E эквивалентной канонической системы подстановок основывается на *методе Кнута — Бендикса* (см. обзоры [Юэт 80, Кучеров 85], в которых можно также найти более полные определения понятий, относящихся к системам подстановок термов и их связи с равенствами).

К языкам эквационального программирования, операционная семантика которых основана на системах подстановок термов, относятся OBJ [Гоген 79], AFFIRM [Массер 80], МАРС [Проскурин 83] и язык, описываемый в [Хоффман 82]. Так как это экспериментальные языки, не претендующие на высокую эффективность реализации, их, возможно, уместнее называть *языками эквациональной спецификации*[1]) [Агафонов 82, 84]. В ряде работ эквациональные языки рассматриваются как языки спецификации по отношению к таким языкам логического программирования, как Пролог, и обсуждается проблема перехода от эквациональной спецификации

[1]) Эквациональные спецификации называют также *алгебраическими*.

к эквивалентной логической программе [Бергман 81, 82; Дерансар 83].

Заметим, что систему подстановок термов, эквивалентную множеству равенств E, можно использовать не только для прямого вычисления функций, но и для выяснения их свойств, т. е. проверки того, выводимо ли из E такое-то равенство, не содержащееся непосредственно в E и выражающее свойства или связи между функциями, вытекающие из E. В [Лескан 83] обсуждается система REVE, предназначенная для экспериментирования с эквациональными теориями и системами подстановок термов. Применение таких систем для анализа эквациональных спецификаций абстрактных типов данных рассматривается в [Массер 82].

Следует отметить, что системы равенств вида (1) рассматривались Эрбраном и Гёделем еще на заре теории алгоритмов как один из способов уточнения понятий алгоритма и алгоритмически вычислимой функции. Переход от определения функций равенствами по Эрбрану — Гёделю к определению в «форме Клини» [1]) (см. [Клини 52, Мальцев 65, Агафонов 84]) можно считать прообразом перехода от недетерминированной подстановочной семантики к более процедурной семантике.

Ближайший по простоте к равенствам класс формул, расширяющий выразительные возможности, это так называемые *условные равенства* (или *квазитождества*), имеющие вид

$$e_1 \wedge e_2 \wedge \ldots \wedge e_n \rightarrow e_{n+1},$$

где каждое e_i — равенство, $\wedge$ — конъюнкция («и»), $\rightarrow$ — импликация («влечет»). Для определения операционной семантики условных равенств используются соответственно *условные правила подстановки*, в которых обычное правило подстановки снабжается дополнительным условием его применения [Каплан 84, Дростен 84, Воробьев 86]. Разумеется, системы условных подстановок можно, как и обычные, использовать самостоятельно, независимо от условных равенств.

Заметим также, что процедурную (операционную) семантику хорновских дизъюнктов $A_1 \wedge A_2 \wedge \ldots \wedge A_n \rightarrow A_{n+1}$ тоже можно рассматривать как «подстановочную» семантику: от импликации $A_1 \wedge A_2 \wedge \ldots \wedge A_n \rightarrow A_{n+1}$ надо перейти к правилу подстановки $A_{n+1} \rightarrow A_1, A_2, \ldots, A_n$ (при унификации атома A_{n+1} получается конкретный применяемый вариант правила) [Бергман 81, 82]. Но здесь правило подстановки

[1]) Описание функций равенствами специального вида, при построении которых используются операторы примитивной рекурсии, суперпозиции и минимизации.

применяется к формуле, которая заменяется последовательностью формул. Это одно из возможных обобщений понятия правила подстановки (замены, редукции, переписывания). Другие варианты и обобщения этого понятия включают в себя продукции Поста и нормальные алгорифмы Маркова (см. [Мальцев 65]); формальные системы, рассматриваемые в книге [Смальян 62], контекстно-свободные грамматики и обобщенные системы продукций, применяемые в экспертных системах (см. обзор [Агафонов 84]).

2.3. Равенства в резолюционном программировании

Чтобы работать резолюционными методами (см. разд. 1.1 и 3.2) с логическими формулами, содержащими равенства, надо включить в рассмотрение аксиомы, выражающие свойства равенства (это законы эквациональной логики (2), переписанные в виде утверждений, т. е. правилу $\frac{p=q,\ q=t}{p=t}$ соответствует аксиома $p=q \wedge q=t \to p=t$ и т. д.). Оказывается, однако, что прямое добавление аксиом равенства приводит к большому увеличению пространства поиска и неэффективности правила резолюции. Поэтому было предложено заменить аксиомы равенства более эффективным правилом вывода — *парамодуляцией.* Для дедуктивной системы, состоящей из резолюции и парамодуляции, разработан ряд стратегий поиска вывода, аналогичных стратегиям для резолюции без равенства (см. книгу [Чень 73]). Обзор методов работы с равенствами в дедуктивных процедурах резолюционного типа содержится в [Дегтярев 86а, Стикель 86].

Особого внимания с точки зрения логического программирования заслуживает проблема введения равенств (в качестве атомов) в хорновские дизъюнкты. Ей посвящены работы [Хеншен 83, Фрибург 84, Дегтярев 86б, Кокс 86].

В [Гоген 84] обсуждаются принципы языка Eqlog, в котором идеи хорновского программирования в духе Пролога сочетаются с эквациональным программированием в духе разд. 2.2. Программа на языке Eqlog может состоять из множества равенств E и множества хорновских дизъюнктов H, содержащих равенства. Для системы равенств E, которой соответствует каноническая система подстановок термов, в реализации языка предполагается использовать алгоритм *E-унификации*[1] Фэя — Юло [Фэй 79, Юло 80]. На хорновские дизъюнкты накладываются некоторые ограничения, чтобы проло-

[1]) Подстановка θ (в смысле разд. 1.1) называется E-унификатором выражений p и q, если $\theta p =_E \theta q$.

говский метод работы с ними не мешал E-унификации. Упомянутый алгоритм E-унификации основан на правиле вывода, называемом *сужением* (narrowing) [Слэйгл 74], которое в случае равенств, обладающих канонической системой подстановок термов, является значительно более эффективным, чем парамодуляция, и используется в других языках [Редди 85].

В языке LOGLISP [Робинсон 82] прологообразное логическое программирование комбинируется с функциональным программированием в духе разд. 2.1, а именно язык Лисп расширяется логическими средствами, с которыми можно работать методом LUSH-резолюции[1]). Взаимоотношениям функциональных и логических языков программирования и методам интеграции характерных для них средств посвящены работы [Кан 81] (язык UNIFORM), [Беллиа 82] (язык LCA), [Барбути 85] (язык LEAF), [Редди 85] и обзор [Беллиа 86]. Объединение резолюционного программирования с функциональным — одно из важных направлений развития логического программирования в широком смысле [Робинсон 83, 86; Кан 81].

3. ЛОГИЧЕСКИЕ СПЕЦИФИКАЦИИ И РАСШИРЕНИЯ ПОНЯТИЯ ЛОГИЧЕСКОЙ ПРОГРАММЫ

Этот раздел посвящен понятиям и методам, относящимся к логическому программированию, но выходящим за рамки прологообразного и эквационального программирования, которое рассматривалось в предыдущих разделах. Среди большого числа работ на эту тему мы постараемся выделить те, которые прямо относятся к логическому программированию, оставив в стороне относящиеся больше к логике и лишь косвенно — к программированию. Однако не следует рассматривать это разделение как догму: «то, что попадает в выделенный класс, это логическое программирование, а то, что не попадает, — это скорее логические спецификации или логические основы программирования». Трудность как раз и заключается в том, что сейчас непонятно, какие методы логики смогут в дальнейшем пригодиться в практическом программировании, а какие нет.

Разделы 3.1—3.3 посвящены методам логического программирования в широком смысле: в разд. 3.1. обсуждаются такие расширения логического программирования, которые наиболее близки к логическому программированию в узком смысле; разд. 3.2 посвящен методам автоматического доказа-

[1]) Другой подход к объединению Лиспа с логическим программированием см. в [Коморовский 82].

тельства теорем, которые имеют (или могут иметь) отношение к логическому программированию: в разд. 3.3. кратко описывается подход к синтезу программ, который можно охарактеризовать тезисом Бишопа [Бишоп 85] — «математика как язык программирования высокого уровня»[1]). По ходу изложения мы будем упоминать (пока немногочисленные) системы, в которых реализованы некоторые из указанных подходов к расширениям понятия логической программы. Из публикаций, посвященных таким системам, читатель может получить полезную информацию о том, что уже практически сделано в этой области нетрадиционного логического программирования.

3.1. Логические спецификации и синтез логических программ

Уже с первых лет развития логического программирования делались попытки расширить Пролог и другие языки с помощью самых различных способов. Они объясняются определенной узостью хорновских дизъюнктов, которые хотя и определяют любую вычислимую функцию и предикат, но не всегда дают для них удобное представление. Расширения, непосредственно относящиеся к Прологу, упоминались в разд. 1. Здесь мы рассмотрим некоторые другие.

Расширения логического программирования тесно связаны с понятием *синтеза программ*. Как определяется в [Непейвода 85], «синтез программ — это конструктивные методы перехода от спецификаций задачи к конкретным программам, реализующим эти спецификации».[2]) При рассмотрении спецификаций вообще, и тем более выходящих за пределы логического программирования, бывает трудно провести грань между *спецификациями и программами* (см. обзор по языкам спецификации программ [Агафонов 84]). Различие между ними можно сформулировать следующим образом: программы — это *эффективно исполнимые* спецификации. Но значение терминов «эффективный» и «исполнимый» весьма неточно и может меняться с развитием теории, методов реализации и с появлением новых высокопроизводительных ЭВМ. Вообще всю историю развития языков высокого уровня можно рассматривать как историю расширения понятия «эффективно исполнимая спецификация».

В качестве спецификаций для синтеза программ обычно стараются выбирать такие, которые наиболее безболезненно

[1]) Аналогичное понимание роли математики в программировании встречается в более ранних работах Глушкова (см. [Глушков 86]).

[2]) Более частное понятие синтеза программ рассматривается в [Лавров 82б, Крайзель 75, 85].

могут быть преобразованы в программы. Однако проблема синтеза программ для традиционных языков программирования наталкивается на одну существенную трудность: спецификация пишется на одном языке, программа на совсем другом и, чтобы связать эти два языка, приходится вводить третий — вроде логик Хоара. В синтезе логических программ эта трудность отсутствует: описание свойств программы и сама программа пишутся *на одном языке* — языке исчисления предикатов, и поэтому проблему синтеза (и верификации) логических программ можно выразить также на этом языке, не прибегая к посторонним языкам вроде логики Хоара. Таким образом, синтез логических программ заключается в конструктивных методах перехода от спецификации на языке исчисления предикатов (вообще говоря, не являющейся непосредственно эффективно исполнимой), к непосредственно исполнимой спецификации на том же языке. Методы перехода могут быть самыми различными. Заметим, что в сформулированном нами виде проблема синтеза логических программ становится проблемой скорее из области математической логики, чем из области программирования. Впервые обратили на это внимание Кларк и Зикель [Кларк 77].

Перейдем к описанию существующих подходов к синтезу логических программ.

В разд. 1.3.1 упоминались расширения логических программ, связанные с встраиванием в Пролог *средств управления* и базирующиеся на идее «алгоритм = логика + управление». Написание программы при таком подходе разбивается на две части — логическую спецификацию алгоритма и указание средств управления для эффективного исполнения этой спецификации.

В [Кларк 82а, б, в] на примере задачи численного интегрирования продемонстрировано одно из уточнений этого подхода: исходная (хорновская) спецификация рассматривается как недетерминированный алгоритм, который представляет класс детерминированных алгоритмов, а детерминированные эффективные версии этого алгоритма получаются при добавлении к хорновским дизъюнктам новых условий, выполняющих роль охран (см. [Дейкстра 76]). Например, если в исходной логической программе имеются определения вида

$$A \leftarrow B_1, \ldots, B_n$$
$$A \leftarrow C_1, \ldots, C_m$$

и мы хотим, чтобы вызов первой процедуры для решения A использовался, только когда выполняется условие D, а вто-

рой — когда выполняется E[1]), то мы меняем эту пару определений на

$$A \leftarrow D,\ B_1,\ \ldots,\ B_n$$
$$A \leftarrow E,\ C_1,\ \ldots,\ C_m$$

Несколько подходов к синтезу логических программ основываются на использовании спецификаций вида

$$P(t_1,\ \ldots,\ t_n) \leftarrow A, \qquad (1)$$

где A может содержать любые логические связки, в том числе квантор всеобщности. Главная причина использования таких подходов — уже упоминавшаяся идея записи программы и ее свойств на одном языке. Простым примером расширенной спецификации может служить определение равенства множеств

$$Равно\ (y, z) \leftarrow (\forall x)(x \in y \leftrightarrow x \in z).$$

Однако такие спецификации могут, вообще говоря, не иметь ясной процедурной семантики в отличие от хорновских дизъюнктов. Непонятно, например, как проверять условие $(\forall x)\,A(x)$. В работе [Сато 84] предлагается преобразование таких расширенных программ с помощью известной техники обращения с отрицанием: показано, как по логической программе, вычисляющей предикат P, построить другую логическую программу, которая вычисляет часть отрицания P (под частью здесь понимается подмножество отношения, задаваемого P). Так как связки $\rightarrow$ и $\leftrightarrow$ выражаются через конъюнкцию и отрицание, а с квантором существования в логическом программировании дело обстоит вполне благополучно, то остается только корректно вычислять подцели вида $(\forall x)\,A(x)$. Это предлагается делать эквивалентной заменой этой формулы на $\neg(\exists x)\neg A(x)$. Такой подход гарантирует корректность логической программы, полученной в результате преобразований, но, естественно, ни о какой полноте не может быть и речи. Некоторые другие модификации *трансформационного синтеза* логических программ описываются в [Хоггер 84, Накагава 86]. Трансформации, корректные относительно семантики Фиттинга (см. разд. 1.3.1), изучались в [Ллойд 84б].

В [Хариди 81, Хансон 82] предложен другой способ программирования с помощью расширенных спецификаций вида (1) — использование вместо метода резолюций *натурального вывода* [Генцен 35, Правиц 65]. Это позволяет работать с

[1]) Если D является отрицанием E, то такая замена аналогична использованию функции *если-то-иначе* в функциональном программировании.

формулами практически произвольного вида, использовать потоки (stream), а также равенство в посылках и заключениях спецификаций. Вместо алгоритма унификации используется так называемый генератор простых равенств. В настоящее время ведутся работы по реализации языка логического программирования, основанного на естественном выводе [Хариди 84], и даже созданию специальной вычислительной машины с параллельной архитектурой для этого языка. В то же время у этого подхода имеются некоторые недостатки, которые, впрочем, неизбежны. (Так как произвольные спецификации могут приводить, например, к невычислимым функциям.) Например, очень трудно предугадать, какая спецификация окажется выполнимой, а какая нет. Так же неприятно дело обстоит с вопросами полноты, корректности и построения подходящих семантик.

В [Хагия 84] предложено рассматривать расширенные спецификации вида (1) на основе теории итерированных индуктивных определений Мартин-Лёфа [Мартин-Лёф 82]. При определенных условиях у такого набора определений может существовать (иерархическая) наименьшая неподвижная точка. Подробно об этом подходе можно прочитать в статье [Хагия 84] в этом сборнике. Индуктивные определения в языках спецификаций рассматриваются также в [Крицкий 86].

Вводя нехорновские спецификации для расширения класса логических программ, необходимо гарантировать их исполнимость, т. е. существование эффективного алгоритма поиска вывода (см. разд. 3.2). Такой подход продемонстрирован в [Мартьянов 87]. Здесь класс спецификаций ограничивается до введенного авторами класса $\alpha\beta$-формул, для которого имеется хороший алгоритм поиска вывода, основанный на методе инвариантных преобразований формул [Мартьянов 82].

В [Гончаров 86] предложено рассматривать более широкую, чем логическое программирование, концепцию — *семантическое программирование*, в основе которой лежат теория моделей и теория допустимых множеств (см. [Барвайз 75]). Одним из уточнений семантического программирования является Σ-программирование [Гончаров 85]. В Σ-программировании за основу берется некоторая модель $\mathfrak{M}$, которая понимается как базовая, а функции и предикаты на этой модели — как встроенные.[1]) Наряду с моделью $\mathfrak{M}$ рассматривается *списочная надстройка* над $\mathfrak{M}$ — списки элементов

[1]) Использование в Прологе встроенных функций и предикатов (сложение, умножение и т. д.) портит логику программ и препятствует построению точных семантик. Поэтому представляется вполне естественным рассматривать такие функции и предикаты отдельно.

$\mathfrak{M}$, списки списков и т. д., на которых определены операции head, tail, cons [1]) $(\text{head}((a_1, \ldots, a_n)) = a_n$, $\text{tail}((a_1, \ldots, a_n)) = (a_1, \ldots, a_{n-1})$, $\text{cons}((a_1, \ldots, a_n), b) = (a_1 \ldots, a_n, b))$ и два предиката: $x \in y$ («x — элемент списка y») и $x \sqsubseteq y$ («x — начало списка y, т. е. $x = (y_1, \ldots, y_l)$, $y = (y_1, \ldots, y_n)$ и $l \leqslant n$»). Язык теории списков очень выразителен, на нем просто и естественно записываются свойства многих конструкций логического программирования, в том числе метауровневых.

Σ-программы состоят из определений предикатов с помощью *Σ-формул*, которые, в свою очередь, строятся из Δ_0-формул с помощью связок $\wedge$, $\vee$ квантора существования $\exists x$ и ограниченных кванторов $(\forall x \in t)$, $(\exists x \in t)$, $(\forall x \sqsubseteq t)$, $(\exists x \sqsubseteq t)$, где t — терм, не содержащий переменной x. *Δ_0-формулы* — это любые формулы логики первого порядка, в которых все кванторы ограниченные.

Например, с помощью Δ_0-формул можно запнсать равенство списков как множеств в виде

$$\textit{Равны}\ (x, y) \leftrightarrow (\forall z \in x)\, z \in y \wedge (\forall z \in y)\, z \in x$$

а определение упорядоченного списка — в виде

$$\textit{Упорядочен}(x) \leftrightarrow (\forall y \sqsubseteq x)\,(y = nil \vee tail(y) = nil \vee head(y) \leqslant head(tail(y)))$$

Σ-программирование основано на тезисе «вычислимость = = Σ-определимость» [Барвайз 75]. Это означает, что класс всех вычислимых предикатов допускает эффективное представление в виде Σ-формул [2]). Кроме того, любой набор спецификаций, заданных в виде Σ-формул, обладает свойствами, которые позволяют говорить о его вычислимости, т. е. имеет процедурную семантику и может поэтому считаться программой. У Σ-программ имеется точная теоретико-модельная семантика [Гончаров 85], [Воронков 87 а, б, в]. Исполнение Σ-программы заключается в проверке истинности запроса на модели. В качестве запросов выступают также Σ-формулы. Эта проверка истинности достаточно эффективна в случае определений, заданных Δ_0-формулами, однако рассмотрение неограниченного квантора существования может привести, вообще говоря, к бесконечному перебору.

Другая процедурная семантика, основанная на поиске по образцу, описана в [Воронков 86а] и уточнена в [Воронков

[1]) Они аналогичны лисповскнм операцням car, cdr, cons, но «головой» списка $(a_1, \ldots, a_n)$ считается a_n, а не a_1, как в Лиспе.

[2]) Для вычислимых предикатов на натуральных числах это известная теорема о диофантовости перечислимых множеств [Матнясевич 70].

87в]. Эта семантика показывает, что Σ-программы и хорновские логические программы легко преобразуются друг в друга. Приведем здесь некоторые детали перевода такой логической программы в Σ-программу и наоборот. Пусть в логической программе все строки, определяющие предикат P, — это

$$P(\bar{t}_1) \leftarrow Q_1(\bar{r}_1), \ \ldots, \ R_1(\bar{s}_1)$$
$$\vdots \qquad\qquad \vdots$$
$$P(\bar{t}_n) \leftarrow Q_n(\bar{r}_n), \ \ldots, \ R_n(\bar{s}_n)$$

Тогда соответствующее определение P в виде Σ-программы выглядит следующим образом:

$$P(\bar{x}) \leftrightarrow (\exists \bar{y}_1)(\bar{x} = \bar{t}_1 \wedge Q_1(\bar{r}_1) \wedge \ \ldots \ \wedge R_1(\bar{s}_1)) \vee \ldots$$
$$\ldots \vee (\exists \bar{y}_n)(\bar{x} = \bar{t}_n \wedge Q_n(\bar{r}_n) \wedge \ \ldots \ \wedge R_n(\bar{s}_n)),$$

где $\bar{y}_i$ — все переменные термов $\bar{t}_i, \bar{r}_i, \ \ldots, \ \bar{s}_i$.

Обратный перевод не так прост. При обратном переводе по заданной Σ-программе вначале строится эквивалентная ей Σ-программа, в определениях которой имеется не более одной логической связки или квантора и нет вхождений символов head, tail. После этого каждое такое определение переводится в одну или несколько строк логической программы по следующим законам (мы продемонстрируем не все случаи):

1) $P(\bar{x}) \leftrightarrow (\exists y) R(\bar{x}, y)$ переводится в $P(\bar{x}) \leftarrow R(\bar{x}, y)$,
2) $P(\bar{x}) \leftrightarrow (\exists y \in t) R(\bar{x}, y)$ переводится в $P(x) \leftarrow x \in t$, $R(\bar{x}, y)$,
3) $P(\bar{x}) \leftrightarrow (\forall y \in t) R(\bar{x}, y)$ переводится в $Q(\bar{x}, \text{nil})$;
$Q(\bar{x}, \text{cons}(z, y)) \leftarrow R(\bar{x}, y), \ Q(\bar{x}, z); \ P(\bar{x}) \leftarrow Q(\bar{x}, t)$,

где Q — новый предикатный символ.

4) $P(\bar{x}) \leftrightarrow (\exists y \sqsubseteq t) R(\bar{x}, y)$ переводится в $P(\bar{x}) \leftarrow y \sqsubseteq t$, $R(\bar{x}, y)$;
5) $P(\bar{x}) \leftrightarrow (\forall y \sqsubseteq t) R(\bar{x}, y)$ переводится в $S(\bar{x}, \text{nil}) \leftarrow R(\bar{x}, \text{nil})$;
$S(\bar{x}, \text{cons}(z, y)) \leftarrow R(\bar{x}, \text{cons}(y, z)), \ S(\bar{x}, z); \ p(\bar{x}) \leftarrow S(\bar{x}, t)$,

где S — новый предикатный символ.

Этот простой и естественный перевод показывает, что определения в виде Σ-формул можно использовать для трансформационного синтеза логических программ. При этом, в отличие от изложенных ранее подходов, гарантируется не только корректность, но и полнота.

Для создания практических систем Σ-программирования нужны специализированные алгоритмы проверки Σ-формул на истинность. Для такой специализации можно предложить по крайней мере следующие подходы:

1) рассмотрение специальных классов моделей (или даже отдельных практически важных моделей);

2) рассмотрение специальных видов Σ-формул;

3) объединение подходов 1) и 2).

Третий подход продемонстрирован в логической части языка Ридэр [Смоян 85], где модель представлена в виде таблиц, т. е. списков ограниченной глубины, а запросы выделяются из подмножества класса Δ_0-формул. За счет таких ограничений исполнение программ логической части Ридэра происходит эффективно.

Первым на возможность использования теории допустимых множеств в программировании указал Ю. Л. Ершов [Ершов 83]. Более подробное изложение мотивов появления и теоретических основ Σ-программирования имеется в [Гончаров 85, 86; Ершов 83, 85, 86; Сазонов 86].

Синтез логических программ из конструктивных доказательств рассматривается в разд. 3.3.

3.2. Логическое программирование и методы автоматического вывода

Для всех систем логического программирования как в узком, так и в широком смысле характерно одно общее обстоятельство: для исполнения программ используются встроенные *системы автоматического поиска вывода*[1]). Схема работы таких систем практически одинакова: на вход программы подается запрос вида «Найти $\bar{x}$, такие, что $P(\bar{x})$», где P — формула, $\bar{x}$ — набор переменных, и система, используя заложенные в нее методы поиска вывода, пытается найти ответ на такой запрос. Поэтому методы поиска вывода, встроенные в систему, определяются видом аксиом (т. е. логической программой) и видом запросов.

В Прологе и подобных ему языках в качестве формул используются хорновские дизъюнкты. Это связано с тем, что, во-первых, большой класс практически важных задач допускает естественное описание с помощью этих формул, а во-вторых, для них существует эффективный метод поиска вывода (см. разд. 1.1).

Механизм поиска вывода, используемый в Прологе, берет свое начало от *метода резолюций* [Робинсон 65]. Формулами

[1]) Часто используется и другая терминология: автоматическое (машинное) доказательство теорем, автоматизированный вывод (дедукция) и т. д.

метода резолюций являются *дизъюнкты*. Они имеют вид $A_1 \vee \ldots \vee A_n$, где $A_1, \ldots, A_n$ — литералы. Атомы называются положительными литералами, а их отрицания — отрицательными. Порядок литералов в дизъюнкте несуществен. Используемые в Прологе формулы A и $A_1 \wedge \ldots \wedge A_n \rightarrow B$ эквивалентны хорновским дизъюнктам A и $\neg A_1 \vee \ldots \vee \neg A_n \vee B$.

Метод резолюций, получив на вход набор дизъюнктов, пытается построить вывод пустого дизъюнкта $\square$ с помощью двух правил — правила резолюции и правила склейки (факторизации). *Правило резолюции* позволяет из дизъюнктов

$$D_1 = \neg A(\bar{t}) \vee A_1 \vee \ldots \vee A_n \text{ и } D_2 = A(\bar{s}) \vee B_1 \vee \ldots \vee B_m$$

получить новый дизъюнкт $D = \theta(A_1 \vee \ldots \vee A_n \vee B_1 \vee \ldots \ldots \vee B_m)$, где θ — наиболее общий унификатор наборов термов $\bar{t}$ и $\bar{s}$. Дизъюнкты D_1 и D_2 называют родителями дизъюнкта D. Из-за того, что правило резолюции применяется к двум дизъюнктам, его иногда называют бинарной резолюцией. Переменные внутри одного дизъюнкта можно переименовать. *Правило склейки* позволяет из дизъюнкта $A(\bar{s}) \vee A(\bar{t}) \vee B_1 \vee \ldots \vee B_n$ (соответственно $\neg(A(\bar{s}) \neg A(\bar{t}) \vee \vee B_1 \vee \ldots \vee B_n$) получить $\theta(A(\bar{s}) \vee B_1 \vee \ldots \vee B_n)$ (соответственно $\theta(\neg A(\bar{s}) \vee B_1 \vee \ldots \vee B_n)$, где θ — наиболее общий унификатор наборов термов $\bar{t}$ и $\bar{s}$.

Если из формул $A_1, \ldots, A_n$ требуется вывести формулу B, то для доказательства методом резолюций вначале необходимо преобразовать формулы $A_1, \ldots, A_n, \neg B$ в множество эквивалентных им по выводимости дизъюнктов (как это делается, см., например, в [Чень 73]). Покажем действие метода резолюций и возможность его использования в логическом программировании на простых примерах. Пусть из формул

Лает(X, Y) → Собака(X) ∧ (Злой(X) ∨ Ненавидит(X, Y))
Собака(X) ∧ Злой(X) → Ненавидит(X, Y)
Лает(Моська, Слон)

требуется вывести формулу

($\exists UV$) Ненавидит (U, V).

В логических программах эта формула представляется в виде запроса «для каких U, V Ненавидит(U, V)?»

Из всех этих формул формируется начальное множество дизъюнктов

(1) ¬ Лает(X, Y) ∨ Собака(X)
(2) ¬ Лает(X, Y) ∨ Злой(X) ∨ Ненавидит(X, Y)
(3) ¬ Собака(X) ∨ ¬ Злой(X) ∨ Ненавидит(X, Y)

(4) Лает(Моська, Слон)
(5) ¬ Ненавидит(*U*, *V*)

Доказательство методом резолюций в этом случае выглядит следующим образом:

(6) Собака(Моська) (резолюция, из 1,4)
(7) ¬ Лает(*X*, *Y*) ∨ ¬ Собака(*X*) ∨ Ненавидит(*X*, *Y*) ∨ Ненавидит(*X*, *Y*) (резолюция, из 2, 3)
(8) ¬ Лает(*X*, *Y*) ∨ ¬ Собака(*X*) ∨ Ненавидит(*X*, *Y*) (склейка, из 7)
(9) ¬ Лает(Моська, *Y*) ∨ Ненавидит(Моська, *Y*), (резолюция из 6, 8)
(10) Ненавидит(Моська, Слон) (резолюция, из 4, 9)
(11) □ (резолюция, из 5, 10)

Прослеживая подстановки, сделанные вместо *U*, *V* в процессе вывода, мы получаем ответ на запрос *U*-Моська, *V*-Слон.

У метода резолюций имеется много разновидностей, которые приведены в [Чень 73, Стикель 86]. Отметим здесь некоторые из них, представляющие интерес в контексте логического программирования (в частности, связанные с Прологом).

Во *входной резолюции* (см. [Чень 73, § 7.3]) разрешается применять правило резолюции только в том случае, если один из родителей — входной дизъюнкт, т. е. дизъюнкт, который имелся в первоначальном множестве. Доказательство входной резолюцией для нашего примера выглядит следующим образом:

(6) ¬ Лает(*X*, *Y*) ∨ ¬ Собака(*X*) ∨ Ненавидит(*X*, *Y*) (резолюция и склейка, из 2, 3)
(7) ¬ Собака(Моська) ∨ Ненавидит(Моська, Слон) (резолюция, из 4, 6)
(8) ¬ Собака(Моська) (резолюция, из 5, 7)
(9) ¬ Лает(Моська, *Y*) (резолюция, из 1, 8)
(10) □ (резолюция, из 4, 9)

Входная резолюция слабее бинарной резолюции, т. е. существуют наборы дизъюнктов, из которых можно вывести пустой дизъюнкт □, пользуясь бинарной резолюцией, но нельзя вывести □ входной резолюцией.

Дизъюнкт называется положительным (отрицательным), если все литералы в нем положительные (отрицательные). *Отрицательная и положительная гиперрезолюции* (см. [Чень 73, § 6.5]) основаны на особом обращении с отрицательными положительными) дизъюнктами. В отрицательной гиперрезолюции входное множество дизъюнктов разбивается на два подмножества: первое состоит из отрицательных дизъюнктов,

ными дизъюнктами, мы как бы «выбиваем» положительные литералы из остальных дизъюнктов, получая, таким образом, новые отрицательные. В тех случаях, когда требуется выбить сразу несколько положительных литералов, правило гиперрезолюции может применяться сразу к нескольким отрицательным дизъюнктам и одному неотрицательному. Пример доказательства отрицательной гиперрезолюцией:

(6) ¬ Собака(X) ∨ ¬ Злой(X) (из 5, 3)
(7) ¬ Собака(X) ∨ ¬ Лает(X, Y) (из 5, 6, 2)
(8) ¬ Лает(X, Y) (из 7, 1)
(9) □ (из 8, 4)

Как положительная, так и отрицательная гиперрезолюции эквивалентны по силе бинарной резолюции.

Все разновидности резолюции предназначены для одной цели — уменьшения пространства поиска. Вывод, используемый в Прологе, является одновременно и выводом с помощью входной резолюции и отрицательной гиперрезолюции, а также, как отмечалось в разд. 1.1, — выводом с помощью SL-резолюций (см. [Стикель 86]).

Отметим несколько модификаций метода резолюций, недавно появившихся под влиянием Пролога. Это sprf-резолюция (simplified problem reduction format) [Плейстид 82], в которой дизъюнкты разных видов трактуются по-разному. Наиболее эффективно метод действует, когда входное множество «не очень» отличается от хорновского. В [Стикель 84] приводится метод, который пытается приспособить «технологию» поиска вывода Пролога (поиск в глубину, организация памяти) для произвольных дизъюнктов. В [Нейман 86] описывается реализация метода выделения подцелей — разновидность метода модельной элиминации [Лавленд 78], — который действует на хорновских дизъюнктах, использует поиск в глубину, но в то же время позволяет избежать в некоторых случаях зацикливания.

Все упомянутые модификации метода резолюций работают с дизъюнктивной нормальной формой доказываемой формулы. Имеется один метод, действующий на конъюнктивной нормальной форме, — это обратный метод Маслова [Маслов 64, 83]. Большая группа методов вообще не использует нормальных форм формул или использует так называемую негативную нормальную форму [Дегтярев 81, Эндрюс 81, Манна 85, 86, Маррэй 82, Бибель 82, Воронков 85]. Для иллюстрации опишем метод *пс-резолюции* [Маррэй 82]. В этом методе из бескванторных формул $A_1, \ldots, A_n$ требуется вывести бескванторную формулу B. Для этого формируется начальное множество формул, состоящее из $A_1, \ldots, A_n, \neg B$ и из них выводится формула Л (ложь). Единственное правило

вывода (пс-резолюция) формулируется следующим образом. Пусть даны две формулы A и B, в A положительно входят подформулы $C(\bar{t}_1), \ldots, C(\bar{t}_n)$, а в B отрицательно входят подформулы $C(\bar{s}_1), \ldots, C(\bar{s}_m)$. Тогда из A и B можно получить новую формулу $\theta(A' \vee B')$, где θ — самый общий унификатор последовательностей термов $\bar{t}_1, \ldots, \bar{t}_n, \bar{s}_1, \ldots, \bar{s}_m$, A' получается из A заменой всех указанных вхождений C на Л, а B' получается из B заменой всех указанных вхождений C на И (истина). После этого проводятся простые преобразования формул вида $A \,\&\, \text{И} \Rightarrow A$, $A \vee \text{И} \Rightarrow \text{И}$, $A \,\&\, \text{Л} \Rightarrow \text{Л}$ и т. д.

Для вышеприведенного примера доказательство выглядит следующим образом (здесь знаком «+» помечаются положительные, а знаком «—» — отрицательные вхождения атомов). Входные формулы:

(1) Лает$^-(X, Y) \rightarrow$ Собака$^+(X) \wedge$ (Злой$^+(X) \vee$ Ненавидит$^+(X, Y)$)

(2) Собака$^-(X) \wedge$ Злой$^-(X) \rightarrow$ Ненавидит$^+(X, Y)$

(3) Лает$^+$(Моська, Слон)

(4) $\neg$ Ненавидит$^-(U, V)$

Доказательство (в скобках указаны формулы, выступающие в роли C):

(5) Лает$^-(X, Y) \rightarrow$ Собака$^+(X) \wedge$ Злой$^+(X)$ (из 1, 4; Ненавидит)

(6) Собака$^+$(Моська) $\wedge$ Злой$^+$(Моська) (из 3, 5; Лает)

(7) Ненавидит$^+$(Моська, Y) (из 2, 6; Собака $\wedge$ Злой)

(8) Л (из 4, 7; Ненавидит)

Во всех приведенных примерах, как и в первом, прослеживая подстановки, сделанные вместо U и V, можно сделать вывод, что ответом на запрос «найти U, V, такие, что Ненавидит (U, V)» будет U-Моська, V-Слон.

Все цитированные нами методы относятся к логике первого порядка. Поиск вывода в логике высших порядков обсуждается в [Эндрюс 81, 84, Юэт 75].

Более подробно с положением дел в области автоматического доказательства теорем можно познакомиться по недавно вышедшим обзорам [Стикель 86, Воронков 86г, 87а, Дегтярев 86, Кучеров 85] и книге [Чень 73]. Некоторые примеры логического программирования с формулами произвольной структуры можно найти в [Хариди 81, Манна 85].

3.3. Конструктивная математика как логическое программирование

Конструктивная математика во многом отличается от классической. Один из основных тезисов конструктивной математики — «существовать — значит быть построенным» — на-

кладывает на средства, используемые в доказательствах, существенные ограничения. Как видно из этого тезиса, конструктивное доказательство теоремы существования объекта должно в принципе указывать способ построения (конструкцию) этого объекта. Если в качестве уточнения понятия «способ построения» взять понятие алгоритма, то это приводит к применению конструктивной математики в программировании. На возможность такого применения первыми указали, по-видимому, Бишоп [Бишоп 67] и Констейбл [Констейбл 71]. Речь идет о *синтезе* (*извлечении*) *программ из конструктивных доказательств* или дедуктивном синтезе программ (см. [Непейвода 78, 79, 85; Лавров 82а, б]).

Поскольку конструктивное доказательство утверждения $(\exists x)A(x)$ дает алгоритм вычисления такого t, что имеет место $A(t)$, то естественно считать этот алгоритм программой для вычисления t. Если объекты представляются термами какой-либо сигнатуры, то мы будем говорить, что терм t извлекается из доказательства формулы $(\exists x)\,A(x)$.

Пусть формула $(\forall \bar{x})\,\exists y\,A(\bar{x}, y)$ доказана конструктивно. (Понятие конструктивного доказательства интуитивно, здесь мы его не уточняем. Попытки дать точное определение для приложений к синтезу программ, охватывающее достаточно широкий класс исчислений, предпринимались, например, в [Воронков 86в, 87б, Свириденко 86]). Тогда такое доказательство указывает алгоритм, или программу f, удовлетворяющую «спецификации» $(\forall \bar{x})\,(\exists y)\,A(\bar{x}, y)$, в том смысле, что имеет место $(\forall \bar{x})\,A(\bar{x}, f(\bar{x}))$. Точно так же конструктивное доказательство П формулы $(\forall \bar{x})\,(A(\bar{x}) \vee \neg A(\bar{x}))$ дает алгоритм, или программу, для вычисления предиката A: по П и по любым конкретным значениям $\bar{t}$ переменных $\bar{x}$ мы можем эффективно узнать, верна или нет формула $A(\bar{t})$.

Одно из наиболее привлекательных свойств дедуктивного синтеза программ состоит в том, что поскольку программа извлекается из доказательства, то его корректность гарантируется правильностью доказательства. Таким образом, отпадает необходимость в верификации построенной программы: а правильность доказательства проверять значительно проще, чем правильность программы.

Основных методов извлечения программ из доказательств по существу два. Это методы реализуемости и нормализации. *Метод реализуемости* основан на семантиках типа реализуемости по Клини [Клини 52], идеи которых восходят еще к работе Колмогорова [Колмогоров 32]. В этих семантиках формулам сопоставляется их конструктивная расшифровка, или реализация. Общая схема такого сопоставления следующая:

1. Реализации атомарных формул считаются заданными (обычно при этом реализуемыми формулами оказываются формулы, истинные в некоторой модели, например в стандартной модели арифметики [Клини 52, Воронков 86, а, б, в, г, Плиско 76]).

2. Реализациями формулы $A \wedge B$ являются пары $\langle a, b\rangle$, такие, что a — реализация A, b — реализация B.

3. Реализациями формулы $A \vee B$ являются пары $\langle a, 0\rangle$, такие, что a — реализация A, и пары $\langle b, 1\rangle$, такие, что b — реализация B.

4. Реализациями формулы $A \supset B$ являются «алгоритмы», которые получив на вход произвольную реализацию формулы A, дают на выходе реализацию формулы B.

5. Реализациями формулы $(\exists x)\, A(x)$ являются пары $\langle t, a\rangle$, такие, что a — реализация $A(t)$.

6. Реализациями формулы $(\forall x)\, A(x)$ являются алгоритмы, которые получив на вход произвольный терм t, дают реализацию формулы $A(t)$.

Извлечение терма из доказательства с помощью семантик типа реализуемости осуществляется следующим образом. Вместе с каждым шагом доказательства строится элемент, реализующий доказанную на этом шаге формулу. После того, как построено доказательство формулы $(\exists x)\, A(x)$, рассматривается ее реализация. Она имеет вид $\langle t, a\rangle$, где t и есть искомый терм.

Нормализация доказательств была предложена Правицем [Правиц 65], но ее идеи восходят еще к процедуре устранения сечения [Генцен 35]. Процедура нормализации состоит из отдельных шагов — редукций выводов. Правило редукции применимо, когда в выводе вводится логическая связка (или квантор), которая сразу после этого удаляется. Для каждой связки имеются свои правила редукции. Например, правила $\wedge$-редукции выглядят следующим образом:

$$\dfrac{\dfrac{\begin{matrix}\Pi_1 & \Pi_2\\ A & B\end{matrix}}{A \wedge B}}{A} \Rightarrow \begin{matrix}\Pi_1\\ A\end{matrix} \qquad \dfrac{\dfrac{\begin{matrix}\Pi_1 & \Pi_2\\ A & B\end{matrix}}{A \wedge B}}{B} \Rightarrow \begin{matrix}\Pi_2\\ B\end{matrix}$$

Корректность процедуры нормализации следует из сильной теоремы нормализации [Правиц 71]: любой вывод нормализуем за конечное число шагов (т. е. приводится к такой форме, в которой ни одно правило редукции не применимо). Нормальная форма доказательства (в интуиционистском исчислении предикатов или интуиционистской арифметике) замк-

нутой формулы $(\exists x)\, A(x)$ имеет вид:

$$\frac{\begin{array}{c}\Pi \\ A(t)\end{array}}{(\exists x)\, Ax} \qquad (2)$$

Таким образом, для нахождения искомого терма t достаточно нормализовать вывод. На процедуре нормализации основаны системы дедуктивного синтеза программ, описанные в [Гоуд 80, Хагия 83].

В качестве математического базиса для теории синтеза программ могут служить и так называемые *теории типов*[1]) [Мартин-Лёф 82, Бизон 86, Феферман 78, де Бруин 80]. Одно из предназначений теории типов в программировании заключается в том, чтобы реализовать математические языки, которые одновременно являлись бы и языками программирования. Как отмечается в [Бизон 86], «трудная часть задачи, как мне кажется, состоит в построении подходящих языков, структура (синтаксис) которых отражает структуры, о которых мы хотим говорить естественным способом». Существующие теории типов хотя и позволяют записывать математические теоремы в достаточно компактной форме, но пока далеки от совершенства в том, что касается возможности простого чтения и записи доказательств.

Теории типов слишком сложны, чтобы мы могли более наглядно представить их здесь, поэтому заинтересованного читателя мы отсылаем к упомянутой литературе (см. также [Нордстрем 83, Кокан 85]).

Среди реализованных систем синтеза программ из доказательств существенно выделяются две — система Гоуда [Гоуд 80] и система Манны — Уолдингера [Манна 85, 86]. Первая осуществляет синтез полностью автоматически из готового доказательства. Во второй доказательство строится в диалоговом режиме и параллельно ему автоматически строится доказательство-программа. Остановимся на этих системах особо.

Перед тем как рассказать о системе Гоуда, рассмотрим некоторые недостатки процедуры нормализации вообще. Для описания программистски полезных типов данных требуется большое число специальных аксиом и правил вывода, а добавление таких аксиом или правил к дедуктивной системе может привести к потере корректности процедуры извлечения программы из доказательства (в том смысле, что нормализованное доказательство не будет иметь вид (2) и, таким образом, не даст терма). Корректность гарантируется, только

[1]) Связь между системами типов в математике и в языках программирования обсуждается в обзоре [Агафонов 82].

если в качестве аксиом используются формулы Харропа [1]) [Харроп 60], но для описания полезных типов данных таких аксиом всегда недостаточно. Для обращения с «нехарроповскими» аксиомами Гоуд ввел понятие леммы и правила редукции для лемм. Леммой называется любая аксиома вида $(\forall \bar{x})\, A(\bar{x})$, такая, что существует процедура γ_A, которая по любому набору $\bar{t}$ замкнутых термов (т. е. конкретных значений переменных $\bar{x}$) дает доказательство формулы $A(\bar{t})$, в котором все аксиомы — формулы Харропа. Правило редукции для лемм выглядит следующим образом:

$$\frac{(\forall \bar{x})\, A(\bar{x})}{A(\bar{t})} \Rightarrow \gamma_A(\bar{t}).$$

Наиболее важный частный случай — это когда A имеет вид $P(\bar{x}) \vee \neg P(\bar{x})$ и предикат P (атомарная формула) разрешим, т. е. по любым конкретным $\bar{t}$ мы можем эффективно узнать, истинна или нет формула $P(\bar{t})$. В этом случае процедура γ имеет вид:

$$\frac{P(\bar{t})}{P(\bar{t}) \vee \neg P(\bar{t})}, \text{ если } P(\bar{t}) \text{ и } \frac{\neg P(\bar{t})}{P(\bar{t}) \vee \neg P(\bar{t})}, \text{ если } \neg P(\bar{t}).$$

Кроме того, Гоуд использовал рекурсивные доказательства, т. е. доказательства, которые не завершены, но их можно эффективно раскручивать до полного доказательства. Кроме обычных правил нормализации выводов в системе Гоуда реализовано еще правило подрезки доказательств, которое было предложено Правицем [Правиц 71].

Перейдем к системе Манны — Уолдингера [Манна 85, 86]. Метод поиска вывода в ней основан на тех же идеях, что и метод матричных редукций [Правиц 70], а также упоминавшаяся выше пс-резолюция [Маррэй 82]. Чтобы избежать неконструктивности, присущей этим методам, расщепления формулы берутся по разрешимым подформулам. Пусть из формул $A_1, \ldots, A_n$ требуется вывести формулу $\exists x\, B(x)$. Тогда вначале составляется таблица следующего вида.

Посылки	Цель	Результат
$A_1, \ldots A_n$	$B(x)$	x

[1]) Формулами Харропа называются формулы, в которых любое вхождение дизъюнкции $\vee$ и квантора существования $\exists$ находится в левой части некоторой импликации или под отрицанием. Если в качестве аксиом использовать не только формулы Харропа, то нормализованный вывод формулы $(\exists x) A(x)$ не обязан, вообще говоря, иметь вид (2).

Затем эта таблица расщепляется на одну или несколько по правилам, аналогичным пс-резолюции. Если, например, в $B(x)$ позитивно входит формула $x \in t$, то мы можем заменить эту таблицу на новую

Посылки	Цель	Результат
$A_1, \ldots, A_n$	$B_1\ (y)$	**if** $x \in t$ **then** y **else** z
$A_1, \ldots, A_n$	$B_2\ (z)$	

где B_1 (соответственно B_2) получается из B заменой $x \in t$ на И, а x на y (соответственно $x \in t$ на Л, а x на z). Если в какой-либо строке таблицы вывод найден, т. е. цель совпадает с одной из посылок, или равна И, то ищется вывод для других строк. Когда вывод построен для всех строк, то, что стоит в графе «результат», является синтезированной программой.

Методы конструктивного вывода, но для подмножества исчисления высказываний используются в планировщике системы ПРИЗ с входным языком УТОПИСТ [Тыугу 70, 84, Минц 83]. Если в случае использования семантик типа реализуемости реализация формулы хранит информацию, достаточную для ее обоснования, и вся эта информация извлекается из доказательства, то в системе ПРИЗ из доказательства извлекается только часть информации, а другая часть задается явно в виде более или менее обычной программы. Аксиомы, с которыми работает планировщик системы, являются формулами исчисления высказываний вида B или вида $A_1 \wedge \ldots \wedge A_n \to B$, где B — атомарная формула (пропозициональная переменная), а формулы $A_1, \ldots, A_n$ или атомарные, или в свою очередь импликации вида $C_1 \wedge \ldots \ldots \wedge C_k \to D$, где $C_1, \ldots, C_k$, D — атомарные[1]). Вместе с каждой такой формулой задается ее реализация, которая в случае формулы $A_1 \wedge \ldots \wedge A_n \to B$ является программой, которая по реализации $A_1, \ldots, A_n$ дает реализацию B.

Для используемого в ПРИЗе класса формул существуют очень эффективные алгоритмы поиска вывода (см. [Волож 82, Диковский 85а, Канович 85]). Логические и алгоритмиче-

[1]) Такие формулы описывают связи между переменными $a_1, \ldots, a_n$, b исходной задачи. Например, формула $A_1 \wedge \ldots \wedge A_n \to B$ выражает вычислимость (значения) переменной b, соответствующей B, из значений $a_1, \ldots, a_n$, соответствующих $A_1, \ldots, A_n$.

ские аспекты использования формул указанного вида и их обобщений в синтезе программ рассматриваются в [Минц 82, 83, Диковский 84, 85а, 85б, Канович 85, 86]. Отметим, что работа планировщика скрыта от пользователя, которому надо только уметь описывать задачу на языке УТОПИСТ, а перевод такого описания в формулы, поиск вывода и синтез программы по готовому выводу осуществляются автоматически[1]).

ПРИЗ — одна из самых ранних систем логического программирования, хотя ее логический смысл в терминах исчисления высказываний был выявлен позднее. Плодотворная идея, воплощенная в системе ПРИЗ, получила развитие в ряде других отечественных систем автоматического синтеза программ. Система СПОРА с входным языком ДЕКАРТ [Бабаев 81, Лавров 82а, б] использует в планировщике систему поиска конструктивного вывода, основанную на подмножестве формул, используемых в Прологе. Специфика этого множества такова, что вместо обычного алгоритма обратной волны (как в Прологе) удобно использовать положительную гиперрезолюцию. В системе НУТ [Тыугу 85] средства ПРИЗа комбинируются с некоторыми средствами Пролога и объектно-ориентированного языка SMALLTALK.

ПРИЗ появился как подход к автоматизации решения некоторого класса задач, в которых важную роль играют описания связей между переменными (им соответствуют формулы указанного выше вида). Еще одна реализованная отечественная система автоматического синтеза программ СИНТЕЗ [Михайлов 83] является результатом логического анализа класса технологических задач, для описания которых выделен специальный класс формул логики первого порядка, допускающий достаточно эффективный алгоритм синтеза.

Синтез логических программ из конструктивных доказательств рассматривается в [Воронков 86]. У этого подхода по сравнению с традиционным подходом к извлечению программ из конструктивных доказательств имеются и преимущества и недостатки. Преимущества заключаются в том, что семантику для извлечения программ из доказательств можно объединить с теоретико-модельной семантикой Пролога на основе общего понятия модели. Вследствие этого можно писать смешанные программы, которые частично состоят из конструктивных доказательств, а частично — из хорновских дизъюнктов. Недостатки этого подхода вытекают из того, что для извлечения из доказательства эффективных программ скорее подходят языки функционального программирования. Для извлечения из доказательств программ-предикатов приходится

[1]) Отметим также реализацию ПРИЗа с помощью Пролога [Ломп 85].

или сужать понятие конструктивного доказательства, или жертвовать эффективностью извлекаемой программы.

Более подробно с проблематикой дедуктивного синтеза программ можно познакомиться по работам [Крайзель 75, 85, Непейвода 78, 79, 85, Лавров 82а, б, Минц 82, 83].

ЛИТЕРАТУРА

[Абрамсон 84] Abramson H. A prological definition of HASL, a purely functional language with unification based conditional binding expressions. — New Generation Computing, 2, 1984, no. 1, p. 3—35.

[Агафонов 82] Агафонов В. Н. Типы и абстракция данных в языках программирования (обзор). — В кн.: Данные в языках программирования. — М.: Мир, 1982, с. 265—327.

[Агафонов 84] Агафонов В. Н. Языки и средства спецификации программ (обзор). — В кн.: Требования и спецификации в разработке программ. — М.: Мир, 1984, с. 285—344.

[Апт 82] Apt K. R., van Emden M. H. Contribution to the theory of logic programming. — J. ACM, 29, 1982, No. 3, p. 841—862.

[Бабаев 81] Бабаев И. О., Новиков Ф. А., Петрушина Т. И. Язык Декарт — входной язык системы СПОРА. В кн.: Прикладная информатика, вып. 1, М.: Финансы и статистика, 1981, с. 35—72.

[Базисный Рефал 77] Базисный РЕФАЛ и его реализация на вычислительных машинах. — М.: ЦНИПИАСС, 1977.

[Барбути 85] Barbuti R., Bellia M., Levi G., Martelli M. LEAF: a language which integrate logic, equations and functions. — В кн.: [ЛП 85].

[Барвайз 75] Barwise J. Admissible sets and structures. — Berlin: Springer Verlag, 1975.

[Бареидрегт 81] Barendregt H. P. The lambda calculus, its syntax and semantics. — Amsterdam: North Holland, 1981. [Имеется перевод: Барендрегт Х. Ламбда-исчисление, его синтаксис и семантика. — М.: Мир, 1985].

[Баттани 83] Battani G., Mellani H. Interpreteure du language de programmation Prolog. — Groupe d'Intelligence Artificielle, Marseille-Lumini, 1983.

[Беллиа 82] Bellia M., Degano P., Levi G., The call by name semantics of a clause language with functions. — В кн.: [ЛП 82].

[Беллиа 86] Bellia M., Levi G. The relation between logic and functional languages: a survey. — J. Logic Programming, 3, 1986, No. 3, p. 217—236.

[Белов 86] Белов В. Н., Брановицкий В. И., Вершинин К. П., Гецко Л. Н., Довгялло А. М., Ефименко С. П., Колос В. В. ПРОЛОГ — язык логического программирования. — Прикладная информатика, 1986, вып. 1, с. 24—58.

[Бергман 81] Bergman M., Derensart P. Abstract data types and rewriting systems: application to the programming of algebraic data types in PROLOG. — Lect. Notes Comp. Sci., 112, 1981, p. 106—116.

[Бергман 81] Bergman M., Algebraic specifications: a constructive methodology in logic programming. — Lect. Notes Comp. Sci., 114, 1982, p. 91—108.

[Бибель 82] Bibel W. Automated theorem proving. — Braunschweig: Vieweg Verlag, 1982.

[Бибель 86] Bibel W. Predicative programming revisited. — Lect. Notes Comp. Sci., 215, 1986, p. 25—40.

[Бизон 86] Beeson M. J. Proving programs and programming proofs.— In: 7th Int. Congress on Logic, Methodology and Philosophy of Science, 1986, p. 51—82.

[Бишоп 67] Bishop E. Foundations of constructive analysis. — New York: McGraw Hill, 1967.

[Бишоп 85] Bishop E. Schisofrenia in contemporary mathematics. — Contemporary Mathematics, 39, 1985, p. 1—33.

[Бойер 84] Boyer R. S., Moore J. S. Proof-checking, theorem proving and program verification. — Contemporary Mathematics, 29, 1984, p. 119—132.

[Борщев 67] Борщев В. Б., Ефимова Е. Н. О сокращении перебора при синтаксическом анализе. — Научно-техническая информация, Сер. 2, 1967, 10, с. 27—33.

[Борщев 72] Борщев В. Б., Хомяков М. В. Схемы для функций и отношений. — Препринт семинара стран — членов СЭВ «Автоматическая обработка текстов на естественных языках», Ереван, 1972. В кн.: Исследования по формализованным языкам и неклассическим логикам. — М.: Наука, 1974, с. 23—49.

[Борщев 76] Борщев В. Б., Хомяков М. В. Клубные системы. — Научно-техническая информация. Сер. 2, 1976, 8, с. 3—6.

[Борщев 83] Борщев В. Б. Схемы на клубных системах и вегетативная машина. — Семиотика и информатика, 1983, вып. 22, с. 3—44.

[Борщев 85] Борщев В. Б. Семантика параметрических конструкций в логическом программировании. — Тезисы докладов школы-семинара «Синтез программ», Устинов, 1985, с. 18—20.

[Борщев 86а] Борщев В. Б. Пролог — основные идеи и конструкции. — Прикладная информатика, 1986, вып. 2, с. 49—76.

[Борщев 86б] Борщев В. Б. Логическое программирование. — Известия АН СССР, Техническая кибернетика, 1986, 2, с. 89—109.

[Боуэн 82] Bowen K. A., Kowalski R. A. Amalgamating language and metalanguage. — В кн.: [ЛП 82].

[Боуэн 83] Bowen D. L., Byrd L. M. A portable Prolog compiler. — Proc. of the Logic Programming Workshop, Portugal, 1983, p. 74—83

[Боуэн 85] Bowen K. A. Meta-level programming and knowledge representation. — New Generation Computing, 3, 1985, No. 4, p. 359—383.

[Бранохе 82] Bruynooghe M. The memory management of PROLOG implementations. В кн.: [ЛП 82], с. 83—98. [Имеется перевод: Бранохе М. Управление памятью в реализациях Пролога., см. наст. сб.]

[Бранохе 84а] Bruynooghe M., Pereira L. M. Deduction revision by intelligent backtracking. — В кн. [Реализация 84], с. 259—267.

[Бранохе 84б] Bruynooghe M. Garbage collection in Prolog interpreters. — В кн. [Реализация 84], с. 194—215.

[Братко 86] Bratko I. Prolog programming for artificial intelligence. L.: Addison-Wesley, 1986.

[Брэйлсфорд 85] Brailsford D. F., Duckworth R. J. The MUSE machine — architecture for structured dataflow computation. — New Generation Computing, 3, 1985, No. 2, p. 181—195.

[Бэкус 78] Backus J. Can programming be liberated from von Neumann style? A functional style and its algebra of programs. — Comm. ACM, 21, 1978, p. 613—641.

[Ван Ту Ле 85а] Van Tu Le. General failure of logic programs. — J. Logic Programming, 2, 1985, No. 2, p. 157—165.

[Ван Ту Ле 85б] Van Tu Le. Negation-as-failure rule for general logic programs with equality. — J. Logic Programming, 2, 1985, No. 4, p. 285—294.

[ван Эмден 76] van Emden M. H., Kowalski R. A. The semantics of predicate logic as a programming language — J. ACM, 23, 1976, No. 4, p. 733—743.

[ван Эмден 82] van Emden M. H., de Lucena G. J. Predicate logic as language for parallel programming. — В кн.: [ЛП 82], с. 189—198.

[ван Эмден 84] van Emden M. H. An interpreting algorithm for Prolog programs. — В кн.: [Реализация 84], с. 93—110.

[Васак 85] Vasak T., Potter J. Metalogical control for logic programs. — J. Logic Programming, 2, 1985, No. 3, p. 203—220.

[Венкен 84] Venken R. A Prolog Meta-interpreter for parallel evaluation and its application to source transformation and query optimization. — European Conf. on Artificial Intelligence, North Holland, 1984, p. 91—104.

[Визе 82] Wise M. J. EPILOG = PROLOG + data flow. Arguments for combining PROLOG with a data driven mechanism. — ACM SIGPLAN Notices, 17, 1982, No. 12, p. 80—84.

[Визе 84] Wise J. EPILOG: re-interpreting and extending Prolog for a multiprocessor environment. — В кн.: [Реализации 84], с. 352—368.

[Волож 82] Волож Б. Б., Мацкин М. В., Минц Г. Е., Тыугу Э. Х. Система ПРИЗ и исчисление высказываний. — Кибернетика, 1982, с. 63—70.

[Воробьев 86] Воробьев С. Г. О применении условных систем подстановок термов в верификации программ. — Программирование, 1986, 4, с. 3—14.

[Воронков 85] Воронков А. А. Один метод поиска доказательства.— Вычислительные системы, 1985, вып. 107, Новосибирск, с. 109—123.

[Воронков 86а] Воронков А. А. Способы выполнения программ в Σ-программировании. — Тезисы докладов Всесоюз. конф. «Применение методов мат. логики», секция «Представление знаний и синтез программ», Таллин, 1986, с. 51—53.

[Воронков 86б] Воронков А. А. Логические программы и их снитез. Новосибирск, Препринт Инстнтута математики СО АН СССР, 1986, № 23.

[Воронков 86в] Воронков А. А. Синтез логических программ. Новосибирск, Препринт Института математики СО АН СССР, 1986, № 24.

[Воронков 86г] Воронков А. А., Дегтярев А. И. Автоматическое доказательство теорем I. — Кибернетика, 1986, 3, с. 27—33.

[Вороиков 87а] Воронков А. А., Дегтярев А. И. Автоматическое доказательство теорем II. — Кибернетнка, 1987, 4, с. 88—96.

[Воронков 87б] Воронков А. А. Конструктивная истинность и конструктивные исчисления. — Доклады АН СССР, 1988, 300.

[Воронков 87в] Воронков А. А. Естественное исчисление для Σ-программ. — Вычислительные системы, 1987, вып. 120, Новосибирск, с. 14—23.

[Габбай 84] Gabbay D. M., Reyle U. N-PROLOG: an extension of PROLOG with hypothetical implications. I. — J. Logic Programming, 1, 1984, No. 4, p. 319—355.

[Габбай 85] Gabbay D. M. N-PROLOG: an extension of PROLOG with hypothetical implications. II. — J. Logic Programming, 2, 1985, No. 4, p. 251—283.

[Габбай 86] Gabbay D. M., Sergot M. J. Negation as inconsistency I.— J. Logic Programming, 3, 1986, No. 1, p. 1—35.

[Галлэр 83] Gallaire H. A study of PROLOG. In Computer Program Synthesis Methodologies, Reidel, 983, p. 173—212.

[Генцен 35] Gentzen G. Untersuchungen über das logische Schliesen. — Mathematische Zeitschrift, 39, 1935, p. 176—210, 405—431. [Имеется перевод: Генцен Г. Исследования логических выводов. — В кн.: Мат. теория логического вывода. — М.: Наука, 1967, с. 9—74.]

[Гергей 84] Gergely T., Szöts M., Cuttable formulas for logic programming. — В кн. [Симпознум 84], с. 299—310.

[Гёбел 86] Goebel R., Poole D. Graçefully adding negation and disjunction to Prolog. — В ки.: [ЛП 86], с. 635—641.

[Глушков 86] Глушков В. М. Кибернетика. Вопросы теории и практики. — М.: Наука, 1986.

[Гоген 79] Goguen J. A., Meseguer J. Some design principles and theory for OBJ-0, a language to express and execute algebraic specifications of programs. — Lec. Notes Comp. Sci., 75, 1979, p. 425—473.

[Гоген 84] Goguen J. A., Meseguer J. Equality, types, modules and (why not) generics for logic programming. — J. Logic Programming. 1, 1984, No. 2, p. 179—210.

[Гончаров 85] Гончаров С. С., Свирнденко Д. И. Σ-программирование. — Вычислительные системы, 1985, вып. 107.

[Гончаров 86] Goncharov S. S., Ershov Yu. L., Sviridenko D. I. Semantic programming. Information Processing, Congres, IFIP, North Holland, 1986, p. 1093—1100.

[Гордон 79] Gordon M., Milner R., Wadsworth C. Edinburgh LCF. — Lec. Notes Comp. Sci., 78, 1979.

[Гото 84] Goto A., Tanaka H., Moto-oka T. Highly parallel inference engine PIE — goal rewriting model and machine architecture.— New Generation Computing, 2, 1984, No. 1, p. 37—58.

[Гоуд 80] Goad C. A. Computational uses of the manipulation of formal proofs. — Stanford Univ. Department of Computer Sci. Technical Report no. STAN-CS-80-819, 1980.

[Даль 83] Dahl V. McCord M. Treating coordination in logic grammars. — American Journal of Computational Linguistics, 9, 1983, No. 2, p. 69—91.

[Дарлингтон 83] Darlington J., Reeve M. ALICE and the partial evaluation of logic programs.— 10th Annual Int. Symp. on Computer Architecture, 1983.

[Дарлингтон 85] Darlington J., Field A. J., Pull H. The unification of functional and logic languages. — В кн.: [ЛП 85].

[Дауни 76] Downy P. L., Sethi R. Correct computation rules for recursive languages. — SIAM J. Computing, 5, 1976, p. 378—401.

[Дебрей 86] Debray S. K., Warren D. S., Detection and optimization of functional computations in Prolog.— В кн.: [Конференция МЛП 86], с. 490—504.

[де Брунн 80] de Bruin N. G. A Survey of the project AUTOMATH.— In: H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism. — New York, Academic Press, 1980, p. 579—606.

[Дегтярев 81] Дегтярев А. И., Лялецкий А. В. Логнческий вывод в системе автоматнзации доказательств. — В кн.: Математические основы систем искусственного интеллекта, Киев, 1981, с. 3—11.

[Дегтярев 86а] Дегтярев А. И., Воронков А. А. Методы управления равенством в машинном доказательстве теорем. — Кибернетика, 1986 № 3, с. 34—41.

[Дегтярев 86б] Дегтярев А. И. Методы обращения с равенством для хорновских множеств.— В кн.: Методы алгоритмизации и реализации процессов решення интеллектуальных задач, Киев, 1986, с. 19—25.

[Дейкстра 76] Dijkstra E. W. A discipline of programming.— Englewood Cliffs: Prentice Hall, 1976. [Имеется перевод: Дейкстра Э. Дисциплина программирования. — М.: Мир, 1978.]

[Дерансар 83] Дерансар П. Компиляция Пролог-программ по алгебраическим спецификациям. — В кн.: Трансляция и оптимизация программ, Новосибирск: ВЦ СО АН СССР, 1983, с. 137—153.

[Дехтярь 85] Дехтярь М. И. О семантнке Рефал-программ. — В кн. Сложностные проблемы мат. логики. Калинин: КГУ, 1985, с. 36—41.

[Джаиессини 84] Gianessini F., Cohen J. Parcer generation and grammar manipulation using PROLOG's infinite trees. J. Logic Programming, 1, 1984, No. 3, p. 253—265.

[Джонсон 84] Johnsson T. Efficient compilation of lazy evaluation. — ACM SIGPLAN Notices, 19, 1984, No. 6, p. 58—69.

[Диковский 84] Диковский А. Я. Детерминированные вычислительные модели.— Известия АН СССР. Технческая кибернетика, 1984, № 5, с. 84—105.

[Диковский 85а] Диковский А. Я. Решение в линейное время алгоритмических проблем, связанных с синтезом ациклнческих программ.— Программирование, 1985, № 3.

[Диковский 85б] Диковский А. Я., Канович М. И. Вычислительные модели с разделяемыми подзадачами. — Известия АН СССР. Техническая кибернетика, 1985, № 5.

[Домёлки 83] Dömölki B., Szeredi P. PROLOG in practice. — Information Processing'83, 1983, p. 627—635. — [Имеется перевод: Домёлки Б., Середи Р. Практическое использование Пролога, см. наст. сб.]

[Дростен 84] Drosten K. Towards executable specification using conditional axioms. — Lect. Notes Comp. Sci., 166, 1984, p. 85—96.

[Ершов А. 78] Erschov A. P. Mixed computations in the class of recursive program schemata. — Acta Cibernetica, 1978, 4, No. 1, p. 19—23. [Имеется перевод: Ершов А. П. Смешанные вычнслення в классе рекурсивных схем программ. — Доклады АН СССР, 245, 1979, 5, с. 1041—1044.]

[Ершов А. 80] Ершов А. П. Смешанные вычисления: потенциальные применения н проблемы использования. — В кн.: Методы математической логики в проблемах искусственного нителлекта и систематическое программнрование. — Внльнюс, 1980, с. 26—55.

[Ершов А. 82] Ershov A. P. Mixed computation: potential applications and problems for study. — Theoretical Computer Science, 18, 1982.

[Ершов Ю. 83] Ершов Ю. Л. Динамическая логнка над допустимыми множествами. — Доклады АН СССР, 273, 1983, 5, с. 1045—1048.

[Ершов Ю. 85] Ершов Ю. Л. Σ-предикаты конечных типов над допустимыми множествами. — Алгебра н логика, 24, 1985, 5, с. 499—536.

[Ершов Ю. 86] Ершов Ю. Л. Язык Σ-выражений. — Вычислительные системы, вып. 116, Новосибирск, 1986, с. 3—10.

[Жорран 86] Jorrand Ph. Term rewriting as a basis for the design of a functional and paralled programming language. A case study: the language FP2. — Lecture Notes Comp. Sci., 232, 1986.

[Зикман 84] Siekman J. Universal unification. — Lecture Notes Comp. Sci., 170, 1984, p. 1—43.

[Йокота 83] Yokota M., Yamamoto A., Tohi K., Nishikawa H., Uchida S. The design and implementation of a personal sequential inference machine: PSI. — New Generation Computing, 1, 1983, p. 125—144.

[Кан 81] Kahn K. M. Uniform: a language based upon unification which unifies much of Lisp, Prolog and Act1. — Proc. 7th Int. Joint Conf. on Artificial Intelligence, 1981.

[Кан 84а] Kahn K. M. A primitive for the control of logic programs. — В кн.: [Симпозиум 84], p. 242—251.

[Кан 84б] Kahn K. M., Garisson M. How to implement Prolog on a LISP-machine. — В кн.: [Реализации 84], с. 117—134.

[Канеда 86] Kaneda Y., Tamura N., Wada K., Matsuda H., Kuo S., Maekawa S. Sequential Prolog machine PEK. — New Generation Computing, 4, p. 51—66.

[Канович 85] Канович М. И. Исчнсление функциональных и неявных зависимостей. — В кн.: Сложностные проблемы математической логики. — Калннин: КГУ, 1985, с. 52—61.

[Канович 86] Канович М. И. Квазиполиномиальные алгоритмы распознавания выполнимости и выводимости пропозициональных формул. — Доклады АН СССР, 290, 1986, № 2.

[Каплан 84] Kaplan S. Conditional rewrite rules. — Theoretical Computer Science, 33, 1984, No. 2—3, p. 139—174.

[Карделли 86] Cardelli K. Amber. — Lecture Notes Comp. Sci., 242, 1986, p. 21—47.

[Кащук 84] Kascuk P. A. Highly parallel Prolog interpreter based on generalized data flow model. — В кн.: [Конференция МЛП 84], с. 195—205.

[Кларк 77] Clark K. L., Sickel S. Predicate logic: a calculus for deriving programs. — 5th Int. Joint Conf. on Artificial Intelligence, Cambridge, Mass., 1977.

[Кларк 78] Clark K. L. Negation as failure. — In: Logic and data bases, N. Y., Plenum Press, 1978, p. 293—322.

[Кларк 81] Clark K. L., Gregory S. A relational language for parallel programming. — In: Functional Programming Languages and Computer Architecture, ACM, 1981.

[Кларк 82а] Clark K. L., McCabe F., Gregory S. IC-PROLOG language features. — В кн.: [ЛП 82], с. 253—266. [Имеется перевод: Кларк К. Л., Маккейб Ф., Грегори С. Особенности языка IC-PROLOG, см. наст. сб.]

[Кларк 82б] Clark K. L., McKeeman W. H., Sickel S. Logic program specification of numerical integration. — В кн.: [ЛП 82], с. 123—140. [Имеется перевод: Кларк К. Л., Маккиман У. Х., Зикель Ш. Спецификация численного интегрирования на языке логических программ. — См. наст. сб.]

[Кларк 82в] Clark K. L., McCabe F. G. Prolog, a language for implementing expert systems. — In: Machine Intelligence 10, Chichester: Ellis Horwood, 1982.

[Кларк 84а] Clark K. L., Gregory S. PARLOG: parallel programming in logic. — Research Report DOC 84/11. Department of Computing, Imperial College, London. 1984.

[Кларк 84б] Clark K. L., McCabe F. G. Micro-PROLOG: programming in logic. — Prentice Hall, 1984. [Имеется перевод: Кларк К. Л., Маккейб Ф. Г. Микро-Пролог: Введение в логическое программирование. — М.: Радно и связь, 1987.]

[Кларк 85] Clark K. L., Gregory S. Notes on the implementation of PARLOG. — J. Logic Programming, 2, 1985, No. 1, p. 17—42.

[Клещев 80] Клещев А. С. Реляцнонная модель вычислений. — Программирование, 1980, 4, с. 20—24.

[Клещев 81] Клещев А. С. Реляционный язык программирования и принципы его реализации на последовательной ЭВМ. — Программирование, 1981, 6, с. 45—53.

[Клини 52] Kleene S. K. Introduction to metamathematics. N. Y.: Van Nostrand Co., 1952. [Имеется перевод: Клини С. К. Введение в метаматематику. — М.: ИЛ, 1957.]

[Клоксин 81] Clocksin W. F., Mellish C. F. Programming in PROLOG. — Berlin: Springer Verlag, 1981. [Имеется перевод: Клоксин У., Меллиш К. Программирование на языке Пролог. — М.: Мир, 1987.]

[Ковальский 71] Kowalski R., Kuehner D. Linear resolution with selection function. — Artif. Intell., 2, 1971, p. 227—260.

[Ковальский 74] Kowalski R. A. Predicate logic as programming language. — In: Information Processing'74 (IFIP Congress 74), 1974. p. 569—574.

[Ковальскнй 79а] Kowalski R. A. Algorithm = logic + control. — Comm. ACM, 22, 1979, p. 424—431.

[Ковальский 79б] Kowalski R. A. Logic for problem solving. — North Holland, 1979.

[Ковальский 82] Kowalski R. A. The use of metalanguage to assemble object level programs and abstract programs. — В кн.: [Конференция МЛП 82], с. 1—9.

[Ковальский 83] Kowalski R. A. Logic programming. — Information Processing'83 (IFIP Congress 83), 1983, p. 133—145. [Имеется перевод: Ковальский Р. А. Логическое программирование. — См. наст. сб.]

[Кодиш 86] Kodish M., Shapiro E. Compiling OR-parallelism into AND-parallelism. — В кн.: [Конференция МЛП 86], с. 283—297.

[Кокан 85] Coquand T., Huet G. Constructions: a higher order proof system for mechanizing mathematics. — Lecture Notes Comp. Sci., 203, 1985, p. 151—185.

[Кокс 84] Cox P. T. Finding backtrack points for intelligent backtracking. — В кн.: [Реализации 84], с. 216—233.

[Кокс 86] Cox P. T., Pietrzykowski T. Incorporating equality into logic programming via surface deduction. — Annals Pure and Applied Logic, 31, 1986. No. 2—3, p. 177—190.

[Колмероэ 73] Colmerauer A., Kanoui H., Pasero R., Roussel P. Une systeme de communication homme-machine en Francais. — Groupe Intelligence Artificielle, Universite Aix Marseille, 1973.

[Колмероэ 78] Colmerauer A. Metamorphosis grammars. — Lect. Notes Comp. Sci., 63, 1978, p. 133—189.

[Колмероэ 82а] Colmerauer A. PROLOG and infinite trees. — В кн.: [ЛП 82], с. 231—252.

[Колмероэ 82б] Colmerauer A. An interesting subset of natural language. — В кн.: [ЛП 82], с. 45—66.

[Колмероэ 83] Colmerauer A., Kanoui H., van Caneghem M. Prolog, bases theoriques et developpements actuels. — Technique et Science Informatiques, 1983, No. 4, p. 277—311. [Имеется перевод: Колмероэ А., Кануи А., ван Канегем М. Пролог: теоретические основы и современное развитие. См. наст. сб.]

[Колмогоров 32] Kolmogoroff A. N. Zur Deutung der intuitionistische Logic. — Mathemathische Zeitschrift, 35, 1932, p. 58—65. [Имеется перевод: Колмогоров А. Н. К толкованию интунционистской логики. — В кн.: Колмогоров А. Н. Избранные труды. Математика и механика. — М.: Наука, 1985, с. 142—148.]

[Коморовский 82] Komorowski H. J. QLOG — The programming environment for PROLOG in LISP. — В кн.: [ЛП 82], с. 315—323.

[Кондратьев 86] Кондратьев Н. В., Руденко В. П., Феодоритова Е. В. Мобильная реалнзация языка программирования Микро-Пролог. — В кн.: Информатика и вычнслнтельная техника. Тез. докл. всесоюз. семинара. М.: Наука, 1986, с. 38.

[Констейбл 71] Constable R. L. Constructive mathematics and automatic program writers. — Information Processing'71 (IFIP Congress 71), North Holland, 1972, p. 229—233.

[Конференция МЛП 82] Proc. 1st Int. Logic Programming Conference, Marseille, 1982.

[Конференция МЛП 84] Proc. 2nd Int. Logic Programming Conference, Uppsala, 1984.

[Конференция МЛП 86] Proc. of the 3rd International Conf. on Logic Programming. — Lect. Notes Comp. Sci., 225, 1986.

[Конференция ЯЛП 85] Logic Programming'85. Proceedings. — Lecture Notes Comp. Sci., 221, 1985.

[Кораблин 74] Кораблин Ю. П., Кутепов В. П., Фальк В. Н. Исчисление функциональных схем. — В кн.: Цифровая вычислительиая техника н программнрование. М.: Сов. радно, 1974, № 8, с. 22—34.

[Крайзель 75] Kreisel G. Some uses of proof theory for finding computer programs. — Colloque Intern. de Logique, 1975. Paris: CNRF, 1977, p. 123—134. [Имеется перевод: Крайзель Г. Некоторые приложения теории доказательств к поиску программ для ЭВМ. — В кн.: Крайзель Г. Исследования по теории доказательств. М.: Мир, 1981, с. 239—256.]

[Крайзель 85] Kreisel G. Proof theory and synthesis of programs: potential and limitations.— Lecture Notes Comp. Sci., 203, 1985, p. 136—150.

[Крипке 75] Kripke S. Outline of a theory of truth. — J. Philos. 72, 1975, p. 690—716.

[Крицкий 86] Крнцкий С. П. Индуктивные определения в логических спецификациях. — Вычислительные системы, вып. 114, Новосибирск, 1986, с. 40—58.

[Курсаве 86] Kursawe P. How to invent a Prolog machine. — В кн.: [Конференция МЛП 86], с. 134—148.

[Кутепов 75] Кутепов В. П. Исчисление функциональных схем и параллельные алгоритмы. — Программирование, 1975, 4, с. 3—11.

[Кучеров 85] Кучеров Г. А. Системы подстановок термов. — Новосибирск: ВЦ СО АН СССР, Препринт 601, 1985.

[Лавленд 78] Loveland D. W. Automated theorem proving: a logical basis. — Amsterdam: North Holland, 1978.

[Лавров 78] Лавров С. С., Силагадзе Г. С. Автоматическая обработка данных. Язык Лисп и его реалнзация. — М.: Наука, 1978.

[Лавров 82а] Лавров С. С., Залогова Л. А., Петрушина Т. И. Прииципы планированнн решения задач в системе автоматического синтеза программ. — Программирование, 1982, 3, с. 35—40.

[Лавров 82б] Лавров С. С. Синтез программ. — Кибернетика, 1982, с. 11—16.

[Леви 83] Levi G., Pegna A. Top-down mathematical semantics and symbolic execution. — RAIRO Inform. Theor., 17, 1983, p. 55—70.

[Леви 86а] Levy J. A GHC abstract machine and instruction set. — В кн.: [Конфереицня МЛП 86], с. 157—171.

[Леви 86б] Levy J. Shared memory execution of commited choice languages. — В ки.: [Конференция МЛП 86], с. 298—312.

[Лескан 83] Lescanne P. Computer experiments with the REVE term rewriting system generator. — Proc. ACM Symp. on Principles of Programming Languages, 1983.

[Ллойд 84а] Lloyd J. W. Foundations of logic programming. — Berlin: Springer Verlag, 1984.

[Ллойд 84б] Lloyd J. W.. Topor R. W. Making PROLOG more expressive. — J. Logic Programming, 1, 1984. No. 3, p. 225—240.

[Ломп 85] Ломп А. Структурный синтез программ с помощью системы Пролог. — Известия АН ЭССР, физ.-мат., 34, 1985, 2, с. 125—132.

[ЛП 82] Logic Programming. — N. Y.: Academic Press, 1982. [Имеется перевод пяти статей в наст. сб.]

[ЛП 86] Logic Programming: functions, relations and equations. — Englewood Cliffs: Prentice Hall, 1986.

[Маккорд 86] McCord M. Design of a Prolog-based machine translation system. — В кн.: [Конфереицня МЛП 86], с. 350—374.

[Мальцев 65] Мальцев А. И. Алгоритмы и рекурсивные функции. — М.: Наука, 1965.

[Маниа 74] Manna Z. Mathematical theory of computation. — L.: McGraw-Hill, 1974. ch. 5. [Имеется перевод: Манна З. Теория неподвижных точек программ. — В кн.: Кибернетический сборник, вып. 15, М.: Мир, 1978, с. 38—100.]

[Манна 85] Manna Z., Waldinger R. The logical basis for computer programming. Vol. 1: Deductive reasoning. L.: Addison-Wesley, 1985.

[Манна 86] Manna Z., Waldinger R. Special relations in automated deduction. — J. A. C. M., 33, 1986, N1, p. 1—59.

[Маррэй 82] Murray N. V. Completely non-clausal theorem-proving. — Artificial Intelligence, 18, 1982, No 1, p. 67—85.

[Мартин-Лёф 71] Martin-Löf P. Haupsatz for the intuitionistic theory of iterated inductive definitions. — 2nd Scandinavian Logic Symp., 1971, p. 179—216.

[Мартин-Лёф 82] Martin-Löf P. Constructive mathematics and computer programming. — Proc. 6th Int. Congress on Logic, Methodology and Philosophy of Sci., 1982, p. 153—178.

[Мартьянов 82] Мартьянов В. И. Методы задания и частичного построения теории на ЭВМ. — Кибернетика, 1982, 6, с. 102—110.

[Мартьянов 87] Мартьянов В. И., Николенко А. В. Язык логического программирования ПИФОР. — В кн.: Вычислительные системы, вып. 122, Новосибирск: ИМ СО АН СССР, 1987.

[Маслов 64] Маслов С. Ю. Обратный метод установления выводимости в классическом исчислении предикатов. — Доклады АН СССР, 159, 1964, с. 17—20.

[Маслов 83] Маслов С. Ю., Минц Г. Е. Теория поиска вывода и обратный метод. — В кн.: [Чень 73] (русскнй перевод), с. 291—314.

[Массер 80] Musser D. R. Abstract data type specification in the AFFIRM system. — IEEE Trans., SE-6, 1980, No. 1, p. 24—32. [Имеется перевод: Массер Д. Р. Спецификация абстрактных типов данных в системе AFFIRM. — В кн.: Требования и спецификацин в разработке программ. — М.: Мир, 1984, с. 199—222.]

[Массер 82] Musser D. R., Kapur D. Rewrite rule theory and abstract data type analysis. — Lecture Notes Comp. Sci., 144, 1982, p. 77—90.

[Матиясевич 70] Матиясевич Ю. В. Диофантовость перечнслимых множеств. — Доклады АН СССР, 191, 1970, с. 279—282.

[Мейра 84] Meira S. M. Optimized combinatorial code for applicative language implementation.— Lecture Notes Comp. Sci., 167, 1984, p. 206—216.

[Микрофт 84] Mycroft A. Logic programs and many-valued logic. — Lecture Notes Comp. Sci., 166, 1984, p. 274—286.

[Минц 82] Минц Г. Е. Снитез программ. — Таллии: АН ЭССР. Препринт, 1982.

[Минц 83] Миниц Г. Е., Тыугу Э. Х. Обоснование структурного синтеза программ. — В кн.: Автоматический синтез программ. Таллин: ИК АН ЭССР, 1983, с. 5—40.

[Минц 86] Минц Г. Е. Полное исчнсление для чистого Пролога. — Известия АН ЭССР, фнз.-мат., 35, 1986, 4, с. 367—380.

[Мнхайлов 83] Михайлов В. Ю. Об одной системе автоматического синтеза программ. — В кн.: Автоматический синтез программ, Таллин: ИК АН ЭССР, 1983, с. 71—88.

[Монтейро 84] Menteiro L. A proposal for distributed programming in logic. — В кн.: [Реализации 84].

[Мосс 86] Moss Ch. Cut and paste — defining the impure primitives of Prolog. — Lecture Notes Comp. Sci., 225, 1986, p. 686—694.

[Мур 82] Moor I. W. An applicative compiler for a parallel machine. — ACM SIGPLAN Notices, 17, 1982, No 6 p. 284—292.

[Накагава 86] Nakagawa H. Prolog program transformation system.— In: IFIP WG2.1 Working Conf. on Program Specifications and Transformations. 1986.

[Накамура 84] Nakamura K. Associative evaluation of Prolog programs. — В кн.: [Реализация 84], с. 135—146.

[Нейман 86] Нейман В. С. Организация быстрого поиска в множестве термов. — Известия АН СССР. Техиическая кибернетика, 1986, 2, с. 196—199.

[Нейман 86] Нейман В. С. Система PROVE — реализация метода выделения подцелей. — Ленинград: ИТА АН СССР (Алгоритмы небесной механикн 50), 1986.

[Непейвода 78] Непейвода Н. Н. Соотношение между правилами естественного вывода и операторами алгоритмическнх языков высокого уровня. — Доклады АН СССР, 239, 3, 1978.

[Непейвода 79] Непейвода Н. Н. Об одном методе построенння правильной программы из правильных подпрограмм. — Программирование, 1979, 1, с. 15—25.

[Непейвода 82] Непейвода Н. Н., Свириденко Д. И. К теории синтеза программ. — Труды Института математики СО АН СССР, т. 2, 1982, с. 159—175.

[Непейвода 85] Непейвода Н. Н. Математическая теория синтеза программ (обзор). — В кн.: Синтез программ. — Устинов, 1985, с. 4—8.

[Нордстрем 83] Nordström B., Petersson K. Types and specifications. — Information Processing 83, IFIP Congress, 1983. p. 915—920.

[Нэш 85] Naish L. Automating control of logic programs. — J. Logic Programming, 2, 1985, N3, p. 167—183.

[Нэш 86а] Naish L. Negation and quantifiers in NU-PROLOG. — В кн.: [Конференция МЛП 86], с. 624—636.

[Нэш 86б] Naish L., Negation and control in Prolog. — Lecture Notes Comp. Sci., 238, 1986.

[О'Доннелл 77] O'Donnell M. J. Computing in systems described by equations. — Lecture Notes Comp. Sci., 58, 1977.

[Онаи 85] Onai R., Aso M., Shimizu H., Masuda K., Matsumoto A. Architecture of a reduction-based parallel inference machine: PIM-R. — New Generation Computing, 3, 1983, p. 197—228.

[Парсэй 83] Parsaye K. Database management, knowledge base management and expert systems developed in Prolog. — В кн.: [Семинар 83].

[Патерсон 78] Paterson M., Wegman M. N. Linear unification. — J. Computer and System Sci., 16, 1978.

[Перейра 80] Pereira F., Warren D. Definite clause grammars for language analysis — a survey of the formalism and a comparison with augmented transition networks. — Artificial Intelligence, 13, 1980, p. 231—278.

[Перейра 84] Pereira L. M. Logic control with logic. — В кн.: [Реализация 84], с. 177—194.

[Перейра 86] Pereira L. M., Monteiro L., Cunha J., Aparicio J. N. Delta Prolog: a distributed backtracking extension with events. — В кн.: [Конференция МЛП 86], с. 69—83.

[Плейстид 82] Plaisted D. A. A simplified problem reduction format.— Artificial Intelligence, 2, 1982, N. 1, p. 227—261.

[Плиско 76] Плиско В. Е. Некоторые варианты понятия реализуемости для предикатных формул.— Доклады АН СССР, 226, 1976, 1, с. 61—64.

[По 84] Poe M. D., Nasr R., Potter J., Slinn J. A KWIC bibliography on Prolog and logic programming. — J. Logic Programming, 1, 1984, N1, p. 81—142.

[Пост 43] Post E. L. Formal reductions of the general combinatorial decision problem. — American J. Math., 65, 1943, p. 197—215.

[Правиц 65] Prawitz D. Natural deduction. — Stockholm: Almquist and Wicksell, 1965.

[Правиц 70] Prawitz D. A proof procedure with matrix reduction. — Lecture Notes Math., 125, 1970. 207—214.

[Правиц 71] Prawitz D. Ideas and results in proof theory. — 2nd Scandinavian Logic Symp., 1971, p. 235—308.

[Пролог 88] Язык Пролог в вычислительных системах пятого поколения. — Пер. с англ. М.: Мир, 1988.

[Пролог-СМ 85] Система программирования Пролог-СМ. Описание языка. Руководство программиста. — Ленинград: ЛЭТИ, (Индекс ЦФАП 516027.00138—01 33 01), 1985.

[Проскурин 83] Проскурин А. В. Построение и оптимизация программ средствами языка алгебранческих спецнфнкаций. — В кн.: Материалы Всесоюз. Снмпоз. «Оптимизация и преобразования программ». Новосибирск: ВЦ СО АН СССР, ч. 2, 1983, с. 52—62.

[Реализация 84] Implementations of PROLOG. — N. Y.: Ellis Horwood, 1984.

[Редди 85] Reddy U. S. On the relationship between logic and func-

[Робинсон 65] Robinson J. A. A machine-oriented logic based on the resolution principle. — J. A. C. M., 12, 1965, p. 23—41. [Имеется перевод: Робинсон Дж. Машинно-ориентированная логика, основанная на принципе резолюции. — В кн.: Кибернетический сборник, вып. 7, 1970, с. 194—218.]

[Робинсон 82] Robinson J. A., Sibert E. E. LOGLISP: motivations, design and implementation. — В кн.: [ЛП 82], с. 299—314. [Имеется перевод: Робинсон Дж., Зиберт Э. Логлисп: Мотивировка, основные возможности и реализация. — См. наст. сб.]

[Робинсон 83] Robinson J. A. Logic programming — past, present and future. — New Generation Computing, 1, 1983, p. 107—124. [Имеется перевод: Робинсон Дж. Логическое программирование — прошлое, настоящее и будущее. — См. наст. сб.]

[Робинсон 86] Robinson J. A. The future of logic programming. — In Information Processing 86, IFIP Congress, 1986.

[Робинсон Я. 86] Robinson Ian. A Prolog processor based on a pattern matching device. — В кн.: [Конференция МЛП 86], с. 172—179.

[Росси 86] Rossi G. Uses of Prolog in implementation of expert systems. — New Generation Computing, 4, 1986, p. 321—329.

[Сазонов 86] Сазонов В. Ю., Свириденко Д. И. Денотационная семантика языка Σ-выражений. — В кн.: Вычислительные системы, вып. 114, Новосибирск: ИМ СО АН СССР, 1986, с. 16—34.

[Сато 84] Sato T., Tamaki H. Transformational logic program synthesis. — Proc. Int. Conf. on Fifth Generation Computer Systems, 1984, p. 195—201.

[Свириденко 86] Свириденко Д. И. Проектирование Σ-программ. θ-оцениваемость. — В кн.: Вычислительные системы, вып. 114, Новосибирск: ИМ СО АН СССР, 1986, с. 59—83.

[Семинар 80] Proc. Int. Workshop on Logic Programming, Debrecen, 1980.

[Семинар 83] Proc. Int. Workshop on Logic Programming, Algavre, 1983.

[Симмонс 82] Simmons R. F. A narrative schema in procedural logic. — В кн.: [ЛП 82], с. 67—80.

[Симпозиум 84] Int. Symp. on Logic Programming. Atlantic City, 1984.

[Слэйгл 74] Slagle J. R. Automated theorem-proving for theories with simplifiers, commutativity and associativity. — J. ACM, 21. 1974, n. 4, p. 662—642.

[Смальян 62] Smullyan R. M. Theory of formal systems. — Princeton: Princeton Univ. Press, 1962. [Имеется перевод: Смальян Р. Теория формальных систем. — М.: Наука, 1981.]

[Смоян 85] Смоян А. М. Ридэр — логический язык для разработки систем обработки данных. — В кн.: Всесоюз. конф. по прикладной логике. Новосибирск: ИМ СО АН СССР, 1985, с. 196—199.

[Степанов 81а] Степанов А. М. Фреймы и смешанные вычисления.— Новосибирск: ВЦ СО АН СССР, Препринт 297, 1981.

[Степанов 81б] Степанов А. М. Экспериментальная система программирования. — Новосибирск: ВЦ СО АН СССР, Препринт 305, 1981.

[Стерлинг 86] Sterling L., Codish M. Pressing for parallelism. A PROLOG program made concurrent. — J. Logic Programming, 3, 1986, N1, p. 75—92.

[Стикель 84] Stickel M. E. A Prolog technology theorem prover. — New Generation Computing, 2, 1984, N4, p. 371—383.

[Стикель 86] Stickel M. E. An introduction to automated deduction. — Lecture Notes Comp. Sci., 232, 1986, p. 75—132.

[Такеути 83] Takeuchi I., Okuno H., Ohsato N. TAO — a harmonic mean of Lisp, Prolog and Smalltalk. — ACM SIGPLAN Notices, 18, 1983, 7, p. 65—74.

[Такеути 86] Takeuchi A., Furukawa K. Parallel Logic programming languages. — Lecture Notes Comp. Sci., 225, 1986, p. 242—254.

[Тёрнер 79] Turner D. A. New implementation techniques for applicative languages. — Software Practice and Experience, 9, 1979, p. N1, p. 31—49.

[Тёрнер 84] Turner D. A. Functional programs as executable specifications. — Philos. Trans. Royal Soc. London, A312, 1984, p. 363—383.

[Тик 84] Tick E., Warren D. Towards a pipelined Prolog processor. — New Generation Computing, 2, 1984, p. 323—345.

[Тыугу 70] Тыугу Э. Х. Решение задач на вычислительных моделях.— Ж. вычисл. матем. и матем. физики, 10, 1970, 5.

[Тыугу 84] Тыугу Э. Х. Концептуальное программирование. М.: Наука, 1984.

[Тыугу 85] Тыугу Э. Х., Мацкин М. Б., Пеньям Я. Э., Эомойс П. В. Объектно-ориентированный язык НУТ. — В кн.: Прикладная информатика, вып. 2(9), М.: Финансы и статистика, 1985.

[Уеда 86а] Ueda K. Guarded Horn Clauses. — Lecture Notes Comp. Sci., 221, 1986, p. 168—179.

[Уеда 86б] Ueda K. Making exhaustive search programs deterministic. — В кн.: [Конференция МЛП 86], с. 270—282.

[Уоллес 84] Wallese H. Communicating with databases in natural language. — Chichester: Ellis Horwood, 1984.

[Уоррен 77] Warren D. Implementing Prolog — compiling predicate logic programs. — D. A. I. Research Report N39, 40. Edinburgh: University of Edinburgh, 1977.

[Уоррен 82а] Warren D. Higher-order extentions to Prolog — are they needed? In: Machine Intelligence, 10, Chichester: Ellis Horwood, 1982, p. 441—454.

[Уоррен 82б] Warren D., Pereira F. An efficient easily adaptable system for interpreting natural language queries. — American J. Computational Linguistics, 8, 1982, N3—4, p. 110—122.

[Уоррен 83] Warren D. An abstract Prolog instruction set. — Tech. note 309, SRI Intern., 1983.

[Феферман 78] Feferman S. Constructive theories of functions and classes. — Logic Colloquium'78. Amsterdam: North Holland, 1979, p. 159—224.

[Фильгуэрос 84] Filguerias M. A Prolog interpreter working with infinite terms. — В кн.: [Реализации 84], с. 250—258.

[Фиттинг 85] Fitting M. A Kripke — Kleene semantics for logic programs. — J. Logic Programming, 2, 1985, N4, p. 295—312.

[Фиттинг 86] Fitting M. Notes on the mathematical aspects of Kripke's theory of truth.— Notre Dame. J. Formal Logic, 27, 1986, N 1, p. 75—77.

[Фланнаган 86] Flannagan T. The consistency of negation as failure.— J. Logic Programming, 3, 1986. N2, p. 93—114.

[Фогельхольм 84] Fogelholm R. Exter Prolog — some thoughts on Prolog design by a Lisp user. — В кн.: [Реализации 84] с. 111—116.

[Фрибург 84] Fribourg L. Oriented equational clauses as a programming language. — J. Logic Programming, 1, 1984, N2, p. 165—177.

[Функциональные языки 85] Functional programming languages and computer architecture. — Lecture Notes Comp. Sci., 201, 1985.

[Футо 83] Futo I., Szeredi J. T-Prolog user manual. Version 4.2. SZKI, Budapest, 1983.

[Фучи 86] Fuchi K.. Furukawa K. The role of logic programming in the Fifth Generation Computer Project. — Lecture Notes Comp. Sci., 225, 1986, p. 1—24.

[Фэй 79] Fay M. First-order unification in an equational theory. — Proc. 4th Workshop on Automated Deduction, 1979, p. 161—171.

[Хагия 83] Hagiya M. A proof description language and its reduction system. — Publ. Res. Inst. Math. Sci., 19, 1983, 1, p. 237—261.

[Хагия 84] Hagiya M., Sakurai T. Foundations of logic programming based on inductive definitions. — New Generation Computing, 2, 1984, N1, p. 59—77. [Имеется перевод: Хагия М., Сакураи Т. Основы логического программирования, базирующиеся на индуктивных определеннях. — См. наст. сб.]

[Халим 86] Halim Z. A data-driven machine for OR-parallel evaluation of logic programs. — New Generation Computing, 4, 1986, 1, p. 5—33.

[Ханссон 82] Hansson A., Haridi S., Tärnlund S. Properties of a logic programming language. — В кн.: [ЛП 82], с. 267—280. [Имеется перевод: Ханссон А. и др. Свойства одного языка логического программирования.— См. наст. сб.]

[Хариди 81] Haridi S. Logic programming based on natural deduction system. — Techn. Report TRITA-OS-8104, Royal Inst. of Technology, Stockholm, 1981.

[Хариди 84] Haridi S., Sahlin D. An abstract machine for LPLO. — Techn. Report CSALAB 1984—1004, Royal Inst. of Technology, Stockholm, 1984.

[Харроп 60] Harrop R. Concerning formulas of the types $A \to B \vee C$, $A \to (Ex)B(x)$ in intuitionistic formal system. J. Symbolic Logic, 25, 1960, n1, p. 27—32.

[Хеллерштейн 86] Hellerstein L., Shapiro E. Implementing parallel algorithms in Concurrent Prolog. The MAXFLOW experiment. — J. Logic Programming, 3, 1986, N2, p. 157—184.

[Хендерсон 80] Henderson P. Functional programming. Application and implementation. — Englewood Cliffs: Prentice Hall Int., 1980. [Имеется перевод: Хендерсон П. Функциональное программирование. Применение и реализация. М.: Мир, 1983.]

[Хеншен 83] Henschen L., McCune W. Semantic paramodulation for Horn sets. — Proc. 8th Int. Joint Conf. Artif. Intelligence, 1983, p. 902—908.

[Херменегилдо 86] Hermenegildo M. V., Nasr R. I. Efficient management of backtracking in AND-parallelism. — В кн.: [Конференция МЛП 86], с. 40—54.

[Хоггер 84] Hogger C. J. Introduction to logic programming. — L.: Academic Press, 1984. [Готовится перевод: Хоггер К. Введение в логическое программирование. М.: Мир, 1988.]

[Хоффман 82] Hoffman C. M., O'Donnell M. J. Programming with equations. — ACM Trans. Progr. Languages and Systems, 4, 1982, p. 83—112.

[Хьюитт 72] Hewitt C. Description and theoretical analysis (using schemata) of PLANNER: a language for proving theorems and manipulating models in a robot. — Techn. Report AI Lab., Massachusets Inst. Technology, Cambridge, 1972.

[Чен 73] Chang C. C., Keisler H. G. Model theory. — Amsterdam: Horth Holland. 1973. [Имеется перевод: Кейслер Г., Чен Ч. Теория моделей. М.: Мир, 1977.]

[Чень 73] Chang C.-L., Lee R. Symbolic logic and mechanical theorem proving. — N. Y.: Academic Press, 1973: [Имеется перевод: Чень Ч., Ли Р. Математическая логика и автоматическое доказательство теорем. М.: Наука, 1983.]

[Шапиро 83] Shapiro E. A subset of Concurrent Prolog and its interpreter. — ICOT Techn. Report, TR-003, 1983.

[Шепердсон 84] Shepherdson J. C. Negation as failure: a comparison of Clark's completed data base and Reiter's closed world assumption. — J. Logic Programming, 1, 1984, N1, p. 51—79.

[Шепердсон 85] Shepherdson J. C. Negation as failure, II. — J. Logic Programming, 2, 1985, N3, p. 185—202.

[Шётц 82] Szötz M. A comparison of two logic programming languages: a case study. — В кн.: [Конференция МЛП 82], с. 41—51.

[Эндрюс 81] Andrews P. B. Theorem proving via general matings. — J. A. C. M., 28, 1981, N2, p. 193—214.

[Эндрюс 84] Andrews P. B., Miller D. A., Cohen E. L., Pfenning F. Automating higher-order logic. Automated theorem proving: after 25 years. Contemporary Mathematics, 29, 1984, p. 169—192.

[Юло 80] Hullot J.-M. Canonical forms and unification. — Lecture Notes Comp. Sci. 87, 1980, p. 318—334.

[Юэт 75] Huet G. A unification algorithm for typed λ-calculus. — Theoretical Comp. Sci., 1, 1975, p. 27—55.

[Юэт 80] Huet G., Oppen D. C. Equations and rewrite rules: a survey. — In: Formal language theory. — N. Y.: Academic Press, 1980, p. 349—406.

СОДЕРЖАНИЕ

УВАЖАЕМЫЙ ЧИТАТЕЛЬ!

Ваши замечания о содержании книги, ее оформлении, качестве перевода и другие просим присылать по адресу: 129820, Москва, И-110, ГСП, 1-й Рижский пер., д. 2, издательство «Мир».

Научное издание

ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

Заведующий редакцией чл.-корр. АН СССР В. И. Арнольд
Зам. зав. редакцией А. С. Попов
Ст. научный редактор А. А. Бряндинская
Научный редактор М. В. Хатунцева
Мл. научный редактор Л. А. Королева
Художник Н. Я. Вовк
Художественный редактор В. И. Шаповалов
Технический редактор А. Л. Гулина
Корректор С. С. Суставова

ИБ № 6283

Сдано в набор 10.12.87. Подписано к печати 07.06.88
Формат 60×90¹/₁₆. Бумага книжно-журнальная. Печать высокая. Гарнитура литературная. Объем 11.50 бум. л. Усл. печ. л. 23.00. Усл. кр.-отт. 23.00. Уч.-изд. л. 22.94. Изд. № 1/5332. Тираж 15 000 экз. Зак. 1534. Цена 2 руб.

Издательство «МИР» В/О «Совэкспорткнига» Государственного комитета СССР по делам издательств, полиграфии и книжной торговли. 129820, ГСП, Москва, И-110, 1-й Рижский пер., 2.

Отпечатано с набора Ленинградской типографии № 2 головного предприятия ордена Трудового Красного Знамени Ленинградского объединения «Техническая книга» им. Евгении Соколовой Союзполиграфпрома при Государственном комитете СССР по делам издательств, полиграфии и книжной торговли. 198052, г. Ленинград, Л-52, Измайловский проспект, 29 в Ленинградской типографии № 4 ордена Трудового Красного Знамени Ленинградского объединения «Техническая книга» им. Евгении Соколовой Союзполиграфпрома при Государственном комитете СССР по делам издательств, полиграфии и книжной торговли. 191126, Ленинград, Социалистическая ул., 14.