Ю. В. КАПИТОНОВА
А. А. ЛЕТИЧЕВСКИЙ

МАТЕМАТИЧЕСКАЯ ТЕОРИЯ ПРОЕКТИРОВАНИЯ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

Ю. В. КАПИТОНОВА
А. А. ЛЕТИЧЕВСКИЙ

МАТЕМАТИЧЕСКАЯ ТЕОРИЯ ПРОЕКТИРОВАНИЯ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

МОСКВА "НАУКА"
ГЛАВНАЯ РЕДАКЦИЯ
ФИЗИКО-МАТЕМАТИЧЕСКОЙ ЛИТЕРАТУРЫ
1988

ББК 22.18
К20
УДК 519.6

Капитонова Ю.В., Летичевский А.А. **Математическая теория проектирования вычислительных систем.** – М.: Наука. Гл. ред. физ.-мат. лит., 1988. – 296 с. ISBN 5-02-013777-4

Излагаются основы математического аппарата и современных методов проектирования систем преобразования информации: аппаратуры вычислительных машин, программ и программных систем, систем управления и обработки данных, основанных на применении средств вычислительной техники.

Первая часть посвящена обзору основных математических моделей вычислительных систем. Вторая часть содержит изложение практических методов проектирования различного типа систем – аппаратуры ЭВМ, последовательных и параллельных программ, компонент общесистемной математики.

Для математиков и инженеров – разработчиков аппаратных и программных средств, а также для студентов и аспирантов в области информатики и вычислительной техники.

Табл. 5 Ил. 43 Библиогр. 93 назв.

Рецензент доктор физико-математических наук *В.Е. Котов*

К $\frac{1702070000\text{-}116}{053\,(02)\text{-}88}$ 22-88

ISBN 5-02-013777-4

ОГЛАВЛЕНИЕ

Светлой памяти нашего дорогого учителя
Виктора Михайловича Глушкова
посвящается

ПРЕДИСЛОВИЕ

В этой книге под вычислительной системой понимается любая техническая система преобразования информации, поведение которой может быть описано алгоритмически. Таким образом, к вычислительным системам относятся программы (вместе с ЭВМ, на которых они выполняются), устройства вычислительных машин, системы программ, программно-технические комплексы, системы управления, содержащие в своем составе специализированные или универсальные ЭВМ. Сложность разработки вычислительных систем, в особенности стоимость создания качественного и надежного программного обеспечения ЭВМ, стимулируют развитие теоретических основ проектирования вычислительных систем, создание теоретически обоснованных методов и средств их разработки, поддержанных автоматизированными инструментальными системами.

Основная цель, которую ставили перед собой авторы при написании книги, состояла в том, чтобы укрепить связи между фундаментальными понятиями общей теории проектирования вычислительных систем и практическими методами проектирования программных и аппаратных средств вычислительной техники. С этой целью книга разделена на две части.

В первой части рассматриваются основные математические модели, используемые для описания и представления объектов проектирования, и исследуются их свойства. Эта часть составляет основу теории проектирования. Поскольку книга носит прикладной характер, авторы не развивают теоретические построения глубоко, ограничиваясь лишь теми результатами, которые имеют непосредственное применение в практике проектирования.

Вторая часть демонстрирует применение базовых математических понятий и конструкций к проектированию вычислительных систем. Основное внимание в ней уделяется проектированию последовательных и параллельных программ. Стиль изложения здесь не такой строгий, как в первой части. Основные положения методологии проектирования иллюстрируются примерами.

Теория программирования и теория проектирования аппаратуры первоначально развивались в значительной степени независимо. Первая базировалась на теории алгоритмов и теории формальных языков, вторая – на алгебре логики, теории автоматов и других разделах дискретной математики. В дальнейшем произошло сближение этих двух направлений, и в настоящее время их взаимное проникновение стало настоятельной необхо-

димостью. Особенно ярко общность моделей программирования и проектирования аппаратуры проявляется в области параллельного программирования, в которое все больше проникают методы проектирования структур ЭВМ. С другой стороны, проектирование БИС и СБИС не может мыслиться без анализа алгоритмов функционирования устройств с использованием алгоритмических языков высокого и сверхвысокого уровня, присущих программированию. Вторая цель книги и состоит в том, чтобы продемонстрировать общность математических моделей и методов проектирования программ и аппаратуры. Поэтому в качестве основного теоретического понятия выступает понятие дискретной динамической системы, которое с равным успехом описывает как процессы вычислений, порождаемые программой, так и процессы функционирования аппаратных компонент вычислительных систем.

Еще одна цель, которую преследовали авторы, состоит в изложении теоретических основ методологии проектирования схемного и программного оборудования ЭВМ, которая развивалась на протяжении ряда лет в Институте кибернетики АН УССР под руководством академика В.М. Глушкова. В 70-х годах эта методология получила название метода формализованных технических заданий или, употребляя более современный термин – формализованных спецификаций [25]. Исторической предпосылкой создания метода формализованных технических заданий послужила методика синтеза цифровых автоматов, предложенная в начале 60-х годов В.М. Глушковым [15]. В этой методике синтез был четко разделен на этапы блочного, абстрактного, структурного и надежностного синтеза. Каждый из этапов предполагал использование определенного класса математических моделей проектируемого устройства (цифрового автомата по терминологии того времени) и базирующегося на соответствующих моделях языка представления устройств. Этапы абстрактного и структурного синтеза были полностью формализованы и снабжены алгоритмами решения задач анализа, синтеза и оптимизации проектируемого устройства. Дальнейшее развитие методики было связано с решением задач блочного и алгоритмического синтеза автоматов применительно к проектированию вычислительных машин [16, 17]. В.М. Глушковым были предложены основные математические модели, на базе которых решаются эти задачи: понятие регистрового автомата и пары микропрограммных алгебр (в дальнейшем они стали называться системами алгоритмических алгебр или алгеброй алгоритмов). В 70-х годах метод формализованных технических заданий был расширен и включил в себя проектирование последовательных, а затем и параллельных программ [21, 23].

В методе формализованных технических заданий процесс проектирования вычислительной системы (программы или устройства) представляется в виде последовательности этапов, на каждом из которых проект системы представлен с помощью совокупности математических моделей, описывающих различные ее части. Различают три основных вида моделей – функциональные, динамические и структурные. Функциональные модели определяют функции, которые вычисляет проектируемая система, динамические модели определяют процессы функционирования системы или процессы вычислений, а структурные модели представляют систему в виде параллельной композиции компонент. Обычно процесс проектирования

протекает от функциональных к динамическим, а затем к структурным моделям (сверху вниз). Этот естественный переход может нарушаться для сложных систем, которые сначала представляются в структурном виде. Но тогда естественный путь проходят в процессе проектирования компоненты системы.

Наиболее общий класс динамических моделей составляют дискретные динамические системы, общая теория которых рассматривается в первой главе. Автоматы и дискретные преобразователи выделяются как важнейшие частные случаи. Во второй главе рассматриваются более конкретные классы динамических систем, используемые для представления алгоритмов функционирования вычислительных систем – абстрактные схемы программ (схемы Янова), схемы программ над памятью и алгебра алгоритмов (алгебра Глушкова). Последнюю модель можно относить как к динамическим, так и к функциональным моделям вычислительных систем, поскольку выражения в алгебре алгоритмов можно рассматривать и как выражения, порождающие процессы вычислений, и как операторы, действующие на информационной компоненте дискретного преобразователя. В качестве основной структурной модели параллельных алгоритмов рассматриваются сети из алгоритмических модулей.

При построении всех рассматриваемых моделей широко применяется алгебраическая точка зрения, при которой операции некоторой базовой алгебры рассматриваются как исходные алгоритмы, используемые для порождения новых объектов из заданных (порождающих элементов алгебры) или построения функций над этими объектами. Суперпозиции операций алгебры D дают элементарные D-функции, а рекурсивные определения, рассматриваемые как функциональные уравнения, порождают алгебраические D-функции. Общая теория рекурсивных функциональных определений над алгебрами с отношением аппроксимации строится в третьей главе. Теорема о неподвижной точке, которая лежит в основе теории рекурсивных определений, появляется уже в первой. Ее частный вариант используется при рассмотрении уравнений в алгебре языков, а общая формулировка для индуктивных частично упорядоченных множеств появляется в связи с алгеброй отношений. Завершением общей теории рекурсивных определений является теорема о пополнении алгебр с аппроксимацией, которая рассматривается в четвертой главе в связи с рекурсивными структурами данных. В главе о рекурсивных определениях решается также в общем виде задача синтеза динамических и отчасти структурных моделей, представляющих алгоритмы вычисления алгебраических D-функций. В иерархию моделей вычислительных систем и их компонент, разрабатываемых на различных этапах проектирования, вплетается также иерархия структур данных. Высший уровень этой иерархии образуют теоретико-множественные структуры данных. Они реализуются рекурсивными или функциональными структурами данных, а рекурсивные и функциональные структуры данных общего вида реализуются функциональными структурами данных специального вида – прямоугольными массивами, расположенными на целочисленной решетке. В конечном счете все может сводиться даже к простым одномерным массивам. Теория структур данных рассматривается в четвертой главе. Особенно важную роль играет здесь теория периодически определенных функций – общая основа проектирования операционных устройств ЭВМ (в особенности на

СБИС) и параллельного распределенного программирования. Эту главу можно рассматривать как базис общей теории функциональных моделей.

Во второй части книги основной материал связан с проектированием последовательных и параллельных программ. Вопросы проектирования аппаратуры сводятся к рассмотрению лишь нескольких примеров в пятой главе, посвященной архитектуре ЭВМ. Эти примеры иллюстрируют общую точку зрения на алгоритмическое и логическое проектирование устройств. Примеры проектирования последовательных программ демонстрируют решение задач синтеза, т.е. перехода от функциональных моделей к динамическим (процедурное представление), а также пошаговое уточнение динамических моделей. Рассматриваются также различные методы программирования – теоретико-множественное, недетерминированное и рекурсивное. На наш взгляд, стиль и методы программирования должны определяться не только и не столько опытом и аппаратом, которым владеют разработчики, сколько природой решаемых задач. В частности, конкретные ситуации могут требовать сочетания различных методов представления моделей на различных этапах проектирования и даже на одном и том же этапе.

Седьмая глава содержит результаты, полученные авторами книги и их коллегами за последние годы в связи с разработкой методов и средств проектирования программ для распределенных многопроцессорных ЭВМ. Этот материал может использоваться также при разработке структур ЭВМ и общесистемного математического обеспечения современных и перспективных ЭВМ. Принцип макроконвейерной обработки данных, предложенный В.М. Глушковым в 1978 г., послужил основным источником идей в этих исследованиях.

К сожалению, мы не имеем возможности достаточно глубоко рассмотреть все вопросы математической теории проектирования вычислительных систем. В частности, вне сферы нашего внимания остались многие вопросы применения методов математической логики, языковые проблемы, теория сложности вычислений, проектирование трансляторов и многое другое. Некоторые из этих вопросов уже достаточно хорошо освещены в монографиях, другие требуют специального внимания. Рассматриваемые в этой книге методы проектирования наиболее близки к трансформационному подходу в программировании, который развивается, например, Бауэром в западногерманском проекте CIP [81], и оригинальному варианту этого подхода, развиваемому в СО АН СССР под руководством академика А.П. Ершова [40, 44].

Авторы позволили себе отклоняться от традиционного математического стиля, поскольку книга рассчитана на более широкую аудиторию, чем специалисты, владеющие в совершенстве математическим аппаратом. Тем не менее, чтение книги потребует от читателя определенного напряжения. Во многих случаях конкретные положения не выделяются в отдельные утверждения и теоремы, хотя для них проводятся достаточно подробные обоснования, которые можно было бы формализовать в виде строгих доказательств. Читателю предоставляется возможность самостоятельно выделить из текста интересные для него факты, сформулировать соответствующие утверждения и провести доказательства на необходимом уровне строгости. Комментарии к главам не претендуют на полноту и в значительной степени выражают субъективную точку зрения авторов.

Часть I

МАТЕМАТИЧЕСКИЕ МОДЕЛИ

Глава 1

ДИСКРЕТНЫЕ СИСТЕМЫ

§ 1. Основные определения

Пусть T — множество моментов времени. Это может быть либо множеством неотрицательных вещественных чисел, либо множеством неотрицательных целых чисел. В первом случае говорят о непрерывном времени, во втором — о дискретном времени. В качестве T можно рассматривать также любое линейно-упорядоченное множество, изоморфное множеству неотрицательных вещественных или целых чисел. В этом случае, как правило, множество T будет отождествляться с соответствующим числовым множеством.

Рассмотрим множество, S которое будем называть *пространством состояний. Процессом* в S назовем произвольное отображение p: $[0:\tau] \to S$. Число τ называется длительностью процесса p и обозначается $|p|$. Если время дискретно, то процесс p можно отождествлять с последовательностью $p(0), p(1), \ldots, p(\tau)$ и рассматривать как слово в алфавите S. Заметим, что на мощность пространства состояний никаких ограничений не накладывается, и, говоря о словах, мы допускаем бесконечные и даже несчетные алфавиты. Заметим также, что длительность процесса в дискретном времени на 1 меньше, чем длина соответствующего слова.

Пусть $|p| = \tau$, $p(0) = s$, $p(\tau) = s'$. В этом случае будем говорить, что p начинается в состоянии s, заканчивается в состоянии s', и записывать это формулой $s \xrightarrow{p} s'$.

Для любых двух процессов p и q таких, что $s \xrightarrow{p} s'$ и $s' \xrightarrow{q} s''$, определим их последовательную композицию $p * q = r$, полагая, что $r(t) =$ (**если** $0 \leqslant t \leqslant |p|$ **то** $p(t)$ **иначе если** $|p| \leqslant t \leqslant |p| + |q|$ **то** $q(t - |p|)$ **иначе неопределено**). Очевидно, что $|p * q| = |p| + |q|$ и $s \xrightarrow{|p| * |q|} s''$.

Определим на множестве всех процессов в S отношение частичного порядка $\leqslant$, полагая

$p \leqslant q \iff$ существует процесс r такой, что $q = p * r$.

Процесс p называется *началом,* а r — *окончанием* процесса q. Операцию последовательной композиции следует отличать от полугруппового умножения (*конкатенации*) слов, представляющих процессы. Для процессов, протекающих в дискретном времени, последовательная композиция выражается через конкатенацию формулой $ps * sq = psq$.

Пусть p – процесс, протекающий в непрерывном времени. Будем говорить, что *p устойчив* в момент времени t $(0 \leqslant t \leqslant |p|)$, если в некоторой достаточно малой окрестности точки t функция p сохраняет постоянное значение. Если p неустойчив в момент t, то t называется *моментом переключения*. В случае дискретного времени моментом переключения называется любой момент времени, для которого определен рассматриваемый процесс. Процесс называется *дискретным*, если он содержит лишь конечное число моментов переключения.

Пусть F есть множество процессов в пространстве состояний S. Пара $\Gamma = (S, F)$ называется *абстрактной динамической системой*, если выполняется условие замкнутости множества F:

$$p \in F \text{ и } q \leqslant p \Rightarrow q \in F.$$

Если не возникает недоразумений, то систему $\Gamma = (S, F)$ будем обозначать и называть так же, как и ее пространство состояний, а множество F будем называть *множеством допустимых процессов* системы S. Условие замкнутости множества F можно перефразировать теперь следующим образом: всякое начало допустимого процесса допустимо.

Абстрактная динамическая система *дискретна*, если каждый ее допустимый процесс дискретен. Понятие процесса можно расширить, рассматривая также бесконечные процессы p: $T \to S$, определенные на множестве всех моментов времени. Понятие последовательной композиции $p * q$ распространяется естественным образом и на случай, когда p – конечный, а q – бесконечный процесс. Поэтому можно говорить о конечных началах бесконечных процессов. Бесконечный процесс в пространстве состояний системы S будем называть *допустимым*, если всякое его конечное начало допустимо. Из определения дискретной системы вытекает, что всякое конечное начало допустимого бесконечного процесса дискретной системы допустимо.

Если $S \triangleq \mathbf{R}^n$, а F – множество решений системы обыкновенных дифференциальных уравнений $\dot{x}_i = f_i(x_1, \ldots, x_n, t)$ $(i = 1, \ldots, n)$, то мы имеем дело с классической динамической системой. Для построения числовых решений уравнения в непрерывном времени заменяются конечно-разностными приближениями и рассматриваются в дискретном времени. Таким образом переходят к дискретным системам.

Любую дискретную систему S с непрерывным временем можно заменить системой S' с дискретным временем, пользуясь следующей стандартной конструкцией. Пусть $u, v, w \in S$. Рассмотрим процессы $p = uv^t w$, определяемые следующим образом: $|p| = t$, $p(0) = u$, $p(t) = w$, $p(\tau) = v$ $(0 < \tau < t)$. Процессы указанного типа назовем *элементарными*. Любой дискретный процесс p в пространстве S можно представить в виде композиции $p = p_0 * p_1 * \ldots * p_m$ элементарных процессов, так что $|p_i|$ есть момент переключения процесса $p_i * p_{i+1}$ $(i = 0, 1, \ldots, m-1)$. Такое представление, очевидно, единственно. В качестве пространства состояний системы S' возьмем множество всех элементарных процессов системы S, а множество ее допустимых процессов F' составим из процессов p' таких, что $p = p'(0) * \ldots * p'(m)$, где $m = |p'|$ есть разложение некоторого допустимого процесса $p \in F$ в последовательную композицию элементарных процессов. Между элементами множеств F и F' существует очевидное

взаимно однозначное соответствие, и система S' содержит полную информацию о системе S.

В дальнейшем, если не оговорено противное, будут рассматриваться только системы с дискретным временем.

Рассмотрим основной способ задания множества допустимых процессов – функцию переходов дискретной системы.

Через $\mathbf{P}(S)$ обозначим множество всех конечных процессов в пространстве S (время дискретно), отождествив его с множеством всех непустых слов в алфавите S. Если к множеству $\mathbf{P}(S)$ добавить пустое слово e, то получим множество $S^* = \mathbf{P}(S) \cup \{e\}$ всех слов в алфавите S. Отождествляя элементы пространства S с процессами нулевой длительности и со словами длины 1, получим $S \subset \mathbf{P}(S)$. Рассмотрим отображение δ: $\mathbf{P}(S) \to 2^S$ множества $\mathbf{P}(S)$ в множество всех подмножеств пространства S, полагая $\delta(p) = \{s \in S \mid ps \in F\}$ для всех $p \in \mathbf{P}(S)$. Функцию δ назовем *функцией переходов* системы S с множеством допустимых процессов F. Из определения функции переходов имеем: процесс ps ненулевой длительности допустим $\Longleftrightarrow S \in \delta(p)$. Состояние s назовем *допустимым начальным состоянием* системы S, если s есть допустимый процесс (нулевой длительности).

Теорема 1.1. *Множество допустимых процессов дискретной системы однозначно определяется ее функцией переходов и множеством допустимых начальных состояний.*

Действительно, множество допустимых процессов F можно представить в виде $F = \bigcup_{t=0}^{\infty} F_t$, где F_t есть множество допустимых процессов длительности t. Тогда F_0 есть множество допустимых начальных состояний, а

$$ps \in F_t \Longleftrightarrow p \in F_{t-1}, \quad ps \in \delta(p), \quad t > 0.$$

Теорема 1.1 показывает, что можно было бы дать другое определение дискретной системы, равносильное первоначальному: дискретная система – это тройка (S, S_0, δ), где S – множество ($S_0 \subset S$), а δ – функция из $\mathbf{P}(S)$ в 2^S, удовлетворяющая условию $s \notin S_0 \Rightarrow \delta(s) = \phi$. Действительно, если в качестве множества допустимых процессов в пространстве S взять множество таких процессов $s_0 s_1 \ldots s_\tau$, что $s \in S_0$, и для любого t такого, что $0 \leqslant t < \tau$, имеет место $s_{t+1} \in \delta(s_0\ s_1 \ldots s_t)$, то получим дискретную систему с множеством допустимых начальных состояний S_0 и функцией переходов δ. Условия

$$s_{t+1} \in \delta(s_0\ s_1 \ldots s_t), \quad s_0 \in S_0,$$

определяют закон функционирования системы S.

Функцию δ можно рассматривать как многозначное отображение из $\mathbf{P}(S)$ в S. Как и всякое многозначное отображение, функция δ определяет естественным образом бинарное отношение $s \in \delta(p)$ между элементами множеств $\mathbf{P}(S)$ и S. Это отношение называется *отношением переходов* системы S. Пару (p, s), находящуюся в этом отношении, будем записывать в виде $p \underset{S}{\to} s$ или просто $p \to s$, если известно, о какой системе идет речь. Если $p \to s$, то говорят, что допустимый процесс p системы S

может быть продолжен переходом в состояние s или что после окончания процесса p система может перейти в состояние s.

Для систем с непрерывным временем функцию переходов естественно привязывать не к моменту, а к интервалу времени. Пусть $P(S)$ есть множество всех конечных процессов в пространстве S с непрерывным временем. Рассмотрим функцию δ_t: $P(S) \to 2^{P(S)}$, полагая $\delta_t(p) = \{ q \in P(S) \mid p * q \in F, \mid q \mid = t \}$. Любая из функций δ_t при $t > 0$ вместе с множеством S_0 допустимых начальных состояний однозначно определяет множество F всех допустимых процессов системы S.

Рассмотрим некоторые виды дискретных систем.

1. Система S с функцией переходов δ называется *детерминированной*, если для любого $p \in \mathbf{P}(S)$ множество $\delta(p)$ состоит не более чем из одного элемента. Если система S детерминирована, то ее функцию переходов можно отождествить с частичным отображением $\delta \subset \mathbf{P}(S) \to S$. Функционирование детерминированной системы и множество ее допустимых процессов определяются уравнением

$$s_{t+1} = \delta(s_0 s_1 \ldots s_t), \qquad s_0 \in S_0.$$

2. Система S с функцией переходов δ называется *автоматной*, если $\delta(ps) = \delta(s)$. Функционирование автоматной системы и множество ее допустимых процессов определяются соотношением

$$s_{t+1} \in \delta(s_t), \quad s_0 \in S_0.$$

Функция переходов автоматной системы однозначно определяется своим ограничением $\bar{\delta}: S \to 2^S$ на множество S. Функция $\bar{\delta}$ является многозначным отображением из S в S, а определяемое ею отношение $s' \in \bar{\delta}(s) \iff$ $\iff s' \in \delta(s)$ есть ограничение отношения переходов на множество S. Процесс $s_0 s_1 \ldots s_t$ автоматной системы допустим $\iff s_0 \in S_0$ и $s_0 \to s_1 \to \ldots \to s_t$ $(s_0 \to s_1, s_1 \to s_2, \ldots, s_{t-1} \to s_t)$. Автоматная система может быть также определена как тройка (S, S_0, δ), где S — множество, $S_0 \subset S$, $\delta: S \to 2^S$, причем из $s \notin S_0$ следует $\delta(s) = \phi$.

3. Система называется *свободной*, если любой процесс является допустимым процессом этой системы. Самостоятельного значения свободные системы не имеют, но могут использоваться в качестве компонент сложных систем.

4. Система называется *конечной*, если множество ее состояний S конечно.

5. Система называется *многокомпонентной*, если множество ее состояний S содержится в декартовом произведении множеств $S \subset S_1 \times \ldots \times S_m$. Множества $S_1, \ldots, S_m$ называются *компонентами* системы S, если $S_1 \times \ldots \times S_m$ есть наименьшее произведение такое, что $S \subset S_1 \times \ldots \times S_m$. Очевидно, что $s_i \in S_i \iff$ существуют $s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_m$ такие, что $(s_1, \ldots, s_m) \in S$. Если некоторые из компонент многокомпонентной системы в свою очередь являются многокомпонентными, то система называется *многоуровневой*.

6. Если $S \subset \bigcup_{m=1}^{\infty} Q_m$, где каждое из множеств Q_m содержится в декартовом произведении, то система S называется *системой с переменной структурой*.

§ 2. Примеры дискретных систем

Всякая реальная система существует в непрерывном времени и изменяет свои состояния непрерывно. Дискретные системы являются моделями реальных систем. Адекватность этих моделей зависит от целей и задач, для решения которых они используются. С другой стороны, дискретные системы могут возникать умозрительно как точные математические объекты, предназначенные для изучения закономерностей, существующих внутри математических теорий. При благоприятном стечении обстоятельств такие системы могут стать источником новых технических идей или применений математики. Одним из таких примеров является машина Тьюринга, придуманная для изучения понятия вычислимости. С нее мы и начнем серию примеров дискретных систем.

1. Как известно, машина Тьюринга состоит из ленты, на которой записано слово в некотором конечном алфавите X, и головки, установленной на один из символов этого слова и находящейся в каждый момент времени в некотором состоянии из конечного множества A. Таким образом, состояние машины Тьюринга представляется тройкой $s = (p, a, q)$, где p и q — слова (быть может, пустые) в алфавите $X, a \in A$. Слово p определяет заполнение ленты слева от ячейки, обозреваемой головкой, включая эту ячейку; слово q — заполнение ленты справа. Множество допустимых процессов порождается программой, которая задается как множество пятерок вида $ax \to a'x'\alpha$, где $a, a' \in A$; $x, x' \in X$; $\alpha = -1, 0, +1$ (соответственно сдвиг влево, остаться на месте, сдвиг вправо). Машина Тьюринга является детерминированной автоматной системой, функция переходов которой определяется хорошо известными правилами. Например, если в программе есть пятерка $ax \to a'x'(+1)$, то $(px, a, yq) \to (px'y, a, q)$, а $(px, a, e) \to (px'\lambda, a, e)$, где λ — пустой символ.

2. Выполнение программы, записанной в некотором алгоритмическом языке, может быть описано с помощью дискретной динамической системы, переходы которой из состояния в состояние представляют шаги вычислений. Если задать соответствие между программами рассматриваемого языка и системами, порождающими процессы выполнения этих программ, получим динамическую, или операционную, семантику языка.

Рассмотрим для примера фрагмент программы, определяющей умножение двух матриц порядка N:

1 **для** I: = 1 **до** N **выполнить**
2 **для** J: = 1 **до** N **выполнить**
3 $C(I, J) := 0$;
4 **для** K: = 1 **до** N **выполнить**
5 $C(I, J)$: = $A(I, K) * B(K, J) + C(I, J)$
6 **конец цикла по** K
7 **конец цикла по** J
8 **конец цикла по** I.

Состояниями дискретной системы, описывающей процессы выполнения рассматриваемой программы, могут служить пары (m, u), где $m = 1, \ldots, 9$ — состояния управления, соответствующие строкам программного текста, $u = (i, j, k, a, b, c)$ — состояние памяти. Здесь i, j, k — целые числа,

значения переменных I, J, K; a, b, c – матрицы порядка N, значения переменных A, B, C. Процесс $s_1 s_2 \ldots$ допустим, если s_{t+1} получается из s_t по правилам определения действия операторов, из которых составлена программа. Пусть $u = (i, j, k, a, b, c)$. Тогда переход $(m, u) \to (m', u')$ может быть определен с помощью следующей таблицы:

m	Условие	m'	u'
1	$N > 0$	2	$(1, j, k, a, b, c)$
2	$N > 0$	3	$(i, 1, k, a, b, c)$
3		4	(i, j, k, a, b, c')
4	$N > 0$	5	$(i, j, 1, a, b, c)$
5		6	(i, j, k, a, b, c'')
6	$k \geqslant N$	7	u
6	$k < N$	5	$(i, j, k+1, a, b, c)$
7	$j \geqslant N$	8	u
7	$j < N$	3	$(i, j+1, k, a, b, c)$
8	$i \geqslant N$	9	u
8	$i < N$	2	$(i+1, j, k, a, b, c)$

В данной таблице c' и c'' – матрицы такие, что c'_{uv} = (если $u = i, v = j$ то 0 **иначе** c_{uv}), c''_{uv} = (если $u = i$, $v = j$ **то** $a_{ik} * b_{kj} + c_{uv}$ **иначе** c_{uv}). Если $N \leqslant 0$, то $(1, u) \to (9, u)$, $(2, u) \to (8, u)$, $(4, u) \to (7, u)$. Описанная система является двухкомпонентной двухуровневой автоматной детерминированной системой. Ее можно представить в виде диаграммы рис. 1.1. В этой диаграмме вершина с номером m представляет множество состояний, у которых первая компонента равна m. Указанную диаграмму можно рассматривать как диаграмму переходов некоторой другой дискретной системы. Эта система представляет собой модель управления рассмотренной программы. Она является конечной автоматной, но не детерминированной.

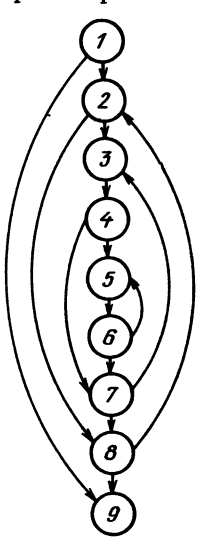


Рис. 1.1

3. Как известно, язык L – это множество слов в алфавите X. Процессы порождения элементов языка L слева направо символ за символом могут быть описаны с помощью дискретной системы $S = X \times \{0, 1\}$ с множеством допустимых процессов, определенных следующим условием. Процесс $(x_1, \alpha_1) \ldots (x_n, \alpha_n)$ допустим, если $x_1 \ldots x_n$ есть начало некоторого слова из L, а $\alpha_i = 1 \Longleftrightarrow x_1 \ldots x_n \in L$. Система S, вообще говоря, не детерминирована и не автоматна.

4. Последовательность формул логического исчисления такова, что каждая из формул является либо аксиомой, либо следствием из предыдущих формул, полученным однократным применением одного из правил вывода, и может рассматриваться как допустимый процесс некоторой дискретной системы. Состояния этой системы – формулы соответствующего языка. Система описывает процессы построения выводов в исчислении.

5. Игра в шахматы может быть описана как дискретная система, состояниями которой являются шахматные позиции, а переходы делаются по правилам игры. Система не является автоматной, поскольку переход зависит не только от позиции, но и от очередности хода, а также от условий допустимости рокировки и взятия на проходе.

6. Головоломки типа Ханойских башен или кубика Рубика представляют собой естественные примеры дискретных систем с конечным числом состояний. Многие задачи искусственного интеллекта формулируются как задачи поиска допустимого процесса, который оканчивается в одном из заданных множеств целевых состояний.

7. Сеть Петри определяется как ориентированный граф с вершинами двух типов. Вершины первого типа называются местами, вершины второго типа – переходами. Дуги могут соединять только вершины различных типов. Состояние сети или разметка мест – это отображение множества мест в множество неотрицательных целых чисел. Значение функции разметок на данном месте интерпретируется как количество фишек, расположенных на этом месте. Сети Петри изображаются графически так, как показано на рис. 1.2. Места изображаются кружочками, переходы – вертикальными линиями. Точки внутри мест – фишки. Места, из которых дуги ведут к некоторому переходу, называются входными местами этого перехода, а места, в которые входят дуги, выходящие из некоторого перехода, называются выходными местами этого перехода. Изменение состояния сети определяется правилами срабатывания переходов. Переход t может осуществиться, если каждое из его входных мест содержит по крайней мере одну фишку. Результатом срабатывания перехода будет новое состояние, которое получается вычитанием по одной фишке из каждого его входного места и добавлением по одной фишке к каждому выходному месту. В состоянии, изображенном на рис. 1.2, могут сработать переходы t_1 и t_3. Изображая разметку шестимерным вектором, координаты которого соответствуют местам, занумерованным, как показано на рисунке, получим допустимую последовательность состояний: $(3,2,0,2,2,0) \xrightarrow{t_1} (2,1,1,1,2,0) \xrightarrow{t_3} (2,1,1,1,2,1) \xrightarrow{t_2} (2,1,1,0,2,1) \xrightarrow{t_3} (2,1,1,0,2,2)$.

8. Аддитивные порождающие системы. Множество состояний есть множество целочисленных векторов размерности n с неотрицательными координатами. Отношение переходов определяется множеством V целочисленных n-мерных векторов. Система может перейти из состояния x в состояние x', если существует $z \in V$ такой, что $x' = x + z$. Любая сеть Петри может быть представлена как аддитивная порождающая система. Размерность такой системы равна числу мест. Элементы множества V соответствуют переходам. Вектор $z \in V$, соответствующий некоторому переходу, есть сумма $z_1 + z_2$. Координаты вектора z_1 равны -1 для компонент, соответствующих входным местам; координаты вектора z_2 равны 1 для выходных мест; остальные координаты векторов z_1 и z_2 равны 0.

Например, переходу t_2 сети рис. 1.2 соответствует вектор $(0,0,-1,-1,0,0) + (0,0,1,0,0,0) = (0,0,0,-1,0,0)$. Аддитивные порождающие системы так же, как и сети Петри, являются автоматными, но не обязательно детерминированными системами.

9. Асинхронные логические сети представляются системами булевых уравнений вида

$$y_i = f_i(x_1, \ldots, x_n, y_1, \ldots, y_m), \qquad i = 1, \ldots, m.$$

Каждое уравнение соответствует элементу, вычисляющему булеву функцию f_i. Переменные $x_1, \ldots, x_n$ соответствуют входным, $y_1, \ldots, y_m$ – внутренним переменным сети. Отношение переходов $(x, y) \to (x', y')$ определяется следующим образом. Значение x' может быть произвольным.

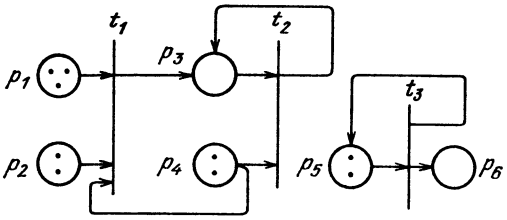


Рис. 1.2

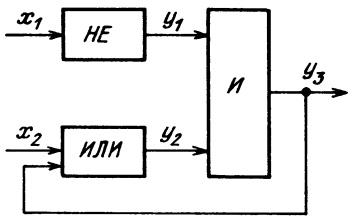


Рис. 1.3

Для вычисления y' подставляем значения $x = (x_1, \ldots, x_n)$ и $y = (y_1, \ldots, y_m)$ в уравнения и определяем уравнения, для которых равенство нарушается. Выбираем произвольным образом какие-либо из таких уравнений и изменяем значения координат вектора y, соответствующих этим уравнениям, на противоположные. Полученный вектор и есть возможное значение y'. На рис. 1.3 изображена асинхронная логическая сеть, построенная из элементов И, ИЛИ и НЕ. Она описывается уравнениями:

$$y_1 = \bar{x}_1;$$

$$y_2 = x_1 \vee y_3;$$

$$y_3 = y_1 \wedge y_2.$$

Обозначая состояния этой сети вектором $(x_1, x_2, y_1, y_2, y_3)$, получим пример допустимого процесса: $(0,0,0,0,0) \to (0,1,0,0,0) \to (0,1,1,0,0) \to (0,1,1,1,0) \to (0,1,1,1,1) \to (0,0,1,1,1)$.

§ 3. Реализация дискретных систем

Пусть S и S' – дискретные системы с множествами допустимых процессов F и F' соответственно. Рассмотрим отображение γ: $S \to S'$. Это отображение естественным образом продолжается до отображения γ: $\mathbf{P}(S) \to \mathbf{P}(S')$, если считать, что $\gamma(s_1 \ldots s_n) = \gamma(s_1) \ldots \gamma(s_n)$. Отображение γ называется *гомоморфизмом* системы S в S', если образ допустимого процесса системы S при этом отображении является допустимым процессом системы S'. Если δ и δ' – функции переходов, а S_0 и S_0' – множества допустимых начальных состояний систем S и S' соответственно, то γ есть гомоморфизм $\Leftrightarrow$ $\gamma(S_0) \subset S_0'$, $\gamma(\delta(p)) \subset \delta'(\gamma(p))$ для любого $p \in \mathbf{P}(S)$. Последнее включение может быть записано также в виде импликации: $p \xrightarrow[S]{} s \Rightarrow \gamma(p) \xrightarrow[S']{} \gamma(s)$. Система (S', F') называется *подсистемой* системы (S, F),

если $S' \subset S, F' \subset F$. *Образом* системы (S, F) при гомоморфизме γ в систему S' называется подсистема $(\gamma(S), \gamma(F))$ системы S'. Взаимно однозначный гомоморфизм называется *изоморфизмом.* Системы (S, F) и (S', F') изоморфны, если существует изоморфизм γ системы S на S' такой, что $\gamma(F) = F'$.

Пусть задан гомоморфизм γ системы S в S'. Тогда S' называется *гомоморфной моделью,* а S – *гомоморфной реализацией* системы S'. Такая терминология объясняется практическими задачами. Если мы хотим изучить некоторые свойства сложной системы S, то изучение их может быть проведено на упрощенной модели. Проще всего связь между исследуемым объектом и его моделью выражается в терминах гомоморфизма. С другой стороны, когда мы хотим реализовать некоторую абстрактную систему S' заданными средствами, понятие изоморфизма может быть слишком жестким, и мы вынуждены вводить дополнительные состояния для того, чтобы реализация была возможна. И в этом случае простейшая связь между реализацией S и исходной моделью S' – гомоморфизм. Наблюдая допустимый процесс p функционирования системы S, которая реализует систему S' при помощи гомоморфизма γ, мы однозначно восстанавливаем процесс функционирования $q = \gamma(p)$ системы S'. Процесс p называется также *реализацией* процесса q, а q – его *моделью.* Заметим, что гомоморфизмов из S в S' может существовать много, и мы говорим о реализации лишь в случае, когда такой гомоморфизм зафиксирован. Этот гомоморфизм называется также *реализующим.*

Гомоморфная реализация S системы S' называется *полной,* если каждый допустимый процесс системы S' имеет реализацию, т.е. $\gamma(F) = F'$.

Теорема 3.1. *Всякая дискретная система S' имеет полную гомоморфную автоматную реализацию S.*

Действительно, в качестве множества состояний системы S возьмем множество F' допустимых процессов системы S'. Определим функцию переходов δ системы S условием $\delta(p) = \{ps \mid ps \in F'\}$. Отображение $\gamma: F' \to S'$, определенное равенством $\gamma(ps) = s$, является гомоморфизмом. Действительно, каждый допустимый процесс системы S имеет вид $p = (s_0)(s_0 s_1) \ldots (s_0 s_1 \ldots s_t)$, где $s_0 s_1 \ldots s_t \in F'$, а $\gamma(p) = s_0 s_1 \ldots s_t = q$. Процесс p является также реализацией процесса q, и, поскольку q произволен, реализация полна.

Метод получения автоматной реализации, примененный для доказательства теоремы 3.1, неэффективен. Действительно, эта реализация имеет бесконечно много состояний, даже если система S' конечна. Вопрос о существовании конечных автоматных реализаций конечных систем будет рассмотрен в следующих параграфах.

Более общее понятие реализации дискретных систем можно получить, если отказаться от необходимости задавать отображение γ только на состояниях, а определять его сразу на множестве всех допустимых процессов.

Пусть F и F' – множества процессов. Отображение $\gamma: F \to F'$ назовем *монотонным,* если из $p \leqslant q$ следует $\gamma(p) \leqslant \gamma(q)$. Из монотонности следует также, что $\gamma(p * r) = \gamma(p) * r'$. Монотонное отображение $\gamma: F_1 \to F_2$ назовем *реализующим отображением системы* (S_1, F_1) в (S_2, F_2). Если задано реализующее отображение, то S_1 называется (монотонной) *реализацией,* а S_2 – *моделью системы* S_1. Так же и для процессов: если $p \in F_1$, то процесс

$q = \gamma(p)$ называется *моделью процесса* p, а p *– реализацией процесса* q. Реализация называется *полной*, если $\gamma(F_1) = F_2$. Так же как и в случае гомоморфной реализации, наблюдая за поведением системы S_1, можно однозначно восстановить соответствующее поведение ее модели. При этом восстановление производится последовательно слева направо.

Понятие реализации дословно переносится на системы с непрерывным временем. Только вместо множества $\mathbf{P}(S)$ следует рассматривать множество $P(S)$ конечных процессов в непрерывном времени. Реализующее отображение можно задавать и для систем с разными множествами моментов времени.

Рассмотрим некоторые типичные способы построения моделей дискретных систем.

1. Пусть $S_0 \subset S' \subset S$, где S_0 – множество допустимых начальных состояний системы S. Всякий допустимый процесс p системы S однозначно можно представить в виде $p = s_1 p_1 s_2 p_2 \ldots s_m p_m$, где $s_1, \ldots, s_m \in S' (m \geqslant 0)$, а слова $p_1, \ldots, p_m$ не содержат вхождений состояний из S', т.е. $p_1, \ldots, p_m \in (S \backslash S')^*$. Положим $\gamma(p) = s_1 \ldots s_m$. Рассмотрим систему S', выбрав в качестве ее допустимых процессов множество всех процессов вида $\gamma(p)$, где p – допустимый процесс системы S. Поскольку функция γ монотонна, система S является полной монотонной реализацией системы S' с реализующей функцией γ. Систему S' будем называть *проекцией* системы S на подмножество состояний $S' \subset S$.

Модель S' обычно получается следующим образом. В системе S выделяются некоторые основные или существенные состояния, которые интересуют нас по тем или иным соображениям. Допустимые процессы получаются путем опускания промежуточных состояний. Например, рассматривая работу программы с подпрограммами, можно игнорировать состояния, соответствующие внутренним операторам подпрограмм, оставив только состояния, соответствующие обращениям к подпрограммам и выходам из них. Методы проектирования последовательных программ сверху вниз обычно сводятся к построению цепочки реализаций, модели которых являются проекциями на подмножества или гомоморфизмами.

Если система S детерминирована или автоматна, то S' также детерминирована или автоматна. Но S' может быть детерминированной или автоматной, даже если S не является таковой.

2. В случае многокомпонентных систем описанная выше конструкция может быть реализована следующим образом. Пусть $S = S_1 \times S_2$ – двухкомпонентная система с множеством начальных состояний $S_0 = S_1^{(0)} \times S_2^{(0)}$ и в ее компонентах выделены подмножества $S_1^{(0)} \subset S_1' \subset S$ и $S_2^{(0)} \subset S_2' \subset S_2$.

Построим систему $S' = S_1' \times S_2'$ следующим образом. Пусть $r = (a_1, b_1) \ldots (a_m, b_m)$ – допустимый процесс системы S. Представим слова $p = a_1 \ldots a_m$ и $q = b_1 \ldots b_m$ в виде $p = s_1 p_1 \ldots s_k p_k$, $q = s_1' p_1' \ldots s_k' p_k'$, где $p_1, \ldots, p_{k-1} \in (S_1 \backslash S_1')^*$, $p_1', \ldots, p_{k-1}' \in (S_2 \backslash S_2')^*$, $s_1, \ldots, s_k \in S_1'$, $s_1', \ldots, s_k' \in S_2'$ и либо $p_k \in (S_1 \backslash S_1')^*$, либо $p_k' \in (S_2 \backslash S_2')^*$. Положим $\gamma(r) = (s_1, s_1') \ldots (s_k, s_k')$. Отображение γ монотонно и является реализующим отображением системы S на систему S', допустимые процессы которой имеют вид $\gamma(r)$, где r – допустимый процесс системы S. Систему S' назовем *асинхронной проекцией* S на $S_1' \times S_2'$.

3. Конструкция перехода от систем с непрерывным временем к системам с дискретным временем, описанная в § 1, может рассматриваться как построение модели с дискретным временем, поскольку отображение $\gamma(p_1 * p_2 * \ldots * p_k) = \gamma(p_1)\,\gamma(p_2)\,\ldots\gamma(p_k)$, где $p_1, \ldots, p_k$ – элементарные процессы, и $p_1, \ldots, p_k$ оканчиваются в моменты переключения монотонно. Если взять $S' = S^3$, $\gamma(uv^tw) = (u, v, w)$, то имеем более простую модель, в которой потеряны длительности, но сохранен порядок смены состояний.

4. Пусть

$$y_i = f_i(x_1, \ldots, x_n, y_1, \ldots, y_m),\ i = 1, \ldots, m,$$

есть система уравнений, определяющая асинхронную логическую сеть. Состояние (x, y) этой сети назовем *устойчивым,* если оно удовлетворяет системе уравнений. Пусть (S, F) – подсистема сети, которая получается наложением следующих двух условий на допустимые процессы:

– каждый допустимый процесс начинается в устойчивом состоянии;

– если $(x, y) \to (x', y')$ и (x, y) не устойчиво, то $x' = x$.

Проекция S на множество S' устойчивых состояний представляет собой модель функционирования сети, в которой игнорируются переходные процессы.

§ 4. Алгебра языков

Множество допустимых процессов дискретной системы представляет собой язык в алфавите состояний. Поэтому для задания множества допустимых процессов можно использовать общие методы задания языков. Эти методы особенно удобны для определения недетерминированных систем с конечным числом состояний. Алгебра языков, определенная в этом параграфе, имеет также и другие применения.

Пусть W – некоторый алфавит. Множество W^* всех слов в алфавите W является свободной полугруппой с единицей относительно операции умножения (приписывания) слов. Пустое слово e является единицей этой полугруппы. Определим на множестве 2^{W^*} всех языков в алфавите W (подмножеств множества W^*) следующие операции.

1. Произведение языков $PQ = \{pq \mid p \in P, q \in Q\}$.

2. Дизъюнкция $P \vee Q$ – теоретико-множественное объединение языков.

3. Итерация P^* языка P; $P^* = \bigcup_{n=0}^{\infty} P^n$, где $P^0 = \{e\}$, $P^{n+1} = P^n P$.

Множество 2^{W^*} вместе с указанными операциями образует алгебру, которая называется *алгеброй всех языков* в алфавите W. Вообще *алгеброй языков* называется произвольное множество языков, замкнутое относительно операций умножения, дизъюнкции и итерации.

Из определения операций вытекают следующие простые утверждения, полезные при доказательстве соотношений в алгебре языков.

1. $p \in PQ \Leftrightarrow$ существуют $p_1 \in P$ и $p_2 \in Q$ такие, что $p = p_1 p_2$.

2. $p \in P \vee Q \Leftrightarrow p \in P$ или $p \in Q$.

Отождествляя символы алфавита W с однобуквенными словами, получим включение $W \subset W^*$. Отождествляя слова с одноэлементными языками, получим также включение $W^* \subset 2^{W^*}$. Элементарными языками будем называть языки e, w $(w \in W)$ и пустой язык ϕ. Замыкая множество эле-

ментарных языков относительно операций алгебры языков, получим алгебру $\mathcal{K}(W)$ регулярных языков. Каждый регулярный язык может быть записан в виде выражения, построенного из символов элементарных языков с помощью операций алгебры языков. Такие выражения называются регулярными. Если L – произвольное множество языков, то, замыкая это множество с помощью операций алгебры языков, получим алгебру $\mathcal{K}(L)$ языков, регулярных относительно L. Каждый язык этой алгебры может быть также записан в виде выражения, построенного из обозначений языков из L и операций алгебры языков. Такие выражения называются *регулярными относительно* L.

Рассмотрим основные соотношения алгебры языков:

$P(QR) = (PQ)R$ (ассоциативность умножения);

$Pe = eP = P$;

$P \vee \phi = \phi \vee P = P$;

$P \vee Q = Q \vee P$ (коммутативность дизъюнкции);

$(P \vee Q) \vee R = P \vee (Q \vee R)$ (ассоциативность дизъюнкции);

$P \vee P = P$ (идемпотентность дизъюнкции);

$P(Q \vee R) = PQ \vee PR$ (левая дистрибутивность);

$(P \vee Q)R = PR \vee QR$ (правая дистрибутивность);

$P^* = e \vee PP^* = e \vee P^*P$ (развертывание итерации);

$P^* = (e \vee P \vee P^2 \vee \ldots \vee P^{n-1})(P^n)^*$;

$P^{**} = P^*$;

$P^*P^* = P^*$;

$(P^* \vee Q)^* = (P \vee Q)^*$;

$P^*P = PP^*$.

Все указанные соотношения являются тождественными, т.е. они выполняются при любых значениях переменных P, Q и R в любой алгебре языков. Доказательства этих соотношений можно получить путем доказательства двух включений, исходя непосредственно из определения операций. Докажем, например, левую дистрибутивность.

Пусть $p \in P(Q \vee R)$. Тогда $p = p_1 p_2, p_1 \in P, p_2 \in Q \vee R$. Если $p_2 \in Q$, то $p \in PQ \subset PQ \vee PR$. Если же $p_2 \in R$, то $p \in PR \subset PQ \vee PR$. Таким образом, $P(Q \vee R) \subset PQ \vee PR$. Аналогично доказывается обратное включение.

Другие соотношения алгебры языков можно получить как следствия из указанных или непосредственным доказательством, исходя из определения операций. Простое описание всех тождеств алгебры языков не известно, хотя существует алгоритм их распознавания, основанный на результатах теории автоматов.

Следующий вопрос – уравнения в алгебре языков. Рассмотрим сначала одно уравнение с одним неизвестным вида

$$X = XP \vee Q, \tag{4.1}$$

где P и Q – известные языки, X – неизвестный язык. Такого типа уравнения будем называть *каноническими линейными уравнениями.*

Уравнение (4.1) имеет решение $X = QP^*$. Действительно, применяя соотношения алгебры языков, получим $X = QP^* = Q(e \vee PP^*) = Q \vee (QP^*)P = XP \vee Q$.

Аналогичным образом правое уравнение $X = PX \vee Q$ имеет решение $X = P^*Q$.

Множество всех языков упорядочено отношением включения. Покажем, что $X = QP^*$ есть наименьшее решение уравнения (4.1). Действительно, пусть Y – какое-нибудь решение. Покажем, что $X \subset Y$. Имеем $p \in QP^* \Leftrightarrow$ $\Leftrightarrow p = p'p'', p'' = p_1 \dots p_m$ $(p_i \in P, p' \in Q)$. Подставляя многократно правую часть уравнения $Y = YP \vee Q$ вместо Y в себя, получим $Y = YP \vee Q = (YP \vee Q)P \vee$ $\vee Q = YP^2 \vee QP \vee Q = \dots = YP^{m+1} \vee QP^m \vee \dots \vee QP \vee Q$. Поскольку $p \in QP^m$, то $p \in Y$.

Вообще говоря, уравнение (4.1) может иметь много решений, но при условии $e \notin P$ имеет место единственное решение. Действительно, если Y – какое-нибудь решение и длина слова p равна m, то из $p \in Y$ и $Y =$ $= YP^{m+1} \vee QP^m \vee \dots \vee QP \vee Q$ следует, что $p \in QP^m \vee \dots \vee Q$, так как все слова в YP^{m+1} имеют длину больше, чем m. Но тогда $p \in QP^* = X$.

Резюмируя сказанное, получаем следующую теорему.

Теорема 4.1. *Уравнение* $X = XP \vee Q$ $(X = PX \vee Q)$ *имеет решение* $X =$ $= QP^*$ $(X = P^*Q)$. *Это решение является наименьшим и, если* $e \notin P$, *единственным.*

Рассмотрим теперь систему уравнений вида

$$X_i = \bigvee_{j=1}^{n} X_j P_{ij} \vee Q_i, \quad i = 1, \dots, n. \tag{4.2}$$

Системы вида (4.2) называются *каноническими* системами линейных уравнений в алгебре языков или *леволинейными* системами. *Праволинейные* системы имеют вид

$$X_i = \bigvee_{j=1}^{n} P_{ij} X_j \vee Q_i, \quad i = 1, \dots, n. \tag{4.3}$$

Теорема 4.2. *Среди решений систем* (4.2) *и* (4.3) *существует наименьшее решение, регулярное относительно коэффициентов* P_{ij} *и свободных членов* Q_i. *Если для всех* $i, j = 1, \dots, n$ *имеет место* $e \notin P_{ij}$, то это решение *единственно.*

Доказательство проводится методом решения систем (4.2) и (4.3), основанным на исключении неизвестных. Каждый шаг исключения делается с применением теоремы 4.1. Рассмотрим первое уравнение системы (4.2). Если X_1 не встречается в правой части уравнения, то X_1 можно исключить, подставив правую часть во все остальные уравнения. Если же X_1 входит в правую часть, то, применяя теорему 4.1, получим, что

$$X_1 = \Big(\bigvee_{j=2}^{n} X_j P_{1j} \vee Q_1\Big) P_{11}^*$$

является наименьшим решением этого уравнения при известных $X_2, \dots, X_n$. Подставив вместо X_1 его значение в другие уравнения, снова получим систему с меньшим числом неизвестных. После раскрытия скобок по закону правой дистрибутивности и приведения подобных членов, т.е. вынесения за скобки неизвестных по закону левой дистрибутивности, снова получим каноническую систему линейных уравнений. Продолжая подобным образом, в конце концов получим наименьшее решение в виде выражения, регулярного относительно коэффициентов и свободных членов. Заметим, что если коэффициенты исходной системы не содержат пустых слов, то

это свойство сохранится и при исключении, а решение, полученное на каждом шаге, будет единственным.

Рассмотрим пример. Решим систему уравнений

$$X = XP \vee Y \vee e,$$

$$Y = XQ \vee YP.$$

Исключая Y из второго уравнения, получим

$$Y = XQP^*.$$

Подставляем Y в первое уравнение и исключаем X:

$$X = XP \vee XQP^* \vee e = X(P \vee QP^*) \vee e =$$

$$= (P \vee QP^*)^* = (P \vee Q)^*.$$

Последнее равенство можно получить из следующих включений, которые устанавливаются непосредственной проверкой:

$$P \vee QP^* \subset (P \vee Q)^* \Rightarrow (P \vee QP^*)^* \subset (P \vee Q)^*,$$

$$P \vee Q \subset P \vee QP^* \Rightarrow (P \vee Q)^* \subset (P \vee QP^*)^*.$$

Окончательно получаем

$$X = (P \vee Q)^*,$$

$$Y = (P \vee Q)^* QP^*.$$

Если $e \in Q$, то $X = Y$, но решение в этом случае не единственно.

§ 5. Конечные системы

В этом параграфе будет изучена связь между конечными и конечными автоматными системами. Метод изучения основан на применении алгебры языков и доказательстве основной теоремы, являющейся обобщением теоремы анализа и синтеза в теории конечных автоматов.

Конечные автоматные системы можно задавать с помощью графов, или диаграмм переходов. Вершинами такого графа служат состояния системы, а дуга связывает две вершины, если возможен переход от первого состояния ко второму. Для того чтобы множество допустимых процессов было определено, необходимо также задавать множество допустимых начальных состояний. На рис. 1.4 представлена диаграмма переходов системы с четырьмя состояниями. Все ее состояния являются допустимыми начальными состояниями, поскольку из каждого возможен переход в некоторые другие $(\delta(s) \neq \phi)$.

Состояние s автоматной системы называется *тупиковым*, если $\delta(s) = \phi$. Только тупиковые состояния могут не быть допустимыми начальными состояниями автоматной системы.

Т е о р е м а 5.1 (теорема анализа конечных автоматных систем). *Множество допустимых процессов конечной автоматной системы регулярно.*

Пусть (S, F) — конечная автоматная система с множеством допустимых начальных состояний S_0 и функцией переходов δ. Рассмотрим множества $F_s = \{ps \in F\}$ и $F^s = \{sp \in F\} \cup \{s\}$, где $s \in S$, $p \in S^*$. Указанные множества

удовлетворяют уравнениям

$$F_s = \alpha(s) \vee \bigvee_{s' \to s} F_{s'} s, \quad s \in S, \tag{5.1}$$

$$F^s = s \vee \bigvee_{s \to s'} s F^{s'}, \quad s \in S, \tag{5.2}$$

где $\alpha(s)$ = (**если** $s \in S_0$ **то** s **иначе** ϕ). Действительно, $p \in F_s \Leftrightarrow p = s \in S_0$ или $p = p's's$, $p's' \in F_{s'}$, $s' \to s$. В первом случае $p \in \alpha(s)$, во втором $p \in F_{s'}s$.

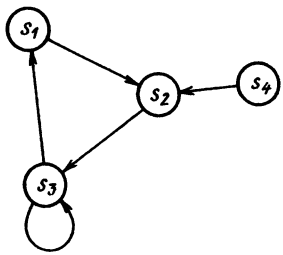


Рис. 1.4

Далее, $p \in F^s \Leftrightarrow p = s$ или $p = ss'p'$, $s \to s'$, $s'p' \in F^{s'}$, что доказывает (5.2). Системы (5.1) и (5,2) являются каноническими системами линейных уравнений в алгебре регулярных языков. Поскольку коэффициенты этих систем не содержат пустого слова, в силу теоремы 4.2 каждое из семейств $(F_s)_{s \in S}$ и $(F^s)_{s \in S}$ образует единственное решение системы (5.1) и (5.2) соответственно. С другой стороны,

$$F = \bigvee_{s \in S} F_s = \bigvee_{s \in S_0} F^s.$$

Доказательство теоремы анализа содержит в себе также алгоритм решения задачи анализа, которая состоит в том, чтобы, зная отношение переходов и множество допустимых состояний, найти регулярное выражение для множества допустимых процессов. Алгоритм состоит из двух этапов. Первый этап – построение системы уравнений, второй этап – получение решения. Решим, например, задачу анализа системы, изображенной на рис. 1.4. В этом случае удобно воспользоваться системой (5.1):

$$F_1 = s_1 \vee F_3 s_1,$$

$$F_2 = s_2 \vee F_1 s_2 \vee F_4 s_2,$$

$$F_3 = s_3 \vee F_2 s_3 \vee F_3 s_3,$$

$$F_4 = s_4.$$

Выражаем F_1 и F_2 через F_3, исключаем F_4, получаем уравнение для F_3 и решаем его:

$$F_3 = s_3 \vee s_2 s_3 \vee F_1 s_2 s_3 \vee s_4 s_2 s_3 \vee F_3 s_3 =$$

$$= s_3 \vee s_2 s_3 \vee s_1 s_2 s_3 \vee s_4 s_2 s_3 \vee F_3 (s_1 s_2 s_3 \vee s_3) =$$

$$= (s_3 \vee s_2 s_3 \vee s_1 s_2 s_3 \vee s_4 s_2 s_3)(s_1 s_2 s_3 \vee s_3)^*.$$

Используем полученный результат для определения F:

$$F = F_1 \vee F_2 \vee F_3 \vee F_4 = s_1 \vee F_3 s_1 \vee s_2 \vee s_4 s_2 \vee (s_1 \vee F_3 s_1) s_2 \vee F_3 \vee s_4 =$$

$$= s_1 \vee s_2 \vee s_4 \vee s_1 s_2 \vee s_4 s_2 \vee F_3 (e \vee s_1 \vee s_1 s_2) =$$

$$= s_1 \vee s_4 \vee Q \vee (e \vee Q) s_3 (s_3 \vee s_1 s_2 s_3)^* (e \vee s_1 \vee s_1 s_2),$$

где $Q = (e \vee s_1 \vee s_4) s_2$.

Пусть (S, F) – автоматная система, $S_1 \subset S_0$, $S_2 \subset S$. Через $L(S_1, S_2)$ обозначим язык $L(S_1, S_2) = \{ p \in F | s_1 \xrightarrow{p} s_2, s_1 \in S_1, s_2 \in S_2 \}$. Пусть γ: $S \to X$ – гомоморфизм системы S в свободную систему X. Образ $\gamma(L(S_1, S_2)) = R$ есть некоторый язык в алфавите X. О языке R будем говорить, что он порождается системой S при множестве начальных состояний S_1, заключительных состояний S_2 и порождающем гомоморфизме γ. Язык называется *конечно-автоматным*, если он порождается конечной автоматной системой.

Т е о р е м а 5.2 (теорема синтеза конечных автоматных систем). *Если язык R регулярен, то $R \backslash e$ конечно-автоматен.*

Для доказательства теоремы 5.2 нужно определить метод построения (синтеза) конечной автоматной системы, порождающей данный язык. Доказательство проведем индукцией по числу операций, использованных в регулярном выражении, представляющем язык R. Базис индукции сводится к рассмотрению случая, когда R является элементарным языком. Шаг индукции требует рассмотрения трех случаев: $R = R_1 R_2$, $R = R_1 \vee R_2$, $R = R_1^*$. Для каждого из трех случаев будет указана конструкция, которая по системам S_1 и S_2, порождающим языки $R_1 \backslash e$ и $R_2 \backslash e$, строит систему S, порождающую язык $R \backslash e$. Системы, порождающие языки x и ϕ, показаны на рис. 1.5.

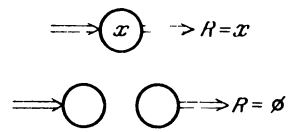


Рис. 1.5

Конструкции для трех операций алгебры языков показаны на рис. 1.6. Двойными стрелками показаны множества начальных и заключительных состояний. Построение системы S для произведения можно описать следующим образом. Диаграммы переходов систем S_1 и S_2 с непересекающимися множествами состояний объединяются. В качестве множества начальных состояний системы S выбирается множество начальных состояний системы S_1, в качестве заключительных – системы S_2. Все заключительные состояния системы S_1 соединяются переходами с начальными состояниями системы S_2. Порождающий гомоморфизм системы S на множестве S_1 действует как на первой системе, на S_2 – как на второй. Если $e \in R_2$, то к заключительным состояниям добавляются заключительные состояния системы S_1, а если $e \in R_1$, то к начальным состояниям системы S_1 добавляются начальные состояния системы S_2. Построение системы S_1 для дизъюнкции видно из рисунка. Для итерации в системе S_2 нужно просто соеди-

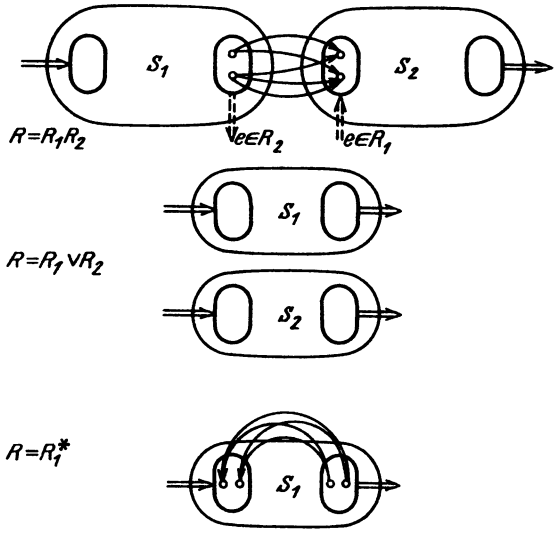


Рис. 1.6

нить переходами заключительные состояния с начальными. Доказательство того, что полученные системы порождают соответствующие языки, предоставляется читателю в качестве легкого упражнения.

Т е о р е м а 5.3. *Язык R регулярен $\Leftrightarrow R\backslash e$ конечно-автоматен.*

Первая часть теоремы совпадает с теоремой 5.2. Пусть $R\backslash e$ – конечно-автоматный язык, а (S, F) – конечная система, которая его порождает, т.е. $R\backslash e = \gamma(L(S_1, S_2))$. Для любой пары состояний $s, s' \in S$ рассмотрим язык $F_{ss'} = \{p \in F \mid s \xrightarrow{p} s'\}$. Языки $F_{ss'}$ удовлетворяют соотношениям

$$F_{ss'} = \beta(s, s') \vee \bigvee_{s'' \to s'} F_{ss''} s' = \beta(s, s') \vee \bigvee_{s \to s''} s F_{s''s'},$$

где $\beta(s, s')$ = (**если** $s \to s'$ **то** ss' **иначе** ϕ). Поэтому все они регулярны. С другой стороны,

$$L(S_1, S_2) = \bigvee_{\substack{s \in S_1 \\ s' \in S_2}} F_{ss'},$$

и для доказательства регулярности $R\backslash e$ осталось показать, что всякий гомоморфизм γ сохраняет регулярность. Прежде чем доказывать это свойство, распространим действие гомоморфизма γ на все слова в данном алфавите, включая пустое слово и полагая $\gamma(e) = e$. Теперь выражение $\gamma(R)$ имеет смысл для произвольного языка, а не только для множества процессов, т.е. непустых слов.

Л е м м а 5.1. *Образ регулярного языка при гомоморфизме γ: $S \to X$ регулярен.*

Это вытекает из соотношений

$\gamma(PQ) = \gamma(P)\gamma(Q)$,

$\gamma(P \vee Q) = \gamma(P) \vee \gamma(Q)$,

$\gamma(R^*) = (\gamma(R))^*$.

Действительно, $p \in \gamma(PQ) \Leftrightarrow p = \gamma(p_1p_2) = \gamma(p_1)\gamma(p_2),\ p_1 \in P,\ p_2 \in Q \Leftrightarrow$ $\Leftrightarrow p \in \gamma(P)\gamma(Q)$ и т.д.

Теперь можно сформулировать и доказать основной результат этого параграфа.

Т е о р е м а 5.4. *Конечная система имеет конечную полную автоматную гомоморфную реализацию $\Leftrightarrow$ множество ее допустимых процессов регулярно.*

$\Rightarrow$ Пусть S есть полная гомоморфная реализация системы S', а γ: $S \to S'$ – реализующий гомоморфизм. Если S – конечная автоматная система, то множество ее допустимых процессов F регулярно и $F' = \gamma(F)$ регулярно в силу леммы 5.1.

$\Leftarrow$ Пусть (S, F) – конечная система с регулярным множеством допустимых процессов F. Построим систему (S', F'), порождающую язык F. В силу теоремы 5.3 систему S' можно сделать конечной автоматной. Кроме того, можно предположить, что из любого состояния системы S' достижимо множество заключительных состояний. Действительно, состояния, из которых не достижимо множество заключительных состояний, можно отбросить, не изменив при этом языка, порождаемого системой. Покажем, что порождающий гомоморфизм γ: $S' \to S$ определяет полную гомоморфную реализацию. Пусть p – допустимый процесс системы S'. Поскольку множество заключительных состояний достижимо, то p можно продолжить до процесса pq, который кончается в одном из заключительных состояний. Тогда $\gamma(pq) = \gamma(p)\gamma(q)$ допустим и в силу замкнутости множества допустимых процессов $\gamma(p)$ также допустим. С другой стороны, поскольку S' порождает F, то для каждого $p \in F$ найдется $p' \in F'$ такой, что $\gamma(p') = p$. Теорема доказана.

§ 6. Многокомпонентные системы

Пусть p и p' – два процесса одной и той же длительности в пространствах S и S' соответственно (время может быть как непрерывным, так и дискретным). Определим параллельную композицию $q = p \otimes p'$ процессов p и p', полагая $q(t) = (p(t), p'(t))$ для всех t $(0 \leqslant t \leqslant |p| = |p'|)$. Композиция $p \otimes p'$ есть процесс в пространстве $S \times S'$. Процессы p и p' называются проекциями процесса q на компоненты S и S' соответственно. Если F и F' – два множества процессов в пространствах S и S', то через $F \otimes F'$ обозначим множество всех процессов вида $p \otimes p'$ $(p \in F, p' \in F')$. Определение параллельной композиции и проекций распространяется очевидным образом на любое конечное или бесконечное число сомножителей.

Систему (S, F) будем называть *параллельной композицией* систем (S_1, F_1), ..., (S_m, F_m), если $S \subset S_1 \times \ldots \times S_m$, $F \subset F_1 \otimes \ldots \otimes F_m$. Множество S называется также *множеством допустимых состояний* композиции (оно может не совпадать с $S_1 \times \ldots \times S_m$). Если $S = S_1 \times \ldots \times S_m$, а $F = F_1 \otimes \ldots \otimes F_m$, то S называется *прямым произведением* систем S_1, ..., S_m и обозначается $S_1 \otimes \ldots \otimes S_m$. Таким образом, произвольная параллельная композиция систем есть подсистема прямого произведения.

Заданную многокомпонентную систему (S, F) можно многими способами разлагать в композицию своих компонент. Одним из таких способов является представление S в виде композиции свободных систем. Другой

способ состоит в том, что в качестве допустимых процессов системы (S_i, F_i) выбирается множество всех проекций на i-ю компоненту всех допустимых процессов системы S. Другие разложения лежат между двумя указанными в том смысле, что если S разложена также в композицию систем (S_i, F_i'), то $(S_i, F_i) \subset (S_i, F_i')$ (состояния рассматриваются как допустимые процессы длительности 0). С другой стороны, $(S_i, F_i') \subset \subset (S_i, \mathbf{P}(S_i))$.

Далее будем снова рассматривать только дискретное время. Пусть $S \subset S_1 \otimes \ldots \otimes S_m$. Определим функции переходов компонент $\delta_i (i = 1, \ldots, m)$ следующим образом. Функция δ_i зависит от i аргументов. Первый аргумент — процесс в пространстве S, остальные $i-1$ аргументов — состояния компонент $S_1, \ldots, S_{i-1}$, $\delta_i(p, s_1, \ldots, s_{i-1}) \subset S_i$, $s_i \in \delta_i(p, s_1, \ldots, s_{i-1}) \Leftrightarrow$ существуют $s_{i+1}, \ldots, s_m$ такие, что процесс $p(s_1, \ldots, s_m)$ допустим.

Т е о р е м а 6.1. *Функция переходов компонент однозначно определяет функцию переходов многокомпонентной системы.*

Доказательство очевидно.

Компонента S_i называется *свободной*, если для любых p, s_1, ..., s_{i-1} $\delta_i(p, s_1, \ldots, s_{i-1}) = S_i$ или ϕ. Компонента называется *детерминированной*, если $\delta_i(p, s_1, \ldots, s_{i-1})$ состоит не более чем из одного элемента. Система S называется *квазидетерминированной*, если каждая ее компонента либо свободна, либо детерминирована.

Т е о р е м а 6.2. *Всякая дискретная система S имеет полную гомоморфную квазидетерминированную реализацию.*

Рассмотрим сначала случай, когда множество S не более, чем счетно. Пусть S' — полная автоматная реализация системы S, которая существует по теореме 3.1, $\gamma\colon S' \to S$ — реализующий гомоморфизм, а δ' — функция переходов системы S'. Из доказательства теоремы 3.1 видно, что в качестве S' можно взять систему со счетным числом состояний. Упорядочим S' линейно по типу натурального ряда отношением $\leqslant$

Пусть для любых $s', s'' \in S'$ $\varphi(s', s'') = \{ s \in \delta'(s'') \mid s' \leqslant s \}$. Рассмотрим автоматную систему $S'' = (S')^2$, определив ее функцию переходов δ'' равенством $\delta''(s', s'') =$ (**если** $\delta'(s'') = \phi$, **то** ϕ, **иначе**, **если** $\varphi(s', s'') = \phi$, **то** $S' \times \{\min \delta'(s'')\}$, **иначе** $S' \times \{\min \varphi(s', s'')\}$). Положим $(S'')_0 = S' \times (S')_0$. Отображение $\gamma'\colon S'' \to S'$, определенное равенством $\gamma'(s', s'') = s''$, есть гомоморфизм, а композиция $\gamma'' = \gamma' \circ \gamma$ дает искомый реализующий гомоморфизм. Для несчетного множества S доказательство дословно повторяется, если S' упорядочить так, чтобы оно стало вполне упорядоченным множеством, что можно сделать с помощью известной теоремы Цорна.

При перестановке компонент многокомпонентной системы и соответствующем изменении функции переходов получаются системы, изоморфные исходной, но с другим упорядочением компонент. Следует заметить, что при этом свойство квазидетерминированности может потеряться. Например, если $S = S_1 \times S_2$ — автоматная система и $\delta(s_1, s_2) = (s_1', f(s_1'))$, где $s_1' \in S_1$ произвольно, а $f\colon S_1 \to S_2$, то система S квазидетерминирована, однако если отображение f не является отображением на все множество S, то после перестановки компонент свойство квазидетерминированности теряется. С другой стороны, если отображение f — взаимно однозначное отображение S_1 на S_2, то при перестановке роли компонент поменяются

и система останется квазидетерминированной. Для того чтобы понятие квазидетерминированной системы сделать независимым от упорядочения компонент, расширим это понятие, считая систему квазидетерминированной, если она становится такой после перестановки компонент.

Свободные компоненты обычно представляют связь системы с внешним окружением, поэтому они изменяются независимо от того, что происходит внутри системы. При построении композиции некоторых систем те из них, которые моделируют внешнее окружение других, становятся компонентами композиции, и на изменение свободных компонент накладываются ограничения, характеризующие обратные связи.

Для автоматных систем функции переходов компонент могут быть определены другим способом. Именно, полагаем δ_i: $S_1 \times \ldots \times S_m \times S_1 \times \ldots \times S_{i-1} \to 2^{S_i}$. При этом процесс $p(s_1, \ldots, s_m)(s_1', \ldots, s_m')$ допустим $\Leftrightarrow s_i' \in \delta_i(s_1, \ldots, s_m, s_1', \ldots, s_{i-1}')$ $(i = 1, \ldots, m)$. Таким образом, $\delta_i(s_1, \ldots, s_m, s_1', \ldots, s_{i-1}')$ представляет возможные значения i-й компоненты в текущий момент времени при условии, что в предыдущий момент времени система находилась в состоянии $(s_1, \ldots, s_m)$, а в текущий момент времени первые $i-1$ компонент перешли в состояния $s_1', \ldots, s_{i-1}'$ соответственно.

Переходя к многоуровневым системам, заметим, что декартово произведение рассматривается как многоместная операция, и расстановка скобок в произведении меняет соответствующее множество. Например, множества $(S_1 \times S_2 \times S_3) \times S_4$ и $S_1 \times (S_2 \times S_3) \times S_4$ считаются разными. Компоненты $S_1, \ldots, S_m$ многоуровневой системы $S \subset S_1 \times \ldots \times S_m$ называются *компонентами верхнего уровня*.

Предположим, что двигаясь вниз по уровням компонент, в каждой из них можно дойти до самого нижнего уровня. Если $T_1, \ldots, T_n$ – все компоненты нижнего уровня, то система S изоморфна одноуровневой системе с множеством состояний, содержащимся в множестве $T_1 \times \ldots \times T_n$. Само множество состояний системы S содержится в множестве, которое получается некоторой расстановкой скобок в произведении $T_1 \times \ldots \times T_n$. Соответствующее выражение называется *многоуровневой структурой* системы S. Каждую компоненту некоторого уровня можно, двигаясь сверху вниз, представить в виде композиции компонент следующего уровня. Если зафиксировать такие представления, то получим многоуровневую композицию, представляющую систему S.

Рассмотрим преобразования многоуровневых систем, не выводящие их из класса изоморфных систем.

1. Поднятие уровня компонент, состоящее в опускании некоторой пары скобок в многоуровневой структуре системы.

2. Понижение уровня компонент, состоящее в том, что в структуру системы добавляется новая пара скобок.

3. Перестановка компонент одного и того же уровня, непосредственно подчиненных, т.е. являющихся компонентами, одной и той же компоненте верхнего уровня.

4. Дублирование (удвоение) компоненты состоит в том, что некоторая компонента повторяется на том же уровне и с той же непосредственной подчиненностью, что и исходная компонента. При этом функция переходов и множество допустимых состояний представляются таким образом, что

в допустимых процессах и состояниях новая и старая компоненты всегда равны. Такие компоненты называются также *отождествленными*.

5. Удаление одной из отождествленных компонент — преобразование, обратное к преобразованию 4.

Пусть $T_1, ..., T_m, T$ — компоненты одного и того же уровня и одинаковой подчиненности. Компоненту T будем называть *зависящей* от компонент $T_1, ..., T_m$, если во всех допустимых состояниях ее значение однозначно определяется состояниями компонент $T_1, ..., T_m$.

6. Добавление зависимых компонент состоит в следующем. Пусть $S_i \subset T_1 \times ... \times T_m$ — одна из компонент некоторого уровня. Заменим S_i на $S_i' \subset T_1 \times ... T_m \times T$ и переопределим функцию переходов и множество допустимых состояний таким образом, чтобы во всех допустимых процессах и всех допустимых состояниях этой новой системы значение компоненты T было бы равно некоторой заданной функции от состояний компонент $T_1, ..., T_m$.

7. Удаление зависимых компонент — преобразование, обратное преобразованию 6.

Легко заметить, что преобразования 4 и 5 являются частными случаями преобразований 6 и 7 соответственно.

Две системы называются *структурно изоморфными*, если одну можно получить из другой при помощи последовательности указанных преобразований. Переход от системы S к структурно изоморфной системе с помощью только преобразований 1 – 3 называется *реструктурированием* системы S.

Одним из основных способов определения композиции дискретных систем является введение соотношений между компонентами прямого произведения. Пусть заданы системы $(S_1, F_1), ..., (S_m, F_m)$, а $R(s_1, ..., s_m)$ есть отношение между элементами множеств $S_1, ..., S_m$, т.е. $R \subset S_1 \times ... \times S_m$. *Композицией* дискретных систем $S_1, ..., S_m$, определенной отношением R, или *R-композицией*, называется подсистема прямого произведения $S_1 \otimes ... \otimes S_m$, множество допустимых состояний которой совпадает с R, а множество допустимых процессов $F = \mathbf{P}(R) \cap (F_1 \otimes ... \otimes F_m)$, т.е. $p_1 \otimes ... \otimes p_m$ допустим $\Leftrightarrow$ все процессы $p_1, ..., p_m$ допустимы, и для всех t таких, что $0 \leqslant t \leqslant |p|$, имеет место $R(p_1(t), ..., p_m(t))$. Очевидно, что R-композиция определена также и для систем с непрерывным временем. Простейшим вариантом отношения R является конъюнкция равенств состояний некоторых компонент $s_{i_1} = s_{j_1} \wedge ... \wedge s_{i_k} = s_{j_k}$. Равенства, входящие в это отношение, называются *уравнениями связей*, а соответствующая композиция — *композицией, определенной системой связей*. Если $S_1, ..., S_m$ сами являются многокомпонентными, то в уравнениях связей будем разрешать использование состояний, являющихся компонентами состояний систем $S_1, ..., S_m$.

Многокомпонентная система называется *полуоткрытой*, если среди ее компонент выделены некоторые свободные компоненты, называемые *входными*. Входные компоненты, или входы, предназначены для получения информации от внешнего мира. Полуоткрытая система называется *открытой*, если среди ее компонент выделены компоненты, называемые *выходными*. Компоненты, которые не являются входными или выходными, называются *внутренними*. Всякая полуоткрытая система структурно изоморфна двухкомпонентной системе $S \subset X \times A$, где X — произведение

всех свободных компонент. Открытая система структурно изоморфна трехкомпонентной системе $S \subset X \times A \times Y$, где X – произведение входных, A – произведение внутренних, Y – произведение выходных компонент. Для получения такого представления в случае, если система не имеет внутренних компонент или некоторые из входных являются внутренними, следует воспользоваться дублированием компонент. Для открытых систем может быть введен особый вид композиции, определяемой системой связей, – сети.

Пусть $S_1, ..., S_m$ – открытые системы. *Сетью*, построенной из $S_1, ..., S_m$, называется композиция систем $S_1, ..., S_m$, определенная системой связей R и удовлетворяющая некоторым дополнительным ограничениям. Для того чтобы сформулировать эти ограничения, рассмотрим отношение эквивалентности на множестве компонент систем $S_1, ..., S_m$. Две компоненты T_1 и T_2 называются *отождествленными*, если равенство $s_1 = s_2$ их состояний является следствием из уравнений связей.

Потребуем теперь выполнения следующих ограничений.

1. В каждом классе отождествленных компонент либо все входные, либо только одна выходная, а остальные входные.

2. Множества состояний отождествленных компонент либо совпадают, либо все содержат множество состояний выходной компоненты.

Смысл введенных ограничений становится ясным, если представить сеть графически. Это можно сделать с помощью ориентированного графа с отмеченными дугами. Вершины этого графа делятся на два класса – узлы и компоненты. Первые представляются на рисунке точками, вторые – например, прямоугольниками. Узлы находятся во взаимно однозначном соответствии с классами отождествленных компонент. Дуга соединяет узел с компонентой (компоненту с узлом), если один из входов (выходов) этой компоненты содержится в соответствующем классе, и отмечается символом переменной, принимающей значения в множестве состояний этого входа (выхода).

Если сеть имеет классы отождествленных компонент, состоящие только из входных компонент, то, выбрав из каждого такого класса по одному представителю, их можно объявить входными компонентами сети. В качестве выходов можно объявить любые из выходов компонент сети. После соответствующего реструктурирования сеть может быть объявлена открытой или полуоткрытой системой. Повторяя операцию построения новых сетей из уже построенных, получаем многоуровневые сети. Такой подход обычно используется при описании сложных многокомпонентных систем.

На рис. 1.7 изображена сеть из трех компонент. Состояние этой сети имеет вид $v = (v_1, v_2, v_3)$, где $v_1 = (x_1, x_2, s_1, y_1)$, $v_2 = (x_3, x_4, s_2, y_2)$, $v_3 = (x_5, x_6, x_7, s_3, y_3, y_4)$. Ее компоненты в текущий момент времени должны удовлетворять следующим условиям:

$$s_1 \in \delta_1(p_1, x_1, x_2),$$

$$s_2 \in \delta_2(p_2, x_3, x_4),$$

$$s_3 \in \delta_3(p_3, x_5, x_6, x_7),$$

$$y_1 \in \delta_4(p_1, x_1, x_2, s_1),$$

$$y_2 \in \delta_5(p_2, x_3, x_4, s_2),$$

$$
\begin{aligned}
&y_3 \in \delta_6(p_3, x_5, x_6, x_7, s_3),\\
&y_4 \in \delta_7(p_3, x_5, x_6, x_7, s_3),\\
&x_3 = x_2,\\
&x_1 = y_3,\\
&x_5 = y_1,\\
&x_6 = x_7 = y_2.
\end{aligned}
\tag{6.1}
$$

Здесь p_1, p_2, p_3 – процессы функционирования компонент, предшествующих текущему моменту времени, δ_1, ..., δ_7 – функции переходов компонент.

Если компоненты сети являются квазидетерминированными системами с детерминированными внутренними и выходными компонентами,

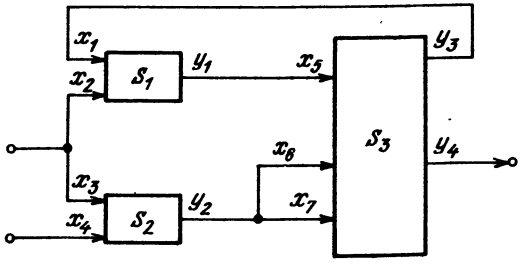


Рис. 1.7

то сеть, вообще говоря, может не быть квазидетерминированной. Действительно, в случае квазидетерминированности систем S_1, S_2, S_3 утверждения о принадлежности состояний и выходов множествам, определяемым функциями переходов компонент, превращаются в равенства, а система соотношений (6.1) становится системой уравнений. Наличие обратных связей приводит к тому, что решение этой системы может не быть единственным.

Рассмотренные композиции дискретных систем соответствуют случаю конечного множества линейно упорядоченных компонент. В общем случае можно рассматривать семейство систем $(S_i)_{i \in I}$ и прямое произведение $S = \underset{i \in I}{\otimes} S_i$, которое определяется следующим образом. Элементами множества S являются функции $s\colon I \to \bigcup\limits_{i \in I} S_i$ такие, что для любого $i \in I$ $s(i) \in S_i$. Множество $F = \underset{i \in I}{\otimes} F_i$ допустимых процессов системы S состоит из всех процессов p в пространстве S таких, что для всякого $i \in I$ процесс p_i, определенный равенством $p_i(t) = s_t(i)$, где $s_t = p(t)$ $(0 \leqslant t \leqslant |p|)$, принадлежит множеству F_i допустимых процессов системы S_i. Процесс p_i называется i-й проекцией процесса p. Сам процесс p однозначно определяется своими проекциями и обозначается $p = \underset{i \in I}{\otimes} p_i$.

При определении функций переходов компонент композиции семейства систем следует учитывать порядок, который должен быть задан на

множестве индексов I этого семейства. Указанный порядок может быть частичным, а функции переходов компоненты с бо́льшим индексом могут зависеть от текущего состояния компонент только с меньшими индексами. С учетом этого порядка можно определить свободные и детерминированные компоненты, квазидетерминированные, открытые и полуоткрытые системы. Следует заметить, что в случае частичного порядка функции переходов компонент не всегда существуют. Например, если состояния двух компонент всегда равны, то индексы этих компонент должны быть сравнимыми, поскольку состояние любой из них зависит от состояния другой.

§ 7. Автоматы

Важным частным случаем открытой и полуоткрытой систем является автомат.

Полуоткрытая автоматная система $X \times A$ называется *X-автоматом*, если функция переходов компоненты A $\delta_2(x, a, x')$ не зависит от x', а множество начальных состояний имеет вид $X \times A_0$, где $A_0 \subset A$. Функция $\delta(a, x) = \delta_2(x, a, x')$ называется *функцией переходов* автомата, а сам автомат обозначается так же, как и его внутренняя компонента A. X-автомат однозначно определяется множествами X, A, A_0 и функцией переходов δ. Поэтому его часто определяют как четверку (X, A, A_0, δ). Если $A_0 = A$, то X-автомат называют *неинициальным* и отождествляют с тройкой (X, A, δ). Если функция δ однозначна, то автомат называют *детерминированным*; если функция δ однозначна и всюду определена, то автомат называют *полностью определенным*. Часто X-автоматом называют неинициальный, детерминированный, полностью определенный автомат, а рассмотренное выше определение — как обобщение с добавлением атрибутов "инициальный", "недетерминированный", "частично определенный".

Поскольку всякая полуоткрытая система структурно изоморфна двухкомпонентной, следующая теорема фактически устанавливает связь между произвольными полуоткрытыми системами и X-автоматами.

Теорема 7.1. *Всякая двухкомпонентная полуоткрытая автоматная система $S \subset X \times A$ является подсистемой полной гомоморфной модели X-автомата B. При этом если S квазидетерминирована, то B детерминирован.*

Рассмотрим X-автомат $B = (X \cup \{x_0\}) \times A$, где $x_0 \notin X$, определив его функцию переходов формулой $\delta((x, a), x') = (x', \delta_2'(x, a, x'))$, где $\delta_2'(x, a, x')$ — произвольное продолжение функции δ_2 переходов второй компоненты системы S на все множество $(X \cup \{x_0\}) \times A \times X$, удовлетворяющее условию $\delta_2'(x_0, a, x) = a$. Положим $B_0 = \{x_0\} \times A$. Определим отображение γ: $X \times ((X \cup \{x_0\}) \times A) \to X \times A$, полагая $\gamma(x, (x', a)) = (x, \delta_2'(x', a, x))$. Это отображение определяет гомоморфизм автомата B в свободную систему с множеством состояний $X \times A$. Образ S' автомата B при этом гомоморфизме является его полной моделью.

Покажем, что S' содержит S в качестве подсистемы. Действительно, пусть $q = (x_1, a_1) \ldots (x_m, a_m)$ — допустимый процесс системы S. Тогда $a_{i+1} \in \delta_2(x_i, a_i, x_{i+1})$ $(i = 1, \ldots, m-1)$, а процесс $p = (x_1, (x_0, a_0)) \times (x_2, (x_1, a_1)) \ldots (x_m, (x_{m-1}, a_{m-1}))$ есть допустимый процесс функ-

ционирования автомата B. Но $\gamma(p) = q$, и, следовательно, q допустим в S. Вторая часть теоремы очевидна.

Трехкомпонентная открытая система $S \subset X \times A \times Y$ называется *X–Y-автоматом*, если компонента $X \times A$ системы $S' \subset (X \times A) \times Y$, полученной из S соответствующим реструктурированием, является X-автоматом, функция переходов компоненты Y зависит только от состояний входной и внутренней компонент в текущий момент времени: $\delta_3(x, a, y, x', a') = \lambda(a', x')$, а множество допустимых состояний (a, x, y) определяется условием $y = \lambda(a, x)$. Функция переходов X-автомата $X \times A$ и функция λ называются *функцией переходов* и *функцией выходов* автомата S соответственно, а сам X–Y-автомат обозначается обычно так же, как и его внутренняя компонента A. Поскольку X–Y-автомат однозначно определяется множествами X, A, A_0, Y (A_0 – множество начальных состояний внутренней компоненты) и функциями δ и λ, то его обычно определяют как шестерку $(X, A, A_0, Y, \delta, \lambda)$ или, если $A_0 = A$, как пятерку $(X, A, Y, \delta, \lambda)$. Так же как и для X-автоматов, вводятся понятия инициальности, детерминированности и полной определенности (требование полной определенности относится к функции переходов и к функции выходов).

Т е о р е м а 7.2. *Всякая открытая автоматная система $S \subset X \times A \times Y$ является подсистемой полной гомоморфной модели X–Y-автомата B. Если S квазидетерминирована, то B детерминирован.*

Доказательство аналогично доказательству теоремы 7.1.

В дальнейшем, если это не оговорено особо, будут рассматриваться только неинициальные, детерминированные, полностью определенные автоматы.

При рассмотрении гомоморфизмов автоматов кроме требования сохранения допустимости процессов добавляются следующие условия. Отображение γ: $S \to S'$ X–Y-автомата $S \subset X \times A \times Y$ в X'–Y'-автомат $S' \subset X' \times A' \times Y'$ называется *гомоморфизмом*, если это отображение есть гомоморфизм системы S в S' и, кроме того, $\gamma(x, a, y) = (\gamma_1(x), \gamma_2(a), \gamma_3(y))$, где γ_1: $X \to X'$, γ_2: $A \to A'$, γ_3: $Y \to Y'$. Очевидно, что тройка $(\gamma_1, \gamma_2, \gamma_3)$ определяет гомоморфизм автомата $(X, A, Y, \delta, \lambda)$ в автомат $(X', A', Y', \delta', \lambda') \Leftrightarrow \gamma_2(\delta(a, x)) = \delta'(\gamma_2(a), \gamma_1(x))$, $\gamma_3(\lambda(a, x)) = \lambda'(\gamma_2(a), \gamma_1(x))$.

С каждым X-автоматом связывается семейство языков в алфавите X, представленное этим автоматом. Язык $L(a_0, A_1)$, представленный в X-автомате A при начальном состоянии a_0 множеством заключительных состояний A_1, – это множество всех слов $p \in X^*$, для которых существует допустимый процесс $p \otimes q$ такой, что $a_0 \xrightarrow{q} a_1 \in A_1$ или $p = e$ и $a_0 \in A_1$. Это определение годится как для детерминированных, так и для недетерминированных X-автоматов. В случае детерминированных автоматов процесс q определяется однозначно.

Т е о р е м а 7.3 (основная теорема теории конечных автоматов). *Класс языков, которые могут быть представлены в конечных X-автоматах, совпадает с классом регулярных языков в алфавите X.*

Всякий конечный X-автомат есть конечная автоматная система, и $L(a_0, A_1) \setminus \{e\} = \gamma(L(X \times \{a_0\}, X \times A_1))$, где $\gamma(x, a) = x$. Поэтому $L(a_0, A_1) \setminus \{e\}$ конечно-автоматен и, следовательно, регулярен. Эта часть теоремы носит название *теоремы анализа.* Алгоритм анализа может быть получен применением метода анализа дискретных автоматных систем или прямым построением соответствующих уравнений и их решением.

Обратное утверждение — это *теорема абстрактного синтеза* конечных автоматов. Его доказательство можно провести с использованием теоремы синтеза конечных автоматных систем. Пусть R — регулярный язык без e. Синтезируем конечную автоматную систему S, порождающую этот язык. Пусть $R = \gamma(L(S_1, S_2))$. Рассмотрим X-автомат $A = 2^S$, определив его функцию переходов равенством $\delta(T, x) = \{s \mid$ существует $s' \in T$ такое, что $\gamma(s') = x, s' \to s\}$, $T \subset S$. Положим $A_1 = \{T \subset S \mid T \cap S_2 \neq \phi\}$. Тогда $L(S_1, A_1) = R$. Действительно, $p \in \gamma(L(S_1, S_2)) \Longleftrightarrow s_1 \to \ldots \to s_n$, $s_1 \in S_1$, $s_n \in S_2$, $\gamma(s_1 \ldots s_n) = x_1 \ldots x_n = p \Longleftrightarrow ((x_1, T_1) \to (x_2, T_2) \to \ldots \to (x_n, T_n)$, $s_1 \in T_1 = S_1$, $s_2 \in T_2, \ldots, s_n \in T_n \Rightarrow T_n \cap S_2 \neq \phi \Rightarrow T_n \in A_1) \Longleftrightarrow p \in L(S_1, A_1)$. Если $e \in R$, то сначала представим $R \setminus \{e\}$ в X-автомате B, считая $R \setminus \{e\} = L(b_0, B_1)$, затем построим автомат A, добавив к состояниям автомата B новое состояние a_0 и положив $\delta(a_0, x) = \delta(b_0, x)$. Получим $R = L(a_0, A_1)$, где $A_1 = B_1 \cup \{a_0\}$.

Рассматривают также представление языков в X—Y-автоматах. Для этого с каждым автоматом $(X, A, Y, \delta, \lambda)$ связывают язык $L(a, y)$, полагая, что $p \in L(a, y) \Longleftrightarrow$ существует допустимый процесс $p \otimes q \otimes r$ такой, что $a \overset{q}{\to} a'$, $y \overset{r}{\to} y'$.

О языке $L(a, y)$ говорят, что он представлен в автомате A при начальном состоянии a выходным сигналом y. Для языков, представимых в X—Y-автоматах, также может быть доказан аналог основной теоремы: *язык R представим в X—Y-автомате $\Longleftrightarrow R$ регулярен.*

Пусть A есть X—Y-автомат, а λ — его функция выходов. Если $\lambda(a, x) = \mu(a)$ не зависит от входного сигнала x, то A называется *автоматом с задержкой*, в общем случае — *автоматом Мили. Автомат Мура* — это автомат Мили, для которого $\lambda(a, x) = \mu(\delta(a, x))$, т.е. выход зависит от состояния в следующий момент времени.

X—Y-автоматы как открытые системы можно использовать для построения сетей из автоматов. При этом как входной, так и выходной алфавиты могут быть многокомпонентными, т.е. представленными в виде декартовых произведений. Поэтому для построения сетей автомат реструктурируется путем перенесения входных и выходных компонент на верхний уровень.

Сеть из автоматов называется *открытой*, если она имеет по крайней мере один класс отождествленных компонент, состоящий из одних входных компонент. В противном случае сеть называется *замкнутой.*

Теорема 7.4. *Открытая сеть из автоматов структурно изоморфна недетерминированному X—Y-автомату.*

Пусть S есть сеть, построенная из автоматов $A_1, ..., A_m$. Реструктурируем эту сеть путем перехода от структуры $((u_1, a_1, v_1), ..., (u_m, a_m, v_m))$ к структуре $((x_1, ..., x_k), (a_1, ..., a_m), (v_1, ..., v_m))$, где $x_1, ..., x_k$ – входные компоненты, взятые по одной из всех классов отождествленных компонент, не содержащих выходных. Выбрав соответствующим образом входной алфавит X и взяв в качестве Y произведение всех выходных алфавитов, получим X–Y-автомат.

Недетерминированность построенного автомата является следствием возможности обратных связей, как показано на примере, рассмотренном в предыдущем параграфе. Если все автоматы, из которых построена сеть, являются автоматами с задержкой, то сеть является детерминированным автоматом. Это условие можно ослабить, получив следующий результат.

Т е о р е м а 7.5. *Открытая сеть из автоматов Мили структурно изоморфна автомату Мили, если каждый цикл в этой сети проходит по крайней мере через один автомат с задержкой.*

Действительно, переход сети из состояния $(..., (u_i, a_i, v_i), ...)$ в следующее состояние $(..., (u_i', a_i', v_i'), ...)$ определяется системой уравнений, состоящей из уравнений

$$a_i' = \delta_i(a_i, u_i),$$

$$v_i' = \lambda_i(a_i', u_i'),$$

и уравнений

$$v_i = \lambda_i(a_i),$$

$$v_i' = \lambda_i(a_i'),$$

для автоматов с задержкой и уравнений связей вида $x_i = x_j$ и $x_i = y_j$, где x_i – входные, y_j – выходные компоненты. Уравнения связей исключаются путем подстановок представителей классов отождествленных компонент в уравнения для состояний и выходов. При этом, если класс содержит выходную компоненту, то именно она и выбирается в качестве представителя для этого класса. Уравнения для выходов разбиваются на уравнения для компонент этих выходов. Например, если $v_i = (y_{i1}, ..., y_{ik})$, то уравнение $v_i' = \lambda_i(\delta_i(a_i, u_i), v_i')$ заменяется системой

$$y_{ij}' = \lambda_{ij}(\delta_i(a_i, u_i), v_i'), \qquad j = 1, ..., k.$$

После этого производятся подстановки выходов в правые части уравнений состояний. Эти подстановки должны привести к полному исключению выходов, ибо в противном случае обнаруживается цикл, не содержащий автомата с задержкой. После исключения выходов для произвольного набора значений свободных входных компонент состояния a_i' определяются однозначно, а их значения однозначно определяют значения выходных компонент.

В дальнейшем сети из автоматов будут рассматриваться с точностью до структурного изоморфизма.

Любая современная система преобразования информации на определенном уровне подробности описания может быть представлена как сеть из автоматов. При этом можно в качестве компонент рассматривать только автоматы Мили или Мура, которые могут служить гомоморфной реализацией произвольных квазидетерминированных автоматных открытых систем. Поэтому проектирование технического оборудования систем преобразования информации в качестве одного из основных этапов содержит этап построения сети из автоматов, реализующей некоторую заданную дискретную систему. Поскольку любая дискретная система может быть реализована автоматом, задача сводится к построению сети из элементарных автоматов, реализующей заданный автомат.

Элементарный автомат определяется технологией изготовления аппаратуры и элементной базой, применяемой в различных отраслях промышленности, производящей технические средства преобразования информации. На первых порах все сводилось к синтезу автоматов на элементах И, ИЛИ, НЕ и триггерах или задержках. Затем с появлением интегральных схем стали использоваться многовходовые логические элементы, элементы арифметико-логических устройств, различного типа запоминающих регистров и т.п. В настоящее время технология сверхбольших интегральных схем предполагает использование достаточно крупных блоков вплоть до отдельных процессоров. В этой области еще много проблем и нерешенных вопросов, связанных с математическими моделями, однако ясно, что многое в конечном счете может быть сведено к задаче построения сети из элементарных автоматов, реализующей заданный автомат. Эта задача называется *задачей структурного синтеза*, и мы рассмотрим ее для некоторого класса наборов элементарных автоматов, которые будем называть *каноническими базисами.*

Канонический базис характеризуется некоторым фиксированным конечным структурным алфавитом Z. Входные и выходные сигналы всех элементарных автоматов являются декартовыми степенями этого алфавита. Чаще всего в качестве структурного алфавита выбирается двоичный алфавит $Z = \{0, 1\}$. Элементы канонического базиса делятся на два класса – функциональные элементы и элементы памяти.

Функциональные элементы характеризуются тем, что их функция выходов зависит только от значения входного сигнала в текущий момент времени. Поскольку состояние внутренней компоненты функционального элемента не влияет на окружающую среду, ею можно пренебречь и рассматривать функциональный элемент как двухкомпонентную открытую систему. Сеть из функциональных элементов без циклов называется *функциональной схемой* или схемой без памяти. Выходы функциональной схемы так же, как и выходы функциональных элементов, зависят только от значений входных компонент в текущий момент времени и являются суперпозициями функций, соответствующих элементам, из которых построена схема. Все эти функции имеют тип $\lambda\colon Z^n \to Z$. Будем предполагать, что функциональные элементы канонического базиса обладают свойством *функциональной полноты.* Это свойство означает, что любая функция $\lambda\colon Z^n \to Z$ может быть представлена в виде суперпозиции функций, реализуемых функциональными элементами. Критерии функциональной полноты хорошо извест-

ны, и мы надеемся, что читатель знает знаменитую теорему Поста о полноте систем булевых функций, соответствующую случаю $Z = \{0, 1\}$.

Относительно элементов памяти будем предполагать, что они содержат по крайней мере один элемент D, удовлетворяющий следующим условиям. D является U–V-автоматом с задержкой. Число внутренних состояний автомата D не меньше двух. Функция выходов $\mu_D(a)$ определяет взаимно однозначное отображение множества D на V. Для каждой пары $a, a' \in D$ существует входной сигнал $u \in U$ такой, что $\delta_D(a, u) = a'$. Автомат, удовлетворяющий перечисленным условиям, называется *автоматом с полной системой переходов и выходов.*

X'–Y'-автомат A' называется *подавтоматом* X–Y-автомата A, если $X' \subset X$, $Y' \subset Y$, $A' \subset A$, а функции переходов и выходов являются ограничениями функций переходов и выходов автомата на соответствующие подмножества.

Теорема 7.6. *Всякий конечный автомат изоморфен подавтомату сети, построенной из автоматов канонического базиса.*

Пусть A есть X–Y-автомат. Рассмотрим взаимно однозначные отображения $\gamma_1\colon X \to Z^m$, $\gamma_2\colon A \to D^n$, $\gamma_3\colon Y \to Z^k$, которые можно построить, выбрав подходящие значения k, m и n. Построим двухуровневую сеть из автоматов канонического базиса. Структура этой сети изображена на рис. 1.8. Компоненты S_1 и S_2 представляют собой сети из функциональных элементов без памяти. Сеть S_1 реализует функцию $u = \varphi(z, v)$, сеть S_3 – функцию $z' = \psi(z, v)$, где $z = (z_1, \ldots, z_m)$, $u = (u_1, \ldots, u_n)$, $v = (v_1, \ldots, v_k)$, $z' = (z'_1, \ldots, z'_k)$, $z_i, z'_i \in Z$, $u_i \in U$, $v_i \in V$. Сеть S_2 представляет собой прямое произведение n экземпляров автомата D. Отбросив все зависимые компоненты, кроме выходов сети S_3, получим, что сеть S структурно изоморфна Z^m–Z^k-автомату с внутренней компонентой D^n. Функция переходов δ_S этого автомата определяется равенством $\delta_S(d, z) = d'$, где $d = (d_1, \ldots, d_n)$, $d' = (d'_1, \ldots, d'_n)$, $d_i, d'_i \in D$, $d'_i = \delta_D(d_i, u_i)$, $(u_1, \ldots, u_n) = \varphi(z, v)$, а функция выходов λ_S равенством $\lambda_S(d, z) = \psi(v, z)$, где $v_i = \lambda_D(d_i, u_i)$. В силу функциональной полноты элементов без памяти функции φ и ψ могут быть выбраны произвольным образом. В частности, так, чтобы отображение $\gamma(x, a, y) = (\gamma_1(x), \gamma_2(a), \gamma_3(y))$ было изоморфизмом автомата A и S. Для этого достаточно выполнения равенств $\gamma_2(\delta_A(a, x)) = \delta_S(\gamma_2(a), \gamma_1(x))$, $\gamma_3(\lambda_A(a, x)) = \lambda_S(\gamma_2(a), \gamma_1(x))$, которые после распространения функций δ_D и μ_D на векторы могут быть переписаны в виде

$$\gamma_2(\delta_A(a, x)) = \delta_D(\gamma_2(a), \varphi(\mu_D(d), \gamma_1(x))),$$

$$\gamma_3(\lambda_A(a, x)) = \psi(\mu_D(d), \gamma_1(x)).$$

Для определения функций φ и ψ на произвольном наборе (v, z) выберем пару (v, z) так, что $v = \mu_D(d)$ для некото-

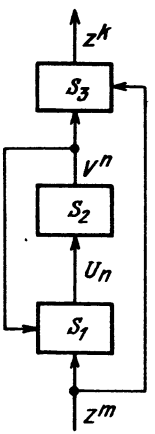


Рис. 1.8

рого $d = \gamma_2(a)$, а $z = \gamma_1(x)$ для некоторого x. В силу взаимной однозначности d, x и a определены однозначно. Поэтому однозначно определены левые части равенств. Второе равенство сразу дает значение ψ, а значение $\varphi(v, z)$ должно быть выбрано так, чтобы выполнялось первое равенство. В силу свойства функции δ_D такое значение существует. Определив функции φ и ψ на других значениях (v, z) произвольным образом, получим, что γ есть изоморфизм A в B. Образ автомата A при этом изоморфизме есть подавтомат автомата S. Теорема доказана.

§ 8. Дискретные преобразователи

Использование понятия дискретной системы в качестве модели вычислительного устройства предполагает, что в множестве состояний выделены не только начальные, но и заключительные состояния. Переход системы в заключительное состояние означает окончание вычислений. Система S, для которой заданы множество S_0 начальных состояний и множество S_1 заключительных состояний, называется *настроенной*. Предполагается, что S_0 содержится в множестве допустимых начальных состояний, но, вообще говоря, не совпадает с ним. Пара (S_0, S_1) называется *настройкой* системы S.

Процессом вычислений настроенной системы называется допустимый процесс p, который начинается в множестве S_0 и для любого t $(0 < t < |p|)$ $p(t) \notin S_1$. Процесс вычислений ненулевой длительности называется *терминальным*, если он оканчивается в множестве S_1.

Со всякой настроенной системой S связывается функция $f_S\colon S_0 \to 2^{S_1}$, которая определяется следующим образом. Состояние $s' \in f_S(s) \Longleftrightarrow$ существует терминальный процесс p такой, что $s \xrightarrow{p} s'$. Относительно функции f_S будем говорить, что она *вычисляется* системой S. Если система S детерминирована, то функция f_S определяет однозначное частичное отображение из S_0 в S_1, и ее можно отождествить с этим отображением. Функция f_S может быть однозначной и тогда, когда система S не является детерминированной. В этом случае настроенную систему S будем называть *глобально детерминированной*. Если всякий процесс вычислений настроенной глобально детерминированной системы можно продолжить до терминального, то функция f_S, которую эта система вычисляет, является однозначной всюду определенной функцией.

Функция f_S определяет также отношение между элементами множеств S_0 и S_1, которое будет обозначаться выражением $s \xrightarrow{f_S} s'$, $s \xrightarrow{f_S} s' \Leftrightarrow s' \in f_S(s)$. Суперпозиция $f \circ g$ многозначных функций $f\colon S_0 \to 2^{S_1}$ и $g\colon S_1 \to 2^{S_2}$ – это суперпозиция соответствующих отношений: $s \xrightarrow{f \circ g} s' \Leftrightarrow$ для некоторого $s'' \in S_1$ имеет место $s \xrightarrow{f} s'' \xrightarrow{g} s'$.

Пусть S и T – настроенные системы, (S_0, S_1) и (T_0, T_1) – их настройки и S – реализация T с помощью реализующего отображения $\gamma\colon \mathbf{P}(S) \to \mathbf{P}(T)$. Будем говорить, что реализация γ *согласуется* с настройкой систем S и T, если образ $\gamma(p)$ процесса p терминален тогда и только тог-

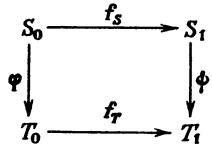
да, когда p терминален и существуют функции $\varphi\colon S_0 \to T_0$ и $\psi\colon S_1 \to T_1$ такие, что для любого терминального процесса p системы S из $s \xrightarrow{p} s'$ следует $\varphi(s) \xrightarrow{\gamma(p)} \psi(s')$. Функции φ и ψ называются *кодирующими функциями* реализации S.

Теорема 8.1. *Если S есть реализация системы T, согласованная с настройкой, γ – реализующее отображение, а φ и ψ – кодирующие функции, то $f_T \subset \varphi^{-1} \circ f_S \circ \psi$. Если при этом реализация S полна, то $f_T = \varphi^{-1} \circ f_S \circ \psi$.*

Действительно, $(t, t') \in \varphi^{-1} \circ f_S \circ \psi \Leftrightarrow t \xrightarrow{\varphi^{-1}} s \xrightarrow{p} s' \xrightarrow{\psi} t'$, где p терминален $\Rightarrow \varphi(s) \xrightarrow{\gamma(p)} \psi(s') \Rightarrow \gamma(p)$ терминален и $t \xrightarrow{\gamma(p)} t' \Rightarrow t \xrightarrow{f_T} t'$. Обратно, если реализация полна, то $t \xrightarrow{f_T} t' \Rightarrow t \xrightarrow{q} t'$, q терминален $\Rightarrow q = \gamma(p)$, $s \xrightarrow{p} s'$, p терминален $\Rightarrow \varphi(s) \xrightarrow{\gamma(p)} \psi(s') \Rightarrow t \xrightarrow{\varphi^{-1}} s \xrightarrow{p} s' \xrightarrow{\psi} t' \Rightarrow (t, t') \in \varphi^{-1} \circ f_S \circ \psi$.

Если S – реализация системы T, согласованная с настройкой, то будем говорить, что S *частично вычисляет* отображение f_T. Если же эта реализация полна, то S вычисляет f_T (при заданном реализующем отображении и кодирующих функциях). Такая терминология оправдана, ибо если система S задана конструктивно или физически в виде действующего вычислительного устройства, то ее можно использовать для вычисления отображения f_T. Действительно, для того чтобы найти какое-нибудь значение $t' \in f_T(t)$, нужно взять некоторое $s \in \varphi^{-1}(t)$, т.е. закодировать исходные данные или представить их начальным состоянием s системы S и запустить соответствующее вычислительное устройство. Если это устройство остановится в заключительном состоянии s', то, применив к этому состоянию кодирующую функцию ψ, получим одно из значений $t' = \psi(s) \in f_T(t)$. Если реализация полна, а $f_T(t) \neq \phi$, то, перебирая всевозможные начальные состояния $s \in \varphi^{-1}(t)$ и всевозможные процессы вычислений, можно найти все элементы множества $f_T(t)$. Если их бесконечно много, то перечислимость множества значений функции переходов системы S и множества $\varphi^{-1}(t)$ гарантирует перечислимость множества $f_T(t)$. В случае физического вычислителя полнота реализации, детерминированность системы S и взаимная однозначность отображения φ гарантируют возможность получения (однозначность) результата, если он существует.

Связь между отображениями f_S и f_T в случае реализации, согласованной с настройкой, может быть представлена в виде диаграммы

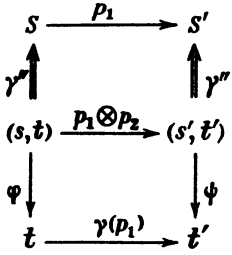


Полнота реализации влечет коммутативность этой диаграммы, так как из теоремы 8.1 следует в этом случае, что $f_S \circ \psi = \varphi \circ f_T$.

Теорема 8.2. *Пусть S есть реализация настроенной системы T с реализующей функцией γ. Тогда существует реализация S', согласован-*

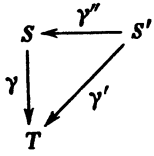
ная с настройкой и реализующей функцией $\gamma' = \gamma'' \circ \gamma$, где γ'' – гомоморфизм S' в S. Если реализация S полна, то S' также полна.

Положим $S' = S \times T$. Определим функцию переходов δ' следующим образом: $\delta'((p_1 \otimes p_2)(s, t)) = \{(s', t') \mid s \underset{S}{\to} s', \gamma(p_1 s s') = qt'$ и qt' есть процесс вычислений$\}$. Положим $S'_0 = \{(s, t) \mid \gamma(s) = tq, t \in T_0\}$, $S'_1 = \{(s, t) \mid t \in T_1\}$ и возьмем пару (S'_0, S'_1) в качестве настройки системы S'. В качестве кодирующих функций возьмем ограничения φ и ψ на множества S'_0 и S'_1 функции $\xi(s, t) = t$. Наконец, положим $\gamma''(s, t) = s$, $\gamma' = \gamma'' \circ \gamma$. Тот факт что γ'' есть гомоморфизм, а γ' – реализующее отображение, вытекает из определения и монотонности γ. Тот факт что реализация S' согласована с настройкой, виден из следующей диаграммы ($p_1 \otimes p_2$ – терминальный процесс системы S'):



Теорема доказана.

Теорему 8.2 можно представить в виде следующей коммутативной диаграммы:



В этой диаграмме γ – произвольная реализующая функция, γ'' – гомоморфизм, а γ' – реализующая функция, согласованная с настройкой.

С каждой функцией f: $X \to 2^Y$ можно связать настроенную систему $T = X \cup Y$, взяв в качестве функции переходов $f(t \underset{T}{\to} t' \Leftrightarrow t \overset{f}{\to} t')$, а в качестве настройки пару (X, Y). Если S есть реализация системы T, согласованная с настройкой, то S (частично) вычисляет функцию f. Эта функция, однозначно определяемая системой T, так же как и сама система T, называется *функциональной моделью* настроенной системы S. Основная задача проектирования дискретных устройств преобразования информации состоит в построении системы S, удовлетворяющей заданным условиям, по ее функциональной модели f. В некоторых случаях эта модель задается с помощью некоторой другой системы S', для которой f является также функциональной моделью. Тогда задача проектирования может быть сведена к построению реализации системы S', играющей роль промежуточной реализации.

Кодирующие функции для функциональных моделей могут быть самыми разнообразными, однако они должны быть достаточно простыми. Простейшим вариантом кодирующих функций, наиболее часто встречающимся на практике, является проектирование состояния многокомпонентной системы на одну из ее компонент. В этом случае мы имеем дело с дискретным преобразователем.

Многокомпонентная настроенная система S называется *дискретным преобразователем* над B, если среди ее компонент выделена компонента B, называемая *информационной*. Множество состояний информационной компоненты B называется также *информационной средой* или *информационным множеством*. Всякий дискретный преобразователь над B структурно изоморфен двухкомпонентной дискретной системе $S \subset A \times B$. Компонента A называется *управляющей*. В дальнейшем дискретный преобразователь над B будем называть так же, как и его управляющую компоненту. Стандартная функциональная модель дискретного преобразователя A над B с настройкой (S_0, S_1) – это функция $f_A = B_0 \to 2^{B_1}$, определяемая кодирующими функциями φ и ψ, которые проектируют состояние (a, b) дискретного преобразователя на компоненту B: $\varphi(a, b) = b$, $(a, b) \in S_0$, $\psi(a, b) = b$, $b \in S_1$, $B_0 = \{ b \mid$ существует $a \in A$ такое, что $(a, b) \in S_0\}$, $B_1 = \{ b \mid$ существует $a \in A$ такое, что $(a, b) \in S_1\}$.

Если A – автоматный детерминированный дискретный преобразователь над B с настройкой (S_0, S_1) такой, что $S_0 = \{ a_0\} \times B$, $S_1 = \{ a_1\} \times B$, то $f_A = B \to B$ есть частичное преобразование множества B. Преобразование f_A называют также *оператором* над B. Этот вариант является основным в теории дискретных преобразователей, и большинство моделей реальных вычислительных систем тем или иным способом сводятся к такому варианту. Дискретный преобразователь в этом случае можно представить в виде композиции двух автоматов. Автоматы, составляющие композицию, могут быть выбраны так, что входной алфавит каждого из них совпадает с выходным алфавитом другого, а множество внутренних состояний одного из них совпадает с множеством состояний информационной компоненты B. Заметим, что такое представление неоднозначно и зависит от выбора входных – выходных алфавитов.

Теорема 8.3. *Пусть настроенная система S имеет функциональную модель $f \subset B_1 \to 2^{B_2}$, определенную кодирующими функциями φ и ψ и реализующим отображением γ. Тогда S является гомоморфной моделью дискретного преобразователя A над $B = B_1 \cup B_2$, имеющего ту же самую функциональную модель с реализующим отображением $\gamma' = \gamma'' \circ \gamma$, где γ'' – гомоморфизм преобразователя A в S. Если S – полная реализация f, то A также определяет полную реализацию f.*

В качестве управляющей компоненты A возьмем множество состояний системы S. Распространим каким-либо образом функцию ψ на все множество B. Функцию переходов δ определим таким образом: $\delta(p_1 \otimes p_2) = \{(s', \psi(s')) \mid p_1 \to s'\}$. Настройка (S_0', S_1'), где $S_0' = \{(s, \varphi(s)) \mid s \in S_0\}$, $S_1' = \{(s, \psi(s)) \mid s \in S_1\}$. Дальнейшее ясно.

§ 9. Алгебра отношений

Функциональные модели настроенных дискретных систем и дискретных преобразований являются многозначными функциями или соответствующими им бинарными отношениями. Настроенная система, таким образом, может рассматриваться как один из способов задания бинарных отношений. В этом параграфе будет рассмотрен другой способ задания отношений. Он состоит в том, что отношения рассматриваются как элементы некоторой алгебры. Выражения в этой алгебре дают способ построения заданного отношения, исходя из некоторых исходных, уже заданных отношений с помощью операций алгебры отношений. Затем будет установлена связь алгебры отношений с дискретными преобразователями.

Зафиксируем некоторое множество S и будем рассматривать всевозможные отношения $f \subset S \times S$ на этом множестве. Определим три основные операции на множестве 2^{S^2} всех таких отношений.

1. Умножение отношений $f \circ f' = g$. Как известно, $s \xrightarrow{g} s' \Leftrightarrow$существует s'' такое, что $s \xrightarrow{f} s'' \xrightarrow{f'} s'$.

2. Объединение отношений $f \cup f' = g$. Это обычное теоретико-множественное объединение: $s \xrightarrow{g} s' \Leftrightarrow s \xrightarrow{f} s'$ или $s \xrightarrow{f'} s'$.

3. Итерация отношения $f^* = \bigcup_{n=0}^{\infty} f^n$, где $f^0 = \epsilon$, $f^{n+1} = f \circ f^n$, ϵ – тождественное отношение: $s \xrightarrow{\epsilon} s' \Leftrightarrow s = s'$. Таким образом, $s \xrightarrow{f^*} s' \Leftrightarrow s = s'$ или существуют $s_1, \ldots, s_n$ такие, что $s = s_1 \xrightarrow{f} s_2 \xrightarrow{f} \ldots \xrightarrow{f} s_n = s'$.

Множество отношений, замкнутое относительно операций умножения, объединения и итерации, а также содержащее тождественное отношение ϵ и пустое отношение ϕ, называется *алгеброй отношений*. Всякое множество отношений $Y \subset 2^{S^2}$ порождает некоторую алгебру отношений $\mathcal{H}(Y)$. Это наименьшее множество отношений, содержащее Y, ϵ и ϕ и замкнутое относительно операций алгебры отношений.

Если ввести символы для обозначения элементов множества Y, то из этих символов можно строить выражения с помощью операций алгебры отношений. Такие выражения будем называть *регулярными*. Каждое регулярное выражение представляет собой некоторое отношение. Это отношение будем называть регулярным относительно Y. Очевидно, что алгебра $\mathcal{H}(Y)$ состоит из всех отношений, регулярных относительно Y. Поскольку одно и то же отношение, регулярное относительно Y, может быть представлено различными регулярными выражениями, в алгебре $\mathcal{H}(Y)$ возникают соотношения, т.е. равенства регулярных выражений, которые превращаются в равенства отношений, если вместо символов Y подставить их значения. Равенство регулярных выражений называется *тождеством алгебры отношений*, если оно истинно при любых подстановках отношений вместо символов Y. Между алгебрами языков и алгебрами отношений существует очевидная связь. В частности, понятия регулярного выражения для этих двух алгебр совпадают. Для того чтобы изучить связь между алгеброй отношений и алгеброй языков более точно, будем различать регуляр-

ное выражение R, язык $\bar{R}$ в алфавите Y и отношение $\bar{\bar{R}}$, которое получается, если вместо символов алфавита Y подставить соответствующие этим символам отношения.

Теорема 9.1. *Для любых двух регулярных выражений P и Q из $\bar{P} = \bar{Q}$ следует $\bar{\bar{P}} = \bar{\bar{Q}}$.*

Лемма 9.1. $s \overset{\bar{\bar{P}}}{\to} s' \Leftrightarrow$ *существует слово* $p \in \bar{P}$ *такое, что* $s \overset{\bar{\bar{p}}}{\to} s'$.

Доказательсто проводится индукцией по числу операций в выражении P. Если операций нет, то $P = y$, $y \in Y$ и утверждение очевидно. Если $P = P_1P_2$, то $s \overset{\bar{\bar{P}}}{\to} s' \Leftrightarrow$ существует s'' такое, что $s \overset{\bar{P}_1}{\to} s'' \overset{\bar{\bar{P}}_2}{\to} s' \Leftrightarrow s \overset{\bar{p}_1}{\to} s'' \overset{\bar{\bar{p}}_2}{\to} s'$, где $p_1 \in \bar{P}_1$, $p_2 \in \bar{P}_2$ по предположению индукции $\Leftrightarrow s \overset{\bar{\bar{p}}_1\bar{\bar{p}}_2}{\longrightarrow} s'$, $\Leftrightarrow s \overset{\overline{\overline{p_1p_2}}}{\longrightarrow} s'$, и так как $p_1p_2 \in \bar{P}_1\bar{P}_2 = \overline{P_1P_2}$, то для этого случая утверждение доказано. Случаи $P = P_1 \cup P_2$ и $P = P_1^*$ рассматриваются аналогично. Лемма доказана.

Предположим теперь, что $\bar{P} = \bar{Q}$. Пусть $s \overset{\bar{\bar{P}}}{\to} s'$, тогда по лемме 9.1 существует $p \in \bar{P}$ такое, что $s \overset{\bar{\bar{p}}}{\to} s'$, но $p \in \bar{Q}$, откуда следует $s \overset{\bar{\bar{Q}}}{\to} s'$. Обратное утверждение доказывается аналогично. Теорема доказана.

Теорема 9.2. *Если P и Q – регулярные выражения над Y, то $P = Q$ есть тождество алгебры отношений* $\Leftrightarrow \bar{P} = \bar{Q}$.

Пусть $P = Q$ – тождество. Тогда $\bar{\bar{P}} = \bar{\bar{Q}}$ при любой подстановке $y \to \bar{\bar{y}}$ $(y \in Y)$. Положим $S = Y^*$ и для каждого $y \in Y$ определим $\bar{\bar{y}}$ так, что $p \to \bar{\bar{y}} \to q \Leftrightarrow q = py$. Из этого определения следует $p \overset{\bar{\bar{q}}}{\to} r \Leftrightarrow r = pq$. Покажем, что $\bar{P} = \bar{Q}$. Действительно, $p \in \bar{P} \Rightarrow e \overset{\bar{\bar{p}}}{\to} p \Rightarrow e \to p \Rightarrow e \overset{\bar{\bar{Q}}}{\to} p \Rightarrow e \overset{\bar{\bar{q}}}{\to} p \Rightarrow q = p \Rightarrow p \in \bar{Q}$. Обратное утверждение теоремы следует из теоремы 9.1. Теорема доказана.

Поскольку всякое соотношение алгебры языков является тождеством, то из теоремы 9.2 следует, что алгебры языков и алгебры отношений обладают одной и той же совокупностью тождеств. Естественно ожидать, что для алгебры отношений можно построить теорию уравнений, аналогичную теории уравнений в алгебре языков. Это действительно так, но для обоснования удобно использовать общую теорему о неподвижной точке. Эта теорема будет неоднократно использоваться в дальнейшем, и мы приведем уже здесь необходимые определения и доказательства.

Частично упорядоченное множество D называется *индуктивным*, если всякая возрастающая цепочка его элементов $d_1 \leqslant d_2 \leqslant \ldots \leqslant d_n \leqslant \ldots$ имеет наименьшую верхнюю грань, т.е. существует такой элемент d, что $d_n \leqslant d$ $(n = 1, 2, \ldots)$, и если $d_n \leqslant d'$ $(n = 1, 2, \ldots)$, то $d \leqslant d'$. Указанный элемент обозначают $d = \bigcup_{n=1}^{\infty} d_n$. Функция $\varphi\colon D \to D$ называется *монотонной*, если из $d \leqslant d'$ следует $\varphi(d) \leqslant \varphi(d')$. Функция φ непрерывна, если для любой возрастающей цепочки элементов $d_1 \leqslant d_2 \leqslant \ldots$ $\varphi(\bigcup_{n=1}^{\infty} d_n) = \bigcup_{n=1}^{\infty} \varphi(d_n)$. Из непрерывности следует монотонность. Действительно , $d \leqslant d' \Leftrightarrow d' = d \cup d' \Rightarrow \varphi(d') = \varphi(d) \cup \varphi(d') \Rightarrow \varphi(d) \leqslant \varphi(d')$. Наименьший элемент

множества D, т.е. такой элемент ϕ, что для всех $d \in D$ $\phi \leqslant D$, называется *нулем* множества D. Элемент d называется *неподвижной точкой* функции φ: $D \to D$, если $\varphi(d) = d$.

Определим k-ю итерацию $\varphi^{(k)}$ функции φ, полагая $\varphi^{(0)}(x) = x$, $\varphi^{(k+1)}(x) = \varphi(\varphi^{(k)}(x))$.

Т е о р е м а 9.3 (теорема о неподвижной точке). *Если D есть индуктивное частично-упорядоченное множество с нулем, а φ: $D \to D$ – непрерывная функция, то $\bigcup_{k=0}^{\infty} \varphi^{(k)}(\phi)$ есть наименьшая неподвижная точка этой функции.*

Действительно,

$$\varphi\Big(\bigcup_{k=0}^{\infty} \varphi^{(k)}(\phi)\Big) = \bigcup_{k=0}^{\infty} \varphi^{(k+1)}(\phi) = \bigcup_{k=1}^{\infty} \varphi^{(k)}(\phi) = \bigcup_{k=0}^{\infty} \varphi^{(k)}(\phi),$$

т.е. элемент $d^{\infty} = \bigcup_{k=0}^{\infty} \varphi^{(k)}(\phi)$ есть неподвижная точка функции φ. Пусть теперь d – неподвижная точка. Из монотонности φ следует

$$\phi \leqslant d \Rightarrow \varphi(\phi) \leqslant \varphi(d) = d,$$

$$\varphi(\phi) \leqslant d \Rightarrow \varphi^2(\phi) \leqslant \varphi(d) = d,$$

. . . . . .

$$\varphi^{(k)}(\phi) \leqslant d \Rightarrow \varphi^{(k+1)}(\phi) \leqslant \varphi(d) = d,$$

откуда по определению наименьшей верхней грани $d^{\infty} \leqslant d$, т.е. d^{∞} есть наименьшая неподвижная точка.

Если D – индуктивное частично-упорядоченное множество с нулем, то множество D^n с порядком, определенным покомпонентно: $(d_1, \ldots, d_n) \leqslant (d'_1, \ldots, d'_n) \Leftrightarrow d_1 \leqslant d'_1, \ldots, d_n \leqslant d'_n$, также является индуктивным и имеет нуль $(\phi, \ldots, \phi)$. Пусть функции φ_i: $D^n \to D$ $(i = 1, 2, \ldots, n)$ непрерывны по каждому из своих аргументов. Тогда функция φ: $D^n \to D^n$, определенная равенством $\varphi(x_1, \ldots, x_n) = (\varphi_1(x_1, \ldots, x_n), \ldots, \varphi_n(x_1, \ldots, x_n))$, также непрерывна. Для доказательства нужно воспользоваться равенством

$$\varphi_i\Big(\bigcup_{m=1}^{\infty} x_1^{(m)}, \ldots, \bigcup_{m=1}^{\infty} x_n^{(m)}\Big) = \bigcup_{m=1}^{\infty} \varphi_i(x_1^{(m)}, \ldots, x_n^{(m)})$$

для возрастающих цепочек $x_i^{(1)} \leqslant x_i^{(2)} \leqslant \ldots$ $(i = 1, \ldots, n)$, которое справедливо, поскольку для каждого набора $m_1, \ldots, m_n$ имеет место $\varphi_i(x_1^{(m_1)}, \ldots, x_n^{(m_n)}) \leqslant \varphi_i(x_1^{(m)}, \ldots, x_n^{(m)})$, где $m = \max(m_1, \ldots, m_n)$. Определим k-ю совместную итерацию функций φ_i по формулам

$$\varphi_i^{(0)}(x_1 \ldots, x_n) = x_i, \tag{9.1}$$

$$\varphi_i^{(k+1)}(x_1, \ldots, x_n) = \varphi_i(\varphi_1^{(k)}(x_1, \ldots, x_n), \ldots, \varphi_n^{(k)}(x_1, \ldots, x_n)).$$

Пусть

$$\varphi_i^{(\infty)}(x_1, \ldots, x_n) = \bigcup_{m=0}^{\infty} \varphi_i^{(m)}(x_1, \ldots, x_n). \tag{9.2}$$

Применяя теорему 9.3 к множеству D^n, получим следующее утверждение.

Теорема 9.4. *Если D – индуктивное частично-упорядоченное множество с нулем, а φ_i: $D^n \to D$ $(i = 1, \ldots, n)$ – функции, непрерывные по каждому из аргументов, то элемент* $d^\infty = (\varphi_1^\infty(\phi, \ldots, \phi), \ldots, \varphi_n^\infty(\phi, \ldots, \phi))$ *есть наименьшая неподвижная точка функции* $\varphi(x_1, \ldots, x_n) = (\varphi_1(x_1, \ldots, x_n), \ldots, \varphi_n(x_1, \ldots, x_n))$.

Для доказательства достаточно заметить, что $\varphi^{(m)}(x_1, \ldots, x_n) = (\varphi_1^{(m)}(x_1, \ldots, x_n), \ldots, \varphi_n^{(m)}(x_1, \ldots, x_n))$.

Перейдем теперь к изучению уравнений в алгебре отношений. Нас будут интересовать канонические системы уравнений, т.е. системы вида

$$x_i = \varphi_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n, \tag{9.3}$$

где $\varphi_i(x_1, \ldots, x_n)$ – регулярные выражения относительно неизвестных отношений $x_1, \ldots, x_n$ и, возможно, некоторых заданных отношений.

Множество 2^{S^2} всех отношений на множестве S частично упорядочено отношением $f \leqslant f'$, которое означает обычное теоретико-множественное включение, т.е. $f \leqslant f' \Leftrightarrow x \overset{f}{\to} y \Rightarrow x \overset{f'}{\to} y$. Наименьшей верхней гранью возрастающей цепочки отношений является их теоретико-множественное объединение, а нулем – пустое отношение ϕ.

Функции $f \circ f'$, $f \cup f'$ и f^* непрерывны по каждому из аргументов. Например, непрерывность умножения по первому аргументу следует из правой дистрибутивности умножения относительно объединения:

$$(\bigcup_{m=1}^{\infty} f_m) \circ f' = \bigcup_{m=1}^{\infty} (f_m \circ f').$$

Действительно, если обозначить левую часть равенства через g, правую через g', а $\bigcup_{m=1}^{\infty} f_m$ через h, то имеем $x \overset{g}{\to} y \Leftrightarrow x \overset{h}{\to} z \overset{f'}{\to} y \Leftrightarrow x \overset{f_m}{\to} z \overset{f'}{\to} y \Leftrightarrow x \overset{g'}{\to} y$.

Аналогично доказываются непрерывность умножения относительно второго аргумента (левая дистрибутивность) и непрерывность объединения. Непрерывность итерации требует установления равенства

$$(\bigcup_{m=1}^{\infty} f_m)^* = \bigcup_{m=1}^{\infty} f_m^*$$

для возрастающей цепочки отношений $f_1 \leqslant f_2 \leqslant \ldots$.

Обозначая левую часть через g, а правую – через g', получим $x \overset{g}{\to} y \Rightarrow x = x_1 \overset{f_{m_1}}{\to} x_2 \overset{f_{m_2}}{\to} \ldots \overset{f_{m_k}}{\to} x_k = y \Rightarrow x = x_1 \overset{f_m}{\to} x_2 \overset{f_m}{\to} \ldots \overset{f_m}{\to} x_k = y$, где $m = \max(m_1, \ldots, m_k)$ (монотонность) $\Rightarrow x \overset{f_m}{\to} y \Rightarrow x \overset{f_m^*}{\to} y \Rightarrow x \overset{g'}{\to} y \Rightarrow x \overset{f_m^*}{\to} y \Rightarrow x \overset{f_m^k}{\to} y \Rightarrow x \overset{g}{\to} y$.

Поскольку суперпозиция непрерывных функций непрерывна, то к системе уравнений (9.3) можно применить теорему 9.4, откуда получаем следующую теорему.

Теорема 9.5. *Система уравнений* (9.3) *имеет наименьшее решение* $(\varphi_1^\infty, \ldots, \varphi_n^\infty)$, *которое определяется формулами* (9.1) *и* (9.2).

Особый интерес так же, как и для алгебры языков, представляют канонические системы линейных уравнений, т.е. линейных уравнений вида

$$x_i = a_{i1} x_1 \cup a_{i2} x_2 \cup \ldots \cup a_{in} x_n \cup a_{i0}, \quad i = 1, \ldots, n,$$

(праволинейные уравнения) или вида

$$x_i = x_1 a_{i1} \cup x_2 a_{i2} \cup \ldots \cup x_n a_{in} \cup a_{i0}, \quad i = 1, \ldots, n,$$

(леволинейные уравнения), где $a_{i1}, a_{i2}, \ldots, a_{i0}$ — известные, $x_1, x_2, \ldots, x_n$ — неизвестные отношения. Рассматривая одно уравнение с одним неизвестным

$$x = ax \cup b$$

или

$$x = xa \cup b,$$

в качестве следствия из теоремы 9.5 получаем, что наименьшее решение первого из этих уравнений может быть определено как $\varphi^{\infty}(\phi)$, где

$$\varphi(x) = ax \cup b, \quad \varphi^{(m)}(x) = a^m x \cup a^{m-1} b \cup \ldots \cup b = a^m x \cup \bigcup_{k=0}^{m-1} a^k b,$$

откуда

$$x = \varphi^{\infty}(\phi) = \bigcup_{m=0}^{\infty} a^m b = \left(\bigcup_{m=0}^{\infty} a^m \right) b = a^* b. \tag{9.4}$$

Аналогично для второго уравнения получаем

$$x = ba^*. \tag{9.5}$$

Формулы (9.4) и (9.5) позволяют получить в явном виде решение канонической системы линейных уравнений в алгебре отношений аналогично тому, как это получалось для уравнений в алгебре языков. Это решение, очевидно, будет представлено в виде регулярного выражения, построенного из коэффициентов уравнений. Окончательно получаем следующий результат.

Т е о р е м а 9.6. *Наименьшее решение канонической системы уравнений в алгебре отношений регулярно относительно коэффициентов.*

Поскольку всякое отношение на B определяет некоторое многозначное преобразование множества B, а операции над отношениями естественным образом переносятся на преобразования, то каждую алгебру отношений можно рассматривать также как алгебру многозначных преобразований или операторов, действующих на B. Строго говоря, это — различные алгебры, и следует говорить об их изоморфизме, в силу которого все доказанные выше теоремы имеют место также и для алгебры многозначных преобразований. Так же как и раньше, мы не будем проводить различия между отношениями и многозначными функциями (преобразованиями), употребляя терминологию, которая больше подходит по контексту.

Применим полученные результаты к анализу дискретных преобразователей. Пусть A — дискретный преобразователь над B. Предположим, что A — автоматный, а настройка имеет вид $(A_0 \times B, A_1 \times B)$. Настройку такого типа будем называть *стандартной*. Функция переходов преобразователя A $(a, b) \to (a', b')$ однозначно определяется функциями переходов компонент: $a' \in \delta_1(a, b)$. $b' \in \delta_2(a, b, a')$. Для каждой пары $(a, a') \in A$ рассмотрим многозначную функцию $y_{aa'} \subset B^2$, полагая $b \xrightarrow{y_{aa'}} b' \iff b' \in \delta(a, b, a') \iff (a, b) \to (a', b')$. Функцию $y_{aa'}$ назовем *элементарным*

преобразованием дискретного преобразователя A, выполняемым при переходе из состояния a в a'.

Т е о р е м а 9.7. *Преобразование f_A, вычисляемое конечным автоматным дискретным преобразователем A над B со стандартной настройкой, регулярно относительно множества всех его элементарных преобразований.*

Не ограничивая общности, можно считать, что все состояния множества $A \times B$ являются допустимыми начальными состояниями, поскольку такое расширение хотя и увеличивает множество допустимых процессов, но не меняет преобразования f_A. Для каждого $a \in A$ определим преобразование $f_A \subset B^2$, полагая $b \xrightarrow{f_a} b' \iff$ существует допустимый процесс p такой, что $(a, b) \xrightarrow{p} (a', b')$, где $a' \in A_1$. Преобразования f_a удовлетворяют системе уравнений

$$f_a = \bigcup_{a' \in A} y_{aa'} f_{a'}, \quad a \in A \setminus A_1,$$
$$f_a = \epsilon, \quad a \in A_1. \tag{9.6}$$

Действительно, $b \xrightarrow{f_a} b' \iff (a, b) \xrightarrow{p} (a_1, b_1)$, $a_1 \in A_1$, p – терминальный процесс вычислений $\iff b \xrightarrow{y_{aa'}} b_1$ или $b \xrightarrow{y_{aa'}} b' \xrightarrow{f_{a'}} b_1 \iff b \xrightarrow{y_{aa'} f_{a'}} b_1$ (в первом случае $a' = a_1$, $f_{a'} = \epsilon$). Покажем теперь, что семейство $(f_a)_{a \in A}$ образует наименьшее решение системы (9.6). Действительно, пусть $(f_{a'})_{a \in A}$ – какое-нибудь решение. Покажем, что $f_a \subset f'_a$. Обозначим через $r(a, b, b')$ наименьшую длительность процесса p такого, что $(a, b) \xrightarrow{p} (a', b')$, где $a' \in A_1$. Индукцией по $n = r(a, b, b')$ доказываем, что $b \xrightarrow{f_a} b' \Rightarrow b \xrightarrow{f'_a} b'$. Если $n = 0$, то $a \in A_1$, откуда $f'_a = f_a = \epsilon$. При $r(a, b, b') = n + 1$, где $n > 0$, имеем $b \xrightarrow{f_a} b' \Rightarrow \xrightarrow{y_{aa_1}} b_1 \xrightarrow{f_{a_1}} b'$, и, следовательно, по

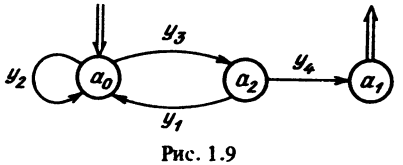


Рис. 1.9

предположению индукции $b \xrightarrow{y_{aa_1}} b_1 \xrightarrow{f'_{a_1}} b' \Rightarrow b \xrightarrow{f'_a} b'$. Осталось заметить, что $f_A = \bigcup_{a \in A_0} f_a$.

Рассмотрим пример. Пусть управляющая компонента дискретного преобразователя A имеет диаграмму переходов, изображенную на рис. 1.9. На стрелках показаны элементарные преобразования, соответствующие переходам, $A_0 = \{a_0\}$, $A_1 = \{a_1\}$. Тогда система уравнений для $f_{a_0} = f_0$

и $f_{a_1} = f_1$ имеет вид

$$f_0 = y_1 f_2 \cup y_2 f_0,$$

$$f_2 = y_3 f_0 \cup y_4.$$

Решая эту систему относительно f_0, получим

$$f_A = f_0 = y_1(y_3 f_0 \cup y_4) \cup y_2 f_0 = (y_1 y_3 \cup y_2) f_0 \cup y_1 y_4 =$$

$$= (y_1 y_3 \cup y_2)^* y_1 y_4.$$

Комментарии к главе 1

Для классических динамических систем множество допустимых процессов задается с помощью систем обыкновенных дифференциальных уравнений, а современные обобщения определяют их однопараметрическими группами диффеоморфизмов конечномерных дифференцируемых многообразий [3]. Понятие управляющей системы, предложенное С.В. Яблонским [79], делает упор на структурном аспекте. Большое влияние на формирование основных понятий кибернетики и информатики оказала теория автоматов, начало которой было положено сборником [1], куда, в частности, вошли основополагающие работы С.К. Клини и Э.Ф. Мура. В этой книге мы следуем традициям алгебраического подхода в теории автоматов, развитого В.М. Глушковым [8], который также много сделал для превращения теории автоматов в прикладную инженерную науку [15].

В общей теории систем [43] используются непрерывные аналоги понятия автомата. Общее понятие дискретной автоматной системы в теории проектирования было предложено З. Павлаком [91] и использовалось в многочисленных работах польских математиков под названием "машина Павлака".

Сети Петри [58] стали в последние годы популярной моделью исследования параллельных процессов. Более глубокие знания по современной теории автоматов можно получить из монографий [29, 59, 78].

Понятие реализации в теоретико-категорных терминах исследовалось Гогеном [86]. Прикладные аспекты реализации дискретных систем рассматриваются в [48].

Понятие дискретного преобразователя как базовой модели ЭВМ содержится в работе В.М. Глушкова [16]. Математические вопросы теории дискретных преобразователей рассматриваются в [28, 85]. Дискретные преобразователи используются в качестве основной модели в автоматизации проектирования ЭВМ [19].

Глава 2

АЛГОРИТМЫ

§ 1. Схемы программ

Последовательная программа, выполняемая на вычислительной машине, может рассматриваться как управляющая компонента дискретного преобразователя, информационная среда которого представляет собой обрабатываемые данные, включая состояния регистров процессора и внешних устройств – внешней памяти, устройств ввода-вывода и т.п. Состояние программы в текущий момент времени определяется выполняемой командой. Специфика программы как управляющей компоненты дискретного преобразователя состоит в том, что выполняемые ею элементарные преобразования расщепляются на две компоненты – проверку условия и преобразование состояния информационной среды. В результате про-

верки условия происходит выбор оператора, действующего на состояние информационной компоненты, и определяется следующее состояние программы. Указанная специфика сохраняется также в процедурных языках программирования высокого уровня, которые можно рассматривать как языки описания моделей реальных программ.

Понятие схемы программы, изучаемое в этом параграфе, представляет собой наиболее распространенную математическую модель программы, которая получается, если отвлечься от конкретной природы информационной среды, над которой она работает. Большая общность понятия схемы программы позволяет использовать его для изучения и представления алгоритмов, реализуемых не только программными, но и микропрограммными, а также аппаратными средствами вычислительных систем. С помощью схем программ можно выражать алгоритмы, реализуемые и другими кибернетическими системами, отличными от ЭВМ, например, биологическими, экономическими и т.п.

Пусть U и Y – множества. Первое будем называть *множеством базовых условий,* второе – *множеством базовых операторов.* Рассмотрим множество $\hat{U}$ пропозициональных булевых функций от базовых условий, т.е. будем рассматривать элементы множества U как переменные, принимающие значения 0, 1, и строить из них функции, также принимающие значения 0, 1. Функции такого рода будем называть *элементарными условиями.*

Схемой программы над базисом (U, Y) или $U - Y$-*схемой* программы называется множество A состояний схемы вместе с множеством $T \subset A \times \hat{U} \times Y^* \times A$ переходов и двумя выделенными состояниями: начальным $a^{(0)}$ и заключительным $a^{(1)}$. Таким образом, формально $U - Y$-схема программы – это шестерка $(A, U, Y, T, a^{(0)}, a^{(1)})$. Слова в алфавите Y, т.е. последовательности базовых операторов, будем называть *элементарными операторами.* Каждый переход из множества T представляет собой четверку (a, u, q, a'), где u – элементарное условие, q – элементарный оператор. Эту четверку так же, как и высказывание о ее принадлежности множеству T, будем записывать в виде $a \xrightarrow{u/q} a'$. Если $u = 1$, то вместо $a \xrightarrow{u/q} a'$ будем писать $a \xrightarrow{q} a'$, если же $q = e$ (пустое слово), то вместо $a \xrightarrow{u/q} a'$ будем писать $a \xrightarrow{u} a'$. Наконец, $a \to a'$ обозначает переход, в котором $u = 1$, $q = e$. Высказывание $a \xrightarrow{u/q} a'$ читается следующим образом: при истинном значении условия u схема A может перейти из состояния a в состояние a', выполнив при этом оператор q. Употребление глагола "может" подчеркивает, что допускается рассмотрение программ с недетерминированным поведением.

Переходы $a_1 \xrightarrow{u_1/q_1} a_2$ и $a_2 \xrightarrow{u_2/q_2} a_3$ называются *сопряженными.* Последовательность переходов, в которой любые два соседних перехода сопряжены, записывается в виде

$$a_1 \xrightarrow{u_1/q_1} a_2 \xrightarrow{u_2/q_2} \ldots \xrightarrow{u_{m-1}/q_{m-1}} a_m \qquad (1.1)$$

и называется *путем* в схеме A. Этот путь начинается в состоянии a_1 и окан-

чивается в состоянии a_m. Путь (1.1) допустим, если $a_2, \ldots, a_{m-1} \neq a^{(1)}$, $u_1, \ldots, u_{m-1} \neq 0$, и из того, что $q_k = q_{k+1} = \ldots = q_l = e$ $(1 \leqslant k \leqslant l \leqslant m-1)$, следует $u_k \wedge \ldots \wedge u_l \neq 0$. Допустимый путь (1.1) называется *инициальным*, если $a_1 = a^{(0)}$, и *терминальным*, если, кроме того, $a_m = a^{(1)}$.

Допустимые пути описывают возможные способы функционирования схемы A. Их удобно также представлять как процессы функционирования дискретной динамической системы $\hat{A} = A \times \hat{U} \times Y^*$, ассоциированной со схемой A. Процесс $(a_1, u_1, q_1) \ldots (a_m, u_m, q_m)$ называется *допустимым*, если $a_1 \xrightarrow{u_1/q_1} \ldots \xrightarrow{u_{m-1}/q_{m-1}} a_m$ есть допустимый путь схемы A и либо этот путь имеет допустимое продолжение $a_m \xrightarrow{u_m/a_m} a_{m+1}$, либо $a_m = a^{(1)}$, $u_m = 1$, $q_m = e$. Система $\hat{A}$ рассматривается как настроенная с настройкой $(\hat{A}_0, \hat{A}_1)$, где $\hat{A}_0 = \{(a^{(0)}, u, q) \mid a^{(0)} \xrightarrow{u/q} a', a' \in A\}$, $\hat{A}_1 = \{(a^{(1)}, 1, e)\}$. Таким образом, между допустимыми инициальными путями схемы A и процессами вычислений системы $\hat{A}$ существует взаимно однозначное соответствие.

Переход $a \xrightarrow{u/q} a'$ назовем *существенным*, если $u \neq 0$. Если все переходы схемы A существенны, то множество допустимых процессов системы $\hat{A}$ однозначно определяет множество T переходов схемы A, и $\hat{A}$ можно рассматривать вместо схемы A. Само множество A также можно рассматривать как дискретную систему, допустимыми процессами которой являются проекции допустимых процессов системы $\hat{A}$ на компоненту A ($a_1 \ldots a_m$ допустим, если существует допустимый процесс $(a_1, u_1, q_1) \ldots (a_m, u_m, q_m)$), а настройка есть пара $(\{a^{(0)}\}, \{a^{(1)}\})$. Система A является моделью схемы программы A, которую будем называть *упрощенной моделью* схемы A. Граф, представляющий упрощенную модель схемы, называется также *управляющим графом* этой схемы.

Отображение γ: $A \to A'$ $U - Y$-схемы программы A в $U - Y$-схему программы A' называется *гомоморфизмом* ($U - Y$-схем программ), если образ начального (заключительного) состояния схемы A является начальным (заключительным) состоянием схемы A' и из $a \xrightarrow{u/q} a'$ в схеме A следует $\gamma(a) \xrightarrow{u/q} \gamma(a')$ в схеме A'. Взаимно однозначный гомоморфизм называется *изоморфизмом*, и две схемы *изоморфны*, если существует изоморфизм одной из них на другую.

Для того чтобы схему программы превратить в модель конкретной программы, необходимо интерпретировать базовые условия и операторы на некотором множестве B, представляющем информационную среду рассматриваемой программы. Такая интерпретация задается путем сопоставления каждому базовому условию α предиката, заданного на множестве B, т.е. отображения f_α: $B \to \{0, 1\}$, и каждому базовому оператору y преобразования множества B, т.е. отображения f_y: $B \to B$. Интерпретация базовых условий и операторов естественным образом переносится на

элементарные условия и операторы. Если $u = \varphi(\alpha_1, \ldots, \alpha_n)$ – элементарное условие, зависящее от базовых условий $\alpha_1, \ldots, \alpha_n$, то ему соответствует предикат f_u, определенный равенством $f_u(b) = \varphi(f_{\alpha_1}(b), \ldots, f_{\alpha_n}(b))$. Если $q = y_1 \ldots y_m$ – элементарный оператор, представленный в виде произведения базовых операторов, то ему соответствует преобразование $f_q = f_{y_1} \circ \ldots \circ f_{y_m}$, равное последовательной композиции преобразований, соответствующих базовым операторам. Пустое слово e (пустой оператор) всегда интерпретируется как тождественное преобразование: $f_e = \epsilon$, $\epsilon(b) = b$. Схема программы называется *интерпретированной*, если задана интерпретация ее базиса. Если схема интерпретирована, то базовые условия и операторы будем отождествлять с их интерпретацией и писать $u(b)$ вместо $f_u(b)$ и by вместо $f_y(b)$, рассматривая y как оператор, действующий справа. Для элементарных операторов имеем $b(q_1q_2) = (bq_1)q_2 = bq_1q_2$.

Каждой схеме программы A, интерпретированной на множестве B, естественным образом сопоставляется дискретный преобразователь $\hat{A}$ над B. Состояние $((a, u, q), b)$ преобразователя $\hat{A}$ будем считать допустимым, если $u(b) = 1$ и $a \xrightarrow{u/q} a'$ для некоторого $a' \in A$ или $a = a^{(1)}$, $u = 1$, $q = e$. Отношение переходов определяется следующим образом: $((a, u, q), b) \to$ $\to ((a', u', q'), b') \Longleftrightarrow b' = bq$, $a \xrightarrow{u/q} a'$. Настройка (S_0, S_1) определяется условиями $((a, u, q), b) \in S_0 \Longleftrightarrow a = a^{(0)}$, $S_1 = \{((a^{(1)}, 1, e), b)\}$. Проекция γ: $S \to \hat{A}$ множества $S \subset \hat{A} \times B$ допустимых состояний дискретного преобразователя $\hat{A}$ на управляющую компоненту $\hat{A}$, т.е. отображение, определенное равенством $\gamma((a, u, q), b) = (a, u, q)$, является гомоморфизмом. Действительно, если $((a_1, u_1, q_1), b_1) \ldots ((a_m, u_m, q_m), b_m)$ есть допустимый процесс, то существует путь $a_1 \xrightarrow{u_1/q_1} \ldots \xrightarrow{u_{m-1}/q_{m-1}} a_m$, причем $u_i(b_i) = 1$. Поэтому, если $q_k = \ldots = q_l = e$, то $b_k = \ldots = b_l$, откуда следует, что $u_k \wedge \ldots \wedge u_l \neq 0$. Все эти условия равны 1 на наборе значений базовых условий $\alpha_1, \ldots, \alpha_n$, равном $(\alpha_1(b_k), \ldots, \alpha_n(b_k))$. Следовательно, рассмотренный путь допустим, и процесс $(a_1, u_1, q_1) \ldots (a_m, u_m, q_m)$ является допустимым процессом системы $\hat{A}$. Таким образом, интерпретированная схема есть гомоморфная реализация системы $\hat{A}$. Следует обратить внимание на то, что эта реализация, вообще говоря, не является полной, поскольку могут существовать допустимые процессы системы $\hat{A}$, не имеющие реализации.

Схема программы A, интерпретированная на множестве B, вычисляет многозначную функцию f_A: $B \to 2^B$, которая определяется как стандартная функциональная модель $f_{\hat{A}}$ дискретного преобразователя $\hat{A}$ над B. Эта функция будет также обозначаться через $f_{A,B}$, если необходимо подчеркнуть, что схема A интерпретирована на B, и называться *оператором* или *преобразованием*, вычисляемым схемой A, интерпретированной на B.

Рассмотрим произвольное состояние a U – Y-схемы программы A, и пусть $a \xrightarrow{u_1/q_1} a_1, \ldots, a \xrightarrow{u_k/q_k} a_k$ – все переходы, которые ведут из состояния a. Схема программы называется *детерминированной*, если для любого состояния a имеет место $u_i \wedge u_j = 0$ при $i \neq j$. Схема называется *полно-*

стью определенной, если $u_1 \vee \ldots \vee u_k = 1$ для любого состояния, кроме, быть может; заключительного. Схема называется *базовой*, если для любого перехода $a \xrightarrow{u/q} a'$ этой схемы $q = e$ или q есть базовый оператор.

Если схема A детерминирована, то любая ее интерпретация также детерминирована и вычисляет однозначную частичную функцию $f_A \subset B \to B$. В этом случае структура дискретного преобразователя, представляющего интерпретированную схему A, может быть сделана более простой, поскольку в качестве управляющей компоненты можно вместо множества $\hat{A}$ взять само A. Действительно, состояние $((a, u, y), b)$ однозначно восстанавливается по паре (a, b). Компоненты u и y получаются из единственного перехода $a \xrightarrow{u/y} a'$ такого, что $u(b) = 1$. Если такого перехода нет, то $\delta(a, b) = \phi$. Таким образом, дискретный преобразователь $S \subset \hat{A} \times B$ структурно изоморфен преобразователю $A \times B$. Упрощенный дискретный преобразователь $A \times B$ имеет ту же самую функциональную модель f_A, что и преобразователь $S \subset \hat{A} \times B$, но в случае недетерминированной схемы его использовать нельзя, поскольку теряется часть информации о функционировании схемы A: не известно, какой из переходов срабатывает на каждом шаге. Поэтому $A \times B$ может не быть реализацией системы $\hat{A}$. В то же время $A \times B$, конечно же, является реализацией упрощенной модели схемы программы A.

Если схема программы A полностью определена, то всякий финальный процесс ее интерпретации либо терминален, либо бесконечен.

Всякая схема может быть сведена к базовой, если переход вида $a \xrightarrow{u/y_1 \cdots y_m} a'$ заменить последовательностью переходов $a \xrightarrow{u/y_1} b_1 \xrightarrow{y_2} \ldots \ldots \xrightarrow{y_{m-1}} b_{m-1} \xrightarrow{y_m} a'$, добавив новые состояния $b_1, \ldots, b_m$. Новая схема при любой интерпретации будет выполнять тот же самый оператор, что и старая. Кроме того, она является достаточно простой реализацией старой схемы.

Схемы программ удобно представлять графами с нагруженными дугами. Вершинами таких графов служат состояния схемы. Если $a \xrightarrow{u/q} a'$, то вершины a и a' соединяются дугой, и эта дуга нагружается парой u/q. На рис. 2.1 представлен пример схемы программы в графовом виде. Двойной стрелкой выделено начальное состояние. Заключительное состояние отмечено знаком *. Остальные состояния отмечены своими обозначениями. Здесь α и β – базовые условия, y и z – базовые операторы. Если в паре u/q условие $u = 1$, то вместо пары $1/q$ пишется сам оператор q. Если же $q = e$, то вместо пары u/e пишется только одно условие u.

В теории программирования, а также при документировании реальных программ обычно используется специальный случай схем программ – бинарные схемы. Схема программы называется *бинарной*, если для любого ее состояния a, кроме заключительного, имеет место одно из двух:

1) из состояния a возможен лишь один переход, который имеет вид $a \xrightarrow{y} a'$, где y – базовый оператор;

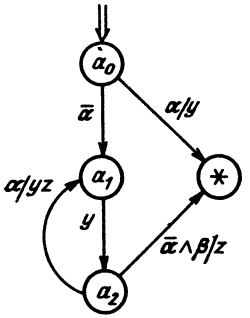


Рис. 2.1

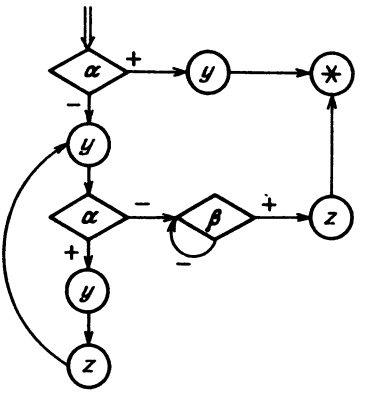


Рис. 2.2

2) из состояния a ведут ровно два перехода, которые имеют вид $a \overset{u}{\to} a'$ и $a \overset{\bar{u}}{\to} a'$, где u – элементарное условие.

Из заключительного состояния переходы невозможны.

Состояния, удовлетворяющие условию 1), называются *преобразователями*, а состояния типа 2) – *распознавателями*. Для бинарных схем программ используется более простое графовое представление. Это представление отличается от описанного выше тем, что разметки дуг переносятся на вершины. Вершины преобразователи отмечаются символами базовых операторов, а вершины распознаватели – символами (или выражениями) соответствующих условий. Для того чтобы различать дуги, выходящие из вершины распознавателя (одна из них соответствует условию, другая – его отрицанию), их размечают знаками +, – или 1, 0 соответственно. Для наглядности вершины распознаватели и вершины преобразователи в графовом представлении рисуют в виде различных геометрических фигур*). На рис. 2.2 изображен пример бинарной схемы программы. Легко увидеть связь между схемами рис. 2.1 и 2.2. Эти две схемы эквивалентны в том смысле, что при любой интерпретации базовых операторов и условий эти схемы выполняют один и тот же оператор (обе схемы детерминированы). Бинарная схема программы всегда детерминирована и полностью определена. Читателю предлагается доказать, что для любой детерминированной схемы программы можно построить эквивалентную ей бинарную схему.

Рассмотренные здесь конструкции допускают естественные обобщения. Например, можно рассматривать схемы программ с многими начальными и многими заключительными состояниями. С помощью схем с несколькими заключительными состояниями удобно представлять распознающие алгоритмы, т.е. такие алгоритмы, результат выполнения которых определяется не только состоянием информационной среды в момент оконча-

*) Существуют даже стандарты на графическое представление схемы программ (блок-схем) в технической документации.

ния вычислений, но и тем, в каком из заключительных состояний этот процесс закончится. Полезно также рассматривать частичные интерпретации, в которых базовые условия интерпретируются как частично определенные предикаты, а базовые функции – как частичные преобразования информационной среды. Можно рассматривать и недетерминированные интерпретации. В этом случае базовые операторы могут интерпретироваться как многозначные преобразования информационной среды . В случае частичных интерпретаций возникает вопрос о том, как доопределить булевы операции, относительно которых замкнуто множество элементарных условий, на частично определенные аргументы. Возникающие при этом трехзначные логики будут рассмотрены в связи с алгеброй алгоритмов в § 4 этой главы. Заметим, что при переходе к частичным интерпретациям, вообще говоря, теряется возможность перехода от произвольных к бинарным схемам программ даже для детерминированного случая.

Множество базовых условий и операторов могут быть наделены определенной структурой. Эта структура может накладывать ограничения на интерпретации, которые разрешается использовать для заданного базиса. Например, в языках программирования в качестве базовых операторов используются присваивания, вызовы подпрограмм, операторы ввода-вывода и т.п., которые имеют сложную структуру, в частности используют выражения,составленные из переменных, констант и операций, определенных на множестве значений переменных. Если внутренняя структура базовых условий и операторов нас не интересует, то мы говорим об абстрактном базисе и абстрактных схемах программ. Если же базис связывается с определенным классом интерпретаций, то добавляются соответствующие слова, характеризующие этот класс интерпретаций. Так, например, возникают базисы и схемы программ над памятью, рассматриваемые в следующем параграфе.

Как уже было сказано, всякая $U-Y$-схема программы A, интерпретированная на множестве B, определяет некоторое (многозначное) преобразование $f_A: B \to 2^B$ этого множества. Возникает вопрос о том, как связано это преобразование с базовыми условиями и операторами. Ответ на этот вопрос может быть получен с помощью аппарата алгебры отношений, рассмотренного в предыдущей главе. Результатом будет теорема анализа схем программ, аналогичная теореме 9.7 гл. 1.

При этом удобно перейти от алгебры отношений к изоморфной ей алгебре многозначных преобразований, рассматривая операции умножения, объединения и итерации как операции над преобразованиями. Все множества, участвующие в определении схемы программы, т.е. множества U, Y, A и T, предполагаются конечными.

Если υ – условие (предикат), определенное на множестве B, а P: $B \to 2^B$ – произвольное преобразование множества B, то через υ/P будем обозначать ограничение P на множество таких $b \in B$, что $\upsilon(b) = 1$, т.е. $b \xrightarrow{\upsilon/P} b' \iff \upsilon(b) = 1,\ b' \in P(b)$. Преобразование υ/P называется *фильтром* P с помощью условия υ. Каждое условие u можно представить с помощью фильтра u/ϵ тождественного преобразования ϵ по этому условию. Если U – множество условий, то $U/\epsilon = \{ u/\epsilon \mid u \in U\}$.

Теорема 1.1 (первая теорема анализа схем программ). *Преобразование f_A, выполняемое $U - Y$-схемой программы A, интерпретированной на множестве B, регулярно относительно множества $\hat{U}/\epsilon \cup Y$. При этом регулярное выражение, представляющее f_A через преобразования множества $\hat{U}/\epsilon \cup Y$, не зависит от интерпретации.*

Пусть $a^{(0)}$ – начальное, $a^{(1)}$ – заключительное состояние схемы A. Рассматривая схему A как управляющую компоненту дискретного преобразователя $A \times B$ и используя построения теоремы 9.7 гл. 1, получим, что $f_A = f_{a^{(0)}}$, где $f_{a^{(0)}}$ есть соответствующая компонента наименьшего решения $(f_A)_{a \in A}$ системы уравнений

$$f_a = \bigcup_{a' \in A} f_{aa'} \circ f_{a'}, \quad a \in A, \quad a \neq a^{(1)},$$

$$f_{a^{(1)}} = \epsilon.$$

Осталось найти элементарные преобразования $f_{aa'}$. Для этого рассмотрим все переходы $a \xrightarrow{u_1/q_1} a', \ldots, a \xrightarrow{u_m/q_m} a'$ из a в a'. Нетрудно заметить, что

$$f_{aa'} = u_1/q_1 \cup \ldots \cup u_m/q_m.$$

Действительно, $b \xrightarrow{f_{aa'}} b' \iff u_i(b) = 1,\ bq_i = b' \iff b \xrightarrow{u_i/q_i} b'$ для некоторого $i = 1, \ldots, m$. Поскольку $u_1/y_1 \ldots y_k = (u/\epsilon) \circ y_1 \circ \ldots \circ y_k$, получаем, что все коэффициенты системы уравнений для f_a регулярны относительно $\hat{U}/\epsilon \cup Y$. Поэтому и наименьшее решение будет регулярным относительно $\hat{U}/\epsilon \cup Y$. Теорема доказана.

Заметим, что теорема 1.1 имеет место как для детерминированных, так и для недетерминированных интерпретаций с частично определенными условиями. В детерминированном случае она может быть усилена путем замены множества $\hat{U}/\epsilon$ множеством $U/\epsilon \cup \bar{U}/\epsilon$, где $\bar{U} = \{\bar{u} \mid u \in U\}$. Действительно, если базовые условия определены, то в выражениях для $f_{aa'}$ можно перейти от элементарных условий к базовым с помощью следующих очевидных соотношений:

$$(u \vee v)/\epsilon = u/\epsilon \cup v/\epsilon,$$

$$(u \vee v)/\epsilon = (u/\epsilon) \cdot (v/\epsilon) = (v/\epsilon) \cdot (u/\epsilon).$$

Интерпретированные схемы программ представляют собой один из наиболее удобных и употребительных способов представления алгоритмов. Разумеется, схемы программ, используемые для этой цели, должны обладать определенными свойствами конструктивности. Например, функции f_α и f_y для базовых операторов и условий должны быть вычислимыми. Более того, если множества U и Y бесконечны, то вычислимыми должны быть также функции $F(\alpha, b) = f_\alpha(b)$ и $G(y, b) = f_y(b)$, где $\alpha \in U$, $y \in Y$. Множество переходов, ведущих из данного состояния, должно быть перечислимым и т.д. Обычно требования конструктивности к самой схеме ограничиваются условием конечности. Если же говорят о бесконечных схемах, то рассматривают различного рода частные случаи.

Рассматривая частичные интерпретации, можно замкнуть множество базовых операторов относительно операции взятия фильтра по элементарно-

му условию. При этом можно перейти к схемам, у которых каждый переход имеет вид $a \xrightarrow{y} a'$, поскольку переход $a \xrightarrow{u/y} a'$ можно будет тогда заменить на $a \xrightarrow{1/(u/y)} a'$. Этот частный случай схем возникает при простейшем способе перехода от произвольного дискретного преобразователя $S \subset A \times B$ к схеме программы. Действительно, S можно рассматривать как интерпретированную схему программы, если для любой пары состояний a и a' управляющей компоненты определить один-единственный переход $a \xrightarrow{1/f_{aa'}} a'$. Для того чтобы получить другие нетривиальные представления S в виде схемы программы, следует выбрать базис (U, Y) так, чтобы каждое элементарное преобразование $f_{aa'}$ можно было представить в виде дизъюнкции фильтров. Задача эта имеет чрезвычайно важное практическое значение и решается обычно для целого класса алгоритмов (дискретных преобразователей).

Выбор базиса для класса алгоритмов, или алгоритмического базиса, во многом определяет технологию программирования и проектирования дискретных систем и характер используемых алгоритмических языков. Однако общих методов и теории, которая имела бы практическое значение, для задачи выбора алгоритмического базиса в настоящее время не существует. Некоторые практические рекомендации будут обсуждены в ч. II книги.

§ 2. Алгоритмические языки

Схемы программ можно представлять в текстовом виде как программы с переходами. Пусть каждое состояние схемы A обозначено некоторым символом — меткой, а для базовых условий и операторов определен соответствующий язык для их записи. Предположим, что схема A бинарная. Тогда каждому состоянию a поставим в соответствие некоторое предложение. Если a есть преобразователь, отмеченный базовым оператором y, а дуга, выходящая из a, ведет в a', то поставим ему в соответствие предложение a: (y; **на** a'). Если же a — распознаватель, отмеченный условием, дуга, отмеченная знаком +, ведет в a', и дуга, отмеченная знаком −, ведет в a'', то поставим в соответствие состоянию a предложение (a: **если** u **то на** a' **иначе на** a''). Заключительному состоянию $a^{(1)}$ поставим в соответствие предложение $a^{(1)}$: СТОП. Выпишем все предложения в произвольном порядке, но так, чтобы предложение, соответствующее начальному состоянию, было первым, а предложение, соответствующее заключительному, — последним. Полученный программный текст определяет схему программы однозначно с точностью до изоморфизма и сам строится по схеме программы однозначно с точностью до перестановки операторов, обозначения состояний и выражений для элементарных условий.

Использованный для представления схем программ язык программирования является очень ограниченным фрагментом реальных алгоритмических языков, используемых на практике. Кроме того, средств этого языка не достаточно для того, чтобы представлять произвольные схемы программ. Сейчас мы рассмотрим более богатый алгоритмический язык АО, достаточный для представления произвольных схем и содержащий

наиболее употребительные конструкции реальных языков программирования. Этот язык в дальнейшем будет расширен и использован для представления примеров программ и алгоритмов проектирования. Рассматриваемый язык является абстрактным в том смысле, что некоторые его конструкции не раскрываются до конца. Добавляя определение соответствующих понятий, можно получать различные конкретные языки, явным образом использующие интерпретацию алгоритмического базиса.

Определим синтаксис языка АО, пользуясь общеупотребительными синтаксическими определениями.

⟨ АО–программа ⟩ ::= ⟨ непустой оператор ⟩.

⟨ оператор ⟩ ::= ⟨ базовый оператор ⟩ | ⟨ пустой оператор ⟩ |
⟨ переход ⟩ | ⟨ фильтр ⟩ | ⟨ выбор ⟩ | ⟨ условный оператор ⟩ |
⟨ составной оператор ⟩

⟨ пустой оператор ⟩ ::=

⟨ переход ⟩ ::= **на** ⟨ метка ⟩

⟨ фильтр ⟩ ::= ⟨ условие ⟩ ⟨ вертикальная черта ⟩ ⟨ оператор ⟩

⟨ условие ⟩ ::= ⟨ конъюнкция ⟩ | ⟨ конъюнкция ⟩ ∨ ⟨ условие ⟩

⟨ конъюнкция ⟩ ::= ⟨ литера ⟩ | ⟨ литера ⟩ ∧ ⟨ конъюнкция ⟩

⟨ литера ⟩ ::= ⟨ базовое условие ⟩ | ¬⟨ литера ⟩ | ((⟨ условие ⟩))

⟨ выбор ⟩ ::= ⟨ оператор отличный от выбора ⟩ ∨ ⟨ оператор ⟩

⟨ условный оператор ⟩ ::= **если** ⟨ условие ⟩ **то** ⟨ последовательность операторов ⟩ **иначе** ⟨ последовательность операторов ⟩ **конец если**

⟨ последовательность операторов ⟩ ::= ⟨ оператор ⟩ | ⟨ оператор ⟩;
⟨ последовательность операторов ⟩ | ⟨ метка ⟩ : ⟨ последовательность операторов ⟩

⟨ составной оператор ⟩ ::= **начало** ⟨ последовательность операторов ⟩ **конец** | **начало метки** ⟨ последовательность меток ⟩;
⟨ последовательность операторов ⟩ **конец**

⟨ последовательность меток ⟩ ::= ⟨ метка ⟩ | ⟨ метка ⟩, ⟨ последовательность меток ⟩

Понятия ⟨ базовое условие ⟩, ⟨ базовый оператор ⟩ и ⟨ метка ⟩ заранее не определяются и уточняются по мере необходимости. Определения понятий ⟨ непустой оператор ⟩ и ⟨ оператор, отличный от выбора ⟩ очевидны. Описанный здесь строгий синтаксис расширяется следующим образом. Перед любым понятием и после него разрешается вставлять любое число пробелов. Перед **то** и **иначе** можно вставлять запятую. Вместо точки с запятой можно употреблять точку с пробелом. Вместо операторных скобок **начало**, **конец** можно употреблять круглые открывающую и закрывающую скобки. Вместо **конец если** можно употреблять **ке**.

Кроме дизъюнкции, конъюнкции и отрицания для образования условий могут использоваться также и другие пропозициональные связки с необходимыми соглашениями относительно старшинства операций. Соответствующие синтаксические правила включаются в определение понятия ⟨ условие ⟩ и в конечном счете выражаются через понятие ⟨ базовое условие ⟩. При уточнении неопределяемых понятий требуется выдержать единственное ограничение: синтаксис всех понятий должен оставаться однозначным, т.е. каждый текст, построенный с помощью некоторого правила, должен единственным образом разлагаться на составляющие в соответствии с единственно определяемой альтернативой правой части этого правила.

Следующая задача состоит в том, чтобы строго определить семантику языка АО. Это будет сделано путем построения отображения, которое каждой программе ставит в соответствие некоторую схему программы над заданным базисом. Для того чтобы определить указанное отображение, на множестве схем программ будут определены некоторые операции, которые превратят это множество в алгебру. При этом мы видоизменим понятие "схема программы", перейдя к схемам с размеченными состояниями. Предположим, что понятие ⟨ метка ⟩ уже определено. *Схемой с размеченными состояниями* называется схема программы вместе с функцией φ: $L \to A$, которая называется *разметкой*. Разметка определена на конечном множестве L меток и каждой метке l из этого множества ставит в соответствие состояние $\varphi(l)$, отмеченное этой меткой. Если множество L пусто, то размеченная схема называется *чистой*. Изоморфизм двух размеченных схем должен сохранять их разметки, т.е. отображение γ: $A \to A'$ размеченной схемы A с разметкой φ в размеченную схему A' с разметкой φ' есть изоморфизм, если γ есть изоморфизм чистых схем и $\varphi(l) = a \Rightarrow \varphi'(l) = \gamma(a)$. Схемы программ, как чистые, так и размеченные, всегда будут рассматриваться с точностью до изоморфизма. Заметим, что разметка не обязана быть взаимно однозначной, поэтому некоторые состояния могут быть отмечены несколькими метками, а некоторые — вовсе неотмеченными.

Прежде чем определять основные операции над схемами программ, введем некоторые вспомогательные операции. Первая операция — *отождествление состояний*. Пусть ρ есть отношение эквивалентности на множестве состояний схемы A. Рассмотрим фактор-схему A/ρ. Состояниями этой схемы являются классы отождествленных состояний. Если Ω и Ω' — два класса, то $\Omega \overset{u/q}{\to} \Omega' \Leftrightarrow$ существуют $a \in \Omega$ и $a' \in \Omega'$ такие, что $a \overset{u/q}{\longrightarrow} a'$. Разметка φ: $L \to A$ преобразуется в новую разметку φ': $L \to A/\rho$ так, что $\varphi'(l) = \rho(\varphi(l))$. Начальное состояние схемы A/ρ есть класс, которому принадлежит начальное состояние схемы A; заключительное состояние определяется аналогично. О схеме A/ρ будем говорить, что она получается из схемы A *отождествлением состояний*.

На рис. 2.3 показаны две схемы. Вторая получается из первой путем отождествления состояний a_0 и a_3. Состояние $\overline{a}_0$ второй схемы представля-

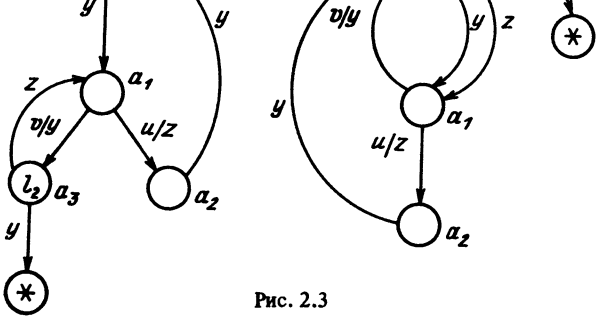


Рис. 2.3

ет собой класс $\{a_0, a_3\}$ отождествленных состояний первой. Обе схемы имеют нетривиальную разметку метками l_1 и l_2, но если в первой эти метки отмечали два разных состояния, то во второй они отмечают одно и то же состояние.

Вторая вспомогательная операция – *объединение схем*. Она применяется к чистым схемам: $(A, T) \cup (A', T') = (A \cup A', T \cup T')$. Таким образом, объединение – это теоретико-множественное объединение состояний и переходов. В результате объединения получается схема с двумя начальными и двумя заключительными состояниями. Поэтому для того, чтобы получить обычную схему, необходимо изменить настройку или выполнить дальнейшие отождествления. Заметим, что при замене одной из объединяемых схем изоморфной результат объединения, вообще говоря, изменится. Впрочем, в дальнейшем будут объединяться только схемы с непересекающимися множествами состояний.

Теперь перейдем к определению основных операций над схемами программ. Их будет пять: две бинарные и три типа унарных операций. Бинарные операции: последовательная композиция $A * A'$ и альтернативная композиция $A \vee A'$. Унарные: отметка $l{:}A$, снятие метки $(l)A$ и фильтр u/A. Здесь A и A' – схемы, l – метка, u – элементарное условие.

Для того чтобы получить последовательную композицию двух размеченных схем (A, φ) и (A', φ') с непересекающимися множествами состояний, сначала строим объединение $A \cup A'$ чистых схем программ, затем делаем отождествления. Отождествляем заключительное состояние схемы A с начальным состоянием схемы A', а также любые два состояния схем A и A', отмеченные одной и той же меткой. Полученную схему размечаем функцией ψ, равной φ на состояниях схемы A и равной φ' на состояниях схемы A'. Выполненное ранее отождествление гарантирует однозначность функции ψ. Если множества состояний схем A и A' пересекаются, то сначала переходим к изоморфным схемам с непересекающимися состояниями, а затем выполняем отождествление. Наконец, в полученной схеме изменяем настройку. Начальным объявляем начальное состояние схемы A, заключительным – заключительное состояние схемы A'. Таким образом, результат операции определен однозначно с точностью до изоморфизма. На рис. 2.4 схематично показано построение последовательной композиции. Операция последовательной композиции ассоциативна.

Для построения альтернативной композиции поступаем следующим образом. Строим объединение чистых схем $A \cup A'$, отождествляем начальные состояния этих схем, их заключительные состояния и состояния, отмеченные одинаковыми метками. Новую разметку строим так же, как и в случае последовательной композиции. Настройку оставляем прежней. Если множества состояний исходных схем пересекаются, то освобождаемся от этого, переходя к изоморфным схемам с непересекающимися множествами состояний. Схема получения альтернативной композиции показана на рис. 2.5. Операция альтернативной композиции ассоциативна и коммутативна.

Операция отметки $l{:}A$ определена для любой метки l и состоит в следующем. Начальное состояние схемы A отмечается меткой l. Если в A есть другое состояние, уже отмеченное меткой l, то это состояние до изменения

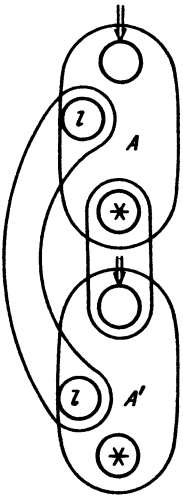


Рис. 2.4

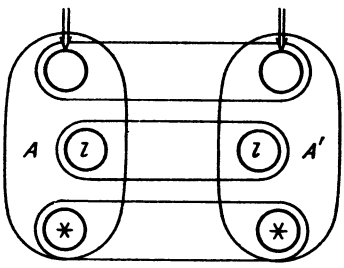


Рис. 2.5

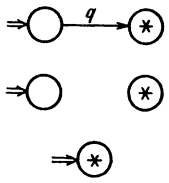


Рис. 2.6

разметки отождествляется с начальным. Операция $(l)A$ снимает метку l с состояния, которое она отмечает, и оставляет схему неизменной, если такого состояния нет. Вместо $(l_1)\ldots(l_m)A$ будем писать $(l_1,\ldots,l_m)A$. Следует обратить внимание на то, что, вообще говоря, $(l)(l\colon A)\neq A$, поскольку после отметки $l\colon A$ в схеме A может произойти отождествление, которое сохранится после снятия метки. Будем предполагать, что множество меток, которые разрешается использовать для построения размеченных схем программ, всегда бесконечно.

Операция взятия фильтра u/A определена для любого элементарного условия u и состоит в том, что любой переход $a^{(0)} \xrightarrow{v/q} a$, ведущий из начального состояния $a^{(0)}$, преобразуется в переход $a^{(0)} \xrightarrow{u\wedge v/q} a$. Если же из начального состояния нет переходов, то к схеме A добавляются новое состояние b и переход $b \xrightarrow{u} a^{(0)}$, после чего изменяется настройка: b становится начальным состоянием вместо $a^{(0)}$.

Схемы, изображенные на рис. 2.6, будем называть *элементарными.* Схему первого типа будем обозначать так же, как и элементарный оператор q, выполняемый на ее единственном переходе. Схему второго типа с двумя состояниями и пустым множеством переходов обозначим через 0. Схему с одним состоянием и пустым множеством переходов обозначим λ.

Теорема 2.1. *Любая конечная размеченная U-Y-схема программы может быть построена из элементарных схем с помощью основных операций алгебры схем программ.*

Действительно, рассмотрим произвольное состояние a схемы A, и пусть $a \xrightarrow{u_1/q_1} a_1,\ldots,a \xrightarrow{u_m/q_m} a_m$ — все переходы, которые ведут из a. Если $m>0$, то поставим в соответствие состоянию a схему $S_a = 0*(l_1\colon\ldots\colon l_k\colon a\colon(u_1/q_1*a_1\colon 0\vee\ldots\vee u_m/q_m*a_m\colon 0))$, где $l_1,\ldots,l_k$ —

все метки, которые отмечают состояние a. Для того чтобы можно было сами состояния использовать в качестве меток, достаточно перейти к изоморфной схеме, использовав в качестве состояний метки, которые еще не были использованы в начальной разметке. Если же $m = 0$, то $S_a = 0 * (l_1 : \ldots : l_k : a : 0)$. Пусть $A = \{a_1, \ldots, a_n\}$, $a^{(0)}$ – начальное, $a^{(1)}$ – заключительное состояние. Тогда A можно представить в виде выражения $(a_1, \ldots, a_m)((a^{(0)} : (S_{a_1} \vee \ldots \vee S_{a_n})) * (a^{(1)} : \lambda))$. Теорема доказана.

Вернемся теперь к языку АО. Предположим, что синтаксис этого языка дополнен правилами определения понятий ⟨ базовое условие ⟩ и ⟨ базовый оператор ⟩. Каждой программе P в языке АО поставим в соответствие U–Y-схему программы $\sigma(P)$, где U – множество всех базовых условий, Y – множество всех базовых операторов, использованных в этой программе. Функцию σ определим не только для программ, но и для других понятий языка АО (это определение будет рекурсивным и строится в соответствии с правилами грамматики):

$\sigma(P.) = \sigma(P)$;
$\sigma(q) = q$;
$\sigma(\ ) = \epsilon$;
$\sigma(\textbf{на}\ l) = l : 0$;
$\sigma(u/P) = u/\sigma(P)$;
$\sigma(P \vee Q) = \sigma(P) \vee \sigma(Q)$;
$\sigma(\textbf{если}\ u\ \textbf{то}\ R\ \textbf{иначе}\ R'\ \textbf{кс}) = u/\sigma(R) \vee \bar{u}/\sigma(R')$;
$\sigma(\textbf{если}\ u\ \textbf{то}\ R\ \textbf{кс}) = u/\sigma(P) \vee \bar{u}/\epsilon$;
$\sigma(\textbf{начало}\ R\ \textbf{конец}) = \sigma(R)$;
$\sigma(\textbf{начало метки}\ l_1, \ldots, l_m ; R\ \textbf{конец}) = (l_1, \ldots, l_m)\sigma(R)$;
$\sigma(P;R) = \sigma(P) * \sigma(R)$;
$\sigma(l : R) = l : \sigma(R)$.

Здесь P и Q – операторы, q – базовый оператор, u – условие, R и R' – последовательности операторов, $l, l_1, \ldots, l_m$ – метки. Сопоставляя правила строгого синтаксиса и равенства, определяющие функцию σ, легко видеть, что эта функция определена на всех программах, удовлетворяющих строгому синтаксису, и каждой такой программе ставит в соответствие схему программы, определенную однозначно с точностью до изоморфизма. Более того, поскольку каждому правилу языка соответствует некоторая операция в алгебре схем программ, выражения языка АО можно рассматривать как выражения этой алгебры, записанные в языке АО.

При определении семантики алгоритмических языков следует различать два вида семантики – динамическую и функциональную. Динамическая семантика дает ответ на вопрос о том, как выполняется программа, т.е. какие процессы вычислений она порождает. Обычно динамическую семантику языка определяют, описывая абстрактную вычислительную машину, выполняющую программы данного языка. Мы предпочитаем более прямой путь – сопоставление каждой программе множества порождаемых ею процессов вычислений. Для алгоритмических языков, употребляемых в практике программирования и проектирования вычислительных систем, наиболее естественно задавать это множество с помощью схем программ. Таким образом, функция σ есть семантическая функция, определяющая динамическую семантику языка АО. Если базовые условия и операторы

интерпретированы на информационной среде B, то дискретный преобразователь $\widehat{\sigma(P)}$ над B (или $\sigma(P)$ над B) порождает процессы вычислений, определенные программой P. Их можно рассматривать также как процессы выполнения программы P. Язык АО можно использовать также и как язык представления неинтерпретированных программ. Тогда возможные процессы выполнения таких программ (при всевозможных интерпретациях) порождаются системой $\widehat{\sigma(P)}$ или самой $\sigma(P)$.

Что же касается абстрактной машины, интерпретирующей рассматриваемый язык, то ее можно определить как дискретную систему, которая при подходящей настройке является реализацией преобразователя $\widehat{\sigma(P)}$ для любой программы P данного языка. Такое описание может быть полезным в связи с рассмотрением вопросов реализации алгоритмических языков. Другой способ задания семантики языка состоит в построении функции перевода рассматриваемого языка в язык с известной семантикой. Язык АО может быть использован в этом смысле как базовый язык, поскольку на нем можно описать любую схему программы. Это следует из следующей теоремы.

Т е о р е м а 2.2. *Для любой конечной размеченной базовой схемы программы A существует программа P языка АО такая, что $\sigma(P)$ изоморфна A.*

Доказательство можно получить с помощью конструкции, использованной при доказательстве теоремы 2.1. Действительно,

σ(**начало метки** l; **на** l **конец**) = 0;

σ(**если** u_1 **то** q_1; **на** a_1 **ке** v ... v **если** u_m **то** q_m; **на** a_m **ке**) =

$= u_1/q_1 * a_1 :\ 0$ v ... v $u_m/q_m * a_m : 0$,

поэтому все выражения доказательства теоремы 2.1 можно записать в языке АО. Теорема доказана.

Функциональная семантика алгоритмического языка дает ответ на вопрос о том, что вычисляет данная программа, какую функцию. Для языка АО так же, как и для других языков, для которых динамическая семантика определена с помощью U–Y-схем программ, функциональная семантика может быть определена как преобразование, выполняемое дискретным преобразователем $\sigma(P)$ над информационной средой, на которой интерпретирован алгоритмический базис (U, Y). В частности, если базис языка АО интерпретирован на B, то его функциональная семантика – это функция $\overline{\sigma}$, которая каждой программе P ставит в соответствие преобразование $\overline{\sigma}(P) = f_{\sigma(P),B} : B \to 2^B$.

Определение функциональной семантики языка АО через его динамическую семантику имеет один недостаток. Он состоит в следующем. Как уже установлено, всякую схему программы A можно записать в виде некоторого выражения алгебры схем программ, зависящего от базовых условий и операторов, использованных в этой схеме. С другой стороны, преобразование f_A выражается через те же базовые условия и операторы в алгебре многозначных преобразований. Однако операции алгебры преобразований и алгебры схем программ связаны очень сложным образом. Поэтому преобразование, выполняемое схемой A, построенной из A_1 и A_2, не выражается через преобразования, выполняемые схемами A_1 и A_2, в алгебре преобразований. Виновата в этом операция отождествления состояний.

Действительно, если A_1 и A_2 не имеют общих меток, то $f_{A_1 * A_2} = f_{A_1} \circ f_{A_2}$, $f_{A_1 \vee A_2} = f_{A_1} \cup f_{A_2}$ и т.д. Однако, если при композиции происходит отождествление, указанные связи немедленно разрушаются. Преодоление трудностей, вызванных рассмотренными обстоятельствами, связано с идеями алгебры алгоритмов и структурного программирования, которые будут рассмотрены в § 4.

§ 3. Схемы программ над памятью

В этом параграфе будет рассмотрена наиболее распространенная модель информационной среды дискретного преобразователя — память. Память определяется прежде всего множеством V переменных и областью D значений, которые эти переменные могут принимать (область данных). Состояние памяти представляет собой частичное отображение $b \subset V \to D$. Переменная $v \in V$ при заданном состоянии памяти b имеет значение $b(v)$, если b определено на v. В противном случае говорят, что значение переменной v не определено. Если значение переменной v определено, то это означает, что в физическом устройстве, используемом для реализации памяти, должно быть выделено место, где хранится значение этой переменной. Это место может находиться на регистрах, в оперативной памяти или на внешних носителях. Оно может быть определено заранее и не зависит от текущего состояния памяти или изменяется с изменением состояния. В первом случае говорят о *статическом*, во втором — о *динамическом* распределении памяти. При статическом распределении памяти переменную можно отождествить с соответствующим запоминающим устройством — регистром, ячейкой оперативной памяти и т.п. Неопределенное значение может интерпретироваться различным образом в зависимости от реализации. Например, при динамическом распределении памяти тот факт, что значение переменной не определено, может означать, что для данной переменной память не выделена, при статическом — что значение переменной может быть произвольным.

Кроме множества переменных и их значений должно быть задано также непустое множество B допустимых состояний памяти, которое включается в множество $\Gamma(V, D)$ всех частичных отображений из V в D. Условие $b \in B$ накладывает определенные ограничения на возможные значения переменных при одном и том же допустимом состоянии памяти. Эти ограничения могут быть двух родов. Во-первых, B ограничивает множество допустимых значений каждой переменной v до множества $D(v) \subset D$. По определению $d \in D(v) \iff$ существует $b \in B$ такое, что $b(v) = d$. Другой тип ограничений состоит в том, что значения переменных могут зависеть друг от друга. Например, значение, записанное на двоичном регистре, есть функция значений, записанных в его разрядах, значение массива однозначно определяется значениями его элементов. Зависимости между значениями переменных в допустимых состояниях памяти могут быть сколь угодно сложными. Однако мы ограничимся здесь рассмотрением лишь двух простых типов памяти — *простой* и *функциональной*, играющих основную роль в приложениях.

Простая память допускает любые состояния b, удовлетворяющие условию: для любой переменной $v \in V$, если $b(v)$ определено, то $b(v) \in D(v)$,

где $D(v)$ – заранее заданные области значений переменных. Если для всех $v \in V$ имеет место $D(v) = D$, память называется *бестиповой*, в противном случае – *типизированной*.

Память (V, B) называется *функциональной*, если в множестве V выделено подмножество V_0, элементы которого называются простыми переменными, а множество допустимых состояний определяется следующим образом. Простая переменная v может принимать любые значения из $D(v)$, а значение любой другой переменной однозначно определяется значениями всех простых переменных. Очевидно, существует взаимно однозначное соответствие между состояниями всей памяти и состояниями простых переменных (т.е. ограничениями всего состояния на множество V_0). Поэтому рассмотрение функциональной памяти может быть сведено к простой, хотя это и не всегда целесообразно.

Рассмотрим простую бестиповую память (V, B), т.е. случай, когда $B = \Gamma(V, D)$. Для задания алгоритмического базиса обычно используют операции и предикаты, заданные на области данных D.

Предположим, что для обозначения операций и предикатов заданы множества символов (сигнатуры) Ω и Π. Каждому символу приписана *арность* – целое неотрицательное число. Если символ ω имеет арность n, то ему сопоставлена n-арная операция (функция) $f_\omega: D^n \to D$. Вместо $f_\omega(d_1, \ldots, d_n)(d_1, \ldots, d_n \in D)$ будем писать $\omega(d_1, \ldots, d_n)$, отождествляя символ операции с самой операцией, если такое отождествление не приводит к недоразумениям. *Нульарные операции* – это обозначения констант, выделенных сигнатурой элементов области D. Элементы π множества Π являются *символами предикатов*. Они также имеют арности, и если символу π приписана арность n, то он определяет *n-местный предикат* $f_\pi: D^n \to \{0, 1\}$. Так же как и для операций, будем писать $\pi(d_1, \ldots, d_n)$ вместо $f_\pi(d_1, \ldots, d_n)$. *Нульместные предикаты* – это различные обозначения для истинностных значений 0, 1. Множество D вместе с операциями $(f_\omega)_{\omega \in \Omega}$ и предикатами $(f_\pi)_{\pi \in \Pi}$ называется *алгеброй данных* с сигнатурами операций и предикатов Ω и Π или *Ω-Π-алгеброй данных**).

Используя символы сигнатуры Ω и символы переменных, можно строить выражения в алгебре D, которые также будем называть *Ω-термами* над V или просто *термами*, если известно, о какой сигнатуре и каких переменных идет речь. Формальное определение терма известно:

1. Всякий символ переменной или символ нульарной операции есть терм.
2. Если ω есть символ n-арной операции, а $t_1, \ldots, t_n$ – термы, то $\omega(t_1, \ldots, t_n)$ тоже терм.
3. Других термов нет.

Вместо синтаксиса $\omega(t_1, \ldots, t_n)$ в конкретных алгебрах могут использоваться другие синтаксические правила для образования выражения, обозначающего результат применения операции ω к термам $t_1, \ldots, t_n$. Например, можно использовать левую или правую бесскобочную запись, инфикс-

---

*) На самом деле D является алгебраической системой, но мы употребляем более простой термин "алгебра", предполагая, что допускается рассмотрение алгебр с предикатами. Впрочем, если истинностные значения являются элементами области D, а предикаты рассматриваются как операции, то употребляемая нами терминология вполне согласуется с терминологией, принятой в алгебраической литературе.

ные обозначения для бинарных операций и вообще любые синтаксические схемы вида $p_1 t_1 \ \ldots \ p_n t_n p_{n+1}$, допускающие однозначный синтаксический анализ. Здесь $p_1, \ldots, p_n$ – слова в некотором алфавите (возможно, пустые), а $p_1(\,) \ \ldots \ (\,)p_{n+1}$ называется *схемой построения терма* с помощью операции ω.

Пусть $t(v_1, \ldots, v_n)$ есть терм над V, построенный из переменных $v_1, \ldots, v_n$. Подставляя вместо $v_1, \ldots, v_n$ значения $d_1, \ldots, d_n \in D$, можно выполнить все операции и получить значение $t(d_1, \ldots, d_n) \in D$. В частности, если $b \in B$ есть допустимое состояние памяти, в котором определены значения переменных $v_1, \ldots, v_n$, то определено значение $t(b(v_1), \ldots, b(v_n))$, которое мы называем *значением терма* $t(v_1, \ldots, v_n)$ при состоянии памяти b и обозначаем через $b(t(v_1, \ldots, v_n))$. Очевидно, что $b(\omega(t_1, \ldots, t_n)) = \omega(b(t_1), \ldots, b(t_n))$ для любой n-арной операции ω.

В некоторых случаях удобно считать, что значение $b(t(v_1, \ldots, v_n))$ определено и в случае, когда не все переменные $v_1, \ldots, v_n$ определены. Классическим примером является функция (**если** $\pi(v_1)$ **то** v_2 **иначе** v_3), рассматриваемая как тернарная операция $\omega(v_1, v_2, v_3)$. Если $\pi(b(v_1)) = 1$, то $\omega(b(v_1), b(v_2), b(v_3))$ можно считать равным $b(v_2)$, даже если $b(v_3)$ не определено. Вопросы такого рода будут рассмотрены в главе о рекурсивных определениях. Сейчас можно считать, что $t(v)$ не определено, если v не определено.

Если π – символ n-арного предиката, а $t_1, \ldots, t_n$ – термы, то выражение $\pi(t_1, \ldots, t_n)$ определяет некоторое условие на B. Значение этого условия при заданном состоянии памяти $b \in B$ определяется как $\pi(t_1, \ldots, t_n)(b) = \pi(b(t_1), \ldots, b(t_n))$.

Множество всех выражений вида $\pi(t_1, \ldots, t_n)$ обозначим через $U(V, \Omega, \Pi)$. Элементы этого множества называются *базовыми условиями* над памятью V, определенными сигнатурой (Ω, Π). Правила вычисления значений базовых условий определяют их интерпретацию на информационной среде B. Эта интерпретация является частичной, поскольку $\pi(t_1, \ldots, t_n)$ определено, лишь если определены значения всех термов $t_1, \ldots, t_n$.

В качестве базовых операторов, действующих на информационной среде B, рассматривают операторы присваивания. Пусть $v_1, \ldots, v_n$ – различные переменные, $t_1, \ldots, t_n$ – термы. Оператор присваивания – это выражение вида $y = (v_1 := t_1, \ldots, v_n := t_n)$. Преобразование, выполняемое y на b, определяется следующим образом: $by = b'$, где $b'(v) = b(t_i)$, если $v = v_i$, и $b'(v) = b(v)$, если $v \neq v_i$ $(i = 1, \ldots, n)$. Равенства для $b'(v)$ требуют комментария, поскольку правая часть этих равенств может быть неопределенной.

Мы считаем, что равенство двух выражений истинно тогда и только тогда, когда обе части принимают одинаковые значения или обе не определены. Таким образом, by всегда определено, даже если правые части некоторых присваиваний не определены. В этом случае просто не будут определены в состоянии b' значения переменных, находящихся в левых частях соответствующих присваиваний. Каждый оператор присваивания y определяет функцию на множестве термов, если положить $y(v_i) = t_i$, $y(v) = v$ при $v \neq v_i$ $(i = 1, \ldots, n)$ и $y(\omega(s_1, \ldots, s_m)) = \omega(y(s_1), \ldots, y(s_m))$. Используя эту функцию, можно определить by с помощью равенства

$$(by)(v) = b(y(v)). \tag{3.1}$$

Множество всех операторов присваивания обозначим через $Y(V, \Omega, \Pi)$. Базис $(U(V, \Omega, \Pi), Y(V, \Omega, \Pi))$ назовем *стандартным базисом* над памятью V, определенным сигнатурой (Ω, Π), или (V, Ω, Π)-*базисом*. Если $U \subset U(V, \Omega, \Pi)$, $Y \subset Y(V, \Omega, \Pi)$, то U–Y-схема программы называется *стандартной схемой программы* над памятью V. Поскольку стандартный базис был введен как базис, интерпретированный средствами алгебры данных D, то схемы над памятью определены как интерпретированные схемы. В то же время легко видеть, что понятия стандартного базиса и стандартной схемы зависят только от множества V переменных и сигнатуры (Ω, Π) и могут рассматриваться независимо от интерпретации символов операций и предикатов, которая определяется семействами функций $(f_\omega)_{\omega \in \Omega}$ и $(f_\pi)_{\pi \in \Pi}$. Поэтому можно рассматривать и неинтерпретированные схемы программ над памятью. Интерпретация схемы над памятью, которая определяется произвольной Ω–Π-алгеброй D, называется *стандартной*. При изучении схем программ над памятью обычно рассматривают только стандартные интерпретации.

Переходя к рассмотрению типизированной памяти, выделим в D семейство подмножеств $(D_\xi)_{\xi \in \Xi}$ и предположим, что $D = \bigcup_{\xi \in \Xi} D_\xi$, и для любой переменной $v \in V$ имеет место $D(v) = D_\xi$ для некоторого $\xi \in \Xi$, однозначно определяемого переменной v. Элемент ξ называется *типом* переменной v, D_ξ – *областью значений* переменных типа ξ, Ξ – *сигнатурой* типов. Множество всех переменных типа ξ обозначим через V_ξ. Очевидно, что $V = \bigcup_{\xi \in \Xi} V_\xi$, а семейство $(V_\xi)_{\xi \in \Xi}$ образует разбиение множества V. Множество V с заданным на нем разбиением $(V_\xi)_{\xi \in \Xi}$ образует *множество типизированных переменных*, а область D с покрытием $(D_\xi)_{\xi \in \Xi}$ – *типизированную область данных*. Если V и D типизированы, то $\Gamma_\Xi(V, D)$ обозначает множество всех частичных отображений $b \subset V \to D$ таких, что $b(V_\xi) \subset D_\xi$ для всех $\xi \in \Xi$.

Символам операций и предикатов сигнатур Ω и Π теперь кроме арностей сопоставлены типы операций и предикатов. Если $\omega \in \Omega$ есть символ n-арной операции, то тип этой операции есть вектор $(\xi_1, ..., \xi_n, \xi)$, составленный из типов переменных, а сама операция представляет собой функцию f_ω: $D_{\xi_1} \times ... \times D_{\xi_n} \to D_\xi$. Аналогично, если $\pi \in \Pi$ есть символ n-арного предиката, то ему сопоставлен вектор $(\xi_1, ..., \xi_n)$ и f_π есть функция f_π: $D_{\xi_1} \times ... \times D_{\xi_n} \to \{0, 1\}$. Таким образом, семейство $(D_\xi)_{\xi \in \Xi}$ рассматривается как *многоосновная алгебра* (с предикатами) сигнатуры (Ω, Π, Ξ), или Ω–Π–Ξ-*алгебра*. Для краткости мы иногда будем обозначать эту алгебру так же, как и соответствующую типизированную область D. Рассматриваемые вместе с предикатами и операциями множества D_ξ называются обычно *типами данных*.

С помощью сигнатур Ω и Π так же, как и раньше, строятся термы и базовые условия над памятью V. Только теперь термы имеют типы, которые должны учитываться при образовании новых термов. Более точно, правила образования термов выглядят теперь так:

1. Всякий символ переменной типа ξ или нульарной операции типа (ξ) есть терм типа ξ.

2. Если ω есть символ операции типа $(\xi_1, ..., \xi_n, \xi)$, а $t_1, ..., t_n$ — термы типов $\xi_1, ..., \xi_n$, то $\omega(t_1, ..., t_n)$ есть терм типа ξ.

3. Других термов нет.

Значение терма при заданном состоянии памяти определяется так же, как и для нетипизированной памяти. В силу определения допустимого состояния памяти $b(\omega(t_1, ..., t_n)) = \omega(b(t_1), ..., b(t_n))$, и это значение определено, если $b(t_1), ..., b(t_n)$ определены. Базовые условия представляются в виде $\pi(t_1, ..., t_n)$, где π имеет тип $(\xi_1, ..., \xi_n)$, $t_1, ..., t_n$ — термы типов $\xi_1, ..., \xi_n$ соответственно.

При построении операторов присваивания правые и левые части также должны иметь одинаковые типы. Это обеспечивает в силу равенства (3.1) сохранение допустимости состояний при действии операторов присваивания, поскольку v и $y(v)$ имеют одинаковые типы. Таким образом, базис $(U(V, \Omega, \Pi, \Xi), Y(V, \Omega, \Pi, \Xi))$ для схем программ над типизированной памятью построен.

Применительно к типизированному базису будем также говорить о стандартных схемах программ над памятью и о стандартных интерпретациях. На многоосновные алгебры естественным образом переносятся все основные понятия универсальных алгебр и алгебраических систем. В частности, если $D = \bigcup_{\xi \in \Xi} D_\xi$ и $D' = \bigcup_{\xi \in \Xi} D'_\xi$ — две однотипные, т.е. с одинаковыми сигнатурами (Ω, Π, Ξ), многоосновные алгебры, то *гомоморфизм* γ: $D \to D'$ называется семейство $\gamma = (\gamma_\xi)_{\xi \in \Xi}$ отображений такое, что для любых $\xi \in \Xi$ γ_ξ: $D_\xi \to D'_\xi$, и если $\omega \in \Omega$ имеет тип $(\xi_1, ..., \xi_n, \xi)$, то $\gamma_\xi(\omega(d_1, ..., d_n)) = \omega(\gamma_{\xi_1}(d_1), ..., \gamma_{\xi_n}(d_n))$, а для предиката $\pi \in \Pi$ типа $(\xi_1, ..., \xi_n)$ из $\pi(d_1, ..., d_n) = 1$ следует $\pi(\gamma_{\xi_1}(d_1), ..., \gamma_{\xi_n}(d_n)) = 1$. Если $\pi(d_1, ..., d_n) = 1 \Leftrightarrow \pi(\gamma_{\xi_1}(d_1), ..., \gamma_{\xi_n}(d_n)) = 1$, то гомоморфизм называется *строгим*.

Рассмотрим теперь две многоосновные алгебры данных D и D', и пусть $B = \Gamma_\Xi(V, D)$, $B' = \Gamma_\Xi(V, D')$. Тогда всякий гомоморфизм γ: $D \to D'$ переносится на множества B и B'. Именно, можно рассматривать отображение γ': $B \to B'$, определенное равенством $(\gamma'(b))(v) = \gamma_\xi(b(v))$, если $v \in V_\xi$. Отображение γ' обладает следующим важным свойством. Если $y = (v_1 := t_1, ..., v_n := t_n)$ — оператор присваивания на типизированной памяти, то

$$\gamma'(by) = (\gamma'(b))y. \tag{3.2}$$

Действительно, $((\gamma'(b))y)(v) = (\gamma'(b))(y(v)) = \gamma_\xi(b(y(v))) = \gamma_\xi((by)(v)) = (\gamma'(by))(v)$.

Пусть теперь A — стандартная схема программы над типизированной памятью V. Множества B и B' представляют собой две различные интерпретации схемы A. Гомоморфизм γ индуцирует еще одно отображение γ'': $\hat{A} \times B \to \hat{A} \times B'$. Это отображение определяется равенством $\gamma''((a, u, y), b) = ((a, u, y), \gamma'(b))$.

Т е о р е м а 3.1. *Отображение γ'', индуцированное строгим гомоморфизмом γ: $D \to D'$, является реализующим гомоморфизмом дискретного преобразователя $\hat{A}$ над B в дискретный преобразователь $\hat{A}$ над B', согласованный с настройкой. Если γ отображает D на D', то $\hat{A}$ над B является пол-*

ной гомоморфной реализацией $\hat{A}$ над B'. Если D и D' изоморфны, то $\hat{A}$ над B и $\hat{A}$ над B' также изоморфны.

Покажем сначала, что образ допустимого состояния $\hat{A}$ над B при отображении γ'' допустим. Действительно, пусть $((a, u, y), b)$ – допустимое состояние $\hat{A}$ над B. Тогда $u(b) = 1$. Пусть $u = \varphi(\alpha_1, ..., \alpha_m)$, где φ – пропозициональная функция, $\alpha_1, ..., \alpha_m$ – базовые условия. Тогда $u(b) = \varphi(\alpha_1(b), ..., \alpha_m(b))$. Пусть $\alpha_i = \pi(t_1, ..., t_n)$. Тогда $\alpha_i(b) = \pi(b(t_1), ..., b(t_n)) = \pi(\gamma_{\xi_1}(b(t_1)), ..., \gamma_{\xi_n}(b(t_n))) = \pi((\gamma'(b))(t_1), ..., (\gamma'(b))(t_n)) = \alpha_i(\gamma'(b))$, где $\xi_1, ..., \xi_n$ – типы термов $t_1, ..., t_n$ соответственно. Поэтому $u(\gamma'(b)) = u(b)$, следовательно, состояние $((a, u, y), \gamma'(b))$ допустимо.

Покажем теперь, что из $((a, u, y), b) \to ((a', u', y'), b')$ следует $((a, u, y), \gamma'(b)) \to ((a', u', y'), \gamma'(b'))$. Действительно, $b' = by$, откуда в силу (3.2) $\gamma'(b') = \gamma'(by) = (\gamma'(b))y$. Кроме того, поскольку состояние $((a', u', y'), b')$ допустимо, то $((a', u', y'), \gamma'(b'))$ также допустимо, и, следовательно, $u'(\gamma'(b')) = 1$.

Из доказанных предложений вытекает первое утверждение теоремы. Заметим, что по определению γ: $D \to D'$ означает, что γ_ξ: $D_\xi \to D'_\xi$. Пусть теперь $b' \in B'$, $v \in V_\xi$. Если $b'(v) = d'$, то выберем $d \in D_\xi$ такое, что $\gamma_\xi(d) = d'$. Если же $b'(v)$ не определено, то положим $b(v)$ тоже не определено. Таким образом, построено состояние $b \in B$ такое, что $\gamma'(b) = b'$, и, значит, γ', а следовательно, и γ'' являются отображениями на. Наконец, если γ есть изоморфизм, то γ'' также является изоморфизмом.

Теорема 3.1 формализует один из типичных приемов, применяемых при разработке программ и алгоритмов, – переход от одного представления алгоритма к другому путем реализации данных и операций над ними. Из теоремы, в частности, следует, что переход от одной алгебры данных к изоморфной ничего не меняет, поскольку схемы программ над памятью с изоморфными алгебрами данных могут служить реализациями друг друга. Поэтому на начальных этапах проектирования алгоритмов алгебры данных рассматриваются с точностью до изоморфизма. В этом случае они называются *абстрактными алгебрами данных*, а их компоненты D_ξ – *абстрактными типами данных*.

Рассмотренный выше тип оператора присваивания является частным случаем операторов присваивания, используемых в языках программирования. Он не включает такие понятия, как переменные с индексами, имена полей в записях, указатели и т.п. Эти понятия можно рассмотреть с единой точки зрения с помощью идеи косвенного именования.

Предположим, что в сигнатуре типов Ξ выделено подмножество Ξ_0 и задано взаимно однозначное отображение α: $\Xi_0 \to \Xi$. Предположим также, что для $\xi \in \Xi_0$ имеет место $D_\xi = V_{\alpha\xi}$. Таким образом, в качестве значений переменных могут использоваться сами переменные. О переменных типа $\xi \in \Xi_0$ можно говорить, что они являются именами переменных типа $\alpha\xi$. Теперь, если терм t имеет тип $\xi \in \Xi_0$, то на состоянии памяти b можно вычислить не только его значение $b(t)$ типа ξ, но и значение второго ранга $b(b(t))$, которое имеет тип $\alpha\xi$. С другой стороны, если $\alpha^{-1}\xi$ определено, то каждую переменную типа ξ можно рассматривать как константу ти-

па $\alpha^{-1}\xi$, т.е. значение переменной типа $\alpha^{-1}\xi$. Типы данных D_ξ и их имена для $\xi \in \Xi_0$ называются *именующими*, а алгебра данных, в которой выделены именующие типы и задана функция α так же, как и память, построенная на этой алгебре, называется *алгеброй* (соответственно *памятью*) *с косвенным именованием*. Для всех допустимых интерпретаций базиса над памятью с косвенным именованием функции α и типы данных D_ξ ($\xi \in \Xi_0$) предполагаются фиксированными.

Для любого допустимого состояния памяти b, терма t и типа ξ определим функцию $\mathrm{val}_\xi(t, b)$ — значение терма t относительно типа ξ при состоянии памяти b. Пусть терм t имеет тип η. Рассмотрим два случая.

1. Если $t = v \in V_\eta$ и существует $m \geqslant 0$ такое, что $\alpha^{m-1}\eta = \xi$, то $\mathrm{val}_\xi(t, b) = b^m(t)$, где m минимально.

2. Пусть $t = \omega(t_1, ..., t_n)$, операция ω имеет тип $(\xi_1, ..., \xi_n, \eta)$, и существует $m \geqslant 0$ такое, что $\alpha^m\eta = \xi$. Тогда $\mathrm{val}_\xi(t, b) = b^m(\omega(d_1, ..., d_n))$, где $d_i = \mathrm{val}_{\xi_i}(t_i, b)$ $(i = 1, ..., n)$, m берется минимальным. Разумеется, все значения $d_1, ..., d_n$ должны быть определены.

Во всех других случаях $\mathrm{val}_\xi(t, b)$ считается неопределенным.

Действие оператора присваивания вида $y = (s := t)$, где s и t — термы, определяется следующим образом. Если s имеет тип ξ и $\alpha^{-1}\xi$ определено, то действие оператора y эквивалентно действию оператора $\mathrm{val}_{\alpha^{-1}\xi}(s, b) := \mathrm{val}_\xi(t, b)$. Если $\alpha^{-1}\xi$ не определено, то действие y также не определено. Групповое присваивание $y = (s_1 := t_1, ..., s_m := t_m)$ выполняется следующим образом. Сначала вычисляются значения $v_i = \mathrm{val}_{\alpha^{-1}\xi_i}(s_i, b)$, где ξ_i — тип терма s_i, если это возможно. Полученные переменные $v_1, ..., v_m$ должны быть все различными. Если это не так, действие y на b не определено. В противном случае выполнение y эквивалентно выполнению оператора $v_1 := \mathrm{val}_{\eta_1}(t_1, b), ..., v_m := \mathrm{val}_{\eta_m}(t_m, b)$, где $\eta_1, ..., \eta_m$ — типы термов $t_1, ..., t_m$ соответственно.

Рассмотрим пример. Пусть в множестве Ξ для каждого типа ξ и набора из $2k$ целых чисел $\mu = (m_1, n_1, ..., m_k, n_k)$ содержится тип $\xi' = (\mu, \xi)$. Значениями переменных типа ξ' являются частичные отображения $f \subset Z^k \to D_\xi$ целочисленной решетки Z^k в D_ξ, области определения которых содержатся в прямоугольном параллелепипеде $[m_1 : n_1, ..., m_k : n_k]$. Значения типа ξ' называются *прямоугольными массивами* элементов типа ξ с границами, определенными набором μ граничных пар.

Если $v \in V_{\xi'}$ есть переменная типа ξ', то в множестве V_ξ должны содержаться переменные вида $v(i_1, ..., i_k)$, где $(i_1, ..., i_k)$ — произвольный набор целых чисел, удовлетворяющих условию $m_1 \leqslant i_1 \leqslant n_1, ..., m_k \leqslant i_k \leqslant n_k$. Нужны еще вспомогательные типы $\mu_1, ..., \mu_k$, областями значений которых являются целочисленные интервалы $[m_1 : n_1], ..., [m_k : n_k]$ и операция $\omega(w, i_1, ..., i_k)$ типа $(\alpha^{-1}\xi', \mu_1, ..., \mu_k, \alpha^{-1}\xi)$. Результат применения этой операции к термам $w, t_1, ..., t_k$ записывается в виде $w(t_1, ..., t_k)$, т.е. операция имеет синтаксическую схему $() ((), ..., ())$. Таким образом, при выполнении оператора присваивания $v(s_1, ..., s_k) := v(t_1, ..., t_k)$ для левой части будет определена переменная $v(i_1, ..., i_k)$, где $i_1, ..., i_k$ являются значениями термов $s_1, ..., s_k$, а для правой части — переменная $v(j_1, ..., j_k)$, где $j_1, ..., j_k$ — значения термов $t_1, ..., t_k$.

Возможна также другая точка зрения. Можно считать, что все типы μ_1, ..., μ_k совпадают в типом целых чисел. Тогда операция доступа к элементу массива $v(i_1, ..., i_k)$ либо должна рассматриваться как частичная, либо для набора значений индексов i_1, ..., i_k, которые не удовлетворяют соответствующим неравенствам, должна давать некоторое специальное значение. Если это значение появляется в левой части оператора присваивания, результат его выполнения объявляется неопределенным. Память с массивами является функциональной. Значения переменных типа "массив" однозначно определяется значениями переменных, определяющих элементы массивов (переменных с индексами), и в конечном счете значениями переменных, типы которых не являются массивами.

Для представления схем программ над простой бестиповой памятью в текстовом виде можно пользоваться языком АО, в котором сигнатура алгебры данных и алгоритмический базис представлены синтаксисом условий, выражений и операторов присваивания. Для типизированной памяти определение выражений (термов) должно учитывать сигнатуру типов. Для каждого типа ξ должен быть определен синтаксис выражений типа ξ. Для обозначения переменных можно использовать идентификаторы, т.е. последовательности букв и цифр, начинающиеся с буквы. Для того чтобы одни и те же идентификаторы использовать для обозначения переменных различных типов в разных программах, описания типов переменных должны быть включены в программу. Эти описания определяют структуру памяти, используемой в данной программе. Язык, в котором используется описание памяти, назовем А1. Его синтаксис имеет следующее представление:

⟨А1–программа⟩::=НАЧАЛО⟨описание памяти⟩⟨оператор⟩КОНЕЦ.
⟨описание памяти⟩::=⟨описание переменной⟩|⟨описание переменной⟩; ⟨описание памяти⟩
⟨описание переменной⟩::=ИМЯ⟨идентификатор⟩:⟨тип⟩|
ИМЕНА⟨список идентификаторов, разделенных запятыми⟩:⟨тип⟩

Этот базовый синтаксис может быть изменен или расширен средствами, которые сводятся к базовому. Понятие ⟨тип⟩ определяется для каждой конкретной алгебры своим синтаксисом. Например, синтаксис массивов может быть определен следующим образом.

⟨тип⟩::=⟨простой тип⟩|⟨массив⟩
⟨массив⟩::= МАССИВ (⟨список граничных пар⟩)⟨тип⟩
⟨список граничных пар⟩::=⟨граничная пара⟩|⟨граничная пара⟩, ⟨список граничных пар⟩
⟨граничная пара⟩::=⟨целое⟩:⟨целое⟩

§ 4. Алгебра алгоритмов

Алгебра алгоритмов, или *алгебра Глушкова,* так же как и алгебра отношений (многозначных преобразований), дает возможность явным образом выразить преобразование, выполняемое детерминированной схемой программы, через базовые условия и операторы. Однако в отличие от алгебры отношений здесь, по существу, используется детерминизм и, с другой сто-

роны, выражения алгебры алгоритмов достаточно близки к программам соответствующих алгоритмических языков и могут использоваться как инструмент программирования. Использование соотношений алгебры алгоритмов дает возможность выполнять глубокие оптимизирующие преобразования. Поэтому она используется как основная математическая модель при проектировании программ и разработке инструментальных средств проектирования.

Алгебра алгоритмов $\mathbf{A} = (\mathbf{Y}, \mathbf{U})$ является двухосновной алгеброй, компоненты которой называются *алгеброй операторов* **Y** и *алгеброй условий* **U**. Алгебра операторов состоит из частичных преобразований $P \subset B \to B$ множества B состояний информационной среды, а алгебра условий – из частичных предикатов $\alpha \subset B \to \{0, 1\}$, заданных на B. Поэтому алгебра алгоритмов **A** называется также *алгеброй алгоритмов над B*. Операции алгебры алгоритмов делятся на операции алгебры операторов и операции алгебры условий. Первые принимают значения в множестве операторов, вторые – в множестве условий.

Операции алгебры операторов:

1. Произведение операторов PQ – последовательная композиция $(PQ)(b) = P(Q(b))$.

2. Операция α-дизъюнкции $\underset{\alpha}{(P \vee Q)}$, определенная для любого условия $\alpha \in \mathbf{U}$ и операторов P и Q.

$$\underset{\alpha}{(P \vee Q)}(b) = \text{если } \alpha(b) = 1 \text{ то } P(b) \text{ иначе если } \alpha(b) = 0 \text{ то } Q(b) \text{ иначе не определено.}$$

3. Операция α-итерации $\underset{\alpha}{\{P\}}$, определенная для любого условия $\alpha \in \mathbf{U}$ и оператора P. $\underset{\alpha}{\{P\}}(b)$ определено $\Leftrightarrow$ существует $n \geqslant 0$ такое, что $\alpha(bP^n) = 1$, и если $n > 0$, то $\alpha(b) = \alpha(bP) = \ldots = \alpha(bP^{n-1}) = 0$. Если $\underset{\alpha}{\{P\}}(b)$ определено, то $\underset{\alpha}{\{P\}}(b) = bP^n$, где n определено предыдущим предложением.

Очевидно, что для α-итерации имеет место следующее утверждение:

$$\underset{\alpha}{\{P\}}(b) = \text{если } \alpha(b) = 1 \text{ то } b \text{ иначе если } \alpha(b) = 0 \text{ то } \underset{\alpha}{\{P\}}(bP) \text{ иначе не определено.}$$

Это утверждение может быть использовано для вычисления $\underset{\alpha}{\{P\}}(b)$, если оно определено.

Операции алгебры условий:

1. Булевы операции $\alpha \vee \beta$, $\alpha \wedge \beta$, $\overline{\alpha}$. На состоянии b информационной среды условие α может принимать одно из двух значений 0, 1 или быть неопределенным. Условие $\overline{\alpha}(b)$ определено $\Leftrightarrow$ $\alpha(b)$ определено и равно отрицанию α, т.е. $\overline{\alpha}(b) = 1 \Leftrightarrow \alpha(b) = 0$, $\overline{\alpha}(b) = 0 \Leftrightarrow \alpha(b) = 1$. Результат применения дизъюнкции и конъюнкции представлен следующей таблицей,

где H означает, что значение не определено.

$\alpha(b)$	$\beta(b)$	$(\alpha \wedge \beta)(b)$	$(\alpha \vee \beta)(b)$	$\alpha(b)$	$\beta(b)$	$(\alpha \wedge \beta)(b)$	$(\alpha \vee \beta)(b)$
H	H	H	H	1	H	H	1
H	0	0	H	1	0	0	1
H	1	H	1	1	1	1	1
0	H	0	H				
0	0	0	0				
0	1	0	1				

Из этого определения следует, что для всюду определенных условий дизъюнкция, конъюнкция и отрицание совпадают с обычными булевыми операциями. В общем случае операции $\vee$, $\wedge$ и $-$ удовлетворяют всем основным законам булевой алгебры, кроме законов исключенного третьего $\alpha \vee \overline{\alpha} = 1$ и противоречия $\alpha\overline{\alpha} = 0$ (знак $\wedge$, как обычно, опускается). Иными словами, дизъюнкция и конъюнкция ассоциативны, коммутативны, идемпотентны, удовлетворяют законам дистрибутивности и правилу Де-Моргана, а отрицание удовлетворяет закону двойного отрицания $\overline{\overline{\alpha}} = \alpha$.

2. Операция $P\alpha$ умножения оператора P на условие α, называемая также *прогнозированием*. Значение условия $P\alpha$ на состоянии b определяется равенством $(P\alpha)(b) = \alpha(bP)$. Таким образом, условие $P\alpha$ – это "α после P".

В некоторых случаях к алгебре алгоритмов удобно добавлять константы: тождественный оператор ϵ, нигде не определенный оператор ϕ, тождественно истинное условие 1 и тождественно ложное условие 0, рассматривая их как нульарные операции.

Таким образом, алгебра алгоритмов – это множество операторов и множество условий, замкнутые относительно операций алгебры алгоритмов.

Если в алгебре операторов выделено множество базовых операторов Y, а в алгебре условий – множество базовых условий U таким образом, что эти множества порождают всю алгебру алгоритмов, то она называется *U-Y-алгеброй алгоритмов* (над B) и обозначается $\mathbf{A}(U, Y)$. Каждый элемент U-Y-алгебры алгоритмов может быть задан с помощью выражения, построенного из символов, обозначающих элементы множеств U и Y, т.е. из базовых условий и операторов, с помощью операций алгебры алгоритмов. Такие выражения называются *регулярными выражениями*, а соответствующие операторы и условия – *операторами* и *условиями, регулярными относительно базиса* (Y, U). Регулярные выражения алгебры операторов называются также *регулярными программами* или *регулярными U-Y-программами*.

Рассмотрим некоторые полезные соотношения алгебры алгоритмов. Операция умножения операторов ассоциативна, т.е. множество операторов является полугруппой, а ϵ и ϕ играют роль единицы и нуля этой полугруппы соответственно:

$$P(QR) = (PQ)R, \tag{4.1}$$

$$P\epsilon = \epsilon P = P, \tag{4.2}$$

$$P\phi = \phi P = \phi. \tag{4.3}$$

1. Соотношения для α-дизъюнкции:

$$\underset{\alpha}{(}P \vee Q) = \underset{\overline{\alpha}}{(}Q \vee P), \tag{4.4}$$

$$\underset{1}{(}P \vee Q) = P, \tag{4.5}$$

$$\underset{0}{(}P \vee Q) = Q, \tag{4.6}$$

$$\underset{\alpha}{(}P \vee Q) = \underset{\alpha}{(}P \vee \underset{\overline{\alpha}}{(}Q \vee \phi)), \tag{4.7}$$

$$\underset{\alpha}{(}P \vee \underset{\beta}{(}Q \vee R)) = \underset{\alpha}{(}P \vee \underset{\overline{\alpha}\beta}{(}Q \vee R)). \tag{4.8}$$

2. Аналоги законов ассоциативности для α-дизъюнкции:

$$\underset{\alpha}{(}P \vee \underset{\beta}{(}Q \vee R)) = \underset{\alpha \vee \overline{\alpha}\beta}{(}\underset{\alpha}{(}P \vee Q) \vee R), \tag{4.9}$$

$$\underset{\alpha\,\beta}{(}\underset{\alpha}{(}P \vee Q) \vee R) = \underset{\alpha\beta}{(}P \vee \underset{\alpha\overline{\beta}}{(}Q \vee R)). \tag{4.10}$$

Если α и β – всюду определенные условия, то соотношения ассоциативности могут быть записаны проще:

$$\underset{\alpha}{(}P \vee \underset{\beta}{(}Q \vee R)) = \underset{\alpha \vee \beta}{(}\underset{\alpha}{(}P \vee Q) \vee R), \tag{4.11}$$

$$\underset{\alpha\,\beta}{(}\underset{\alpha}{(}P \vee Q) \vee R) = \underset{\alpha\beta}{(}P \vee \underset{\overline{\beta}}{(}Q \vee R)). \tag{4.12}$$

3. Аналог идемпотентности для α-дизъюнкции:

$$\underset{\alpha}{(}P \vee P) = \underset{\alpha \vee \overline{\alpha}}{(}P \vee \phi). \tag{4.13}$$

Если α всюду определено, то

$$\underset{\alpha}{(}P \vee P) = P. \tag{4.14}$$

4. Соотношения дистрибутивности:

$$\underset{\alpha}{(}P \vee Q)R = \underset{\alpha}{(}PR \vee QR), \tag{4.15}$$

$$R\underset{\alpha}{(}P \vee Q) = \underset{R\alpha}{(}RP \vee RQ). \tag{4.16}$$

С помощью соотношений ассоциативности и дистрибутивности любая регулярная программа, не содержащая итераций, может быть приведена к виду $\underset{\alpha_1}{(}P_1 \vee \underset{\alpha_2}{(}P_2 \vee \ldots \underset{\alpha_n}{(}P_n \vee Q) \ldots))$, где $P_1, \ldots, P_n, Q$ – произведения базовых операторов, $\alpha_1, \ldots, \alpha_n$ – произвольные условия. При этом, если базовые условия и базовые операторы всюду определены, то можно сделать так, что $\alpha_i \wedge \alpha_j = 0$ при $i \neq j$.

5. Соотношения для умножения оператора на условие:

$$P(\alpha \vee \beta) = P\alpha \vee P\beta, \tag{4.17}$$

$$P(\alpha \wedge \beta) = P\alpha \wedge P\beta, \tag{4.18}$$

$$\overline{P\alpha} = P\overline{\alpha}, \tag{4.19}$$

$$(P \underset{\alpha}{\vee} Q)\beta = \alpha \wedge (P\beta) \vee \overline{\alpha} \wedge (Q\beta), \tag{4.20}$$

$$(PQ)\alpha = P(Q\beta). \tag{4.21}$$

6. Соотношения для итерации:

$$\{P\}_{\alpha} = (\epsilon \vee P\{P\}_{\alpha}) = (\epsilon \vee \{P\}_{P\alpha} P). \tag{4.22}$$

Все рассмотренные соотношения являются тождествами алгебры алгоритмов, т.е. они выполняются при любых значениях входящих в них операторов и условий, заданных на любой информационной среде B. Доказательства всех соотношений проводятся непосредственным обращением к определениям операций алгебры алгоритмов. При этом, поскольку речь идет о равенствах частичных функций, каждое доказательство состоит из двух частей: если левая часть определена, то правая тоже определена и они равны и, наоборот, если правая часть определена, то определена и левая и они равны.

Докажем, например, равенство (4.20). Выберем произвольное состояние b информационной среды. Левую часть равенства обозначим через u, правую — через v. Пусть $u(b)$ определено. Тогда $\alpha(b)$ также определено. Рассмотрим два случая: $\alpha(b) = 1 \Rightarrow u(b) = \beta(b(P \underset{\alpha}{\vee} Q)) = \beta(bP) = v(b)$, $\alpha(b) = 0 \Rightarrow u(b) = \beta(bQ) = v(b)$. Пусть теперь $v(b)$ определено. Тогда $\alpha(b)$ снова определено, и рассмотрение тех же двух случаев дает равенство $u(b) = v(b)$.

Рассмотрим теперь равенства (4.22). Первое из них называется *левым развертыванием* итерации, второе — *правым развертыванием*. Докажем второе равенство. Обозначим $A = \{P\}_{\alpha}$, $B = (\epsilon \vee \{P\}_{P\alpha} P)$. Пусть $A(b)$ определено. Тогда существует $n \geqslant 0$ такое, что $\alpha(bP^n) = 1, A(b) = bP^n$, а если $n \geqslant 1$, то $\alpha(b) = \alpha(bP) = \ldots = \alpha(bP^{n-1}) = 0$. Если $n = 0$, то $\alpha(b) = 1$ и $B(b) = b = A(b)$. Если же $n > 0$, то $B(b) = (\{P\}_{P\alpha} P)(b) = P(\{P\}_{P\alpha}(b))$. Снова получаем два случая: $n = 1 \Rightarrow (P\alpha)(b) = \alpha(bP) = 1 \Rightarrow B(b) = bP = A(b)$ и $n > 1 \Rightarrow (P\alpha)(b) = \alpha(bP) = (P\alpha)(bP^2) = \alpha(bP^3) = \ldots = (P\alpha) \times (bP^{n-2}) = \alpha(bP^{n-1}) = 0$, $(P\alpha)(bP^{n-1}) = \alpha(bP^n) = 1 \Rightarrow B(b) = P(bP^{n-1}) = bP^n = A(b)$. Обратное утверждение доказывается аналогично. Предлагаем читателю провести подробные доказательства для тех утверждений, которые на первый взгляд не представляются очевидными.

Операции алгебры операторов легко выражаются через операции алгебры многозначных преобразований множества B. Умножение просто совпадает на алгебре операторов с умножением многозначных преобразований, а α-дизъюнкция и α-итерация выражаются следующими

формулами:

$$(\underset{\alpha}{P} \vee Q) = \alpha/P \vee \overline{\alpha}/Q, \tag{4.23}$$

$$\{\underset{\alpha}{P}\} = (\overline{\alpha}/P)^*(\alpha/\epsilon). \tag{4.24}$$

Таким образом, U-Y-алгебра операторов вкладывается в алгебру многозначных преобразований, порожденную множеством $\mathrm{U}(U, Y)/\epsilon \cup Y$, где $\mathrm{U}(U, Y)/\epsilon$ — множество всех фильтров тождественного оператора ϵ по условиям U-Y-алгебры условий. Операция взятия фильтра выражается через операции алгебры алгоритмов:

$$\alpha/\epsilon = \{\underset{\alpha}{\epsilon}\},$$

$$\alpha/P = \{\underset{\alpha}{\epsilon}\}P.$$

Поэтому соотношения (4.23) и (4.24) можно рассматривать как соотношения в алгебре многозначных преобразований, расширенной добавлением операций α-дизъюнкции и α-итерации, или в алгебре операторов, расширенной добавлением недетерминированной дизъюнкции и недетерминированной итерации. Такой переход к расширенной алгебре алгоритмов иногда удобно делать для поиска и вывода новых тождеств или конкретных соотношений. Хотя условия сами по себе не погружаются в алгебру многозначных преобразований, но с фильтрами связаны еще несколько полезных соотношений:

$$(\alpha \vee \beta)/\epsilon = \alpha/\epsilon \vee \beta/\epsilon, \tag{4.25}$$

$$(\alpha \wedge \beta)/\epsilon = (\alpha/\epsilon)(\beta/\epsilon), \tag{4.26}$$

$$((P\alpha)/\epsilon)P = P(\alpha/\epsilon) = (P\alpha)/P. \tag{4.27}$$

Расширенная алгебра алгоритмов может быть практически полезной в ряде случаев. Во-первых, некоторые тождества и соотношения алгебры алгоритмов удобно доказывать, переходя сначала к недетерминированному представлению, а затем, выполнив преобразования в алгебре многозначных преобразований, возвращаясь к детерминированному случаю. Таким образом, можно, в частности, взять на вооружение все тождества алгебры языков, которые, как было показано в § 9 гл. 1, являются тождествами и алгебры отношений (многозначных преобразований). С другой стороны, недетерминированность удобно использовать при описании моделей проектируемых систем, предполагая, что детерминизация произойдет на этапе построения реализации путем выбора конкретных процессов вычислений из множества процессов, приводящих к одному и тому же результату. Наконец, в некоторых случаях недетерминированность указывает на возможность параллельных вычислений. Например, соотношение (4.25) показывает, что дизъюнкцию можно вычислять путем параллельного выполнения операторов α/ϵ и β/ϵ. Важность такого параллелизма станет особенно наглядной, если предположить, что, например, $\alpha = \{\underset{\alpha'}{P}\}\alpha''$, а $\beta = \{\underset{\beta'}{Q}\}\beta''$. Не следует, однако, думать, что все вопросы параллельных вычислений сводятся к недетерминизму. В дальнейшем у нас будет повод поговорить об этом более подробно.

Поскольку алгебра операторов содержится в алгебре преобразований, естественно ожидать, что на нее можно перенести технику решения уравнений. Действительно, рассмотрим уравнение вида

$$X = \underset{\alpha}{(}PX \vee Q), \tag{4.28}$$

в котором X — неизвестный оператор. Такое уравнение назовем *леволинейным каноническим уравнением с одним неизвестным.* Решением уравнения (4.28) может служить регулярная программа

$$X_0 = \underset{\bar{\alpha}}{\{}\ P\}Q. \tag{4.29}$$

Действительно, применяя левое развертывание итерации (4.22), правую дистрибутивность (4.15) и перестановочность для α-дизъюнкции (4.4), получим

$$X_0 = \underset{\bar{\alpha}}{\{}\ P\}Q = \underset{\bar{\alpha}}{(}\epsilon \vee P \underset{\bar{\alpha}}{\{}\ P\})Q = \underset{\bar{\alpha}}{(}Q \vee P\underset{\bar{\alpha}}{\{}P\}Q) =$$

$$= \underset{\alpha}{(}P(\underset{\bar{\alpha}}{\{}\ P\}Q) \vee Q) = \underset{\alpha}{(}PX_0 \vee Q);$$

следовательно, регулярная программа (4.29), а точнее, представляемый ею оператор, удовлетворяет уравнению (4.28). Решение (4.29) является наименьшим. Действительно, пусть X_1 — другое решение. Включение $X_0 \subset X_1$ означает, что если bX_0 определено, то и bX_1 определено и $bX_0 = bX_1$. Пусть bX_0 определено. Тогда $bX_0 = bP^nQ$, причем $\alpha(b) = \ldots = \alpha(bP^{n-1}) = 1$, $\alpha(bP^n) = 0$. С другой стороны, $X_1 = \underset{\alpha}{(}PX_1 \vee Q) =$

$= \underset{\alpha}{(}P\underset{\alpha}{(}PX_1 \vee Q) \vee Q) = \underset{\alpha}{(}\underset{P\alpha}{(}P^2X_1 \vee Q) \vee Q) = \ldots = \underset{\alpha}{(}\underset{P\alpha}{(}\ldots \underset{P^{n-1}\alpha}{(}P^n\underset{P^n\alpha}{(}P^{n+1}X_1 \vee$
$\vee\ Q)\ \vee Q) \vee \ldots \vee Q)\ \vee Q)$, поэтому $bX_1 = bP^n\underset{P^n\alpha}{(}P^{n+1}X_1 \vee Q) =$
$= bP^nQ = bX_0$.

Перейдем теперь к рассмотрению системы линейных уравнений вида

$$X_i = F_i(X_1, \ldots, X_n), \quad i = 1, \ldots, n, \tag{4.30}$$

где X_i — неизвестные операторы, F_i — леволинейные регулярные выражения, т.е. выражения вида

$$F_i(X_1, \ldots, X_n) = \underset{\alpha_1}{(}P_1X_1 \vee \underset{\alpha_2}{(}P_2X_2 \vee \ldots \underset{\alpha_n}{(}P_nX_n \vee P_{n+1}) \ldots)),$$

$P_1, \ldots, P_{n+1}$ — известные операторы (коэффициенты и свободный член), $\alpha_1, \ldots, \alpha_n$ — известные условия. Систему (4.30) называют *канонической системой линейных уравнений.* Ее можно решать методом исключения неизвестных. Если правая часть первого уравнения не зависит от X_1, т.е. коэффициент при X_1 равен ϕ, то X_1 можно исключить, если же этот коэффициент не пуст, то

$$X_1 = \underset{\alpha_1}{(}PX_1 \vee F_1'(X_2, \ldots, X_n)) = \underset{\bar{\alpha}_1}{\{}\ P\}F_1'(X_2, \ldots, X_n)$$

и снова X_1 исключается. После исключения X_1 неизвестных стало на одно

меньше, а система снова может быть приведена к каноническому виду путем применения законов левой дистрибутивности, раскрытия и перестановки скобок с помощью левой дистрибутивности и ассоциативности, а также вынесения неизвестных за скобки с помощью правой дистрибутивности. В результате решение будет найдено в виде выражения, регулярного относительно коэффициентов и свободных членов. Поскольку на каждом шаге получается наименьшее решение, имеет место следующая теорема.

Т е о р е м а 4.1. *Каноническая система линейных уравнений в алгебре алгоритмов имеет наименьшее решение, регулярное относительно коэффициентов и свободных членов.*

С помощью теоремы 4.1 можно доказать важную теорему Глушкова о регуляризации схем программ.

Т е о р е м а 4.2. *Если A – детерминированная $U-Y$-схема программы, интерпретированная на множестве B, а базовые условия всюду определены, то f_A регулярно относительно U и Y.*

Действительно, пусть a – произвольное состояние, отличное от заключительного, $a \xrightarrow{u_1/y_1} a_1, \ldots, a \xrightarrow{u_m/y_m} a_m$ – все переходы, которые начинаются в этом состоянии. Тогда преобразование f_a – оператор, который выполняется схемой при настройке $(a, a^{(1)})$, – удовлетворяет соотношению

$$f_a = \underset{u_1}{(} y_1 f_{a_1} \vee \underset{u_2}{(} y_2 f_{a_2} \vee \ldots \vee \underset{u_m}{(} y_m f_{a_m} \vee \phi) \ldots)),$$

справедливость которого вытекает из независимости условий переходов (детерминированность). Вместе с соотношением

$$f_{a^{(1)}} = \epsilon$$

указанные соотношения образуют каноническую систему линейных уравнений, для которой $(f_a)_{a \in A}$ является наименьшим решением, что было доказано в § 1. Поэтому все f_a регулярны. Поскольку $f_A = f_{a^{(0)}}$, теорема доказана.

Операции алгебры операторов α-дизъюнкция и α-итерация естественным образом выражаются в текстовом виде. $\underset{\alpha}{(P \vee Q)}$ – это то же самое, что и

ЕСЛИ α ТО P ИНАЧЕ QKE.

$\underset{\alpha}{\{P\}}$ обычно выражается в виде оператора цикла:

ПОКА $\neg\alpha$ ВЫПОЛНЯТЬ P КОНЕЦ ЦИКЛА.

В практических языках программирования помимо цикла по условию применяются и другие виды операторов цикла. Например,

ДЛЯ $I := I_0$ ШАГ H ДО I_1 ВЫП P КЦ

или

ДЛЯ ВСЕХ $x \in X$ ВЫП P КЦ.

Семантика таких операторов естественным образом может быть выражена через цикл по условию. Например, первый оператор при $H > 0$ может быть определен как эквивалентный оператору

НАЧАЛО

$I := I_0$;

ПОКА $I \leqslant I_1$ ВЫП

P;

$I := I + H$

КОНЕЦ ЦИКЛА

КОНЕЦ.

Второй оператор можно определить так:

ПОКА $X \neq \phi$ ВЫПОЛНИТЬ

ВЗЯТЬ x ИЗ X;

P

КОНЕЦ ЦИКЛА;

Возможно более ограниченное толкование операторов ДЛЯ $I := \ldots$, не допускающее изменение оператором P параметров цикла. Для первого оператора такое толкование может быть выражено с помощью оператора;

НАЧАЛО

$I' := I_0$, $I'_1 := I_1$, $H' := H$;

ПОКА $I' \leqslant I'_1$ ВЫПОЛНЯТЬ

P; $I' := I' + H'$; $I := I'$

КОНЕЦ ЦИКЛА

КОНЕЦ

Для второго оператора возможно такое толкование:

НАЧАЛО

$X' := X$;

ПОКА $X' \neq \phi$ ВЫПОЛНИТЬ

ВЗЯТЬ x ИЗ X'; P

КОНЕЦ ЦИКЛА

КОНЕЦ

Ограниченное толкование операторов цикла более естественно для структурного программирования. Если оператор P не содержит изменения параметров, то рассмотренные виды циклов можно также толковать просто как произведения:

$$\prod_{I=0}^{[I_1/H]} P(I_0 + I * H) = P(I_0)P(I_0 + H) \ldots P(I_0 + H * [I_1/H])$$

и

$$\prod_{x \in X} P(x) = P(x_1) \ldots P(x_n), \text{ если } X = \{x_1, \ldots, x_n\}.$$

Всякую регулярную U-Y-программу над B, не использующую операцию прогнозирования, можно рассматривать как U-Y-схему программы, интерпретированную на B. Действительно, операции умножения и α-дизъюнкции имеют соответствующие аналоги в алгебре схем программ, а α-итерация оператора P может быть представлена текстом:

НАЧАЛО МЕТКИ L, M;

M: ЕСЛИ α ТО НА L;

P;

НА М;

L:

КОНЕЦ.

Теорема о регуляризации показывает, что всякую U-Y-схему программы можно представить в виде регулярной U-Y-программы. Если множество U базовых условий замкнуто относительно операции прогнозирования, то, переходя от регулярной программы к U-Y-схеме, получаем возможность устранить произвольные переходы, заменив их циклами по условию, регулярно вложенными друг в друга. Такие схемы программ будем называть *структурными*, а представляющие их тексты в языке АО, расширенном введением операторов цикла, – *структурными программами*. Язык АО без оператора перехода, но с операторами цикла будем называть *структурным вариантом* языка АО. Этот язык обладает тем важным преимуществом, что его функциональная семантика может быть непосредственно выражена путем сопоставления каждой структурной программе P вычисляемого этой программой преобразования $\overline{\sigma}(P)$ без перехода к схеме программы. При этом преобразование $\overline{\sigma}(P)$ выражается через преобразования алгоритмического базиса с помощью операций алгебры алгоритмов.

Возможность устранения операции прогнозирования, которая появляется в доказательстве теоремы о регуляризации, может быть проиллюстрирована следующим примером. Пусть $\alpha = (a \leqslant b)$, $y = (a := 2b - a)$. Тогда $y\alpha = (2b - a \leqslant b) = (b \leqslant a)$. В общем случае элиминировать прогнозирование можно, расширив информационную среду B. Действительно, рассмотрим множество B^* всех конечных последовательностей, составленных из элементов множества B. Отождествляя одноэлементные последовательности с элементами множества B, получим $B \subset B^*$. Распространим действие операторов и условий на B^*, полагая $(b_1, \ldots, b_n)y = (b_1, \ldots, b_n y)$, $\alpha(b_1, \ldots, b_n) = \alpha(b_n)$. Введем на множестве B^* два новых оператора $\uparrow$ и $\downarrow$. Действие этих операторов определяется следующими правилами:

$$(b_1, \ldots, b_m)\uparrow = (b_1, \ldots, b_m, b_m),$$

$$(b_1, \ldots, b_m)\downarrow = (b_1, \ldots, b_{m-1}).$$

Имеет место следующая теорема синтеза схемы программы по регулярному выражению.

Т е о р е м а 4.3. *Для любой регулярной U-Y-программы P над B существует U-Y* $\cup \{\uparrow, \downarrow\}$*-схема программы A над* B^* *такая, что* $f_A(b) = bP$ *для любых* $b \in B$.

Для доказательства достаточно заметить, что

$$(\underset{P\alpha}{} Q \vee R) = \uparrow P(\underset{\alpha}{} \downarrow Q \vee \downarrow R),$$

$$\{\underset{P\alpha}{} Q\} = \uparrow P \{\underset{\alpha}{} \downarrow Q \uparrow P\} \downarrow.$$

Операторы $\uparrow$ и $\downarrow$ дают, таким образом, возможность проверки условия $P\alpha$ с помощью одного базового условия α. Оператор $\uparrow$ позволяет запомнить текущее состояние информационной среды, α проверяется после выполнения P, а оператор $\downarrow$ восстанавливает состояние информационной среды, измененное оператором P.

§ 5. Логика алгоритмов

Первая компонента Y алгебры алгоритмов (Y, U) над B, как мы уже видели, может быть использована для построения языка представления алгоритмов преобразования множества B. Это язык регулярных программ или структурный вариант языка АО, которые получаются, если определить язык для записи элементов алгоритмического базиса (U, Y). Алгебра условий, в свою очередь может быть использована для построения языка представления свойств алгоритмов, действующих на информационной среде B. Такой язык необходим для развития общих методов доказательства утверждений об алгоритмах и программах, т.е. логики алгоритмов.

Язык логики алгоритмов прежде всего должен содержать средства для представления свойств произвольного текущего состояния b информационной среды B. Простейшими, или базисными, высказываниями о состояниях информационной среды могут служить элементы базиса U. Если α – базисное всюду определенное условие, то его можно использовать как высказывание, которое истинно на состоянии b, если $\alpha(b) = 1$, и ложно, если $\alpha(b) = 0$. В случае частично определенных условий α можно рассматривать высказывания $\alpha = 1$, $\alpha = 0$ и $\alpha = H$, где H обозначает неопределенное значение. Высказывание $\alpha = 1$, например, истинно на состоянии b, если $\alpha(b) = 1$, и ложно, если $\alpha(b) = 0$ или не определено.

Более сложные высказывания образуются из базисных с применением обычных пропозициональных связок. Таким образом, получаем элементарные высказывания языка логики алгоритмов, которые соответствуют элементарным условиям $\hat{U}$. Базисных условий может быть недостаточно для построения рассуждений о программах. Поэтому элементарные высказывания языка логики алгоритмов строятся, исходя из некоторого расширенного множества базисных высказываний, которое, как правило, включает в себя высказывания, построенные из базисных условий. Если $B \subset \Gamma(V, D)$ есть память над Ω-П-алгеброй D, то язык элементарных высказываний фактически совпадает с языком бескванторных формул исчисления предикатов.

Для построения сложных высказываний кроме пропозициональных связок могут теперь применяться также и кванторы по переменным, про-

бегающим область D. При этом для связывания переменных кванторами следует использовать расширенный алфавит переменных, не допуская связывания кванторами переменных из множества V, которые фактически играют роль констант в элементарных высказываниях. Действительно, каждое такое высказывание относится к произвольному, но фиксированному состоянию памяти b, а переменные из множества V представляют свои значения при этом состоянии памяти. Для памяти над областью D расширение языка базисных высказываний может быть выполнено путем расширения сигнатур Ω и Π, а также добавлением констант из области D. В случае многоосновных алгебр данных может быть одновременно расширена и сигнатура Ξ типов данных.

Итак, предположим, что зафиксирован некоторый язык элементарных высказываний логики алгоритмов, предложения которого могут быть истинными или ложными в зависимости от выбранного состояния информационной среды. Элементарные высказывания могут также зависеть от параметров, пробегающих те или иные области значений, и тогда истинность высказываний зависит не только от состояния информационной среды, но и от значений параметров. Следующий уровень высказываний может быть построен с помощью идеи прогнозирования. Если P – программа в некотором алгоритмическом языке, интерпретированном на информационной среде B, а α – элементарное высказывание, то можно рассматривать высказывание $(P\alpha) = 1$, которое истинно на состоянии информационной среды b в том и только в том случае, когда α истинно на состоянии bP. Это высказывание записывают также в виде $[P]\alpha$.

Заметим, что из истинности высказывания $[P]\alpha$ на b следует, что действие программы P на b определено. Высказывание $[P]\alpha$ можно читать так: "P останавливается и α после P". Иногда полезно рассматривать более слабое высказывание, также связанное с прогнозированием: $\langle P\rangle\alpha$. Это высказывание имеет следующий смысл: $\langle P\rangle\alpha$ истинно на состоянии b тогда и только тогда, когда из того, что bP определено, следует, что α истинно на bP. Читается это высказывание так: "если P останавливается, то α после P". Таким образом, помимо операции прогнозирования $P\alpha$ имеем два способа образования прогнозирующих высказываний логики алгоритмов: строгое прогнозирование $[P]\alpha$ и слабое прогнозирование $\langle P\rangle\alpha$.

Связь строгого и слабого прогнозирования с операцией прогнозирования может быть выражена следующими эквивалентностями:

$$[P]\alpha \iff P\alpha = 1,$$

$$\langle P\rangle\alpha \iff P\alpha \neq 0.$$

Применяя к прогнозирующим и элементарным высказывания пропозициональные связки, а также связывая кванторами параметры, от которых могут зависеть элементарные высказывания, получим новый уровень высказываний языка логики алгоритмов. Эти высказывания можно использовать для образования новых высказываний прогнозирующего типа и т.д. Описанный путь построения языка логики алгоритмов можно было бы описать более точно, зафиксировав язык для задания алгоритмического базиса. Мы этого здесь делать не будем, а ограничимся лишь рассмотре-

нием некоторых специальных видов высказываний, играющих важную роль в практических задачах.

Высказывание языка логики алгоритмов называется *тождественно истинным* на B, если оно истинно при любых состояниях информационной среды B и при любых допустимых наборах значений свободных параметров, от которых зависит это высказывание. Тождественно истинные высказывания можно рассматривать как выражение свойств программ, которые в них входят. Язык логики алгоритмов можно строить и независимо от конкретной интерпретации, рассматривая только сигнатуру абстрактного алгоритмического базиса или алгоритмического базиса над памятью (возможно, с необходимыми расширениями). В таком случае можно говорить о тождественно истинных высказываниях как о высказываниях, истинных для любых состояний информационной среды, допустимых при произвольных интерпретациях алгоритмического базиса.

На практике удобно фиксировать лишь некоторые свойства рассматриваемой интерпретации и говорить о тождественной истинности высказываний логики алгоритмов относительно класса интерпретаций, обладающих заданными свойствами.

Рассмотрим высказывание вида

$$\alpha \Rightarrow [P]\,\beta, \tag{5.1}$$

где α и β – элементарные высказывания (не зависящие от программ); P – программа, интерпретированная на B; α и β могут зависеть от параметров. Высказывания вида (5.1) употребляются для определения понятия правильности программы относительно требований, которые к ней предъявляются. Условие α называется *начальным* условием, β – *заключительным.* Условия α и β определяют требование к программе P, выраженное высказыванием (5.1). Это требование состоит в следующем.

Каковы бы ни были исходные данные, представленные начальным состоянием информационной среды b, если только выполняется начальное условие, то программа P останавливается и на заключительном состоянии информационной среды bP выполняется условие β. Иначе говоря, требование состоит в том, что высказывание (5.1) должно быть тождественно истинным. Программа P, удовлетворяющая условию (5.1), называется также *правильной* или *корректной* относительно условий α и β.

В некоторых случаях вместо условия (5.1) рассматривают более слабое условие

$$\alpha \Rightarrow \langle P \rangle\,\beta, \tag{5.2}$$

которое называется *свойством частичной корректности* программы P относительно условий α и β. Для того чтобы программа была корректной, необходимо и достаточно, чтобы она удовлетворяла двум условиям – условию частичной корректности (5.2) и условию завершимости:

$$\alpha \Rightarrow [P]\,1. \tag{5.3}$$

Разделение условия полной корректности на частичную корректность и завершимость оправдывается тем, что методы доказательства этих двух свойств различны.

Рассмотрим пример. Пусть R – следующая программа:

НАЧАЛО

$x := a,\ y := b;$

ПОКА $y - x > \epsilon$ ВЫПОЛНЯТЬ

$z := (x + y)/2;$

$w := f(z);$

ЕСЛИ $w < 0$ ТО $x := z$ ИНАЧЕ

ЕСЛИ $w > 0$ ТО $y := z$

ИНАЧЕ $x := z,\ y := z$ КЕ

КЦ;

$z := (x + y)/2.$

КОНЕЦ.

Программа P ищет приближенное значение одного из корней уравнения $f(x) = 0$ на интервале $[a, b]$. Предполагается, что числовая функция f определена и непрерывна на интервале $[a, b]$, а на его концах принимает значения разных знаков: $f(a) < 0$, $f(b) > 0$. Кроме того, $\epsilon > 0$. Эти предположения составляют начальное условие α. Заключительное условие β формулируется следующим образом: существует c такое, что $a < c < b$, $f(c) = 0$ и $|c - z| < \epsilon$, т.е. z есть приближенное состояние одного из корней уравнения $f(x) = 0$. Докажем корректность программы P относительно условий (α, β).

Программа R является регулярной программой и имеет вид

$$R = P_1 \underset{\gamma}{\{Q\}} P_2.$$

Здесь $\gamma \Longleftrightarrow y - x \leqslant \epsilon$, смысл операторов P_1, P_2 и Q очевиден. Обозначим через α' условие $a \leqslant x \leqslant y \leqslant b$, $f(x) \leqslant 0$, $f(y) \geqslant 0$. Рассматривая работу операторов P_1, P_2 и Q, нетрудно убедиться в справедливости следующих условий:

$$\alpha \Rightarrow \langle P_1 \rangle \alpha',$$

$$\alpha' \Rightarrow \langle Q \rangle \alpha',$$

$$\alpha' \Rightarrow \langle P_2 \rangle \alpha'.$$

Из второго условия следует также, что

$$\alpha' \Rightarrow \langle \underset{\gamma}{\{Q\}} \rangle \alpha'.$$

Кроме того, очевидно, что $\langle \underset{\gamma}{\{Q\}} \rangle \gamma$ и $\langle P_2 \rangle \alpha''$, где $\alpha'' \Longleftrightarrow x \leqslant z \leqslant y$. Поэтому имеет место условие

$$\alpha \Rightarrow \langle R \rangle \gamma \wedge \alpha' \wedge \alpha'',$$

откуда вытекает частичная корректность программы P. Действительно, $\gamma \wedge \alpha' \wedge \alpha'' \Rightarrow \beta$ в силу непрерывности функции f.

Для доказательства завершимости программы R достаточно заметить, что после каждого выполнения оператора Q, который повторяется в цикле, разность $y - x$ уменьшается вдвое, и поэтому через конечное число шагов она станет меньше ϵ, что приведет к завершению цикла $\{Q\}_{\gamma}$. Таким образом, программа R корректна относительно условий (α, β).

Доказательство частичной корректности программы P является частным случаем общего метода доказательства частичной корректности, который носит название *метод Хоара.* Идея этого метода состоит в сведении доказательства утверждения типа (5.2) к доказательству некоторого количества элементарных утверждений. При этом предполагается, что программа P представлена как регулярная программа и не содержит операции прогнозирования в условиях.

Сведение осуществляется по правилам, соответствующим трем основным операциям алгебры операторов и вытекающим из следующих утверждений:

$$\frac{\alpha \Rightarrow \langle P \rangle \gamma,\ \gamma \Rightarrow \langle Q \rangle \beta}{\alpha \Rightarrow \langle PQ \rangle \beta}\ ; \tag{5.4}$$

$$\frac{\alpha \wedge u \Rightarrow \langle P \rangle \beta,\ \alpha \wedge \bar{u} \Rightarrow \langle Q \rangle \beta}{\alpha \Rightarrow \langle (P \vee Q) \rangle_{u} \beta}\ ; \tag{5.5}$$

$$\frac{\alpha \Rightarrow \delta,\ \delta \wedge \bar{u} \Rightarrow \langle P \rangle \delta,\ \delta \wedge u \Rightarrow \beta}{\alpha \Rightarrow \langle \{P\} \rangle_{u} \beta}\ . \tag{5.6}$$

Утверждения (5.4)–(5.6) представлены в форме правил вывода логики алгоритмов, и каждое из них означает, что если все высказывания, написанные над чертой, истинны, то истинно также и высказывание, написанное под чертой. Правила сведения противоположны правилам вывода и предписывают переходить от следствия к посылкам.

Правило 1. Для того чтобы доказать утверждение вида $\alpha \Rightarrow \langle PQ \rangle \beta$, достаточно найти элементарное высказывание γ такое, что $\alpha \Rightarrow \langle P \rangle \gamma$ и $\gamma \Rightarrow \langle Q \rangle \beta$.

Правило 2. Для того чтобы доказать утверждение вида $\alpha \Rightarrow \langle (P \vee Q) \rangle_{u} \beta$, достаточно доказать посылки утверждения (5.5).

Правило 3. Для того чтобы доказать утверждение $\alpha \Rightarrow \langle \{P\} \rangle_{u} \beta$, достаточно найти условие δ, для которого можно доказать посылки утверждения (5.6).

Условие δ, которое фигурирует в правиле 3, называется *инвариантом цикла.*

Докажем утверждения (5.4)–(5.6).

1. Пусть верны посылки правила (5.4). Рассмотрим произвольное состояние b информационной среды. Предположим, что $\alpha(b) = 1$ и bPQ определено. Тогда bP определено, и $\gamma(bP) = 1$. Поскольку $(bP)Q$ определено, то $\beta(bPQ) = 1$. Доказано.

2. Утверждение (5.5) доказывается аналогично путем рассмотрения двух случаев: $u(b) = 1$ и $u(b) = 0$.

3. Предположим, что имеют место посылки утверждения (5.6). Рассмотрим состояние b информационной среды такое, что $\alpha(b) = 1$ и $b\{P\}$ определено. Тогда $\delta(b) = 1$. Если $u(b) = 1$, то $b\{P\} = b$ и $\beta(b) = 1$. В противном случае существует такое n, что $u(b) = u(bP) = \ldots = u(bP^{n-1}) = 0$, $u(bP^n) = 1$ и $b\{P\} = bP^n$. Учитывая предположения, имеем $\delta(b) = \ldots = \delta(bP^n) = 1$, откуда $\beta(bP^n) = 1$. Доказано.

Применяя правила сведения 1 – 3 к произвольной регулярной программе P и условиям ее корректности (α, β), получим, что для доказательства утверждения $\alpha \Rightarrow \langle P \rangle \beta$ достаточно доказать некоторое множество элементарных утверждений и утверждений типа $u \Rightarrow \langle y \rangle v$ (где y – базовый оператор, u, v – элементарные высказывания). Если высказывания вида $\langle y \rangle v$ эквивалентны элементарным высказываниям, то доказательство утверждения о частичной корректности полностью сведено к доказательству элементарных утверждений. Такое сведение, в частности, возможно для схем программ над памятью с операторами присваивания в качестве базовых. Действительно, если $y = (v_1 := t_1, \ldots, v_n := t_n)$, то $\langle y \rangle \beta(v_1, \ldots, v_n) \Leftrightarrow \Leftrightarrow \beta(t_1, \ldots, t_n)$. Поэтому для операторов присваивания имеет место правило вывода

$$\frac{\alpha \Rightarrow \beta(t_1, \ldots, t_n)}{\alpha \Rightarrow \langle v_1 := t_1, \ldots, v_n := t_n \rangle \beta(v_1, \ldots, v_n)} \qquad (5.7)$$

и соответствующее правило сведения:

П р а в и л о 4. Для доказательства утверждения $\alpha \Rightarrow \langle (v_1 := t_1, \ldots, v_n := t_n) \rangle \beta(v_1, \ldots, v_n)$ достаточно доказать, что $\alpha \Rightarrow \beta(t_1, \ldots, t_n)$.

Сделаем некоторые замечания относительно действий с числовыми данными. Хорошо известно, что операции над вещественными числами в реальной вычислительной машине выполняются приближенно. Поэтому все рассуждения о корректности программы R имеют смысл только для идеальной машины с точными вычислениями. Разумеется, такие вычисления можно промоделировать на любой универсальной машине, однако в большинстве практических задач предполагается, что арифметические операции, реализованные аппаратно, выполняются приближенно. В этом случае обоснование корректности программы R неправомочно. Она вообще не будет корректной относительно (α, β), хотя при некоторых достаточно сильных ограничениях на функцию f программу все же можно использовать.

Как же поступать для доказательства свойств приближенных числовых программ? Есть два возможных подхода. Первый подход состоит в том, что арифметические операции интерпретируются на конечной области чисел, которые могут быть представлены в рассматриваемой машине, и обозначают именно те операции, которые в ней реализованы. Недостатком такого подхода является то, что теряются основные свойства этих операций – даже такие, например, как коммутативность, ассоциативность и тем более дистрибутивность. Теряют смысл также любые рассуждения, связанные с непрерывностью.

Другой подход состоит в том, что все арифметические операции сохраняют свой первоначальный математический смысл и интерпретируются на области вещественных чисел, но свойства оператора присваивания, выраженные общим правилом (5.7), трансформируются с учетом приближенного вычисления выражений $t_1, \ldots, t_n$. Действие приближенного оператора присваивания может быть охарактеризовано следующим основным свойством:

$$\upsilon_1 = x_1 \wedge \ldots \wedge \upsilon_n = x_n \wedge \alpha_0(x_1, \ldots, x_n) \Rightarrow \exists(\epsilon_1, \ldots, \epsilon_n), \quad |\epsilon_1| \leqslant \delta_1 \wedge \ldots$$
$$\ldots \wedge |\epsilon_n| \leqslant \delta_n \wedge [\upsilon_1 := t_1(\upsilon), \ldots, \upsilon_n := t_n(\upsilon)](\upsilon_1 = t_1(x) + \epsilon_1 \wedge \ldots$$
$$\ldots \wedge \upsilon_n = t_n(x) + \epsilon_n). \tag{5.8}$$

Здесь $x = (x_1, \ldots, x_n)$ – произвольный набор чисел, $\upsilon = (\upsilon_1, \ldots, \upsilon_n)$. Условие $\alpha_0(x_1, \ldots, x_n)$, которому должны удовлетворять числовые значения переменных $\upsilon_1, \ldots, \upsilon_n$, а также значения констант $\delta_1, \ldots, \delta_n$ зависят от структуры выражений $t_1, \ldots, t_n$. В частности, если t_i есть просто переменная, то естественно считать, что ее значение передается точно и $\delta_i = 0$. Условие α_0 определяет диапазоны возможных значений $x_1, \ldots, x_n$ и обычно выражается в виде системы неравенств, в которых фигурируют наибольшее и наименьшее из чисел, представимых в рассматриваемой машине. Вместо абсолютных оценок погрешностей, разумеется, могут быть использованы более точные относительные оценки. В соответствии с рассмотренным основным свойством приближенного присваивания правило (5.7) должно быть теперь заменено более общим правилом:

$$\frac{\alpha \Rightarrow \alpha_0(\upsilon_1, \ldots, \upsilon_n) \wedge \forall(\epsilon_1, \ldots, \epsilon_n)|\epsilon_1| \leqslant \delta_1 \wedge \ldots \wedge |\epsilon_n| \leqslant \delta_n \Rightarrow \beta(t_1(\upsilon)+\epsilon_1, \ldots, t_n(\upsilon)+\epsilon_n)}{\alpha \Rightarrow [\upsilon_1 := t_1(\upsilon), \ldots, \upsilon_n := t_n(\upsilon)]\beta(\upsilon_1, \ldots, \upsilon_n)}. \tag{5.9}$$

Разумеется, квадратные скобки могут быть заменены угловыми. Справедливость правила (5.9) непосредственно вытекает из условия (5.8). Действительно, пусть $y = (\upsilon_1 := t_1(\upsilon), \ldots, \upsilon_n := t_n(\upsilon))$ и посылка правила (5.9) имеет место. Рассмотрим произвольное состояние памяти b, на котором истинно условие α. Пусть в состоянии b переменные $\upsilon_1, \ldots, \upsilon_n$ имеют значения $x_1, \ldots, x_n$ соответственно. Из α следует $\alpha_0(x_1, \ldots, x_n)$, откуда в состоянии by переменные $\upsilon_1, \ldots, \upsilon_n$ примут значения $t_1(x) + \epsilon_1, \ldots, t_n(x) + \epsilon_n$ соответственно для подходящих $\epsilon_1, \ldots, \epsilon_n$. Тогда по предположению условие $\beta(t_1(x) + \epsilon_1, \ldots, t_n(x) + \epsilon_n)$ истинно и, следовательно, в состоянии by истинно условие $\beta(\upsilon_1, \ldots, \upsilon_n)$, что и требовалось.

В случае точных вычислений $\delta_1 = \ldots = \delta_n = 0$, и основное свойство присваивания превращается в свой частный случай:

$$\upsilon_1 = x_1, \ldots, \upsilon_n = x_n \Rightarrow [(\upsilon_1 := t_1(\upsilon), \ldots, \upsilon_n := t_n(\upsilon))](\upsilon_1 = t_1(x), \ldots, \upsilon_n = t_n(x)).$$

Пусть $y = (\upsilon_1 := t_1(\upsilon), \ldots, \upsilon_n := t_n(\upsilon))$ и $y' = (\upsilon_1 := t_1'(\upsilon), \ldots, \upsilon_n := t_n'(\upsilon))$ – два оператора присваивания. В случае точных вычислений существует оператор присваивания z такой, что $z = yy'$. В качестве z можно взять оператор $z = (\upsilon_1 := t_1'(t_1(\upsilon), \ldots, t_n(\upsilon)), \ldots, \upsilon_n := t_n'(t_1(\upsilon), \ldots, t_n(\upsilon)))$. Композиции присваиваний удобно использовать для доказа-

тельства корректности бесцикловых программ. При этом бесцикловую программу удобно приводить к канонической форме вида $\underset{\alpha_1}{(} P_1 \vee \underset{\alpha_2}{(} P_2 \vee \ldots$ $\ldots \vee \underset{\alpha_n}{(} P_n \vee P_{n+1}) \ldots))$, где $P_1, \ldots, P_{n+1}$ – произведения присваиваний. Доказательство корректности такой программы сводится к разбору случаев ($\alpha_1 = 1$, $\alpha_1 = 0 \wedge \alpha_2 = 1$, ...) и доказательству корректности для присваиваний. Заметим, что условия $\alpha_1, \ldots, \alpha_n$ не содержат операции прогнозирования, поскольку она исключается применением соотношения

$$(v_1 := t_1, \ldots, v_n := t_n)\alpha(v_1, \ldots, v_n) = \alpha(t_1, \ldots, t_n).$$

В случае приближенных вычислений дело обстоит несколько сложнее, поскольку произведение yy' уже нельзя заменить одним присваиванием. Однако можно воспользоваться следующим свойством:

$$v_1 = x_1 \wedge \ldots \wedge v_n = x_n \Rightarrow \exists(\epsilon_1, \ldots, \epsilon_n, \epsilon_1', \ldots, \epsilon_n') \mid \epsilon_1 \mid \leqslant$$

$$\leqslant \delta_1 \wedge \ldots \wedge \mid \epsilon_n \mid \leqslant \delta_n \wedge \mid \epsilon_1' \mid \leqslant \delta_1' \wedge \ldots \wedge \mid \epsilon_n' \mid \leqslant \delta_n' \wedge v_1 =$$

$$= t_1'(t_1 + \epsilon_1, \ldots, t_n + \epsilon_n) + \epsilon_1' \wedge \ldots \wedge v_n = t_n'(t_1 + \epsilon_1, \ldots, t_n + \epsilon_n) + \epsilon_n'.$$

Это свойство и соответствующая модификация правила (5.9) легко обобщаются на произведение любого числа операторов присваивания и могут быть использованы при доказательстве утверждений о программах с приближенными вычислениями.

В качестве примера рассмотрим программу R', которая так же, как и R, ищет корень уравнения $f(x) = 0$, но с учетом приближенных вычислений.

Программа R' имеет следующий вид:

НАЧАЛО
  $x := a$, $y := b$;
  $z := (x + y)/2$;
  $w := f(z)$.
  ПОКА $\mid w \mid > \epsilon$ ИЛИ $y - x > \epsilon$ ВЫПОЛНЯТЬ
    ЕСЛИ $w < -\epsilon$ ТО $x := z$ ИНАЧЕ
    ЕСЛИ $w > \epsilon$ ТО $y := z$ КЕ;
    $z := (x + y)/2$;
    $w := f(z)$;
  КЦ;
    $z_1 := (z + y)/2$; $w := f(z_1)$;
  ПОКА $\mid w \mid > \epsilon$ ИЛИ $y - z > \epsilon$ ВЫПОЛНЯТЬ
    ЕСЛИ $w > \epsilon$ ТО $y := z_1$ КЕ;
    $z_1 := (z + y)/2$;
    $w := f(z_1)$;
  КЦ;
  $z_1 := (x + z)/2$; $w := f(z_1)$;
  ПОКА $\mid w \mid < -\epsilon$ ИЛИ $z - x > \epsilon$ ВЫПОЛНЯТЬ

ЕСЛИ $w < -\epsilon$ ТО $x := z_1$ КЕ;
$z_1 := (z + x)/2$;
$w := f(z_1)$
КЦ.
КОНЕЦ.

Программа R' представляется регулярным выражением $R' = P_1 \{ Q_1 \}_{\gamma_1} \times$ $\times P_2 \{ Q_2 \}_{\gamma_2} P_3 \{ Q_3 \}_{\gamma_3}$. В первом цикле интервал $[a, b]$, в котором находится искомый корень уравнения $f(x) = 0$, сокращается путем последовательного деления пополам. Цикл кончается, если интервал $[x, y]$ стал меньше ϵ или внутри его функция принимает значения, близкие к 0, и знак этих значений не может быть точно определен. Два следующих цикла делают попытки дальнейшего сокращения интервала путем уменьшения его правой границы и увеличения левой границы. Начальное условие α для программы R' содержит ту же самую информацию, что и начальное условие для R, и, кроме того, некоторое условие $\alpha_0(a, b)$, которое является достаточным для возможности вычисления приближенного значения $f(x)$ в любых точках x этого интервала. Заключительное условие β для программы R' существенно ослабляется.

Мы можем доказать только, что после выполнения программы R' внутри интервала $[x, y]$ существует такое c, что $f(c) = 0$. Доказательство корректности программы R' существенным образом опирается на правило (5.9), заменяющее теперь (5.7). Применительно к конкретным операторам, используемым в программе R', будем считать, что

$$\langle z := (x + y)/2 \rangle \quad (z = (x + y)/2 + \epsilon_1),$$

$$\langle w := f(z) \rangle \quad (w = f(z) + \epsilon_1)$$

для некоторого ϵ_1 такого, что $|\epsilon_1| \leqslant \epsilon_0$. Предполагается, что константа ϵ, используемая в программе, удовлетворяет неравенству $b - a > \epsilon$ и что $\epsilon > 2\epsilon_0$. Для применения метода Хоара достаточно найти инварианты трех циклов — α_1, α_2 и α_3 соответственно — и доказать следующие утверждения:

$$\alpha \Rightarrow \langle P_1 \rangle \alpha_1,$$

$$\bar{\gamma}_1 \wedge \alpha_1 \Rightarrow \langle Q_1 \rangle \alpha_1,$$

$$\gamma_1 \wedge \alpha_1 \Rightarrow \langle P_2 \rangle \alpha_2,$$

$$\bar{\gamma}_2 \wedge \alpha_2 \Rightarrow \langle Q_2 \rangle \alpha_2,$$

$$\gamma_2 \wedge \alpha_2 \Rightarrow \langle P_3 \rangle \alpha_3,$$

$$\bar{\gamma}_3 \wedge \alpha_3 \Rightarrow \langle Q_3 \rangle \alpha_3,$$

$$\gamma_3 \wedge \alpha_3 \Rightarrow \beta.$$

Заметим, что при таком подходе фактически отсутствуют промежуточные условия для произведения, поскольку в качестве таких условий

используются инварианты циклов. Это общий прием, который вместе с приведенными выше соображениями о доказательствах корректности бесцикловых программ показывает, что искусство применения метода Хоара состоит прежде всего в отыскании инвариантов циклов.

Для рассматриваемого примера инварианты могут быть определены следующим образом. Все инварианты α_1, α_2, α_3 содержат условие α', использованное для доказательства корректности программы R: на концах интервала $[x, y] \subset [a, b]$ функция f принимает значения разных знаков, $f(x) < 0$, $f(y) > 0$. Условие α_1 включает в себя также условия $|z - (x+y)/2| < \epsilon_0$, $|f(z) - w| < \epsilon_0$. Условие α_2 включает неравенства $x < z < z_1 < y$, $|z_1 - (x+y)/2| < \epsilon_0$, $|f(z_1) - w| < \epsilon_0$, а α_3 – неравенства $x < z_1 < y$, $|z_1 - (x+z)/2| < \epsilon_0$, $|f(z_1) - w| < \epsilon_0$.

Докажем, например, что $\overline{\gamma}_1 \wedge \alpha_1 \Rightarrow \langle Q_1 \rangle \alpha_1$. Предположим, что $\overline{\gamma}_1 \wedge \alpha_1$, и рассмотрим два случая.

1. $w < -\epsilon$. Тогда Q_1 эквивалентен последовательности присваиваний $(x := z,\ z := (x+y)/2;\ w := f(z))$. Доказательство условия $\overline{\gamma}_1 \wedge \alpha_1 \Rightarrow \langle Q_1 \rangle \alpha_1$ сводится в этом случае к доказательству утверждения $\overline{\gamma}_1 \wedge \alpha_1(x, y, z, w) \Rightarrow \alpha_1(z, y, (x+y)/2 + \epsilon_1, f((x+y)/2 + \epsilon_1) + \epsilon_2))$ для произвольных ϵ_1 и ϵ_2 таких, что $|\epsilon_1|$, $|\epsilon_2| \leqslant \epsilon_0$, или, выписав явно все условия, получим, что требуется доказать импликацию

$$w < -\epsilon \wedge (|w| > \epsilon \vee y - x > \epsilon) \wedge a \leqslant x < y \leqslant b \wedge f(x) < 0 \wedge f(y) > 0 \wedge$$

$$\wedge |z - (x+y)/2| < \epsilon_0 \wedge |f(z) - w| < \epsilon_0 \Rightarrow a \leqslant z < y \leqslant b \wedge$$

$$\wedge f(z) < 0 \wedge f(y) > 0 \wedge |(x+y)/2 + \epsilon_1 - (x+y)/2| < \epsilon_0 \wedge$$

$$\wedge |f((x+y)/2 + \epsilon_1) - f((x+y)/2 + \epsilon_1) - \epsilon_2| < \epsilon_0.$$

Доказательство тривиально и предоставляется читателю.

2. $w > \epsilon$. Доказывается аналогично.

Все остальные условия также получают тривиальные доказательства после сведения их к элементарным.

Метод Хоара применяется для доказательства корректности регулярных программ. Для программ, представленных в виде схем, применяется другой метод, называемый *методом Флойда*. Пусть A – детерминированная U–Y-схема программы, интерпретированная на B. Рассмотрим путь $p = a_1 \xrightarrow{u_1/y_1} a_2 \xrightarrow{u_2/y_2} \ldots \xrightarrow{u_m/y_m} a_{m+1}$. Сопоставим пути p условие $\alpha(p)$ и оператор $y(p)$, определив их следующим образом:

$$y(p) = y_1 \ldots y_m,$$

$$\alpha(p) = u_1 \wedge \langle y_1 \rangle u_2 \wedge \langle y_1 y_2 \rangle u_3 \wedge \ldots \wedge \langle y_1 \ldots y_{m-1} \rangle u_m.$$

Очевидно, что $y(p)$ – это оператор, который выполняется при прохождении пути p, а $\alpha(p)$ – условие, которое истинно тогда и только тогда, когда программа A пройдет по пути p из состояния a_1.

Метод Флойда применяется для доказательства утверждения вида $\alpha \Rightarrow \langle f_A \rangle \beta$ и состоит в следующем. В множестве A состояний схемы программы выбирается подмножество $C = A$, элементы которого называются *контрольными точками.* Множество C должно содержать начальное и заключительное состояния схемы. Путь p называется *контрольным*, если он начинается и оканчивается в контрольных точках, но ни одно из промежуточных состояний не принадлежит множеству C. Множество контрольных точек выбирается таким образом, чтобы любые две контрольные точки соединялись не более, чем конечным числом контрольных путей. Каждому состоянию $a \in C$ ставится в соответствие некоторое условие $\gamma(a)$, называемое *инвариантом* состояния a. Далее строятся условия, называемые *условиями корректности* схемы A. Каждое условие корректности соответствует начальному или заключительному состоянию либо паре контрольных точек. Начальному состоянию $a^{(0)}$ соответствует условие $\alpha \Rightarrow \gamma(a^{(0)})$, заключительному – условие $\gamma(a^{(1)}) \Rightarrow \beta$. Паре контрольных состояний a, a' соответствует условие

$$\bigvee_{a \xrightarrow{p} a'} \gamma(a) \wedge \alpha(p) \Rightarrow \langle y(p) \rangle \gamma(a'),$$

где дизъюнкция берется только по контрольным путям.

Т е о р е м а (Флойда). *Если условия корректности схемы A истинны, то схема A частично корректна.*

Действительно, пусть $f_A(b)$ определено и $\alpha(b) = 1$. Тогда существует путь p такой, что $a^{(0)} \xrightarrow{p} a^{(1)}$, $\alpha(p) = 1$ на b и $f_A(b) = b' = by(p)$. Путь p может быть разложен единственным образом в последовательную композицию контрольных путей: $a^{(0)} \xrightarrow{p_1} a_1 \xrightarrow{p_2} a_2 \to \ldots \xrightarrow{p_m} a_m = a^{(1)}$. В моменты прохождения контрольных точек $a^{(0)}, a_1, \ldots, a_m = a^{(1)}$ информационная среда находится в состояниях $b = b_0, b_1, \ldots, b_m = b'$, где $b_{i+1} = b_i y(p_i)$ $(i = 0, \ldots, m-1)$.

Покажем индукцией по $i = 0, 1, \ldots$, что $\gamma(a_i) = 1$ на состоянии b_i. $\gamma(a^{(0)}) = 1$, поскольку $\alpha \Rightarrow \gamma(a^{(0)})$. Пусть $\gamma(a_i) = 1$ на b_i. По условию корректности $\gamma(a_i) \wedge \alpha(p_i) \Rightarrow \langle y(p_i) \rangle \gamma(a_{i+1})$, откуда $\gamma(a_{i+1}) = 1$ на b_{i+1}. Наконец, поскольку $\gamma(a^{(1)}) \Rightarrow \beta$, получаем, что $\beta(b') = 1$. Теорема доказана.

Поскольку регулярную программу всегда можно представить в виде схемы, метод Флойда применим и к регулярным программам. В некоторых случаях его удобнее применять, чем метод Хоара.

§ 6. Параллельные алгоритмы

Наиболее общая модель параллельных вычислений – это многоуровневая многокомпонентная система. Она может быть настроенной, если используется в качестве модели параллельной программы, или ненастроенной, если используется в качестве модели вычислительной системы или компо-

нент общесистемного математического обеспечения. Многокомпонентная система может иметь также переменную структуру, т.е. менять время от времени число компонент на различных уровнях. В процессе функционирования компоненты выполняют вычисления над данными, расположенными в них, а также взаимодействуют с другими компонентами, порождая вычисления над распределенными данными. Многоуровневая иерархия компонент определяет их организацию и распределение функций по управлению, вычислениям, перемещениям данных и перестройке различных частей системы в процессе решения данной задачи или совокупности задач.

Для того чтобы иметь возможность удобного описания различных конкретных классов многоуровневых систем, рассматривают различные специальные способы организации взаимодействия компонент в таких системах. Проще всего взаимодействие организуется в сетях из автоматов. При синхронной работе сети считается, что изменения состояний входных компонент происходят мгновенно и достаточно редко для того, чтобы внутри сети успевали закончиться все переходные процессы. Сети из автоматов применяются обычно для описания проектируемой аппаратуры. Выполнение указанных условий обеспечивается, например, с помощью генераторов тактовых сигналов, разрешающих реакцию на входные воздействия после окончания их изменения. Тактовая частота является основным параметром, характеризующим быстродействие аппаратуры. Использование асинхронных сетей из автоматов требует их усложнения для обеспечения детерминированности на уровне моделей более высокого уровня, но позволяет добиться лучших временных характеристик.

Алгоритмическое описание взаимодействия устройств и программ требует перехода от синхронного описания функционирования сетей к асинхронному. Действительно, время в различных компонентах протекает по-разному, поскольку количество тактов, которое затрачивается на один переход, зависит от реализации действий, совершаемых на этом переходе. Лишь для некоторых переходов требуется, чтобы они происходили быстрее, чем изменение состояния некоторых входных сигналов.

Например, автомат, управляющий устройством ввода-вывода или внешним запоминающим устройством, инициирует работу этого устройства, посылая управляющий сигнал, после чего на его вход подается с определенной частотой последовательность сигналов. Эти сигналы необходимо принять и запомнить, или следует произвести некоторую их обработку в реальном масштабе времени. Очевидно, что переход из очередного состояния ожидания поступления сигнала с внешнего устройства в следующее состояние ожидания должен произойти до того, как поступит новый сигнал.

Асинхронное взаимодействие компонент удобно представлять с помощью сетей из алгоритмических модулей. Алгоритмический модуль — это дискретный преобразователь над памятью, информационная среда которого кроме собственной внутренней памяти включает в себя дополнительные компоненты, выполняющие особые функции и предназначенные для описания взаимодействия модулей между собой. К ним относятся входные и выходные компоненты и общая память. Все эти компоненты называются *внешними.* Если их структурно объединить в одну внешнюю компоненту, то получится, что алгоритмический модуль структурно изоморфен трех-

компонентной системе, состоящей из управляющей компоненты, внутренней памяти и внешней компоненты.

Входные и выходные компоненты устроены так же, как и обычная память, т.е. состоят из переменных, каждая из которых имеет свою область значений, однако базис операторов и условий определен таким образом, что управляющая компонента, как правило, только изменяет, но не использует выходные переменные и использует, но не изменяет значения входных переменных. Кроме того, будем рассматривать некоторые дополнительные характеристики входных и выходных переменных. Будем различать переменные с памятью и без памяти, с простыми очередями и ветвящимися очередями.

Переменные с памятью меняют свои значения так же, как и переменные обычной памяти, т.е. в результате выполнения операторов присваивания. Некоторые из выходных переменных с памятью могут быть одновременно и внутренними. В этом случае модуль может не только изменять, но и использовать значения этих переменных. Переменные без памяти перевычисляются на каждом такте работы алгоритмического модуля с помощью функции выходов $y = \varphi(a, b, x)$, заданной для каждой выходной переменной y. Эта функция зависит от состояния a управляющей компоненты модуля, состояния памяти b (внутренней и общей), а также от состояния x входных переменных, исключая переменные с очередями.

Переменная с простой очередью принимает в качестве значений последовательности, составленные из элементов некоторой области D. Изменение значения переменной с очередью x происходит при выполнении операторов ПЕРЕДАТЬ y В x и ПРИНЯТЬ y ИЗ x. Первый оператор применяется в алгоритмическом модуле, для которого эта переменная является выходной, второй — в алгоритмическом модуле, для которого эта переменная входная. Если $x = d_1 \ldots d_n$, то после выполнения оператора ПЕРЕДАТЬ y В x она получит значение $d_1 \ldots d_n d_{n+1}$, где d_{n+1} есть значение выражения y в текущем состоянии памяти. В операторе ПРИНЯТЬ y ИЗ x y есть переменная. В результате выполнения этого оператора новым значением x будет $d_2 \ldots d_n$, а новым значением переменной y — значение d_1. Если $n = 1$, новым значением x будет пустая последовательность e ($ed = d$). Если $x = e$, то оператор приема выполняется как оператор ожидания момента, когда x станет непустым, после чего будет выполнен прием. В условном операторе прием и передача по времени не совмещаются с проверкой условия. Это значит, что оператор ЕСЛИ u ТО ПРИНЯТЬ ... КЕ выполняется как последовательность из двух операторов:

ЕСЛИ u ТО НА l КЕ;

l: ПРИНЯТЬ ... .

Переменная с ветвящейся очередью может быть использована в качестве общей входной переменной для нескольких алгоритмических модулей. При этом оператор ПРИНЯТЬ не отбрасывает первый элемент очереди, а лишь продвигает данный модуль по очереди для того, чтобы при следующем выполнении оператора ПРИНЯТЬ был принят следующий элемент очереди. Первый элемент очереди x отбрасывается лишь после того, как он был принят всеми модулями, для которых x есть входная переменная.

Сети из алгоритмических модулей получаются так же, как и сети из автоматов, — путем отождествления некоторых внешних компонент. Отождествления должны удовлетворять следующим условиям корректности.

1. В каждом классе отождествленных компонент лишь одна выходная, а все другие входные, либо все входные, либо все переменные общей памяти.

2. Если в классе отождествленных компонент одна выходная, то ее область значений содержится в областях значений всех других, и все они имеют одинаковые характеристики (с памятью, без памяти и т.д.).

3. Если в классе отождествленных компонент все входные, то они имеют одинаковые характеристики и области значений.

4. Если в классе отождествленных компонент все общие, то они должны иметь одинаковые области значений.

Понятия открытой, полуоткрытой и замкнутой сети из алгоритмических модулей вводятся аналогично тому, как это было сделано для сетей из открытых систем. При этом следует только учесть, что для определения этих понятий кроме входных и выходных переменных следует использовать также понятия переменных общей памяти, которые играют роль одновременно как входных, так и выходных переменных. Кроме того, переменные с очередями ведут себя не как свободные компоненты, а изменяют свои значения определенным образом. А именно, если x есть входная компонента сети, то в произвольный момент времени ее значение может измениться только путем добавления к очереди нового элемента или удаления первого при выполнении оператора ПРИНЯТЬ в одном из модулей.

Таким образом, сеть из алгоритмических модулей сама может рассматриваться как алгоритмический модуль и использоваться для построения многоуровневых сетей. Замкнутая сеть из алгоритмических модулей может рассматриваться как дискретный преобразователь, если выделить информационную компоненту и определить настройку. Открытая сеть также может рассматриваться как дискретный преобразователь, если она не имеет входных и выходных переменных без памяти, а значения очередей и общей памяти изменяются только самой сетью. В случае открытой сети нас может интересовать ее взаимодействие с некоторой внешней средой, представленной своей дискретной моделью. В этом случае основная характеристика сети есть проекция композиции сети и внешней среды на внешнюю среду и внешнюю компоненту сети.

Для того чтобы уточнить правила функционирования сети из алгоритмических модулей, предположим, что управляющая компонента каждого модуля представлена U—Y-схемой программы над памятью. Базис (U, Y) является базисом операторов и условий над памятью данного модуля и, разумеется, зависит от этого модуля. Базис операторов Y кроме операторов над памятью содержит операторы обращения к очередям.

Для каждого оператора $P \in Y$ и для каждого состояния b информационной среды модуля M заданы множества $\mathrm{In}(P, b)$ и $\mathrm{Out}(P, b)$ переменных. Множества In и Out определяют переменные, которые оператор y использует и вырабатывает соответственно. Зависимость множеств In и Out от

текущего состояния b информационной среды необходимо учитывать при работе с массивами и другими видами памяти косвенным именованием. Схема может быть недетерминированной или частично определенной. Состояние сети есть набор состояний ее компонент, включая состояния входных и выходных компонент и общей памяти сети. Для сети задана система равенств, определяющих связи (отождествление компонент).

Рассмотрим произвольное текущее состояние b сети. Переход $a \xrightarrow{u/P} a'$ называется *выполнимым*, если $u(b) = 1$ или $u = 1$ (независимо от текущего состояния сети), P есть оператор приема из входной очереди x (значение x отлично от e). В состоянии a управляющая компонента каждого модуля определяет некоторое множество выполнимых переходов, либо находится в заключительном состоянии, либо находится в тупиковом состоянии, т.е. в таком состоянии, что все условия всех переходов для этого состояния либо ложны, либо не определены. Если все модули находятся в заключительном состоянии, считаем, что сеть перешла в заключительное состояние. Если хотя бы один модуль находится в тупиковом состоянии, считаем, что вся сеть находится в тупиковом состоянии.

Пусть текущее состояние сети не является ни заключительным, ни тупиковым. Рассмотрим на множестве переходов отношение совместимости. Два перехода *совместимы*, если операторы, соответствующие этим переходам, не изменяют одновременно одну и ту же переменную общей памяти и не используют одновременно оператором ПРИНЯТЬ одну и ту же переменную с простой очередью (все это с учетом отождествлений, определяемых связями). Переход в следующее состояние происходит в два этапа.

На первом этапе выполняются некоторые из переходов, на втором определяются значения переменных без памяти. Операторы P' и P'' модулей M' и M'' соответственно называются *совместимыми* относительно состояния b информационной среды сети, если $\mathrm{Out}(P', b') \cap \mathrm{Out}(P'', b'') = \phi$, где b' и b'' — компоненты информационной среды, соответствующие модулям M' и M'', а $\mathrm{In}(P', b') \cap \mathrm{In}(P'', b'')$ не содержит переменных с простыми очередями (все это с учетом отождествлений, определяемых связями). Если сеть имеет общую память, то для выполнения первого этапа выберем сначала произвольным образом оператор $Q_0 = (v_1 := d_1, \ldots, v_m := d_m)$, изменяющий переменные $v_1, \ldots, v_m$ общей памяти, и выберем операторы $Q_1, Q_2, \ldots$, изменяющие некоторые из входных и выходных переменных сети с очередями. Выберем теперь некоторое множество выполнимых переходов так, чтобы операторы $P_1, P_2, \ldots$, определенные этими переходами, вместе с оператором Q_0 были попарно совместимы. Выполним все выбранные операторы. Первый этап выполнен.

Для выполнения второго этапа определим сначала произвольным образом значения входных переменных без памяти для всей сети и рассмотрим систему уравнений $v = \varphi(a, b, x)$, определяющих значения выходных переменных без памяти для компонент сети. Если эта система имеет решение, то возьмем это решение в качестве нового состояния выходных переменных компонент. Если решения нет, то переход не определен. Модули, для

которых были выполнены переходы в данный момент времени, называются *активными.* Остальные модули называются *задержанными.*

К описанным правилам можно добавить еще одно: ни один модуль не может быть задержанным бесконечно долго. Впрочем, это дополнение существенно лишь в случае, если нас интересует рассмотрение бесконечных процессов. Состояние a модуля называется состоянием ожидания, если имеется переход $a \xrightarrow{u/\epsilon} a$ и условие u истинно. Если все модули замкнутой сети находятся в состоянии ожидания, то такое состояние также называется тупиковым. Из тупикового состояния выйти невозможно.

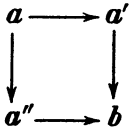
Очевидно, сеть из алгоритмических модулей, вообще говоря, является недетерминированной системой, даже если все модули управляются детерминированными схемами программ, а сеть замкнута. Основным источником недетерминизма является произвол в выборе модулей, которые реализуют свои переходы на некотором шаге.

Если замкнутая сеть из алгоритмических модулей используется для реализации однозначного преобразования, то важным свойством сети является ее глобальная детерминированность. Источники глобальной недетерминированности — общая память и входные переменные, за исключением ветвящихся очередей. Действительно, если два модуля вырабатывают значение некоторой переменной из общей памяти, результат зависит от того, какой из них сделает это раньше. Очередное изменение значения переменной с памятью может произойти раньше, чем предыдущее значение будет использовано. Наконец, если простая очередь является входом одновременно для нескольких модулей, то один из них может прочесть и, следовательно, удалить очередное значение из очереди раньше, чем другой.

Интересная модель глобально детерминированной системы получается, если ограничиться рассмотрением входных и выходных переменных с простыми очередями без ветвлений, т.е. каждый выход соединен не более чем с одним входом. Такие сети будем называть *простыми асинхронными сетями.*

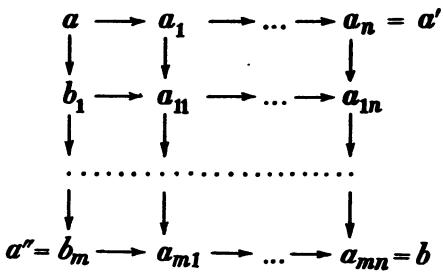
Теорема 6.1. *Простая асинхронная замкнутая сеть из алгоритмических модулей, управляемых детерминированными схемами программ, глобально детерминирована.*

Справедливость этого утверждения достаточно очевидна, но мы рассмотрим его доказательство для демонстрации общего метода. Рассмотрим сначала важное понятие коммутативного отношения. Отношение $a \to a'$ на множестве A назовем *коммутативным*, если из $a \to a'$ и $a \to a''$ вытекает существование такого $b \in A$, что $a' \to b$ и $a'' \to b$. Это условие можно также изобразить в виде коммутативной диаграммы:

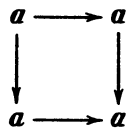


Лемма 6.1. *Рефлексивно-транзитивное замыкание коммутативного отношения коммутативно.*

Доказательство получается заполнением коммутативной диаграммы:



В случае же, если $a_n = b_m$, имеем коммутативную диаграмму:



Докажем теперь теорему 6.1. Пусть S – сеть, удовлетворяющая условиям теоремы. Рассмотрим произвольное состояние s сети S, и пусть T – множество всех выполнимых переходов этой сети. Выберем произвольным образом $T' \subset T$. Очевидно, операторы, порождаемые всеми переходами из T', совместимы. Выполнив их, получим новое состояние сети s': $s \to s'$. Пусть $T'' \subset T$ – другое подмножество множества T. Выполняя переходы из T'', получим состояние s'': $s \to s''$. В состоянии s' все переходы множества $T \setminus T'$ останутся выполнимыми, так же как и все переходы множества $T \setminus T''$ в состоянии s''. Но тогда, если $s' \to s'''$ путем выполнения переходов множества $T'' \setminus T'$, то $s'' \to s'''$ путем выполнения переходов множества $T' \setminus T''$. Таким образом, отношение непосредственных переходов коммутативно и по лемме 6.1 будет коммутативным и его рефлексивно-транзитивное замыкание, т.е. отношение достижимости. Поэтому если $s \overset{p}{\to} s'$ и $s \overset{q}{\to} s''$ – два терминальных процесса, то $s' = s''$. Теорема доказана.

Частным случаем простых асинхронных сетей является сеть, в которой любые два модуля M_i и M_j связаны очередью x_{ij} (выходной для M_i и входной для M_j). В этом случае в операторах приема и передачи вместо имен каналов можно обращаться к именам модулей. Например, вместо ПРИНЯТЬ y ИЗ x_{ij} можно писать ПРИНЯТЬ y ИЗ M_j. Такая модель соответствует взаимодействующим процессам Хоара.

При работе с моделями рассмотренного типа следует соблюдать осторожность. Так, например, для работы с очередями удобно иметь возможность пользоваться условием $x = e$, где x – входная переменная с очередью. Уже одно это обстоятельство может нарушить глобальную детерминированность, поскольку оператор ЕСЛИ $x = e$ ТО P ИНАЧЕ Q КЕ является недетерминированным, и эта недетерминированность может повлиять на окончательный результат.

Рассмотренный набор средств взаимодействия параллельных процессов, протекающих в сетях из алгоритмических модулей, является в некотором смысле избыточным. Одни из них могут быть выражены через другие. С другой стороны, рассмотренный набор средств дает возможность программировать более сложные средства взаимодействия параллельных процессов. Рассмотрим некоторые из таких конструкций.

Ветвящиеся очереди можно заменить простыми очередями, вводя для каждой ветвящейся очереди дополнительный модуль. Пусть, например, z есть выходная ветвящаяся очередь некоторого модуля сети S, связанная со входами $x_1, \ldots, x_m$ некоторых других модулей. Введем новый модуль M, который имеет одну простую входную очередь x и m выходных $y_1, \ldots, y_m$.

Алгоритм работы модуля M имеет вид

```
ПОКА α ВЫПОЛНЯТЬ
  ЕСЛИ z ≠ e ТО
    ПРОЧИТАТЬ z В u;
    ДЛЯ i := 1 ДО m ВЫПОЛНИТЬ
      ЗАПИСАТЬ u В y_i
    КЦ
  КЕ
КОНЕЦ ЦИКЛА.
```

Условие α используется для того, чтобы можно было прекратить, если это нужно, работу модуля M. Теперь в сети S переменные $z, x_1, \ldots, x_m$ можно заменить простыми очередями, разорвать между ними связи и добавить модуль M, связав его с компонентами данной сети связями $z = x$ и $y_1 = x_1, \ldots, y_m = x_m$. Для того чтобы сохранить неизменной функциональную модель сети S, т.е. вычисляемую ею функцию f_S (предполагается, что S есть дискретный преобразователь), необходимо обеспечить прекращение работы компоненты M после того, как все остальные перешли в заключительное состояние. Для этого можно ввести общую переменную p, значение которой в начальный момент времени всегда равно 0, а в каждый модуль добавить в конце оператор $p := p + 1$. Если n – общее число модулей в исходной сети, то в качестве условия α следует взять условие $p < n$.

Ограничения реализации могут потребовать использования очередей ограниченного объема. Этого можно добиться, если для каждой очереди использовать переменную общей памяти, хранящую в произвольный момент времени число элементов, находящихся в очереди (счетчик очереди). Каждая передача в очередь должна сопровождаться увеличением этого числа, а каждый прием – уменьшением. Если число элементов в очереди превышает заданный порог, то модуль, который пытается передать данные в очередь, должен ждать уменьшения числа элементов очереди. Частным случаем является очередь длины 1. В этом случае счетчик принимает лишь два значения – 0 и 1, а сама очередь превращается в переменную с памятью (входную или выходную).

Переход от сети с неограниченными очередями к сетям с ограниченными очередями, вообще говоря, сокращает область определения функции, вычисляемой сетью, поскольку могут добавиться тупиковые состояния, связанные с ожиданием освобождения места в очереди.

Сети из алгоритмических модулей сохраняют свою структуру на протяжении всего времени их функционирования. Во многих случаях естественным образом возникают модели параллельных вычислений с переменной структурой. Простейший способ введения переменной структуры – порождение параллельных ветвей внутри одного алгоритмического модуля. Для этой цели расширим понятие перехода. Вместо простого перехода $a \xrightarrow{u/P} a'$ будем рассматривать параллельный переход $a \xrightarrow{u/\bar{P}} a_1, \ldots, a_m$. Интерпретация схемы (вернее, ее базиса) определяется так же, как и для обычных схем программ, но функционирование определяется иначе. А именно, состояние интерпретированной параллельной схемы программы A – это пара $((a_1, \ldots, a_m), b)$, где $a_1, \ldots, a_m \in A$, $b \in B$ – состояние информационной среды (внутренняя и внешняя компоненты алгоритмического модуля). Переход в следующее состояние определяется так, как если бы на общей информационной среде действовали m экземпляров одного и того же алгоритмического модуля. А именно, для того чтобы выполнить переход в новое состояние $((a'_1, \ldots, a'_l), b')$, необходимо сделать следующее. Выбираем множество T выполнимых переходов для состояний $a_1, \ldots, a_m$ так, чтобы выполнялись следующие условия:

1. Все операторы переходов множества T попарно совместимы.

2. Операторы приема из входных переменных с очередями применяются только для непустых очередей.

После того как T выбрано, выполняем все операторы, получая новое состояние информационной среды b'. Новый вектор $(a'_1, \ldots, a'_l)$ состояния управления получается следующим образом. Если переход $a_i \xrightarrow{u/P} a_{i_1}, \ldots, a_{i_k}$ попал в множество T, то состояние a_i заменяется на вектор $(a_{i_1}, \ldots, a_{i_k})$. После подстановки вместо всех состояний соответствующих векторов скобки опускаются, чтобы получить одноуровневый вектор. Если среди новых состояний имеются заключительные, то они вычеркиваются. Если же все новые состояния оказались заключительными, то весь вектор заменяется одним заключительным состоянием.

Заметим, что параллельная схема программы фактически эквивалентна некоторой последовательной схеме, в которой параллелизм выражается в терминах групповых операторов.

Другой тип изменения структуры сети из алгоритмических модулей в процессе ее функционирования может быть представлен понятием управляемой сети из алгоритмических модулей. Формально это дискретный преобразователь, информационная компонента которого представляет собой сеть из алгоритмических модулей. В состав информационной компоненты могут также входить и другие компоненты, например память. Часть этой памяти может быть внешней компонентой данной сети. Управляемая сеть может быть также алгоритмическим модулем с внешней компонентой. Такая точка зрения на управляемую сеть позволяет ввести ряд операторов, с помощью которых можно изменять структуру и поведение сети в процессе ее функционирования. Если модули, из которых построена сеть, настроены, то каждый из них при переходе в заключительное состояние останавливается. Если все модули сети перешли в заключительное

состояние, т.е. сеть закончила работу, она может быть вновь запущена оператором установки некоторых модулей в определенное начальное состояние. Работа сети может прерываться и возобновляться. Изменение структуры состоит в добавлении или удалении новых компонент и связей.

В управляемых сетях из алгоритмических модулей можно получить различные варианты алгоритмов функционирования сетей, рассматривая конкретные методы выбора реализуемых переходов на каждом шаге функционирования сети.

В управляемых сетях возможно реализовать также различные глобальные механизмы управления и синхронизации, зависящие от свойств состояния сети в целом в отличие от механизмов локального управления и синхронизации, которые определяются в результате попарного взаимодействия модулей. Один из таких механизмов реализуется в синхронизированных сетях, устроенных следующим образом. В управляющей компоненте каждого модуля выделяется множество синхронизируемых состояний. Если некоторый модуль попадает в такое состояние, он остается в нем до тех пор, пока все модули сети не перейдут также в синхронизированное состояние. Таким образом, процесс функционирования синхронизированной сети распадается в последовательную композицию процессов, называемых *сегментами синхронизации.* В конце каждого сегмента синхронизации каждый модуль сети находится либо в синхронизированном, либо в заключительном состоянии.

В качестве алгоритмического языка для описания сетей из алгоритмических модулей можно использовать язык АЗ со следующим примерным синтаксисом:

⟨ АЗ-программа ⟩ ::= СЕТЬ ⟨ имя ⟩; ⟨ описание внешней компоненты сети ⟩; ⟨ список описаний модулей ⟩; ⟨ список связей ⟩ КОНЕЦ.

⟨ описание модуля ⟩ ::= МОДУЛЬ ⟨ имя ⟩; ⟨ описание информационной среды модуля ⟩ ⟨ оператор ⟩ КОНЕЦ МОДУЛЯ

⟨ связь ⟩ ::= ⟨ входная переменная модуля ⟩ = ⟨ выражение ⟩

Язык может расширяться разнообразными средствами для описания семейств модулей и связей, зависящих от параметров.

Комментарии к главе 2

Первые подходы к построению и изучению математических моделей программ были найдены А. А. Ляпуновым [66] и его учеником Ю. И. Яновым [80], которые заложили основы теории схем программ. Ю. И. Янов впервые ввел отношение эквивалентности (равносильности) абстрактных схем программ и доказал алгоритмическую разрешимость этого отношения. Бинарные детерминированные абстрактные схемы программ называются в литературе схемами Янова. Дальнейшее развитие теории схем программ в СССР связано с именами А. П. Ершова, С. С. Лаврова, В. В. Мартынюка. Обзоры результатов, полученных в теории схем программ, сыгравшей важную роль в развитии теоретического программирования, можно найти в работах [42, 57]. Понятие интерпретированной схемы программы над памятью содержится в работе А. П. Ершова [41]. На Западе теорией схем программ активно стали заниматься лишь в конце 60-х, начале 70-х годов. Значительный вклад в развитие этой теории сделали Лакхэм, Парк, Патерсон, Чандра, Манна и др.

Использование понятия дискретного преобразователя в качестве автоматной модели программы обсуждается в работе [27]. Фактически идея рассмотрения программы (микропрограммы) как автомата содержится в [12], а в [11] явно устанавливается

связь автоматов со схемами Янова и обобщаются результаты о равносильности схем программ. В отличие от работы Рутледжа [4], в которой основные понятия работы Янова лишь переводятся на автоматный язык, работы В.М. Глушкова положили начало новому направлению в прикладной теории алгоритмов, обогатив ее автоматно-алгебраическими методами исследования.

Развитию математических моделей программ способствовал анализ семантики практических языков программирования от фортрана и алгола до паскаля и ады, а также исследования по теории трансляции [4]. Интересный обзор взаимодействия прикладных и теоретических идей дает статья П. Вегнера [93]. Применение недетерминированных конструкций при разработке программ обсуждается Э. Дейкстрой [39] (охраняемые команды). Идея косвенной адресации еще в 60-х годах была использована в одной из первых отечественных технологий – адресном программировании [76].

Алгебра алгоритмов впервые определена в работе [17], где также дано первое доказательство теоремы о регуляризации. Операции алгебры алгоритмов предвосхитили основные конструкции структурного программирования, завоевавшего большую популярность в 70-х годах. Обычно идея структурного программирования связывается с высказываниями Э. Дейкстры о программировании без GO TO [82]. Более широкий взгляд на эту проблему содержится в книге [37]. Приведение произвольных программ к структурному виду рассматривается также в [83]. В отличие от теоремы Глушкова метод Бема и Джакопини требует перехода к новой информационной среде (введение новых логических переменных) и не затрагивает проблем преобразования регулярных программ. Проблемы использования алгебры алгоритмов в качестве технологического средства структурного программирования обсуждаются в монографии [31].

Исследования по логике алгоритмов активно начались после работы Хоора [87], в которой был предложен аксиоматический подход к определению семантики алгоритмических языков и построено исчисление, в терминах которого сформулирован метод доказательства частичной корректности регулярных программ. Его можно рассматривать как частный случай метода Флойда, построенного в [84] для произвольных программ с переходами. Обзоры исследований в области верификации представлены в [71, 77], а обзор по программным логикам – в [72].

Исследование математических моделей параллельных алгоритмов и программ имеет достаточно обширную литературу. Сети из алгоритмических модулей были введены в работе [27], где рассмотрены также и другие структуры и динамические модели параллельных вычислений.

Теорема, аналогичная теореме 6.1, доказана Келлером в [88].

Глава 3

РЕКУРСИВНЫЕ ОПРЕДЕЛЕНИЯ

§ 1. Функциональные уравнения

Если заданы некоторая область данных D и на ней система базовых функций и предикатов, то можно получать новые функции и предикаты, рассматривая множество состояний $B = \Gamma(V, D)$ памяти $V = \{v_1, ..., v_n\}$ и стандартные схемы программ над этой памятью. Действительно, если A – схема программы над B, а f_A – реализуемое ею отображение, то, отождествляя состояние памяти b с набором $(b(v_1), ..., b(v_n))$ элементов множества D, получим, что f_A есть (частично определенная) функция типа $D^n \to D^n$, при этом $f_A(v_1, ..., v_n) = (f_1(v_1, ..., v_n), ..., f_n(v_1, ..., v_n))$. Схема программы A определяет также набор функций $f_1, ..., f_n$ типа $D^n \to D$. Это определение конструктивно в том смысле, что если базовые

функции и предикаты вычислимы, то и функции $f_1, \ldots, f_n$ также вычислимы, а схема программы A, интерпретированная на B, дает алгоритм их вычисления.

Другой конструктивный способ построения новых функций из уже заданных – рекурсивные определения. Этот способ является основным методом в теории рекурсивных функций натурального аргумента. Современная точка зрения на рекурсивные определения трактует их как специальный вид функциональных уравнений. Для того чтобы уяснить эту точку зрения, вспомним основные конструкции теории рекурсивных функций.

Рассмотрим сначала оператор примитивной рекурсии. Он, в частности, дает возможность строить новую функцию $f(x, y)$, исходя из двух заданных функций: $g(x, y, z)$ и $h(x)$. Функции f, g и h связаны соотношениями

$$f(x, y) = g(f(x, y-1), x, y), \; y > 0,$$

$$f(x, 0) = h(x).$$

Эти соотношения определяют функцию $f(x, y)$ однозначно. Действительно, $f(x, y)$ определено $\Longleftrightarrow f(x, 0), \ldots, f(x, y-1)$ определены, поэтому $g(f(x, y-1), x, y)$ также определено при $y > 0$ или $h(x)$ определено при $y = 0$. Соотношения для примитивной рекурсии можно переписать в виде одного равенства, используя функцию **если ... то ... иначе ...**:

$$f(x, y) = \textbf{если } y = 0 \textbf{ то } h(x) \textbf{ иначе } g(f(x, y-1), x, y).$$

Это определение имеет вид

$$f(x, y) = F(f, x, y),$$

где F – функционал, аргументами которого являются функция и числовые значения. Последнее равенство можно рассматривать как функциональное уравнение с неизвестной функцией f. В случае примитивной рекурсии такое уравнение имеет единственное решение.

Рассмотрим теперь оператор наименьшего корня $\mu y (f(x, y) = 0) = g(x)$. Функция $g(x)$ определена и равна y тогда и только тогда, когда $f(x, 0), f(x, 1), \ldots, f(x, y-1)$ все определены и отличны от нуля, а $f(x, y) = 0$. Грубо говоря, $\mu y f(x, y)$ есть наименьшее y такое, что $f(x, y) = 0$. Для того чтобы придать этому определению вид функционального уравнения, нужно ввести вспомогательную функцию $h(x, y) = y + \mu z (f(x, y+z) = 0)$. Функции f, g и h связаны следующими соотношениями:

$$g(x) = h(x, 0),$$

$$h(x, y) = \textbf{если } f(x, y) = 0 \textbf{ то } y \textbf{ иначе } h(x, y+1).$$

Таким образом, приходим к системе уравнений с двумя неизвестными функциями g и h. Кроме решения $g(x) = \mu y (f(x, y) = 0)$ и $h(x, y) = y + \mu z (f(x, y+z) = 0)$ рассмотренная система может иметь и другие решения. Например, функция $\varphi(x, y)$ такая, что $\varphi(x, y) = h(x, y)$, если $h(x, y)$ определено, и $\varphi(x, y) = a$, если $h(x, y)$ не определено, удовлетворяет второму уравнению. Правда, функция $\varphi(x, y)$, вообще говоря, не является вычислимой. С другой стороны, основное вычислимое решение (g, h) является наименьшим в том смысле, что любое другое решение продолжает основное.

Рассмотренные примеры подсказывают естественное обобщение, состоящее в рассмотрении систем функциональных уравнений вида

$$f_i(x_1, \ldots, x_n) = F_i(f_1, \ldots, f_m, x_1, \ldots, x_n), \quad i = 1, \ldots, m, \tag{1.1}$$

где F_i — выражения, построенные из аргументов $x_1, \ldots, x_n$, символов неизвестных функций $f_1, \ldots, f_m$, констант и операций, заданных на области D, которую пробегают аргументы. Такого вида уравнения часто встречаются на практике. Для их изучения удобно использовать теорему о неподвижной точке, доказанную в § 9 гл. 1.

Переходя к общему случаю, рассмотрим сначала область D_0, на которой задана система базовых операций с сигнатурой Ω (алгебра данных). Нас будут интересовать частично определенные функции $f \subset D_0^n \to D_0$. Пример функции **если ... то ... иначе ...** показывает, что рассматриваемые функции могут быть определены и при неопределенных значениях некоторых аргументов. Для того чтобы формализовать эту ситуацию, удобно расширить область D_0 до области $D = D_0 \cup \{w\}$, добавив к ней неопределенный элемент w. Тогда частичная функция над D_0 будет моделироваться всюду определенной функцией над D. При этом равенство $f(x) = w$ интерпретируется как то, что $f(x)$ не определено на наборе $x = (x_1, \ldots, x_n)$. Если же какие-либо из компонент вектора x равны w, функция f, первоначально заданная на D_0, может быть продолжена на эти наборы произвольным образом. Область D называется *стандартным расширением* области D_0.

На D определяется частичный порядок $\sqsubset$ такой, что $w \sqsubset x$ для любого элемента $x \in D$, а все элементы из D_0 попарно не сравнимы, т.е. из $x \sqsubset y$ и $x \neq w$ следует $x = y$. Отношение $\sqsubset$, имея в виду дальнейшие построения, будем называть *отношением аппроксимации* ($x \sqsubset y$ читается: "x аппроксимирует y" или "y продолжает x"). В стандартных расширениях областей данных отношение аппроксимации тривиально: неопределенный элемент аппроксимирует все, остальные могут аппроксимировать только себя. Однако в дальнейшем будут использоваться и другие конструкции областей данных с отношением аппроксимации, более богатым по сравнению со стандартным расширением. Другая интерпретация отношения $\sqsubset$ состоит в том, что это отношение вводит степени определенности объектов области D. В этом случае, если $x \sqsubset y$, говорят, что x менее определен, чем y, или равен ему.

Итак, рассмотрим произвольную алгебру данных D с отношением аппроксимации $\sqsubset$, относительно которого будем предполагать только, что это отношение есть частичный порядок с нулевым (наименьшим) элементом w. Элемент w и в общем случае будем называть *неопределенным*, а при рассмотрении функций на D в случае, когда $\varphi(x) = w$, будем говорить, что φ не определена на x. Важным свойством отношения аппроксимации является индуктивность, т.е. существование в множестве D наименьшей верхней грани произвольной возрастающей цепочки элементов. Если D индуктивно, то наименьшую верхнюю грань цепочки $x_1 \sqsubset x_2 \sqsubset \ldots$ будем обозначать через $\bigsqcup_{i=1}^{\infty} x_i$.

Рассмотрим некоторые конструкции, сохраняющие индуктивность. Если $D_1, \ldots, D_n$ — индуктивные частично упорядоченные множества с нулями, то $D_1 \times \ldots \times D_n$ также является индуктивным частично упоря-

доченным множеством с нулем $(w,\ldots,w)$ относительно порядка, определенного отношением $(x_1,\ldots,x_n) \sqsubset (x_1',\ldots,x_n') \Longleftrightarrow x_1 \sqsubset x_1', \ldots, x_n \sqsubset x_n'$. Действительно, $\bigsqcup_{k=1}^{\infty} (x_1^{(k)},\ldots,x_n^{(k)}) = (\bigsqcup_{k=1}^{\infty} x_1^{(k)},\ldots,\bigsqcup_{k=1}^{\infty} x_n^{(k)})$. В области $D_1 \times \ldots \times D_n$ кроме неопределенного элемента $(w,\ldots,w)$ возможны также частично определенные элементы $(x_1,\ldots,x_n)$, для которых некоторые из x_i (но не все) равны w. Здесь возможны длинные возрастающие цепочки $x^{(1)} \sqsubset x^{(2)} \sqsubset \ldots$, состоящие из различных элементов, но всякая бесконечная цепочка должна стабилизироваться через конечное число шагов, поскольку элементы, не содержащие w, могут аппроксимировать только самих себя. Если D_1 и D_2 индуктивны, то множество $D_2^{D_1}$ всех отображений из D_1 в D_2 также индуктивно. Частичный порядок определяется следующим образом. Если $\varphi_1 : D_1 \to D_2$ и $\varphi_2 : D_1 \to D_2$, то $\varphi_1 \sqsubset \varphi_2 \Longleftrightarrow$ для всех $x \in D_1$ имеет место $\varphi_1(x) \sqsubset \varphi_2(x)$. Наименьшим элементом служит нигде не определенная функция φ_w, т.е. такая функция, что $\varphi_w(x) = w$ для любого $x \in D_1$. Наименьшая верхняя грань цепочки $\varphi_1 \sqsubset \varphi_2 \sqsubset \ldots$ есть функция φ такая, что $\varphi(x) = \bigsqcup_{i=1}^{\infty} \varphi_i(x)$, $x \in D_1$. Если D_1 и D_2 — стандартные расширения некоторых областей, то отношение $\varphi_1 \sqsubset \varphi_2$ означает, что функция φ_2 есть продолжение функции φ_1, т.е. из того, что $\varphi_1(x) \neq w$, следует, что $\varphi_2(x) = \varphi_1(x)$.

Функции типа $D^n \to D$ будем называть D-функциями. Из базовых D-функций можно образовывать новые функции с помощью суперпозиции, т.е. определений вида

$$f(x_1,\ldots,x_n) = F(x_1,\ldots,x_n),$$

где $F(x_1,\ldots,x_n)$ — выражение (терм), построенное из базовых функций и символов переменных $x_1,\ldots,x_n$. Суперпозиции базовых D-функций будем называть *элементарными D-функциями*. Таким образом, всякая элементарная D-функция есть либо селекторная функция $f(x_1,\ldots,x_n) = x_i$, либо базовая константа $f(x_1,\ldots,x_n) = a$, где a есть 0-арная базовая функция, либо может быть представлена в виде $f(x_1,\ldots,x_n) = g(h_1(x_1,\ldots,x_n),\ldots,h_m(x_1,\ldots,x_n))$, где g — базовая, а $h_1,\ldots,h_m$ — элементарные базовые функции, построенные раньше.

Важными свойствами D-функций являются монотонность и непрерывность, которые определяются через отношение аппроксимации на множестве D. Эти свойства определяются для функций типа $f : D_1 \to D_2$ и произвольных областей данных D_1 и D_2. Функция f монотонна, если из $x \sqsubset y$ следует $f(x) \sqsubset f(y)$ для произвольных элементов $x, y \in D_1$. Неопределенность определяется для индуктивных областей D_1 и D_2. Функция f непрерывна, если для любой возрастающей цепочки $x_1 \sqsubset x_2 \sqsubset \ldots$ элементов области D_1 имеет место $f(\bigsqcup_{i=1}^{\infty} x_i) = \bigsqcup_{i=1}^{\infty} f(x_i)$. Рассматривая n-арную D-функцию как отображение области D^n в D, получим понятие монотонности и непрерывности D-функций. Если зафиксировать значения всех аргументов функции $f(x_1,\ldots,x_n)$, кроме некоторого одного, получим функцию из D в D. Если эта функция монотонна или непрерывна, то говорят о

монотонности или непрерывности функции f по соответствующему аргументу. Нетрудно видеть, что D-функция монотонна и (непрерывна) тогда и только тогда, когда она монотонна (непрерывна) по каждому из своих аргументов. Действительно, если f монотонна по каждому аргументу, то $(x_1, \ldots, x_n) \sqsubset (x_1', \ldots, x_n') \Rightarrow f(x_1, \ldots, x_n) \sqsubset f(x_1', x_2, \ldots, x_n) \sqsubset \ldots$ $\ldots \sqsubset f(x_1', \ldots, x_n')$. Если же f непрерывна по каждому аргументу, то $f(\bigsqcup_{k=1}^{\infty} x_1^{(k)}, \ldots, \bigsqcup_{k=1}^{\infty} x_n^{(k)}) = \bigsqcup_{k=1}^{\infty} f(x_1^{(k)}, \ldots, x_n^{(k)})$, поскольку $(x_1^{(k_1)}, \ldots$ $\ldots, x_n^{(k_n)}) \sqsubset (x_1^{(k)}, \ldots, x_n^{(k)})$, где $k = \max\{k_1, \ldots, k_n\}$.

Заметим также, что из непрерывности следует монотонность. Действительно, $x_1 \sqsubset x_2 \Rightarrow f(x_2) = f(x_1 \sqcup x_2) = f(x_1) \sqcup f(x_2) \Rightarrow f(x_1) \sqsubset f(x_2)$. С другой стороны, если D – стандартное расширение некоторой области данных, то всякая монотонная D-функция непрерывна. Действительно, если $x_1 \sqsubset x_2 \sqsubset \ldots$ – возрастающая цепочка элементов области D, то либо $x_1 =$ $= x_2 = \ldots = \bigsqcup_{i=1}^{\infty} x_i = w$, либо $x_1 = \ldots = x_k = w, x_{k+1} = x_{k+2} = \ldots = \bigsqcup_{i=1}^{\infty} x_i = a$. В первом случае $f(\bigsqcup_{i=1}^{\infty} x_i) = \bigsqcup_{i=1}^{\infty} f(x_i) = f(w)$, во втором $f(\bigsqcup_{i=1}^{\infty} x_i) =$ $= f(a) = f(w) \sqcup f(a) = \bigsqcup_{i=1}^{\infty} f(x_i)$.

Суперпозиция $f(x_1, \ldots, x_n)$ монотонных (непрерывных) D-функций также монотонна (непрерывна), что доказывается индукцией по длине выражения $F(x_1, \ldots, x_n)$, которое определяет значение $f(x_1, \ldots, x_n)$ через $x_1, \ldots, x_n$ и заданные D-функции. Действительно, селекторные функции и константы, очевидно, монотонны и непрерывны. Далее, если $f(x_1, \ldots$ $\ldots, x_n) = f(x) = g(h_1(x), \ldots, h_m(x)) = g(h(x))$, где $g, h_1, \ldots, h_m$ – монотонные (непрерывные) функции, то $x \sqsubset x' \Rightarrow h(x) \sqsubset h(x') \Rightarrow g(h(x)) \sqsubset$ $\sqsubset g(h(x'))$ и $f(h(\bigsqcup_{k=1}^{\infty} x^{(k)})) = f(\bigsqcup_{k=1}^{\infty} h(x^{(k)})) = \bigsqcup_{k=1}^{\infty} f(h(x^{(k)}))$. В частности, если базовые D-функции монотонны (непрерывны), то и элементарные D-функции также монотонны (непрерывны).

При построении стандартных расширений областей данных возникает вопрос о продолжении операций. Как правило, эти продолжения должны быть монотонными. Простейший способ обеспечить монотонность состоит в следующем. Операция $\omega(x_1, \ldots, x_n)$ продолжается таким образом, что если хотя бы один из аргументов не определен, то значение $\omega(x_1, \ldots, x_n)$ также не определено, т.е. равно w. Такое продолжение называется *стандартным*. Возможны и другие, не стандартные монотонные продолжения. Например, если для любого продолжения $(y_1, \ldots, y_n)$ набора $(x_1, \ldots, x_n)$ до определенного функция $\omega(y_1, \ldots, y_n)$ принимает одно и то же значение a, то можно положить $\omega(x_1, \ldots, x_n) = a$. Именно таким образом были построены продолжения пропозициональных функций в алгебре условий.

С помощью базовых D-функций можно строить не только новые D-функции, но также и функционалы над D, т.е. функции, аргументы и зна-

чения которых являются D-функциями. Пусть $\Phi_n \subset D^{D^n}$ – множество всех непрерывных D-функций n аргументов. Функционал $\tau\colon \Phi_n^m \to \Phi_n^k$ назовем *элементарным функционалом* над D, если $\tau(f_1,\ldots,f_m)(x) = \tau(f,x) = (F_1(f,x),\ldots,F_k(f,x)) = F(f,x)$, где $F_1(f,x),\ldots,F_k(f,x)$ – выражения, построенные из символов предметных переменных с помощью базовых функций и символов переменных функций $f_1,\ldots,f_m$.

Теорема 1.1. *Пусть D – индуктивная область данных. Если базовые D-операции непрерывны, то элементарные функционалы над D также непрерывны.*

Действительно, пусть $\tau\colon \Phi_n^m \to \Phi_n^k$ – элементарный функционал и $\tau(f)(x) = F(f,x)$. Тогда $\tau(\bigsqcup_{l=1}^{\infty} f^{(l)})(x) = (F_1(\bigsqcup_{l=1}^{\infty} f^{(l)},x),\ldots,F_k(\bigsqcup_{l=1}^{\infty} f^{(l)},x)) = (\bigsqcup_{l=1}^{\infty} F_1(f^{(l)},x),\ldots,\bigsqcup_{l=1}^{\infty} F_k(f^{(l)},x)) = \bigsqcup_{l=1}^{\infty}(F(f^{(l)},x)) = \bigsqcup_{l=1}^{\infty}\tau(f^{(l)})(x)$, откуда $\tau(\bigsqcup_{l=1}^{\infty} f^{(l)}) = \bigsqcup_{l=1}^{\infty}\tau(f^{(l)})$.

Понятие элементарного функционала позволяет перейти к рассмотрению функциональных уравнений вида (1.1), правые части которых построены из базовых переменных $x_1,\ldots,x_n$ с помощью базовых и неизвестных D-функций.

Теорема 1.2. *Пусть D – индуктивная область данных, а базовые D-функции непрерывны. Тогда система уравнений* (1.1) *имеет наименьшее решение, которое определяется следующими соотношениями:*

$$f_i = \bigsqcup_{k=0}^{\infty} f_i^{(k)},$$

$$f_i^{(0)}(x) = w, \qquad x \in D^n,$$

$$f_i^{(k+1)}(x) = F_i(f_1^{(k)},\ldots,f_m^{(k)},x), \qquad x \in D^n.$$

Действительно, с уравнением (1.1) связан элементарный функционал $\tau\colon \Phi_n^m \to \Phi_n^m$, определенный на индуктивном частично упорядоченном множестве Φ_n^m с нулем с помощью соотношений $\tau(f)(x) = (F_1(f,x),\ldots,F_m(f,x))$. Каждое решение уравнения (1.1) является неподвижной точкой этого функционала. Поскольку τ непрерывен, можем применить теорему о неподвижной точке § 9 гл. 1. Получим, что наименьшая неподвижная точка этого функционала определяется как объединение итераций нулевого элемента (нигде не определенной функции): $f = \bigsqcup_{k=0}^{\infty}\tau^{(k)}(w) = \bigsqcup_{k=0}^{\infty} f^{(k)}$. Поскольку $f^{(k+1)}(x) = \tau(f^{(k)})(x) = F(f^{(k)},x)$, получаем утверждение теоремы.

Система функциональных уравнений (1.1) называется также *канонической системой* функциональных уравнений в алгебре D. Ее можно рассматривать как систему рекурсивных определений функций $f_1,\ldots,f_m$,

имея в виду, что эти функции определяются как компоненты наименьшей неподвижной точки функционала τ или как наименьшее решение системы (1.1). Функции, которые могут быть определены таким образом, называются алгебраическими D-функциями.

Конструктивность алгебраических D-функций можно подтвердить следующими рассуждениями. Предположим сначала, что D есть стандартное расширение некоторой области данных. D-функция f называется *вычислимой,* если существует алгоритм, который по значению аргументов $x_1, \ldots, x_n$ находит значение $f(x_1, \ldots, x_n)$, если оно определено, т.е. алгоритм останавливается и дает результат для всех случаев, когда $f(x_1, \ldots, x_n) \neq w$. Это неформальное определение зависит, разумеется, от способа представления наборов $(x_1, \ldots, x_n)$ состояниями информационной среды, над которой работает алгоритм, и может быть сделано точным с использованием любого из известных понятий алгоритма.

Если базовые D-функции вычислимы и монотонны, то алгебраические D-функции также вычислимы. Покажем сначала, что элементарные функции вычислимы. Для селекторных функций и констант это очевидно. Пусть $f(x_1, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n))$, где $g, h_1, \ldots, h_m$ – монотонные вычислимые функции. Алгоритм вычисления значения $f(x_1, \ldots, x_n)$ можно тогда построить следующим образом. Запустим одновременно $m+1$ вычислительных процессов. Первые m процессов вычисляют значения $h_1, \ldots, h_m$, а $m+1$-й – значение $g(y_1, \ldots, y_n)$ для $y_1 = \ldots = y_n = w$. Следим за всеми процессами. Если какой-либо из первых m процессов завершился и получено значение, скажем, $h_i(x_1, \ldots, x_n)$, то прерываем процесс вычисления $g(y_1, \ldots, y_n)$, заменяем y_i на полученное значение $h_i(x)$ и запускаем этот процесс сначала с новыми значениями $y_1, \ldots, y_n$. Если процесс вычисления $g(y_1, \ldots, y_n)$ завершился, прекращаем все остальные процессы и берем в качестве результата значение $g(y_1, \ldots, y_n) \neq w$. Монотонность функции g гарантирует правильность полученного результата.

Указанный метод вычисления суперпозиции можно также реализовать с помощью последовательного алгоритма, который моделирует все $m+1$ параллельных процессов. Алгоритм вычисления значения алгебраической D-функции требует последовательного запуска неограниченного числа параллельных процессов вычисления элементарных D-функций $f_1^{(k)}(x), \ldots, f_m^{(k)}(x)$, аппроксимирующих компоненты наименьшего решения $f_1, \ldots, f_m$ для $k = 0, 1, 2, \ldots$. Если один из этих процессов для интересующей нас компоненты завершится, то полученное значение и есть искомое. Все запущенные к этому времени процессы можно остановить и получить результат.

Если отношение аппроксимации устроено более сложно, чем в стандартном расширении, то конструктивность алгебраических D-функций приобретает более сложный характер и требует пересмотра самого понятия вычислимости. Дело в том, что в отличие от стандартного расширения, где на каждом шаге значение любой вычисляемой функции либо полностью не определено, либо полностью определено, в общем случае возможны различные степени определенности и предельные значения, которые нельзя получить за конечное число шагов.

В заключение отметим, что частично рекурсивные функции можно получить как алгебраические D-функции, где в качестве области D берется стандартное расширение множества $N = \{0, 1, 2, \ldots\}$ натуральных чисел, а в качестве базовых функций $-0, x+1$ и **если** $x = 0$ **то** y **иначе** z. Монотонное продолжение функции $x+1$ определяется так, что $w+1 = w$. Для функции **если** ... **то** ... **иначе** ... справедливы равенства (**если** $w = 0$ **то** y **иначе** z) $= w$, (**если** $0 = 0$ **то** y **иначе** z) $= y$, (**если** $x = 0$ **то** y **иначе** z) $= z$, если $x \neq w$ и $x \neq 0$.

Упражнения

1. Описать в каком-либо конкретном алгоритмическом базисе сеть из алгоритмических модулей для вычисления суперпозиции D-функций.

2. Доказать, что наименьшее решение уравнения

$f(x)$ = **если** $x > 100$ **то** $x - 10$ **иначе** $f(f(x+11))$,

рассматриваемого в области целых чисел, есть функция

$f(x)$ = **если** $x > 100$ **то** $x - 10$ **иначе** 91.

§ 2. Анализ схем программ над памятью

Естественно ожидать, что функции, вычислимые детерминированными схемами программ над памятью, интерпретированными с помощью стандартной интерпретации, определенной Ω – П-алгеброй данных D, будут алгебраическими D-функциями. В этом параграфе будет получен ответ на вопрос достаточно ли базовых функций, определенных сигнатурой $(\Omega, \text{П})$, или множество базовых функций необходимо расширить, а если да, то как.

Прежде всего заметим, что, рассматривая схемы на памятью, мы допускаем неопределенные значения переменных, откуда следует, что вычисление значений термов и условий должно быть определено и для неопределенных значений некоторых переменных. Поэтому естественно предполагать, что неопределенное значение w рассматривается как элемент области D. Это предположение не противоречит построениям, рассмотренным в связи с понятием схемы программы над памятью. Предположим также, что на D задано отношение аппроксимации $\sqsubset$, неопределенный элемент есть наименьший элемент этого отношения, а все функции и предикаты сигнатуры $(\Omega, \text{П})$ монотонны относительно аппроксимации. Для того чтобы иметь дело только с D-функциями, удобно также предполагать, что истинностные значения 0 и 1 входят в область D.

В схемах программ кроме базовых условий разрешается пользоваться элементарными условиями, которые получаются путем применения произвольных пропозициональных функций к базовым условиям. Для частичных интерпретаций, которые теперь допускаются, необходимы соглашения о монотонных продолжениях произвольных пропозициональных функций. Проще всего предположить, что при переходе к интерпретации допускаются любые монотонные продолжения пропозициональных функций. При этом, однако, требуется, чтобы для каждого элементарного условия в неинтерпретированной схеме были явно указаны базовые условия, от которых зависит данное элементарное условие (не обязательно

существенно). Действительно, функция 1, например, имеет единственное монотонное продолжение, а равная 1 фукнция $x \vee \bar{x}$ имеет два монотонных продолжения. Таким образом, зафиксировав интерпретацию базовых предикатов П, необходимо расширить это множество до множества $\hat{\Pi}$ предикатов, которые получаются путем применения к базовым предикатам произвольных монотонных продолжений произвольных пропозициональных функций. Элементы множества $\hat{\Pi}$ рассматриваются как D-функции, принимающие значений 0, 1 и w. Частичные предикаты (условия) $u_1(x_1, \ldots, x_n)$ и $u_2(x_1, \ldots, x_n)$ называются *несовместимыми*, если из того, что $u_1(x) \neq w$ и $u_2(x) \neq w$, следует $u_1(x) \wedge u_2(x) = 0$.

Пусть $u_1(x), \ldots, u_m(x)$ – попарно несовместные условия, зависящие от переменных $x_1, \ldots, x_n$, пробегающих область D ($x = (x_1, \ldots, x_n)$). Рассмотрим $n + m$-арную функцию $\varphi(x_1, \ldots, x_n, x'_1, \ldots, x'_m) = u_1(x)/x'_1 \vee \ldots \vee u_m(x)/x'_m$, равную x'_i, если $u_i(x) = 1$, и равную w, если ни одно из условий $u_1(x), \ldots, u_m(x)$ не равно 1. В силу попарной несовместности условий $u_1, \ldots, u_m$ значение функции φ определено однозначно. Функция вида $u_1(x)/x'_1 \vee \ldots \vee u_m(x)/x'_m$ называются *функциями параллельного выбора.* Если $u(x)$ и $\overline{u(x)}$ интерпретированы таким образом, что для любых x они одновременно определены или не определены, то функцию $u(x)/x' \vee \overline{u(x)}/x''$ можно заменить последовательным выбором (**если** $u(x)$ **то** x' **иначе** x''). Вообще, если попарно несовместные условия $u_1(x), \ldots, u_m(x)$ одновременно все определены или все не определены, то $u_1(x)/x'_1 \vee \ldots \vee u_m(x)/x'_m =$ **если** $u_1(x)$ **то** x'_1 **иначе если** $u_2(x)$ **то** x'_2 **иначе** ... **иначе если** $u_m(x)$ **то** x'_m **иначе** w.

Обозначим через $F(\Omega, \Pi)$ множество всех функций параллельного выбора, которые можно построить с помощью условий из множества $\hat{\Pi}$. В качестве базовых D-функций возьмем множество $\Omega \cup F(\Omega, \Pi)$.

Пусть A есть детерминированная $U - Y$-схема программы над конечной памятью $V = \{v_1, \ldots, v_n\}$, интерпретированная с помощью Ω-П-алгебры D с отношением аппроксимации. Поскольку память конечна, а $w \in D$, то ее состояние b: $V \to D$ можно отождествить с конечным вектором $(b(v_1), \ldots, b(v_n))$, а частичное отображение f_A со всюду определенным отображением f_A: $D^n \to D^n$, отождествляя неопределенное значение $f_A(x)$ с вектором $w = (w, \ldots, w)$. Каждая компонента y_j вектора $(y_1, \ldots, y_n) = f_A(x_1, \ldots, x_n) = f_A(x)$ есть D-функция, которую обозначим через $f_{Aj}(x)$ и будем называть D-функцией, вычисляемой схемой программы A.

Теорема 2.1 (вторая теорема анализа схем программ). *Всякая D-функция, вычисляемая детерминированной схемой программы над конечной памятью, интерпретированной с помощью Ω – П-алгебры D, является алгебраической D-функцией.*

В силу построений, использованных в доказательстве первой теоремы анализа схем программ (теорема 1.1 гл. 2), имеем

$$f_A = f_{a(0)},$$

где $f_{a(0)}$ определяется как соответствующая компонента наименьшего

решения $(f_a)_{a\in A}$ системы уравнений

$$f_a = \bigcup_{a'\in A} f_{aa'} \circ f_{a'}, \quad a \in A, \ a \neq a^{(1)},$$

$$f_{a^{(1)}} = \epsilon.$$

Заметим, что отношение аппроксимации, которое использовалось при доказательстве первой теоремы анализа, на множестве однозначных функций есть отношение продолжения. Это отношение является более сильным, чем отношение аппроксимации на множестве функций, индуцированное отношением аппроксимации на D.

Действительно, если f' есть продолжение f, то $f \sqsubset f'$, поскольку из $f(x) \neq w$ следует, что $f'(x) = f(x)$. Поэтому наименьшее решение в смысле продолжения остается наименьшим и в смысле аппроксимации. Пусть $a \xrightarrow{u_1/y_1} a_1, \ldots, a \xrightarrow{u_m/y_m} a_m$ – все переходы, которые ведут из состояния $a \neq a^{(1)}$. Тогда уравнение для f_a после раскрытия скобок может быть переписано в следующем виде:

$$f_a = u_1/(y_1 \circ f_{a_1}) \cup \ldots \cup u_m/(y_m \circ f_{a_m}).$$

Обозначим через $f_{aj}(x)$ j-ю компоненту вектора $f_a(x)$. Принимая во внимание, что в силу детерминированности условия $u_1, \ldots, u_m$ попарно несовместны, получим

$$f_{aj}(x) = u_1(x)/f_{a_1 j}(y_1(x)) \vee \ldots \vee u_m(x)/f_{a_m j}(y_m(x)), \ j = 1, \ldots, n.$$

Если $y = (v_1 := t_1(v), \ldots, v_n := t_n(v))$, то $y(x) = (t_1(x), \ldots, t_n(x))$. Поэтому уравнения для f_{aj} являются уравнениями в алгебре D с базовыми функциями $\Omega \cup F(\Omega, \Pi)$. Будучи компонентами наименьшего решения, функции f_{aj} сами образуют наименьшее решение и, следовательно, являются алгебраическими D-функциями. Теорема доказана.

Вопрос о том, можно ли произвольную алгебраическую D-функцию вычислить схемой программы над памятью, интерпретированной с помощью алгебры D, более сложен и будет рассмотрен в дальнейшем после соответствующей подготовки.

Упражнения

1. Доказать, что все функции множества $F(\Omega, \Pi)$ вычислимы схемами программ над памятью, интерпретированными с помощью Ω – Π-алгебры D.

2. Показать, что многозначные функции, вычислимые с помощью недетерминированных схем программ, можно рассматривать как алгебраические 2^D-функции.

§ 3. Вычисление алгебраических D-функций

Рассмотрим снова каноническую систему функциональных уравнений

$$f_i(x_1, \ldots, x_n) = F_i(f_1, \ldots, f_m, x_1, \ldots, x_n), \quad i = 1, \ldots, m, \tag{3.1}$$

в алгебре D с индуктивным отношением аппроксимации и непрерывными операциями. Нас будут интересовать способы вычисления значений $f_i(x_1, \ldots, x_n)$ в предположении, что известно, как вычислять элементарные D-функции. Общий метод вычисления содержится уже в самой формулировке основной теоремы 1.2 и описан в § 1. Этот метод будем назы-

вать *методом параллельного вычисления* последовательных приближений или просто *методом последовательных приближений*. На практике обычно базовые функции и отношение аппроксимации устроены значительно проще, чем в общем случае, что дает возможность искать более простые и эффективные алгоритмы вычисления значений алгебраических D-функций.

Собственно отыскание таких алгоритмов и представляет собой основную цель проектирования алгоритмов, поскольку построение канонической системы функциональных уравнений, или, что то же самое, отыскание рекурсивного определения для вычисляемых объектов, есть лишь первый шаг, устанавливающий существование алгоритма решения задачи или конструктивность ее постановки. Дальнейшее проектирование состоит в отыскании эффективных алгоритмов с использованием свойств алгебры данных D и функций, использованных в системе (3.1).

Методы, рассмотренные в этом параграфе, носят достаточно общий характер, но они могут служить исходной позицией для поиска эффективных решений.

Предположим, что отношение аппроксимации получено путем стандартного расширения области данных D_0, и, следовательно, из $x \sqsubset y$ и $x \neq w$ вытекает $x = y$. Другое предположение состоит в том, что для каждой базовой, а следовательно, и элементарной D-функции f существует алгоритм, который за конечное число шагов устанавливает, определено ли значение $f(x)$ для произвольного $x \in D$. Иными словами, элементарные функции являются общевычислимыми. Это предположение дает возможность освободиться от необходимости обязательной организации параллельных вычислений или их моделирования и вычислять значения функций $f_i^{(0)}(x)$, $f_i^{(1)}(x)$, ... последовательно. При этом, разумеется, возможность использования параллелизма для ускорения вычисления указанных значений остается, но она должна учитывать наличие реальных средств организации параллельных вычислений.

Другой метод вычислений, отличный от метода последовательных приближений, состоит в следующем. Вычисления производятся по шагам. На каждом шаге требуется вычислять значение выражения $G(f_1, \ldots, f_m, d_1, \ldots, d_n)$ в алгебре D с сигнатурой операций, расширенной символами неизвестных функций $f_1, \ldots, f_m$ арности n, для заданного набора $d_1, \ldots, d_n$ элементов области D. На первом шаге в качестве такого выражения выбирается $f_i(d_1, \ldots, d_n)$, где f_i — та из функций, определенных системой (3.1), значение которой необходимо вычислить. Заканчиваются вычисления в момент, когда выражение, полученное на очередном шаге, есть элемент области D. Шаг вычислений состоит из трех этапов:

1. Выделение вхождений неизвестных функций $f_1, \ldots, f_m$ в выражении G для подстановки.

2. Подстановка вместо подвыражений вида $f_j(E_1, \ldots, E_n)$, где f_j — выделенное вхождение, правых частей системы (3.1), т.е. выражений $F_j(f_1, \ldots, f_m, E_1, \ldots, E_n)$.

3. Упрощение полученного выражения. Если для некоторого подвыражения $E(f_1, \ldots, f_m)$ имеет место равенство $E(w, \ldots, w) = a \neq w$, то это выражение заменяется на a, а все подвыражения, не содержащие символов неизвестных функций, заменяются их значениями в D.

Первый этап шага вычислений – выделение вхождений – недетерминирован и разрешает выделить любую непустую совокупность вхождений. Формально это выделение состоит в замене вхождения символа f_i на его дубликат в другом алфавите. Будем обозначать этот дубликат $\overset{\uparrow}{f_i}$. Второй этап детерминирован. Его можно определить как переход от выражения G к выражению $\sigma(G)$, где σ – функция, определенная следующими соотношениями:

$$\sigma(z) = z, \quad z \in D,$$

$$\sigma(\omega(G_1, \ldots, G_n)) = \omega(\sigma(G_1), \ldots, \sigma(G_k)), \quad \omega \in \Omega,$$

$$\sigma(f_i(G_1, \ldots, G_n)) = f_i(\sigma(G_1), \ldots, \sigma(G_k)), \quad i = 1, \ldots, m,$$

$$\sigma(\overset{\uparrow}{f_i}(G_1, \ldots, G_n)) = F_i(f_1, \ldots, f_m, \sigma(G_1), \ldots, \sigma(G_k)), \quad i = 1, \ldots, m.$$

Третий этап выполняется по всем подвыражениям E выражения G. Если подвыражение E не содержит символов неизвестных функций, то просто вычисляется его значение в алгебре D, и это значение подставляется вместо E. Если $E = E(f_1, \ldots, f_m)$ содержит вхождения неизвестных функций, то эти вхождения вместе с их аргументами заменяются на w, и вычисляется значение полученного выражения, т.е. вычисляется $E(w, \ldots, w)$. Если полученное значение отлично от w, то E заменяется этим значением, в противном случае E остается нетронутым. Процесс упрощения повторяется до тех пор, пока в G не останется подвыражений, которые можно упростить. Очевидно, процесс упрощения всегда завершается, а результат не зависит от порядка рассмотрения подвыражений. Таким образом, третий этап также детерминирован. Результат упрощения выражения G обозначим через $val(E)$.

Описанный метод будем называть *методом преобразования D-выражений*. Его можно определить также с помощью дискретной системы $S(f, F)$. Состояниями этой системы являются выражения алгебры D в расширенной сигнатуре: к операциям сигнатуры Ω добавляются символы неизвестных функций из набора $f = (f_1, \ldots, f_m)$, а также символы констант – произвольных элементов области D. Отношение переходов $E \to E' \Leftrightarrow E'$ можно получить за один шаг при некотором выборе вхождений для подстановки. Система $S = S(f, F)$ является настроенной. Настройка определяется парой (S, D), т.е. начальные состояния – все D-выражения, а заключительные – выражения, которые сводятся к константам. Как обычно, многозначную функцию, вычисляемую системой S, обозначим через f_S. Пусть $\tilde{f} = (\tilde{f}_1, \ldots, \tilde{f}_m)$ – наименьшее решение системы (3.1), $G(f)$ – D-выражение.

Теорема 3.1. *Если* $f_S(G(f)) \neq \phi$, *то* $f_S(G(f)) = G(\tilde{f})$.

Действительно, легко видеть, что если $E(f) \to E'(f)$, то $E(f) = E'(f)$ для любых функций $f_1, \ldots, f_m$, которые образуют решение системы (3.1). Поскольку $\tilde{f}$ – одно из них, то отсюда следует утверждение теоремы. Из теоремы (3.1), в частности, следует, что, вообще говоря, недетерминированная система $S(f, F)$ глобально детерминирована.

Обратное утверждение также имеет место: если $G(\tilde{f})$ определено, то $f_S(G(f)) = G(\tilde{f})$, но доказательство его не так тривиально.

Рассмотрим некоторые полезные детерминированные подсистемы системы S. Каждая из этих подсистем получается путем детерминизации первого этапа шага вычислений, т.е. путем фиксирования некоторого алгоритма выбора вхождений для подстановки. Зафиксировав алгоритм выбора вхождений, получаем также алгоритм выполнения шага вычислений. Этот алгоритм называется *правилом вычислений.* Следующие правила вычислений, обозначаемые в дальнейшем символами $\sigma_1, \sigma_2, \ldots$, оказались наиболее интересными на практике.

σ_1 (правило вызова по значению). Для подстановки выбирается одно самое левое и самое внутреннее вхождение, т.е. такое вхождение символа неизвестной функции, что подвыражение, которое начинается этим вхождением, не содержит других вхождений неизвестных функций.

σ_2 (правило вызова по наименованию). Для подстановки выбирается самое левое и самое внешнее вхождение, т.е. такое, которое не содержится внутри подвыражения, начинающегося символом неизвестной функции.

σ_3 (параллельная внутренняя подстановка). Для подстановки выбираются все самые внутренние вхождения.

σ_4 (параллельная внешняя подстановка). Для подстановки выбираются все самые внешние вхождения.

σ_5 (полная подстановка). Для подстановки выбираются все вхождения.

Подсистему системы S, которая получается путем выбора правила вычисления ξ, обозначим через S_ξ. Правило ξ называется *корректным*, если из того, что $\tilde{f}_i(x)$ определено, вытекает, $f_{S_\xi}(f_i(x)) = \tilde{f}_i(x)$.

Простой пример показывает, что правила σ_1, σ_2 и σ_3 не являются корректными. Пусть $D_0 = Z$ – множество целых чисел. Рассмотрим монотонное продолжение операции умножения целых чисел, определенное следующими соотношениями:

$w \cdot 0 = 0 \cdot w = 0,$

$w \cdot x = x \cdot w = w, \; x \neq 0.$

Функциональное уравнение

$f(x, y) =$ **если** $x = 0$ **то** 0 **иначе** $f(x + 1, f(x, y)) \cdot f(x - 1, f(x, y))$ имеет наименьшее решение $f(x, y) = 0$. В то же время правила σ_1, σ_2 и σ_3 для $x \neq 0$ порождают бесконечные процессы вычислений, а правила σ_4 и σ_5 дают правильное решение. Выбор вхождений для подстановок, определяемый правилами $\sigma_1, \ldots, \sigma_5$, для правой части рассмотренного уравнения дает следующие результаты:

σ_1: **если** $x = 0$ **то** 0 **иначе** $f(x + 1, \underset{\uparrow}{f}(x, y)) \cdot f(x - 1, f(x, y))$;

σ_2: . . . . . $\underset{\uparrow}{f}(x + 1, f(x, y)) \cdot f(x - 1, f(x, y))$;

σ_3: . . . . . $f(x + 1, \underset{\uparrow}{f}(x, y)) \cdot f(x - 1, \underset{\uparrow}{f}(x, y))$;

σ_4: . . . . . $\underset{\uparrow}{f}(x + 1, f(x, y)) \cdot \underset{\uparrow}{f}(x - 1, f(x, y))$;

σ_5: . . . . . $\underset{\uparrow}{f}(x + 1, \underset{\uparrow}{f}(x, y)) \cdot \underset{\uparrow}{f}(x - 1, \underset{\uparrow}{f}(x, y))$.

Правила σ_4 и σ_5 оказываются корректными. Рассмотрим доказательство корректности правила σ_5. Сначала получим некоторые общие свойства полной функциональной подстановки.

Пусть $x = (x_1, \ldots, x_n)$ – набор символов переменных. Рассматриваемые дальше D-выражения, кроме констант, могут также содержать переменные из этого списка. Пусть $G = (G_1, \ldots, G_n)$ – набор D-выражений, H – некоторое D-выражение. Определим предметную подстановку $\sigma_x(G, H)$ с помощью следующих правил:

1) $\sigma_x(G, x_j) = val(G_j)$, $j = 1, \ldots, n$;

2) $\sigma_x(G, \varphi(H_1, \ldots, H_k)) = val(\varphi(\sigma_x(G, H_1), \ldots, \sigma_x(G, H_k)))$,

здесь φ – операция алгебры D, или неизвестная функция, или константа (0-арная операция, $k = 0$).

Пусть теперь $f = (f_1, \ldots, f_m)$ – набор символов неизвестных функций, $E = (E_1, \ldots, E_m)$ – набор D-выражений. Функциональная подстановка $\sigma_{f,x}(E, H) = \sigma_f(E, H)$ определяется по правилам:

1) $\sigma_f(E, z) = z$, если z – переменная или константа;

2) $\sigma_f(E, \omega(H_1, \ldots, H_k)) = val(\omega(\sigma_f(E, H_1), \ldots, \sigma_f(E, H_k)))$, $\omega \in \Omega$,

3) $\sigma_f(E, f_j(H_1, \ldots, H_k)) = val(\sigma_x((\sigma_f(E, H_1), \ldots, \sigma_f(E, H_k)), E_j))$, $j = 1, \ldots, m$.

Для функции val можно пользоваться следующим определением:

1) $val(z) = z$, z – переменная или константа;

2) $val(f_j(H_1, \ldots, H_n)) = f_j(val(H_1), \ldots, val(H_n))$, $j = 1, \ldots, m$;

3) $val(\omega(H_1, \ldots, H_k))$ = если $\sigma_f(w, \omega(val(H_1), \ldots, val(H_k))) = d \neq w$ то d иначе $\omega(val(H_1), \ldots, val(H_k))$.

Очевидно, что $\sigma_5(G) = \sigma_{f,x}(F, G)$, где $F = (F_1, \ldots, F_m)$ – набор правых частей системы (3.1). Функции σ_x, σ_f и val – это функции над D-выражениями, поэтому в пунктах определения этих функций равенство есть равенство D-выражений (исключением является равенство в условии правой части п. 3) определения функции val).

Более широкое понимание равенства D-выражений означает следующее. Равенство $H(f, x) = H'(f, x)$ истинно $\Leftrightarrow$ для набора D-функций $g = (g_1, \ldots, g_m)$ и любого набора $d = (d_1, \ldots, d_n)$ элементов множества D имеет место равенство значений $H(g, d) = H'(g, d)$. В дальнейшем, если не оговорено противное, равенство D-выражений понимается именно в этом смысле, т.е. как функциональное тождество в алгебре D.

Заметим, что в силу монотонности D-функций и из того, что $H(w, x) = d \neq w$, следует, что $H(f, x) = d$ тождественно. Поэтому из определения функции val вытекает, что $val(H) = H$ (строгое доказательство получается индукцией по числу операций в выражении H). Поскольку $H(g, d) = \sigma_x((d_1, \ldots, d_n), \sigma_f((g_1(x), \ldots, g_m(x)), H(f, x)))$, то из равенства $H = H'$ вытекают равенства $\sigma_x(G, H) = \sigma_x(G, H')$ и $\sigma_f(E, H) = \sigma_f(E, H')$. Индукцией по числу операций в H доказываются импликации $G = G' \Rightarrow \sigma_x(G, H) = \sigma_x(G', H)$ и $E = E' \Rightarrow \sigma_f(E, H) = \sigma_f(E', H)$. Комбинируя эти факты получим $G = G'$ и $H = H' \Rightarrow \sigma_x(G, H) = \sigma_x(G', H')$, $E = E'$ и $H = H' \Rightarrow \sigma_f(E, H) = \sigma_f(E', H')$. Действительно, $\sigma_x(G, H) = \sigma_x(G', H) = \sigma_x(G', H')$; аналогично для σ_f.

Теорема 3.2. *Правило* σ_5 *корректно.*

Доказательство использует следующие леммы, устанавливающие ряд полезных свойств функциональных и предметных подстановок.

Лемма 3.1. $\sigma_f(E, \sigma_x(G, H)) = \sigma_x((\sigma_f(E, G_1), \ldots, \sigma_f(E, G_n)), \sigma_f(E, H)$.

Лемма 3.2. $\sigma_f(E, \sigma_f(G, H)) = \sigma_f((\sigma_f(E, G_1), \ldots, \sigma_f(E, G_m)), H)$.

Обозначим через $\sigma_f^k(E, H)$ k-кратное применение подстановки E в H, т.е. $\sigma_f^0(E, H) = H$, $\sigma_f^{k+1}(E, H) = \sigma_f(E, \sigma^k(E, H))$.

Лемма 3.3. $\sigma_f^k(E, H) = \sigma_f((\sigma^k(E, f_1(x)), \ldots, \sigma^k(E, f_m(x))), H)$.

Лемма 3.4. $f_j^{(k)}(d) = \sigma_f(w, \sigma_f^k(F, f_j(d)))$.

Докажем теорему 3.2. Пусть $\tilde{f}_j(d) = a \neq w$. Тогда для некоторого k $\tilde{f}_j(d) = f_j^{(k)}(d) = a$. В системе S_{σ_5} имеем $f_j(d) \to F_j^{(1)} \to \ldots \to F_j^{(k)}$, где $F_j^{(i+1)} = \sigma(F, F_j^{(i)}) = \sigma^{i+1}(F, f_j(d))$ $(i = 0, 1, \ldots, k-1)$. Применяя лемму 4, получим $\sigma_f(w, F_j^{(k)}) = \sigma_f(w, \sigma_f^k(F, f_j(d))) = f_j^{(k)}(d) = a \Rightarrow F_j^{(k)} = a \Rightarrow f_{S_{\sigma_5}}(f_j(d)) = a = \tilde{f}_j(d)$. Теорема доказана.

Перейдем к доказательству лемм. Лемма 3.1 доказывается индукцией по числу операций в выражении H. Доказательство сводится к тривиальному разбору случаев определения подстановок и предоставляется читателю. Лемма 3.2 также доказывается индукцией по числу операций в H. Случаи, когда H есть константа, переменная или $H = \omega(H_1, \ldots, H_k)$, разбираются тривиально. Рассмотрим случай, когда $H = f_j(H_1, \ldots, H_k)$. Тогда $\sigma_f((\sigma_f(E, G_1), \ldots, \sigma_f(E, G_m)), H) = \sigma_f(\sigma_f(E, G), f_j(H_1, \ldots, H_k)) = \sigma_x((\sigma_f(\sigma_f(E, G), H_1), \ldots, \sigma_f(\sigma_f(E, G), H_k)), \sigma_f(E, G_j)) = \sigma_x((\sigma_f(E, \sigma_f(G, H_1)), \ldots, \sigma_f(E, \sigma_f(G, H_n))), \sigma_f(E, G_j))$ (предположение индукции) $= \sigma_f(E, \sigma_x((\sigma_f(G, H_1), \ldots, \sigma_f(G, H_n)), G_j))$ (лемма 1) $= \sigma_f(E, \sigma_f(G, H))$.

Лемма 3.3 доказывается индукцией по k. Для $k = 0$ доказательство очевидно. Шаг индукции $\sigma^{k+1}(E, H) = \sigma_f(E, \sigma^k(E, H)) = \sigma_f(E, \sigma_f((\sigma^k(E, f_1(x)), \ldots, \sigma^k(E, f_m(x))), H))$ (предположение индукции) $= \sigma_f(\sigma_f(E, \sigma^k(E, f_1(x))), \ldots, \sigma_f(E, \sigma^k(E, f_m(x))), H)$ (лемма 3.2) $= \sigma_f((\sigma_f^{k+1}(E, f_1(x)), \ldots, \sigma_f^{k+1}(E, f_m(x))), H)$.

Лемма 3.4 также доказывается индукцией по k. Для $k = 0$ доказательство очевидно. Шаг индукции $f_j^{(k+1)}(d) = \sigma_f((f_1^{(k)}(d), \ldots, f_m^{(k)}(d)), F_j) = \sigma_f((\sigma_f(w, \sigma_f^k(F, f_1(d))), \ldots, \sigma_f(w, \sigma_f^k(F, f_m(d)))), F_j)$ (предположение) $= \sigma_f(w, \sigma_f((\sigma^k(F, f_1(d)), \ldots, \sigma^k(F, f_m(d))), F_j)$ (лемма 3.2) $= \sigma_f(w, \sigma_f^k(F, F_j))$ (лемма 3) $= \sigma_f(w, \sigma^k(F, \sigma_f(F, f_j(d)))) = \sigma_f(w, \sigma_f^{k+1}(F, f_j(d)))$.

Поскольку система S_{σ_5} является подсистемой системы $S(f, F)$, то в качестве следствия из теорем 3.1 и 3.2 получаем теорему 3.3.

Теорема 3.3. $G(\tilde{f}) = a \neq w \Leftrightarrow f_S(G(f)) = a \neq w$.

Вообще говоря, система S дает несколько больше. В некоторых случаях она останавливается и дает в качестве верного результата и неопределенное значение. Однако, разумеется, возможны случаи, когда $G(\tilde{f}) = w$, а S не останавливается.

Правила σ_1, σ_2, σ_3, несмотря на некорректность, имеют практическое значение. Например, если доказано, что система S не имеет бесконечных процессов, начинающихся в состояниях вида $f_j(d)$, то любое из этих правил для соответствующей системы дает правильный результат. Простота реализации правил σ_1 и σ_2 привела к тому, что именно они чаще всего используются в системах программирования, допускающих функциональные рекурсивные определения.

Упражнение

Доказать корректность правила параллельной внешней замены. Указание: показать, что для этого правила справедливы леммы 3.1 и 3.2.

§ 4. Функционалы высших типов

Теорию функциональных уравнений, изложенную в § 1, нетрудно перенести на случай, когда D представляет собой многоосновную алгебру данных $D = (D_\xi)_{\xi \in \Xi}$. Для этого достаточно предположить, что каждое множество D_ξ частично упорядочено отношением аппроксимации с минимальным элементом w, общим для всех множеств, а операции алгебры D монотонны. Теперь D-функции различаются по типам. Тип D-функции, определенной на множестве $D_{\xi_1} \times \ldots \times D_{\xi_n}$ и принимающей значения в множестве D_ξ, будем обозначать $(\xi_1, \ldots, \xi_n) \to \xi$ (вместо обозначения $(\xi_1, \ldots, \xi_n, \xi)$, использованного в § 3 гл. 2 которое мы сохраним для обозначения типа декартова произведения). Каноническая система функциональных уравнений теперь имеет вид

$$f_i(x_{i1}, \ldots, x_{in_i}) = F_i(f_1, \ldots, f_m, x_{i1}, \ldots, x_{in_i}), \quad i = 1, \ldots, m, \qquad (4.1)$$

где F_i — выражения многоосновной алгебры D в расширенной сигнатуре, а символам неизвестных функций $f_1, \ldots, f_m$ и переменным $x_{i1}, \ldots, x_{in_i}$ приписаны соответствующие типы. Отношение аппроксимации переносится естественным образом на множество $\Gamma(D_{\xi_1} \times \ldots \times D_{\xi_n}, D_\xi) = D_\xi^{D_{\xi_1} \times \ldots \times D_{\xi_n}}$ всех функций из $D_{\xi_1} \times \ldots \times D_{\xi_n}$ в D_ξ, а правые части уравнений определяют элементарный функционал над D, т.е. отображение множества $\Gamma_1 \times \ldots \times \Gamma_m$ в себя, где $\Gamma_1, \ldots, \Gamma_m$ — множества функций соответствующих типов. Таким образом, для многоосновной области имеет место теорема 1.1, а также все остальные утверждения и построения § 1. Единственное изменение состоит в том, что при образовании термов должны быть выполнены условия согласования типов: если операция (функция) φ имеет тип $(\xi_1, \ldots, \xi_n) \to \xi$, то составляющие $t_1, \ldots, t_n$ терма $\varphi(t_1, \ldots, t_n)$ должны иметь типы $\xi_1, \ldots, \xi_n$ соответственно.

Из области D можно строить новые области данных, переходя от уже имеющихся областей к декартовым произведениям и областям вида $\Gamma(D_1, D_2)$. В случае многоосновной области D для образования новых областей используются ее компоненты D_ξ. Так возникает широко используемая в математической практике иерархия функциональных объектов. Более точно ее можно определить следующим образом. Расширим исходное множество типов Ξ до множества Ξ_1, полагая, что Ξ_1 есть наименьшее множество, содержащее Ξ и вместе с каждым набором типов $\xi_1, \ldots, \xi_n$

также типы $(\xi_1, \ldots, \xi_n)$ и $\xi_1 \to \xi_2$. Каждому типу $\xi \in \Xi_1$ поставим в соответствие область D_ξ следующим образом. Для $\xi \in \Xi$ области D_ξ уже определены. Будем называть эти области и соответствующие им типы *базовыми*. Типу $\xi = (\xi_1, \ldots, \xi_n)$ поставим в соответствие область $D_\xi = D_{\xi_1} \times \ldots \times D_{\xi_n}$, а типу $\xi_1 \to \xi_2$ — область $D_{\xi_1 \to \xi_2} = \Gamma(D_{\xi_1}, D_{\xi_2}) = D_{\xi_2} D_{\xi_1}$. Элементы области $D_{\xi_1 \to \xi_2}$ будем называть *функционалами типа* $\xi_1 \to \xi_2$ и, если ξ_1 или ξ_2 отличны от декартова произведения базовых типов, *функционалами высших типов*. Функционалы, построенные с помощью семейства D базовых областей, будем называть *D-функционалами* или *функционалами над D*.

Семейство всех функционалов будем обозначать через $\Phi(D) = (D_\xi)_{\xi \in \Xi_1}$. На области функционалов высших типов и на декартовы произведения переносится отношение аппроксимации, заданное на базовых областях. Этот перенос делается индукцией по длине выражения, определяющего тип ξ рассматриваемой области. Пусть на областях $D_{\xi_1}, \ldots, D_{\xi_n}$ отношение аппроксимации уже определено. Тогда, если $\xi = (\xi_1, \ldots, \xi_n)$, отношение аппроксимации на D_ξ определяется условием $(d_1, \ldots, d_n) \sqsubset (d'_1, \ldots, d'_n) \Leftrightarrow d_1 \sqsubset d'_1, \ldots, d_n \sqsubset d'_n$. Если же $\xi = (\xi_1 \to \xi_2)$, то $f \sqsubset g\,(f, g \in D_{\xi_1 \to \xi_2}) \Leftrightarrow$ для любого $x \in D_{\xi_1}$ имеет место $f(x) \sqsubset g(x)$.

Поскольку D_ξ состоит из всех объектов типа ξ, то из индуктивности базовых областей следует индуктивность областей D_ξ для $\xi \in \Xi_1$.

Прикладное значение функциональной иерархии состоит в том, что с ее помощью удобно описывать структуры данных высших типов, а отношение аппроксимации позволяет исследовать вопросы вычислимости. Помимо функциональной иерархии в математике используются и другие способы построения сложных структурных объектов, которые используются для описания (определения) сложных структур данных. В частности, важную роль играет теоретико-множественная иерархия, позволяющая конструировать новые множества из уже построенных с помощью декартовых произведений и образования множества всех подмножеств. Теоретико-множественная и функциональная иерархия легко моделируют одна другую, поскольку множество можно заменить характеристической функцией, а функцию — ее графиком. Однако с практической точки зрения во многих случаях удобно различать множества и функции. Теоретико-множественную и функциональную иерархии удобно также использовать для построения состояний сложных динамических систем.

Для того чтобы говорить о вычислимости функционалов высших типов, на компонентах семейства $\Phi(D) = (D_\xi)_{\xi \in \Xi_1}$ необходимо определить базовые операции. Помимо операций, связанных со спецификой базовых областей данных, есть целый ряд полезных операций, которые могут быть определены над D-функционалами независимо от природы области D. Рассмотрим некоторые из них.

Наиболее ясной и естественной операцией является операция применения функции к своему аргументу, называемая иногда *аппликацией*. Эта операция имеет тип $(\xi \to \tau, \xi) \to \tau$. Точнее говоря, для каждой пары типов ξ и τ определяется своя операция применения соответствующего типа. Но, как обычно, для всех этих операций используется одна и та же синтаксическая схема $()(())$, и, следовательно, результат применения этой операции

обозначается для всех типов одинаково: $f(x)$. Операция аппликации монотонна по первому аргументу: $f \sqsubset g \Rightarrow f(x) \sqsubset g(x)$. Она также непрерывна по первому аргументу, что следует из определения наименьшей верхней грани. Но по второму аргументу аппликация даже не монотонна, поскольку $f(x) \sqsubset f(x')$ следует из $x \sqsubset x'$ только для монотонных функций, но не всякая функция монотонна.

Другой общий метод образования операций на областях высших типов состоит в переносе на эти области операций, определенных для базовых областей. Например, пусть ω есть операция типа $\xi \to \eta$. Тогда можно рассмотреть новую операцию ω' типа $(\zeta \to \xi) \to (\zeta \to \eta)$, полагая $(\omega'(f))(x) = \omega(f(x))$ для любого $x \in D_\zeta$. Относительно операции ω' можно говорить, что она получается путем переноса операции ω на область $D_{\zeta \to \xi}$. Поскольку операции любого типа являются функциями, их можно рассматривать как константы соответствующего типа и выделять в соответствующих областях как 0-арные операции. Поскольку каждая из областей $D_\xi, \xi \in \Xi_1$ частично упорядочена отношением аппроксимации, можно говорить о пересечении и объединении элементов этой области. Вообще говоря, эти операции определены не всегда. Но если, например, в области D_η пересечения $x \sqcap y$ и объединения $x \sqcup y$ определены для любых пар (x, y), то они будут также определены и для функций типа $\xi \to \eta$, поскольку $(f \sqcap g)(x) = f(x) \sqcap g(x)$ и $(f \sqcup g)(x) = f(x) \sqcup g(x)$.

Уже говорилось о том, что базовые операции над D-функционалами не всегда монотонны (непрерывны). Поэтому обычно рассматривают подмножества областей D_ξ $(\xi \in \Xi_1)$, замкнутые относительно выбранной системы базовых операций, на которых эти операции оказываются монотонными (непрерывными), т.е. рассматривают подалгебру алгебры $\Phi(D)$ всех D-функционалов. Важным примером такой алгебры является алгебра всех непрерывных функционалов. Поскольку применение непрерывной функции к своему аргументу непрерывно по обоим аргументам, аппликация может рассматриваться как операция этой алгебры.

Пусть теперь $E = (E_\lambda)_{\lambda \in \Lambda}$ – произвольная алгебра D-функционалов, $\Xi \subset \Lambda \subset \Xi_1$, $E_\lambda \subset D_\lambda$. Выбор базовых операций определяет понятие элементарного D-функционала и, если базовые операции непрерывны, понятие алгебраического D-функционала, частным случаем которого является понятие алгебраической D-функции. Что же касается вычислимости алгебраических D-функционалов, то здесь встречаются две принципиальные трудности.

Первая трудность состоит в том, что область определения и область значений функционала могут содержать неконструктивные объекты, поскольку компоненты D_ξ $(\xi \in \Xi_1)$ могут иметь мощность континуума и выше; поэтому возникает вопрос о конструктивных способах задания аргументов и значений функционалов высших типов. Если область D конструктивна, т.е. любые ее элементы могут быть заданы выражениями в некотором языке, если также сигнатура базовых операций задана конструктивно, то конструктивно могут быть заданы все элементарные и алгебраические D-функционалы, поскольку для их задания могут использоваться D-выражения и канонические системы функциональных уравнений. Другие объекты могут задаваться как пределы возрастающих последовательностей элементарных или алгебраических функционалов. При

этом можно рассматривать и неконструктивные последовательности, учитывая, что их конечные начальные отрезки задаются конструктивно; и, увеличивая длину этих отрезков, можно получить любую степень приближения к предельному объекту. Поскольку алгебраические D-функционалы непрерывны, то, вычисляя значения заданного функционала на элементах последовательности, получим новую последовательность, предел которой дает значение функционала на предельном элементе заданной последовательности.

Вторая трудность состоит в следующем. Представим себе, что происходит вычисление значения алгебраического функционала f_1, заданного как первая компонента наименьшего решения $(f_1, \ldots, f_m)$ канонической системы уравнений (4.1), на элементе z. Для этого вычисляются последовательные приближения $f_i^{(k)}(x_i)$ функционалов f_i по формуле $f_i^{(k+1)}(x_i) = F_i(f^{(k)}, x_i)$. Последовательность

$$f_1^{(0)}(z) \sqsubset f_1^{(1)}(z) \sqsubset \ldots \tag{4.2}$$

сходится к результату $f_1(z)$. В случае обычных функций над областью D, заданной как стандартное расширение множества определенных значений, последовательность (4.2) либо вся состояла из неопределенного значения, либо стабилизировалась на конечном шаге. Теперь, когда значениями $f_i^{(k)}(z)$ могут быть функции, эта последовательность может состоять из различных элементов и сходиться в пределе к определенному значению $f_1(z)$. В этом случае остается удовлетвориться тем, что последовательность (4.2) сходится, и считать ее результатом вычислений. В случае, когда сам аргумент z задан как предел последовательности $z_0 \sqsubset z_1 \sqsubset \ldots$, в качестве результата можно рассматривать последовательность $f_1^{(0)}(z_0) \sqsubset \sqsubset f_1^{(1)}(z_1) \sqsubset \ldots$, которая также сходится к результату $f_1(z)$. Действительно, $f_1^{(k)}(z_k) \sqsubset f_1^{(k)}(z) \sqsubset f_1(z)$, т.е. $f_1(z)$ есть верхняя грань всех элементов $f_1^{(k)}(z_k)$. С другой стороны, если $f_1^{(k)}(z_k) \sqsubset u$ для всех k, то $f_1^{(k+l)}(z_k) \sqsubset f_1^{(k+l)}(z_{k+l}) \sqsubset u$ для всех k и l, откуда $f_1(z_k) \sqsubset u$ для всех u; следовательно, $f_1(z) \sqsubset u$, т.е. $f_1(z)$ есть наименьшая верхняя грань (предел последовательности $f_1^{(k)}(z_k)$ $(k = 0, 1, \ldots)$).

Таким образом, всегда можно говорить о приближенном вычислении алгебраических D-функционалов. Вопрос о конструктивности самих приближений зависит от того, каким образом заданы операции алгебры функционалов, и сводится к изучению конструктивности элементарных функций. На практике дело упрощается тем, что в большинстве случаев достаточно рассматривать элементарные функционалы с конечными областями определения, ограниченными некоторой заданной областью, а конструктивность базовых функций достаточна для распознавания стабилизации возрастающей последовательности таких функционалов. Такие функционалы достаточно хорошо описывают различные виды многоуровневых функциональных структур данных, используемых при проектировании алгоритмов, в частности, многоуровневые массивы и файлы с прямым доступом. В гл. 4 будут рассмотрены конкретные алгебры функционалов, для которых могут быть построены эффективные методы вычислений.

§ 5. Рекурсивные программы

Теория рекурсивных определений дает возможность точно определить понятие рекурсивного вызова программ (подпрограмм), которое играет большую роль в программировании. Начнем с определения *системы детерминированных схем программ.* Так будем называть семейство $(A_\lambda)_{\lambda \in \Lambda}$, членами которого являются U_λ-Y_λ-схемы программ и в котором выделена схема $A_{\lambda_0} = A_0$, называемая *основной*. Базис операторов Y_λ $(\lambda \in \Lambda)$ имеет определенную структуру: $Y_\lambda = Y_\lambda^{(0)} \cup \bigcup\limits_{\mu \in \Lambda} Y_{\lambda\mu}^{(0)}$. Операторы $y \in Y_\lambda^{(0)}$ называются *внутренними,* а операторы $y \in Y_{\lambda\mu}^{(0)}$ – *вызовами* программы A_μ из программы A_λ. Различные операторы вызова одной и той же программы могут различаться способами передачи параметров и возвращения результатов.

Интерпретация системы (A_λ) задается следующим образом. Для каждого индекса $\lambda \in \Lambda$ выбирается своя информационная среда B_λ, на которой интерпретируются элементы базиса (U_λ, Y_λ). Предполагается, что на множестве B_λ задано индуктивное отношение аппроксимации. Каждое условие $u \in U_\lambda$ интерпретируется как непрерывная функция $u: B_\lambda \to \{0, 1, w\}$ (частичный предикат), а каждый оператор $y \in Y_\lambda^{(0)}$ – как непрерывное преобразование $y: B_\lambda \to B_\lambda$ (для того чтобы упростить обозначения, мы отождествляем, например, символ оператора y с интерпретирующим его преобразованием f_y). Оператору вызова $y \in Y_{\lambda\mu}^{(1)}$ сопоставляются два непрерывных преобразования $y^{(0)}: B_\lambda \to B_\mu$ и $y^{(1)}: B_\lambda \times B_\mu \to B_\lambda$. Первое из этих преобразований соответствует передаче параметров, второе – передаче результатов после окончания работы вызванной программы. Сам оператор y интерпретируется как преобразование $y: B_\lambda \to B_\lambda$, которое определяется соотношением

$$y(b) = y^{(1)}(b, f_\mu(y^{(0)}(b))). \tag{5.1}$$

Преобразование $f_\mu: B_\mu \to B_\mu$ – это функция, вычисляемая схемой программы A_μ при заданной интерпретации. Функции f_λ находятся из системы уравнений, которые строятся по схемам программ обычным образом. Именно, пусть $a \xrightarrow{u_1/y_1} a_1, \ldots, a \xrightarrow{u_m/y_m} a_m$ – все переходы из состояния a схемы A_λ. Сопоставим этому состоянию уравнение

$$f_{a,\lambda}(b) = (u_1/y_1 \circ f_{a_1,\lambda} \vee \ldots \vee u_m/y_m \circ f_{a_m,\lambda})(b). \tag{5.2}$$

Для любой системы $u_1, \ldots, u_m$ попарно не пересекающихся условий на B_λ определена функция параллельного выбора $\varphi(b, b_1, \ldots, b_m) = u_1(b)/b_1 \vee \ldots \vee u_m(b)/b_m$, равная тому из элементов $b_1, \ldots, b_m$, для которого условие $u_i(b)$ истинно. Эта функция равна w, если все условия ложны или не определены. Очевидно, функция параллельного выбора непрерывна. Пользуясь этой функцией, уравнение (5.2) можно переписать в виде

$$f_{a,\lambda}(b) = u_1(b)/f_{a_1,\lambda}(y_1(b)) \vee \ldots \vee u_m(b)/f_{a_m,\lambda}(y_m(b)).$$

Подставляя для операторов $y \in Y_{\lambda\mu}^{(1)}$ вместо $y(b)$ правую часть равенства (5.1) и полагая $f_\lambda = f_{a_0,\lambda}$, где a_0 – начальное состояние схемы A_λ и $f_{a^{(1)},\lambda}(b) = b$ для заключительного состояния $a^{(1)}$ схемы A_λ, получим

окончательно интересующую нас систему уравнений для определения функций $f_\lambda (\lambda \in \Lambda)$. Если множество Λ конечно, то и система уравнений также конечна.

Семейство множеств $(B_\lambda)_{\lambda \in B}$ является многоосновной алгеброй с унарными операциями $y \in Y_\lambda^{(0)}$, $y^{(0)} \in Y_{\lambda\mu}^{(1)}$, бинарными операциями $y^{(1)} \in Y_{\lambda\mu}^{(1)}$ и операциями параллельного детерминированного выбора. Все эти операции непрерывны, и система уравнений вида (5.2) имеет наименьшее решение, которое определяет функции f_λ и, в частности, функцию $f_0 = f_{\lambda_0}$, вычисляемую основной схемой A_0.

Интерпретированная система схем программ, представленных в некотором языке программирования, называется также *схемой рекурсивных программ* или *программ с рекурсивными вызовами*. Иногда термин "программа" относят только к основной программе, а все другие члены семейства (A_λ) называют ее подпрограммами. Наконец, термин "рекурсивный" часто относят к тому случаю, когда некоторая программа, возможно, через цепочку вызовов обращается к самой себе. Таким образом, функциональная семантика рекурсивных программ построена.

Основным механизмом взаимодействия программ при вызовах является аппарат формальных и фактических параметров. Его можно описать в терминах схем программ над памятью. Каждый из базисов (U_λ, Y_λ) определяется как стандартный базис над памятью V_λ с помощью сигнатуры $(\Omega_\lambda, \Pi_\lambda)$ операций и условий, интерпретированных на области D_λ. Память V_λ может быть типизированной. В этом случае $V_\lambda = \bigcup_{\xi \in \Xi_\lambda} V_{\lambda\xi}$, $D_\lambda = \bigcup_{\xi \in \Xi_\lambda} D_{\lambda\xi}$. Допускается также использование косвенного именования.

Операторы $y \in Y_\lambda^{(0)}$ – это обычные операторы присваивания, а операторы вызова $y \in Y_{\lambda\mu}^{(1)}$ имеют следующую структуру. Для каждой программы A_μ заданы конечные последовательности In_μ и Out_μ соответственно входных и выходных параметров, составленные из переменных множества V_μ. Если $In_\mu = (u_1, ..., u_m)$, $Out_\mu = (v_1, ..., v_n)$, то вызов y может иметь синтаксическое представление вида $(s_1, ..., s_n) := P_\mu(t_1, ..., t_m)$, где $s_1, ..., s_n \in V_\lambda$, $t_1, ..., t_m$ – Ω_λ-термы над V_λ, P_μ – имя программы A_μ. В дальнейшем для экономии обозначений вместо P_μ будем употреблять A_μ. В случае типизированной памяти должны выполняться условия соответствия типов. Именно, области значений термов $t_1, ..., t_m$ должны содержаться в областях значений входных параметров $u_1, ..., u_m$, а области значений выходных параметров должны содержаться в областях значений переменных $s_1, ..., s_k$. В случае косвенного именования $s_1, ..., s_k$ – это термы, принимающие значения в классах имен, типы которых согласованы с типами выходных параметров.

Функции $y^{(0)}$ и $y^{(1)}$ определяются следующим образом. Если $b: V_\lambda \to D_\lambda$, то $y^{(0)}(b) = b_0 z$, где $b_0: V_\mu \to D_\mu$ – нигде не определенное состояние памяти, т.е. $b_0(v) = w$ для всех $v \in V_\mu$, а

$$z = (u_1 := b(t_1), ..., u_m := b(t_m)).$$

Пусть теперь $b: V_\lambda \to D_\lambda$, $b': V_\mu \to D_\mu$. Тогда $y^{(1)}(b, b') = bz$, где

$$z = (s_1 := b'(v_1), ..., s_n := b'(v_n)).$$

Система программ (интерпретированных схем) над памятью, для которой семантика вызовов определяется описанным выше способом, называется *системой программ с распределенной памятью.* Другая семантика вызовов используется для *систем программ над общей памятью.* Такие системы используют общий базис (U, Y), интерпретированный на одной и той же памяти V (все множества V_λ и B_λ совпадают). Функция $y^{(0)}$ эквивалентна оператору присваивания

$$(u_1 := t_1, ..., u_m := t_m),$$

а функция $y^{(1)}$ определяется так, что $y^{(1)}(b, b') = b'z$, где

$$z = (s_1 := v_1, ..., s_n := v_n).$$

На практике часто применяют смешанные варианты, разделяя всю память на части, используя некоторые из этих частей как общую память для одних программ, а некоторые локализуя внутри других. Структура памяти обычно определяется путем взаимного расположения текстов программ внутри общего текста, представляющего семейство. Для текстового представления схем программ можно использовать язык А2 со следующим примерным синтаксисом:

⟨А2-программа⟩ ::= НАЧАЛО ⟨последовательность описаний⟩ ;
⟨оператор⟩ КОНЕЦ
⟨описание⟩ ::= ⟨описание переменной⟩ | ⟨описание подпрограммы⟩
⟨описание подпрограммы⟩ ::= ⟨заголовок⟩ | ⟨заголовок⟩ ⟨ тело⟩
⟨заголовок⟩ ::= ПОДПРОГРАММА ⟨идентификатор⟩ ((список входных параметров)) РЕЗУЛЬТАТ ((список выходных параметров))
⟨входной параметр⟩ ::= ⟨имя⟩ : ⟨тип⟩
⟨выходной параметр⟩ ::= ⟨имя⟩ : ⟨тип⟩
⟨тело⟩ ::= НАЧАЛО ПОДПРОГРАММЫ ⟨последовательность описаний⟩ ; ⟨оператор⟩ КОНЕЦ ПОДПРОГРАММЫ.

Отсутствие тела подпрограммы означает возможность подстановки любого описания, согласованного с заголовком. Оператор вызова такой подпрограммы оказывается неинтерпретированным, и мы получаем возможность использования частично интерпретированных схем программ над памятью. С функциональной точки зрения подпрограмму с неполным описанием можно рассматривать как аргумент основной программы, вместо которого могут подставляться соответствующие функциональные значения. Так получается возможность описания функционалов высших типов.

Для реализации рекурсивных вызовов программами подпрограмм используется магазинная память, которую можно реализовать путем следующего расширения базиса. Пусть (U, Y) – стандартный базис над памятью V, интерпретированный на области D. Предполагая память V типизированной, введем новый тип данных D^* (множество всех конечных последовательностей элементов D, включая пустую). Элементы множества D^* будем называть *состояниями магазина,* а новые переменные, принимающие значения в D^*, – *магазинными переменными.* Определим на D^* операции $x \circ y$, $head(x)$, $tail(x)$, полагая, что $x \circ y$ есть конкатенация, $head(d_1 d_2 ...) = d_1$, $tail(d_1 d_2 ...) = d_2 ...$, $head(e) = tail(e) = e$.

Пусть V' – расширенная память, а (U', Y') – расширенный базис. Пусть $(A_\lambda)_{\lambda \in \Lambda}$ – система программ над распределенной памятью. Рассмотрим объединенный базис (U, Y), где $U = \bigcup_{\lambda \in \Lambda} U_\lambda$, $Y = \bigcup_{\lambda \in \Lambda} Y_\lambda$, над памятью $V = \bigcup_{\lambda \in \Lambda} V_\lambda$. Для того чтобы не возникало коллизий, можно считать без ограничения общности, что сигнатуры операций и предикатов, а также множества переменный для различных базисов не пересекаются. Расширим полученный базис до (U', Y'), добавив магазинный тип. Удалим теперь из базиса операторов Y' все операторы вызова подпрограмм. Новый базис операторов обозначим через Y''. Пусть $V_0 = V_{\lambda_0}$ (λ_0 – индекс основной схемы программы семейства). Базис (U', Y'') интерпретирован на области $D' = D^*$, где $D = \bigcup_{\lambda \in \Lambda} D_\lambda$ (элементы области $D \subset D^*$ отождествляются с одноэлементными последовательностями). Поскольку $U_\lambda \subset U$, $Y_\lambda \subset Y$, то все программы A_μ можно рассматривать как U-Y-программы над объединенной памятью V, использующие и изменяющие только свои части V_μ этой памяти.

Будем считать, что область D содержит бесконечно много элементов $a_1, a_2, \ldots$ одного и того же типа; с помощью условий сигнатуры Π можно реализовать каждый из предикатов $v = a_1$, $v = a_2, \ldots$. Предположим также, что для всех типов переменных допускается оператор $v := w$ присвоения неопределенного значения, память V функциональна и ее состояние однозначно определяется значениями переменных $z_1, \ldots, z_h$. При этих условиях имеет место следующая теорема.

Т е о р е м а 5.1 (об устранении рекурсии). *Для любой конечной системы программ (A_λ) над распределенной памятью, интерпретированной на области D, существует (U', Y'')-схема программы A' над памятью V' такая, что для всех $v \in V_0$ имеет место $(f_0(b))(v) = (f_{A'}(b'))(b)$, где b' совпадает с b на V_0 и не определено на всех других переменных.*

Построение программ A' состоит в последовательном преобразовании схем программ A_λ. На одном шаге преобразования устраняются все вызовы одной из программ A_λ из всех программ. Преобразованию подвергаются тела описаний подпрограмм, представляющих программы A_λ в текстовом виде. После устранения всех вызовов основная программа будет представлять программу A'. Перед началом преобразования к памяти каждой из программ добавляется переменная p магазинного типа. Эта переменная рассматривается как общая переменная всех программ A_λ, т.е. ее значение сохраняется при любых вызовах (вместо этого предположения можно было бы p добавить в качестве входного и выходного параметров каждой из программ, передавать и возвращать значение этой переменной при всех вызовах).

Рассмотрим устранение вызовов программы A_λ. Пусть $u_1, \ldots, u_m$ – входные, а $v_1, \ldots, v_n$ – выходные параметры этой программы и пусть в текстах программ A_μ встречается k вызовов программы A_λ. Сопоставим каждому из этих вызовов метку l_i $(i = 1, \ldots, k)$ и установим взаимно однозначное соответствие меток $l_1, \ldots, l_k$ и элементов $a_1, \ldots, a_k \in D$. Введем оператор $x \to p$ записи в магазин и оператор $x \leftarrow p$ чтения из магазина, полагая

$(x \to p) = (p := x \circ p)$,

$(x \leftarrow p) = (x := head(p),\ p := tail(x))$.

Положим также $(x_1, ..., x_s \to p) = (x_s \to p, ..., x_1 \to p)$ а $(x_1, ..., x_s \leftarrow p) = (x_1 \leftarrow p; ...; x_s \leftarrow p)$. Очевидно, что $(x_1, ..., x_s \to p)(x_1, ..., x_s \leftarrow p) = \epsilon$ (тождественный оператор).

Сопоставим программе A_λ оператор Q_λ следующего вида:

λ: НАЧАЛО

$z_1 := w$; ...; $z_h := w$;

$u_1, ..., u_m \leftarrow p$;

⟨оператор из тела описания A_λ⟩;

$r \leftarrow p$;

$v_1, ..., v_n \to p$;

ВЕТВЛЕНИЕ ПО r:

$a_1 \Rightarrow$ НА l_1,

$a_2 \Rightarrow$ НА l_2,

. . .

$a_k \Rightarrow$ НА l_k

КОНЕЦ ВЕТВЛ.

КОНЕЦ.

Вставим этот оператор в каждую из программ A_μ, окружив его обходом:

(НА l; Q_λ; l: ϵ).

Очевидно, что такая вставка не изменит функционирования программы A_μ. Теперь устраним вызовы. Вызов вида $(s_1, ..., s_n) := A_\lambda(t_1, ..., t_m)$, сопоставленный метке l_i, заменим оператором S_i:

НАЧАЛО

$t_1, ..., t_m, a_i, z_1, ..., z_h \to p$;

НА λ;

l_i: $x_1, ..., x_n \leftarrow p$;

$z_1, ..., z_h \leftarrow p$;

$(s_1 := x_1, ..., s_n := x_n)$

$(x_1 := w, ..., x_n := w, r := w)$

КОНЕЦ.

Систему уравнений (5.2) можно решить относительно операторов вызова в алгебре алгоритмов. Выделяя в этом решении вхождения $T_1, ..., T_k$ вызовов программы A_λ, получим

$$f_\mu = R_\mu(T_1, ..., T_k), \quad \mu \in \Lambda;$$
$$T_i(b) = y_i^{(1)}(b, f_\lambda(y_i^{(0)}(b))), \quad i = 1, ..., k. \tag{5.3}$$

В этих равенствах предполагается, что f_μ обозначают компоненты наименьшего решения системы (5.2). Аналогичную систему равенств можно сопоставить преобразованным программам A_μ, рассматриваемым как программы над объединенной памятью $V = \bigcup_{\mu \in \Lambda} V_\mu$.

Введем следующие обозначения. Пусть i-й вызов программы A_λ находится внутри программы A_μ. Тогда через $Q_{i,\lambda}$ обозначим функцию (оператор), вычисляемую программой A_μ при следующей настройке: начальное состояние – состояние, отмеченное меткой λ: заключительное состояние – состояние, в котором выполняется ветвление в операторе Q_λ в программе A_μ. Рассмотрим теперь настройку программы A_μ, при которой в качестве начального состояния выбирается состояние входа в оператор S_i, а в качестве заключительного – состояние выхода из этого оператора. Функцию, вычисляемую программой A_μ при такой настройке, обозначим так же, как и оператор S_i. Если через g_μ обозначить функцию, вычисляемую преобразованной программой A_μ, то получим

$$g_\mu = R_\mu(S_1, ..., S_k), \quad \mu \in \Lambda;$$

$$S_i = z_i^{(0)} Q_{i,\lambda} z_i^{(1)}, \quad i = 1, ..., k. \tag{5.4}$$

Здесь $z_i^{(0)}$ – запись в магазин текущего состояния памяти, предшествующая переходу на λ в S_i; $z_i^{(1)}$ – восстановление памяти и передача результатов после возвращения на метку l_i.

Из (5.3) вытекает система

$$T_i(b) = y_i^{(1)}(b, R_\lambda(T_1, ..., T_k)(y_i^{(0)}(b))), \quad i = 1, ..., k,$$

а из (5.4) – система

$$S_i = z_i^{(0)} R_{i,\lambda}(S_1, ..., S_k) z_i^{(1)}, \quad i = 1, ..., k,$$

где $R_{i,\lambda}$ – регулярная программа, которая выражает $Q_{i,\lambda}$ через операторы $S_1, ..., S_k$. Эта программа может быть получена путем анализа схемы A_μ, в которую входит i-й вызов A_λ (теорема 1.1 гл. 2). Поскольку $T_1, ..., T_k$ и $S_1, ..., S_k$ образуют наименьшие решения полученных систем, то

$$T_i = \bigcup_{m=0}^{\infty} T_i^{(m)}, \quad S_i = \bigcup_{m=0}^{\infty} S_i^{(m)},$$

где

$$T_i^{(m+1)}(b) = y_i^{(1)}(b, R_\lambda(T_1^{(m)}, ..., T_k^{(m)})(y_i^{(0)}(b))),$$

$$S_i^{(m+1)} = z_i^{(0)} R_{i,\lambda}(S_1^{(m)}, ..., S_k^{(m)}) z_i^{(1)}, \quad T_i^{(0)} = S_i^{(0)} = w, \quad i = 1, ..., k.$$

Индукцией по m доказывается, что $T_i^{(m)} = S_i^{(m)}$. Действительно, пусть b – произвольное состояние памяти, в котором значения вспомогательных переменных $x_1, ..., x_n$ и r не определены. Оператор $T_i^{(m+1)}$ преобразует его в состояние, которое можно получить, выполняя следующую последовательность операторов:

НАЧАЛО

$(z_1 := w, ..., z_h := w);$

$(u_1 := b(t_1), ..., u_m := b(t_m));$

$R_\lambda(T_1^{(m)}, ..., T_k^{(m)});$

$(z_1 := b(z_1), ..., z_h := b(z_h));$

$(s_1 := b'(v_1), ..., s_n := b'(v_n))$

КОНЕЦ.

Здесь b' – состояние памяти, полученное после выполнения первых трех операторов. Все это вытекает из определения операторов $y_i^{(0)}$ и $y_i^{(1)}$ для вызова $(s_1, ..., s_n) := A_\lambda(t_1, ..., t_m)$. Очевидно, что значения вспомогательных переменных $x_1, ..., x_n$ и r, а также значение магазина p остаются неизменными. Действие оператора $S_i^{(m+1)}$ эквивалентно выполнению следующей последовательности операторов:

НАЧАЛО

$t_1, ..., t_m, \ a_i, \ z_1, ..., z_h \to p;$

$(z_1 := w, ..., z_h := w);$

$u_1, ..., u_m \leftarrow p;$

$R_\lambda(S_1^{(m)}, ..., S_k^{(m)}); \ r \leftarrow p;$

$x_1, ..., x_n \leftarrow p;$

$(s_1 := x_1, ..., s_n := x_n);$

$(x_1 := w, ..., x_n := w, \ r := w)$

КОНЕЦ.

Это верно, поскольку по предположению индукции $T_i^{(m)} = S_i^{(m)}$, и, следовательно, оператор $R_\lambda(S_1^{(m)}, ..., S_k^{(m)})$ не меняет магазина. Применяя предположение индукции, получим, что $T_i^{(m+1)}(b) = S_i^{(m+1)}(b)$. Таким образом, $T_i = S_i$, откуда и вытекает справедливость теоремы (5.1).

Упражнения

1. Уточнить вариант языка А2 и построить его строгую динамическую семантику, определив функцию, которая по тексту А2-программы строит систему схем программ вместе с их базисами.

2. Доказать аналог теоремы 5.1 для систем программ над общей памятью.

Комментарии к главе 3

Классическая теория рекурсивных определений содержится в монографиях [55, 67]. Первые применения рекурсивных определений в программировании связаны с языком ЛИСП [90]. Теорема 1.2 для частично рекурсивных функционалов доказана Клини [55] и известна как первая теорема Клини о рекурсии. Общая теория аппроксимации и теория функционалов высших типов разрабатывались Д. Скоттом [74] и Ю.Л. Ершовым [43]. Методы вычисления функций, заданных рекурсивными определениями, рассматривались в [68]. Алгебры с аппроксимацией рассматриваются в [60].

Глава 4

СТРУКТУРЫ ДАННЫХ

§ 1. Функциональные структуры данных

Пусть D – базовая Ω-алгебра данных с индуктивным отношением аппроксимации. Рассмотрим множество $\Gamma(C, D)$ всех отображений из C в D. Элементы этого множества будем называть *функциональными структурами данных*, расположенными на C и принимающими значения в D. Поскольку в D есть неопределенный элемент w, структуру данных можно рассматривать как частичное отображение области расположения C в область значений D. Удобно считать, что любые два элемента d_1 и d_2 алгебры D имеют наименьшую верхнюю грань $d_1 \sqcup d_2$ (относительно аппроксимации). Если D является стандартным расширением области определенных элементов, то эту область можно еще раз расширить, добавив наибольший элемент $\bar{w}$. Тогда объединением двух неравных определенных элементов будет $\bar{w}$. Иногда $\bar{w}$ называют переопределенным элементом.

Операцию объединения будем считать одной из операций базовой алгебры данных.

Одним из наиболее важных примеров функциональных структур данных является структура данных, расположенная на целочисленной решетке Z^n и принимающая значения в D. Если область определения структуры данных $x \in \Gamma(Z^n, D)$ есть прямоугольный параллелепипед $[a_1 : b_1, ..., a_n : b_n]$, то мы имеем дело с прямоугольным многомерным массивом элементов типа D. Алгебра D обычно рассматривается как одноосновная алгебра и считается определенной с точностью до изоморфизма, т.е. как абстрактный тип данных. Это не исключает возможности для D быть компонентой $D = D_\xi$ некоторой многоосновной алгебры. В этом случае среди операций сигнатуры Ω могут быть кроме внутренних еще и внешние операции, использующие данные из других компонент.

В качестве D можно использовать и многоосновную алгебру. Однако в этом случае удобно заменить ее одноосновной алгеброй, объединив непересекающиеся компоненты и считая операции принимающими значение w в случае нарушения соответствия типов. Для того чтобы сохранить структуру типов, можно ввести в рассмотрение одноместные предикаты, выделяющие различные типы из объединения.

Определим на множестве $\Gamma(C, D)$ операции, превратив его в алгебру (структур данных).

1. Базовые операции сигнатуры Ω над структурами данных получаются путем перенесения операций базовой алгебры D на $\Gamma(C, D)$. Если $x_1, ..., x_n \in \Gamma(C, D)$, то $\omega(x_1, ..., x_n) = y$ есть структура данных такая, что для всех $c \in C$ имеет место

$$y(c) = \omega(x_1(c), ..., x_n(c)).$$

2. Операция вырезки x/H определена для любого множества $H \subset C$. Структура данных $y = x/H$ определяется равенством

$$y(c) = \textbf{если } c \in H \textbf{ то } x(c) \textbf{ иначе } w.$$

Сдвигом области расположения называется частичное преобразование $g \subset C \to C$ области C. Действие сдвига g на точку c обозначается cg.

3. Операция сдвига x^g структуры данных $x \in \Gamma(C, D)$ с помощью преобразования $g \subset C \to C$ определяется следующим образом. Если $y = x^g$, то для всех $c \in C$

$y(c) =$ **если** cg **определено то** $x(cg)$ **иначе** w.

4. Операция наложения $x \sqcup y$ структур данных x и y определяется как их наименьшая верхняя грань. Если $z = x \sqcup y$, то для всех $c \in C$

$z(c) = x(c) \sqcup y(c)$.

Наложение можно также рассматривать как одну из базовых операций, однако в связи с его особой ролью оно выделяется и как самостоятельная операция, которая может быть определена и в том случае, когда базовая алгебра не имеет объединения. Требуется только, чтобы для структур данных с непересекающимися областями определения их наложение совпадало с объединением.

Обычно для определения операций вырезки и сдвигов используются не произвольные подмножества и преобразования области C, а лишь некоторые допустимые. При этом допустимые подмножества должны быть замкнутыми относительно следующих операций.

1. Множество допустимых сдвигов G замкнуто относительно умножения (последовательного действия) и образует полугруппу сдвигов.

2. Совокупность допустимых множеств замкнуто относительно булевых операций (объединение, пересечение и дополнение), а также относительно умножения на допустимые сдвиги $Hg = \{cg \mid c \in H\}$ и деления на допустимые сдвиги $H/g = \{c \mid cg \in H\}$.

Алгеброй структур данных, расположенных на C и принимающих значения в D, в этом параграфе называется любое подмножество множества $\Gamma(C, D)$, замкнутое относительно базовых операций, допустимых вырезок, сдвигов и наложений.

Отметим некоторые тождественные соотношения алгебры структур данных ($x_1, \ldots, x_n, x, y$ – произвольные структуры данных):

$$(x^g)^h = x^{hg}, \tag{1.1}$$

$$\omega(\ldots, x \sqcup y, \ldots) = \omega(\ldots, x, \ldots) \sqcup \omega(\ldots, y, \ldots), \tag{1.2}$$

$$\omega(\ldots, x/H, \ldots) = \omega(\ldots, x, \ldots)/H \sqcup \omega(\ldots, w, \ldots)/\bar{H}. \tag{1.3}$$

В соотношении (1.3) символ w обозначает нигде не определенную структуру данных.

$$(x \sqcup y)/H = x/H \sqcup y/H, \tag{1.4}$$

$$(x/H)/H' = x/H \cap H', \tag{1.5}$$

$$\omega(x_1, \ldots, x_n)^g = \omega(x_1^g, \ldots, x_n^g), \tag{1.6}$$

$$(x/H)^g = x^g/(H/g), \tag{1.7}$$

$$(x \sqcup y)^g = x^g \sqcup y^g, \tag{1.8}$$

$$x/H_1 \cup H_2 = x/H_1 \sqcup x/H_2. \tag{1.9}$$

Соотношение (1.2) является следствием непрерывности операций базовой алгебры. Остальные доказываются исходя из определения операций путем сравнения значений соответствующих структур данных на произвольном элементе области расположения. Докажем, например, соотношение (1.7). По определению для произвольного $c \in C$ имеем $((x/H)^g)(c) = (x/H)(cg) =$ **если** $cg \in H$ **то** $x(cg)$ **иначе** $w =$ **если** $c \in H/g$ **то** $x^g(c)$ **иначе** $w = (x^g/(H/g))(c)$, откуда вытекает (1.7).

В соотношениях (1.2), (1.3) и (1.6) вместо операции ω базовой алгебры данных можно, очевидно, взять любую суперпозицию базовых функций (элементарную базовую функцию). Если $\varphi(x_1, ..., x_n)$ – элементарная базовая функция (точнее, выражение, представляющее такую функцию), то выражение $\varphi(x_1^{g_1}, ..., x_n^{g_n})$, где $x_1, ..., x_n$ – переменные структуры данных или константы, будем называть *базовым термом*.

Если базовые операции дают определенное значение только в случае, когда определены все аргументы, т.е. $\omega(..., w, ...) = w$, то соотношение (1.3) можно переписать в более простом виде:

$$\omega(..., x/H, ...) = \omega(..., x, ...)/H.$$

В качестве следствия в этом случае можно получить также соотношение

$$\omega(x_1, ..., x_n)/H = \omega(x_1/H, ..., x_n/H).$$

С помощью соотношений (1.1)–(1.9) любое выражение алгебры структур данных может быть приведено к виду

$$F(x_1, ..., x_n) = \bigsqcup_{i=1}^{k} F_i/H_i, \tag{1.10}$$

где $x_1, ..., x_n$ – произвольные структуры данных, $F_1, ..., F_k$ – базовые термы и $H_i \cap H_j = \phi$ при $i \neq j$.

Действительно, применяя соотношения (1.1)–(1.8) слева направо в любой последовательности, пока это возможно, получим наложение базовых термов и вырезок базовых термов. Вводя вырезки, если их нет, с помощью соотношения

$$x = x/C \tag{1.11}$$

получим выражение вида (1.10). Для того чтобы множества H_i не пересекались, применяем соотношение

$$x/H \sqcup y/H' = x/(H \cap \bar{H}' \cup H \cap H') \sqcup y/(H' \cap H \cup H' \cap \bar{H}) =$$
$$= x/H \cap \bar{H}' \sqcap (x \sqcup y)/H \cap H' \sqcup y/H' \cap \bar{H},$$

которое вытекает из (1.9). Дальнейшие упрощения можно получить с помощью соотношений

$$x/\phi = w, \tag{1.12}$$

$$x \sqcup w = x, \tag{1.13}$$

а также тождеств базовой алгебры данных.

Функция, аргументы и значения которой содержатся в $\gamma(C, D)$, называется *функцией* над структурами данных или структурной функцией. Структурная функция называется *элементарной*, если она выражается через опе-

рации алгебры структуры данных и, возможно, некоторые константы. Элементарные структурные функции допускают другое представление, которое называется *скалярным*.

Пусть $y = F(x_1, ..., x_n)$ – структурная функция, заданная с помощью канонической формы (1.10), $x_1, ..., x_n$ – переменные структуры данных. Тогда ее можно представить системой соотношений, выражающих значения структуры данных y в точках c области расположения:

$$y(c) = F_i(z_{i1}(cg_{i1}), ..., z_{im_i}(cg_{im_i})), \quad c \in H_i, \quad i = 1, ..., k, \tag{1.14}$$

где $F_i = F_i(z_{i1}^{g_{i1}}, ..., z_{im_i}^{g_{im_i}}), z_{i1}, ..., z_{im_i} \in \{x_1, ..., x_n\}$.

Соотношения (1.14) связывают значения скалярных переменных вида $y(c)$ и $x(cg)$ (x, y – структурные переменные) и в совокупности составляют скалярное представление структурной функции. Возможность скалярного представления определяет практическое значение алгебры структур данных, поскольку именно в такой форме представляются основные операции над структурами данных, которые встречаются в программировании и проектировании аппаратуры.

Например, формула

$$z_{ij} = \sum_{k=1}^{n} x_{ik} * y_{kj}, \quad i, j = 1, ..., n, \tag{1.15}$$

которая определяет произведение матриц порядка n, может рассматриваться как определение структурной функции $z = x * y$ над структурами данных, расположенными на двумерной решетке Z^2 и принимающими значение в кольце D, расширенном присоединением неопределенного элемента w и переопределенного элемента $\bar{w}$. Операции сложения и умножения продолжаются стандартным образом. Заметим, что расширенная алгебра данных не является кольцом, поскольку, например, соотношение $x - x = 0$ уже не является тождеством ($w - w = w$), но все тождества кольца сохраняются на множестве $D_0 \subset D$ определенных элементов. Если определить сдвиги g_k и h_k решетки Z^2 формулами

$$g_k(i, j) = (i, k),$$
$$h_k(i, j) = (k, j),$$

то формулу (1.15) можно переписать в виде

$$z(i, j) = \sum_{k=1}^{n} x(g_k(i, j)) * y(h_k(i, j)), \quad (i, j) \in [1{:}\, n, 1{:}\, n]$$

или в структурном виде:

$$z = (x^{g_1} * y^{h_1} + ... + g^{g_n} + y^{h_n})/[1{:}\, n, 1{:}\, n] =$$

$$= \Big(\sum_{k=1}^{n} x^{g_k} * y^{h_k} \Big)/ [1{:}\, n, 1{:}\, n].$$

Если x и y – матрицы порядка n, т.е. структуры данных, области определения которых совпадают с квадратом $[1{:}\, n, 1{:}\, n]$, то z – их произведение. Таким образом, умножение матриц заданного порядка можно рассматри-

вать как элементарную структурную функцию, ограниченную на матрицы этого порядка.

Пусть заданы две элементарные структурные функции $F(x_1, ..., x_n)$ и $F'(x_1, ..., x_n)$. Как узнать, равны ли они, т.е. является ли равенство

$$F(x_1, ..., x_n) = F'(x_1, ..., x_n)$$

тождеством? Приведем каждое из выражений, определяющих функции F и F', к канонической форме:

$$F(x_1, ..., x_n) = \bigcup_{i=1}^{k} F_i/H_i,$$

$$F'(x_1, ..., x_n) = \bigcup_{i=1}^{l} F'_i/H'_i.$$

Если некоторое H_i не пересекается ни с одним из множеств H'_j, то должно быть $F_i/H_i = w$, в противном случае функции не равны. Если указанное равенство имеет место, соответствующий член можно отбросить. Аналогично для множеств H'_i.

После отбрасывания лишних членов выполним следующие преобразования. Если $H_i \cap H'_j \neq \phi$, то заменим H_i объединением $H_i \cap H'_j \cup H_i \cap \overline{H'_j}$ и разложим соответствующий член с помощью соотношения (1.9) в наложение вырезок по множествам $H_i \cap H'_j$ и $H_i \cap \overline{H'_j}$, т.е. применим соотношение

$$x/H = x/H \cap H' \sqcup x/H \cap \overline{H'},$$

которое вытекает из (1.9). Это соотношение будем называть *разложением* вырезки x/H по множеству H'. Применяя взаимные разложения вырезок двух канонических форм до тех пор, пока возможно, и переставляя слагаемые, в конце концов придем к тому, что k станет равным l и $H_i = H'_i$. Теперь для равенства функций F и F' необходимо и достаточно выполнения равенств

$$F_i/H_i = F'_i/H_i.$$

Таким образом, задача свелась к распознаванию равенств вида

$$F(z_1^{g_1}, ..., z_m^{g_m})/H = F'(z_1^{g_1}, ..., z_m^{g_m})/H, \qquad (1.16)$$

где F и F' – базовые термы; $z_1, ..., z_m \in \{x_1, ..., x_n\}$ – переменные структуры данных. Предположим, что в H найдется такой элемент c, что все сдвиги $cg_1, ..., cg_m$ определены, и если $z_i = z_j$, то $cg_i \neq cg_j$. Поскольку в этом случае все скалярные переменные $z_1(cg_1), ..., z_m(cg_m)$ могут принимать любые значения в алгебре D независимо друг от друга, равенство (1.16) имеет место тогда и только тогда, когда в алгебре D, а следовательно, и в $\Gamma(C, D)$ выполняется тождество

$$F(z_1, ..., z_m) = F'(z_1, ..., z_m).$$

Вырезка базового терма $F(z_1^{g_1}, ..., z_m^{g_m})$ называется *свободной*, если существует $c \in H$ такое, что все $cg_1, ..., cg_m$ определены и из $z_i = z_j$ следует $cg_i \neq cg_j$. Каноническая форма (1.9) называется *строгой*, если все ее вырезки свободны.

Рассмотрим вопрос о том, как получить строгую каноническую форму. Для произвольных сдвигов g, g_1 и g_2 определим множества $H_0(g)$ и $H_1(g_1, g_2)$, полагая

$$H_0(g) = \{ c \mid gc \text{ определено} \},$$

$$H_1(g_1, g_2) = \{ c \mid cg_1 = cg_2 \}.$$

Предположим, что для допустимых сдвигов оба множества допустимы. Тогда имеют место следующие соотношения:

$$F(x^g)/\overline{H_0(g)} = F(w)/\overline{H_0(g)}, \tag{1.17}$$

$$F(x^{g_1}, x^{g_2})/H_1(g_1, g_2) = F(x^{g_1}, x^{g_1})/H_1(g_1, g_2), \tag{1.18}$$

где $F(x^g)$ и $F(x^{g_1}, x^{g_2})$ – базовые термы. Теперь строгая каноническая форма может быть получена путем разложения членов канонической формы (1.10) по множествам H_0 и H_1 и применением соотношений (1.17) и (1.18). Действительно, если вырезка $F(z_1^{g_1}, \ldots, z_m^{g_m})/H$ не свободна, то для любого c либо cg_i не определено, либо для некоторой пары i и j имеет место $z_i = z_j$ и $cg_i = cg_j$. Если для некоторых c и i имеет место первый случай, то разложим вырезку по $H_0(g_i)$; если для некоторого c и пары (i, j) имеет место второй, разложим ее по $H_1(g_i, g_j)$. После некоторого числа таких разложений, применения соотношений (1.17) и (1.18), а также отбрасывания одного из двух повторяющихся аргументов после (1.18), рассматриваемая вырезка превратится в наложение свободных.

Заметим, что при рассмотрении равенств вида (1.16) для того, чтобы формально в левой и правой частях иметь одинаковые аргументы $z_1^{g_1}, \ldots, z_m^{g_m}$, возможно, приходится добавлять такие, от которых вырезка зависит фиктивно. В этом случае свойство свободы может нарушиться, и необходимо сделать еще несколько шагов для одновременного приведения частей равенства (1.16) к свободному виду, сохраняя одинаковые аргументы.

Т е о р е м а 1.1. *Соотношения* (1.1) – (1.9), (1.10) – (1.13), (1.17) – (1.18) *вместе с тождествами базовой алгебры данных образуют полную систему тождеств для алгебры структур данных, в которой множества* $H_0(g)$ *и* $H_1(g_1, g_2)$ *допустимы для любых допустимых сдвигов* g, g_1 *и* g_2.

Действительно, для приведения двух равных выражений в алгебре структур данных к общей строгой канонической форме достаточно применения тождеств, указанных в формулировке теоремы.

Упражнения

1. Доказать тождества (1.1) – (1.9) и (1.16) – (1.18).

2. Построить алгоритм распознавания тождественного равенства выражений алгебры структур данных, расположенных на Z^n, если известен алгоритм распознавания тождеств базовой алгебры данных; полугруппа сдвигов состоит из параллельных переносов ($cg = c + c'$), а допустимые множества порождаются параллелепипедами $[a_1 : b_1, \ldots, a_n : b_n]$.

3. Обобщить результат предыдущего упражнения на случай, когда полугруппа сдвигов состоит из произвольных аффинных преобразований модуля Z^n над кольцом целых чисел.

§ 2. Периодически определенные функции

Рассмотрим систему уравнений

$$y_i = F_i(y, x), \quad i = 1, \ldots, m, \tag{2.1}$$

в алгебре структур данных $\Gamma(C, D)$, расположенных на C и принимающих значения в D. Здесь $y = (y_1, \ldots, y_m)$ — неизвестные структуры данных, $x = (x_1, \ldots, x_n)$ — произвольные фиксированные структуры данных (параметры), $F_i(y, x)$ — элементарные структурные функции. Отношение аппроксимации, определенное на D, переносится на множество $\Gamma(C, D)$ и превращает его в индуктивное, частично упорядоченное множество с нулем (нигде не определенная структура данных w). Отношение $x \sqsubset y$ означает, что для всех $c \in C$ имеет место $x(c) \sqsubset y(c)$, а $\bigsqcup_{n=1}^{\infty} x_n$ для возрастающей цепочки $x_1 \sqsubset x_2 \sqsubset \ldots$ структур данных есть такая структура данных y, что для всех $c \in C$ имеет место $y(c) = \bigsqcup_{n=1}^{\infty} x_n(c)$.

Нетрудно показать, что все операции алгебры структур данных, а следовательно, и элементарные функции непрерывны. Поэтому система (2.1) имеет наименьшее решение, которое определяется формулами:

$$\begin{aligned} y_i &= \bigsqcup_{k=0}^{\infty} y_i^{(k)}, \\ y_i^{(0)} &= w, \\ y_i^{(k+1)} &= F_i(y_1^{(k)}, \ldots, y_m^{(k)}, x). \end{aligned} \tag{2.2}$$

Структуры данных y_i являются функциями, зависящими от параметров $x_1, \ldots, x_n$. Систему (2.1) можно рассматривать также как каноническую систему функциональных уравнений специального вида: $y_i(x) = F_i(y(x), x)$. Ее решение также определяется соотношениями (2.2) и является непрерывным по всем своим аргументам $x_1, \ldots, x_n$. Структурные функции, которые могут быть определены системами уравнений вида (2.1), называются *периодически определенными структурными функциями.*

Основное свойство периодически определенных функций состоит в том, что, как и элементарные функции, они допускают скалярное представление. Для того чтобы получить это представление, приведем правые части уравнений (2.1) к канонической форме:

$$y_i = \bigsqcup_{j=1}^{k_i} Q_{ij}(z_{i1}^{g_{i1}}, \ldots, z_{il_i}^{g_{il_i}}) / H_{ij}, \quad i = 1, \ldots, m. \tag{2.3}$$

Переходя к скалярным переменным, получим

$$y_i(c) = Q_{ij}(z_{i1}(cg_{i1}), \ldots, z_{il_i}(cg_{il_i})), \quad c \in H_{ij}, \tag{2.4}$$

$$i = 1, \ldots, m, \; j = 1, \ldots, k_i.$$

Здесь $z_{i1}, \ldots, z_{il_i} \in \{x_1, \ldots, x_n, y_1, \ldots, y_m\}$. В отличие от элементарных функций скалярное представление периодически определенных функций представляет собой систему уравнений, где неизвестными являются скалярные переменные $y_t(c)$ $(c \in C)$. Рассмотрим несколько примеров.

1. Пусть $C = \mathbf{Z} = \{\ldots, -1, 0, 1, 2, \ldots\}$. Допустимые сдвиги – это прибавление целочисленной константы. Допустимые множества – конечные или дополнения конечных множеств. Алгебра D – двухэлементная булева алгебра, стандартным образом расширенная и продолженная.

Рассмотрим систему соотношений

$$z(i) = x(i) + y(i) + p(i), \qquad i \geqslant 0,$$

$$p(i) = x(i-1) \wedge y(i-1) \vee x(i-1) \wedge p(i-1) \vee y(i-1) \wedge p(i-1), \quad i > 0,$$

$$p(0) = 0.$$

Здесь + обозначает сложение по модулю 2. Очевидно, что если последовательности $x(n)x(n-1) \ldots x(0)$ и $y(n)y(n-1) \ldots y(0)$ представляют два целых числа, записанных в двоичной системе, то $z(n)z(n-1) \ldots z(0)$ представляет собой их сумму, если она меньше чем 2^{n+1}, $p(i)$ – перенос в i-й разряд $(i \leqslant n)$, $p(n+1)$ – признак переполнения. Для того чтобы указанные соотношения согласовывались с общим видом скалярного представления периодически определенной функции $z(x, y)$, нужно только последнее равенство переписать в виде $p(i) = 0$, $i = 0$. Алгебраическое представление рассматриваемой функции имеет вид

$$z = (x + y + p)/H_0,$$

$$p = (x^g \wedge y^g \vee x^g \wedge p^g \vee y^g \wedge p^g)/H_1,$$

где $H_0 = \{0, 1, \ldots\}$, $H_1 = \{1, 2, \ldots\}$, $cg = c - 1$.

2. Соотношение

$$x_t = (b_t - \sum_{j=t+1}^{n} a_{tj}x_j)/a_{tt}, \qquad t = 1, \ldots, n,$$

выражает решение x системы линейных алгебраических уравнений порядка n с верхней треугольной матрицей коэффициентов a и вектором свободных членов b.

Особенность этого соотношения состоит в том, что в нем участвуют структуры данных, имеющие различные области расположения – векторы и матрицы. Для того чтобы ликвидировать это различие, будем рассматривать переменные с одним индексом как переменные с двумя индексами, у которых один из индексов фиксирован, например, $x_t = x_{t0}$, $b_t = b_{t0}$. Введем в рассмотрение еще одну переменную s_{uv}, полагая

$$s_{uv} = \sum_{j=u+1}^{v} a_{uj}x_j.$$

Теперь имеем:

$$x_{t0} = (b_{t0} - s_{tn})/a_{tt}, \quad t = 1, \ldots, n,$$

$$s_{uv} = s_{uv-1} + a_{uv}x_{v0}, \quad u = 1, \ldots, n, \quad v = u+1, \ldots, n,$$

$$s_{uu} = 0, \quad u = 1, \ldots, n.$$

Связь со скалярным представлением периодически определенной функции $x = \varphi(a, b)$ над структурами данных, расположенных на двумерной решетке Z^2, очевидна.

3. Пусть C есть множество вершин ориентированного графа, из каждой вершины которого выходит ровно m дуг. Если $c \in C$, то $cg_1, \ldots, cg_m$ – все вершины, в которые ведут дуги, выходящие из c. Алгебра D состоит из целых чисел.

Рассмотрим скалярное представление функции $y = \varphi(x_1, \ldots, x_m)$:

$$y(c) = 0, \quad c \in H_0,$$

$$y(c) = \min(y(cg_1) + x_1(c), \ldots, y(cg_m) + x_m(c)), \quad c \in H_1.$$

Пусть $x_i(c)$ определяет стоимость перехода из c в cg_i. Тогда, если граф C не имеет циклов и $H_1 g_i \subset H_1 \cup H_0$ $(i = 1, \ldots, m)$, то $y(c)$ – минимальная стоимость пути из c в H_0.

Рассмотрим способы вычисления периодически определенных функций. Будем предполагать, что алгебра данных D является стандартным расширением алгебры определенных элементов D_0 со стандартным продолжением операций ($D = D_0 \cup \{w, \bar{w}\}$, $\omega(\ldots, w, \ldots) = w$, $\omega(\ldots, \bar{w}, \ldots) = \bar{w}$). Систему соотношений (2.4) можно переписать в виде

$$y_i(c) = \textbf{если } c \in H_{i1} \textbf{ то } Q_{i1} \textbf{ иначе}$$

$$\textbf{если } c \in H_{i2} \textbf{ то } Q_{i2} \textbf{ иначе}$$

$$\ldots$$

и рассматривать как систему рекурсивных определений функций y_i из C в D. К этой системе можно применить общие методы вычисления алгебраических функций над двухосновной алгеброй (C, D) при стандартном расширении области C и добавлении предикатов, распознающих принадлежность множествам H_{ij}. Однако интерес представляют специальные методы вычислений периодически определенных функций, учитывающие их специфику и в особенности приспособленность для описания параллельных вычислений.

Рассмотрим систему периодически определенных функций $y_1(x), \ldots, y_m(x)$, заданных системой (2.1), и предположим, что нужно вычислить значения структур данных $y_1(x)/H_1, \ldots, y_m(x)/H_m$ (некоторые из множеств $H_1, \ldots, H_m$ могут быть пустыми). Предположим, что система (2.1) приведена к канонической форме (2.3). Основным методом вычисления является метод, основанный на формулах (2.2), который будем называть *методом синхронного вычисления структурных функций*. Обозначим через

$E_{ik} \subset C$ область определения структуры данных $y_i^{(k)}$, т.е. множество таких точек $c \in C$, что $y_i^{(k)}(c) \neq w$.

Пусть $C_{ik} = E_{ik} \setminus E_{ik-1}$ $(k \geqslant 1,\ i = 1, \ldots, m)$. Вычисления выполняются по шагам. На k-м шаге одновременно вычисляются значения всех структур данных $y_1, \ldots, y_m$ во всех точках областей $C_{1k}, \ldots, C_{mk}$ соответственно, а также в тех точках множеств E_{ik}, где значение структуры y_i может стать переопределенным. Если известно, что структуры данных $x_1, \ldots, x_n$ нигде не переопределены, то в силу стандартности продолжения y_i может стать переопределенным лишь тогда, когда в выражениях Q_{ij} используется операция наложения. Поэтому если внутренних наложений нет, то вычисления производятся только в точках областей C_{ik}. Вычисления выполняются до тех пор, пока последовательность значений структур данных $y_1/H_1, \ldots, y_m/H_m$ не стабилизируется. Разумеется, реально такие вычисления можно провести, лишь если множества H_i и C_{ik} конечны, хотя теоретическую модель можно рассматривать и с бесконечными множествами.

Конструктивность рассмотренного метода вычислений определяется конструктивностью операций базовой алгебры данных, задания множеств H_i, H_{ij} и сдвигов.

Для того чтобы изучить структуру множеств E_{ik} и C_{ik}, рассмотрим ориентированный граф (S, R) информационных зависимостей. Множество S вершин этого графа состоит из всех скалярных переменных $y_1(c), \ldots, y_m(c), x_1(c), \ldots, x_n(c)$ $(c \in C)$. Отношение $R \subset S^2$ связывает вершину $z(c')$ с вершиной $y_i(c)$ тогда и только тогда, когда $c \in H_{ij}$, $c' = cg_{ip}$, $z = z_{ip}$ для некоторого $j = 1, \ldots, k_i$. Для упрощения рассуждений, связанных с операцией наложения, предположим, что эта операция встречается только внутри выражений Q_{ij}, имеющих вид $z_1 \sqcup \ldots \sqcup z_k$, где $z_1, \ldots, z_k$ — скалярные переменные. Такое предположение не ограничивает общности, поскольку к указанному виду всегда можно перейти, вводя дополнительные переменные. Вершина $y_i(c)$ графа S называется *дизъюнктивной*, если $c \in H_{ij}$ и Q_{ij} есть наложение сдвигов скалярных переменных.

Граф S позволяет легко определить множества E_{ik}, исходя из условия стандартного продолжения операций. Если вершина $y_i(c)$ не является дизъюнктивной, то значение переменной $y_i(c)$ на k-м шаге вычислений определено тогда и только тогда, когда на $k-1$-м шаге определены значения всех переменных $z(c') \in R^{-1}(y_i(c))$, что вытекает из определения отношения R информационной зависимости. Если же вершина $y_i(c)$ дизъюнктивна, то $y_i(c)$ на k-м шаге определено тогда и только тогда, когда значение хотя бы одной из переменных $z(c') \in R^{-1}(y_i(c))$ определено на $k-1$-м шаге. Определим множества $S_k \subset S$, полагая $s \in S_0 \Longleftrightarrow s = x_j(c)$ для некоторого $j = 1, \ldots, n$ и $x_j(c)$ определено. Для $k > 0$ полагаем $s \in S_k \Longleftrightarrow s \in S_{k-1}$ или $s = y_i(c)$ для некоторого $i = 1, \ldots, m$ и $R^{-1}(s) \subset S_{k-1}$ для недизъюнктивной вершины и $R^{-1}(s) \cap S_{k-1} \neq \phi$ для дизъюнктивной вершины. Таким образом, значение переменной s на k-м шаге синхронных вычислений определено $\Longleftrightarrow s \in S_k$, и, следовательно, $c \in E_{ik} \Longleftrightarrow y_i(c) \in S_k$.

Множество $S_k \setminus S_{k-1}$ $(k \geqslant 1)$ называется *k-м фронтом волны вычислений.* Оно состоит из тех переменных, значения которых вычисляются на k-м шаге. C_{ik} – фронт волны вычислений переменной y_i состоит из всех точек c таких, что $y_i(c) \in S_k \setminus S_{k-1}$.

Синхронные вычисления по рассмотренной схеме обладают некоторой избыточностью. Они не учитывают того, что заданы множества $H_1, \ldots, H_m$, на которых следует вычислять $y_1, \ldots, y_m$. На самом деле вычисления $y_i(c)$ нужно проводить не во всех точках области расположения, а только в точках, связанных информационными зависимостями с точками областей $H_1, \ldots, H_m$. Пусть $P_i = \{ y_i(c) \mid c \in H_i,\ i = 1, \ldots, m \}$. Рассмотрим множество переменных $T \subset S$, полагая $T = \bigcup_{i=1}^{m} (R^{-1})^*(P_i)$ ($(R^{-1})^*$ – транзитивно-рефлексивное замыкание отношения R^{-1}). Очевидно, что для вычисления значений переменных из множеств S_i достаточно вычислять только значения переменных из T.

Рассмотрение синхронных вычислений равносильно рассмотрению детерминированной дискретной динамической системы $B = D^S$ (память скалярных переменных), функция переходов δ которой определяется так, что $\delta(b) = b'$, где $b'(s)$ вычисляется по формулам (2.2) для всех переменных $s \in T$. Начальные состояния системы B – это состояния b, для которых $b(s)$ может быть отличным от w лишь для переменных s вида $x_i(c)$. Процесс вычислений в точке s завершается, если через конечное число шагов значения переменных стабилизируются во всех точках множества $(R^{-1})^*(s)$. Процесс завершается на множестве T, если он завершается во всех точках этого множества и число шагов стабилизации для всех точек этого множества ограничено.

Элемент $s \in S$ назовем *конечным*, если все пути, которые оканчиваются в s, имеют конечную длину. Из этого определения, в частности, следует, что никакой из конечных элементов не лежит на цикле. Если существует n такое, что все пути, которые оканчиваются в s, имеют длину не больше, чем n, то наименьшее из таких n называется *высотой* элемента s.

Лемма 2.1. *Все конечные элементы имеют конечную высоту.*

Действительно, если пути, которые оканчиваются в s, имеют неограниченную длину, то существует и бесконечный путь, оканчивающийся в s. Действительно, если s не имеет конечной высоты, то среди элементов множества $R^{-1}(s)$ также существует элемент, который не имеет конечной высоты. Пусть это будет элемент s_1; для него также найдется неконечный элемент s_2 и т.д.

Теорема 2.1. *Если граф информационных зависимостей не имеет дизъюнктивных вершин, то процесс синхронных вычислений в любой точке s всегда завершается через конечное число шагов.*

Если элемент не является конечным, то скажем, что он имеет *бесконечную высоту.* Из леммы 2.1 следует, что значения элементов бесконечной высоты всегда не определены (стандартное расширение операций), а значение элемента s высоты n, если определено, то не позже, чем через n шагов. Действительно, все элементы множества S_k имеют высоту k. Теорема доказана.

Обозначим через X множество всех скалярных переменных вида $x_t(c)$. Каждая переменная $s = y_j(c)$ конечной высоты является элементарной D-функцией от переменных из множества $X_s = (R^{-1})^*(s) \cap X$. Поэтому для того, чтобы значение s было определено, необходимо и достаточно, чтобы все значения переменных из X_s были определены.

Если в графе есть дизъюнктивные вершины, то утверждение теоремы 2.1 может не быть верным. Пусть, например, $y_i = x_{i+1} \sqcup z_i$ $(i = 0, 1, 2, \ldots)$, где $z_i = x_i$, если $i = 2^n$ $(n = 1, 2, \ldots)$, и $z_i = y_{i+1}$ для остальных i. Значение y_i определено, если хотя бы одна из переменных x_{2^n} определена и принимает определенное значение, и все определенные значения x_{2^n} равны. Если не все определенные значения x_{2^n} равны, то y_0 переопределено. Очевидно, что процесс вычислений в точке y_0 завершается, лишь если это значение не определено или переопределено.

Аналогичные трудности возникают при отказе от стандартного продолжения операций и стандартного расширения области D. Отсутствие априорных оценок или признаков завершения процессов вычислений оставляет только возможность удовлетворяться приближенными значениями. С другой стороны, нестандартные расширения могут более адекватным образом описывать реальные процессы. В примере 1 при стандартном расширении процесс вычисления переноса $p(i)$ является чисто последовательным. Но если считать, что $0 \wedge x = 0$, $1 \vee x = x$ при любых x (включая $x = w$), то этот процесс становится параллельным, что и происходит в реальных арифметических устройствах и может быть использовано для ускорения их работы.

В случае конечных областей H_{ij} при стандартном расширении завершимость процесса вычислений гарантирована. Число элементов множества C_k называется *шириной фронта* волны вычислений и характеризует возможность их параллельного выполнения, а максимальная высота элементов множества переменных вида $y_i(c)$, $c \in H_i$, характеризует время вычислений.

Другие способы вычисления периодически определенных функций, отличные от синхронных вычислений, получаются добавлением других возможных переходов к дискретной системе $B = D^S$. Максимально асинхронная система получается, если допустить все переходы вида $b \to b'$, где b' получается из b вычислением значений каких-либо переменных из множества T (не обязательно всех, которые могут быть вычислены). Все другие способы вычислений, включая последовательные, получаются как подсистемы максимально асинхронной. Модели, в которых учитывается время вычисления по формулам (2.4) при асинхронном выполнении, могут дать лучший результат, чем при синхронном.

Упражнение

Построить фронт волны вычислений для примеров, рассмотренных в параграфе.

§ 3. Многоосновные алгебры структур данных

Алгебры структур данных, рассмотренные в § 1, являются одноосновными. Основным множеством каждой такой алгебры является подмножество множества $\Gamma(C, D)$, замкнутое относительно операций алгебры структур данных, определенных с помощью полугруппы G допустимых сдвигов и множества $\mathfrak{R}$ допустимых множеств. Понятие допустимости формулируется в терминах операций, определенных на G и $\mathfrak{R}$, и поэтому естественно рассматривать эти множества наряду с множеством структур данных как компоненты многоосновной алгебры структур данных. В качестве основных компонент к такой алгебре добавляются также множества C и D со своими операциями. Таким образом получается пятиосновная алгебра структур данных $(S, C, D, G, \mathfrak{R})$. Операции алгебры S уже рассматривались, D является базовой Ω-алгеброй, G — полугруппой, а $\mathfrak{R}$ — булевой алгеброй с операциями умножения и деления на сдвиги. Соотношения (1.1)—(1.9), рассмотренные в § 1, фактически являются соотношениями в многоосновной алгебре.

Кроме рассмотренных, могут быть определены смешанные операции на C и D. Например, если $C = Z^n$ и $Z \subset D$, то на D может быть определена операция $c = (z_1, \ldots, z_n)$, которая по набору чисел $z_1, \ldots, z_n$ дает точку области расположения. Эта операция, вообще говоря, частичная, и для ее полноправного рассмотрения область C целесообразно расширить добавлением неопределенных элементов. Другой путь состоит в рассмотрении алгебры D как многоосновной. Материал этой алгебры может использоваться также для построения полугруппы сдвигов.

В общем случае, если $\varphi\colon D^n \to C$ есть взаимно однозначное отображение D^n в C, то в качестве сдвигов можно взять все преобразования, порожденные элементарными D-функциями. Именно, если $\alpha_i(z_1, \ldots, z_n)$ $(i = 1, \ldots, n)$ — набор элементарных функций, то ему соответствует сдвиг g такой, что $g(c) = \varphi(\alpha_1(z_1, \ldots, z_n), \ldots, \alpha_n(z_1, \ldots, z_n))$; где $(z_1, \ldots, z_n) = \varphi^{-1}(c)$, если этот вектор определен. При переходе к многосновной базовой алгебре $D = (D_\xi)$ естественно и S сделать многоосновной, рассматривая семейство $S_\xi = \Gamma(C, D_\xi)$, а базовые операции определять как смешанные операции многоосновной алгебры. Один из возможных путей расширения набора базовых операций, отражающий практические ситуации, состоит в рассмотрении операций $\omega(x_1, \ldots, x_n)$ над структурами данных $x_1, \ldots, x_n$ таких, что $\omega(x_1, \ldots, x_n)(c) = \omega(x_1(c), \ldots, x_n(c), c)$, где ω — уже смешанная операция, использующая не только элементы D, но и элементы области расположения.

Дальнейшее обогащение алгебры структур данных связано с переходом к рассмотрению структур данных, имеющих различные области расположения. Такие структуры данных часто встречаются на практике. Например, массивы могут иметь различную размерность, их области расположения — целочисленные решетки с разным числом измерений. Теоретически этот случай легко свести к рассмотрению одной области расположения (для массивов это достигается вложением в решетку максимальной размерности). Однако вынужденное использование общей области расположения уменьшает наглядность представления преобразований информации в виде

выражений алгебры структур данных и скрывает иногда важные технические детали процесса проектирования программ.

Рассмотрим класс Σ множеств, которые могут быть использованы в качестве областей расположения структур данных. Элементы класса Σ будем называть *допустимыми областями расположения*. Пусть $C, C' \in \Sigma$. Частичное отображение $g \subset C \to C'$ называется сдвигом области C в C'. Пусть G — класс сдвигов, связывающих допустимые области. Предположим, что пара (Σ, G) удовлетворяет следующим условиям. Обозначим через $\mathrm{Hom}(C, C')$ множество всех допустимых сдвигов из C в C'. Тогда:

1. Для любых $g \in \mathrm{Hom}(C, C')$ и $h \in \mathrm{Hom}(C', C'')$ их композиция gh принадлежит $\mathrm{Hom}(C, C'')$.

2. Для произвольного $C \in \Sigma$ в $\mathrm{Hom}(C, C)$ содержится тождественный сдвиг ϵ.

Эти условия означают, что пара (Σ, G) образует категорию с объектами из Σ и морфизмами из G. Для любого $g \in \mathrm{Hom}(C', C)$ определим сдвиг x^g структуры данных $x \in \Gamma(C, D)$ с помощью сдвига g, полагая для любого $c \in C'$ $x^g(c) =$ **если** cg **определено то** $x(cg)$ **иначе** w. Таким образом, $x^g \in \Gamma(C', D)$, и операция сдвига имеет тип $\Gamma(C, D) \times \mathrm{Hom}(C', C) \to \to \Gamma(C', D)$ (сдвиг структур данных, расположенных на C, в структуры данных, расположенные на C'). Сдвиг, определенный для одноосновной алгебры, получается как частный случай при $g \in \mathrm{Hom}(C, C)$. Если $H \subset C \in \Sigma$, а $g \in \mathrm{Hom}(C', C)$, то можно определить деление H/g области H на g, полагая $H/g = \{c \in C' \mid cg \in H\}$. С каждой допустимой областью C свяжем множество $\mathfrak{R}(C)$ допустимых множеств и будем предполагать замкнутость этого множества относительно булевых операций и операции деления на допустимые сдвиги.

Для многоосновных алгебр структур данных сохраняются все тождества, рассмотренные в § 1, вернее, их обобщенные варианты. Многоосновные алгебры дают возможность введения новых типов операций, изменяющих области расположения и определяемых с помощью свертывающих операций типа суммирования элементов массивов.

Пусть $\nu \in \Omega$ — бинарная коммутативная и ассоциативная операция алгебры D с нейтральным элементом ξ, т.е. $\nu(\xi, d) = \nu(d, \xi) = d$. Примером могут служить операции сложения и умножения чисел. В первом случае нейтральным элементом служит 0, во втором — 1. Для произвольного множества $H \subset C$ и операции ν определим функционал $\Phi_{H,\nu}$: $\Gamma(C, D) \to D$, полагая:

1) $\Phi_{\phi,\nu}(x) = \xi$;

2) $\Phi_{\{c\},\nu}(x) = x(c)$;

3) $\Phi_{H_1 \cup H_2,\nu}(x) = \nu(\Phi_{H_1,\nu}(x), \Phi_{H_2,\nu}(x))$, $H_1 \cap H_2 = \phi$.

Очевидно, что если множество H конечно, то условия 1) — 3) определяют функционал $\Phi_{H,\nu}$ однозначно. В некоторых случаях функционал $\Phi_{H,\nu}$ может быть определен и для бесконечных множеств. Например, если ν —

конъюнкция или дизъюнкция (кванторы) либо min, max для числовых значений, ограниченных в совокупности.

При этом, однако, следует иметь в виду, что при переходе к бесконечным множествам может потеряться свойство непрерывности.

Для специальных операций ν применяются соответствующие обозначения. Например, если ν есть сложение, то

$$\Phi_{H,\,\nu}(x) = \sum_{c \in H} x(c),$$

если ν — конъюнкция, то

$$\Phi_{H,\,\nu}(x) = \bigwedge_{c \in H} x(c) = (\forall c \in H) x(c) \text{ и т.д.}$$

Скалярные представления удобно употреблять и в общем случае, используя обозначение

$$\Phi_{H,\,\nu}(x) = \Phi_{\nu}\ x(c).$$
$$c \in H$$

Например, равенство

$$y(c) = \Phi_{\nu}\ x(c, c'),\ c \in C,\ H \subset C',$$
$$c' \in H$$

определяет структурную функцию φ: $\Gamma(C \times C', D) \to \Gamma(C, D)$. Аргумент x этой функции расположен на $C \times C'$, а ее значение $y = \varphi(x)$ — на C. Если H можно линейно упорядочить с помощью сдвига g, т.е. $H = \{c_0, c_0 g, \ldots, c_0 g^m\}$, то функционал Φ_{ν} из последнего равенства можно элиминировать, полагая

$$y(c) = z(c, c_0),$$

$$z(c, c') = \nu(z(c, c'g), x(c, c')),\ c' \in H,\ c' \neq c_0 g^m,$$

$$z(c, c_0 g^m) = x(c, c_0 g^m),$$

или в структурных обозначениях

$$y = z^{h_1},$$

$$z = \nu(z^{h_2}, x)/H_1 \sqcup x/H_2,$$

где $h_1(c) = (c, c_0)$, $h_2(c, c') = (c, c'g)$, $H_1 = C \times H \setminus \{c_0 g^m\}$, $H_2 = C \times \{c_0 g^m\}$. Аддитивные функционалы допускают параллельное вычисление, поскольку $\Phi_{H,\nu}$ можно параллельно вычислять на непересекающихся подмножествах множества H. Если множество H и его подмножества разбивать на примерно равные части, то общее время вычисления $\Phi_{H,\nu}(x)$ будет иметь порядок $O(\log n)$, где n — число элементов множества H. После элиминации функционал Φ_{ν} будет вычисляться последовательно, однако если структура данных x сама определяется с помощью уравнения

$x = \varphi(x)$, то изменение способа вычисления функционала может не дать выигрыша. Например, замена первого из соотношений

$$y_i = \sum_{j=1}^{n} x_{ij},$$

$$x_{ij} = \omega(x_{i-1\,j}, z_{ij}),$$

$$x_{0j} = a$$

на

$$y_i = u_n,$$

$$u_j = u_{j-1} + x_{ij},$$

$$u_0 = 0$$

не ухудшит времени вычисления значений y_i на интервале $1 \leqslant i \leqslant m$, которое при $m > n$ и при любом распараллеливании будет иметь порядок не меньше, чем $O(n)$.

Зафиксировав набор операций для многоосновной алгебры структур данных, быть может, расширив его путем добавления уже исследованных структурных функций, снова можно рассматривать элементарные структурные функции и системы уравнений вида

$$y = F(y, x),$$

где y, x — наборы структур данных, возможно, с различными областями расположения, а F — набор элементарных структурных функций. Применяя общую теорию рекурсивных определений, получим некоторый класс вычислимых алгебраических структурных функций, которые можно исследовать аналогично тому, как были исследованы периодически определенные функции.

§ 4. Рекурсивные структуры данных

Функциональные структуры данных определяются таким образом, что для каждого класса (алгебры) структур данных фиксируется область (или несколько областей) расположения, а в процессе вычислений меняются только значения структур данных в различных точках области расположения и, возможно, изменяются в тех или иных пределах их области определения. Рекурсивные структуры данных, рассматриваемые в этом параграфе, более тесно связывают области расположения и элементарные значения. В таких структурах данных в процессе обработки могут изменяться не только элементарные значения, но также области расположения и связи между элементами этих областей. Далеким прототипом таких структур данных являются списочные структуры языка лисп, а также динамические структуры данных, определяемые с помощью указателей (ссылок, связей) в современных языках программирования.

В качестве основного понятия рассматривается понятие составного объекта, частным случаем которого является нагруженное дерево. На мно-

жестве деревьев легко определить алгебру, которая оказывается изоморфной алгебре термов или свободной универсальной алгебре, если составные объекты рассматривать с точностью до изоморфизма. Для того чтобы распространить алгебраическую точку зрения на составные без циклов, необходимо ввести эквивалентность более слабую, чем изоморфизм, и рассмотрение произвольных структур данных приводит к каноническим системам уравнений и предельному переходу. Возникающие здесь конструкции могут быть распространены на произвольные алгебры с отношением аппроксимации.

Перейдем к рассмотрению понятия составного. *Составной объект*, или просто *составной*, есть совокупность вершин и связей между ними. Каждой вершине поставлена в соответствие некоторая информация – отметка вершины и упорядоченное множество вершин, с которыми данная вершина непосредственно связана. Эти вершины называются также подчиненными данной вершине. Количество вершин, непосредственно подчиненных данной, однозначно определяется ее отметкой и называется *арностью* этой отметки, а следовательно, и вершины. Транзитивные замыкания отношений "непосредственно связана" и "непосредственно подчинена" дают отношения "связана" и "подчинена". Вместо термина "подчинена" используют также термин "достижима". Таким образом, вершина a связана с вершиной b (или, что то же самое, b подчинена a) $\Leftrightarrow$ существует путь от a к b, т.е. последовательность вершин $a = a_1, a_2, \ldots, a_n = b$ такая, что a_i непосредственно связана с a_{i+1} $(i = 1, \ldots, n-1, n > 1)$. Отношение достижимости не обязательно рефлексивно, т.е. связь вершины с самой собой может входить или не входить в составной объект. Среди вершин составного объекта выделена начальная вершина, связанная со всеми другими вершинами этого составного объекта.

Все сказанное можно формализовать следующим образом. Пусть Ω – сигнатура операций. *Ω-графом* называется тройка (C, φ, ψ), где $\varphi: C \to \Omega$, $\psi \subset C \times N \to C$, такая, что $\psi(c, i)$ определено $\Leftrightarrow 1 \leqslant i \leqslant n$, где n есть арность операции $\omega = \varphi(c)$. Если $\psi(c, i) = c'$, то говорят, что вершина c непосредственно связана с вершиной c' i-й связью, а c' непосредственно подчинена вершине c. Обозначается это записью $c \to c'$ или $c \overset{i}{\to} c'$. Ω-граф C, в котором выделена начальная вершина c_0, т.е. четверка (C, φ, ψ, c_0), называется *составным объектом*, если все его вершины, кроме, быть может, начальной, достижимы из c_0.

Каждая вершина составного объекта порождает новый составной объект. Начальной вершиной этого составного является данная вершина, и он состоит из данной и всех вершин, подчиненных ей, а также связей между ними. Отношения подчиненности и связанности так же, как и отметки вершин, переносятся с вершин на объекты, порождаемые этими вершинами. Объект, подчиненный данному объекту, называется его *подобъектом.* Объект, который не имеет непосредственно подчиненных, называется *первичным составным объектом.* Отметки вершин первичных имеют арность 0.

Для любого составного объекта t однозначно определены отметка α его начальной вершины и объекты $t_1, \ldots, t_n$, непосредственно подчиненные объекту t. Если c_0 – начальная вершина объекта t, то $t_1, \ldots, t_n$ порождают-

ся вершинами $c_1 = \psi(c_0, 1), ..., c_n = \psi(c_0, n)$ соответственно. В этом случае пишут $t \to \alpha(t_1, ..., t_n)$ и называют это выражение *разложением* объекта t. Первичные объекты имеют тривиальные разложения вида $t \to \alpha$, где α – отметка единственной вершины первичного составного.

Отношение $t \to \alpha(t_1, ..., t_n)$, рассматриваемое как отношение арности $n + 1$ на множестве составных (при фиксированном α), не является функциональным по первому аргументу, т.е. из $t \to \alpha(t_1, ..., t_n)$ и $t' \to \alpha(t_1, ..., t_n)$ не следует, вообще говоря, что $t = t'$. Действительно, на рис. 4.1 изображен составной объект t. Его подобъекты t' и t'' имеют разложения $t' \to F(a, b)$ и $t'' \to F(a, b)$, но $t'' \neq t'$. Здесь a и b – первичные объекты с отметками α и β соответственно. Рисунок 4.1 демонстрирует способ изображения составных объектов в виде ориентированных графов. При этом стрелки упорядочиваются слева направо или отмечаются своими номерами.

Пусть $t_1 = (C_1, \varphi_1, \psi_1, a_1)$, $t_2 = (C_2, \varphi_2, \psi_2, a_2)$. Отображение γ: $C_1 \to C_2$ называется гомоморфизмом объекта t_1 на t_2, если $\gamma(a_1) = a_2$, и для любой вершины $c \in C_1$ $\varphi_1(c) = \varphi_2(\gamma(c))$, $\gamma(\psi_1(c, i)) = \psi_2(\gamma(c), i)$ $(1 \leqslant i \leqslant$ арность $(c))$. Гомоморфизм γ называется *изоморфизмом*, если γ взаимно однозначно. Составные объекты изоморфны, если существует изоморфизм одного из них на другой. Гомоморфизм γ естественным образом переносится с вершин на подобъекты. При этом, если t есть подобъект объекта t_1 и $t \to \alpha(u_1, ..., u_m)$, то $\gamma(t) \to \alpha(\gamma(u_1), ..., \gamma(u_m))$. Для изоморфизма оба разложения эквивалентны. Если составные t и t' изоморфны, то изоморфизм между ними определяется однозначно.

Составные объекты часто рассматриваются с точностью до изоморфизма, и в таких случаях изоморфные объекты отождествляются и считаются равными. Составные объекты, рассматриваемые с точностью до изоморфизма, называются также *абстрактными* составными объектами. Абстрактный составной объект можно отождествлять с классом всех объектов, изоморфных ему. Первичный объект с точностью до изоморфизма определяется отметкой своей вершины и, рассматриваемый как абстрактный, отождествляется с ней.

Отношение $t \to \alpha(t_1, ..., t_n)$ переносится также на абстрактные объекты. Теперь уже объекты t' и t'', изображенные на рис. 4.1, как абстрактные оказываются равными. Но в общем случае отношение подчиненности не

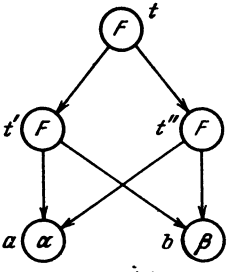


Рис. 4.1

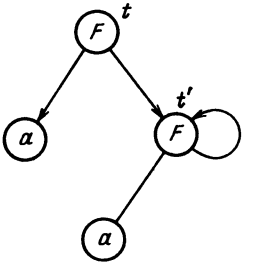


Рис. 4.2

является функциональным, т.е. из $t \to \alpha(t_1, ..., t_n)$ и $t' \to \alpha(t_1, ..., t_n)$ не следует, что t и t' изоморфны. Пример показан на рис. 4.2. Очевидно, что $t \to F(a, t!)$ и $t' \to F(a, t!)$, но t и t' не изоморфны. В этих формулах два разных первичных объекта рассматриваются как абстрактные и отождествляются с их отметкой a.

Каждая отметка α арности $n > 0$ определяет n-арную операцию, действующую на абстрактных составных объектах. Результат применения этой операции к объектам $(t_1, ..., t_n)$ обозначается через $\alpha(t_1, ..., t_n)$ и представляет собой абстрактный составной объект. Для того чтобы получить граф этого объекта, следует представить $t_1, ..., t_n$ графами без общих вершин, добавить новую вершину с отметкой α и связать ее с начальными вершинами объектов $t_1, ..., t_n$. Если $t = \alpha(t_1, ..., t_n)$, то очевидно, что $t \to \alpha(t_1, ..., t_n)$, но обратное, вообще говоря, не верно.

Любое множество абстрактных составных объектов, замкнутое относительно операций, определенных отметками, образует алгебру составных. Всякая алгебра составных является абсолютно свободной универсальной Ω^+-алгеброй, порожденной неразложимыми составными объектами, т.е. такими объектами, которые нельзя представить в виде $t = \alpha(t_1, ..., t_n)$ $(n > 0)$. Здесь Ω^+ состоит из всех операций сигнатуры Ω арности $n > 0$. Частным случаем алгебры составных является алгебра, порожденная первичными составными объектами. Эта алгебра называется *алгеброй деревьев* (точнее, Ω-деревьев), а ее элементы называются *деревьями*. Если t – дерево, то $t \to \alpha(t_1, ..., t_n) \iff t = \alpha(t_1, ..., t_n)$. Всякое дерево может быть представлено выражением в алгебре деревьев, т.е. выражением, построенным из символов, обозначающих первичные объекты и операции. Это представление единственно с точностью до синтаксических подробностей представления операций (включая правила старшинства, используемые для опускания скобок). Поэтому алгебра Ω-деревьев изоморфна алгебре Ω^+-термов, порожденной первичными составными. Рассматривая алгебру составных, можно некоторые из первичных объектов рассматривать как 0-арные операции и включать их в сигнатуру. Тогда это будет Ω'-алгебра, где $\Omega^+ \subset \Omega \subset \subset \Omega$. Что понимается под Ω', следует каждый раз оговаривать особо.

Для абстрактных составных объектов понятие разложения допускает естественное обобщение. Если $F(t_1, ..., t_n)$ есть выражение в алгебре абстрактных составных объектов, построенное из (абстрактных) составных $t_1, ..., t_n$, то разложение $t \to F(t_1, ..., t_n)$ означает, что составной t может быть построен из составных $t_1, ..., t_n$ следующим образом. Строим составной $t' = F(t_1, ..., t_n)$. Если t отличен от $t_1, ..., t_n$, то $t = t'$. Если же, например, $t = t_i$, то t получается из t' путем отождествления начальной вершины этого составного с начальной вершиной его подобъекта t_i.

Любой конечный составной объект с точностью до изоморфизма может быть задан в виде совокупности разложений

$$\begin{aligned} x_1 &\to F_1(x_1, ..., x_n), \\ &\cdots\cdots\cdots\cdots \\ x_n &\to F_n(x_1, ..., x_n), \end{aligned} \tag{4.1}$$

в которой выделено начальное разложение (например, первое). Здесь $x_1, \ldots, x_n$ – некоторые символы (параметры); $F_1(x_1, \ldots, x_n), \ldots, F_n(x_1, \ldots, x_n)$ – выражения в алгебре деревьев, которые, кроме первичных объектов, используют символы $x_1, \ldots, x_n$. Составной объект, заданный таким разложением, строится следующим образом. Сначала строим деревья, представленные выражениями $F_1(x), \ldots, F_n(x)$, рассматривая параметры как отметки первичных. Затем каждую вершину a_i, отмеченную символом x_i, отождествляем с начальной вершиной объекта $F_i(x)$, удаляя при этом отметку x_i и заменяя ее отметкой начальной вершины дерева $F_i(x)$. В качестве начальной вершины полученного составного выбираем начальную вершину начального разложения.

Если обозначить через t_i составной, порожденный вершиной a_i, то имеет место разложение $t_i \to F_i(t_1, \ldots, t_n)$. Система разложений (4.1) называется *параметрической* системой разложений. Если $F_i(x_1, \ldots, x_n) = \alpha(x_{i_1}, \ldots, x_{i_k})$ $(\alpha \in \Omega,\ k \geqslant 0)$, то параметрическая система разложений называется *канонической*. Каноническая система разложений определяется однозначно с точностью до переименования параметров и перестановки разложений.

Рассмотрим примеры.

Объект, изображенный на рис. 4.1, можно задать в виде разложений

$$t \to F(F(x, y), F(x, y)),$$

$$x \to \alpha,$$

$$y \to \beta.$$

Объект, изображенный на рис. 4.2 – в виде разложений

$$t \to F(a, t'),$$

$$t' \to F(a, t').$$

Разложение

$$t \to F(F(\alpha, \beta), F(\alpha, \beta))$$

задает объект, изображенный на рис. 4.3. Он не изоморфен объекту рис. 4.1.

Определение составного объекта не исключает случая, когда множество его вершин бесконечно. Бесконечные составные объекты можно получить, разворачивая циклы конечных составных. Формально операцию развертывания можно определить следующим образом. Пусть $t = (C, \varphi, \psi, c_0)$. Рассмотрим составной $\bar{t} = (\bar{C}, \bar{\varphi}, \bar{\psi}, \bar{c}_0)$. $\bar{C}$ – это множество всех конечных инициальных путей $c_0 \to c_1 \to \ldots \to c_m$ составного t. Начальная вершина $\bar{c}_0$ – это путь нулевой длины c_0, $\bar{\varphi}(c_0 \to \ldots \to c_m) = \varphi(c_m)$, $\bar{\psi}(c_0 \to \ldots \to c_m, i) = c_0 \to \ldots \to c_m \to \psi(c_m, i)$, если $\psi(c_m, i)$ определено и не определено в противном случае. Составной $\bar{t}$ называется *развертыванием* составного t. Он всегда является деревом (возможно, бесконечным) в том смысле, что для любой вершины существует единственный путь, соединяющий с ней начальную вершину.

Операция развертывания приводит к важному понятию строгой эквивалентности составных объектов. Составные t_1 и t_2 *строго эквивалентны*, если их развертывания $\overline{t}_1$ и $\overline{t}_2$ изоморфны.

С каждым составным объектом t свяжем язык $L(t)$, порождаемый этим объектом. Язык $L(t)$ состоит из слов в алфавите $Z \subset \Omega \times N$, $N = \{0, 1, \ldots\}$. Пара $(\omega, i) \in Z \Leftrightarrow$ арность $(\omega) = 0$ и $i = 0$ или арность $(\omega) = n > 0$ и $1 \leqslant i \leqslant n$. Слово $(\alpha_0, 0)(\alpha_1, i_1) \ldots (\alpha_m, i_m) \in L(t) \Longleftrightarrow$ существует

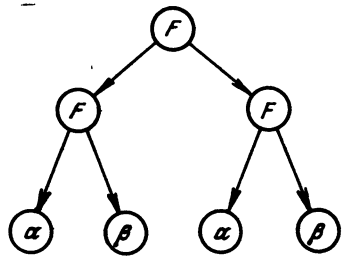


Рис. 4.3

инициальный путь в t вида $c_0 \xrightarrow{i_1} c_1 \xrightarrow{i_2} \ldots \xrightarrow{i_m} c_m$ такой, что $\alpha_i = \varphi(c_i)$ $(i = 1, \ldots, m)$.

Теорема 4.1. *Составные t и t' эквивалентны* $\Leftrightarrow L(t) = L(t')$.

Действительно, соответствие между инициальными путями в составном и элементами порождаемого им языка взаимно однозначно, поэтому из равенства $L(t) = L(t')$ следует изоморфизм развертываний составных t и t'.

Легко доказывается также следующее свойство развертываний:

Теорема 4.2. *Если* $t \to F(t_1, \ldots, t_n)$, *то* $\overline{t} = F(\overline{t}_1, \ldots, \overline{t}_n)$.

В этой теореме $F(t_1, \ldots, t_n)$ – выражение в алгебре деревьев, $t_1, \ldots, t_n$ – составные. Теорема доказывается индукцией по числу операций в выражении $F(t_1, \ldots, t_n)$.

Отношение строгой эквивалентности является отношением конгруэнтности в алгебре S конечных составных объектов, и можно рассматривать фактор-алгебру $\overline{S}$ по этому отношению. Элементами этой алгебры являются фактор-объекты, т.е. составные, рассматриваемые с точностью до строгой эквивалентности. Поскольку всякий такой объект однозначно определяется некоторым деревом (возможно, бесконечным), фактор-алгебра $\overline{S}$ изоморфна алгебре деревьев, которые получаются развертыванием составных из S. Дерево (вообще говоря, бесконечное), которое является развертыванием конечного составного объекта, называется *регулярным*. Алгебра конечных составных объектов, рассматриваемых с точностью до строгой эквивалентности, изоморфна, таким образом, алгебре регулярных деревьев.

Если $t' \to F(t_1, \ldots, t_n)$, $t'' \to F(t_1, \ldots, t_n)$, то составные t' и t'' строго эквивалентны, поскольку их развертывания в силу теоремы 4.2 изоморфны. Поэтому в алгебре $\overline{S}$ разложения превращаются в равенства, а системы параметрических разложений – в канонические системы уравнений в этой алгебре.

Теорема 4.3. *Каждая каноническая система уравнений в алгебре $\overline{S}$ имеет единственное решение.*

Если

$$x_i = F_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n, \tag{4.2}$$

есть каноническая система уравнений, то составные $t_1, \ldots, t_n$, заданные параметрической системой разложений

$$x_i \to F_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n,$$

и рассматриваемые с точностью до строгой эквивалентности, образуют решение этой системы в алгебре $\bar{S}$. Соотношения (4.2) однозначно определяют также языки $L(t_1), \ldots, L(t_n)$, откуда следует единственность решения.

Алгебру $\bar{S}$ можно построить также другим путем, используя теорию аппроксимации. Для этого рассмотрим алгебру $D = T_\Omega(A)$ конечных Ω-деревьев, порожденную первичными составными, образующими множество A. Выделим в множестве A пустой (неопределенный) составной объект w и введем на D отношение частичного порядка $\sqsubset$. Будем считать, что $t \sqsubset t' \Leftrightarrow t'$ получается из t подстановкой деревьев вместо некоторых вхождений пустого составного объекта. Иными словами, $t' = g(t_1, \ldots, t_n)$, $t = g(w, \ldots, w)$. Отношение $\sqsubset$ является отношением частичного порядка с наименьшим элементом w, и его можно рассматривать как отношение аппроксимации. Множество D не является индуктивным. Бесконечная последовательность деревьев сходится в $D \Leftrightarrow$ она стабилизируется через конечное число шагов. Однако D можно вложить в индуктивную алгебру с аппроксимацией, относительно которой операции оказываются непрерывными. Идея состоит в том, чтобы пределы последовательностей отождествить с самими последовательностями. Эта важная конструкция может быть применена к более широкому классу алгебр, чем алгебры деревьев, и мы рассмотрим ее в общем виде.

Итак, пусть теперь D – произвольная Ω-алгебра с отношением аппроксимации $\sqsubset$, наименьшим элементом w, монотонными и непрерывными операциями. Непрерывность монотонной функции f: $D \to D$ означает здесь следующее: если возрастающая последовательность $x_1 \sqsubset x_2 \sqsubset \ldots$ элементов D сходится, т.е. имеет наименьшую верхнюю грань $\bigsqcup_{i=1}^{\infty} x_i$, то последовательность $f(x_1) \sqsubset f(x_2) \sqsubset \ldots$ также сходится и $f(\bigsqcup_{i=1}^{\infty} x_i) = \bigsqcup_{i=1}^{\infty} f(x_i)$.

Рассмотрим множество D^N всех функций натурального аргумента, принимающих значения в D, и выделим из них множество $M(D)$ монотонных функций, т.е. таких функций f, что $f(n) \sqsubset f(n+1)$ для всех $n = 0, 1, \ldots$. Записывая аргументы в виде индексов, получим, что всякую функцию $f \in M(D)$ можно отождествить с бесконечной возрастающей последовательностью $f_0 \sqsubset f_1 \sqsubset \ldots$. Превратим множество $M(D)$ в Ω-алгебру, полагая для $f^{(1)}, \ldots, f^{(m)} \in M$ результат применения операции ω равным $\omega(f^{(1)}, \ldots, f^{(m)}) = f = (\omega(f_0^{(1)}, \ldots, f_0^{(m)}), \ldots, \omega(f_n^{(1)}, \ldots, f_n^{(m)}), \ldots)$. Каждому элементу $d \in D$ сопоставим последовательность $d \sqsubset d \sqsubset \ldots$. Это сопоставление задает вложение алгебры D в $M(D)$ в качестве подалгебры. Определим на $M(D)$ отношение $\sqsubset$, полагая $f \sqsubset f' \Leftrightarrow$ для каждого $n \in N$ существует $k \geqslant 0$ такое, что $f_n \sqsubset f'_{n+k}$ или f' сходится к элементу d в D, а $f_n \sqsubset d$ для всех $n \in \mathrm{N}$. В первом случае будем говорить, что f' *мажорирует* f, во втором – что *предел последовательности* f' *мажори-*

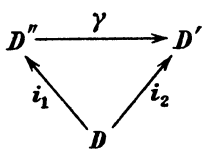
рует f. Таким образом, $f \sqsubset f' \Leftrightarrow f$ мажорируется последовательностью f' или ее пределом.

Отношение $\sqsubset$ совпадает на D с отношением аппроксимации, а на $M(D)$ является квазипорядком. Рассмотрим отношение эквивалентности ϵ на множестве $M(D)$, полагая $f = f'(\epsilon) \Leftrightarrow f \sqsubset f'$ и $f' \sqsubset f$. На фактор-множестве $M(D)/\epsilon$ отношение $\sqsubset$ будет частичным порядком.

Покажем, что ϵ является конгруэнтностью. Пусть $f^{(1)} = g^{(1)}(\epsilon), \ldots, f^{(m)} = g^{(m)}(\epsilon)$. Нужно показать, что $\omega(f^{(1)}, \ldots, f^{(m)}) = \omega(g^{(1)}, \ldots, g^{(m)})(\epsilon)$. Последнее сравнение, в свою очередь, требует доказательства двух включений. В силу симметрии достаточно доказать одно из них. Например, первое вытекает из следующей цепочки включений: $\omega(f^{(1)}, \ldots, f^{(m)}) \sqsubset \omega(g^{(1)}, f^{(2)}, \ldots, f^{(m)}) \sqsubset \omega(g^{(1)}, g^{(2)}, f^{(3)}, \ldots) \sqsubset \ldots \ldots \sqsubset \omega(g^{(1)}, \ldots, g^{(m)})$. Докажем, например, первое в этой цепочке (остальное доказывается аналогично). По предположению $f^{(1)} \sqsubset g^{(1)}$. Если $g^{(1)}$ мажорирует $f^{(1)}$, то $\omega(f_n^{(1)}, \ldots, f_n^{(m)}) \sqsubset \omega(g_{n+k}^{(1)}, f_{n+k}^{(2)}, \ldots)$; если же $f^{(1)}$ мажорируется пределом $h = \bigcup_{n=1}^{\infty} g_n^{(1)}$, то $\omega(f_n^{(1)}, \ldots, f_n^{(m)}) \sqsubset \sqsubset \omega(h, f_n^{(2)}, \ldots, f_n^{(m)})$. Следовательно, $\omega(f^{(1)}, \ldots, f^{(m)}) \sqsubset \omega(h, f^{(2)}, \ldots, f^{(m)})$. Таким образом, $M(D)/\epsilon$ есть Ω-алгебра с отношением аппроксимации. Алгебра D по-прежнему вкладывается в $M(D)/\epsilon$, поскольку на D отношение ϵ совпадает с равенством. Это вложение сохраняет отношение аппроксимации и все пределы. Кроме того, каждая возрастающая последовательность алгебры D имеет теперь предел в $M(D)/\epsilon$, совпадающий с этой последовательностью, рассматриваемой с точностью до отношения эквивалентности ϵ. Сохранение пределов вытекает из того, что если $\bigcup_{n=0}^{\infty} d_n = d$, то последовательности $d_0 \sqsubset d_1 \sqsubset \ldots$ и $d \sqsubset d \sqsubset \ldots$ ϵ-эквивалентны.

Алгебра $M(D)/\epsilon$ является максимальным расширением алгебры D, состоящим из пределов произвольных последовательностей элементов D. Точный смысл этого утверждения раскрывается следующей теоремой.

Теорема 4.4. *Пусть D' – алгебра с отношением аппроксимации и непрерывными операциями, а D вкладывается изоморфно в D' с сохранением отношения аппроксимации и пределов. Тогда если каждый элемент алгебры D' является пределом возрастающей последовательности элементов из D, то существует гомоморфизм γ: $D'' \to D'$ некоторой подалгебры D'' алгебры $M(D)/\epsilon$, содержащей D, на D', причем диаграмма*



в которой i_1 и i_2 – соответствующие (изоморфные) вложения алгебры D, коммутативна.

Если $f \in M(D)$, то через $\bar{f}$ обозначим класс всех последовательностей, ϵ-эквивалентных последовательности f. Если f имеет предел в D', то обоз-

начим этот предел $\sqcup f$. В качестве D'' возьмем множество всех классов вида $\bar{f}$, где f имеет предел в D'. Если $f^{(1)}, \ldots, f^{(m)}$ имеют пределы в D', то в силу непрерывности операций $\omega(f^{(1)}, \ldots, f^{(m)})$ также имеет предел $\sqcup\omega(f^{(1)}, \ldots, f^{(m)}) = \omega(\sqcup f^{(1)}, \ldots, \sqcup f^{(m)})$. Поэтому $\omega(\overline{f^{(1)}}, \ldots, \overline{f^{(m)}}) = \overline{\omega(f^{(1)}, \ldots, f^{(m)})} \in D''$ вместе с $\overline{f^{(1)}}, \ldots, \overline{f^{(m)}}$, т.е. D'' есть подалгебра алгебры $M(D)/\epsilon$.

Положим $\gamma(\bar{f}) = \sqcup f$. Независимость этого определения от выбора представителя f класса $\bar{f}$ вытекает из того, что если $f = f'(\epsilon)$ и f имеет предел в D', то f' также имеет предел и $\sqcup f = \sqcup f'$. Докажем это. Пусть $f = f'(\epsilon)$ и $\sqcup f = d$. Если f' имеет предел d' в D, то $\sqcup f' = d'$. Из ϵ-эквивалентности f и f' вытекает, что d есть верхняя грань для f', а d' – верхняя грань для f. Поэтому $d = d'$. Если же f' не имеет предела в D, то f' мажорирует f и, следовательно, любая верхняя грань последовательности f' является верхней гранью для f. Поскольку d есть верхняя грань для f', то и, обратно, всякая верхняя грань для f есть верхняя грань для f' и, следовательно, d есть наименьшая верхняя грань: $\sqcup f' = d$.

Отображение γ является гомоморфизмом. Действительно, $\gamma(\omega(\overline{f^{(1)}}, \ldots, \overline{f^{(m)}})) = \gamma(\overline{\omega(f^{(1)}, \ldots, f^{(m)})}) = \sqcup\omega(f^{(1)}, \ldots, f^{(m)}) = \omega(\sqcup f^{(1)}, \ldots, \sqcup f^{(m)}) = \omega(\gamma(\overline{f^{(1)}}, \ldots, \gamma(\overline{f^{(m)}}))$. То, что γ есть гомоморфизм на, а диаграмма коммутативна, очевидно. Теорема доказана.

С помощью алгебры $M(D)/\epsilon$ можно строить различные расширения алгебры D, добавляя пределы тех или иных последовательностей. Важным частным случаем является минимальное расширение $D^* \subset M(D)/\epsilon$, в котором каждая каноническая система уравнений имеет наименьшее решение. Оно строится следующим образом. Рассмотрим каноническую систему уравнений

$$x_i = F_i(x_1, \ldots, x_m), \quad i = 1, \ldots, m, \tag{4.3}$$

в алгебре D. Она определяет m бесконечных последовательностей $f^{(1)}, \ldots, f^{(m)}$, определяемых рекурсивно:

$$f_n^{(i)} = F_i(f_{n-1}^{(1)}, \ldots, f_{n-1}^{(m)}), \quad n = 1, 2, \ldots,$$

$$f_0^{(i)} = w, \quad i = 1, \ldots, m.$$

Подалгебра D^* состоит из всех последовательностей, порождаемых таким образом с помощью канонических систем уравнений (рассматриваемых с точностью до ϵ-эквивалентности). Алгебра D^* или любая изоморфная ей алгебра с изоморфным вложением в нее алгебры D называется *регулярным расширением* алгебры D. В алгебре D^* каждая каноническая система уравнений вида (4.3) имеет наименьшее решение $\sqcup f^{(1)}, \ldots, \sqcup f^{(m)}$. Рассмотренная выше алгебра $\bar{S}$ конечных составных объектов, рассматриваемых с точностью до строгой эквивалентности, является, таким образом, регулярным расширением алгебры $T_\Omega(A)$ конечных деревьев, порожденных первичными A с выделенным пустым составным w и определенным выше отношением аппроксимации. Для этой алгебры примем обозначение $T_\Omega^*(A)$ и будем называть ее элементы *абсолютно свободными* составными объектами, подчеркивая этим, что в основе их построения лежит абсолютно свободная алгебра.

В ряде приложений составные объекты рассматриваются с точностью до некоторой более слабой эквивалентности, чем строгая эквивалентность. Пусть D – некоторая Ω-алгебра, $A \subset D$ – ее подмножество. Тогда $T_\Omega(A)$ можно рассматривать как множество выражений в алгебре D. Два выражения D эквивалентны, если они представляют один и тот же элемент алгебры D. Множество $T_\Omega(A)$, рассматриваемое с точностью до D-эквивалентности, является Ω-алгеброй, изоморфной подалгебре алгебры D, порожденной множеством A. Эту алгебру будем обозначать $T_\Omega(A)/D$ или просто $T_\Omega(A)$, если не возникает недоразумений. Пусть на D задано отношение аппроксимации. Оно переносится на $T_\Omega(A)$ с сохранением непрерывности. Тогда можно построить регулярное замыкание $T^*_\Omega(A)$. Элементами его являются составные, рассматриваемые с точностью до D-эквивалентности, которые мы будем называть *составными над алгеброй D*.

С помощью составных можно определить алгебру элементарных D-функций с константами из A. Для этой цели добавим к множеству A множество переменных X и рассмотрим алгебру $T_\Omega(A \cup X)$ выражений, которые кроме констант могут содержать переменные. Два выражения $t_1(x_1, \ldots, x_n)$ и $t_2(x_1, \ldots, x_n)$ называются D-эквивалентными, если $t_1(d_1, \ldots, d_n) = t_2(d_1, \ldots, d_n)$ для любых $d_1, \ldots, d_n \in D$. Факторизуя $T_\Omega(A \cup X)$ по отношению D-эквивалентности, получим алгебру, которую будем обозначать $T_\Omega(A, X, D)$.

Если выражения t_1 и t_2 D-эквивалентны, то соотношение $t_1 = t_2$ называется *тождеством* алгебры D. Алгебра $T_\Omega(A, X, D)$ получается факторизацией алгебры $T_\Omega(A \cup X)$ с помощью соотношений, которые включают в себя соотношения подалгебры алгебры D, порожденной множеством A, и все тождества алгебры D. Она изоморфна алгебре элементарных D-функций с константами из A. Иногда удобно рассматривать не все, а только некоторые из тождеств алгебры D. Если факторизовать $T_\Omega(A \cup X)$ с помощью соотношений из A и тождеств из множества $\mathcal{H}$, то получим алгебру, которую будем обозначать $T_\Omega(A, X, \mathcal{H})$. Если $\mathcal{H}$ содержит все соотношения, то эта алгебра совпадает с $T_\Omega(A, X, D)$. Если D – алгебра с аппроксимацией, то отношение аппроксимации переносится и на $T_\Omega(A, X, D)$, если положить, что $t_1(x_1, \ldots, x_n) \sqsubset t_2(x_1, \ldots, x_n)$, и если для любых $d_1, \ldots, d_n \in D$ имеет место $t_1(d_1, \ldots, d_n) \sqsubset t_2(d_1, \ldots, d_n)$. При этом непрерывность операций сохраняется, и можно построить регулярное расширение $T^*_\Omega(A, X, D)$. В случае произвольной системы тождеств $\mathcal{H}$ возможность перенесения отношения аппроксимации зависит от системы тождеств $\mathcal{H}$. Если такое перенесение сделано и построено регулярное расширение $T^*_\Omega(A, X, \mathcal{H})$, то его элементы будем называть составными $\mathcal{H}$-объектами над алгеброй D или составными над D, если $\mathcal{H}$ совпадает с множеством всех тождеств алгебры D.

Рассмотрим примеры.

1. Пусть Ω содержит единственную бинарную операцию $x \cdot y$. Алгебра деревьев с этой операцией и множеством первичных A – это алгебра бесцикловых списочных структур. Списками обычно называют структуры вида $x_1 \cdot (x_2 \cdot (\ldots (x_n \cdot w) \ldots))$, где w – пустой список. Произвольные составные объекты над этой сигнатурой – списочные структуры с циклами. Абстрактные составные могут использоваться, например, для задания регулярных языков (см. упражнение 2). Если множество пер-

вичных состоит из чисел (целых или вещественных) и переменных, а сигнатура расширена введением соответствующих операций над числами, получаем структуры данных языка лисп.

2. Пусть снова сигнатура операций содержит единственную бинарную операцию $x \cdot y$, которая объявляется ассоциативной. Множество $T_\Omega(A)$ в этом случае является свободной полугруппой. Элементами ее являются строки или слова в алфавите A.

3. Добавим к предыдущему примеру еще одну бинарную ассоциативную операцию $\vee$ и тождества $x \vee x = x, x \vee y = y \vee x$, $(x \vee y)z = xz \vee yz, z(x \vee y) = zx \vee zy$. Для неопределенного объекта w положим $w \vee x = x$, $wx = xw = w$. Алгебра $T_\Omega(A)$ в этом случае изоморфна алгебре конечных языков (конечных множеств слов в алфавите A). Элемент w играет роль пустого языка. Уместно добавить еще один первичный e (пустое слово) и соотношения $xe = ex = x$. В качестве отношения аппроксимации естественно взять отношение теоретико-множественного включения. Известно, что элементами множества $T^*_\Omega(A)$ являются произвольные контекстно-свободные языки. Составные объекты в сигнатуре Ω используются в качестве способа задания порождающей грамматики для некоторого контекстно-свободного языка. Эквивалентность составных в этом случае означает совпадение порождаемых ими языков.

4. Алгебра отношений, алгебра алгоритмов, алгебра D-функций и функционалов высших ступеней могут служить материалом для построения структур данных. Для каждой из этих алгебр могут быть построены полезные отношения эквивалентности, начиная от строгой и кончая равенством в соответствующих алгебрах.

Упражнения

1. Доказать, что гомоморфизм составного t на t', если существует, то единствен.
2. Доказать, что язык $L(t)$, порожденный составным объектом t, регулярен.
3. Продолжить рассмотрение п. 4 примеров, пользуясь материалом предыдущих глав.
4. Доказать, что алгебра $T^*_\Omega(\phi, X, D)$ изоморфна алгебре алгебраических D-функций.

§ 5. Теоретико-множественные структуры данных

Функциональные структуры данных представляют собой частный случай более общего понятия теоретико-множественных структур данных, которое позволяет с единой точки зрения охватить различные типы структур данных, используемые в процессе алгоритмического проектирования систем преобразования информации.

На начальном этапе проектирования алгоритма мы имеем дело с предметной областью задачи. Результат проектирования нужно выразить в терминах предметной области вычислительной системы, с помощью которой должен быть реализован алгоритм. Алгебраическая точка зрения на процесс проектирования состоит в том, что в качестве модели предметной области задачи или системы используются многоосновная алгебра теоретико-множественных структур данных. Различные алгебры такого типа получаются с использованием стандартных теоретико-множественных конструкций, которые в общих чертах рассмотрены в этом параграфе.

Основой для построения модели предметной области является базовая алгебра данных. Ее компоненты называются базовыми типами данных. Предполагается, что базовая алгебра данных задана с точностью до изоморфизма, т.е. базовые типы являются абстрактными типами данных. Это значит, что все функции и предикаты, которые можно определить на базовых типах данных, должны выражаться через исходные операции и предикаты, уже определенные на них изначально. На некоторых из базовых типов данных определено отношение аппроксимации. В этом случае операции должны быть монотонными и непрерывными. При необходимости отношение аппроксимации может быть определено с помощью стандартного расширения.

Над базовыми типами строится иерархия допустимых областей данных, из которых получаются структурные типы данных путем определения на этих областях операций и предикатов.

Рассмотрим основные способы образования допустимых областей данных.

1. Декартово произведение $A_1 \times \ldots \times A_n$ уже построенных областей $A_1, \ldots, A_n$. Напомним, что декартово произведение множеств рассматривается как n-арная операция над множествами.

2. Множество B^A всех отображений из A в B, где A и B – уже построенные области. Если A – бесконечное множество, то область B^A неконструктивна, и возникает вопрос, можно ли элементы этой области рассматривать как структуры данных. На самом деле при реализации в работе будут участвовать лишь те элементы области B^A, которые допускаются конструктивное задание. Однако на начальных этапах проектирования, может быть, еще не известны принципы выделения таких элементов. Кроме того, произвольное, даже не конструктивное отображение может быть аппроксимировано конструктивным. Как допустимый тип B^A может быть использовано для определения подтипов, т.е. множеств отображений, обладающих теми или иными свойствами.

3. Множество $\Gamma(A, B)$ всех непрерывных отображений из A и B, где A и B – уже построенные области с заданными на них отношениями аппроксимации. Если A – стандартное расширение области A_0, то $\Gamma(A, B)$ может быть отождествлено с множеством частичных отображений из A_0 в B.

4. Множество 2^A всех подмножеств множества A.

5. Множество A^* всех конечных последовательностей элементов области A. Это множество можно рассматривать и как множество всех слов в алфавите A. Множество A^* включает в себя также и пустую последовательность (пустое слово).

6. Если A и B – допустимые области, то допустимой является область $A \cup B$. Допускаются также бесконечные объединения допустимых областей. Если $A_1, A_2, \ldots$ – последовательность допустимых областей, то область $A = \bigcup_{i=1}^{\infty} A_i$ также является допустимой.

Следующие конструкции требуют применения средств логического языка для определения предикатов на областях данных. В качестве такого языка используем язык многосортного исчисления предикатов, интерпретированный на совокупности допустимых областей. Атомарные формулы тако-

го языка строятся с помощью предикатов, заданных на допустимых областях, предиката равенства и отношения аппроксимации. Эти предикаты применяются к выражениям, построенным с помощью операций, заданных на областях данных, констант и переменных. К атомарным формулам можно применять пропозициональные связки и ограниченные кванторы по переменным, пробегающим допустимые области данных. Зафиксируем некоторый язык такого рода и будем называть его базовым логическим языком.

7. Пусть $P(x)$ – формула базового логического языка, содержащая единственную свободную переменную x, пробегающую допустимую область A. Тогда можем построить новую допустимую область $B = \{x \in A \mid P(x)\}$. Эта область получается выделением из A элементов, обладающих свойством P. В частности, если A есть подмножество декартова произведения, то B определяет допустимое отношение между элементами соответствующих множеств.

8. Пусть $P(x)$ – формула базового языка, x пробегает множество $A \subset 2^B$ и существует единственный элемент $C \in A$ такой, что $P(C)$. Тогда C есть допустимая область.

9. Пусть $B \subset A^2$ – допустимое отношение, которое является отношением эквивалентности. Тогда фактор-множество A/B является допустимой областью. Если A есть Ω-алгебра, а B – отношение конгруэнтности, то A/B также есть Ω-алгебра.

10. Пусть D – допустимая Ω-алгебра (возможно, многоосновная), $A \subset D$ – допустимое подмножество множества D, $\mathcal{H}$ – совокупность тождеств алгебры D (не обязательно всех), X – допустимая область. Тогда допустимой является алгебра $T_\Omega(A, X, \mathcal{H})$, и если на ней определено допустимое отношение аппроксимации, то допустимой является алгебра $T^*_\Omega(A, X, \mathcal{H})$.

11. Пусть D – допустимая Ω-алгебра. Тогда любая Ω-алгебра, изоморфная алгебре D, также допустима. Эта конструкция дает возможность вводить новые абстрактные типы данных.

Для построения модели предметной области с помощью операций 1 – 11 строится некоторое конечное множество допустимых областей; на этих областях определяются операции и предикаты, которые превращают их в алгебры. Области базовой алгебры данных, а также те, которые получаются с помощью операций 1 – 6 и 10 – 11, называются основными. Ясно, что каждая допустимая область содержится в некоторой основной области. Операции, определенные на основных областях, переносятся на подобласти, если последние замкнуты относительно этих операций. Элементы допустимых областей называются *допустимыми объектами.* Поскольку, кроме абстрактных объектов, составляющих базовые области и области, полученные путем абстрагирования с помощью операции 11, среди допустимых объектов есть объекты, имеющие конкретную природу (функции, множества и т.п.), для них определен ряд специальных операций и предикатов, которые можно использовать для превращения допустимых областей в алгебры. Рассмотрим некоторые из таких универсальных операций.

Для декартова произведения областей определены проекции или селекторные операции. Если $z = (z_1, \ldots, z_m) \in A_1 \times \ldots \times A_m$, то $\varphi(z) = z_i$ есть проекция на i-ю координату. Возможны операции типа $\varphi(z) = (z_{i_1}, \ldots, z_{i_k})$.

Операция $\varphi(z_1, \ldots, z_m) = z$ дает возможность "собирать" вектор из его компонент. Мы не рассматриваем конкретные обозначения для соответствующих операций, которые могут выбираться исходя из соображений удобства и стиля проектирования. Например, в языке паскаль наименование типа $\underline{\text{record}}(A : t_1, B : \underline{\text{record}}(C : t_2, D : t_3))$ определяет область $M_1 \times (M_2 \times M_3)$, где M_1, M_2, M_3 – множества значений типов t_1, t_2, t_3 соответственно, и одновременно вводит обозначения для селекторных операций. Пусть $A(z_1, (z_2, z_3)) = z_1$, $B.C((z_1, (z_2, z_3)) = z_2$. Если $A_1, \ldots, A_n$ суть Ω-алгебры, то на $A_1 \times \ldots \times A_n$ определены операции прямого произведения (покомпонентные).

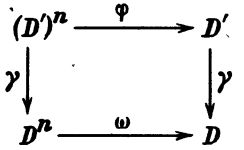
Для функциональных типов данных определена операция применения функции к аргументу: $f(x)$, где $f \in A^B$, $x \in B$. Если $f \in B^A$, $g \in C^B$, то $f \circ g \in C^A$ есть суперпозиция функций f и g. Предикат $\in$ определен для всех пар допустимых объектов. Значение выражения $x \in y$ есть истина тогда и только тогда, когда y – множество, а x – элемент этого множества. Аналогично определяется предикат $\subset$. Операции $x \cup y$, $x \cap y$, $x \setminus y$ определены, если x и y множества и т.п. Если A есть Ω-алгебра, а B – отношение конгруэнтности, то A/B также есть Ω-алгебра. Для множеств $T(A, X, \mathcal{H})$ и $T^*(A, X, \mathcal{H})$ можно рассматривать операции, которые выражаются через подстановки элементов алгебры D вместо переменных. Наконец, новые операции могут быть определены средствами базового логического языка. Если, например, $P(x, y, z)$ – формула базового языка и утверждение $\forall x \in A\ \forall y \in B\ \exists! z \in C P(x, y, z)$ истинно ($\exists!$ – существует единственный), то $P(x, y, z)$ определяет операцию типа $A \times B \to C$. Область $A = \bigcup_{i=1}^{\infty} A_i$, где $A_{i+1} = A_i^* \cup A_i$, есть область списочных структур. На ней определены операции, используемые в языке лисп: $\text{cons}(x, y)$ – присоединение элемента x к списку y; $\text{car}(x)$ и $\text{cdr}(x)$ – первый элемент и оставшаяся часть списка соответственно.

Здесь не ставилась задача дать исчерпывающего описания всех математических конструкций, которые могут быть полезными при описании областей данных, на которых действуют функции, вычисляемые проектируемыми системами. Однако то, что здесь описано, покрывает большинство ситуаций, встречающихся на практике. Все описанные конструкции могут быть формализованы в рамках некоторого логического языка, в который помимо традиционных логических средств связок и кванторов входят развитые средства конструирования объектов. Такой язык может использоваться для автоматизации исследовательского этапа проектирования алгоритмов, который включает в себя дедуктивные построения, связанные с обоснованием корректности конструирования типов, выводом полезных свойств исследуемых объектов, доказательством существования решения рассматриваемых задач.

Переход от моделей предметной области задачи к предметной области вычислительной системы сопровождается реализацией одних структур данных с помощью других. В общем случае реализация типа данных D с помощью типа D' задается реализующим отображением γ: $D' \to D$. Если такое отображение задано и $\gamma(d') = d$, то говорят, что структура данных d' реализует, или представляет, структуру d. Если каждая структура данных

имеет реализацию, т.е. γ является отображением D' на D, то реализация называется *полной*. Не требуется взаимной однозначности реализации. Более того, как правило, реализация многозначна. Например, естественные отображения множества составных в множество абстрактных составных, абстрактных составных в свободные и свободных в составные над алгеброй являются полезными примерами реализующих отображений, которые не являются взаимно однозначными.

Если γ: $D' \to D$ – реализующее отображение, а D является Ω-алгеброй, то желательно реализовать операции алгебры D на множестве D'. Говорят, что функция φ: $(D')^n \to D'$ реализует операцию ω: $D^n \to D$, если коммутативна диаграмма



Проще всего реализация получается, если D' также есть Ω-алгебра, а γ – гомоморфизм. Если D' есть Ω'-алгебра, то удобно реализовать операции алгебры D с помощью элементарных функций на D'. Допускается также реализация с помощью алгебраических D'-функций, которые должны быть в свою очередь реализованы алгоритмами.

Рассмотрим некоторые практически важные способы реализации теоретико-множественных структур данных.

Представление множеств. Предположим, что задано некоторое множество A и речь идет о представлении некоторых подмножеств этого множества (не обязательно всех). Основные операции, которые следует реализовать, – это предикаты принадлежности, включения и сравнения, булевы операции $\cup$, $\cap$, $\setminus$ (относительно взятия дополнения рассматриваемое множество подмножеств может быть не замкнуто).

Полезно также иметь алгоритмы перечисления множеств, которые можно представлять в виде оператора цикла:

для всех $x \in M$ **выполнить** $Q(x)$ **кц.**

Рассмотрим следующие способы представления: списки, определения, алгоритмы распознавания и алгоритмы порождения.

Списки используются для представления конечных множеств. Они могут быть также различных типов. Простой список можно рассматривать как элемент множества A^*. Можно различать списки с повторениями и без повторений. Если допускаются списки с повторениями, то операция объединения выражается через конкатенацию; все же остальные операции реализуются алгоритмами. При этом только определение принадлежности выполняется за линейное время; остальные операции выполняются за время, пропорциональное произведению числа элементов двух множеств, участвующих в операции (при условии что сравнение двух элементов множества A выполняется за ограниченное время). При отсутствии повторений время построения объединения также пропорционально произведению. Для ускорения времени выполнения операций применяют различные приемы. Простейший из них – упорядочение. Если множество A линейно упоря-

дочено некоторым отношением порядка, то элементы в списке можно считать упорядоченными по убыванию или возрастанию. Все операции над упорядоченными списками выполняются за линейное время.

Списки – это структуры данных с последовательным доступом. Их можно обрабатывать, двигаясь последовательно по элементам слева направо. Если списки задавать с помощью составных объектов, то можно организовать двухсторонние списки, по которым можно двигаться в одну и другую сторону, а также циклические списки. Возможно также использование списков с прямым доступом – одномерных массивов. Точнее, такой список задается парой, состоящей из числа элементов списка и одномерного массива, составленного из элементов множества A (функциональная структура данных, расположенная на множестве целых чисел). Упорядоченный список с прямым доступом из n элементов позволяет выполнить распознавание принадлежности за время порядка $n \log n$.

Следует также упомянуть представление множеств их характеристическими функциями, т.е. функциями, заданными на A и принимающими значение 1 для элементов, принадлежащих данному множеству, и 0 для элементов, не принадлежащих ему. Булевские операции над такими функциями можно выполнять параллельно, и это потребует всего лишь ограниченного времени.

Задание множеств с помощью определений позволяет работать с бесконечными множествами. Рассмотрим следующие типы определений: явные определения, порождающие схемы, рекурсивные и аксиоматические определения.

Явные определения выражают определяемые множества через уже известные с помощью булевских операций, операций выделения подмножества с помощью условия, а также, возможно, других операций. Явные определения можно рассматривать как составные объекты над соответствующей алгеброй. Для того чтобы эффективно выполнять разпознавание принадлежности и включения, их следует преобразовывать, выполняя, если это возможно, упрощения, приводя к каноническим формам и т.п. Для алгоритмов порождения явное задание множества удобно привести к форме $B \cap C$, где B содержит возможно меньшее число элементов. Тогда при порождении следует перебирать только элементы множества B, проверяя их принадлежность множеству C.

Порождающей схемой называется выражение $t(x_1, \ldots, x_n)$ некоторой (вообще говоря, многоосновной) алгебры, принимающее значения в A и зависящее от параметров $x_1, \ldots, x_n$. Для каждого из параметров задано множество допустимых значений. Схема порождает множество, состоящее из значений, которые принимает данное выражение при всевозможных допустимых подстановках параметров. Схемы удобно применять в случае, если требуется порождать элементы данного множества. Для распознавания принадлежности элемента множеству, заданному схемой, требуется решать уравнение $t(x_1, \ldots, x_n) = a$. Применение булевских операций приводит к смешанному способу задания множеств, использующему как средства явного определения, так и порождающие схемы.

Одним из важных видов рекурсивных определений является использование порождающих схем вида $t(x_1, \ldots, x_n, y_1, \ldots, y_m)$, где $y_1, \ldots, y_m$ – параметры, пробегающие заданные множества, а параметры $x_1, \ldots, x_n$

пробегают само порождаемое множество. Для рекурсивных определений должно быть задано некоторое начальное подмножество порождаемого множества. Иными словами, рекурсивное определение множества X состоит из нескольких условий вида $B \subset X$ и $x_1,...,x_n \in X \wedge p(x_1,...,x_n, y_1,...,y_m) \Rightarrow t(x_1,...,x_n, y_1,...,y_m) \in X$, где p – некоторое условие, t – выражение, принимающее значения в A, $y_1, ..., y_m$ – параметры, пробегающие заданные множества. С помощью рекурсивных определений, в частности, определяются подалгебры данной алгебры, порожденные заданными системами элементов. Рекурсивное задание множества может быть использовано для порождения его элементов. Что же касается алгоритмов распознавания принадлежности, то они могут быть сложными, и для их эффективной реализации потребуется дальнейшее исследование свойств заданного множества.

Аксиоматическое определение множества состоит из некоторой совокупности условий – аксиом, которым удовлетворяет это множество. В отличие от предыдущих случаев аксиоматическое определение задает множество неоднозначно. Вопрос о принадлежности или непринадлежности некоторого элемента такому множеству решается положительно только в том случае, если соответствующее утверждение является логическим следствием из аксиом этого множества. Однако нельзя, например, говорить об алгоритме, который перечисляет элементы данного множества.

Смысл распознающих и порождающих алгоритмов очевиден. Если эти алгоритмы представляются составными объектами, то необходим общий интерпретирующий алгоритм, который применяет алгоритм распознавания к конечным объектам и реализует порождение некоторых элементов множества в цикле.

Представление бинарных отношений. Основные способы представления бинарных отношений подобны способам представления множеств. В частности, всякое бинарное отношение можно представлять как множество пар. К представлению с помощью списков добавляется представление двойным списком. Двойной список – это список пар. Первый член пары – элемент соответствующего множества. Второй член – список всех элементов, которые находятся в данном отношении с первым членом пары.

Бинарное отношение на множестве A можно задавать также в виде ориентированного графа, вершинам которого сопоставлены элементы множества A. Такие графы можно задавать с помощью составных объектов.

Для специальных типов бинарных отношений целесообразно применять специальные способы задания. Можно выделить следующие типы отношений: транзитивные, квазипорядок, эквивалентности, частичные порядки, иерархии, функциональные отношения (функции).

Рассмотрим некоторые полезные способы задания отношений. Вместо транзитивного отношения можно задавать любым из рассмотренных способов отношение такое, что заданное отношение является его транзитивным замыканием. Такое задание может значительно сэкономить память для конечных отношений и упростить задание в других случаях. Для транзитивных отношений полезно также задавать порождающее множество, т.е. множество таких элементов, для каждого из которых найдется элемент, находящийся с ним в заданном отношении. Следует обратить внимание, что при таком задании транзитивного отношения время распознавания принадлежности увеличивается. Например, при задании отношения графом рас-

сматривамое отношение превращается в отношение достижимости, которое имеет сложность распознавания, пропорциональную числу дуг в этом графе.

Отношение квазипорядка (рефлексивное и транзитивное отношения) определяет эквивалентность $x \leqslant y \wedge y \leqslant x$ и частичный порядок на множестве классов этой эквивалентности. Удобно поэтому квазипорядок задавать двумя отношениями – эквивалентностью и частичным порядком. При этом классы могут задаваться либо сами по себе, либо своими представлениями.

Отношение эквивалентности удобно задавать как множество классов эквивалентности, определяемых этим отношением, или как функцию, которая каждому элементу области определения ставит в соответствие класс эквивалентности этого элемента.

Частичные порядки задаются отношением непосредственного следования или непосредственного предшествования. Иерархия – это частичный порядок, в котором каждый элемент, кроме максимальных, непосредственно подчинен в точности одному элементу. Иерархии (объединения деревьев) представляются так же, как частичные порядки, но для их обработки применяются более простые алгоритмы, чем для произвольных транзитивных отношений или частичных порядков.

Функциональные отношения кроме обычных способов можно задавать алгоритмами, вычисляющими значения. Их можно рассматривать также как функциональные структуры данных, расположенные на соответствующих областях, и использовать как массивы с прямым доступом.

Многие множества удобно задавать через функции, например, в виде множества значений функции или полного прообраза некоторого значения или значений, удовлетворяющих определенным условиям.

Многоместные предикаты и функции. Предикаты могут рассматриваться как бинарные отношения, составленные из пар, первый член которых принадлежит декартовому произведению. Аналогично многоместную функцию можно рассматривать как одноместную, определенную на декартовом произведении. Следует выделить способ задания $n+1$ местного предиката (или n-арной функции) в виде такого списка пар, где первый элемент пары есть элемент x области определения первого аргумента, а второй – n-местное отношение (или функция арности $n-1$, если $n > 1$), которое получается, если зафиксировать x в качестве первого элемента набора из $n+1$ элементов, находящихся в данном отношении.

Комментарии к главе 4

Теория функциональных структур данных, изложенная в § 1, была построена в работе [26]. Понятие периодически определенной функции введено В.М. Глушковым [16]. Соотношения в алгебре структур данных изучались В.П. Горшковым и С.П. Горлачем [35, 36]. Результаты §§ 2. 3 были опубликованы в [63, 64]. Изложение теории рекурсивных структур данных следует работам [7, 34, 60]. Теоретико-множественные структуры данных использовались в языке сетл [92] и Σ-программировании [33]. Другие направления в теории структур данных хорошо представлены в сборнике [38] и обзоре В.Е. Агафонова, заключающем этот сборник. Вопросы реализации теоретико-множественных структур данных рассматривались в [20]. В монографии [5] рассматриваются, в частности, вопросы использования структур данных для построения эффективных алгоритмов.

Часть II

ПРОЕКТИРОВАНИЕ

Глава 5

АРХИТЕКТУРА ЭВМ

§ 1. Структура неймановской ЭВМ

Вычислительная машина (компьютер или ЭВМ, поскольку подавляющее большинство современных компьютеров построено на основе электроники) представляет собой автоматическое устройство, предназначенное для обработки информации во взаимодействии с человеком по схеме, изображенной на рис. 5.1. Эта схема представляет собой трехкомпонентную дискретную систему, в которой две активные компоненты – человек и центральная часть ЭВМ – взаимодействуют, согласованно изменяя состояния общей части информационной среды, реализованной обычно с помощью устройств ввода-вывода и внешних запоминающих устройств. В традиционных структурах ЭВМ неймановского типа роль человека в процессе обработки информации сводится к минимуму. Его задача состоит лишь в установке или смене подготовленных заранее носителей информации (перфокарты, магнитная лента, диски и т.п.) и выдаче управляющих сигналов о запуске или прекращении работы ЭВМ.

Более подробно структура неймановской ЭВМ представлена на рис. 5.2. Внешняя компонента информационной среды здесь разложена на две составляющие – совокупность устройств ввода-вывода (ВВ) и внешних запоминающих устройств (ВЗУ). Центральная часть ЭВМ состоит из основной оперативной памяти (ОЗУ) и процессора, который подразделяется на

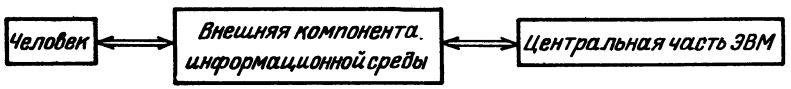


Рис. 5.1

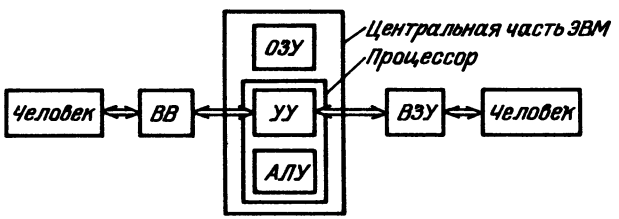


Рис. 5.2

устройство управления (УУ) и арифметико-логическое обрабатывающее устройство (АЛУ). Цель данного параграфа состоит в том, чтобы получить общий вид достаточно подробного алгоритмического описания ЭВМ традиционной архитектуры и обсудить некоторые вопросы алгоритмического и логического этапов проектирования таких ЭВМ. При этом будем двигаться сверху – вниз, начиная с самого общего описания, и последовательно его детализировать.

В наиболее общем виде алгоритм функционирования центральной части ЭВМ может быть представлен следующим образом:

ЦИКЛ
    Ждать ПУСК.
    Ввести начальную программу в ОЗУ.
    Выполнить программу.
КОНЕЦ ЦИКЛА.

Сигнал ПУСК вводится человеком с пульта управления (одно из устройств ввода-вывода). Простейший способ ввода начальной программы состоит в обращении к определенному раз и навсегда фиксированному устройству ввода, стандартизации размера начальной программы и места ее расположения в ОЗУ. Дальнейшая детализация алгоритма функционирования ЭВМ требует уточнения характеристик ОЗУ и понятия программы.

Для неймановской архитектуры типичным является использование в качестве ОЗУ однородной линейно адресуемой памяти. Логически такую память можно рассматривать как линейный массив M, составленный из машинных слов – двоичных кодов определенной длины. Элементы памяти называются ячейками. Содержимое ячейки – слово. Объем памяти, т.е. число ячеек, разрядность машинного слова и время выборки слова являются основными параметрами, характеризующими ОЗУ. Основная память обычно представляет собой память прямого доступа. Это означает, что время выборки практически не зависит от расположения слова, которое обычно выбирается за одно обращение к памяти.

Неймановская машина имеет фиксированные форматы команд и данных. Каждое слово может рассматриваться либо как представление единицы данных, либо как команда. Данные – это числа: целые со знаком или без знака, вещественные с фиксированной или плавающей запятой, двоичные коды фиксированной разрядности или последовательности символов. Каждая команда изменяет состояние памяти или осуществляет операцию ввода-вывода, т.е. обмен информацией между устройствами ввода-вывода или ВЗУ и основной памятью. Кроме того, каждая команда определяет однозначно своего преемника – следующую команду – или завершает выполнение программы. Таким образом, неймановская машина на уровне команд представляет собой последовательную ЭВМ.

В современных вариантах неймановской ЭВМ форматы команд и данных могут иметь различную длину, которая не обязательно совпадает с размером машинного слова. Так, например, в одном слове может быть записано больше одной команды или часть команды, а одна единица данных может занимать часть слова или несколько слов. Это приводит к тому, что адресуются не только слова, но и части слов, обычно байты – 8-битовые коды, а слово содержит целое число байтов. Поэтому адрес байта однозначно определяет и адрес слова, в котором этот байт находится.

Принципиальной особенностью неймановской ЭВМ является локальный характер изменения состояния памяти в процессе выполнения команд. Это изменение эквивалентно выполнению оператора присваивания

$$y := \omega(x_1, \ldots, x_m).$$

Операция ω – это либо арифметическая операция над числами, либо операция над двоичными кодами (обычно поразрядные операции и сдвиги). Адреса операндов $x_1, \ldots, x_m$ и результата y либо явно указаны в команде, либо вычисляются некоторым простым способом с использованием дополнительных операндов $z_1, \ldots, z_k$. Числа m и k обычно не превышают 2 – 3. Первоначально адреса задавались явно в команде, а их изменение производилось путем изменения (модификации) самих команд, которые ничем не отличаются от данных и могут служить операндами. Впоследствии, однако, идея использования индексных регистров и косвенной адресации полностью вытеснила необходимость изменять программу в процессе выполнения. Программы стали перемещаемыми.

Выполнение операции происходит в АЛУ. Поэтому прежде, чем она начнет выполняться, операнды должны быть переданы в АЛУ. Первоначально, когда число регистров сводилось к минимуму, все операнды передавались из основной памяти, а результат отправлялся в нее. Исключение составлял накапливающий сумматор. В современных ЭВМ в состав АЛУ может входить некоторое число регистров общего назначения. Они также могут служить как источниками операндов, так и местами хранения результатов выполнения команд. Кроме основного результата команда может вырабатывать некоторые дополнительные результаты, например признаки такие, как знак результата или признак переполнения, младшие разряды произведения и т.п. Эти дополнительные результаты остаются в АЛУ и могут быть использованы в следующих командах.

Выбор очередной команды обычно определяется естественным порядком их следования, т.е. адрес следующей команды получается прибавлением к адресу предыдущей некоторого числа, определяемого длиной прдыдущей команды (в простейшем случае 1). Этот порядок нарушается при выполнении команд условного или безусловного перехода. Адрес текущей команды хранится на регистре, который называется счетчиком команд (СК) и относится к устройству управления. К устройству управления относится также регистр команд (РК), в котором хранится выполняемая команда. Теперь можно алгоритм функционирования центральной части ЭВМ представить в следующем виде:

```
ЦИКЛ
    Ждать ПУСК.
    Ввести начальную программу в ОЗУ.
    Установить начальное значение СК.
    ЦИКЛ
        Прочитать очередную команду в РК.
        ЕСЛИ РК есть команда остановки, ТО
                завершить выполнение программы;
                вывести информацию о завершении;
                выйти из ЦИКЛА
```

КОНЕЦ ЕСЛИ.
Выполнить команду.
ЕСЛИ аварийное завершение, ТО
вывести анформацию об аварийном завершении;
выйти из ЦИКЛА
КОНЕЦ ЕСЛИ.
КОНЕЦ ЦИКЛА.
КОНЕЦ ЦИКЛА.

Дальнейшая детализация алгоритма требует более полного описания системы команд и способов взаимодействия с внешней средой. В качестве примера рассмотрим простой вариант архитектуры, отражающий некоторые черты современных ЭВМ неймановского типа и достаточный для начального обсуждения общих вопросов проектирования архитектуры ЭВМ. Рассматриваемую машину назовем ПНМ (простейшая неймановская машина).

Предположим, что оперативная память состоит из слов разрядности 32 бита. Каждое слово состоит из двух полуслов по 16 бит, а полуслово – из двух байтов. Адресуются байты, т.е. память представляется, в виде массива M: массив $(0:2^m - 1)$ байтов. Число определяет разрядность полного адреса байта. Представляя адрес в виде двоичного кода, получим, что старшие $m - 2$ разрядов этого адреса указывают номер слова, а младшие – номер байта в слове. Таким образом, байты нумеруются с нуля (рис. 5.3). Разрядность машинного слова, т.е. единицы данных, которая выбирается за одно обращение к памяти, пока не фиксируется, она может быть равной одному, двум или четырем байтам.

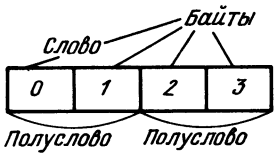


Рис. 5.3

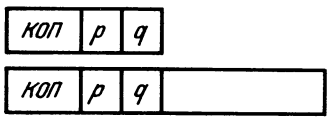


Рис. 5.4

Кроме оперативной памяти машина имеет память общих регистров, которая состоит из 16 32-битовых слов, имеющих номера от 0 до 15. Регистровую память будем также рассматривать как массив R: массив $(0:15)$ слов.

Команды машинного языка имеют два формата. Короткие команды занимают 2 байта (полуслово), длинные – 4 байта (слово). Форматы команд представлены на рис. 5.4. Они являются аналогами форматов RR и RX системы команд ЕС ЭВМ. Нулевой байт всегда содержит код операции. В первом байте записаны адреса (номера) двух регистров p и q, которые обычно используются для формирования операндов. Если команда длинная, то два последних байта содержат целое число d, называемое смещением и используемое для формирования адреса операнда, расположенного в оперативной памяти.

Любая команда, за некоторыми исключениями, о которых будет сказано особо, работает с двумя операндами r и z. Первым операндом всегда яв-

ляется переменная $r = R(p)$, значение которой хранится в регистре с номером p. Второй операнд z определяется одним из четырех способов:

1) $z = R(p)$;
2) $z = M(R(q) : R(q) + 3)$;
3) $z = M(R(q) + d : R(q) + d + 3)$;
4) $z = M(d : d + 3)$.

Здесь $M(a : b)$ обозначает двоичный код, составленный из байтов, расположенных последовательно в ОЗУ таким образом, что адрес первого равен a, адрес последнего – b.

Способ определения второго операнда задается первыми двумя байтами кода операции:: 00 – первый, 01 – второй, 10 – третий, 11 – четвертый. Таким образом, получается, что первый бит определяет формат команды (0 – короткая, 1 – длинная).

Все команды разделим на две группы: команды, выполняемые устройством управления, и команды АЛУ. Команды АЛУ имеют один из двух видов: $r := \omega(r, s)$ – бинарные операции и $r := \omega(z)$ – унарные операции.

Не обсуждая в деталях конкретные операции, отметим лишь некоторый стандартный минимум. К нему относятся арифметические операции (сложение, вычитание, умножение и деление) чисел с фиксированной и плавающей запятой, сложение и умножение неотрицательных целых, логические поразрядные операции, сдвиги кодов (вправо, влево) на конкретное число разрядов и т.п. К унарным операциям относятся такие, например, как изменение знака, нормализация, поразрядное инвертирование и т.п. Будем предполагать, что в АЛУ имеется двухразрядный регистр ПР (признак результата), который принимает одно из четырех значений – 0, 1, 2, 3 – в зависимости от условий, которым удовлетворяет результат выполнения операции в АЛУ. Например, в случае арифметических операций значение регистра ПР может иметь следующий смысл:

0 – результат равен 0;
1 – результат больше 0;
2 – результат меньше 0;
3 – переполнение разрядности.

Значение регистра ПР может быть использовано следующей командой условного перехода.

Команды устройства управления разделим на следующие группы:

– команды обмена между регистровой и оперативной памятью;
– команды управления;
– команды ввода-вывода.

Команды обмена возможны двух типов:

$r := z$ (ЗАГРУЗКА);
$z := r$ (ПЕРЕСЫЛКА).

Команды управления имеют вид

ЕСЛИ α ТО НА z КЕ (ПЕРЕХОД).

Условие перехода кодируется четырьмя битами, записанными в разрядах с номерами 8 – 11, т.е. там, где обычно записывается номер p регистра, в котором хранится первый операнд. Условие α есть дизъюнкция: $\alpha = \alpha_0 \wedge \beta_0 \vee \alpha_1 \wedge \beta_1 \vee \alpha_2 \wedge \beta_2 \vee \alpha_3 \wedge \beta_3$, где $\alpha_i = 1 \Longleftrightarrow$ ПР $= i$, $\beta_0\beta_1\beta_2\beta_3$ – код, записанный на месте p. Таким образом, если например,

после выполнения арифметической операции требуется выполнить переход при результате, равном 0, следует взять $p = 8\,(1000)$, если же следует перейти, когда результат больше или равен 0, то $p = 12\,(1100)$. Следующая команда дает возможность организовать переход с возвратом, а также определить адрес команды, следующей за той, которая выполняется в текущий момент времени:

$r :=$ СК $+ u$; НА z (ПЕРЕХОД С ВОЗВРАТОМ).

Здесь $u = 2$, если выполняемая команда короткая, и $u = 4$, если длинная. Следующая команда позволяет выполнить команду, которая является значением операнда z:

Выполнить команду z (ВЫПОЛНИТЬ).

Команда ВЫПОЛНИТЬ может быть использована, например, для программирования отладочных программ. Следующая команда полезна при программировании циклов ДЛЯ $i := 1$ ДО n ВЫП...

$r := r - 1$; ЕСЛИ $r = 0$ ТО НА z КЕ (ПЕРЕХОД ПО СЧЕТЧИКУ).

Выполнение команды остановки

СТОП

вызывает прекращение работы машины.

Для описания команд ввода-вывода предположим, что ЭВМ оснащена следующими периферийными устройствами:

– устройство ввода с перфокарт (ПК);
– устройство вывода на бумагу (АЦПУ);
– экранный пульт (ЭКР);
– магнитный диск (МД);
– магнитная лента (МЛ).

Названия команд:

ВВОД ПК;
ВЫВОД АЦПУ;
ВЫВОД ЭКР;
ВВОД ЭКР;
ВЫВОД МД;
ВВОД МД;
ВЫВОД МЛ;
ВВОД МЛ;
РАЗМЕТКА МД;
РАЗМЕТКА МЛ;
ПЕРЕМОТКА МЛ.

Во всех этих командах предполагается, что значением операнда z является адрес в ОЗУ, с которого начинается область ввода-вывода, т.е. область, из которой берутся данные при выводе и куда помещаются данные при вводе. Длина этой области, а также все остальные данные, необходимые для выполнения операции ввода-вывода, определяются значением операнда r.

Название команды однозначно определяется 6-битовым кодом, который получается из кода операции отбрасыванием первых двух битов, характеризующих формат и способ образования второго операнда. В дальнейшем название команды будем отождествлять с соответствующим кодом.

Теперь после предварительных замечаний о системе команд можно сделать следующий шаг в уточнении алгоритма функционирования центральной части ПНМ (уточняющие комментарии даны ниже).

```
АЛГОРИТМ ПНМ:
  ЦИКЛ
    ЖДАТЬ ПУСК.
    ПРИНЯТЬ ВХОД ПК В М(0:79);
    СК := 0;
    ЦИКЛ
      РК(0:15) := М(СК:СК+1);
      ЕСЛИ РК(0) = 1 ТО РК (16:31) := М(СК+2:СК+3),
      СК := СК + 4
      ИНАЧЕ СК := СК + 2 КЕ.
```

Анализ команды и подготовка операндов:

```
ЕСЛИ КОП(2:7) = СТОП ТО вывести информацию о завершении,
        выйти из ЦИКЛА ИНАЧЕ
    ЕСЛИ КОП(2:7) = ПЕРЕСЫЛКА ТО
      ЕСЛИ КОП(0:1) = 0 ТО R(q) := R(p) ИНАЧЕ
        ЕСЛИ КОП(0:1) = 1 ТО M(R(q): R(q) + 3) := R(p) ИНАЧЕ
          ЕСЛИ КОП(0:1) = 2 ТО М(R(q) + d : R(q) + d + 3) := R(p)
          ИНАЧЕ М(d : d + 3) := R(p) КЕ,
НА ЗАВЕРШЕНИЕ ИНАЧЕ
  ЕСЛИ КОП(0:1) = 0 ТО z := R(q) ИНАЧЕ
  ЕСЛИ КОП(0:1) = 1 ТО z := М(R(q): R(q) + 3) ИНАЧЕ
  ЕСЛИ КОП(0:1) = 2 ТО z := М(R(q) + d : R(q) + d + 3)
  ИНАЧЕ z := М(d : d + 3) КЕ
КОНЕЦ ЕСЛИ.
```

Выполнение команды:

```
ВЕТВЛЕНИЕ ПО КОП(2:7):
  ЗАГРУЗКА → R(p) := z;
  ПЕРЕХОД → ЕСЛИ ПР = 0 ∧ РК(8) = 1 ∨ ПР = 1 ∧ РК(9) = 1
    ∨ ПР = 2 ∧ РК(10) = 1 ∨ ПР = 3 ∧ РК(11) = 1 ТО СК := z КЕ;
  ПЕРЕХОД С ВОЗВРАТОМ → R(p) := СК, СК := z;
  ВЫПОЛНИТЬ → РК := М(z : z + 3), НА анализ команды и
    подготовку операндов;
  ПЕРЕХОД ПО СЧЕТЧИКУ → НАЧАЛО
    R(p) := R(p) − 1; ЕСЛИ R(p) = 0 ТО СК := z КЕ
КОНЕЦ;
  ВВОД ПК → ВЫПОЛНИТЬ R(p) РАЗ
  ПРИНЯТЬ ВХОД ПК В М(z : z + 79); z := z + 80;
  КОНЕЦ ЦИКЛА:
```

ВЫВОД АЦПУ → ВЫПОЛНИТЬ $R(p)$ РАЗ
ПЕРЕДАТЬ М(z) В ВЫХОД АЦПУ
$z := z + 1$
КОНЕЦ ЦИКЛА;
ВЫВОД ЭКР → ЭКР($R(p)$) := М($z : z + 79$);
ВВОД ЭКР → М($z : z + 959$) := ЭКР;
ВЫВОД МД → МД($R(p)$ (0), $R(p)$ (1),
$R(p)$ (2)) := М($z : z + R(p)$ (3) − 1);
ВВОД МД → М($z : z + R(p)$ (3) − 1) := МД($R(p)$ (0),
$R(p)$ (1), $R(p)$ (2));
ВЫВОД МЛ → ПРОЧИТАТЬ МЛ М($z : z + R(p)$ (3) − 1);
ВВОД МЛ → ЗАПИСАТЬ МЛ М($z : z + R(p)$ (3) − 1);
РАЗМЕТКА МД → . . . ;
РАЗМЕТКА МЛ → . . . ;
ПЕРЕМОТКА МЛ → . . . ;
БИН 0 → $R(p) := \omega_0(R(p), z)$, . . . ;
БИН 1 → $R(p) := \omega_1(R(p), z)$, . . . ;
. . .
БИН 31 → $R(p) := \omega_{31}(R(p), z)$, . . . ;
УН 0 → $R(p) := \delta_0(z)$, . . . ;
. . .
УН 7 → $R(p) := \delta_7(z)$, . . . ;
КОНЕЦ ВЕТВЛЕНИЯ.
Завершение:
ЕСЛИ аварийное завершение ТО вывести информацию об аварийном завершении, выйти из ЦИКЛА
КОНЕЦ ЕСЛИ.
КОНЕЦ ЦИКЛА.
КОНЕЦ ЦИКЛА.

В этом описании центральная часть ПНМ рассматривается как алгоритмический модуль, связанный с внешним миром периферийными устройствами, которые на данном уровне описания рассматриваются как входные-выходные очереди и общая память. Устройство ввода с перфокарт рассматривается как входная очередь ВХОД ПК, элементами которой являются 80-байтные двоичные последовательности. Каждая такая последовательность соответствует одной перфокарте с 80 колонками. Устройство вывода на бумагу рассматривается как выходная очередь, элементами которой являются байты. Каждый байт представляется некоторым символом, который при выводе печатается на бумаге. Некоторые байты играют роль управляющих сигналов для печатающего механизма — таких, как сигнал перехода на новую строку или пропуска строки.

Экранный пульт — это общая память, имеющая структуру ЭКР : массив (1 : 12) массивов (1 : 80) байтов. Магнитный диск также рассматривается как общая память, имеющая структуру МД : массив ($1 : m_1$, $1 : m_2$, $1 : m_3$) записей. Здесь m_1 — число цилиндров, m_2 — число дорожек, m_3 — число

записей на дорожке. Каждая запись есть последовательность байтов фиксированной длины. Структура МД может меняться. При фиксированном числе цилиндров и дорожек может меняться число записей на дорожке и длина записи. Это изменение происходит в результате выполнения команды РАЗМЕТКА МД. Наконец, магнитная лента представляется в виде двух взаимосвязанных очередей. Одна из этих очередей входная ВХОД МЛ, другая выходная ВЫХОД МЛ. Элементами обеих очередей являются записи постоянной длины, составленные из байтов. Обращение к МЛ выполняется с помощью операторов ПРОЧИТАТЬ МЛ X и ЗАПИСАТЬ МЛ X. Первый из этих операторов эквивалентен последовательности:

ПРИНЯТЬ ВХОД МЛ В X;
ПЕРЕДАТЬ X В ВЫХОД МЛ.

Второй эквивалентен последовательности:

ПЕРЕДАТЬ X В ВЫХОД МЛ;
ПРИНЯТЬ ВХОД МЛ В Y.

Эквивалентность здесь имеется в виду только по результату. При этом переменная Y играет чисто формальную роль. Ее значение после выполнения оператора недоступно. Команда РАЗМЕТКА МЛ меняет размер записи на МЛ. Команда ПЕРЕМОТКА переписывает ВЫХОД МЛ на ВХОД МЛ. После ее выполнения выход становится пустым, и если перед выполнением ВЫХОД МЛ = $x_1 \ldots x_n$, а ВХОД МЛ = $x_{n+1} \ldots x_m$, то после выполнения ВХОД МЛ = $x_1 \ldots x_m$. Это сложное действие, разумеется, должно выполняться некоторым внешним модулем, связанным с МЛ.

В алгоритме считается, что начальная программа всегда располагается на одной перфокарте и помещена в начале входной очереди устройства ввода с перфокарт в момент приема сигнала ПУСК, который рассматривается как входной сигнал без памяти. Символы p и q обозначают соответствующие части команды: $p = \text{РК}(8:11)$, $q = \text{РК}(12:15)$. Регистр команд РК рассматривается как последовательность битов. В других случаях слово рассматривается как последовательность из четырех байтов. В ряде случаев, ясных из контекста, последовательность битов рассматривается как представление целого неотрицательного числа. Поэтому к двоичным кодам могут применяться арифметические операции – сложение и вычитание (если результат остается неотрицательным).

Оператор ВЕТВЛЕНИЕ ПО x: $a_1 \to P_1; \ldots; a_m \to P_m$ КОНЕЦ ВЕТВЛЕНИЯ эквивалентен условном оператору:

ЕСЛИ $x = a_1$ ТО P_1 ИНАЧЕ
...
ЕСЛИ $x = a_m$ ТО P_m КЕ.

Выполнение команд разметки и перемотки ленты требует более подробного описания внешних запоминающих устройств. Поэтому соответствующие части алгоритма пропущены. Разрядность кода операции и способ использования его первых двух битов показывает, что язык ПНМ может иметь до 64 команд. Предполагается, что 32 из этих команд – команды выполнения бинарных операций и 8 команд – унарных операций. Эти команды названы здесь БИН 0, БИН 1, ..., УН 0, УН 1, ... Каждая из команд АЛУ вырабатывает не только основной результат в

регистре $R(p)$, но также и некоторые дополнительные результаты, в частности значение регистра ПР. Это показано многоточиями в операторах выполнения команд АЛУ. Соответствующие уточнения должны быть сделаны на следующем уровне описания, когда будет точно определен набор операций, выполняемых в АЛУ.

Алгоритм ПНМ является неполным также из-за того, что в нем отсутствует явное определение условий аварийного окончания выполнения команды. Кроме переполнения разрядной сетки или деления на 0 аварийные ситуации возникают, например, при выполнении операций ввода-вывода, если размер области обмена в ОЗУ не соответствует размерам записей на МД и МЛ. Соответствующие уточнения также должны быть сделаны на следующих уровнях описания.

Упражнения

1. Дать точное описание того, как выполняется команда ВЫПОЛНИТЬ.
2. Рассмотреть какую-нибудь известную систему команд АЛУ и уточнить алгоритм ПНМ, включив в него алгоритмы выполнения этих команд.
3. Улучшить алгоритмический проект ПНМ, варьируя две основные характеристики: возможности системы команд и сложность алгоритма функционирования.
4. Написать алгоритм функционирования любой из известных вам ЭВМ, пользуясь информацией, представленной в документации.

§ 2. Логическое проектирование алгоритмических модулей

На этапе логического проектирования аппаратных средств систем преобразования информации в качестве исходного материала выступает алгоритмическое описание системы в виде сети из алгоритмических модулей. Часть модулей этой сети представляет собой модели уже спроектированных компонент. К ним относятся, в частности, устройства ввода-вывода, запоминающие устройства, различного рода датчики и преобразователи сигналов для систем реального времени, элементы системы связи и т.п. Некоторые из модулей являются компонентами модели внешней среды, с которой взаимодействует разрабатываемая система, включая взаимодействие с человеком. Сеть может быть представлена как открытая система. В этом случае модель внешней среды может быть задана неявно в виде требований и ограничений, которым должны удовлетворять процессы изменения входных и выходных компонент.

Результатом логического проектирования является синхронная или асинхронная сеть из автоматов, компоненты которой являются моделями элементов используемого базиса. В качестве промежуточного описания проектируемого устройства, не зависящего или слабо зависящего от элементного базиса и технологии изготовления интегральных схем, может использоваться сеть из алгоритмических модулей базовых типов. Будем различать три основных базовых типа модулей – функциональные, управляющие и операционные.

Каждый из базовых типов модулей представляет собой автомат, входные и выходные переменные которого являются булевыми векторами. Различаются два типа входных и выходных переменных – управляющие и информационные. Управляющие переменные являются одноразрядными булевыми переменными с нейтральным значением 0, и при асинхронном

взаимодействии компонент их можно воспринимать только в состоянии ожидания. Информационные переменные могут быть переменными с памятью или без памяти. Переменные с памятью сохраняют свое значение между двумя очередными присваиваниями.

Функциональные модули – это автоматы без памяти. Если $x_1, \ldots, x_n$ – входные, а $y_1, \ldots, y_m$ – выходные переменные функционального модуля, то его функционирование определяется системой соотношений

$$y_1 = f_1(x_1, \ldots, x_n),$$
$$\ldots\ldots\ldots\ldots\ldots$$
$$y_m = f_m(x_1, \ldots, x_n),$$

где $f_1, \ldots, f_m$ – векторные булевы функции. Разумеется, эту систему можно заменить системой обычных булевых функций, если перейти от булевых векторов к их двоичным компонентам. Выходные переменные функционального модуля являются переменными без памяти. При этом если все входные функционального модуля информационные, то и выходные также информационные. Если же среди входных есть хоть одна управляющая, то все выходные – тоже управляющие.

Управляющие компоненты – это автоматы с задержкой, у которых все выходные сигналы (переменные) управляющие, а входные могут быть как управляющими, так и информационными. Управляющие компоненты не имеют внутренних переменных, и при описании их с помощью схем программ состояния схемы совпадают с внутренними состояниями автомата.

Операционные модули обладают внутренней (регистровой) памятью, состоящей из векторных булевых переменных $z_1, \ldots, z_m$. Часть из этих переменных является одновременно выходными переменными с памятью. Операционный модуль имеет входные управляющие переменные $x_1, \ldots, x_n$ и информационные переменные $u_1, \ldots, u_k$.

Алгоритм функционирования операционного модуля имеет следующий вид:

НАЧАЛО
L: ЖДАТЬ $x_1, \ldots, x_n \neq 0$:
ЕСЛИ v_1 ТО P_1 КЕ,
ЕСЛИ v_2 ТО P_2 КЕ,
. . . . . .
ЕСЛИ v_l ТО P_l КЕ, НА L.
КОНЕЦ

Здесь $v_1, \ldots, v_l$ – условия, зависящие только от входных переменных, $P_1, \ldots, P_l$ – операторы присваивания, использующие входные переменные и вырабатывающие внутренние переменные. Для каждого операционного модуля определены допустимые наборы значений управляющих сигналов, и если условия v_i и v_j непротиворечивы, т.е. могут одновременно быть истинными, то операторы P_i и P_j должны быть совместимыми.

Таким образом, операционный модуль изменяет свое внутреннее состояние только в результате воздействия на него управляющих переменных. Это изменение происходит за один такт и состоит в выполнении операторов присваивания. Следует сказать, какие допускаются операции над значения-

ми внутренних переменных. Предполагается, что все внутренние переменные рассматриваются как структуры данных, расположенные на некоторых областях и принимающие значения в двухэлементной булевой алгебре, а каждая операция определяется как периодически определенная функция над соответствующей алгеброй структур данных. В качестве областей расположения обычно рассматриваются одномерные или двумерные целочисленные решетки, а сдвиги — это прибавление или вычитание констант.

Сеть, составленная из модулей базовых типов, называется одноуровневой базовой сетью. Одноуровневая сеть может быть синхронной или асинхронной. Базовая сеть уровня n — это синхронная или асинхронная сеть, компонентами которой являются модули базовых типов или базовые сети уровня $n - 1$. Понятия базового типа и базовой сети допускают некоторые изменения, связанные с конкретной технологией проектирования и зависящие от элементной базы. Определяющим является то, что для базовых типов модулей должна быть стандартная методика технического проектирования, позволяющая реализовать эти модули с помощью заданной элементной базы. Рассмотренные здесь базовые типы допускают реализацию как на схемах малой и средней интеграции, так и на больших и сверхбольших интегральных схемах.

Для реализации управляющих компонент используются методы структурного синтеза конечных автоматов, аналогичные тому, который использован в § 7 гл. 1 первой части для доказательства теоремы о полноте. Кроме того, для реализации автоматов могут быть использованы различного типа пассивные запоминающие устройства. Код состояния, который хранится на регистре состояния управляющей компоненты, используется как адрес ячейки ПЗУ (пассивного ЗУ), а содержимое этой ячейки есть набор выходных сигналов и информация, которая требуется для определения следующего состояния. Такой способ реализации обычно называется микропрограммным. Он позволяет экономно реализовать управляющие модули с большим числом состояний и хорошо реализуется на БИС и СБИС.

Использование ПЗУ применяется также при реализации сложных функциональных блоков. Операционный блок — это набор одномерных или двумерных регистров с функциональными схемами, реализующих периодически определенные преобразования и управляемых входными управляющими сигналами.

Для того чтобы преобразовать алгоритмический модуль M в композицию модулей базовых типов, необходимо прежде всего закодировать (реализовать) структуры данных, представляемые его внутренними переменными, с помощью булевых векторов или матриц таким образом, чтобы операции над исходными структурами данных превратились в периодически определенные функции. В силу ограниченности размеров данных такое преобразование всегда возможно. Задача состоит в том, чтобы размеры областей однородности были как можно больше, поскольку от этого зависит сложность коммутации и технологии изготовления схем (особенно для технологии БИС и СБИС). Чем больше области однородности, тем лучше технологические характеристики. После приведения структур данных к булевым можно разделить модуль M на две компоненты — управляющую и операционную, пользуясь следующим общим приемом.

Рассмотрим все простые операторы присваивания, которые встречаются в алгоритме функционирования модуля. Пусть это будут операторы $r_1 := t_1, \ldots, r_m := t_m$. Введем m управляющих переменных $u_1, \ldots, u_m$ и операционный блок Q с алгоритмом функционирования

ЦИКЛ
ЖДАТЬ $u_1, \ldots, u_m$:
ЕСЛИ u_1 ТО $r_1 := t_1$ КЕ,
. . . . . .
ЕСЛИ u_m ТО $r_m := t_m$ КЕ,
КОНЕЦ ЦИКЛА.

Теперь из алгоритма M удаляем все внутренние переменные, переносим их в модуль Q и заменяем каждый из операторов $r_i := t_i$ оператором $u_i := 1$. Переменные $u_1, \ldots, u_m$ будут выходными управляющими переменными преобразованного модуля M и входными переменными модуля Q. Кроме того, входными переменными модуля Q будут также все входные переменные M, от которых зависят правые части присваиваний. Операторы, выполняемые внутри операционного модуля, называются микрооператорами или иногда микрооперациями, а сигналы $u_1, \ldots, u_m$ – сигналами соответствующих микрооператоров. Сложность реализации этих микрооператоров определяет сложность операционного модуля. Очевидно, рациональный выбор структуры микрооператоров может быть выполнен путем эквивалентных преобразований алгоритма функционирования модуля еще до выделения операционного блока.

Стандартные модули – такие, как память или устройства ввода-вывода, – должны быть выделены еще раньше, а обращения к ним должны быть выражены через алгоритмы взаимодействия модуля M с этими устройствами через управляющие и информационные переменные в соответствии со структурой этих устройств. Реализация общей памяти и очередей зависит от природы запоминающих устройств, используемых для их реализации, и способов управления (разрешение конфликтов). При реализации общей памяти могут также использоваться характеристики алгоритмов, взаимодействующих через нее. В частности, если обращения к общей памяти синхронизованы внешним образом, т.е. любые одновременные обращения непротиворечивы, то такая память может быть реализована просто как операционный модуль с операциями записи и чтения.

Важной проблемой является обеспечение связей между регистрами внутри операционного модуля. Эти связи могут быть сделаны явными, если операционный модуль расщепить на несколько модулей, каждый из которых имеет только одну внутреннюю переменную. После такого расщепления каждый из модулей должен быть связан информационными входами с выходами тех модулей, от которых зависят правые части присваиваний, выполняемых внутри данного модуля. Тот факт, что при выполнении различных микрооператоров используются не все, а лишь некоторые из входных переменных, позволяет уменьшить число входов в данный модуль путем введения специального коммутационного модуля, который подключает ко входам операционного модуля выходы других модулей, необходимые для выполнения данной операции. Поскольку коммутационный модуль требует собственного управления, введение его несколько замед-

ляет выполнение микрооператоров, но позволяет в ряде случаев экономить аппаратуру.

Дальнейшее проектирование операционных модулей может быть проведено путем разделения регистров как наборов запоминающих элементов от функциональных схем, реализующих микрооперации. Однако это разделение зависит уже от структуры используемых стандартных типов запоминающих элементов, которые определяются не только типами используемых триггеров, но и логическими схемами, которые заранее привязаны к их входам.

Еще один вопрос логического проектирования заслуживает внимания. Это проблема синхронизации работы управляющей и операционной компонент алгоритмического модуля. На уровне модели эти устройства работают синхронно, т.е. в едином дискретном времени. При реализации, естественно, требуется учитывать реальные задержки, которые происходят в функциональных схемах. В конечном счете достаточно выбрать частоту синхронизирующих сигналов, определяющих в свою очередь частоту переключения управляющей компоненты таким образом, чтобы за одно переключение – один рабочий такт – в операционных устройствах успела выполниться наиболее длинная микрооперация. Можно, конечно, управлять и более тонко, выбрав рабочую частоту по длине наиболее короткой микрооперации и учитывая длительности более длинных. В ряде случаев можно добиться ускорения работы модуля путем перекрытия по времени выполнения соседних микрооператоров.

С проблемой синхронизации связаны также информационные зависимости в микрооператорах. Если правая часть присваивания $r := t$ зависит от переменной r, то при выполнении присваивания за один такт есть риск, что правая часть изменится еще до окончания присваивания. Другая вредная зависимость, которая также может привести к недетерминированному переходу – это зависимость условия u от переменной r в условном операторе ЕСЛИ u ТО $r := t, \ldots$ ИНАЧЕ $\ldots$ Попытка выполнить этот оператор за один такт может привести к тому, что условие u изменит свое значение еще до окончания выполнения данного оператора. Проблемы развязывания такого рода зависимостей могут быть решены путем эквивалентных преобразований алгоритма – разнесения по времени проверки условия и присваивания – или введением дополнительных пересылок для того, чтобы правая часть присваивания не зависела от левой. Проигрыш по времени может быть компенсирован перекрытием микроопераций при более надежном функционировании аппаратуры.

Рассмотрим вопросы логического проектирования центральной части ПНМ, описанной в § 1. Прежде всего необходимо перейти к более точным моделям периферийных устройств и модели ОЗУ.

Рассмотрим для примера устройство ввода с перфокарт УВПК. Грубая модель этого устройства – очередь перфокарт. Каждая перфокарта – это одномерный массив из 80 байтов, а каждый байт – последовательность из 8 битов. Более подробная модель этого устройства учитывает, что прием из очереди происходит последовательно по тактам. На каждом такте принимается параллельно по восьми каналам один байт. Начало приема перфокарты вызывается входным управляющим сигналом. Таким образом, УВПК можно рассматривать как алгоритмический модуль, внеш-

нее описание которого представлено на рис. 5.5. Здесь КПК есть входная очередь перфокарт (колода перфокарт), СМК – общая одноразрядная переменная (смена колоды). Перед добавлением новых перфокарт (передача в очередь) значение СМК должно быть установлено в 1. Выходные переменные ЗАКР (закрыто), ПУСТО и БАЙТ – переменные с памятью. Переменные ПРИЕМ и ГОТОВ – входная и выходная управляющие переменные. В скобках указаны разрядности соответствующих переменных.

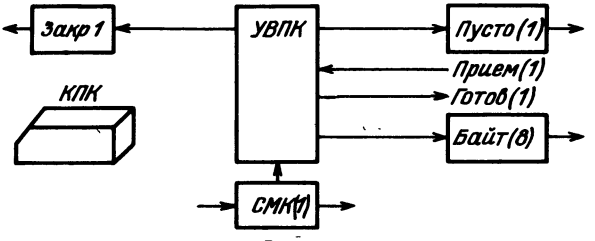


Рис. 5.5

Переменные ПУСТО, БАЙТ и СМК являются входными переменными центральной части ПНМ, а ГОТОВ – ее выходная переменная.

Алгоритм функционирования УВПК можно представить следующим образом:

```
АЛГОРИТМ УВПК.
  НАЧАЛО
    ПУСТО := 0;
    ЦИКЛ
      ЖДАТЬ СМК = 0;
        ЕСЛИ КПК = e ТО ПУСТО := 1 ИНАЧЕ ПУСТО := 0 КЕ.
      ЦИКЛ.
        ЖДАТЬ ПРИЕМ = 1 v СМК = 1:
          ЕСЛИ СМК = 1 ТО ПУСТО := 1, ВЫЙТИ ИЗ ЦИКЛА КЕ;
          ПРОЧИТАТЬ КПК В K;
          ЕСЛИ КПК = e ТО ПУСТО := 1 КЕ;
        ДЛЯ I := 1 ДО 80 ВЫПОЛНИТЬ
          БАЙТ := К(1);
          ГОТОВ := 1;
          ЗАДЕРЖКА
        КЦ
    КОНЕЦ ЦИКЛА
  КОНЕЦ ЦИКЛА
КОНЕЦ.
```

Оператор ЗАДЕРЖКА эквивалентен тождественному оператору, но выполняется за несколько тактов. Количественно величина задержки определяется реализацией. Логически она необходима для того, чтобы принимающее устройство успело обработать очередной байт. Технически время задержки определяется скоростью движения перфокарты во время считывания.

Рассмотрим теперь структуру ОЗУ. Проще всего представить память в виде операционного модуля, который представляет массив M. Внешнее описание такого модуля изображено на рис. 5.6. Его алгоритм функционирования представляется следующим циклом:

```
АЛГОРИТМ ОЗУ : ЦИКЛ
  ЖДАТЬ ЗАП, ЧТН:
    ЕСЛИ ЗАП ТО
      ЯЧК := M(АДР : АДР + k) КЕ,
    ЕСЛИ ЧТН ТО
      M(АДР : АДР + k) := ЯЧК КЕ
        КОНЕЦ ЦИКЛА.
```

Из этого описания видно, что ячейка памяти, т.е. единица обмена, имеет длину k байтов ($k = 1, 2, 4$). Переменная ЯЧК является общей переменной, и обращение к памяти выполняется по алгоритму, зависящему от k и m. Предположим, например, что $k = 4$, а $m = 32$. Тогда оператор

РК(0:15) := М(СК:СК + 1)

может быть реализован следующей последовательностью операторов:

```
АДР := СК;
ЧТН := 1;
ЗАДЕРЖКА;
РК(0:15) := ЯЧК(0:15).
```

Заметим, что при $k = 4$ последовательность операторов команды может быть упрощена следующим образом:

```
АДР := СК;
ЧТН := 1;
ЗАДЕРЖКА;
РК := ЯЧК;
ЕСЛИ РК(0) = 1 ТО СК := СК + 4 ИНАЧЕ СК := СК + 2 КЕ.
```

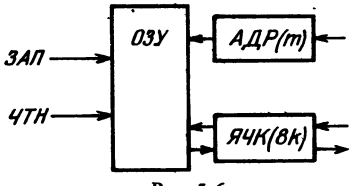


Рис. 5.6

Задержка необходима, если время обращения к памяти больше, чем длительность рабочего такта управляющего модуля ПНМ.

Исходя из описанных моделей УВПК и ОЗУ, рассмотрим теперь реализацию оператора

ПРИНЯТЬ ВХОД ПК В М($z : z + 79$).

Эта реализация может быть выполнена с помощью следующего алгоритма:

```
НАЧАЛО
  ЦИКЛ
    ЖДАТЬ ПУСТО = 0;
```

```
    ПРИЕМ := 1;
    ЖДАТЬ ГОТОВ = 1 ∨ ПУСТО = 1:
      ЕСЛИ ГОТОВ = 1 ТО ВЫЙТИ ИЗ ЦИКЛА КЕ
  КОНЕЦ ЦИКЛА;
  ПОВТОРИТЬ 20 РАЗ
    ДЛЯ I := 0 ДО 3 ВЫПОЛНИТЬ
      ЯЧК(I * 8: (I + 1) * 8 – 1) := БАЙТ;
      ЖДАТЬ ГОТОВ = 1
    КЦ;
    ЗАП := 1;
    ЗАДЕРЖКА, z := z + 4
  КОНЕЦ ЦИКЛА.
КОНЕЦ
```

Аналогичным образом вводятся и используются модели других периферийных устройств.

Следующий вопрос, который требует уточнения, – это выбор алгоритмов выполнения бинарных и унарных операций. Уточнение этих алгоритмов связано с решением одной из центральных проблем проектирования аппаратуры – с выбором системы микроопераций (микрооператоров). Самое прямое решение состоит в том, что одна операция машинного языка соответствует одной микрооперации и выполняется, следовательно, за один рабочий такт. Такое решение позволяет получить максимальное быстродействие, но связано с большими затратами аппаратуры. Поэтому обычно выбирают некоторое промежуточное решение, добиваясь в конечном счете разумного соотношения между производительностью и стоимостью аппаратуры.

Поскольку основными операциями машины фон Неймана являются арифметические операции, то в качестве основных микроопераций обычно используют сложение целых чисел, сдвиги, поразрядные операции и некоторые другие микрооперации, связанные с обработкой частей слов. Через эти микрооперации можно выразить все операции языка команд ПНМ. Например, вычитание сводится к сложению дополнительных или обратных кодов, умножение и деление – к сложениям и сдвигам, операции над числами с плавающей запятой сводятся к арифметическим операциям над частями слов – мантиссой и порядком.

Например, алгоритм умножения двух целых чисел в двоичной системе можно представить следующим образом:

```
НАЧАЛО
  r3 := 0, u3 := u1 ∧ u2, υ := 0;
  ЦИКЛ
    ЕСЛИ r2(n) = 1 ТО υ * r3 := r3 + r1 КЕ;
    ЕСЛИ υ = 1 ТО ВЫЙТИ КЕ;
      СДВИНУТЬ r2 ВПРАВО;
    ЕСЛИ r2 = 0 ТО ВЫЙТИ КЕ;
    ЕСЛИ r1(1) = 1 ТО υ := 1; ВЫЙТИ КЕ;
      СДВИНУТЬ r1 ВЛЕВО
```

КЦ
КОНЕЦ.

В этом алгоритме r_1, r_2, r_3 – целые числа, представленные в прямом коде, разрядности n; u_1, u_2, u_3 – их знаки; v – признак переполнения. Разряды нумеруются слева направо: младший имеет номер n, старший – 1.

После того как выбран набор микроопераций и определены алгоритмы выполнения операций, следует решить, на каких регистрах будут выполняться соответствующие операции. Принципиально возможен такой вариант, который предусматривает выполнение на любом регистре любых микроопераций. В этом случае получается выигрыш по времени, поскольку отсутствуют дополнительные пересылки между регистрами, однако стоимость аппаратуры получается наибольшая. Снова приходится искать компромисс. В классическом варианте все микрооперации реализуются на минимальном числе регистров, а дополнительные регистры используются лишь для промежуточного хранения величин или для развязывания информационных зависимостей внутри операторов присваивания. После выбора регистров, на которых выполняются микрооперации, алгоритм работы проектируемого модуля преобразуется в соответствии с принятыми решениями.

Например, если операция ω_i выполняется за один такт на регистре r, то оператор $R(p) := \omega_i(R(p), z)$ в алгоритме ПНМ следует преобразовать в последовательность операторов:

$r := \omega_i(R(p), z);$

$R(p) := r;$

а для уменьшения связей регистра r с другими регистрами, может быть, целесообразно использовать еще один промежуточный регистр s:

$s := R(p);$

$r := \omega_i(s, z);$

$R(p) := r.$

Таким образом, три основные задачи логического проектирования центральной части ПНМ– выбор алгоритмов выполнения операций, выбор системы микроопераций и выбор распределения аппаратных средств между регистрами, включая связи между ними, – могут быть решены путем преобразования процедурного алгоритмического описания соответствующего алгоритмического модуля. При этом могут использоваться общие методы преобразования и оптимизации алгоритмов в процессе проектирования, которые рассматриваются в следующей главе.

Упражнения

1. Рассмотреть ОЗУ, алгоритм которого работает в предположении, что АДР есть адрес слова (т.е. делится на 4) при k = 4. Как изменятся операторы обращения к памяти в алгоритме ПНМ?

2. Сформулировать условия и доказать корректность алгоритма умножения, приведенного в тексте.

3. Выбрать минимальный набор операций (скажем, сложение, вычитание, умножение, деление и сравнение чисел с плавающей и фиксированной запятой, поразрядные

конъюнкция, дизъюнкция, сложение по модулю 2 и инвертирование, сдвиги вправо и влево на заданное число разрядов) и провести логическое проектирование ПНМ на уровне алгоритмического описания по методике, рассмотренной в данном параграфе.

§ 3. Развитие неймановской концепции

Новые идеи в архитектуре и структуре вычислительных машин развивались на протяжении многих лет под влиянием трех основных проблем, вытекающих из практики их использования:

– проблемы надежности;

– повышения производительности;

– удобства и эффективности использования ЭВМ в прикладных областях.

Исследования, которые проводятся в поисках путей решения этих проблем, стимулируются постоянной неудовлетворенностью текущим положением дел, поскольку всякое продвижение открывает новые возможности, расширяет сферы применения средств вычислительной техники и в конце концов приводит к новым, повышенным требованиям в части надежности, производительности и эффективности применений. Продвижения в решении основных проблем поддерживаются также быстрым ростом технологии, позволившим за 40 лет на несколько порядков улучшить основные технические и экономические характеристики средств вычислительной техники – такие, как стоимость, быстродействие, габариты и т.п. Эти продвижения поддерживаются накоплением опыта и знаний, а также опережающим развитием программного обеспечения, позволяющим заглянуть также и в будущее технических решений.

Следует отметить, что наряду с новыми решениями в архитектуре современных компьютеров элементы неймановской структуры продолжают играть важную роль, и современный микрокомпьютер по своей структуре ближе к исходной концепции фон Неймана, чем некоторые установки второго и третьего поколений. Хотя, конечно, принципиальное отличие современного неймановского компьютера, выполненного на нескольких кристаллах кремния, от классического варианта состоит в том, что он может служить не только центральной частью автономной вычислительной установки, но и элементом сложной системы, содержащей сотни и тысячи таких компьютеров.

Рассмотрим коротко основные идеи, используемые в структурах современных ЭВМ, с точки зрения указанных трех основных проблем.

Проблема технической надежности состоит в своевременном обнаружении сбоев и неисправностей в работе аппаратуры, устранении их влияния на состояние вычислений и в восстановлении работоспособности с минимальными затратами времени и средств. В этом плане рассмотренный проект ПНМ не является удовлетворительным, поскольку не предусматривает никаких средств, обеспечивающих реакцию на аварийные ситуации.

Помимо аварийных ситуаций, вызванных неправильной работой аппаратуры, возможны также аварийные ситуации, вызванные ошибками в программах и данных. Некоторые из этих ошибок проявляются и могут быть обнаружены при выполнении команд ПНМ. К ним, в частности, относятся ошибки, приводящие к переполнению, делению на нуль, неправильные команды, неправильные адреса при обращении к ОЗУ и ВЗУ, обращение

к пустым устройствам ввода и неподготовленным устройствам вывода (это уже могут быть ошибки в обслуживании машины).

Другие ошибки принципиально не могут быть обнаружены аппаратно в силу слишком слабого понятия корректности программы и ее выполнения, заложенного в алгоритме функционирования машины. Более того, в духе концепции фон Неймана определять алгоритм функционирования машины так, чтобы любая программа и любой процесс ее выполнения были допустимы. Так, при определении адреса можно отбрасывать лишние разряды, признак переполнения может проверяться самой программой, неправильную команду можно просто пропускать. Ситуация обостряется также тем, что ошибки в программах не только могут быть изначально, но появляются из-за сбоев в аппаратуре при хранении и передачах информации, а также при копировании программ, составляющих программное обеспечение вычислительной системы.

Одним из наиболее распространенных средств аппаратного контроля является использование избыточного кодирования информации, позволяющего проверять правильность передачи информации при обменах между центральной частью ЭВМ, запоминающими и периферийными устройствами. Обычно эта избыточность реализуется добавлением к двоичному коду, составляющему единицу обмена, контрольных разрядов, которые проверяются при обменах. Реакция на аварийные ситуации, обнаруживаемые схемами контроля, определяется алгоритмом функционирования центральной части. Другие средства программно-аппаратного обнаружения аварийных ситуаций тесно связаны с общей проблемой эффективного использования ЭВМ и реализуются в комплексе со средствами, обеспечивающими решение этой проблемы. Такие общие методы повышения надежности, как дублирование аппаратуры на различных уровнях, применяются лишь в случаях использования машин в особо ответственных системах, где требуется высокая степень надежности и готовности.

Основным свойством машины Неймана является ее универсальность. Действительно, достаточно одной только операции прибавления единицы, сравнения целого с нулем и условного перехода для того, чтобы вычислять любую частично рекурсивную функцию, правда, в предположении неограниченной разрядности слов. От этого ограничения можно избавиться, промоделировав арифметические операции над числами произвольной длины в оперативной памяти. Тогда начнет сказываться ограниченность ОЗУ. Но и это ограничение можно снять, используя внешнюю память со сменными носителями (диски, ленты). Осознание универсальности машины фон Неймана сыграло исторически важную роль, поскольку стимулировало поиски машинных алгоритмов решения задач невычислительного характера – комбинаторной оптимизации, управления сложными дискретными системами, поиска доказательств теорем, обработки текстов, файлов и т.п. Однако наиболее широкое практическое использование принципа универсальности обуславливалось тем, что с помощью машины фон Неймана оказалось возможным моделирование других структур, более удобных для использования и обладающих значительно более широкими возможностями. Машина фон Неймана, оснащенная соответствующим программным обеспечением, воспринимает программы, написанные как на процедурных, так и на непроцедурных языках высокого уровня (функциональных или

логических) и может их реализовать путем трансляции или интерпретации. С помощью языка управления заданиями машина может воспринимать и осуществлять сложные действия, в которых выполнение отдельной программы является элементарным актом. Программное обеспечение позволяет поддерживать не только автоматический режим выполнения программ, но и взаимодействие машины с человеком в процессе решения задач (интерактивный режим работы).

В результате развития программного обеспечения ЭВМ постепенно формируется представление о том, что собой представляет идеальная машина и как с ней нужно работать. Важную роль в этом представлении играет различение основных путей использования ЭВМ:

1. Использование в качестве элемента производственной системы, функционирующей по законам определенной технологии, например системы управления реального времени (технологические процессы, движущиеся системы и т.п.), системы организационного управления, системы обслуживания запросов, вычислительного центра коллективного пользования и т.п.

2. Выполнение научно-исследовательских и проектно-конструкторских разработок, гибкие автоматизированные производства, учебный процесс и т.п.

3. Разработка, отладка и сопровождение новых программ и программных систем, модификация и перенесение на новую техническую базу разработанных ранее систем.

Каждый из указанных путей использования ЭВМ предъявляет свои требования к внешним характеристикам систем, способам представления данных, алгоритмам и средствам взаимодействия с внешней средой. Для приложений первого типа характерным является высокий уровень специализации, работа по заранее определенным алгоритмам, жесткие временные характеристики, повышенные требования к надежности. В приложениях второго типа на первый план выдвигается универсальность на уровне крупных программных модулей, выполняющих определенные технологические операции по обработке данных, характерные для рассматриваемой предметной области, интерактивный режим работы, проблемная и профессиональная ориентация языков взаимодействия с ЭВМ. Разработка новых систем требует: использования языков программирования различного уровня, приспособленных к используемым структурам технических средств; доступа к информации о состоянии вычислительных процессов на различных уровнях; инструментальных систем, помогающих выполнять технологические операции процессов разработки и отладки программ с высокой степенью эффективности.

Несмотря на алгоритмическую универсальность, машина Неймана не может удовлетворить требованиям всех путей использования ЭВМ, поскольку необходимое программное обеспечение оказывается очень сложным и не может обеспечить необходимой производительности в процессе своего функционирования. Абсолютное повышение скорости работы отдельных компонент, хотя и повышает производительность системы в целом, может не дать желаемого эффекта, поскольку в конечном счете производительность может определяться самым медленным устройством.

Поскольку самыми медленными устройствами ЭВМ являются устройства ввода-вывода и внешние запоминающие устройства, то первым ради-

кальным усовершенствованием архитектуры машины Неймана стало совмещение работы центрального процессора с работой внешних устройств. При анализе функционирования ПНМ видно, что команды ввода-вывода связаны с ожиданиями. При вводе с перфокарт процессор ожидает момента поступления очередного байта, при обращении к магнитному диску — момента установки головки на требуемую дорожку, момента начала поступления очередной записи и т.д. В периоды ожидания процессор простаивает. Поскольку скорости работы внешних устройств на несколько порядков отличаются от скорости работы центральной части, резерв повышения быстродействия оказывается значительным.

Архитектура с использованием совмещения работы внешних устройств с работой процессора изображена на рис. 5.7. Устройство управления (УУ) работает над общей памятью с устройством обмена (УО). Получив команду ввода-вывода, УУ передает ее устройству обмена, а само переходит к выполнению следующей команды. Пока устройство обмена выполняет команду ввода-вывода, УУ может продвигаться дальше в выполнении программы. Это продвижение, однако, может выполняться лишь до тех пор, пока не понадобятся результаты выполнения команды ввода или не потребуется изменить память, используемую при выполнении команды вывода. Если к этому моменту команда ввода-вывода не завершится, возникает ожидание.

Таким образом, для того чтобы эффективно использовать возможность совмещения, необходимо учитывать эту возможность в самой программе и помещать команды ввода-вывода в ней так, чтобы эти команды выполнять заранее — до того, как понадобятся результаты их выполнения. При этом следует иметь в виду, что не для любой программы это возможно. В частности, если время решения задачи внутри процессора меньше, чем время ввода-вывода, то совмещение не даст никакого эффекта.

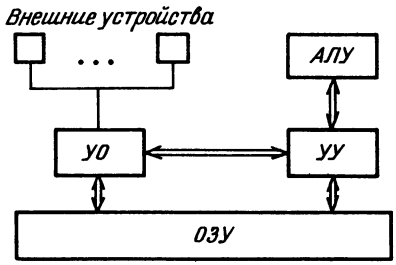


Рис. 5.7

Существует, однако, другой подход, при котором от указанных ограничений в значительной степени можно освободиться. Это — режим многопрограммной работы (мультипрограммирование). Идея многопрограммной работы состоит в следующем. В машину первоначально вводится не одна программа, а пакет из нескольких программ. Затем эти программы выполняются одна за другой. Каждая программа выполняется до первой команды ввода-вывода. Как только процессор дойдет до такой команды, она передается устройству обмена, программа прерывается и, пока выполняется команда ввода-вывода, процессор начинает выполнять новую программу или одну из программ, прерванных ранее. При удачном составлении

пакета — в частности, таком, в котором общее время работы процессора значительно больше, чем время работы устройства ввода-вывода, — возможно полное устранение периодов ожидания. Это означает, что вычислительная машина работает со скоростью ее наиболее быстродействующей центральной части.

При многопрограммной работе возникает ряд дополнительных проблем. Важнейшими из этих проблем являются проблема управления разделением ресурсов и проблема защиты. Необходимость разделения ресурсов возникает из-за того, что разные программы могут пользоваться одними и теми же ресурсами: устройствами ввода-вывода, наборами данных, областями в ОЗУ. Поэтому необходимо организовать очереди и обеспечивать их прохождение. Проблема защиты возникает из-за того, что в некоторых программах одного пакета могут быть ошибки. Программа, содержащая ошибку, может воздействовать на информацию, обрабатываемую другой программой, что приведет к получению неправильных результатов с помощью уже отлаженной программы.

Таким образом, алгоритм управления многопрограммной работой получается достаточно сложным. Поэтому он, как правило, реализуется программами операционной системы и лишь некоторые наиболее ответственные его части реализуются на уровне аппаратуры устройства управления.

Особенно выгоден режим многопрограммной работы в сочетании с интерактивной обработкой данных с использованием экранных пультов (дисплеев) при разделении времени процессора. Разделение времени процессора между несколькими пользователями состоит в том, что каждой из

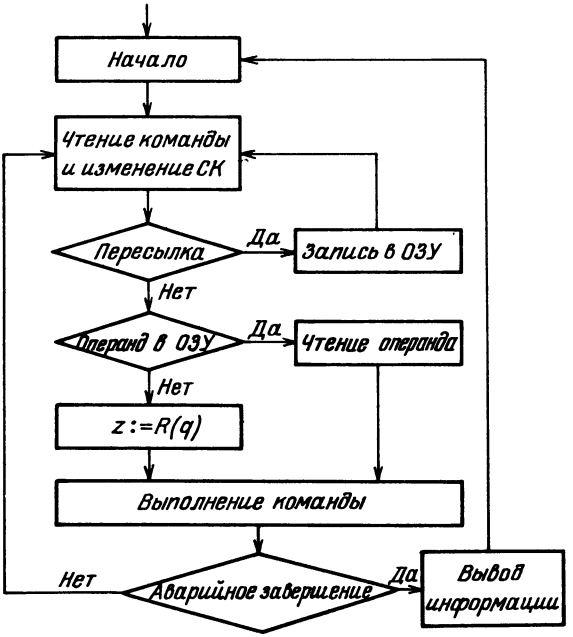


Рис. 5.8

программ пользователей выделяется определенный квант времени (обычно доли секунды), процессор периодически опрашивает все программы и выполняет в течение выделенного кванта времени те из них, которые не находятся в состоянии ожидания. При этом для каждого пользователя создается иллюзия того, что он работает сам, владея полностью всеми ресурсами вычислительной системы. В больших машинах третьего поколения устройство обмена может быть реализовано в виде специализированного программно-управляемого процессора (канальный процессор), который позволяет обслуживать одновременно несколько групп устройств, работающих параллельно.

Дальнейшие возможности ускорения работы вычислительной машины можно обнаружить, анализируя алгоритм функционирования ее центральной части. Представим для примера алгоритм ПНМ в виде схемы рис. 5.8. Анализируя эту схему, видим, что основной рабочий цикл алгоритма при отсутствии аварийных ситуаций складывается из двух частей – обращения к памяти и выполнения команды. При этом, если текущая команда не является командой условного перехода, то адрес следующей уже определен, и обращение к ОЗУ для чтения очередной команды можно начинать прежде, чем закончится выполнение текущей команды. Для этого необходимо выделить из устройства управления модуль УВК (устройство выполнения команд), который реализует выполнение команды. Для того чтобы модуль УВК мог работать параллельно с УУ, необходимо продублировать регистр z, а также части КОП и p регистра РК.

Теперь получим структуру центральной части, изображенную на рис. 5.9. Здесь R – регистровая память АЛУ, S – регистр, через который УУ передает информацию в УВК (z, КОП, p) и получает информацию об окончании выполнения команды. Устройства УУ и УВК работает с перекрытием. Если время обращения к памяти меньше, чем время выполнения команды, то выборка и запись происходят полностью на фоне работы устройства УВК и фактически не требуют времени. Быстродействие машины полностью

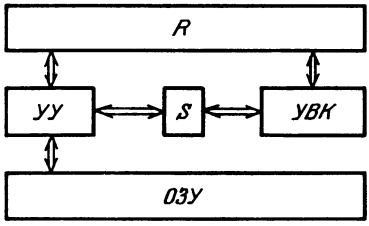


Рис. 5.9

определяется скоростью выполнения операций. Единственный случай, когда устройства УУ и УВК работают последовательно, – это условный переход. Действительно, в этом случае выборка следующей команды не может начаться до тер пор, пока не будет выполнена предыдущая команда, которая должна выработать значение регистра ПР. Если УВК работает быстрее, чем УУ, быстродействие определяется временем обращения к ОЗУ.

Прямые методы повышения быстродействия ОЗУ связаны с ростом стоимости и сокращением объема. Логически можно получить выигрыш,

если разделить ОЗУ на несколько отдельных модулей, которые могут работать параллельно. В течение выполнения одной команды УУ может обращаться к ОЗУ до двух раз (выборка команды и выборка операнда или пересылка результата). Если команда и операнд расположены в разных модулях, то соответствующие обращения к памяти могут выполняться параллельно. Если времена выполнения команд различны и есть команды, которые по времени выполнения превосходят время обращения к памяти,

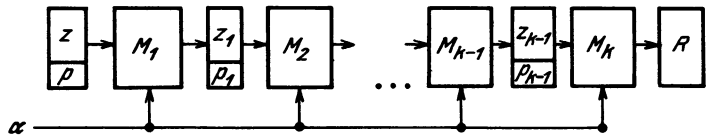


Рис. 5.10

то некоторый выигрыш можно получить, заменив регистр S двумя очередями (одна от УУ к УВК, другая от УВК к УУ). Выигрыш происходит за счет того, что во время выполнения длинных операций происходит подготовка команд и операндов, соответствующих коротким операциям. Реальный выигрыш зависит от объема буфера (очереди), соотношения времен выполнения различных команд и их взаимного расположения в программе.

Ускорение работы УВК имеет смысл лишь при условии, что время обращения к ОЗУ меньше времени выполнения команд (по крайней мере некоторых). Основным методом ускорения выполнения команд является конвейеризация.

Простейший вариант конвейерного выполнения команды состоит в следующем. Пусть команда выполняется по алгоритму, имеющему следующий вид:

$$P = (z := f_1(z); \ldots ; z := f_k(z); R(p) := z),$$

где z – набор переменных (регистров), участвующих в выполнении команды, p – адрес регистровой или оперативной памяти R, куда должен быть записан результат. Представим алгоритм P в виде

$$P = (z_1 := f_1(z), p_1 := p); (z_2 := f_2(z_1), p_2 := p_1); \ldots ; R(p_{k-1}) := f_k(z_{k-1}).$$

Теперь этот алгоритм можно реализовать с помощью сети из алгоритмических модулей, изображенной на рис. 5.10. Здесь α – управляющая входная переменная без памяти; $z, z_1, \ldots, z_{k-1}, p, p_1, \ldots, p_{k-1}$ – входные (выходные) переменные с памятью; R – общая память.

Алгоритм модуля M_i для $i = 1, \ldots, k$ имеет вид $(z_0 = z, p_0 = p)$

ЦИКЛ
  ЖДАТЬ α:
  $z_i := f_i(z_{i-1})$
КЦ.

Модуль M_k:
ЦИКЛ
  ЖДАТЬ α:
  $R(p_{i-1}) := f_k(z_{k-1})$
КЦ.

Устройство подобного типа называется k-ступенчатым (линейным) конвейером. Переменная α – тактирующая переменная. Начав выполнение первой команды, устройство уже готово к выполнению следующей. Одновременно в устройстве могут выполняться k команд и, если на его вход данные подаются с частотой изменения тактирующей переменной α, на выходе будут с такой же частотой вырабатываться результаты. Замедление работы конвейера может получиться, если операндом одной из следующих команд является результат выполнения предыдущей. Тогда следующая команда не сможет начать выполняться, пока не закончится предыдущая.

Конвейеризацию обычно применяют в сочетании с использованием нескольких устройств выполнения команд, которые могут работать параллельно. Как правило, это специализированные устройства, например устройство для арифметических операций с фиксированной или плавающей запятой, устройство для логических операций и т.п. Применение методов ускорения работы процессора дает хороший эффект лишь при сбалансированных временных характеристиках функционирования ОЗУ и УВК. Для того чтобы уменьшить влияние дисбаланса, возникающего в результате разброса характеристик отдельных команд и их распределения в потоке команд, определяемом выполняемой программой, применяют промежуточную память между ОЗУ и регистровой памятью. В ней хранятся команды и данные, которые были недавно в работе. Такая память называется иногда КЭШ-памятью. Она обладает бóльшим быстродействием, чем ОЗУ, но обычно меньшим объемом. После чтения очередной команды из ОЗУ она помещается в КЭШ и остается там, пока не будет вытеснена другими командами или данными. В каждой момент времени определено взаимно однозначное соответствие между адресами КЭШ-памяти и адресами ОЗУ. Если процессору требуется содержимое ячейки с заданным адресом, то нужно быстро определить, находится ли это содержимое в ОЗУ или в КЭШ-памяти, после чего, если надо, происходит обмен ОЗУ – КЭШ. Примером эффективного использования КЭШ-памяти может служить многократное выполнение коротких циклов. При первом выполнении команды такого цикла переносятся одна за другой в КЭШ. После этого до окончания работы цикла обращения к ОЗУ за командами вообще не будет.

Другое направление в развитии архитектуры ЭВМ связано с развитием машинного языка. Основным недостатком языка машины Неймана является его низкий уровень и необходимость разбивать действия на большое число элементарных шагов. Причем значительная часть этих шагов связана с получением промежуточных результатов, их размещением в памяти и перемещением между ОЗУ и регистрами. При этом процессор в каждый момент времени видит лишь одну команду и не имеет никакой информации о том, в каком контексте эта команда выполняется, в частности в каких ячейках памяти расположены команды, в каких – данные, поскольку между ними нет никакого различия.

Повышение уровня языка происходит несколькими путями. Первый путь состоит в укрупнении операций, выполняемых командами машины. Наиболее ясным путем такого укрупнения является включение в систему команд векторных и матричных операций. При выполнении таких команд особенно эффективно работают конвейерные устройства, поскольку процессор в этом случае обращается в ОЗУ только за данными, а при наличии

большой КЭШ-памяти и повторяющихся операций с длинными векторами или матрицами можно при разумном планировании действий достаточно плотно загрузить конвейер, разгоняя его до максимально возможных скоростей. Другой путь развития языка состоит в приближении его к алгоритмическим языкам высокого уровня. Первым шагом на этом пути является включение в язык средств представления арифметических выражений. Обычно при этом используется правая бесскобочная запись, которая облегчает реализацию вычисления значения такого выражения за один просмотр слева направо с запоминанием промежуточных значений в магазинной памяти. Эффективность повышается за счет сокращения длины программы.

Вместо арифметических выражений иногда в обычную систему команд включают команды для работы с магазином (запись, чтение, выполнение опраций над последними записанными значениями). Использование магазина имеет ряд преимуществ по сравнению с общими регистрами и широко применяется в системах команд микропроцессорных ЭВМ.

Следующий шаг приближения к языкам программирования – включение типов данных и их машинное представление. В этом случае процессор может различать различные виды информации, расположенной в памяти, интерпретировать операции в соответствии с типами данных и структур данных, а также обнаруживать различного рода аварийные ситуации, вызванные ошибками в программе или сбоями оборудования.

Таким образом, повышение уровня машинного языка позволяет существенно продвинуться в решении проблемы надежности. Ярким примером является проверка границ индексов при обращении к элементам массивов. В обычной системе команд эту проверку можно реализовать только за счет значительного увеличения времени работы.

При встроенных описаниях типов массивов процессор имеет доступ к нужной информации и может осуществлять соответствующие проверки аппаратно без существенных потерь времени. Например, наиболее радикальным решением является прямая интерпретация языка высокого уровня. Это направление тесно связано с развитием внутреннего интеллекта ЭВМ. Действительно, в языке высокого уровня короткие программы выражают сложные действия, выполняя которые, процессор может целесообразно планировать использование оборудования. Сокращение объемов программ значительно расширяет возможности разработки сложных прикладных систем, а отсутствие барьера между программистом и ЭВМ в виде системы программирования облегчает отладку программ и значительно сокращает сроки разработки программных систем.

Основным ограничением архитектуры машины Неймана является последовательный обмен командами и данными между процессором и основной памятью. Это ограничение часто называют "узким горлышком" машины Неймана. Каким бы образом ни ускорять выполнение операций в АЛУ, машина не может работать быстрее, чем происходит обмен УУ и ОЗУ. Поэтому естественным шагом в развитии архитектуры ЭВМ является переход к многопроцессорным системам.

Существующие в настоящее время многопроцессорные системы можно классифицировать в зависимости от того, как решаются три основные

проблемы, связанные с их организацией. Этими проблемами являются управление, память и связь.

Проблема управления имеет два основных решения – централизованное и распределенное. При централизованном управлении многопроцессорная система представляет собой управляемую синхронизированную сеть из алгоритмических модулей, в которой управляющий модуль производит установку начального состояния всех процессоров в начале каждого сегмента синхронизации. Частным случаем централизованных систем являются такие, в которых все процессоры на одном этапе выполняют одну и ту же команду, но над разными данными. Такие системы называются системами типа ОКМД (одиночный поток команд, множественный поток данных) или системами типа SIMD (single instruction, multiple data) по классификации Флинна. Распределенное управление означает асинхронное функционирование процессоров, образующих одноуровневую сеть из алгоритмических модулей. Такие системы соответствуют системам типа МКМД (множественный поток команд, множественные данные), или системам типа MIMD (multiple instruction, multiple data) по классификации Флинна.

Память может быть общей или распределенной. Речь идет, разумеется, об основной памяти, в которой находятся программы и данные во время их исполнения. При достаточно большом числе процессоров общая память обычно разбивается на модули так, чтобы разные процессоры могли обращаться к разным модулям памяти одновременно. Применение распределенной памяти означает, что каждый процессор имеет свою собственную память, объем которой достаточно велик для самостоятельного выполнения заданий.

Связь между процессорами представляет собой наиболее сложную проблему и имеет много решений. Удобно различать универсальные и специальные системы связи. Универсальная система связи позволяет осуществлять любые попарные соединения между процессорами (каждый с каждым). Универсальная связь технически может быть реализована либо с помощью общей шины, либо с помощью коммутационной системы. В первом случае в каждый момент времени обмениваться могут только два процессора, во втором допустим одновременный обмен между многими процессорами. В специальных системах связи каждый процессор может непосредственно взаимодействовать лишь с ограниченным множеством других процессоров. Процессоры в таких системах образуют сеть, которая может иметь самые разнообразные топологические формы. Например, это может быть двумерная или трехмерная решетка, в которой допустимы соединения процессоров, расположенных в соседних точках. Применяются деревообразные сети, двоичные кубы, торы и т.п. В специальных системах связи часто реализуется возможность одновременной передачи от одного процессора ко всем или к нескольким другим процессорам.

Границы между рассматриваемыми архитектурами не являются достаточно четкими. Например, в системе с общей памятью каждый процессор может иметь КЭШ-память достаточно большого объема, и в некоторых случаях она может использоваться как основная память, что приводит к архитектуре систем с распределенной памятью. В системах со специальной системой связи могут быть реализованы аппаратные средства для осуществления транзитных обменов, и в этом случае связь становится универсаль-

ной. Возможны также смешанные варианты архитектуры многопроцессорных систем. Все они достаточно точно могут быть описаны в терминах многоуровневых сетей из алгоритмических модулей с постоянной структурой.

Применение многопроцессорных систем является в настоящее время основным резервом повышения производительности ЭВМ, поскольку возможности одного процессора близки к исчерпанию. В системах с распределенным управлением естественным образом реализуется режим многопрограммной работы, который дает возможность совмещать не только работу внешних устройств с одним процессором, но и нескольких процессоров, выполняющих различные программы, что открывает более широкие возможности. Системы с централизованным управлением требуют специальных языковых средств программирования и наиболее эффективно используются в качестве специализированных процессоров, предназначенных для выполнения, например, матричных или векторных операций. Системы с распределенным управлением дают возможность реализовать универсальные языки параллельного программирования высокого уровня.

В настоящее время широко обсуждаются возможности аппаратной реализации непроцедурных языков программирования – логических и функциональных. Поскольку в таких языках имеет место широкий параллелизм, они должны хорошо реализоваться на многопроцессорных ЭВМ. Проблема, однако, здесь состоит не столько в разработке подходящей архитектуры, сколько в отыскании хороших алгоритмов интерпретации, позволяющих получить высокую эффективность на широком классе программ.

Комментарии к главе 5

Развитие структур и практика создания ЭВМ довольно длительный период базировалась на концепции автоматического исполнения команды либо последовательности команд. ЭВМ этого вида получили в последствии название неймановских [13, 14, 15]. ЭВМ неймановской структуры могли развиваться лишь при наличии достаточно развитых средств автоматизации проектирования. При этом большое значение имели математические модели ЭВМ и формальные методы, отражающие реальные процессы проектирования. Одна из первых методик проектирования ЭВМ неймановской структуры базировалась на теории цифровых автоматов [15] и легла в основу известных экспериментальных систем ПРОЕКТ [19]. Развитие неймановских структур ЭВМ поддерживалось теорией проектирования ЭВМ. Следует заметить, что и в 60–70-е годы – период активного использования неймановской концепции – осуществлялись проекты ЭВМ структуры, отличной от неймановской (например, машины ГАММА-60, ИЛИАК-4, машины серии МИР, Б-5000 и др.), представление о которых можно составить по [6, 32]. В работе [18] были детально проанализированы достоинства и недостатки концепции неймановских ЭВМ и предложена новая концепция ЭВМ, отличная от неймановской.

Глава 6

ПРОЕКТИРОВАНИЕ ПОСЛЕДОВАТЕЛЬНЫХ ПРОГРАММ

§ 1. Основные этапы проектирования программ

Процесс проектирования программы представляет собой построение последовательности моделей этой программы и ее частей, каждая из которых уточняет или дополняет модели, полученные ранее. Построенные модели используются для получения окончательного результата — текста объектной программы во входном языке вычислительной системы, используемой для ее выполнения. Программа должна вычислять некоторую функцию $f: A \to B$, определенную в терминах математической модели соответствующей предметной области, обладать определенными сложностными характеристиками и функционировать в заданной програмной среде.

Каждый шаг проектирования, т.е. построение новой модели, связан с принятием некоторых решений относительно структур данных, алгоритмов выполнения тех или иных операций (операторов), взаимодействия частей проектируемой программы и т.п. В тех случаях, когда принятые решения не обеспечивают выполнение требований к результату, приходится возвращаться к начальным этапам проектирования программы и искать новое программное решение. Описание функции f часто называют спецификацией программы. Для того чтобы избежать путаницы с более специальными случаями употребления этого термина, будем называть это описание главной функциональной моделью проектируемой программы. Процесс проектирования может быть вырожденным в том случае, когда программа составляется сразу во входном языке системы по своей функциональной модели. Более того, уже построенная программа иногда может сама служить определением той функции, которую она вычисляет. Такой подход можно назвать прямым программированием. Он может быть использован при разработке простых программ, при высоком уровне входного языка и при слабых требованиях к характеристике проектируемой программы. В других случаях процесс проектирования требует более сложной организации в виде последовательности этапов, обеспечивающих движение сверху вниз и использующих принцип сведения задачи к подзадачам — основной метод преодоления сложности при проектировании больших систем.

Можно выделить следующие основные этапы проектирования программ:

1. Построение главной функциональной модели.
2. Конструктивизация функциональных моделей.
3. Преобразование функциональных моделей.
4. Синтез процедурных представлений.
5. Преобразование процедурных представлений.

Главная функциональная модель строится средствами математической модели предметной области (§ 5 гл. 4) как функция теоретико-множественных структур данных. Процесс построения главной функциональной модели может включать в себя построение модели предметной области, если последняя недостаточно формализована. В случае областей, использующих аппарат классической прикладной математики, таких, как физика или механика, основные объекты предметной области представляются числовыми функциональными пространствами различных типов, а функ-

циональные модели представляются системами уравнений в этих пространствах (дифференциальными, интегральными, алгебраическими). Но и здесь может потребоваться создание новых моделей, скажем при работе с объектами, имеющими сложную геометрию, различного рода комбинациями дискретных и непрерывных систем и т.п. Во многих случаях задача, которую требуется решать, сводится к поиску процессов в пространстве состояний некоторой дискретной системы. Так, например, обстоит дело в области искусственного интеллекта. Хорошо известно, что правильный выбор структуры пространства состояний, в частности определение функции переходов, играет решающую роль при поиске эффективных алгоритмов решения задач. В целом первый этап проектирования не формализован и существенным образом зависит от предметной области.

Главная модель может быть построена с использованием вспомогательных моделей, которые будут реализованы с помощью подпрограмм объектной программы. Возможны различные способы представления функциональных моделей программ.

Логическое представление описывает свойства функции $y = f(x)$ средствами базового логического языка в виде предиката $P(x, y)$ такого, что $P(x, f(x))$ истинно для любых $x \in A$. Такое описание может быть неоднозначным, а выбор для каждого $x \in A$ единственного значения y такого, что $P(x, y)$, предполагается осуществить на последующих этапах проектирования. В случае, когда требуется найти все значения y такие, что $P(x, y)$, задача сводится к вычислению однозначной функции, принимающей значения в множестве подмножеств. Если логическое представление предполагает однозначное описание функции $f: A \to B$, то должна быть доказана теорема о том, что для всякого $x \in A$ существует единственное $y \in B$ такое, что $P(x, y)$. Если функция f может быть частично определенной, то достаточно доказать, что для каждого $x \in A$ существует не более одного $y \in B$ такого, что $P(x, y)$.

Функция f может быть описана также с помощью предиката $P(f, x) = Q(t_1, ..., t_n)$, в котором термы $t_1, ..., t_n$ зависят от x и могут использовать f в качестве функционального символа. Разумеется, не исключаются случаи, когда $x = (x_1, ..., x_m)$, $f = (f_1, ..., f_k)$, т.е. рассматривается спецификация системы функций многих переменных. Частным случаем такого описания является система функциональных уравнений $F_i(f, x) = G_i(f, x)$ $(i = 1, ..., m)$ в алгебре структур данных. Такую систему можно рассматривать как алгебраическое представление функции f.

Функциональная модель может быть задана также с помощью настроенной дискретной системы S, а функция f определена как функция f_S, вычисляемая этой системой, либо выражена через f_S и другие вспомогательные функции. Система S может быть недетерминированной, а функция f_S — многозначной. В этом случае f может быть определена как любая из функций, содержащихся в f_S. Такое представление так же, как и в случае логического представления, неоднозначно.

Задача конструктивизации функциональной модели состоит в нахождении какого-либо алгоритма вычисления функции f. Обычно она решается путем построения новой модели, которая реализует заданную и для которой известен алгоритм вычисления соответствующей функции. В случае логического представления формулой $P(x, y)$ существование алгоритма оз-

начает перечислимость данного предиката. Если f: $A \to B$, а области A и B наделены отношением аппроксимации, то, вообще говоря, не требуется, чтобы алгоритм давал значение $f(x)$, если оно не определено ($f(x) = w$). При нетривиальном отношении аппроксимации можно ограничиться и более слабыми требованиями. Например, вместо f вычислять другую функцию f': $A_0 \to B_0$, где A_0 и B_0 – всюду плотные конструктивные подмножества множеств A и B соответственно, т.е. всякий элемент $x \in A$ есть предел $\bigcup_{i=1}^{\infty} x_i$ возрастающей цепи элементов множества A_0 (аналогично для B).

Функция f' должна удовлетворять условию

$$x = \bigcup_{i=1}^{\infty} x_i \Rightarrow \bigcup_{i=1}^{\infty} f'(x_i) = f(x).$$

Задача конструктивизации решается обычно в предположении конструктивности операций и отношений базовой алгебры данных. Из этого предположения следует, что всякая бескванторная формула $P(x, y)$ задает перечислимый предикат. Формулы с кванторами, ограниченными конечными множествами конструктивно заданных элементов алгебры данных, т.е. кванторами вида $\forall x \in A\ P(x, y)$ и $\exists x \in A\ P(x, y)$, где A – конечное множество, также задают перечислимые предикаты. Конструктивными будут также рекурсивные определения функций над конструктивными алгебрами с аппроксимацией или функции, заданные с помощью конструктивных дискретных систем. Конструктивность операций над структурами данных требует специальных исследований в каждом конкретном случае.

Коструктивизация модели, заданной системой уравнений, может состоять в преобразовании ее к такому виду, для которого существует общий алгоритм решения, например определение отношения аппроксимации и приведение к канонической форме. Конструктивизация главной функциональной модели означает существование алгоритма ее вычисления. Например, если эта модель задана бескванторной формулой $P(x, y)$, то такой алгоритм может быть определен циклом:

Для всех $z \in B$ выполнить

Если $P(x, z)$ то $y := z$, выйти КЕ

КЦ

Реализация цикла требует использования алгоритма перечисления всех элементов множества B. Существование такого алгоритма должно следовать из конструктивности этой области. Прямой алгоритм, реализующий главную модель, обычно неэффективен, и переход к другому представлению функциональной модели может дать более эффективную реализацию.

Предварительные оценки эффективности для функциональных моделей могут выполняться путем подсчета числа операций базовой алгебры, с использованием предварительных оценок времени и памяти, которые требуются для их выполнения, и оценок пространства поиска в случае переборных алгоритмов. Построение удовлетворительных функциональных моделей выполняется путем их преобразований с использованием знаний о предметной области, т.е. ее математической теории. Переход к процедурной форме представления алгоритмов вычисления функций, составляющих главную функциональную модель, или синтез программы, осуществляется обычно

с помощью некоторого общего метода синтеза, рассчитанного на класс конструктивных функциональных моделей. Полученная программа подвергается затем последовательности преобразований, целью которых является переход к модели вычислительной системы. При этом возможны следующие основные виды преобразований:

1. Переход к новой базовой алгебре.
2. Реализация структур данных.
3. Переход к новой информационной среде.
4. Преобразование программы без изменения базиса.

При переходе к новой базовой алгебре данных D' от алгебры D операции алгебры D должны быть реализованы операциями алгебры D'. Если эта реализация определяется строгим гомоморфизмом D' в D, то с помощью теоремы 3.1 гл. 2 для каждой схемы программы, интерпретированной на D, может быть построена ее полная гомоморфная реализация над D'. В качестве D' может быть использована базовая алгебра модели вычислительной системы, но в некоторых случаях целесообразно сделать это через промежуточную алгебру, рассматривая дополнительный уровень реализации. Реализация алгебры D может быть выполнена не только с помощью гомоморфизма, но и другими способами. Например, достаточно установить соответствие между элементами D и D' так, чтобы каждую элементарную функцию на D можно было отобразить на некоторую элементарную или алгебраическую функцию алгебры D' с однозначным восстановлением исходной функции. Такая реализация может быть сведена к гомоморфной, если вместе с основными операциями в алгебре рассматривать все элементарные и алгебраические функции. Если алгоритм работает со структурами данных, то переход к новой базовой алгебре сохраняет способы образования структур.

Реализация структур данных также предполагает переход к новой алгебре данных. Только это делается не на уровне базовых алгебр, а на уровне алгебр структур данных. Еще более общий способ реализации программы предполагается при переходе к новой информационной среде. В этом случае может быть использована не только гомоморфная, но и автоматная реализация дискретных систем. Обоснование перехода выполняется уже не на уровне операций базовой алгебры данных, а на уровне операторов и последовательностей операторов.

Наконец, преобразования, сохраняющие информационную среду, — это произвольные преобразования, сохраняющие главную функциональную модель с учетом всех предыдущих этапов, определяющих последовательную композицию реализующих отображений. Такие преобразования могут иметь разную силу. Наиболее сильные преобразования не зависят от интерпретации схемы программы, более слабые используют ту или иную информацию о строении предметной области. Во всех случаях переходов к следующим этапам проектирования совокупность моделей программы должна оставаться реализацией главной функциональной модели.

Проектирование последовательных программ, которые функционируют в среде многокомпонентных систем, скажем программы функционирования компоненты сети из алгоритмических модулей, хотя и обладает определенной спецификой, во многих случаях сводится к проектированию программ, реализующих функциональные модели. Главная функциональная модель описывает многозначную функцию переходов дискретной системы,

моделирующей сеть, в которую погружена проектируемая компонента. Состояниями системы являются наборы значений внешних компонент алгоритмических модулей, составляющих сеть, а также внешней компоненты сети. При асинхронном взаимодействии задача сводится к проектированию программы, вычисляющей нужные функции при переходах между двумя соседними моментами времени, в которые происходят взаимодействия с окружающей средой.

Проектирование программ, предназначенных для выполнения на многопроцессорных ЭВМ, кроме функциональных и процедурных моделей, требует также использования структурных моделей. Более подробно эти вопросы будут рассмотрены в гл. 7.

§ 2. Вычисление элементарных функций

Предположим, что главная функциональная модель определяет функцию $f(x_1, ..., x_n) = (y_1, ..., y_m)$, которая может быть представлена в виде

$$y_i = F_i(x_1, ..., x_n), \quad i = 1, ..., m, \tag{2.1}$$

где переменные $x_1, ..., x_n$ пробегают область $A \subset D_1 \times ... \times D_n$, $D_1, ..., D_n$ – компоненты базовой многоосновной алгебры данных D, область A определяется бесквантворной формулой $\alpha_0(x_1, ..., x_n)$, записанной в сигнатуре П предикатов базовой алгебры, т.е. $(x_1, ..., x_n) \in A \Leftrightarrow \alpha_0(x_1, ..., x_n)$, а функции F_i элементарны, т.е. представлены в виде суперпозиции алгебры D базовых функций. Функциональная модель, удовлетворяющая перечисленным условиям, называется *элементарной*. Ее представление в виде (2.1) является решением задачи конструктивизации и может быть положено в основу построения U–Y-программы над памятью V со стандартным базисом, определенным сигнатурой (Ω, Π) базовой алгебры интерпретированной на этой алгебре.

Простейшим решением задачи синтеза для функциональной модели (2.1) является программа, состоящая из единственного оператора присваивания:

$$(y_1 := F_1(x), ..., y_m := F_m(x)); \tag{2.2}$$

символы $x_1, ..., x_n$ и $y_1, ..., y_n$ включаются в множество V и называются *основными* (входными и выходными) *переменными*. Более надежная программа выполняет также проверку начального условия α_0 и печатает в случае его нарушения сообщение о неправильных данных:

если $\alpha_0(x)$ **то** $y := F(x)$ **иначе вывод** ("ошибка в данных") **ке.**

Преобразования процедурного представления могут иметь целью улучшение программы по времени, памяти и другим характеристикам. Время выполнения программы (2.2) определяется количеством операций, которые использованы в выражениях $F_1(x), ..., F_m(x)$, с учетом времен выполнения каждой из этих операций. Изменение выполняемых операций и, следовательно, времени выполнения программы может быть сделано с помощью применения тождественных соотношений алгебры данных. Равенство $t(z_1, ..., z_n) = t'(z_1, ..., z_n)$ называется тождеством, если оно истинно при любых значениях переменных $z_1, ..., z_n$, взятых из соответствующих компонент алгебры данных. В выражениях $t(z)$ и $t'(z)$ могут участвовать константы, которые в этом случае выступают как 0-арные операции. Если $t = t'$

есть тождество базовой алгебры данных, то операторы $x := t$ и $x := t'$ эквивалентны и могут заменять друг друга. Другой вид экономии времени может быть сделан за счет выделения совпадающих подвыражений. Соответствующее преобразование состоит в замене оператора

$$(x_1 := F_1(G_1, ..., G_m), ..., x_n := F_n(G_1, ..., G_m))$$

последовательностью

НАЧАЛО

$y_1 := G_1, ..., y_m := G_m;$

$x_1 := F_1(y_1, ..., y_m), ..., x_n := F_n(y_1, ..., y_m)$

КОНЕЦ.

Более тонкая оптимизация может быть выполнена с помощью условных тождеств. Так называются соотношения вида

$$\alpha(z_1, ..., z_n) \Rightarrow t(z_1, ..., z_n) = t'(z_1, ..., z_n),$$

которые истинны при любых допустимых подстановках вместо переменных $z_1, ..., z_n$. Применение условных тождеств основано на следующем правиле. Пусть $\alpha \Rightarrow t = t'$ – условное тождество алгебры данных, P – оператор и $\langle P \rangle \alpha$ – истинное утверждение. Тогда последовательность операторов

$$(P; (..., x := t, ...))$$

можно заменить на

$$(P; (..., x := t', ...)).$$

Если t' вычисляется быстрее, чем t, а условие α проверяется достаточно просто, то оператор

$$x := t$$

можно заменить на

если α **то** $x := t'$ **иначе** $x := t$ **ке.**

Указанное преобразование можно применять также с учетом начальных условий для программы в целом. Предположим, что имеет место условие $\beta \Rightarrow \langle P \rangle \alpha$. Тогда если β является следствием начальных условий для программы $(P; (..., x := t, ...))$, а $\alpha \Rightarrow t = t'$ есть условное тождество, то программа

$$(P; (..., x := t', ...))$$

эквивалентна исходной на множестве состояний памяти, удовлетворяющих начальным условиям. Применение тождеств и условных тождеств для выполнения оптимизирующих преобразований может происходить как на уровне функциональной модели, так и на уровне процедурного представления.

Выделение совпадающих подвыражений хотя и сокращает общее число операций, но требует увеличения используемой памяти, поскольку вводится новая вспомогательная переменная для хранения промежуточного результата. В общем случае поиск приемлемого компромисса для различных критериев приводит к сложным комбинаторным задачам, решение которых требует соответствующей автоматической поддержки. Оптимизирующие преобразований программ вычисления элементарных функций особенно важны, когда эти программы входят как модули в другие программы.

Уместно выделить здесь еще один важный критерий оптимизации, связанный с использованием вычислительных систем с автоматическим распараллеливанием вычислительных процессов, например с использованием конвейеризации на уровне команд. Если представлять выражения деревьями, то для эффективной работы конвейера выгодно иметь деревья возможно меньшей высоты, а следовательно, с большим количеством независимых поддеревьев, которые могут вычисляться параллельно. Так, например, схема Горнера для вычисления значения многочлена, соответствующая дереву максимальной высоты, хотя и дает минимальное число операций, плохо загрузит конвейер, а представление в виде произведения линейных и квадратных множителей – хорошо.

После получения окончательного варианта программы вычисления элементарных функций задача сводится к реализации базовых операций и предикатов. Если алгебра D есть алгебра вычислительной системы, то задача проектирования в основном решена. Возможно, еще потребуется преобразование операторов присваивания, если базис операторов и условий является собственным подмножеством стандартного базиса. Рассмотрим, например, следующие ограничения: допускаются только простые присваивания и любое выражение алгебры данных содержит не более одной операции. О таких операторах будем говорить, что они имеют ранг 1. Эти ограничения характерны для машинных языков или языков типа ассемблера. Преобразование программы к требуемому виду выполняется с введением вспомогательных переменных. Например, для реализации оператора $x := \omega(t_1, t_2)$ при условии, что x, t_1, t_2 имеют одинаковые типы, а t_1 и t_2 содержат хотя бы по одной операции, необходимо ввести новую переменную y и преобразовать этот оператор к виду

НАЧАЛО

$y := t_1$;

$x := t_2$;

$x := \omega(y, x)$

КОНЕЦ

а затем аналогичным образом разложить первые два присваивания. Термы t_1 и t_2 можно поменять местами. Число вспомогательных переменных зависит от того, в каком порядке вычислять термы t_1 и t_2.

Известен простой алгоритм, который позволяет получить минимальное число вспомогательных переменных при реализации присваивания $x := t$ в предположении что все промежуточные результаты имеют один и тот же тип. Он основан на следующем соображении. Обозначим через $\varphi(t)$ минимальное число дополнительных переменных, которые необходимо использовать для реализации оператора $x := t$. Тогда, если $t = f(t_1, \ldots, t_n)$, где f получается применением некоторой базовой операции к выражениям $t_1, \ldots, t_n$ и еще, быть может, к некоторым константам и переменным, то $\varphi(t) = \min \max(\varphi(t_{i_1}), \varphi(t_{i_2}) + 1, \ldots, \varphi(t_{i_n}) + n - 1)$, где min берется по всем перестановкам $(i_1, \ldots, i_n)$. Очевидно, этот минимум достигается на перестановке, которая упорядочивает $\varphi(t_1), \ldots, \varphi(t_n)$ по убыванию: $\varphi(t_{i_1}) \geqslant \varphi(t_{i_2}) \geqslant \ldots \geqslant \varphi(t_{i_n})$. А оптимальная

программа получается из программы:

НАЧАЛО

$z := t_{i_1}$;

. . . . . .

$z_{n-1} := t_{i_{n-1}}$;

$x := t_{i_n}$;

$x := f(z_{i_1}, \ldots, z_{i_n})$

КОНЕЦ

дальнейшим разложением первых присваиваний ($z_n = x$). Для того чтобы реализовать указанный алгоритм, нужно исходное выражение t представить в виде дерева (составного объекта), затем, обходя это дерево в глубину, вычислить рекурсивно значения функции φ для всех поддеревьев, а затем построить окончательную программу так, чтобы подвыражения вычислялись в порядке, соответствующем убыванию функции φ.

Рассмотрим теперь ситуацию, когда алгебра D не совпадает с базовой алгеброй вычислительной системы. Тогда возможны два случая. Первый — алгебра D уже реализована средствами базовой алгебры D_0 вычислительной системы (проектирование снизу вверх), второй — алгебру D еще предстоит реализовать (проектирование сверху вниз). Переход к новой алгебре данных означает изменение интерпретации. В случае гомоморфной реализации базовые операции и предикаты можно реализовать, добавив описания соответствующих функций (подпрограмм). Другой способ реализации состоит в подстановке тел соответствующих подпрограмм в места, где они вызываются. Например, если операция ω реализуется подпрограммой, имеющей описание

ПОДПРОГРАММА $A(x_1, \ldots, x_n)$ РЕЗУЛЬТАТ (y) НАЧАЛО

$P(y)$ КОНЕЦ,

то оператор

$x := \omega(t_1, \ldots, t_n)$

можно заменить на последовательность

НАЧАЛО

$x_1 := t_1, \ldots, x_n := t_n$;

$P(x)$

КОНЕЦ

Если внутри подпрограммы есть описания, то их следует добавить к описаниям исходной программы. Подстановки, разумеется, должны производиться с учетом возможных коллизий и переименованием, если нужно, переменных. В описании подпрограммы для краткости не указаны типы параметров, которые должны быть учтены при включении описания входных параметров как новых переменных программы. После подстановки тел подпрограмм могут появиться новые возможности для проведения оптимизирующих преобразований. Возможна также смешанная стратегия реализации операций, при которой некоторые из операций реализуются вызовами подпрограмм, другие — подстановками.

В настоящее время трудно дать исчерпывающее решение даже для простых оптимизационных задач. Поэтому интересно иметь возможно более богатый набор преобразований программ, который давал бы достаточно подробное представление о структуре пространства поиска решений в процессе проектирования. Преобразования эти можно классифицировать по двум основным параметрам. Первый параметр характеризует инварианты преобразований, т.е. свойства программ, которые сохраняются при выполнении преобразований данного класса. Он может быть, в частности, определен эквивалентностью, порождаемой данными преобразованиями. Второй параметр характеризует уровень математической модели, на котором выполняются преобразования (выражения алгебры данных, операторы, условия и т.д.).

Рассмотрим соответствующую классификацию в том виде, как она сложилась в теоретическом программировании. Наиболее общим объектом, определяющим процедурную модель последовательной программы, является детерминированная рекурсивная $U-Y$-схема программы над типизированной памятью с косвенным именованием, интерпретированная на алгебре структур данных D над базовой алгеброй D_0. Взаимодействие подпрограмм в общем случае допускает использование как общей, так и распределенной памяти. Тексты рекурсивных программ могут быть представлены средствами языка типа A2. Относительно базиса (U, Y) предполагается, что он является расширением стандартного базиса (операторы присваивания) путем добавления операторов вызова подпрограмм. Другие модели получаются путем введения ограничений на структуру программ и базиса, в частности рассмотрение только регулярных программ, либо абстрагированием от тех или иных структурных характеристик базиса и его интерпретации.

Основные виды эквивалентности программ следующие:

– строгая эквивалентность;

– функциональная эквивалентность схем программ над памятью;

– эквивалентность относительно многообразия (квазимногообразия) алгебр данных;

– эквивалентность относительно заданной интерпретации.

Две схемы программы эквивалентны относительно заданной интерпретации, если они вычисляют одно и то же преобразование на множестве всех состояний памяти. Эта эквивалентность допускает некоторые ослабления.

Первое ослабление связано с выделением входных и выходных переменных программы. Если такие переменные выделены, то в качестве главной функциональной модели можно рассматривать функцию, которая отображает множество всех состояний входных переменных в множество состояний выходных переменных при условии, что в начальном состоянии памяти значения всех переменных, которые не являются входными, не определены. Две схемы программ эквивалентны относительно входных и выходных переменных при заданной интерпретации, если они определяют одну и ту же функциональную модель указанного типа.

Другое ослабление эквивалентности связано с начальными условиями. Две схемы эквивалентны относительно начального условия α, если их главные функциональные модели совпадают на множестве всех состояний входных переменных, удовлетворяющих условию α.

При переходе от заданной интерпретации к другой интерпретации – с изоморфной алгеброй данных – отношение эквивалентности программ, очевидно, не изменится. Дальше будет показано, что для программ с простой памятью эквивалентность сохраняется также при переходе к новой интерпретации с тем же самым множеством условных тождеств.

Многие важные преобразования программ выполняются с учетом лишь некоторых тождеств, имеющих место в алгебре данных. Например, для приведения арифметических выражений к виду, удобному для конвейеризации, достаточно использовать лишь соотношения коммутативности и ассоциативности, реже – дистрибутивность. Такие преобразования определяют эквивалентность, которая сохраняется при переходе от одной интерпретации к другой с сохранением выделенных тождеств. Точнее, указанную эквивалентность можно определить следующим образом. Пусть $\mathcal{H}$ есть некоторое множество тождеств или условных тождеств алгебры данных. Эти тождества могут включать в себя также и тождества алгебры условий, рассматриваемой как одна из компонент алгебры данных. Множество $\mathcal{H}$ определяет класс интерпретаций, который порождается многообразием алгебр данных, если $\mathcal{H}$ состоит из тождеств, или квазимногообразием, и $\mathcal{H}$ содержит также условные тождества (квазитождества). Две схемы программ называются эквивалентными относительно $\mathcal{H}$, если они эквивалентны относительно класса всех интерпретаций, порождаемых множеством $\mathcal{H}$. Так же, как и выше, некоторое ослабление можно получить, если учитывать входные, выходные переменные и начальные условия.

Если множество $\mathcal{H}$ пусто, то получается эквивалентность относительно класса всех интерпретаций, допустимых для схем программ над памятью. Такая эквивалентность называется функциональной эквивалентностью схем программ над памятью. Она учитывает только синтаксическую структуру операторов и условий, а также структуру памяти схем программ. Преобразования, определяемые функциональной эквивалентностью, позволяют решать многие задачи проектирования. В частности, описанное выше решение задачи реализации присваивания с минимальным числом вспомогательных переменных выполняется с учетом входных и выходных переменных с помощью преобразований функциональной эквивалентности.

Если отвлечься от синтаксической структуры базиса рассматриваемого класса схем программ и рассматривать схемы этого класса как абстрактные $U-Y$-схемы, приходим к классу всех вообще интерпретаций, не обязательно допустимых, и понятию строгой эквивалентности схем программ. Две схемы строго эквивалентны, если они эквивалентны относительно любой интерпретации базиса (U, Y). Строгая эквивалентность – это самая сильная эквивалентность схем программ. Она порождает мало преобразований, но наиболее просто обозрима. Различные ослабления эквивалентности абстрактных схем программ можно получить, фиксируя те или иные соотношения между операторами и условиями, выразимые в виде соотношений алгебры алгоритмов.

Рассмотрим теперь классификацию преобразований по уровням математических моделей. В качестве таких уровней можно выделить следующие:

– выражения алгебры данных;

– базовые операторы и условия;

– полугруппа операторов;

– алгебра условий (с операцией умножения условия на оператор);

– бесцикловые программы, регулярные программы и схемы программ;

– программы с циклами, регулярные программы и схемы программ (без подпрограмм);

– рекурсивные программы.

Некоторые виды эквивалентности программ будут рассмотрены дальше. Здесь ограничимся рассмотрением преобразований на первых уровнях, включая бесцикловые программы, которые применяются для реализации вычисления элементарных функций.

Выражения алгебры данных. Два выражения t и t' эквивалентны (интерпретация фиксирована), если их значения совпадают при любых состояниях памяти. Поскольку имеет смысл сравнивать только однотипные выражения, то эквивалентность выражений t и t' типа ξ означает равенство $\mathrm{val}_\xi(t, b) = \mathrm{val}_\xi(t', b)$ для любых состояний памяти b. Если память простая, то эквивалентность выражений $t(v_1, \ldots, v_n)$ и $t'(v_1, \ldots, v_n)$, зависящих от переменных $v_1, \ldots, v_n \in V$, означает, что равенство $t(v_1, \ldots, v_n) = t'(v_1, \ldots, v_n)$ является тождеством алгебры данных. При этом следует помнить, что если допускаются частично определенные состояния памяти, то неопределенный элемент w должен принадлежать алгебре данных. В дальнейшем будем предполагать, что состояния памяти всюду определены, а частичная определенность, если она нужна, моделируется с помощью неопределенного элемента. В случае косвенного именования или нетривиальных ограничений на допустимые состояния памяти эквивалентность выражений может не совпадать с тождеством. Например, пусть в алгебре данных определена операция $\varphi(x, i)$. Первый аргумент – одномерный числовой массив, второй – целое число. Операция определена равенством $\varphi(x, i) = x(i) + x(i + 1)$. Очевидно, что выражения, из которых составлено равенство, эквивалентны. Однако это равенство нельзя рассматривать как тождество алгебры данных по двум причинам. Первая состоит в том, что в левую часть вместо x следует подставлять массив, а вместо i – число, что после выполнения операции даст снова число. В то же время в правую часть следует подставлять только значение i, что после выполнения операций даст выражение (скажем, $x(1) + x(2)$ при $i = 1$). Вторая причина состоит в том, что если теперь будет сделана попытка подставить произвольные значения переменных $x(1)$ и $x(2)$, то будет потеряна связь со значением массива x, которое уже подставлено в левую часть.

И все же эквивалентность выражений может быть сведена к тождествам, если алгебру данных расширить путем введения компоненты B, состоящей из всевозможных семейств $(b_\xi)_{\xi \in \Xi}$, где $b_\xi : V_\xi \to D_\xi$. Элементы $b \in B$ состояния памяти можно рассматривать как теоретико-множественные структуры данных над исходной алгеброй. Кроме B, следует еще рассмотреть компоненты $T_\Omega(D_\xi^{(0)}, V_\xi, \phi)$, представляющие выражения типа ξ, где $D_\xi^{(0)}$ – множество констант типа ξ, которые разрешается использовать в выражениях этого типа. Тогда функцию $\mathrm{val}_\xi(t, b)$ можно рассматривать как операцию расширенной алгебры данных, и эквивалентность t и t' сведется к тождественному равенству $\mathrm{val}_\xi(t, b) = \mathrm{val}_\xi(t', b)$. Следует обратить внимание на то, что в этом равенстве b – переменная, а t и t' – кон-

станты. Вместо операции val_ξ можно рассматривать операцию $b_\xi(t)$ применения функции b_ξ к аргументу t. Поскольку b_ξ можно пронести через операции типа ξ до именующих выражений соответствующих типов, то можно обойтись без компонент вида $T_\Omega(D_\xi^{(0)}, V_\xi, \phi)$, а равенство $\mathrm{val}_\xi(t, b) = \mathrm{val}_\xi(t', b)$ после использования определений § 3 гл. 2, превратится в функциональное равенство с произвольными функциями b_ξ.

В частных случаях можно также обойтись без функций b_ξ. Например, пусть косвенное именование реализуется с помощью переменных с индексами $x(i_1, \ldots, i_m)$. Тогда если вместо x не могут подставляться именующие выражения, то каждую переменную x можно рассматривать как произвольную функцию на целых числах (с учетом области определения), и эквивалентность выражений сведется к тождеству в алгебре функциональных структур данных.

Поскольку эквивалентность выражений сводится к их тождественному равенству в алгебре данных (возможно, обогащенной функциональными структурами), то изменение алгебры данных с сохранением тождеств не меняет эквивалентности выражений. Поэтому если тождества алгебры данных известны достаточно хорошо, то, рассматривая выражения этой алгебры, можно игнорировать реальную структуру ее элементов, отождествив эти элементы с выражениями алгебры данных, рассматриваемыми с точностью до тождественных соотношений. Такая точка зрения соответствует тому, что в качестве представителя многообразия всех алгебр, обладающих данной системой тождеств, выбирается алгебра, свободная в этом многообразии (инициальная алгебра в терминологии абстрактных типов данных). Ее можно получить, переходя от алгебры $(D_\xi)_{\xi \in \Xi}$ к алгебре с компонентами $T_\Omega(D_\xi^{(0)}, V_\xi, \mathcal{H}_\xi)$, где $D_\xi^{(0)}$ — константы, используемые в выражениях, $(V_\xi)_{\xi \in \Xi}$ — память, $\mathcal{H}_\xi$ — тождества алгебры D_ξ.

Рассмотрим несколько типичных примеров.

1. Числовые алгебры с арифметическими операциями. Сложение и вычитание вместе с константой 0 удовлетворяют аксиомам абелевой группы независимо от того, какие значения принимают переменные — целые, рациональные или комплексные. Алгебра выражений над переменными $v_1, \ldots, v_n$ простой памяти изоморфна свободной абелевой группе с n образующими. Добавление констант, не связанных линейным соотношением, только увеличивает размерность.

Добавим умножение на константы, рассматриваемые как 0-арные операции. Если множество констант замкнуто относительно сложения, вычитания и умножения, то алгебра выражений становится модулем над кольцом констант. Если константы образуют поле, алгебра данных становится линейным пространством.

При рассмотрении сложения, вычитания и умножения получаем кольцо многочленов, а деление превращает алгебру данных в поле рациональных функций.

2. Пусть V_m — множество переменных m-мерных числовых массивов ($m \geqslant 0$, размеры неопределенные). Индексы принимают целочисленные значения, их можно складывать и вычитать. Аргументами операции $x[i_1, \ldots, i_m]$ являются элемент x множества V_m и набор целых $i_1, \ldots, i_m$.

Алгебра выражений является абелевой группой, на которой действуют дополнительные операции образования переменных с индексами. Эти операции не входят ни в какие тождества, т.е. алгебра только с этими операциями абсолютно свободна. Полученная абелева группа имеет бесконечное число образующих, поскольку любой набор переменных с индексами (индексы могут быть также вложенными: $x[y[i, x[j]]]$) порождает свободную подгруппу.

3. Слова с операцией конкатенации образуют свободную полугруппу. Единственное тождество — ассоциативность. Добавив (смешанную) операцию $p(i:j)$ — часть слова от i-го символа по j-й включительно (i, j — целые >0), получим новые тождества такого, например, типа: $p(i:i+m) * p(i+m+1:i+m+n) = p(i:i+m+n)$. Точная характеристика тождеств зависит от того, как определяется $p(i:j)$ для случаев, когда $i \leqslant 0$, или $j < i$, или $j - i + 1$ больше длины слова.

4. Множество записей данного типа, например типа записи (A: целое, B: запись (C: целое, D: слово)), образует компоненту алгебры данных с операциями, соответствующими данному типу. Например, если X есть запись указанного типа, то $X.A$ — целое, $X.B$ — запись типа записи (C: целое, D: слово), $X.B.C$ — целое, $X.B.D$ — слово. Все четыре примера — унарные операции. Ни в каких тождествах они не участвуют (действуют свободно).

5. Алгебра функциональных структур данных подробно изучалась в гл.4.

Базовые операторы и условия. Два оператора эквивалентны, если они определяют одно и то же преобразование информационной среды (при заданной интерпретации или при всех интерпретациях из заданного класса интерпретаций). Рассмотрим сначала групповые операторы над простой памятью. Тождественный оператор ϵ и нигде не определенный оператор w будем рассматривать как частные случаи. В операторе $(x_1 := t_1, \ldots, x_n := t_n)$ все переменные в левой части должны быть различными. Следующие преобразования сохраняют функциональную эквивалентность операторов.

1. Перестановка простых присваиваний.

2. Добавление присваивания $x := x$, если x не входит в левую часть ни одного из простых присваиваний.

3. Удаление оператора $x := x$.

4. $(x := x) = \epsilon$.

Операторы $(x_1 := t_1, \ldots, x_n := t_n)$ и $(x_1 := t'_1, \ldots, x_n := t'_n)$ эквивалентны относительно заданной интерпретации, если $t_1 = t'_1, \ldots, t_n = t'_n$ (равенство означает эквивалентность выражений, а следовательно, и тождественное равенство этих выражений в алгебре данных). Таким образом, следующее преобразование сохраняет эквивалентность относительно заданной интерпретации.

5. Замена правой части простого присваивания эквивалентным выражением.

Преобразования 1–5, очевидно, образуют полную систему эквивалентных преобразований операторов, поскольку любые два эквивалентных оператора с помощью этих преобразований превращаются друг в друга. Если сравнивать операторы на множестве входных и выходных переменных, то добавляется еще одно преобразование:

6. Добавление или удаление простых операторов, в которых переменная левой части не является выходной.

Если операторы сравниваются относительно начального условия α, то возможен еще один тип преобразования:

7. Если имеет место условное тождество $\alpha \Rightarrow t = t'$, то в правой части присваивания t может быть заменено на t'.

Для операторов вызова подпрограмм $(y_1, \ldots, y_m) := A(t_1, \ldots, t_n)$ дело обстоит несколько сложнее. Не зная, как устроена программа A, можно говорить только о возможности замены фактических параметров $y_1, \ldots, y_m$ и $t_1, \ldots, t_n$ эквивалентными им выражениями. Более тонкое сравнение вызовов сводится к сравнению соответствующих им программ.

Сравнение групповых операторов над памятью с косвенным именованием – задача более сложная, поэтому ограничимся рассмотрением только простых присваиваний. Из определения выполнения присваиваний над памятью с косвенным именованием (§ 3 гл. 2) следует, что операторы $s := t$ и $s' := t'$ эквивалентны тогда и только тогда, когда термы t и t' имеют один и тот же тип ξ, $\alpha^{-1}\xi$ определено и для любого состояния памяти b имеют место равенства $\mathrm{val}_{\alpha^{-1}\xi}(s, b) = \mathrm{val}_{\alpha^{-1}\xi}(s', b)$, $\mathrm{val}_{\xi}(t, b) = \mathrm{val}_{\xi}(t', b)$.

Базовые условия удобно рассматривать как выражения алгебры данных, которые принимают значения в компоненте этой алгебры, состоящей из двух или трех элементов (0, 1 или 0, 1, w) в зависимости от того, допускается ли неопределенное значение для базовых условий (а следовательно, и выражений вообще) или не допускается. Операциями алгебры базовых условий являются базовые предикаты $\pi(d_1, \ldots, d_n)$, аргументы которых пробегают соответствующие компоненты алгебры данных. Кроме того, на алгебре базовых условий могут быть определены и другие операции, например пропозициональные связки $\wedge$, $\vee$, $\neg$, распространенные, если нужно, на неопределенное значение (монотонным образом для алгебр с аппроксимацией). Таким образом, эквивалентные преобразования базовых условий над простой памятью сводятся к тождественным преобразованиям выражений алгебры условий, а эквивалентность над памятью с косвенным именованием – к тождествам алгебры структур данных.

Полугруппа операторов. Каждый оператор выполняет преобразование информационной среды (памяти), а последовательному выполнению соответствует произведение этих преобразований. Таким образом, множество базовых операторов порождает полугруппу преобразований. Соотношения в этой полугруппе – основной источник преобразований линейных программ, т.е. программ, представляющих собой последовательную композицию базовых операторов. Множество базовых операторов присваивания над простой памятью даже замкнуто относительно умножения в силу основного соотношения

$$\begin{aligned}&(x_1 := t_1, \ldots, x_n := t_n;\ x_1 := t'_1, \ldots, x_n := t'_n) = \\ &= (x_1 := t'_1(t_1, \ldots, t_n), \ldots, x_n := t'_n(t_1, \ldots, t_n)).\end{aligned} \tag{2.2}$$

С помощью этого соотношения любая последовательность операторов присваивания может быть свернута в одно присваивание, и эквивалентность линейных программ таким образом сводится к эквивалентности операторов.

Эквивалентность линейных программ относительно основных переменных — более слабая эквивалентность по сравнению с полугрупповой. Но для программ над простой памятью она может быть сведена к полугрупповой эквивалентности. Действительно, пусть P и Q — две линейные программы над конечной памятью V, V_0 — множество входных переменных, V_1 — множество выходных переменных. Пусть

$$P_0 = (z_1 := w, \ldots, z_m := w),$$

$$P_1 = (z_1' := w, \ldots, z_n' := w),$$

где $\{z_1, \ldots, z_m\} = V \setminus V_0$, $\{z_1', \ldots, z_n'\} = V \setminus V_1$. Операторы P и Q эквивалентны относительно основных переменных V_0 и V_1 тогда и только тогда, когда $P_0PP_1 = P_0QP_1$ в полугруппе, порожденной базовыми операторами. Вместо неопределенного элемента w, разумеется, можно использовать любую другую (одну и ту же для каждого типа) константу алгебры данных.

Преобразования 1—5 вместе с соотношением (2.2) образуют полную систему эквивалентных преобразований линейных программ над простой памятью. Существует, однако, много других полезных преобразований линейных программ, которые прямо из этих преобразований не выводятся. Примером может служить упоминавшееся в начале параграфа преобразование с использованием условных тождеств алгебры данных.

Алгебра условий. Алгебра условий над алгеброй данных D и памятью V рассматривается как промежуточный уровень при построении алгебры алгоритмов. Эта алгебра порождается базовыми условиями с помощью операций дизъюнкции, конъюнкции, отрицания и операции умножения оператора на условие. Сами условия — это выражения алгебры условий, рассматриваемые с точностью до эквивалентности относительно заданной интерпретации или класса интерпретаций. Некоторые тождества алгебры условий рассматривались в § 4 гл. 2.

Если операторы — это присваивания над простой памятью, то с помощью этих тождеств, а также соотношения

$$(x_1 := t_1, \ldots, x_n := t_n)(\alpha(x_1, \ldots, x_n)) \Longleftrightarrow \alpha(t_1, \ldots, t_n)$$

всякое выражение в алгебре условий можно свести к выражению в алгебре базовых условий с логическими операциями или к элементарному условию (пропозициональной функции от базовых). Таким образом, эквивалентность условий сводится к эквивалентности базовых (элементарных) условий. В случае памяти с косвенным именованием такое сведение также возможно, но несколько более сложным образом.

Рассмотрим пример. Пусть $\alpha = (x[g] := t)\beta(x[g_1], \ldots, x[g_k])$ и $M_0, M_1, \ldots, M_{2^k-1}$ — все подмножества множества $\{1, \ldots, k\}$. Рассмотрим 2^k условий $\alpha_0, \alpha_1, \ldots$, определенных формулой $\alpha_i = \bigwedge\limits_{j \in M_i} g = g_j \wedge \bigwedge\limits_{j \notin M_i} g \neq g_j$, и 2^k условий $\beta_0, \beta_1, \ldots$, определенных формулой $\beta_i = \beta(t_{i1}, \ldots, t_{ik})$, где $t_{ij} = t$, если $j \in M_i$, и $t_{ij} = x[g_j]$, если $j \notin M_i$. Теперь можно утверждать что $\alpha = \alpha_0 \wedge \beta_0 \vee \alpha_1 \wedge \beta_1 \vee \ldots \vee \alpha_{2^k-1} \wedge \beta_{2^k-1}$. Разумеется, для конкретных выражений все упрощается. Аналогично можно поступать и в других случаях.

Бесцикловые программы. Регулярные бесцикловые программы с помощью соотношений § 4 гл. 2 приводятся к канонической форме вида

если α_1 **то** P_1 **иначе**

если α_2 **то** P_2 **иначе** (2.3)

. . . . . .

если α_m **то** P_m **иначе** P_{m+1} **ке**,

где $\alpha_1, \ldots, \alpha_m$ – элементарные условия, а $P_1, \ldots, P_m$ – линейные программы. Если каждое из P_i есть произведение присваиваний над простой памятью, то каждое из них можно привести к одному присваиванию.

Эквивалентность двух бесцикловых программ сводится к эквивалентности операторов относительно начальных условий. Действительно, если P есть программа (2.3), а Q имеет вид

если β_1 **то** Q_1 **иначе**

если β_2 **то** Q_2 **иначе**

. . . . . .

если β_n **то** Q_n **иначе** Q_{n+1} **ке**,

то программы P и Q эквивалентны тогда и только тогда, когда для любых $i = 1, \ldots, m+1$ и $j = 1, \ldots, n+1$ операторы P_i и Q_j эквивалентны относительно начального условия $\bar{\alpha}_1 \wedge \ldots \wedge \bar{\alpha}_{i-1} \wedge \alpha_i \wedge \bar{\beta}_1 \wedge \ldots \wedge \bar{\beta}_{j-1} \wedge \beta_j$ (при условии что $\alpha_{m+1} = \bar{\alpha}_1 \wedge \ldots \wedge \bar{\alpha}_m$, а $\beta_{n+1} = \bar{\beta}_1 \wedge \ldots \wedge \bar{\beta}_n$).

Для сравнения произвольных бесцикловых детерминированных схем программ проще всего их регуляризовать и сравнивать регулярные схемы.

Если в алгебру данных ввести операцию (**если** α **то** d_1 **иначе** d_2), то присваивания с условиями можно преобразовать в безусловные присваивания с помощью соотношений типа

(**если** α **то** $x_1 := t_1, \ldots, x_n := t_n$ **иначе** $x_1 := t'_1, \ldots,$
$x_n := t'_n$ **ке**) = (x_1 := **если** α **то** t_1 **иначе** $t'_1, \ldots, x_n :=$ **если**
α **то** t_n **иначе** t'_n).

С помощью этого преобразования любая бесцикловая программа над простой памятью может быть приведена к одному присваиванию, а это присваивание – к системе равенств вида (2.1), определяющих функциональную модель бесцикловой программы.

Упражнения

1. Написать программу для вычисления многочлена $y = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0$ в виде последовательности простых присваиваний ранга не больше 1 так, чтобы эта программа обеспечивала максимальную загрузку конвейера.

2. Найти полную систему тождеств двухосновной алгебры (D_1, D_2), где D_1 – полугруппа слов, D_2 – множество целых чисел с операциями сложения, вычитания и константой 0 (абелева группа), с операцией $p(i:j)$, которая выделяет часть слова (нумерация символов начиная с 1), если полагать, что $p(i:j) = e$ при $j < i$ и при $j > n$, где n – длина слова p, а при $i \leqslant j$, если $i < 1$; то $p(i:j) = p(1:j)$; если же $j > n$, то $p(i:j) = p(i:n)$.

3. Найти условия, при которых операторы $(x(t_1) := g_1,\ x(t_2) := g_2)$ и $(x(t'_1) := g'_1,\ x(t'_2) := g'_2)$ эквивалентны.

4. Сформулировать необходимые и достаточные условия эквивалентности групповых присваиваний над памятью с косвенным именованием.

5. Выписать явно полную систему эквивалентных преобразований бесцикловых регулярных программ.

§ 3. Вычисление функций над структурами данных

В этом параграфе будут рассматриваться функциональные структуры данных. Общая теория функций над такими структурами изложена в гл. 4. Здесь будут рассмотрены некоторые более конкретные вопросы реализации вычисления периодически определенных функций – следующего по сложности проектирования класса функций после элементарных. Программы, вычисляющие периодически определенные функции, так же, как и программы вычисления элементарных функций, обычно используются как части более сложных программ.

В качестве функциональных моделей служат канонические системы уравнений в алгебре структур данных, которые будем записывать в скалярном виде:

$$y_i(c) = Q_{ij}(z_{i1}(cg_{i1}), \ldots, z_{il_i}(cg_{il_i})), \quad c \in H_{ij}, \tag{3.1}$$

$i = 1, \ldots, m$; $j = 1, \ldots, k_i$; $z_{i1}, \ldots, z_{il_i} \in \{y_1, \ldots, y_m, x_1, \ldots, x_n\}$ (см. § 2 гл. 4). Задача состоит в том, чтобы вычислить значения функций $y_1, \ldots, y_m$ во всех точках областей $H_1, \ldots, H_m \subset C$ соответственно. Рассмотрим сначала случай, когда D есть стандартное расширение области D_0 определенных элементов путем добавления w и $\bar{w}$ со стандартным доопределением базовых операций, а операция $\sqcup$ не встречается в Q_{ij} (нет дизъюнктивных вершин). В соответствии с общей теорией вычисления следует производить только для переменных из множества $T = \bigcup_{i=1}^{m} (R^{-1})^*(P_i)$, где R – отношение информационной зависимости, представленное графом (S, R), $P_i = \{y_i(c) \mid c \in H_i\}$. Предполагаем, что множество T конечно. Все возможные последовательные вычисления функций $y_1, \ldots, y_m$ описываются подсистемой максимально асинхронной системы $B = D^S$, в которой на каждом шаге вычисляется значение только одной скалярной переменной. Иными словами, функция переходов δ этой системы переводит состояние b в состояние $b' \Longleftrightarrow b' = bP$, где P – любой из операторов присваивания вида

$$y_i(c) := Q_{ij}(z_{i1}(cg_{i1}), \ldots), \quad i = 1, \ldots, m, \quad j = 1, \ldots, k_i,$$

такой, что все значения $z_{i1}(cg_{i1}), \ldots$ в состоянии b определены. Если граф (S, R) является бесцикловым (по крайней мере на множестве T), то последовательный порядок вычисления переменных $y_i(c)$ определяется некоторой линеаризацией частичного порядка R^*, порожденного отношением R: если $y_i(c) \xrightarrow{R^*} y_j(c')$, то $y_j(c')$ может вычисляться лишь после того, как будет вычислена переменная $y_i(c)$. Если же среди точек множества T есть такие, которые лежат на циклах, то их следует отбросить, поскольку значения этих переменных всегда не определены.

Общий метод вычисления может быть описан с помощью простой рекурсивной программы, использующей произвольный линейный порядок на T. Все точки этого множества обходятся в заданном порядке, и для каждой делается попытка вычисления функции $y_1, \ldots, y_m$ в этой точке с помощью рекурсивной программы ВЫЧИСЛИТЬ y_i В ТОЧКЕ (c). Эта программа проверяет возможность вычисления переменной $y_i(c)$ путем проверки, вычислены ли уже значения скалярных переменных $z_1(c_1)$, $z_2(c_2), \ldots$, от которых зависит $y_i(c)$ непосредственно. В случае невозможности производится попытка вычисления значений $z_1(c_1)$, $z_2(c_2), \ldots$ путем рекурсивного обращения к программе ВЫЧИСЛИТЬ z_j В ТОЧКЕ (c_j).

Программа вычисления структур данных $y_1(x)/H_1, \ldots, y_m(x)/H_m$ имеет следующий вид:

ПРОГРАММА $F(x_1, \ldots, x_n)$ РЕЗУЛЬТАТ $(y_1, \ldots, y_m)$;
НАЧАЛО
  ДЛЯ ВСЕХ $y_i(c) \in T$ ВЫПОЛНИТЬ
    ЕСЛИ $y_i(c)$ НЕ ВЫЧИСЛЕНО
    ТО ВЫЧИСЛИТЬ (y_i) В ТОЧКЕ (c) КЕ
  КЦ.
КОНЕЦ.

Кроме указанных формальных параметров программа может содержать другие параметры, характер которых зависит от того, какие элементы функционального описания фиксированы, а какие варьируются путем задания фактических параметров:

ПРОГРАММА ВЫЧИСЛИТЬ (y_i) В ТОЧКЕ (c).
НАЧАЛО.
  ДЛЯ $j := 1$ ДО k_i ВЫПОЛНИТЬ
    ЕСЛИ $c \in H_{ij}$ ТО
      ЕСЛИ $z_{i1}(c_{i1})$ НЕ ВЫЧИСЛЕНО, ТО
        ВЫЧИСЛИТЬ (z_{i1}) В ТОЧКЕ (cg_{i1}) КЕ;
      ЕСЛИ $z_{i2}(cg_{i2})$ НЕ ВЫЧИСЛЕНО, ТО
        ВЫЧИСЛИТЬ (z_{i2}) В ТОЧКЕ (cg_{i2}) КЕ;
      . . . . . .
      ЕСЛИ $z_{il_i}(cg_{il_i})$ НЕ ВЫЧИСЛЕНО, ТО
        ВЫЧИСЛИТЬ (z_{il_i}) В ТОЧКЕ (cg_{il_i}) КЕ
        $y_i(c) := Q_{ij}(z_{i1}(cg_{i1}), \ldots, z_{il_i}(cg_{il_i}))$
      ЗАПОМНИТЬ, ЧТО $y_i(c)$ ВЫЧИСЛЕНО
      ВЫЙТИ ИЗ ЦИКЛА
    КОНЕЦ ЕСЛИ
  КОНЕЦ ЦИКЛА.
КОНЕЦ.

Программа ВЫЧИСЛИТЬ не имеет локальных переменных, и все ее вызовы работают над общей памятью. Признак ВЫЧИСЛЕНО вначале приписан всем переменным типа $x_i(c)$, а для всех $y_i(c)$ он принимает значение "ложно". Заметим, что на самом деле вычисленная переменная может иметь и неопределенное значение, если в базовой алгебре предусмотрено его представление и, скажем, переменные $x_i(c)$ определены не во всех точках c таких, что $x_i(c) \in T$. Дискретная система, соответствующая данной программе, получается, если управляющую компоненту этой

программы спроецировать на состояния, в которых выполняются присваивания $y_i(c) := Q_{ij}$.

Программа F будет работать правильно и в том случае, если заголовок ДЛЯ ВСЕХ $y_i(c) \in T$ ВЫПОЛНИТЬ заменить на заголовок ДЛЯ ВСЕХ $y_i(c) \in \bigcup_{j=1}^{m} P_j$ ВЫПОЛНИТЬ, поскольку обращение к другим переменным множества T произойдет при рекурсивных вызовах программы ВЫЧИСЛИТЬ. Это позволяет не определять заранее все множество T, которое может быть устроено достаточно сложно. Если заранее не известно, что граф (S, R) не имеет циклов (на множестве T), то соответствующие проверки необходимо встроить в программу, например путем использования признака ВЫЧИСЛЕНИЕ НАЧАТО, который вырабатывается перед каждым обращением к программе ВЫЧИСЛИТЬ. Тогда, если пришлось вторично обратиться к вычислению переменной, для которой ВЫЧИСЛЕНИЕ НАЧАТО, фиксируется наличие цикла. Если программа допускает оперирование с неопределенным значением, соответствующая переменная получает значение w, и вырабатывается признак ВЫЧИСЛЕНО. В противном случае программа останавливается, передавая аварийное сообщение.

В некоторых случаях удается заранее упорядочить множество T так, что этот порядок будет линеаризацией отношения R^*. Тогда внутри цикла по c, выполняемого в соответствии с таким порядком, переменные, от которых зависит $y_i(c)$, уже всегда вычислены, и рекурсивного обращения к программе ВЫЧИСЛИТЬ делать не нужно. После подстановки ее тела в основную программу и соответствующих упрощений последняя получит следующий вид:

```
НАЧАЛО
  ДЛЯ ВСЕХ y_i(c) ∈ T ВЫПОЛНИТЬ
    ДЛЯ j := 1 ДО k_i ВЫПОЛНИТЬ
      ЕСЛИ c ∈ H_ij ТО
        y_i(c) := Q_ij;
        ВЫЙТИ
      КЕ
    КЦ.
  КЦ
КОНЕЦ.
```

Если область D не является стандартным расширением области определенных элементов, то приходится пользоваться общей теорией, а хорошие алгоритмы получаются лишь при определенных предположениях. Рассмотрим следующую, практически важную ситуацию. Пусть D есть алгебра с отношением аппроксимации, которое удовлетворяет условию обрыва возрастающих цепей, т.е. всякая возрастающая последовательность $d_1 \sqcup d_2 \sqcup \ldots$ элементов области D стабилизируется через конечное число шагов. Примером может служить алгебра $D = \Gamma(C', D')$ структур данных, расположенных на конечной области расположения C', т.е. случай двухуровневых структур данных $\Gamma(C, D) = \Gamma(C, \Gamma(C', D'))$. Пусть также известно, что T является конечным множеством, а T_0 – множество перемен-

ных $z \in T$, для которых $R^{-1}(z) = \phi$. Тогда для вычисления функций $y_1, \ldots, y_m$ может быть использована следующая программа:

ПРОГРАММА $F_1(x)$ РЕЗУЛЬТАТ (y);
НАЧАЛО.
$T_1 := T_0$; $T_2 := \phi$;
ЦИКЛ.
ДЛЯ ВСЕХ $z \in T$ ТАКИХ, ЧТО $R^{-1}(z) \cap T_1 \neq \phi$, ВЫПОЛНИТЬ
$z' := Q(z)$;
ЕСЛИ $z' \neq z$ ТО
$z := z'$; ДОБАВИТЬ z К T_2
КЦ;
ЕСЛИ $T_2 \neq \phi$ ТО $T_1 := T_2$; $T_2 := \phi$ ИНАЧЕ ВЫЙТИ КЕ
КЦ
КОНЕЦ.

Оператор $z' := Q(z)$ определен таким образом, что если $z = y_i(c)$, а $c \in H_{ij}$, то он эквивалентен присваиванию $y_i(c) := Q_{ij}(z_{i1}(cg_{i1}), \ldots)$. В отличие от программы F, время выполнения которой пропорционально числу элементов множества T, время работы программы F_1 зависит от скорости стабилизации возрастающих цепочек значений переменных множества T, а это в свою очередь может существенно зависеть от порядка перебора элементов множества T во внутреннем цикле. Выбор соответствующего порядка следует осуществить на последующих этапах проектирования программы T_1, зависящих от структуры алгебры D.

Дальнейшее проектирование программ F и F_1 требует также принятия решений о представлении области C, допустимых множеств и сдвигов. Эти решения уже могли быть приняты заранее, и тогда дело сводится к тому, чтобы сделать следующий шаг уточнения программы с использованием заданной информации.

Рассмотрим два случая, наиболее часто встречающиеся на практике. Первый случай соответствует произвольным конечным объектам, второй — целочисленной решетке с линейными сдвигами и допустимыми областями, задаваемыми линейными равенствами и неравенствами.

Итак, пусть C — конечное множество, а полугруппа сдвигов порождается сдвигами $g_1, \ldots, g_m$. Множество C удобно представлять как множество вершин ориентированного графа, дуги которого отмечены базовыми сдвигами $g_1, \ldots, g_m$. Дуга, отмеченная сдвигом g_i, соединяет вершину c с вершиной c' тогда и только тогда, когда $cg_i = c'$. Если из c не выходит дуга, отмеченная сдвигом g_i, то cg_i считается неопределенным. Обратно, любой конечный граф, у которого степень исхода вершин не превосходит m, удобно представлять как область C, на которой действует полугруппа сдвигов, порожденная элементами $g_1, \ldots, g_m$, и который получается произвольной разметкой дуг. Структуры данных, расположенные на таких областях C, — это функции на графах. Можно перенумеровать элементы множества C, скажем, числами от 1 до n и отождествить вершину c с ее номером. Тогда базовые сдвиги могут быть заданы с помощью двумерного массива СДВ: МАС$(1:n, 1:m)$ЦЕЛЫХ, заполненного таким образом, что cg_i = = СДВ(c, i), если только cg_i определено. Если же cg_i не определено, можно

положить СДВ$(c, i) = 0$. Теперь любая структура данных, расположенная на C и принимающая значения в области D, может быть представлена в виде массива МАС$(1 : n)$ элементов типа D. Если в качестве допустимых множеств выбираются произвольные подмножества множества C, то каждое из них может быть задано своей характеристической функцией, определенной с помощью такого же массива и принимающей значения в множестве $\{0, 1\}$. Для некоторых специальных множеств можно, конечно, предусмотреть другие, более экономные представления (прежде всего по памяти).

Пример 1. Рассмотрим задачу определения минимальных стоимостей путей на графе C без циклов. Функциональная модель этой задачи была рассмотрена в § 2 гл. 4. Эта модель определяет структуру данных y с помощью системы скалярных уравнений

$$y(c) = \min(y(cg_1) + x_1(c), \ldots, y(cg_m) + x_m(c)), \quad c \in H_1 \subset C;$$

$$y(c) = 0, \quad c \in H_0 \subset C.$$

Сдвиги g_i предполагаются всюду определенными на множестве $C \setminus H_0$, и $H_1 g_i \subset H_1 \cup H_0$. Множество T в этом случае совпадает с $H_1 \cup H_0$, но внешний цикл программы F можно выполнять по элементам множества H_1. С учетом принятой реализации структур данных программу F можно теперь переписать следующим образом.

```
ПРОГРАММА F(x_1, ..., x_n:МАС(1:N) ЦЕЛ, СДВ:МАС(1:N, 1:M)
ЦЕЛ, H_0, H_1:МАС(1:n) ЦЕЛ, M, N:ЦЕЛ)РЕЗ(y:МАС(1:N)ЦЕЛ);
  ВЫЧИСЛЕНО:МАС(1:N) ЦЕЛ;
    ПОДПРОГРАММА ВЫЧИСЛИТЬ (c:ЦЕЛ);
    НАЧАЛО.
      ДЛЯ i:=1 ДО m ВЫПОЛНИТЬ
        ЕСЛИ НЕ ВЫЧИСЛЕНО (СДВ(c, i)) ТО
          ВЫЧИСЛИТЬ (СДВ(c, i)) КЕ;
      КЦ
      y(c):=min(y(СДВ(c,i))+x_1(c), ..., y(СДВ(c,i))+x_m(c));
      ВЫЧИСЛЕНО (c):=1
    КОНЕЦ
  НАЧАЛО
  ДЛЯ c:=1 ДО N ВЫП
  ЕСЛИ H_0(c) ТО ВЫЧИСЛЕНО (c):=1; y(c):=0
  ИНАЧЕ ВЫЧИСЛЕНО (c):=0 КЕ
  КЦ;
  ДЛЯ c:=1 ДО N ВЫПОЛНИТЬ
    ЕСЛИ H_1(c) И НЕ ВЫЧИСЛЕНО (c) ТО ВЫЧИСЛИТЬ (c) КЕ
  КЦ
КОНЕЦ.
```

Пусть теперь $C = Z^m$ – целочисленная решетка размерности m. В качестве сдвигов используются обычно аффинные линейные преобразования пространства Z^m, рассматриваемого как модуль над кольцом целых чисел, а в качестве допустимых множеств – множества, определенные

системами линейных неравенств и их булевы комбинации. На практике используются обычно достаточно простые наборы сдвигов и множеств, для которых геометрия движения фронта волны вычислений достаточно ясна, и определение порядка обхода точек области T не представляет труда.

Пример 2. Пусть H есть прямоугольная область $(a:b,\ c:d)$ на двумерной решетке Z^2 (рис. 5.7). Функция $y = f(x)$, где x и y – структуры данных, расположенные на Z^2, определяется следующими соотношениями:

$$y_{ij} = \varphi_1(y_{i+1\,j}, z_{ij}), \quad a \leqslant i \leqslant b-1, \quad c \leqslant j \leqslant d,$$
$$z_{ij} = \varphi_2(z_{i-1\,j}, v_{ij}), \quad a+1 \leqslant i \leqslant b, \quad c \leqslant j \leqslant d,$$
$$v_{ij} = \varphi_3(v_{ij+1}, u_{ij}), \quad a \leqslant i \leqslant b, \quad c \leqslant j \leqslant d-1,$$
$$u_{ij} = \varphi_4(u_{ij-1}, x_{ij}), \quad a \leqslant i \leqslant b, \quad c \leqslant j \leqslant d,$$
$$y_{ij} = \psi_1(z_{ij}), \quad i = b, \quad c \leqslant j \leqslant d,$$
$$z_{ij} = \psi_2(v_{ij}), \quad i = a, \quad c \leqslant j \leqslant d,$$
$$v_{ij} = \psi_3(u_{ij}), \quad a \leqslant i \leqslant b, \quad j = d,$$
$$u_{ij} = \psi_4(x_{ij}), \quad a \leqslant i \leqslant b, \quad j = c.$$

Обозначения с индексами использованы для того, чтобы подчеркнуть обычную форму представления таких соотношений. На рис. 6.1 стрелками показаны направления движения фронтов волны вычислений

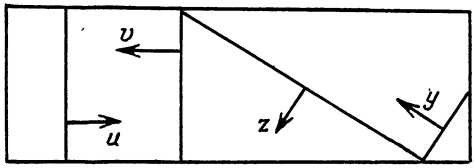


Рис. 6.1

для каждой из вспомогательных переменных. Вычисления по синхронной схеме происходят в три этапа. На первом этапе вычисляется только переменная u. Ее фронт – вертикальная линия – движется слева направо. После того как фронт вычисления u достигнет правого края, начинают двигаться фронт волны вычисления v и одновременно с ним фронт волны вычисления z. Фронт переменной z представляет собой наклонную линию, которая движется в направлении, показанном на рисунке. Как только нижний край этой линии достигнет правого нижнего угла, начинает движение фронт вычисления переменной y.

При последовательном вычислении функции $f(x)$ естественно весь процесс разделить на следующие три этапа. Первый этап – вычисление u, второй – вычисление v и z и третий – вычисление y. При вычислении u массив H можно проходить по строкам или столбцам, но так, чтобы столбцы проходились по возрастанию номеров, строки же могут проходиться либо по возрастанию, либо по убыванию номеров. И в том и в другом случае получаем лексикографическое упорядочение точек массива H. Для того чтобы переменные v и z могли вычисляться одновременно, необходимо H проходить так, чтобы номера строк возрастали, а номера столбцов убывали. Переменную y можно вычислять произвольно, но так, чтобы номера строк

изменялись по убыванию. Одна из реализаций, удовлетворяющих сформулированным условиям, имеет следующий вид:

НАЧАЛО
ДЛЯ $i := a$ ДО b ВЫПОЛНИТЬ
ДЛЯ $j := c$ ДО d ВЫПОЛНИТЬ
$u(i, j) :=$ ЕСЛИ $j = c$ ТО $\psi_4(x(i, j))$ ИНАЧЕ $\varphi_4(u(i, j-1),\ x(i, j))$;
КЦ КЦ;
ДЛЯ $i := a$ ДО b ВЫПОЛНИТЬ
ДЛЯ $j := d$ ШАГ -1 ДО c ВЫПОЛНИТЬ
$v(i, j) :=$ ЕСЛИ $j = d$ ТО $\psi_3(u(i, j))$ ИНАЧЕ $\varphi_3(v(i, j+1),\ u(i, j))$;
$z(i, j) :=$ ЕСЛИ $i = a$ ТО $\psi_2(v(i, j))$ ИНАЧЕ $\varphi_2((i-1, j),\ v(i, j))$;
КЦ КЦ;
ДЛЯ $i := b$ ШАГ -1 ДО a ВЫПОЛНИТЬ
ДЛЯ $j := c$ ДО d ВЫПОЛНИТЬ
$y(i, j) :=$ ЕСЛИ $i = b$ ТО $\psi_1(z(i, j))$ ИНАЧЕ $\varphi_1((i+1, j),\ z(i, j))$;
КЦ КЦ
КОНЕЦ.

Поскольку переменные u, v и z являются вспомогательными и не нужны для окончательного результата, в случае однотипных значений можно сэкономить память, используя для запоминания массивов u, z и y один и тот же массив y, а для массива v запоминать только последний вычисленный элемент. Тогда в первом цикле присваивание преобразуется в

$$y(i, j) := \text{ЕСЛИ } j = c \text{ ТО } \psi_4(x(i, j)) \text{ ИНАЧЕ } \varphi_4(y(i, j-1),\ x(i, j)),$$

а во втором цикле присваивания следует заменить на:

$$v := \text{ЕСЛИ } j = d \text{ ТО } \psi_3(y(i, j)) \text{ ИНАЧЕ } \varphi_3(v, u(i, j));$$

$$y(i, j) := \text{ЕСЛИ } i = a \text{ ТО } \psi_2(v) \text{ ИНАЧЕ } \varphi_2(y(i-1, j), v).$$

Общий метод построения программы для вычисления периодически определенных функций, заданных системой соотношений (3.1), при условии конечности множества T и бесцикловости графа информационной зависимости состоит в следующем.

Пусть M – множество скалярных переменных. Областью расположения множества M называется наименьшее множество H точек такое, что значения всех переменных из M берутся только в точках множества H. Множество M скалярных переменных, расположенных на H, называется однородным, если из $y_t(c) \in M$ следует, что $H \subset H_{tj}$ для некоторого j.

Пусть M – однородное множество, расположенное на H, а Y – множество структурных переменных, из которых образованы скалярные переменные множества M. Определим $\xrightarrow{M}$ на H, полагая $c \xrightarrow{M} c' \iff$ существуют переменные $z, z' \in Y$ такие, что $z(c) \xrightarrow{R} z'(c')$. Транзитивно-реф-

лексивное замыкание отношения $\xrightarrow{M}$ обозначим через $\xrightarrow{M}{}^{*}$. Упорядочим также переменные множества Y, полагая $z \xrightarrow{M} z' \Leftrightarrow$ для некоторого $c \in H$ $z(c) \xrightarrow{R} z'(c)$. В силу однородности, если $z \xrightarrow{M} z'$, то $z(c) \xrightarrow{R} z'(c)$ для любого $c \in H$. Граф отношения $z \xrightarrow{M} z'$, очевидно, не имеет циклов, а его транзитивно-рефлексивное замыкание есть частичный порядок. Отношение $c \xrightarrow{M}{}^{*} c'$, вообще говоря, не является отношением частичного порядка. Но если H частично упорядочено этим отношением, то значения всех переменных из M можно вычислить за один проход по множеству H, доопределив частичные порядки на H и на Y до линейных.

Таким образом, задача свелась к тому, чтобы разложить множество T на непересекающиеся подмножества $T = T_1 \cup \ldots \cup T_l$ так, чтобы отношение $c \xrightarrow{T_t}{}^{*} c'$ на каждом из этих множеств было бы частичным порядком. Такое разложение, конечно, всегда существует, например, разложение на одноэлементные множества, но нас интересует случай, когда l невелико (по сравнению с общим объемом множества T), а классы разложения получаются комбинированием множеств переменных и пересечений областей однородности H_{ij}. Если указанное разложение найдено, то программу можно построить в виде последовательной композиции программ обработки каждого из классов разложения. Пусть, например, если T_t расположен на H, а множество структурных переменных, соответствующих этому классу, есть Y. Пусть порядок на Y доопределен до линейного порядка и $(z_1, \ldots, z_k)$ – все элементы Y, расположенные в этом порядке. Доопределим отношение $c \xrightarrow{M}{}^{*} c'$ на H до линейного порядка. Тогда программа обработки класса T_t может быть представлена в виде цикла

ДЛЯ ВСЕХ $c \in H$ ВЫПОЛНИТЬ

$z_1(c) := t_1;$

. . . . . .

$z_k(c) := t_k$

КЦ

в котором элементы множества H перебираются в соответствии с установленным порядком, а выражения $t_1, \ldots, t_k$ определяются по соотношениям (3.1).

Вопрос о том, является ли граф отношения $c \xrightarrow{M} c'$ на множестве H бесцикловым и как его линеаризировать, в общем случае непростой и сводится к чисто алгебраической задаче о том, существует ли в заданной полугруппе линейных аффинных преобразований модуля элемент, имеющий неподвижную точку в заданной области. Действительно, пусть $g_1, \ldots, g_n$ – все сдвиги, которые участвуют в зависимостях вида $z(cg_t) \xrightarrow{M} z'(c)$ на множестве M. В силу однородности множество $\{g_1, \ldots, g_n\}$ не зависит от выбора точки c. Рассмотрим полугруппу G, порожденную элементами $g_1, \ldots, g_n$. Наличие цикла означает равенство $ch_1 \ldots h_k = c$ для некоторой точки $c \in H$, где $h_1, \ldots, h_k \in \{g_1, \ldots, g_n\}$ и $ch_1 \ldots h_j \in H$ $(j = 1, \ldots, k-1)$. При этом c является неподвижной точкой преобразования $h = h_1 \ldots h_k \in G$.

Рассмотрим случай, когда g_i – чисто аффинные преобразования (движения), т.е. $cg_i = c + a_i$, a_i, $c \in Z^m$ $(i = 1, \ldots, n)$. Элементы полугруппы G в этом случае можно отождествить с линейными комбинациями $g = \sum_{i=1}^{n} \lambda_i a_i$, в которых λ_i – целые коэффициенты, $\lambda_i \geqslant 0$, $\sum_{i=1}^{n} \lambda_i > 0$. Поскольку $c + a = c \Leftrightarrow a = 0$, дело сводится к распознаванию того, содержит ли полугруппа G нулевой элемент (поскольку G коммутативна, операцию записываем аддитивно) и выразим ли этот элемент в виде суммы $h_1 + \ldots + h_k$ так, что для некоторого $c \in H$ все элементы $c + h_1 + \ldots + h_j$ также принадлежат H $(j = 1, \ldots, k,\ h_1, \ldots, h_k \in \{a_1, \ldots, a_n\})$. Если размеры множества H достаточно велики вдоль каждой из координатных осей множества Z^m, то последнее условие можно отбросить, и задача сведется к тому, имеет ли уравнение $\sum_{i=1}^{n} \lambda_i a_i = 0$ ненулевое решение в целых неотрицательных числах (если принять во внимание, что $a_i = (a_{i1}, \ldots, a_{im})$, то речь идет о решении системы уравнений с целочисленными коэффициентами).

Таким образом, отсутствие целых неотрицательных решений уравнения $\sum_{i=1}^{n} \lambda_i a_i = 0$ является достаточным условием бесцикловости графа, порожденного отношением $c \overset{M}{\rightarrow} c'$ (в случае больших размеров H это условие также необходимо). Предположим, что указанное условие выполняется. Тогда полугруппа G содержится в множестве целочисленных точек некоторого выпуклого конуса $\bar{G}$ (Z^m предполагается вложенным в $\mathbf{R}^m$), натянутого на векторы с целочисленными координатами, и лежит по одну сторону от некоторой гиперплоскости H_1, проходящей через начало координат и также натянутой на целочисленные векторы. Плоскость H_1 назовем опорной гиперплоскостью конуса $\bar{G}$. Ее можно принять в качестве фронта волны вычислений, общего для всех переменных множества Y. Направление перемещения плоскости H_1 определяется вектором h с целочисленными координатами, который выбирается из следующих соображений:

1) $H_1 + \lambda h \cap G = \phi$ для любого $\lambda > 0$, т.е. при перемещении H_1 из начала координат на λh G остается по ту же сторону от плоскости H_1, что и до перемещения;

2) между плоскостями H_1 и $H_1 + h$ нет точек с целочисленными координатами, т.е. h перемещает H_1 на минимально возможное расстояние, обеспечивающее прохождение через все промежуточные точки решетки Z^m.

Начальное положение фронта волны вычислений определяется вектором a_0, который выбирается, исходя из следующих соображений:

1) множества $G + a_0$ и H лежат по разные стороны от плоскости $H_1 + a_0$;

2) $H_1 + a_0 - h \cap H = \phi$, $H_1 + a \cap H \neq \phi$, т.е. $H_1 + a_0$ расположено "на краю" множества H.

Если $G \cap H_1 = \phi$, то вычисления в точках фронта волны вычислений можно выполнять в любом порядке. Пусть $Y = \{z_1, \ldots, z_k\}$, причем указанный порядок согласован с отношением информационной зависимости. Тогда программа вычисления $z_1, \ldots, z_k$ на области H имеет следующий вид:

ДЛЯ $a := a_0$ ШАГ h ПОКА $(H_1 + a) \cap H \neq \phi$ ВЫПОЛНЯТЬ
ДЛЯ ВСЕХ $c \in H_1 + a \cap H$ ВЫПОЛНЯТЬ
$z_1(c) := t_1$;
. . . . . .
$z_k(c) := t_k$;
КЦ КЦ.

Здесь, конечно, предполагается, что область H односвязная. Если $G \cap H_1 \neq \phi$ (что не исключается), то обход точек области должен выполняться в соответствии с информационными зависимостями. Для того чтобы определить порядок обхода точек области H_1, следует изучить пересечение $G \cap H_1$ и решить задачу, аналогичную уже решенной, но для меньшей размерности. Распознавание условий пустоты пересечения в заголовке первого цикла, а также определения областей изменения параметров внутреннего цикла зависит от того, как задано множество H. Если H определено с помощью линейных неравенств, то задачи снова сводятся к линейной алгебре в модулях. При рассмотрении этих задач во всей общности снова возникают определенные трудности, но на практике имеют дело с простыми частными случаями, которые анализируются достаточно легко.

Пример 3. Пусть нужно вычислить значение $y(i, j)$ в прямоугольнике $H = (0 : m, 0 : n)$ с помощью соотношения

$$y(i, j) = F(y(i - 1, j + 3),$$

$$y(i + 1, j - 4)),$$

$$0 \leqslant i \leqslant m, \ 0 \leqslant j \leqslant n.$$

Предполагается, что вне прямоугольника структура y задана. Полугруппа G порождается векторами $(-1, 3)$ и $(1, -4)$. В качестве опорной гиперплоскости H_1 можно взять прямую $3i - j = 0$ или $4i + j = 0$. В первом случае H_1 проходит через точку $(-1, 3)$, во втором случае – через $(1, -4)$. Двигаться по прямой H_1 в любом случае следует в сторону возрастания индексов. В качестве шага сдвига прямой H_1 следует взять $h = (0, 1)$. В качестве a_0 следует взять точку $(m, 0)$. Выберем в качестве H_1 прямую $3i - j = 0$. Удобно представить ее в параметрической форме: $i = \lambda$, $j = 3\lambda$ $(\lambda \in Z)$. Тогда $H_1 + a = H_1 + (a_1, a_2)$ имеет параметрическое представление $i = \lambda + a_1$, $j = 3\lambda + a_2$. Переходя к новому параметру $\mu = \lambda + a_1$ и принимая во внимание, что $a_1 = m$, получим $i = \mu$, $j = 3\mu - 3m + a_2$. Пересечение $(H_1 + a) \cap H$ определяется неравенствами

$$0 \leqslant \mu \leqslant m,$$

$$0 \leqslant 3\mu - 3m + a_2 \leqslant n.$$

Это пересечение непусто тогда и только тогда, когда

$$0 \leqslant a_2 \leqslant 3m + n.$$

Внутри пересечения параметр μ изменяется в пределах

$$\max\left(0, \frac{3m - a_2}{3}\right) \leqslant \mu \leqslant \min\left(n, \frac{3m + n - a_2}{3}\right).$$

Окончательно получаем программу

```
ДЛЯ a2 := 0 ДО 3m + n ВЫПОЛНИТЬ
  ЕСЛИ 3m – a2 < 0 ТО μ0 := 0
  ИНАЧЕ μ0 := (3m – a2) ÷ 3 + 1 КЕ;
  ЕСЛИ n ⩽ (3m + n – a2)/3 ТО μ1 := n
  ИНАЧЕ μ1 := (3m + n – a2) ÷ 3 КЕ;
ДЛЯ i := μ0 ДО μ1 ВЫПОЛНИТЬ
  j := 3i – 3m + a2;
  y (i, j) := F (y (i – 1, j + 3), y (i + 1, j – 4))
КЦ.
КЦ.
```

Упражнения

1. Доказать, что если для представления одного элемента алгебры данных требуется ограниченный объем памяти, а для выполнения базовой операции – ограниченное время, то для вычисления периодически определенной функции с помощью программы F_1 требуются линейное время и память в зависимости от числа элементов области T.

2. Уточнить до конца метод построения программы F вычисления периодически определенной функции для случая, когда сдвигами служат движения, областью расположения структур данных – двумерная решетка Z^2, а областями однородности – прямоугольники.

3. Доказать корректность программы F_1.

4. Преобразовать программу примера 3 так, чтобы в ней не применялась операция целочисленного деления. Оптимизировать по числу операций бесцикловые части этой программы.

§ 4. Теоретико-множественное программирование

В этом параграфе будут рассмотрены несколько примеров разработки программ с использованием теоретико-множественных структур данных. Примеры эти носят достаточно общий характер, и полученные при их рассмотрении результаты могут быть использованы при решении многих конкретных задач.

Построение компонент связности неориентированного графа. Неориентированный граф (G, ρ) задается множеством G вершин и симметричным бинарным отношением $\rho \subset G^2$ на этом множестве. Множество G можно считать компонентой базовой алгебры данных, а отношение ρ – базовым отношением. Рефлексивно-транзитивное замыкание ρ^* отношения ρ дает отношение достижимости. Отношение ρ^* есть отношение эквивалентности, классы которого называются компонентами связности графа G. Компонента связности, которой принадлежит элемент g, обозначается через $\rho^*(g)$. Задача состоит в том, чтобы найти число компонент связности графа G. Следующая программа решает эту задачу путем последовательного построения компонент связности и их подсчета с помощью функции φ: $G \to \{0, 1, \ldots\}$, которая каждому элементу $g \in G$ сопоставляет номер компоненты, которой этот элемент принадлежит.

```
ПРОГРАММА СВЯЗНОСТЬ (G, ρ) РЕЗУЛЬТАТ (k);
НАЧАЛО
  k: = 0;
  ДЛЯ ВСЕХ g ∈ G ВЫПОЛНИТЬ φ(g): = 0 КЦ.
  ЦИКЛ.
    НАЙТИ g ∈ G ТАКОЙ, ЧТО φ(g) = 0;
    ЕСЛИ НЕТ, ТО ВЫЙТИ;
    k: = k + 1;
    ДЛЯ ВСЕХ h ∈ ρ*(g) ВЫПОЛНИТЬ φ(h): = k КЦ
  КЦ.
КОНЕЦ.
```

В этой программе так же, как и в других, описания типов переменных и формальных параметров, если они ясны из контекста, опускаются. В программе использованы следующие теоретико-множественные конструкции:

1. Алгебра отношений с операцией рефлексивно-транзитивного замыкания.

2. Множество всех подмножеств множества G с операцией $\tau(g)$, которая отношению τ и элементу g ставит в соответствие множество всех $h \in G$ таких, что $(g, h) \in \tau$.

3. Функциональные структуры данных, расположенные на G и принимающие значения в множестве целых чисел.

Начальное условие программы СВЯЗНОСТЬ состоит в том, что ρ есть симметричное отношение на множестве G, а заключительное имеет вид: k есть число компонент связности графа (G, ρ). Корректность программы доказывается с помощью следующего инварианта основного цикла: для всех $l = 1, \ldots, k$ $\varphi^{-1}(l)$ есть компонента связности графа G и $\varphi^{-1}(l) \cap \cap \varphi^{-1}(0) = \phi$; для всех $g \in G$ $0 \leqslant \varphi(g) \leqslant k$. Обозначим это условие через $\alpha(k)$.

Перед первым прохождением цикла условие будет верным, поскольку для всех $g \in G$ в этот момент времени $\varphi(g) = 0$ и $k = 0$. Сохранение инварианта при следующих прохождениях цикла обеспечивается определением функции $\rho^*(g)$.

При выходе из цикла $\varphi^{-1}(0) = \phi$ и, следовательно, k есть число компонент связности.

Следующий шаг проектирования программы требует принятия решения относительно представления данных. Пусть граф G состоит из n вершин, а степени вершин не превышают m. Тогда его вершины можно занумеровать числами $1, \ldots, n$ $(G = \{g_1, \ldots, g_n\})$, а отношение ρ представить с помощью целочисленного массива R $(1 : n, 1 : m)$ так, чтобы выполнялось следующее условие: множество ненулевых элементов i-й строки массива R совпадает с множеством номеров вершин, инцидентных вершине с номером i.

Функция φ реализуется тогда целочисленным массивом F $(1 : n)$ так, что $\varphi(g_i) = F(i)$ $(i = 1, \ldots, n)$. Программа СВЯЗНОСТЬ имеет вид

```
НАЧАЛО
  k: = 0; Q1;
```

```
    ЦИКЛ
      Q2; ЕСЛИ НЕТ, ТО ВЫЙТИ КЕ;
      k: = k + 1; Q3;
    КЦ;
  КОНЕЦ.
```

Для того чтобы реализовать эту программу, достаточно реализовать операторы Q_1, Q_2 и Q_3, используя принятые решения о представлении данных. Реализация операторов Q_1 и Q_2 трудностей не вызывает. Рассмотрим реализацию оператора Q_3. Предположим, что s есть номер элемента g_s такого, что $\varphi(g_s) = 0$. Этот номер найден оператором Q_2. Начальное и заключительное условия, обеспечивающие корректность оператора Q_3, можно определить как $\alpha(k-1)$ и $\alpha(k)$ соответственно. Этим условиям удовлетворяет следующий оператор:

```
НАЧАЛО F(s): = k;
  ЦИКЛ
    ИЗМЕНЕНИЕ: = 0;
    ДЛЯ i := 1 ДО n ВЫПОЛНИТЬ
    ЕСЛИ F(i) = k ТО
      ДЛЯ j : = 1 ДО m ВЫПОЛНИТЬ
        ЕСЛИ R (i, j) ≠ 0 И F (R (i, j)) ≠ k ТО
          F (R (i, j)) : = k, ИЗМЕНЕНИЕ: = 1;
        КЕ;
      КЦ;
    КЕ;
  КЦ;
    ЕСЛИ ИЗМЕНЕНИЕ, ТО ПОВТОРИТЬ КЕ
  КЦ.
КОНЕЦ.
```

Этот оператор основан на следующем простом соображении.

Компонента связности $\rho^*(g)$ есть наименьшее множество, которое содержит g и вместе с каждой вершиной h содержит все инцидентные с ней вершины. Поэтому $\rho^*(g)$ можно строить так. Отнесем к этой компоненте элемент g, отметив его числом k. Затем будем искать такие h, которые отмечены числом k и имеют инцидентную вершину h', которая не отмечена этим числом. Если такое h' найдено, отметим его числом k, т.е. отнесем к классу $\rho^*(g)$. Поиск следует повторять до тех пор, пока вершин h, удовлетворяющих данному свойству, не будет. В теоретико-множественных структурах данных этот алгоритм имеет следующий вид:

```
НАЧАЛО
  φ (g): = k;
  ЦИКЛ
    НАЙТИ h ∈ G ТАКОЕ, ЧТО φ (h) = k И h' ∈ ρ (h)
    ТАКОЕ, ЧТО φ (h') ≠ k;
    ЕСЛИ НЕТ, ТО ВЫЙТИ КЕ;
      ДЛЯ ВСЕХ h'' ∈ ρ (h) ВЫПОЛНИТЬ
      φ (h'') : = k
    КЦ
```

КОНЕЦ ЦИКЛА
КОНЕЦ.

Рассмотренный выше оператор реализует данный алгоритм в функциональных структурах данных. Корректность Q_3 можно доказать с использованием следующего инварианта внешнего цикла: для всех $l = 1, \ldots, k-1$ $\varphi^{-1}(l)$ есть компонента связности, $\varphi^{-1}(l) \cap (\varphi^{-1}(0) \cup \cup \varphi^{-1}(k)) = \phi$ и $\varphi^{-1}(k)$ целиком лежит в некоторой компоненте связности.

Проверка инвариантности этого высказывания труда не представляет. Заключительное условие $\alpha(k)$ является следствием того, что выход из внешнего цикла происходит при условии, что во внутреннем цикле переменная ИЗМЕНЕНИЕ не получила значения, равного 1. А это возможно лишь в том случае, когда ни для каких i и j не выполняется условие $F(i) = k$ и $R(i, j) \neq 0$ и $F(R(i, j)) \neq k$ $(\varphi(g_i) = k, (g_i, h) \in \rho$ и $\varphi(h) \neq k)$, т.е. когда $\varphi^{-1}(k)$ есть компонента связности.

Окончательный вариант программы СВЯЗНОСТЬ в функциональных структурах данных имеет вид:

```
ПРОГРАММА СВЯЗНОСТЬ 1 (m, n : ЦЕЛЫЕ, R : МАССИВ (1 : n, 1 : m)
ЦЕЛЫХ) РЕЗУЛЬТАТ (k : ЦЕЛОЕ);
ИМЕНА F : МАССИВ (1 : n) ЦЕЛЫХ, НЕТ, ИЗМЕНЕНИЕ:
ЛОГИЧЕСКИЕ, s, l : ЦЕЛЫЕ.
НАЧАЛО
        k : = 0;
  ДЛЯ i : = 1 ДО n ВЫПОЛНИТЬ F(i): = 0 КЦ;
    ЦИКЛ. НЕТ : = 0;
      ДЛЯ i : = 1 ДО n ВЫПОЛНИТЬ
      ЕСЛИ F(i) = 0, ТО НЕТ : = 1, s : = i; ВЫЙТИ КЕ;
    КЦ;
      ЕСЛИ НЕТ, ТО ВЫЙТИ;
      k : = k + 1; F(s): = k;
    ЦИКЛ. ИЗМЕНЕНИЕ : = 0;
      ДЛЯ i : = 1 ДО n ВЫПОЛНИТЬ
        ЕСЛИ F(i) ≠ k ТО ПОВТОРИТЬ КЕ;
        ДЛЯ j : = 1 ДО m ВЫПОЛНИТЬ
          l : = R(i, j);
          ЕСЛИ l ≠ 0 И F(l) ≠ k, ТО F(l): = k;
            ИЗМЕНЕНИЕ : = 1;
          КЕ;
        КЦ;
      КЦ;
      ЕСЛИ ИЗМЕНЕНИЕ, ТО ПОВТОРИТЬ КЕ;
    КЦ
  КЦ;
КОНЕЦ.
```

Анализируя время выполнения программы, которое определяется максимальным числом прохождения оператора, который находится внутри

наиболее глубоко вложенных друг в друга циклов, получим оценку $O(mn^2)$. Действительно, количество повторений самого внешнего цикла равно числу компонент связности. Следующий по цепочке вложенности цикл повторяется не более, чем число элементов в текущей компоненте. Поэтому тело этого цикла повторяется не более, чем n раз. Это тело, в свою очередь, состоит из двух вложенных циклов, которые повторяются n и m раз соответственно. Но поскольку внутренний цикл может обходиться, произведение mn дает лишь верхнюю оценку времени выполнения обоих циклов.

При небольших величинах m и n время может быть вполне приемлемым. Однако при больших n имеет смысл попытаться улучшить оценку. Ближайшая цель состоит в том, чтобы получить такое улучшение путем формальных преобразований программы СВЯЗНОСТЬ 1. Для этой цели удобно перейти к теоретико-множественному представлению этой программы с сохранением функциональных структур данных (R и F). Перепишем тело программы следующим образом:

```
НАЧАЛО
  k: = 0;
  ДЛЯ i: = 1 ДО n ВЫПОЛНИТЬ F(i): = 0 КЦ;
  ЦИКЛ.
    НАЙТИ s ∈ [1: n] ТАКОЕ, ЧТО F(s) = 0;
    ЕСЛИ НЕТ, ТО ВЫЙТИ КЕ;
    k: = k + 1, F(s): = k;
    ЦИКЛ. ИЗМЕНЕНИЕ: = 0;
      ДЛЯ ВСЕХ i ∈ [1: n] ТАКИХ, ЧТО F(i) = k ВЫПОЛНИТЬ
        ДЛЯ ВСЕХ j ∈ [1: m] ТАКИХ, ЧТО R(i,j) ≠ 0 И
          F(R(i,j)) ≠ k ВЫПОЛНИТЬ
          F(R(i,j)): = k, ИЗМЕНЕНИЕ: = 1
      КЦ; КЦ;
      ЕСЛИ ИЗМЕНЕНИЕ, ТО ПОВТОРИТЬ КЕ
    КЦ;
  КЦ;
КОНЕЦ.
```

Ясно, что полученная программа является моделью программы СВЯЗНОСТЬ 1. Выделение в заголовках циклов в явном виде условий, при которых внутри циклов вообще выполняются какие-либо действия, показывает путь сокращения времени выполнения программы. Он состоит в том, чтобы ввести вспомогательные структуры данных – множества (в данном случае числовые), которые позволили бы сократить пространство поиска или перебора значений в циклах.

Для первого цикла (поиск s) можно ввести множество $V = \{ i \in (1: n) \mid F(i) = 0\}$. Тогда вместо цикла по s можно использовать оператор

ЕСЛИ $V = \phi$ ТО ВЫЙТИ ИНАЧЕ ВЗЯТЬ s ИЗ V КЕ.

Семантику оператора ВЗЯТЬ следует при этом определить так, что элемент, который берется из множества, удаляется из него. Для того чтобы перед каждым повторением внешнего цикла множество V обладало нужным свойством, т.е. совпадало с множеством всех i, на которых $F(i) = 0$, в программу следует ввести операторы удаления элементов из V всякий раз, когда на этих элементах функции F присваивается значение, отличное от нуля.

Для второго цикла, вложенного во внешний цикл, который строит компоненту связности $\rho^*(g_s)$, хорошо иметь множество W такое, что при каждом повторении этого цикла все $i \in (1: n)$ такие, что $F(i) = k$ и для которых существуют $j \in (1: m)$ такие, что $F(R(i,j)) \neq k$, находятся внутри W. Если указанное условие обеспечено, то цикл построения компоненты связности можно заменить циклом вида

```
ПОКА W ≠ φ ВЫПОЛНИТЬ
  ВЗЯТЬ i ИЗ W;
    ...
КЦ.
```

Перед выполнением этого цикла $W = \{s\}$. Для того чтобы сохранялось основное свойство множества W, достаточно добавлять к этому множеству новый элемент всякий раз, когда функция F на этом элементе становится равной k. Преобразованная программа имеет следующий вид:

```
НАЧАЛО
  k: = 0;
  ДЛЯ i: = 1 ДО n ВЫПОЛНИТЬ F(i): = 0 КЦ;
    V: = {1, ..., n}.
  ПОКА V ≠ φ ВЫПОЛНЯТЬ
    ВЗЯТЬ s ИЗ V;
    k: = k +1; F(s): = k; W: = {s};
    ПОКА W ≠ φ ВЫПОЛНЯТЬ
      ВЗЯТЬ i ИЗ W;
      ДЛЯ ВСЕХ j ∈ (1: m) ТАКИХ, ЧТО R(i,j) ≠ 0 И
        F(R(i,j)) ≠ k ВЫПОЛНИТЬ
        F(R(i,j)): = k;
      УДАЛИТЬ R(i,j) ИЗ V;
      ДОБАВИТЬ R(i,j) К W;
    КЦ;
  КЦ
  КЦ КОНЕЦ.
```

Если новые структуры данных реализовать таким образом, чтобы операторы ВЗЯТЬ, УДАЛИТЬ и ДОБАВИТЬ, а также проверка на пустоту

выполнялись за ограниченное время, то время выполнения программы сократится до $O(mn)$. При малом m оценка снова удовлетворительна. Если m имеет такой же порядок, что и n, можно еще снизить оценку до $O(m + n)$, вводя подходящие структуры данных для самого внутреннего цикла.

Рассмотрим реализацию структур данных V и W. Их естественно задавать списками. При этом, поскольку к W применяются только операторы ВЗЯТЬ и ДОБАВИТЬ, достаточно одностороннего списка. Что же касается V, то здесь следует использовать двухсторонний список, поскольку применяется оператор УДАЛИТЬ. Кроме того, поскольку может потребоваться удалять любой элемент этого списка, желательно иметь к нему прямой доступ, т.е. по номеру элемента прямо обращаться к тому месту в списке, где расположен удаляемый элемент. Указанным требованиям удовлетворяет структура данных, представленная в виде целочисленного массива M: МАССИВ (1: n, 1: 2) ЦЕЛЫХ, столбцы которого представляют прямые и обратные связки, вместе с переменной v, указывающей на начало списка. Если множество M состоит из элементов i_1, ..., i_k, то массив M должен удовлетворять условию (для некоторого порядка расположения элементов): $v = i_1$, $M(i_1, 1) = i_2$, $M(i_2, 1) = i_3$, ..., $M(i_{k-1}, 1) = i_k$, $M(i_k, 1) = 0$, $M(i_k, 2) = i_{k-1}$, $M(i_{k-1}, 2) = i_{k-2}$, ..., $M(i_2, 2) = i_1$, $M(i_1, 2) = 0$. Если множество V пусто, полагаем $v = 0$. Поскольку множества V и W никогда не пересекаются, то W можно представлять в том же самом массиве M, что и V, но для связывания его элементов использовать только прямые связи. В качестве указателя начального элемента используется переменная w. Реализация операторов ВЗЯТЬ, УДАЛИТЬ и ДОБАВИТЬ имеет вид

```
ВЗЯТЬ s ИЗ V:
НАЧАЛО
  s: = v;
  УДАЛИТЬ s ИЗ V
КОНЕЦ.

УДАЛИТЬ s ИЗ V:
  ЕСЛИ M(s, 1) = 0 ТО v: = 0 ИНАЧЕ
  ЕСЛИ M(s, 2) = 0 ТО
    v: = M(s, 1), M(s, 2): = 0
  ИНАЧЕ M(M(s, 2), 1): = M(s, 1);
        M(M(s, 1), 2): = M(s, 2)
  КОНЕЦ ЕСЛИ.
ВЗЯТЬ s ИЗ W:
  (s: = w, w: = M(w, 1))
ДОБАВИТЬ s К W:
  ЕСЛИ w = 0 ТО w: = s, M(s, 1): = 0
  ИНАЧЕ M(s, 1): = w, w: = s КЕ.
```

Несмотря на простоту реализации, требуется тщательная проверка правильности построенных операторов. Главное здесь — точно определить условия корректности, которые извлекаются из формальных требований к операторам ВЗЯТЬ, УДАЛИТЬ и ДОБАВИТЬ, а также из определения реализации структур данных V и W. Следует обратить внимание на то, что множества V и W представляются не всеми, а только некоторыми, допустимыми состояниями памяти, определяющими возможные значения массива M. Значение массива M допустимо, по определению, тогда и только тогда, когда существуют последовательности $(i_1, \ldots, i_k)$ и $(j_1, \ldots, j_l)$, составленные из различных положительных целых чисел, не превосходящих n, и такие, что:

1) если $v \neq 0$, то

$$v = i_1,\ M(i_1, 1) = i_2, \ldots,\ M(i_k, 1) = 0,$$
$$M(i_1, 2) = 0, \ldots,\ M(i_k, 2) = i_{k-1},\ k \geqslant 1;$$

2) если $w \neq 0$, то

$$w = j_1,\ M(j_1, 2) = j_2, \ldots,\ M(j_l, 1) = 0.$$

Если состояние памяти таково, что значение M допустимо, то оно представляет множества V и W такие, что если $v = 0$, то $V = \phi$, иначе $V = \{i_1, \ldots, i_k\}$, если $w = 0$, то $W = \phi$, иначе $W = \{j_1, \ldots, j_l\}$. Условие допустимости значения M обозначим через $u(v, w, M)$, а множества $\{i_1, \ldots, i_k\}$ и $\{j_1, \ldots, j_l\}$ — через $M_1(v, M)$ и $M_2(w, M)$ соответственно.

Пусть $\alpha(V, M)$ обозначает условие $u(v, w, M) \wedge M_1(v, M) = V \wedge M_2(w, M) = W$. Тогда условия корректности операторов ВЗЯТЬ, УДАЛИТЬ, ДОБАВИТЬ записываются следующим образом:

$$\alpha(V, W) \wedge V \neq \phi \wedge v = 1 \Rightarrow \langle \text{ВЗЯТЬ } s \text{ ИЗ } V \rangle\ s = i \wedge \alpha(V \setminus \{i\}, W);$$

$$\alpha(V, W) \wedge s = i \wedge i \in V \Rightarrow \langle \text{УДАЛИТЬ } s \text{ ИЗ } V \rangle\ \alpha(V \setminus \{i\}, W);$$

$$\alpha(V, W) \wedge w = j \Rightarrow \langle \text{ВЗЯТЬ } s \text{ ИЗ } W \rangle\ s = j \wedge \alpha(V, W \setminus \{j\});$$

$$\alpha(V, W) \wedge s = i \Rightarrow \langle \text{ДОБАВИТЬ } s \text{ К } W \rangle\ \alpha(V, W \cup \{i\}).$$

Доказательства легко получаются из определений. Определения операторов ВЗЯТЬ, УДАЛИТЬ и ДОБАВИТЬ подставляются в программу (как макроопределения). При подстановке оператора УДАЛИТЬ s ИЗ V в оператор ВЗЯТЬ s ИЗ V можно сделать некоторые упрощения. Действительно, в операторе ВЗЯТЬ имеет место $s = v$, поэтому $M(s, 2) = 0$, и оператор УДАЛИТЬ можно трансформировать в

ЕСЛИ $M(s, 1) = 0$ ТО $v := 0$ ИНАЧЕ $v := M(s, 1), M(s, 2) := 0$ КЕ.

Порождение множеств с помощью рекурсивных определений. Пусть заданы множества S, X, $P \subset X$ и отношение $R \subset X^2 \times S$. Определим множество $Q \subset X$ как наименьшее множество, удовлетворяющее следующим двум условиям:

1) $P \subset Q$;

2) для любых $x \in Q$, $s \in S$ и $x' \in X$, если $R(x', x, s)$, то $x' \in Q$.

Задача состоит в том, чтобы построить множество Q при условии, что множества P, X и S конечны. Программа, которая выполняет это построение, работает по шагам. На каждом шаге порождается некоторое число

новых элементов множества Q. Для этого просматриваются элементы $x \in Q_0$, порожденные на предыдущем шаге, и все элементы x' множества X. Если для пары (x', x) найдется $s \in S$ такой, что $R(x', x, s)$, то x' добавляется к элементам множества Q_1, порожденным на данном шаге, если только он не был получен ранее.

```
ПРОГРАММА ПОРОЖДЕНИЕ (P, S, X, R) РЕЗУЛЬТАТ (Q);
ИМЕНА Q0, Q1 : ПОДМНОЖЕСТВА МНОЖЕСТВА X; x, x' ∈ X;
НАЧАЛО
  Q := φ, Q0 := P;
  ЦИКЛ
    Q1 := φ;
    ДЛЯ ВСЕХ x ∈ Q0 ВЫПОЛНИТЬ
    ДЛЯ ВСЕХ x' ∈ X ТАКИХ, ЧТО ДЛЯ НЕКОТОРОГО
      s ∈ S, R(x', x, s) ВЫПОЛНИТЬ
    ЕСЛИ x' ∉ Q ∪ Q0 ∪ Q1 ТО ДОБАВИТЬ x' К Q1 КЕ
    КЦ КЦ;
    Q := Q ∪ Q0;
    ЕСЛИ Q1 = φ ТО ВЫЙТИ КЕ;
    Q0 := Q1
  КЦ.
КОНЕЦ.
```

Обозначим искомое множество через $C(R, P, S, X)$. Доказательство корректности программы ПОРОЖДЕНИЕ выполняется с помощью следующих инвариантов основного цикла:

1. $P \subset Q$.
2. $x \in Q,\ s \in S,\ x' \in X \Rightarrow x' \in Q \cup Q_1$.
3. $Q \subset C(R, P, S, X)$.

После выхода из основного цикла $Q_1 = \phi$, и, следовательно, множество Q с требуемыми свойствами построено. Завершимость алгоритма вытекает из конечности множеств P, S и X и вследствие того, что при каждом прохождении основного цикла, кроме последнего, множество Q обязательно увеличивается.

Недостатком программы ПОРОЖДЕНИЕ является то, что в ней многократно рассматриваются одни и те же элементы множеств. Для сокращения перебора во внутреннем цикле можно построить функцию $G(x) = \{x' \mid \exists s \in S, R(x', x, s)\}$ и внутренний цикл выполнять только по элементам x' множества $G(x)$. Другие сокращения возможны с использованием информации о структуре множества X. Предположим, например, что $X = X_1 \times X_2$. Определим следующие структуры данных. Функции F, F_0, $F_1 : X_2 \to 2^{X_1}$ будем вычислять таким образом, чтобы выполнялись усло-

вия: $(x_1, x_2) \in Q \Leftrightarrow x_1 \in F(x_2), (x_1, x_2) \in Q_0 \Leftrightarrow x_1 \in F_0(x_2), (x_1, x_2) \in Q_1 \Leftrightarrow$ $\Leftrightarrow x_1 \in F_1(x_2)$. Введем также множества $B, B_0, B_1 \subset X_2$ и обеспечим выполнение в соответствующих местах программы условий $B = \{ x_2 \mid F(x_2) \neq \phi \}$, $B_0 = \{ x_2 \mid F_0(x_2) \neq \phi \}$, $B_1 = \{ x_2 \mid F_1(x_2) \neq \phi \}$. Теперь структуры данных Q, Q_0 и Q_1 становятся ненужными, а программу ПОРОЖДЕНИЕ можно переписать с использованием новых структур данных (описания структур данных определены в тексте и поэтому в программе отсутствуют):

```
ПРОГРАММА ПОРОЖДЕНИЕ 1 (P, S, X, R) РЕЗУЛЬТАТ (B, F);
НАЧАЛО
  B: = B0: = B1: = φ·
  ДЛЯ ВСЕХ x2 ∈ X2 ВЫПОЛНИТЬ
      F(x2): = F0(x2): = F1(x2): = φ;
  ДЛЯ ВСЕХ x1 ∈ X1 ВЫПОЛНИТЬ
      G(x1, x2): = {x' | ∃ s ∈ S, R(x', (x1, x2), s)};
    ЕСЛИ (x1, x2) ∈ P ТО ДОБАВИТЬ x1 К F0(x2);
                         ДОБАВИТЬ x2 К B0 КЕ
  КЦ КЦ:
  ЦИКЛ
    ДЛЯ ВСЕХ x2 ∈ B1 ВЫПОЛНИТЬ F1(x2): = φ;
                         УДАЛИТЬ x2 ИЗ B1 КЦ;
    ДЛЯ ВСЕХ x2 ∈ B0 ВЫПОЛНИТЬ
    ДЛЯ ВСЕХ x1 ∈ F0(x2) ВЫПОЛНИТЬ
    ДЛЯ ВСЕХ (x1', x2') ∈ G(x1, x2) ВЫПОЛНИТЬ
      ЕСЛИ x1' ∉ F0(x2') ∪ F1(x2') ∪ F(x2') ТО
        ДОБАВИТЬ x1' К F1(x2');
        ДОБАВИТЬ x2' К B1
      КЕ;
    КЦ КЦ КЦ;
    ДЛЯ ВСЕХ x2 ∈ B0 ВЫПОЛНИТЬ
    ДЛЯ ВСЕХ x1 ∈ F0(x2) ВЫПОЛНИТЬ
      ДОБАВИТЬ x2 К B;
      ДОБАВИТЬ x1 К F(x2)
    КЦ КЦ;
    ЕСЛИ B1 = φ ТО ВЫЙТИ КЕ;
      B0: = B1; F0: = F1
  КОНЕЦ ЦИКЛА
КОНЕЦ.
```

Разумеется, польза от введения структур данных будет получена лишь в том случае, если операции над ними будут реализованы достаточно быстрыми процедурами. Программа показывает, какие именно операции требуют быстрой реализации. Это добавление элементов к множествам, перебор элементов множеств B_0 и $F_0(x_2)$, распознавание принадлежности множествам $F(x_2)$, $F_0(x_2)$, $F_1(x_2)$, а также теоретико-множественные присваивания. Если множества X_1 и X_2 заданы списками, то все вспомогательные структуры данных можно привязать к элементам этих списков, что поможет организовать быстрый доступ к значениям вычисляемых функций.

Соответствие между операторами программ ПОРОЖДЕНИЕ 1 и ПОРОЖДЕНИЕ, обеспечивающее корректность реализации, выполняется достаточно просто, и мы предоставляем сделать это читателю.

Поиск существенных переменных. Рассмотрим применение метода ускорения вычислений, использованного в предыдущем примере, для решения одной из важных задач потокового анализа программ – поиска существенных переменных. Рассмотрим $U-Y$-схему программы A над памятью R. Для каждого оператора $y \in Y$ определены множества $\mathrm{In}(y)$ и $\mathrm{Out}(y)$ переменных, используемых и изменяемых оператором y. Также для каждого элементарного условия $u \in \hat{U}$ определено множество $\mathrm{In}(u)$ используемых переменных. Определим для каждого состояния $a \in A$ множества In и Out, полагая

$$\mathrm{In}(a) = \bigcup_{a \xrightarrow{u/y} a'} \mathrm{In}(u) \cup \mathrm{In}(y),$$

$$\mathrm{Out}(a) = \bigcup_{a \xrightarrow{u/y} a'} \mathrm{Out}(y)$$

Основное определение формулируется следующим образом. Переменная r называется *существенной* в состоянии a, если $r \in \mathrm{In}(a)$ или существует путь $a \xrightarrow{u_1/y_1} a_1 \xrightarrow{u_2/y_2} \ldots \xrightarrow{u_n/y_n} a_n = a'$ такой, что $r \in \mathrm{In}(a')$ и $r \notin \mathrm{Out}(y_i)$ $(i = 1, \ldots, n)$.

Определим множество $Q_A \subset R \times A$ как наименьшее множество, удовлетворяющее условиям

1) $r \in \mathrm{In}(a) \Rightarrow (a, r) \in Q_A$;

2) $(a, r) \in Q_A$, $a' \xrightarrow{u/y} a$, $r \notin \mathrm{Out}(y) \Rightarrow (a', r) \in Q_A$.

Нетрудно доказать следующее утверждение.

Т е о р е м а 4.1. *Переменная r существенна в состоянии a тогда и только тогда, когда $(a, r) \in Q_A$.*

Конкретизируя программу ПОРОЖДЕНИЕ для рассматриваемой задачи, получим:

```
ПРОГРАММА СУЩЕСТВЕННЫЕ ПЕРЕМЕННЫЕ (A, T) РЕЗУЛЬТАТ (Q);
НАЧАЛО Q: = φ; Q0: = { (a, r) | r ∈ In(a)};
  ЦИКЛ
```

$Q_1 := \phi$;
ДЛЯ ВСЕХ $(a, r) \in Q_0$ ВЫПОЛНИТЬ
ДЛЯ ВСЕХ $a' \in A$ ТАКИХ, ЧТО ДЛЯ НЕКОТОРОГО ПЕРЕХОДА

$a' \xrightarrow{u/y} a \in T$, $r \notin \mathrm{Out}(y)$ ВЫПОЛНИТЬ
ЕСЛИ $(a', r) \notin Q_0 \cup Q_1 \cup Q$ ТО ДОБАВИТЬ (a', r) К Q_1 КЕ

КЦ КЦ;
$Q := Q \cup Q_0$;
ЕСЛИ $Q_1 = \phi$ ТО ВЫЙТИ КЕ;
$Q_0 := Q_1$
КЦ
КОНЕЦ.

Переходя к использованию программы ПОРОЖДЕНИЕ 1, можно теперь более точно оценить эффект, который дают эти идеи. Предположим, что множество состояний A задано списком, а множество переходов T — с помощью функции T_1, заданной на A и принимающей значения в $\hat{U} \times Y \times A$ так, что $(u, y, a') \in T_1(a) \Leftrightarrow a \xrightarrow{u/y} a'$. Нас интересует оценка времени работы алгоритма порождения существенных переменных в зависимости от числа состояний схемы программы. При этом считается, что n может быть сколь угодно большим, в то время как другие параметры — такие, как число переменных или максимальное число переходов в одном состоянии, — считаются фиксированными. Оценка первой программы при естественной реализации структур данных имеет порядок n^3. Для сокращения этого времени вводим функции F, F_0, F_1 на A и множества B, B_0, B_1, полагая $(a, r) \in Q \Leftrightarrow r \in F(a)$, $a \in B \Leftrightarrow F(a) \neq \phi$; аналогично для F_0, F_1, B_0, B_1. Для сокращения перебора во внутреннем цикле достаточно определить функцию G: $A \to A$ такую, что $a' \in G(a) \Leftrightarrow$ существует переход $p \in T$ такой, что $a' \xrightarrow{p} a$. Теперь переписываем программу с использованием новых структур данных и очевидными трансформациями:

ПРОГРАММА СУЩЕСТВЕННЫЕ 1 (A, T_1) РЕЗУЛЬТАТ (B, F);
НАЧАЛО
$B := B_0 := B_1 := \phi$;
ДЛЯ ВСЕХ $a \in A$ ВЫПОЛНИТЬ
$F(a) := F_0(a) := F_1(a) := G(a) := \phi$
КОНЕЦ ЦИКЛА;
ДЛЯ ВСЕХ $a \in A$ ВЫПОЛНИТЬ
ДЛЯ ВСЕХ $r \in R$ ВЫПОЛНИТЬ

```
        ЕСЛИ r ∈ In(a) ТО ДОБАВИТЬ r К F0(a) КЕ
      КЦ;
      ЕСЛИ F0(a) ≠ φ ТО ДОБАВИТЬ a К B0;
      ДЛЯ ВСЕХ (u, y, a') ∈ T1(a) ВЫПОЛНИТЬ
        ДОБАВИТЬ a К G(a')
      КОНЕЦ ЦИКЛА
    КОНЕЦ ЦИКЛА;
    ЦИКЛ
      ДЛЯ ВСЕХ a ∈ B1 ВЫПОЛНИТЬ
      F1(a): = φ; УДАЛИТЬ a ИЗ B1
    КЦ;
    ДЛЯ ВСЕХ a ∈ B ВЫПОЛНИТЬ
    ДЛЯ ВСЕХ r ∈ F0(a) ВЫПОЛНИТЬ
    ДЛЯ ВСЕХ a' ∈ G(a) ВЫПОЛНИТЬ
      ЕСЛИ ДЛЯ НЕКОТОРЫХ (u, y, a) ∈ T1(a'), r ∉ Out(y) ТО
      ЕСЛИ r ∉ F0(a') ∪ F1(a') ∪ F(a') ТО
        ДОБАВИТЬ r К F1(a');
        ДОБАВИТЬ a' К B1
      КЕ
    КЕ;
  КЦ КЦ КЦ;
  ДЛЯ ВСЕХ a ∈ B0 ВЫПОЛНИТЬ
  ДЛЯ ВСЕХ r ∈ F0(a) ВЫПОЛНИТЬ
    ДОБАВИТЬ a К B;
    ДОБАВИТЬ r К F(a)
  КЦ КЦ
  ЕСЛИ B1 = φ ТО ВЫЙТИ КЕ;
    B0: = B1; F0: = F1
  КОНЕЦ ЦИКЛА
КОНЕЦ.
```

Теперь, если все базовые операторы, кроме, быть может, присваиваний $B_0 := B_1$ и $F_0 := F_1$, выполняются, а условия проверяются за ограниченное время, то, учитывая, что каждая пара (a, r) попадает в множество Q_0 и, следовательно, извлекается из него не более одного раза, получим линейную по n оценку времени работы программы СУЩЕСТВЕННЫЕ 1. Отметим, что для линейности важна ограниченность не только числа переходов из данного состояния, но и ограниченность числа переходов в данное состо-

яние. В случае, когда таких переходов много, нужно строить такую же структуру данных G, что и в программе ПОРОЖДЕНИЕ. В результате можно получить оценку, линейную по сумме числа состояний и числа переходов.

Упражнения

1. Доказать, что оценка $O(mn^2)$ для программы СВЯЗНОСТЬ достигается.
2. Реализовать вычисление числа компонент связности с оценкой $O(m+n)$.
3. Доказать корректность программы ПОРОЖДЕНИЕ.
4. Доказать теорему 4.1.
5. Рассмотреть задачу порождения множества Q при условии, что оно задается рекурсивными определениями следующих типов:

1) а) $P \subset Q$.

   б) $x_1, x_2 \in Q,\ s \in S,\ x' \in X,\ R(x', x_1, x_2, s) \Rightarrow x' \in Q$.

2) а) $P \subset Q;\ S_0 \subset S$.

   б) $x \in Q, s \in S,\ x' \in X,\ R(x', x, s) \Rightarrow x' \in Q$.

   в) $x \in Q,\ s \in S, s' \in Y,\ R'(s', x, s) \Rightarrow s' \in S$.

§ 5. Недетерминированное программирование

Элементы недетерминированных вычислений встречаются уже в самых простых программах. Даже в обычном операторе присваивания два различных подвыражения могут быть вычислены в произвольном порядке (если, конечно, язык не допускает противоречащие принципам структурного программирования побочные эффекты при вызове подпрограмм-функций). Если же выражения рассматривать с точностью до некоторых тождеств (скажем, коммутативность и ассоциативность), вариантов реализации получается еще больше. Другой тип недетерминированности — порядок перебора значений параметра цикла в циклах по элементам множеств. Недетерминированными являются также теоретико-множественные операторы типа ВЗЯТЬ x ИЗ A, НАЙТИ x ТАКОЙ, ЧТО $P(x)$ и т.п. Детерминизация таких операторов происходит при выборе способов представления множеств, которые, как правило, сопровождаются полным или частичным упорядочением их элементов.

Наиболее естественно недетерминизм проявляется при задании преобразований информации с помощью правил преобразования. Пусть D — некоторое множество объектов. Это может быть множество состояний информационной среды, в частности память, или компонента алгебры данных. Правило преобразования R множества D — это просто отношение, заданное на множестве D. Правило применимо к элементу $d \in D$, если существует $d' \in D$ такой, что $(d, d') \in R\left(d \xrightarrow{R} d'\right)$. Пусть $R_1, \ldots, R_m$ — некоторый набор правил преобразования множества D. Применить этот набор правил к элементу d — это значит применять их недетерминированным образом до тех пор, пока возможно. Если последовательность $d \xrightarrow{R_{i_1}} d_1 \xrightarrow{R_{i_2}} \ldots \xrightarrow{R_{i_k}} d_k = d'$ обрывается элементом d', к которому уже не применимо ни одно правило, то говорят, что система правил $R_1, \ldots, R_m$ применима к d и дает результат d'. Если последовательность бесконечна, то применение правил резуль-

тата не дает. Вообще говоря, результат применения правил преобразования определен неоднозначно. Более того, при одних последовательностях применения правил результат может быть определен, при других — нет. Система правил называется всюду определенной, если результат ее применения определен всегда независимо от порядка применения правил. Система называется однозначной, если она всюду определена и результат определен однозначно независимо от порядка применения правил.

Множество D вместе с системой правил $\mathcal{H} = \{R_1, ..., R_m\}$ можно, разумеется, рассматривать как дискретную систему с функцией переходов, определяемой объединением отношений $R_1, ..., R_m$. Состояние d такой системы называется тупиковым, если к нему не применимо ни одно из правил. Настроив систему D парой (D, D_1), где D_1 — множество всех тупиковых состояний, получим преобразование f_D, которое определяется системой правил. Система правил всюду определена, если все допустимые процессы системы D конечны, и однозначна, если система D, кроме того, глобально детерминирована. Глобальная детерминированность является следствием коммутативности отношения достижимости.

Часто применяют логическую терминологию, называя правила преобразования правилами вывода. В этом случае коммутативность отношения достижимости называют также свойством Черча–Россера.

Правила преобразования в случае их всюду определенности удобно использовать для задания функциональных моделей либо как промежуточную форму описания алгоритмов.

Рассмотрим пример. Задан ориентированный граф (S, Q) и функция $\varphi: S^2 \to N \cup \{\infty\}$ $(N = \{0, 1, ...\})$ такая, что если $(s, s') \notin Q$, то $\varphi(s, s') = \infty$. Функция φ задает расстояния между двумя соседними вершинами, а также длину путей (сумму расстояний). Требуется определить кратчайшие расстояния всех вершин графа S от заданного множества $H \subset S$. Иными словами, требуется найти целочисленную функцию f, определенную на множестве S так, что $f(s)$ есть кратчайшее расстояние. Определим функцию $f_0(s)$, полагая $f_0(s) = 0$ для $s \in H$ и $f_0(s) = \infty$ для $s \notin H$. Затем применим к f_0 систему правил $R(s)$, $s \in S \setminus H$, полагая, что правило $R(s)$ применимо к функции f тогда и только тогда, когда $f(s) > \min\limits_{(s, s') \in Q} (f(s') + \varphi(s, s'))$. Результатом применения этого правила будет функция f' которая отличается от f только в точке s и такая, что $f'(s) = \min\limits_{(s, s') \in Q} f(s')$. Коротко данное правило можно записать в виде импликации:

$$R(s): f(s) > \min_{(s, s') \in Q} (f(s') + \varphi(s, s')) \Rightarrow f(s) := \min_{(s, s') \in Q} (f(s') + \varphi(s, s')).$$

Система правил $R(s)$ будет вполне определенной и однозначной. Действительно, каждое применение правила уменьшает значение функции в некоторой точке, а поскольку область определения конечна, цепочка применений правил обязательно оборвется. Докажем однозначность. Пусть f — такая функция, к которой не применимо ни одно из правил, и $f(s) = 0$ для $s \in H$. Тогда f есть искомая функция, т.е. $f(s)$ есть кратчайшее расстояние. Действительно, пусть это не так, и для некоторой вершины s значение $f(s)$ больше, чем кратчайшее расстояние. Пусть $s = s_1 \to s_2 \to ... \to s_n = s'$ — кратчайший путь из s в $s' \in H$. Тогда его окончания тоже будут кратчайши-

ми путями в H для вершин, находящихся на этом пути, и найдется такая вершина s_m, для которой $f(s_m)$ больше, чем кратчайшее расстояние, а для следующей $f(s_{m+1})$ есть кратчайшее расстояние (ясно, что $m + 1 < n$). Но тогда $f(s_m) > f(s_{m+1}) + \varphi(s_m, s_{m+1})$, и правило $R(s_m)$ применимо. Противоречие доказывает, что $f(s)$ есть кратчайшее расстояние. Интересно, что многие хорошие алгоритмы поиска кратчайших путей получаются путем детерминизации семейства правил $R(s)$. Например, алгоритм с оценкой $O(n^2)$, где n – число вершин, может быть получен путем упорядочения применений системы $R(s)$, которое описывается следующим алгоритмом:

НАЧАЛО
  $H_0 := H$; $f := f_0$;
  ЦИКЛ
    $H_1 := \{ s \in S \setminus H_0 \mid Q(s) \cap H_0 \neq \phi \}$;
    ЕСЛИ $H_1 = 0$ ТО ВЫЙТИ КЕ;
    ДЛЯ ВСЕХ $s \in H_1$ ПРИМЕНИТЬ $R(s)$ КЦ;
    $m := \min\{ f(s) \mid s \in H_1 \}$;
    $H_0 := H_0 \cup \{ s \in H \mid f(s) = m \}$
  КЦ
КОНЕЦ.

Корректность алгоритма вытекает из того, что к элементам множества H_0 правила $R(s)$ никогда не применимы, а после выхода из цикла $H_0 = S$. Недетерминированность остается в этом алгоритме в цикле по $s \in H_1$.

Рассмотрим пример. Пусть $M \subset 2^A$ есть покрытие множества A, т.е. $\bigcup_{A' \in M} A' = A$. Следующая система правил позволяет сократить число множеств, входящих в покрытие:

$R_n : A' \subset A_1 \cup \ldots \cup A_n,\ A_1, \ldots, A_n \in A \Rightarrow$ УДАЛИТЬ A' ИЗ M.

Полная определенность системы правил R_n при конечном M и ограниченном n очевидна. Система правил неоднозначна. Она дает только тупиковые покрытия. Для получения минимального покрытия требуется перебирать все возможные пути применения правил. Упорядочивая тем или иным способом перебор, можно кое-что улучшить, но в целом проблема получения минимального покрытия по сумме числа элементов в классах покрытия, или числу элементов покрытия, или по другому критерию остается переборной задачей.

Представление правила в виде импликации, в левой части которой записано условие применения правила, а в правой – действие, которое необходимо совершить для достижения определенной цели, является популярным способом представления знаний о предметной области в системах искусственного интеллекта. Управление последовательным применением правил либо поиск такой последовательности их применения, которая бы привела к желаемым результатам, возлагается на специальные алгоритмы.

Преобразования слов. Систему продукций $\xi_i \to p_i$ $(i = 1, \ldots, n)$ контекстно-свободной грамматики (ξ_i – элементы нетерминального алфавита N, p_i – слова в объединенном алфавите $T \cup N$, где T – алфавит терминальных символов) можно рассматривать как систему правил для обработки слов. Правило $\xi_i \to p_i$ применимо только к словам вида $q\xi_i q'$, а результат его применения есть слово qp_iq'. Конечно, такая система правил обычно не

удовлетворяет ни условию определенности, ни условию однозначности. Продукции грамматики используются для порождения множеств слов (языков) или определения отношения выводимости на множестве слов в алфавите $N \cup T$: q выводимо из $p (p \vdash q) \Leftrightarrow p = q$ или существует последовательность применения правил, которая преобразует p в q. В частности, множество всех терминальных слов (слов в алфавите T), выводимых из символа ξ_i, называется языком, порожденным этим символом в заданной грамматике, а язык, порожденный выделенным символом ξ_0, – языком, порожденным грамматикой.

Обращения продукций грамматики $p_i \to \xi_i$, рассматриваемые как правила преобразования слов, могут быть полезными для определения алгоритмов распознавания принадлежности слова языку, порожденному заданной грамматикой (распознавание сверткой). Такая система правил уже является всюду определенной (если отсутствуют правила вида $\xi_i \to \xi_j$, которые можно известным способом элиминировать). Однако однозначность имеет место далеко не для всех грамматик. В некоторых случаях однозначность может быть обеспечена, если свертку производить слева направо, выбирая для подстановки первое вхождение нетерминального символа.

Системы продукций контекстно-свободных грамматик являются очень частным случаем правил преобразования слов – словарных подстановок, т.е. подстановок вида $p \to q$, где p и q – произвольные слова в заданном алфавите. Подстановка $p \to q$ применима к слову r, если $r = r_1 p r_2$, и преобразует его в слово $r_1 q r_2$.

Система подстановок слов, или ассоциативные исчисления, появились в математической логике в процессе поиска точных понятий алгоритма и вычислимости. Алгоритм А.А. Маркова (нормальный алгоритм) получается как детерминизация ассоциативных исчислений путем добавления правил, регулирующих порядок применения подстановок (применять первое применимое к первому вхождению). Правила преобразования слов типа словарных подстановок широко используются в области обработки строк и в языках, ориентированных на синтаксическую обработку данных. Общий вид такого правила:

$$P(x_1, ..., x_n),\ p_1 x_{i_1} ... x_{i_m} p_{m+1} \to q_1 x_{j_1} ... x_{j_l} q_{l+1},$$

где $P(x_1, ..., x_n)$ – словарный n-местный предикат, $x_1, ..., x_n$ – переменные слова, $p_1, ..., p_{m+1}$, $q_1, ..., q_{l+1}$ – слова в заданном алфавите. Правило применимо к слову r, если $r = r' p_1 r_{i_1} ... r_{i_m} p_{m+1} r''$, $P(r_1, ..., r_n) = 1$, а результат применения правила есть $q_1 r_{j_1} ... r_{j_n} q_{l+1}$.

Иногда применяют более строгое правило, в котором $r' = r'' = e$, но это не является ограничением по существу. Такого рода правила удобны, в частности, для некоторых видов алгоритмов перевода с одного языка на другой. В частности, если $P(x_1, ..., x_n)$ есть конъюнкция высказываний вида $\xi_1 \vdash x_1, ..., \xi_n \vdash x_n$, где $\xi_1, ..., \xi_n$ – нетерминальные символы некоторой контекстно-свободной грамматики, получается контекстно-свободный перевод.

Правила преобразований слов часто применяются в сочетании с правилами, регулирующими порядок применений подстановок, а также с приписыванием правилам дополнительных действий над состоянием информационной среды, которые выполняются всякий раз, когда применяется пра-

вило. Задача распознавания применимости такого правила известна как задача распознавания по образцу (pattern matching).

Обработка термов. Естественной областью применения идей недетерминированного программирования является область формульных преобразований. Объектом преобразований являются выражения некоторой алгебры, равенства, формулы логического языка. Правила обычно имеют вид равенства или соотношений

$$t(x_1, ..., x_n) = t'(x_1, ..., x_n),$$

где $t(x_1, ..., x_n)$ и $t'(x_1, ..., x_n)$ — термы, $x_1, ..., x_n$ — переменные. Термы рассматриваются как элементы алгебры $T_\Omega(A, X, D)$ или $T_\Omega(A, X, \mathcal{H})$. Соотношение $t = t'$ применимо к терму s (не содержащему переменных $x_1, ..., x_n$), если s можно представить в виде $s = \varphi(s')$, и уравнение $s' = t(x_1, ..., x_n)$ имеет решение $(s_1, ..., s_n)$ в алгебре термов. Результат применения соотношения $t = t'$ к терму s в этом случае имеет вид $\varphi(t'(s_1, ..., s_n))$. Чтобы подчеркнуть несимметричность правила, часто говорят об ориентированных соотношениях и вместо равенства употребляют стрелку: $t \to t'$. Системы соотношений называют также системами переписывающих правил (rewriting rules) или системами подстановок термов. Обычно системы подстановок термов рассматривают как системы соотношений в абсолютно свободной алгебре термов, считая $\mathcal{H} = \phi$. Однако многие задачи требуют рассмотрения термов с точностью до некоторых тождеств. Чаще всего это коммутативность, ассоциативность, а также вычисления с константами и некоторые тождества типа свойств единиц и нулей.

Простейшие задачи, которые естественным образом выражаются в терминах систем соотношений, — это всевозможные приведения к каноническим формам. Так, например, соотношения

$$xx^{-1} = e,$$
$$x^{-1}x = e,$$
$$ex = x,$$
$$xe = e,$$
$$(xy)^{-1} = y^{-1}x^{-1},$$
$$x(yz) = (xy)z,$$
$$(xy)y^{-1} = x,$$
$$(xy^{-1})y = x$$

позволяют приводить к канонической форме выражения в свободной группе, т.е. термы, порожденные символами образующих элементов группы, символом единицы e и операциями умножения и обращения. В канонической форме операция обращения (возможно, многократного) применяется только к образующим; два рядом стоящих элемента не могут быть обращениями один другого, и все скобки сгруппированы влево. Если эти выражения рассматривать с точностью до ассоциативности, то последние три соотношения становятся лишними.

Соотношение

$$x^n x^m = x^{n+m}$$

и аналогичное соотношение

$$nx + mx = (n + m)x$$

в аддитивной записи позволяют приводить подобные члены в полугруппе (сложение не предполагается коммутативным). Однако его следует рассматривать как соотношение в двухосновной алгебре: x – элемент полугруппы, n и m – целые числа, x^m – смешанная операция, первый аргумент – элемент полугруппы, второй – число, результат – снова элемент полугруппы. Эти соотношения особенно полезны, если выражения рассматриваются с точностью до операции над целочисленными константами.

В выражении $\{x_1, x_2, ..., x_n\}$, используемом для обозначения конечных множеств (для конкретного n, причем многоточие не входит в выражение, а использовано лишь для сокращения), символ запятой может рассматриваться как символ бинарной ассоциативной операции. Соотношение

$$(x, x) = x$$

позволяет упрощать такие выражения. Его одного достаточно для получения перечисления без повторений, если запятая рассматривается как коммутативная операция. Если же коммутативность не является используемым неявно тождеством, то добавление соотношения

$$(x, y) = (y, x)$$

ничего не дает, поскольку сразу нарушается всюду определенность (это соотношение может применяться бесконечно много раз).

Существенное расширение возможностей техники применения соотношений получается, если пользоваться условными соотношениями. Условное соотношение имеет вид

$$P(x_1, ..., x_n) \Rightarrow t(x_1, ..., x_n) = t'(x_1, ..., x_n).$$

Оно применимо к терму $s = \varphi(s')$, если существует решение $s_1, ..., s_n$ уравнения $s' = t(x_1, ..., x_n)$ такое, что $P(s_1, ..., s_n)$, и применяется как обычное соотношение, т.е. s заменяется на $\varphi(t'(s_1, ..., s_n))$.

Применяя условные соотношения, коммутативность, например, можно заменить упорядоченностью. Предположим, что на множестве элементов, из которых строятся теоретико-множественные выражения, задано отношение линейного порядка $\leqslant$. Тогда если к соотношению $(x, x) = x$ добавить условное соотношение

$$x > y \Rightarrow (x, y) = (y, x),$$

то эти два соотношения дадут возможность до конца упрощать выражения вида $\{x_1, ..., x_n\}$.

Приведение к канонической форме многочленов легко описать с помощью следующей системы:

$$y + z \notin Q_0 \Rightarrow x(y + z) = xy + xz,$$

$$a, b \in Q_0 \Rightarrow ax + bx = (a + b)x, \quad x + ax = (1 + a)x,$$

$$x + x = 2x,$$

$$0 \cdot x = 0, \quad 1 \cdot x = x, \quad x + 0 = x,$$

$$x^m x^n = x^{m+n},$$

$$m > 0 \Rightarrow (x + y)^m = x(x + y)^{m-1} + y(x + y)^{m-1},$$

$$x \neq 0 \Rightarrow x^0 = 1.$$

Операции сложения и умножения рассматриваются как коммутативные и ассоциативные; Q_0 — множество многочленов степени 0 (коэффициенты); показатели степеней неотрицательные.

Технику применения соотношений можно использовать не только для преобразования термов, но также и для преобразования составных объектов над алгебрами, т.е. элементов множества $T^*(A, X, \mathcal{H})$. Такие объекты задаются системами канонических уравнений, и для определения применимости соотношения $t = t'$ к составному объекту s необходимо решать систему уравнений, в которую входят не только уравнения $s' = t(x_1, ..., x_n)$, но и уравнения, которые определяют s. Если же t или t' в свою очередь являются составными, то нужно включать в систему уравнений также и уравнения, определяющие t и t'. Для случая абсолютно свободной алгебры D задача определения применимости соотношения $t = t'$ имеет конструктивное решение, однако следует иметь в виду, что может быть бесконечно много способов применения одного и того же соотношения, связанных с возможностью развертывания деревьев и получения бесконечно многих функций φ в равенстве $s = \varphi(s')$.

После того как алгоритм решения задачи записан с применением недетерминированных элементов в виде систем правил преобразования объектов, есть два пути дальнейшей реализации. Первый путь состоит в применении стандартных универсальных операторов типа

ПРИМЕНИТЬ ПРАВИЛА (R, S).

Параметрами такого оператора являются система правил R из некоторого класса правил, записанная в соответствующем языке и представленная соответствующими структурами данных, а также объект S, к которому требуется применить правила. Оператор может иметь также дополнительные параметры, указывающие порядок применения правил. Если таких параметров нет, используется некоторый фиксированный порядок. Такой подход является довольно удобным, поскольку операторы типа ПРИМЕНИТЬ программируются один раз и могут обслуживать большой класс задач. При этом, однако, могут происходить значительные потери эффективности, поскольку универсальный оператор может не учитывать особенности конкретной системы правил. Другой подход состоит в том, что для заданной системы правил разрабатывается отдельная программа, учитывающая все особенности этой системы. Такой подход, в частности, целесообразен при реализации систем соотношений, которые часто применяются (например, приведение многочлена к канонической форме).

Упражнения

1. Сделать следующий шаг в проектировании алгоритма вычисления кратчайших расстояний.

2. Написать алгоритм построения тупикового покрытия, основанный на применении правил R_n.

3. Написать алгоритм применения правила $p_1 x_{i_1} \dots x_{i_m} p_{m+1} \to q_1 x_{j_1} \dots x_{j_l} q_{l+1}$ (включая проверку применимости) для случая, когда слово, к которому применяется правило, задано в виде одномерного массива.

4. Написать соотношения для приведения к канонической форме рациональных выражений, в которых кроме сложения и умножения допускается возведение в отрицательную целую степень.

§ 6. Рекурсивное программирование

Использование рекурсивных определений и рекурсивных программ наиболее естественно при разработке алгоритмов, использующих рекурсивные структуры данных. В свою очередь, рекурсивные структуры данных используются для реализации теоретико-множественных и функциональных структур данных со сложными областями расположения и определения. В этом параграфе будут рассмотрены некоторые базовые средства реализации и обработки рекурсивных структур данных. Применение этих средств иллюстрируется на примерах программ, поддерживающих работу с соотношениями.

При разработке программ, обрабатывающих рекурсивные структуры данных, выделяются несколько уровней разработки, которые соответствуют уровням абстракции представления моделей программ и данных на разных этапах проектирования. На начальных этапах рекурсивные структуры данных рассматриваются как составные над алгебрами с учетом всех или некоторых тождественных соотношений, действующих в этих алгебрах. На следующих этапах проектирования составные над алгебрами реализуются абсолютно свободными составными, которые, в свою очередь, реализуются абстрактными составными объектами. Уже уровень абстрактных составных допускает ряд важных приемов, обеспечивающих эффективность программ, которые не могут быть выражены на уровне абсолютно свободных составных объектов.

В качестве одного из таких приемов можно назвать склеивание в составном эквивалентных подобъектов (в частности, изоморфных поддеревьев в дереве). Этот прием может дать большую экономию памяти, поскольку число вершин в составном, который получается развертыванием бесциклового составного в дерево, может иметь порядок, равный экспоненте числа вершин (рис. 6.2).

На этапах, предшествующих машинно-зависимым представлениям, часто бывает необходимо работать с составными без какой-либо абстракции. Такая точка зрения позволяет, например, рассматривать различные составные объекты, имеющие общие подобъекты, что, конечно, невозможно при рассмотрение абстрактных составных, поскольку для абстрактных составных t и t' нельзя даже ответить на вопрос, является ли t' подобъектом объекта t, а можно лишь утверждать, что один из них изоморфен одному из подобъектов другого.

Рис. 6.2

На уровне конкретных представлений составных удобно использовать некоторый теоретико-множественный тип данных, элементы которого называются *клубками* составных объектов. Множество M составных объектов называется *замкнутым*, если оно содержит пустой составной w и вместе с каждым объектом все его подобъекты.

Пусть Ω – сигнатура операций, $\Omega = \Omega^+ \cup \Omega_0$, где Ω^+ – множество операций арности большей нуля, Ω_0 – 0-арные операции (первичные составные объекты). Будем рассматривать составные сигнатуры Ω. Выделим в множестве Ω_0 некоторое подмножество X, элементы которого называются *именами* составных. Отображение $v: X \to M$ будем называть *именованием* множества

составных M. Обозначим через $C(M)$ множество всех вершин составных множества M. Замкнутое множества M составных сигнатуры Ω вместе с именованием υ: $X \to M$ называется *клубком* составных, если каждый объект множества M является подобъектом объекта $\upsilon(x)$ для некоторого $x \in X$. Другими словами, каждая вершина $c \in C(M)$ достижима из начальной вершины некоторого составного $\upsilon(x)$, имя которого x. Клубок составных можно рассматривать как четверку (M, Ω, X, υ). Клубки называются *однотипными*, если совпадают их сигнатуры и множества имен.

Пусть (M, Ω, X, υ) и $(M', \Omega, X, \upsilon')$ – два однотипных клубка, $C' = C(M')$, $C = C(M)$. Взаимно однозначное отображение γ множества C на C' называется *изоморфизмом*, если оно индуцирует изоморфизм соответствующих составных объектов (объект, порожденный вершиной c, изоморфен объекту, порожденному вершиной $\gamma(c)$) и сохраняет именование, т.е. образ $\gamma(c)$ начальной вершины c составного $\upsilon(x)$ совпадает с начальной вершиной составного $\upsilon'(x)$.

Обозначим через $c(t)$ начальную вершину составного t. Тогда последнее условие можно записать в виде равенства $\gamma(c(\upsilon(x))) = c(\upsilon'(x))$. Два однотипных клубка изоморфны, если существует изоморфизм одного из них на другой. Клубки составных удобно рассматривать с точностью до изоморфизма, поскольку природа вершин составных объектов в действительности значения не имеет, по крайней мере если рассмотрения носят машинно-независимый характер. Если нужно подчеркнуть, что клубок рассматривается с точностью до изоморфизма, то будем говорить об *абстрактных* клубках.

Клубки составных можно рассматривать двумя различными способами. С одной стороны, клубок можно рассматривать как теоретико-множественную структуру данных определенного типа, а множество $K(\Omega, X)$ всех однотипных клубков – как множество значений этого типа. С другой строны, $K(\Omega, X)$ можно рассматривать как особый вид информационной среды, тогда клубки – это состояния информационной среды. Такой подход удобен для описания преобразований клубков в терминах операторов и условий, действующих на $K(\Omega, X)$. Информационную среду $K(\Omega, X)$ можно рассматривать как память, состояниями которой являются отображения υ: $X \to M$. Эта память не является ни простой, ни функциональной, поскольку, например, всякое допустимое состояние должно удовлетворять следующему условию: если начальные вершины составных $\upsilon(x)$ и $\upsilon(y)$ совпадают, то эти составные равны. Правда, ее можно промоделировать памятью с косвенным именованием, если в качестве множества переменных V взять $X \cup C$, где $C = C(M)$, а в качестве состояния памяти взять отображение b: $V \to \Omega$, полагая $b(x) = \upsilon(x)$ для $x \in X$ и $b(c) = \varphi(c)$ – отметка вершины c, если $c \in C$. При этом следует учесть, что множество C может изменяться в процессе функционирования программ (динамическая память) либо рассматривать частично определенные состояния, взяв в качестве C произвольное счетное множество такое, что всегда $C(M) \subset C$. Такая модель соответствует понятию указателя (pointer), которое широко используется в современных языках программирования. Множество X при этом соответствует указателям, а C – элементам памяти, динамически выделяемым для хранения единиц информации – отметок и связей. Аппарат указателей можно с успехом использовать для реализации клубков.

Перейдем к описанию средств обработки клубков, рассматривая их как состояния информационной среды.

Будет определен некоторый минимальный набор средств, который может быть расширен и обогащен в различных направлениях. Сначала рассмотрим несколько вспомогательных понятий, используемых для описания преобразований клубков. Тройка $c \xrightarrow{i} c'$ называется *связью* клубка M, если c' есть вершина клубка, непосредственно подчиненная по i-й связи вершине c, или пустой составной объект w, если такой вершины нет. Если вершина c имеет отметку ω арности $n \geqslant 0$ и связи $c \xrightarrow{i} c_i$ $(i = 1, \ldots, n)$, то скажем, что вершина c определяется своим разложением $c \to \omega(c_1, \ldots, c_n)$. Кроме связей вершин будем рассматривать также именующие связи $x \xrightarrow{v} c$, где $x \in X$, $c = c(v(x))$. При этом если $v(x) = w$ — пустой составной, то именующая связь имеет вид $x \xrightarrow{v} w$. Пустой составной рассматривается как первичный составной, т.е. он имеет арность 0, но не имеет вершины, на которой он расположен. Он может использоваться как неопределенное значение при использовании отношения аппроксимации на множестве составных. Каждый конкретный клубок можно однозначно задать одним из двух способов:

1) множеством разложений всех вершин множества $C(M)$ и именующих связей для всех $x \in X$;

2) множеством всех своих связей (как связей вершин, так и именующих связей) вместе с функцией *отметок* $\varphi : C(M) \to \Omega$.

Связи удобно использовать для графического изображения клубков. Так, на рис. 6.3 изображен клубок, в котором, например, $c_8 \to \omega(w, w, c_2)$, $u \xrightarrow{v} c_8$, $c_7 \to c_3 - c_6$, $c_3 \to c_4 + c_5$, $c_3 \xrightarrow{1} c_4$, $c_8 \xrightarrow{1} w$, $c_4 \to x$, $c_5 \to 1$ и т.д. Нумерация связей, если ее нет на стрелках, восстанавливается, начиная от левой

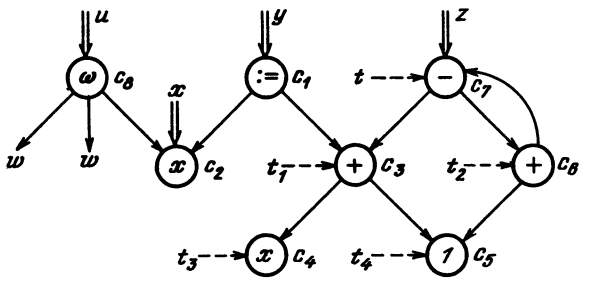


Рис. 6.3

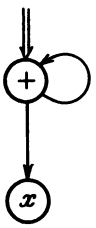


Рис. 6.4

горизонтали, против часовой стрелки. Каждый составной в клубке можно рассматривать как представление однозначно определенного абстрактного или абсолютно свободного составного объекта. Так, например, объект $t = v(z)$ как абстрактный задается канонической системой разложений

$t = t_1 - t_2,$
$t_1 = t_3 + t_4,$
$t_2 = t_4 + t,$
$t_3 = x,$
$t_4 = 1,$

а как абсолютно свободный он является решением уравнения

$$t = (x + 1) - (1 + t)$$

в алгебре абсолютно свободных составных. Если же t рассматривать как элемент регулярного расширения алгебры целых чисел, то он является наименьшим решением уравнения

$$t = x + t$$

и может быть представлен составным рис. 6.4. Интересно обратить внимание на сокращение описаний при переходе от одного уровня абстракции к более высокому.

Базовыми операторами, используемыми для преобразования клубков составных, являются два типа присваиваний – *установка* и *замена*. Первый имеет вид $r \to s$, второй – вид $r := s$, где r – именующее выражение, s – выражение. Выполнение *оператора присваивания* состоит из трех этапов. На первом вычисляется значение правой части, затем вычисляется значение левой части и выполняется третий, завершающий этап присваивания. Значением правой части всегда является составной объект, расположенный в клубке. Обозначим через c_0 начальную вершину вычисленного составного или пустой составной, если вычисленное значение есть w. При выполнении установки значением левой части будет связь. Пусть эта связь есть $c_1 \xrightarrow{i} c_2$. Тогда эта связь удаляется из клубка и добавляется новая связь $c_1 \xrightarrow{i} c_0$. Если же связь именующая, например $x \xrightarrow{v} c_1$, то она заменяется новой связью $x \xrightarrow{v} c_0$, т.е. происходит изменение именования. В случае замены результатом вычисления левой части является вершина c клубка. Если эта вершина имеет разложение $c \to \omega'(c'_1, \ldots, c'_m)$, то оно удаляется из клубка и заменяется разложением $c \to \omega''(c''_1, \ldots, c''_n)$, где $c_0 \to \omega''(c''_1, \ldots, c''_n)$ есть разложение вершины c_0. В случае, когда $c_0 = w$, действие замены состоит в том, что вершина c должна быть удалена из клубка, а все связи вида $c' \to c$ должны быть заменены связями $c' \to w$. Для того чтобы это действие было реализовано эффективно, требуются специальные средства. Чтобы избежать сложности, можно такой случай просто запретить и считать действие оператора $x := w$ неопределенным. Возможен случай, когда левая часть присваивания не определена. Тогда действие присваивания также не определено. В результате выполнения присваивания в клубке могут появиться недостижимые вершины. Такие вершины на заключительном этапе удаляются из клубка.

Различие между установкой и заменой видно на рис. 6.5, где показано действие установки и замены для случая, когда r и s – переменные. Если переменная s находится в правой части, то ее значение есть $v(s)$. Значением переменной r левой части установки является именующая связь $r \xrightarrow{v} c(v(r))$, а левой части замены – вершина $c(v(r))$.

Определим теперь понятие выражения. Именующие выражения – это имена, а также выражения вида $\mathrm{ARG}(t, t_1, \ldots, t_k)$ и $\mathrm{VAL}(x)$, где t, t_1, $\ldots$ $\ldots$, t_n – выражения, x – именующее выражение. Первичные выражения – это именующие выражения, первичные составные, т.е. изображения элементов множества Ω_0, а также выражения вида $\mathrm{VAL}(t)$ и $\mathrm{CAM}(t)$, где t – выражение. Предполагается, что Ω_0, кроме имен, содержит еще целые

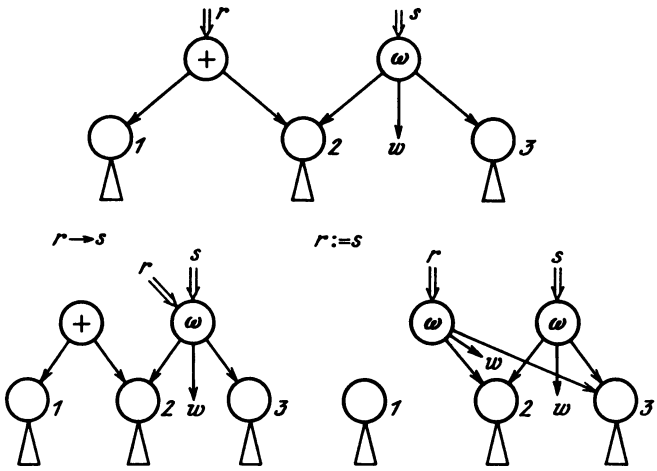


Рис. 6.5

числа. Более сложные выражения получаются из первичных и уже построенных путем применения операций сигнатуры Ω^+. Для этих операций зафиксированы определенный синтаксис, старшинство операций и правила восстановления пропущенных скобок. Кроме операций сигнатуры Ω^+ допускается использование операторов обращения к подпрограммам-функциям, фактическими параметрами которых являются выражения данного клубка и которые вырабатывают в качестве результатов составные, расположенные в этом же клубке.

При вычислении значения правой части может измениться множество M составных клубка (M, Ω, X, υ). Однако эти изменения не должны менять именование υ, от которого зависит вычисляемое значение. Поэтому изменение множества M добавляет лишь недостижимые составные, которые могут стать достижимыми после завершения присваивания. Обозначим значение правой части t присваивания через $\mathrm{val}(t)$. Функция val определяется следующими условиями.

1. Если $x \in X$, то $\mathrm{val}(x) = \upsilon(x)$.

2. $\mathrm{val}(\mathrm{ARG}(t, t_1, \ldots, t_k)) = \arg(\mathrm{val}(t), \mathrm{val}(t_1), \ldots, \mathrm{val}(t_k))$, где $\arg(t, i_1, \ldots, i_k) = s$, если в t определен путь $c(t) \xrightarrow{i_1} c_1 \xrightarrow{i_2} \ldots \xrightarrow{i_k} c_k$ и $c(s) = c_k$. Если же такого пути нет или $c_k = w$, то $\arg(t, i_1, \ldots, i_k) = w$.

3. $\mathrm{val}(\mathrm{VAL}(t)) = \mathrm{val}(\mathrm{val}(t))$.

4. Если выражение t есть выражение алгебры составных, т.е. использует лишь операции сигнатуры Ω, то $\mathrm{val}(\mathrm{CAM}(t)) = t$. Составной t добавляется к множеству M вместе со своими вершинами, которые выбираются так, чтобы они были отличны от вершин множества $C(M)$.

5. Пусть $t = \omega(t_1, \ldots, t_n)$, где $\omega \in \Omega$ (в частности, $\omega \in \Omega_0$ при $n = 0$). Тогда $\mathrm{val}(t) = \omega(\mathrm{val}(t_1), \ldots, \mathrm{val}(t_n))$. Если $c(\mathrm{val}(t_1)) = c_1, \ldots, c(\mathrm{val}(t_n)) = c_n$, то в клубок добавляется новая вершина c и разложение $c \to \omega(c_1, \ldots, c_n)$. В частности, если t — первичный, отличный от имени составного, то при вычислении его значения он просто копируется в клубке.

6. Если $t = F(t_1, \ldots, t_n)$, где F — символ подпрограммы-функции, имеющей заголовок ПРОГРАММА $F(x_1, \ldots, x_n)$ РЕЗУЛЬТАТ (y), то для того, чтобы можно было вычислить с помощью этой программы значение, должны выполняться следующие условия: а) формальные параметры $x_1, \ldots, x_n$ и y должны быть именами клубка M; б) с программой связан магазин значений параметров, который используется для обеспечения их локализации и возможности рекурсивных вызовов.

Вычисление $\mathrm{val}(t)$ происходит следующим образом. Сначала текущие значения формальных параметров $x_1, \ldots, x_n$ и y упрятываются в магазин параметров программы F, затем имена $x_1, \ldots, x_n$ устанавливаются на значения $\mathrm{val}(t_1), \ldots, \mathrm{val}(t_n)$ фактических параметров, т.е. выполняется оператор $x_1 \to t_1, \ldots, x_n \to t_n$; имя y устанавливается на пустой объект $y \to w$. Затем выполняется программа и определяется значение $\mathrm{val}(t) = v(y)$. Наконец, при выходе из программы восстанавливаются упрятанные в магазин значения формальных параметров $x_1, \ldots, x_n, y$. Корректная программа должна работать так, чтобы не изменять именования, т.е. побочный эффект запрещается. Этот запрет может быть обеспечен синтаксически путем полной локализации промежуточных данных, но при этом потребуются копирование фактических параметров и запрет обращения к внешним именам, даже косвенно через VAL. При этом может пострадать эффективность. Поэтому лучше пользоваться какими-либо достаточными условиями корректности или доказывать ее в каждом конкретном случае.

Перейдем к вычислению значения левой части $\mathrm{val}_L(t)$. Приведем правила для вычисления левой части для установки:

1) $x \in X \Rightarrow \mathrm{val}_L(x) = (x \xrightarrow{v} c(v(x)))$ (с учетом условия $c(w) = w$);

2) $\mathrm{val}_L(\mathrm{ARG}(t, t_1, \ldots, t_k)) = (c_{k-1} \xrightarrow{i_k} c_k)$, если существует путь $c(\mathrm{val}(t)) \xrightarrow{i_1} c_1 \xrightarrow{i_2} \ldots \xrightarrow{i_k} c_k)$, где $i_1 = \mathrm{val}(t_1), \ldots, i_k = \mathrm{val}(t_k)$ (случай $c_k = w$ не исключается), в противном случае значение не определено;

3) $\mathrm{val}_L(\mathrm{VAL}(t)) = \mathrm{val}_L(\mathrm{val}(t))$. Это значение может быть определено, лишь если $\mathrm{val}(t)$ есть имя или операция ARG входит в Ω^+ и $\mathrm{val}(t)$ есть ARG. Приведем правила вычисления левой части для замены:

1) если $v(x) \neq w$, то $\mathrm{val}_L(x) = c(v(x))$; если $v(x) = w$, то $\mathrm{val}_L(x) = c$, где c — новая вершина, которая добавляется к клубку;

2) $\mathrm{val}_L(\mathrm{APG}(t, t_1, \ldots, t_n)) = c_k$, если существует путь $c(\mathrm{val}(t)) \xrightarrow{i_1} c_1 \xrightarrow{i_2} \ldots \ldots \xrightarrow{i_k} c_k$, $i_1 = \mathrm{val}(t_1), \ldots, i_k = \mathrm{val}(t_k)$; при этом, если $c_{k-1} \xrightarrow{i_k} w$, в качестве c_k выбирается новая вершина, а связь $c_{k-1} \xrightarrow{i_k} w$ заменяется на $c_{k-1} \xrightarrow{i_k} c_k$; если нужного пути нет, то значение левой части не определено;

3) $\mathrm{val}_L(\mathrm{VAL}(t)) = \mathrm{val}_L(\mathrm{val}(t))$.

Если t — составной, то функции АРНОСТЬ (t) и ОТМЕТКА (t) позволяют получить характеристики его начальной вершины. Далее, используя отношения и предикаты, заданные на множестве отметок Ω, можно определять базовые условия и строить условные операторы, циклы и другие средства управления вычислительным процессом. Алгоритмы преобразования клубков составных объектов удобно строить с использованием других структур данных, связывая их с составными через первичные объекты, которые могут включаться в типы базовой алгебры. При этом следует

различать действие операций базовой алгебры над первичными и действие операций сигнатуры Ω^+.

Клубки составных объектов естественным образом реализуются с помощью функциональных структур данных (массивов). Эта реализация требует решения ряда важных вопросов, существенным образом влияющих на ее эффективность. Отметим некоторые из этих вопросов. Первый вопрос – организация динамической памяти, выделяемой для клубков. Простейшее решение – выделить постоянную область в основной памяти для расположения элементов клубка и область для именования. Размеры выделяемых областей определяют максимальный объем клубка и максимально возможное количество имен. При работе с большими клубками, а также при необходимости рационального распределения основной памяти применяют страничную организацию или другие средства, позволяющие динамически изменять общий объем памяти, выделенной для хранения клубка.

Использование памяти второй ступени приводит к проблеме расположения составных в страницах. Если составной расположен неудачно, при его обработке может потребоваться частая смена страниц, что снижает эффективность. В некоторых случаях можно пойти на то, чтобы время от времени переписывать клубок или отдельные составные так, чтобы связанные между собой вершины по возможности находились на одной странице и общее число страниц, используемых для записи одного составного, было возможно меньшим. Универсальные процедуры управления динамической памятью не могут обеспечить одинаковой эффективности во всех случаях. Поэтому желательно иметь возможность программно влиять на расположение клубка внутри страничных файлов. Еще лучше проектировать это расположение, учитывая особенности прикладной системы, которая реализуется на базе клубков составных объектов.

Второй вопрос, связанный с реализацией клубков, – это вопрос расположения составных внутри связных областей памяти (например, страниц). Удобно применять следующую терминологию. Будем считать, что клубки располагаются на страницах фиксированной длины, которые объединяются в связки. Число элементов связки изменяется динамически (увеличивается или уменьшается) в зависимости от текущих размеров клубка. Каждая страница представляет собой одномерный массив элементов определенного типа (байты, слова, записи определенного типа и т.п.) Адреса этих элементов или их номера в определенной системе нумерации представляют вершины составных объектов, а связи и разложения вершин представляются наборами элементов. Возможны два основных решения. Либо разложение каждой вершины занимает несколько подряд идущих элементов, либо эти элементы могут быть расположены произвольным образом внутри одной страницы (гораздо реже разрешается использование элементов разных страниц). В первом случае экономится память, но усложняется проблема поиска свободного места при необходимости ввести новую вершину. Во втором случае необходимы дополнительные связи между элементами страниц, но упрощается поиск свободных элементов. Все они связываются в один общий список, из которого при необходимости выбирается нужное число элементов. Возможны также смешанные представления, при которых последовательность подряд идущих элементов используется для представления разложений нескольких вершин.

Серьезную проблему представляет собой "сборка мусора". Дело в том, что вершина из клубка и, следовательно, выделенная для нее память освобождаются в том случае, когда они становятся недостижимыми для вершин, доступных с помощью именования. Для того чтобы определить это, необходимо иметь информацию о связях, входящих в данную вершину. Если таких связей нет, то вершина недостижима и может быть удалена. Для того чтобы иметь необходимую информацию, можно каждой вершине приписать счетчик, значение которого равно числу связей, входящих в данную вершину. Этот счетчик должен обновляться каждый раз при изменении количества связей, т.е. при выполнении присваиваний, в которых принимает участие данная вершина. На это затрачиваются дополнительные время и память. Другая альтернатива состоит в том, что сборка мусора происходит лишь в тот момент, когда необходимо ввести новую вершину, а свободная память отсутствует. Отделить свободную память от занятой можно, обойдя все вершины клубка, достижимые через именующие связи и выделив эти вершины определенным признаком. Такую периодическую сборку мусора можно сочетать с переписыванием клубка для получения рационального распределения его вершин по страницам. Наиболее эффективной является программная сборка мусора сразу, когда он появляется. Для этой цели нужно иметь оператор освобождения вершины клубка. Программирование усложняется, однако сложности можно спрятать на низших уровнях описания проектируемой программы.

Обычно в качестве первичных составных используются простые объекты – такие, как имена, числа, символы, – и для их представления используются элементы клубка. Если же первичные объекты имеют разные размеры, например если Ω_0 представляет собой объединение нескольких типов данных или даже структур данных, то эти объекты целесообразно хранить в отдельной памяти первичных составных, имеющей структуру, соответствующую типам первичных, а в памяти клубка представлять их ссылками на память первичных. То же самое можно сказать и об операциях множества Ω^+, которые могут быть представлены структурами данных, расположенными в отдельной памяти.

Во многих задачах множество имен составных нельзя заранее зафиксировать, и желательно иметь простой механизм генерации имен. Удобно считать, что в качестве множества имен X выбрано некоторое бесконечное множество, например занумерованное целыми числами, но в каждый момент времени лишь конечное число имен занято. Генерация нового имени осуществляется с помощью оператора, скажем, x := НОВОЕ ИМЯ. Если представляется возможность освобождать ненужные номера, например, с помощью оператора ОСВОБОДИТЬ ИМЯ x, то механизм генерации становится несколько более сложным. Введение новых имен и их удаление сопровождаются действиями по управлению динамической памятью, выделяемой для представления именования.

Выражения, используемые в операторах присваивания, можно рассматривать как выражения алгебры составных объектов, если Ω^+ расширить недостающими операциями такими, как ARG, VAL, CAM и операции обращения к программам-функциям. Это дает возможность расширить определение функции VAL и вычислять значения в более богатой алгебре составных объектов. Аналогичным образом в алгебру составных погружаются

базовые операторы и условия, а затем и схемы программ над клубками составных объектов. Осуществив такое погружение, можно определить теперь оператор интерпретации составных объектов, рассматриваемых как представления программ над клубками. Оператор ИНТЕРПРЕТИРОВАТЬ (x, M) РЕЗУЛЬТАТ (M) осуществляет обход составного $\upsilon(x)$, расположенного в клубке M, рассматривая его как программу над этим же клубком, проверяет условия и выполняет операторы в соответствии с их семантикой. Поскольку внутри клубка нет различия между программами и данными, программы могут изменять сами себя. Поэтому функции, вычисляемые такими программами, могут быть устроены довольно сложно. Использование такой возможности требует большой осторожности и внимания, но гибкость и возможность достичь в ряде случаев большой эффективности делают это средство интересным и полезным.

Рассмотрим пример разработки программы, ориентированной на работу с клубками составных объектов. Предметом дальнейшего рассмотрения будет задача решения системы уравнений в алгебре свободных составных объектов. Пусть

$$\begin{aligned} &t_1(x_1, \ldots, x_n) = t'_1(x_1, \ldots, x_n), \\ &\ldots\ldots\ldots \\ &t_m(x_1, \ldots, x_n) = t'_m(x_1, \ldots, x_n) \end{aligned} \qquad (6.1)$$

суть система соотношений в алгебре свободных составных объектов, $x = (x_1, \ldots, x_n)$ – неизвестные. При построении объектов $t_i(x)$ и $t'_i(x)$ неизвестные используются как первичные составные объекты, которые при погружении в клубок становятся именами составных. Вектор $s = (s_1, \ldots, s_n)$ называется решением системы (6.1), если равенства

$$\begin{aligned} &t_1(s_1, \ldots, s_n) = t'_1(s_1, \ldots, s_n), \\ &\ldots\ldots\ldots \\ &t_m(s_1, \ldots, s_n) = t'_m(s_1, \ldots, s_n) \end{aligned}$$

истинны в алгебре свободных составных. Функции $t_i(x_1, \ldots, x_n)$ и $t'_i(x_1, \ldots, x_n)$ не обязательно элементарны. Они могут быть алгебраическими, т.е. допускаются составные с циклами. Система (6.1) либо вовсе не имеет решения, либо, как будет показано ниже, имеет наименьшее решение.

Частным случаем рассматриваемой задачи является задача унификации термов, известная в математической логике в связи с методом резолюций поиска доказательств теорем. В задаче унификации t_i и t'_i являются термами (бесцикловыми составными), и решение также ищется в алгебре термов. Наименьшее решение называется при этом наиболее общим унификатором.

Покажем, что из существования решения системы (6.1) вытекает и существование наименьшего решения. Каждый из составных системы уравнений (6.1) как алгебраическая функция может быть представлен в виде наименьшего решения канонической системы уравнений. Поэтому система соотношений (6.1) при произвольном x равносильна системе

вида

$$
\begin{aligned}
&u_1(x, y) = u_1'(x, y),\\
&\cdots\cdots\cdots\\
&u_m(x, y) = u_m'(x, y),\\
&y_1 = v_1(x, y),\\
&\cdots\cdots\cdots\\
&y_k = v_k(x, y),
\end{aligned}
\tag{6.2}
$$

где $y = (y_1, \ldots, y_k)$ образует наименьшее решение системы

$$
\begin{aligned}
&y_1 = v_1(x, y),\\
&\cdots\cdots\cdots\\
&y_k = v_k(x, y),
\end{aligned}
\tag{6.3}
$$

которая определяет все составные t_i и t_i' (они равны правым частям некоторых из уравнений (6.3)), а функции u_i, u_i', v_j уже элементарны (бесцикловые составные).

Среди уравнений (6.2) могут встретиться уравнения вида $z = z'$, где $z, z' \in X = \{x_1, \ldots, x_n, y_1, \ldots, y_m\}$. Обозначим множество всех таких уравнений через A_0. Множество A_0 порождает отношение эквивалентности α на множестве X — наименьшее отношение эквивалентности такое, что из $z = z' \in A_0$ следует $z = z' (\mathrm{mod}\,\alpha)$ (рефлексивно-симметрично-транзитивное замыкание множества A_0). Если $A_0 = \phi$, то отношение α есть равенство символов множества X.

Множество всех уравнений системы (6.2) обозначим через A и будем рассматривать каждое уравнение с точностью до перестановки левой и правой частей. Преобразуем множество A с помощью следующих правил:

1) $\omega(s_1, \ldots, s_l) = \omega(s_1', \ldots, s_l') \Rightarrow s_1 = s_1', \ldots, s_l = s_l'$;

2) $z = \omega(s_1, \ldots, s_l)$, $z' = \omega(s_1', \ldots, s_l')$, $z = z'(\alpha) \Rightarrow z = \omega(s_1, \ldots, s_l)$, $s_1 = s_1', \ldots, s_l = s_l'$.

В этих правилах $s_1, \ldots, s_l, s_1', \ldots, s_l'$ — термы, $z, z' \in X$. Правило применимо к A, если уравнения левой части содержатся в A. Результатом применения правила будет новое множество уравнений. Оно получается заменой уравнений левой части на уравнения правой. С изменением множества A меняется A_0 и порождаемое им отношение эквивалентности α. Система правил 1–2 всюду определена, поскольку применение каждого из правил сокращает общее число операций в системе. Кроме того, каждый переход к новой системе сохраняет все решения предыдущей.

Уравнение вида $\omega(s_1, \ldots, s_p) = \omega'(s_1', \ldots, s_q')$ называется противоречивым, если $\omega \neq \omega'$. Такое уравнение, а вместе с ним и система, в которую оно входит, не имеет решения. Противоречивой также является подсистема $z = \omega(s_1, \ldots, s_p)$, $z' = \omega'(s_1', \ldots, s_q')$, если $z = z' (\mathrm{mod}\,\alpha)$. Наличие такой подсистемы является достаточным условием отсутствия решений.

Рассмотрим систему A, которая получается из системы (6.2) применением правил 1–2. Если в нее входит хотя бы одно противоречивое уравнение или подсистема, то исходная система решений не имеет. Если же все уравнения и подсистемы непротиворечивы, то система может быть приве-

дена к канонической форме

$$\begin{aligned} x_1 &= f_1(x, y), \\ &\cdot \\ &\cdot \\ x_n &= f_n(x, y), \\ y_1 &= g_1(x, y), \\ &\cdot \\ &\cdot \\ y_k &= g_k(x, y) \end{aligned} \qquad (6.4)$$

Действительно, каждое уравнение системы A имеет вид $z = t$, где $z \in X$, иначе было бы применимо правило 1. Рассмотрим классы отношения эквивалентности α для системы A. В каждом из этих классов существует не более одного элемента z такого, что уравнение вида $z = \omega(s_1, \ldots, s_l) \in A$. Если такое уравнение существует, то возьмем это уравнение в (6.2), добавив к нему уравнения $z' = z$ для всех z', отличных от z из этого класса. Если такого уравнения нет, то возможны два случая. Первый случай – в класс входит некоторый элемент y_i. Тогда в (6.4) поместим все уравнения вида $z = y_i$, где z – отличные от y_i элементы рассматриваемого класса. Второй случай – класс состоит только из элементов множества $\{x_1, \ldots, x_n\}$. Такой класс будем называть независимым. Выберем из него произвольным образом представителя x_i и для всех элементов z, отличных от x_i, включим в (6.4) уравнения $z = x_i$. Полученная система равносильна системе A и имеет канонический вид.

Система (6.4) имеет наименьшее решение. В этом решении все независимые переменные, т.е. переменные, выбранные из независимых классов, принимают неопределенное значение w. Любое другое решение получается путем выбора в качестве значений независимых переменных произвольных значений, значения же остальных однозначно определяются соотношениями (6.4). Если все составные наименьшего решения бесцикловые, то оно также дает решение проблемы унификации и определяет наиболее общий унификатор. Таким образом, доказана следующая

Т е о р е м а 6.1. *Система уравнений* (6.1) *в алгебре регулярных абсолютно свободных составных объектов либо не имеет решения, либо имеет наименьшее решение и может быть путем эквивалентных преобразований приведена к канонической форме* (6.4).

Поскольку система (6.4) определяет однозначно составные $x_1, \ldots, x_n$, ее можно считать решением системы (6.1). Если же все составные $x_1, \ldots, x_n$ бесцикловые, то система (6.1) определяет также наиболее общий унификатор.

В доказательстве теоремы 6.1 содержится и алгоритм решения системы (6.1), который будет реализован программой РЕШИТЬ СИСТЕМУ (A, M) РЕЗУЛЬТАТ (ДА, M). В этой программе M – клубок составных объектов сигнатуры Ω, A – система уравнений, представленная как множество пар термов вида $t = t'$, где термы t и t' представлены составными объектами сигнатуры Ω. В результате выполнения данной программы логическая переменная ДА получает значение 1, если решение существует, и 0, если его нет. Новое состояние клубка M представляет наименьшее решение своим именованием. А именно, если x – неизвестная, то $v(x)$ – ее значение в наименьшем решении.

Неизвестные в системе A представлены именами клубка M, значения которых равны пустому составному объекту w. Все остальные имена имеют определенные (непустые) значения и являются параметрами системы. Не исключается случай, когда терм в уравнении из A содержит вхождение имени, значение которого хоть и определено, но зависит от неизвестных. Таким образом, система уравнений, которая определяется выражением A, кроме элементов множества A включает в себя также уравнения вида $x = \upsilon(x)$, где x – имя, которое входит в одно из уравнений множества A, υ – именование клубка M. Более того, система уравнений, определяемая множеством A, – это наименьшее множество $R(A)$, которое включает в себя A и вместе с каждым уравнением $t = t'$ все равенства вида $x = \upsilon(x)$, где x – имя, которое входит в t или t'.

Множество V неизвестных системы $R(A)$ – это наименьшее множество, которое включает в себя множество имен, входящих в $R(A)$ и имеющих неопределенное значение, а также все имена, которые входят в $R(A)$ и значения которых зависят от неизвестных Все остальные имена, которые входят в $R(A)$, называются параметрами. Будем предполагать, что значения всех имен, которые входят в $R(A)$, суть бесцикловые составные (этого всегда можно добиться введением дополнительных параметров). Кроме этого, будем считать, что отношение $\upsilon(x) = y$ на множестве имен X клубка M есть частичный порядок, т.е. всякая цепочка имен $x_1 = \upsilon(x)$, $x_2 = \upsilon(x_1), \ldots, x_k = \upsilon(x_{k-1})$ обрывается на таком имени x_k, что $\upsilon(x_k) = w$ или $\upsilon(x_k) \notin X$. Для каждого имени x имя x_k определяется однозначно. Обозначим его через $\beta(x)$. Отношение $\beta(x) = \beta(y)$ является тогда отношением эквивалентности на множестве X, а на множестве X_0 неизвестных и параметров системы $R(A)$ это отношение совпадает с отношением α.

Рассмотрим программу РЕШИТЬ СИСТЕМУ. В этой программе оператор НАЙТИ (z, t) устанавливает значение имени z на $\beta(t)$ и выполняет некоторые другие действия, о которых будет сказано ниже. Оператор ОБЪЕДИНИТЬ$_*(z, z')$ в простейшем случае эквивалентен установке VAL$(z') \rightarrow z$.

В результате его выполнения классы, которым принадлежат переменные z и z' (точнее, переменные, которые являются значениями выражений z и z'), объединяются. Этот оператор применяется лишь при условии VAL$(z') = w$, поэтому в результате объединения все переменные, α-эквивалентные переменной z', получают значения, равные значению переменной z.

```
ПРОГРАММА РЕШИТЬ СИСТЕМУ (A, M) РЕЗУЛЬТАТ (ДА, M);
НАЧАЛО
   ДА:= 1;
      Образовать отношение α;
      ПОКА A ≠ φ ВЫПОЛНЯТЬ
         Взять и удалить R из A;
         t → ARG(R, 1); t' → ARG(R, 2);
         ЕСЛИ t есть имя и t' есть имя ТО
            НАЙТИ (z, t);
            НАЙТИ (z', t');
            t → VAL(z); t' → VAL(z');
ЕСЛИ t ≠ w И t' ≠ w, ТО
      VAL(z') → w;
   РАСЩЕПИТЬ (t, t')
```

```
        КЕ;
        ОБЪЕДИНИТЬ (z, z');
      ИНАЧЕ ЕСЛИ t есть имя ИЛИ t' есть имя ТО
        ЕСЛИ t' есть имя ТО t → t', t' → t' КЕ;
        НАЙТИ (z, t); t → VAL(z);
        ЕСЛИ t ≠ w ТО
          РАСЩЕПИТЬ (t, t')
        ИНАЧЕ VAL(z) → t' КЕ
      ИНАЧЕ РАСЩЕПИТЬ (t, t') КЕ
    КЦ
    Завершить решение
  КОНЕЦ.
```

Для начала можно предположить, что операторы "образовать отношение α" и "завершить рещение" пустые.

```
  ПОДПРОГРАММА РАСЩЕПИТЬ (t, t');
  НАЧАЛО
    ω :=операция (t); ω':=операция (t');
    ЕСЛИ ω = ω' ТО
      m:= арность (t);
      ДЛЯ i:=1 ДО m ВЫПОЛНИТЬ
              (ARG(t, i) = ARG(t', i)) добавить к A;
      КЦ;
    ИНАЧЕ ДА:=0; КОНЧИТЬ КЕ
  КОНЕЦ.
```

Корректность программы РЕШИТЬ СИСТЕМУ вытекает из того, что каждое прохождение цикла эквивалентно применению одного из правил 1–2 к системе $R(A)$ или сохраняет $R(A)$, уменьшая число элементов A (случай, когда t есть имя или t' есть имя и $\mathrm{VAL}(t) = w$), а когда $A = \phi$ ($R(A)$ при этом не пусто, так как содержит равенства вида $x = v(x)$), система $R(A)$ уже представлена в канонической форме.

Количество повторений основного цикла программы не превосходит числа n, равного сумме числа операций и числа уравнений в множестве A. Время выполнения каждого цикла складывается из времени выполнения операторов ОБЪЕДИНИТЬ и НАЙТИ и из времени выполнения остальных операторов, которое может быть сделано ограниченным. Время выполнения оператора НАЙТИ в худшем случае пропорционально числу имен множества X_0. Поэтому, если число элементов этого множества считать ограниченным, время выполнения программы будет $O(n)$. Время выполнения оператора НАЙТИ можно существенно улучшить, если при каждом обращении сокращать путь от переменной x до переменной $\beta(x)$. Для этой цели может быть использована следующая реализация оператора НАЙТИ:

```
  ПОДПРОГРАММА НАЙТИ (z, t);
  НАЧАЛО
    z → t;
    ПОКА VAL(z) есть имя ВЫПОЛНЯТЬ
      z → VAL(z)
    КЦ;
    ПОКА t ≠ z ВЫПОЛНЯТЬ
        VAL(t) → z;
```

$t \to \mathrm{VAL}(t)$

КЦ

КОНЕЦ.

Действие подпрограммы объясняет рис. 6.6, на котором изображены состояния клубка после установки $z \to t$, после первого цикла, в процессе выполнения второго цикла и в конце выполнения оператора НАЙТИ (z, t).

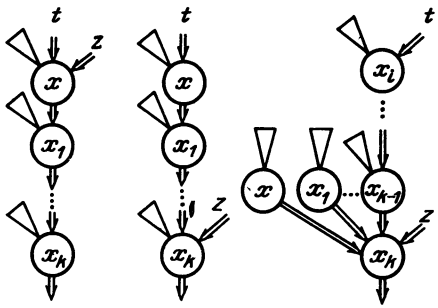


Рис. 6.6

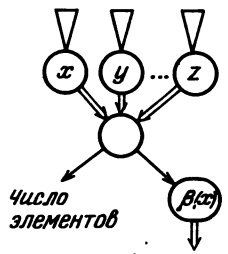


Рис. 6.7

Двойные стрелки обозначают именующие связи. Дальнейшее сокращение времени работы оператора НАЙТИ можно получить, если в операторе ОБЪЕДИНИТЬ связь между представителями объединяемых классов проводить не произвольным образом, а от класса с меньшим числом элементов к классу с большим числом элементов. Для этого необходимо иметь информацию о числе элементов класса. Ее можно разместить в клубке, вставив дополнительную вершину перед представителем класса $\beta(x)$, как показано на рис. 6.7. При объединении числа элементов объединяемых классов складываются. Вставка дополнительных вершин и их удаление выполняются операторами "образовать отношение α" и "завершить решение" соответственно. Известно, что при такой организации работы операторов НАЙТИ–ОБЪЕДИНИТЬ выполнение последовательности из $O(n)$ вызовов выполняется за почти линейное время, т.е. за время $O(nG(n))$, где $G(n)$ – функция, которая растет медленнее, чем любая итерация логарифма.

Комментарии к главе 6

Обзор методов оптимизации программ содержится в [54]. Теория эквивалентности схем программ и дискретных преобразователей составляет важный раздел теоретического программирования [42, 28, 57]. В работе [17] В.М. Глушковым показано, что в алгебре алгоритмов можно выполнять довольно глубокие оптимизирующие преобразования. Метод В.М. Глушкова был обобщен в работе [61]. Метод оптимизации программ, примененный в § 4, известен как метод введения и удаления избыточных вычислений. Технология программирования с применением соотношений применялась в языке аналитик [49] и рассматривалась в [61]. Алгоритм решения систем уравнений, описанный в § 6, является модификацией алгоритма Мартелли–Росси [89]. Структуры данных и свойства операторов ОБЪЕДИНИТЬ–НАЙТИ подробно изучаются в [5].

Глава 7

РАСПРЕДЕЛЕННЫЕ МНОГОПРОЦЕССОРНЫЕ СИСТЕМЫ

§ 1. Принцип макроконвейера

В этой главе рассматриваются вопросы проектирования программ для распределенных многопроцессорных систем, т.е. многопроцессорных систем с распределенным управлением, распределенной памятью и универсальной системой связи. Распределенные системы имеют определенные преимущества перед другими типами многопроцессорных систем. Прежде всего, поскольку каждый процессор обладает собственной основной памятью, можно предположить, что обмены между процессорами будут проходить с частотой значительно меньшей, чем обмен между процессором и его собственной памятью. Поэтому можно использовать системы связи со скоростями обменов, более низкими по отношению к скорости обмена с памятью. Это означает, что сложность, а следовательно, и стоимость системы связи будут расти значительно медленней, чем квадрат числа процессоров. Практически, используя иерархическую организацию, подобную той, которая используется в телефонных сетях, можно ограничиться порядком роста $n\log n$, где n – число процессоров. Скорости обменов при увеличении числа процессоров падают незначительно. Таким образом обеспечивается возможность неограниченного наращивания потенциальной производительности как за счет увеличения числа процессоров в системе с практически линейным ростом стоимости, так и за счет увеличения производительности каждого процессора.

Другим преимуществом распределенных многопроцессорных систем является их гибкость, т.е. возможность произвольно организовывать систему, устанавливая необходимые связи между процессорами. Эти связи могут перестраиваться динамически с учетом требований задач, решаемых в системе.

Наконец, системы с распределенной памятью позволяют эффективно использовать оперативную память большого объема. При этом реальная производительность для многих задач растет просто за счет отказа от использования памяти второй ступени во время решения задачи, а с ростом числа процессоров растет степень распараллеливания доступа к оперативной памяти.

Основной проблемой использования распределенных многопроцессорных систем является проблема организации вычислений. В случае многопрограммной работы или при решении задач, которые сводятся к большому числу независимых или слабозависимых подзадач, ситуация достаточно ясна. Трудности возникают когда такое сведение явно не задано. Для их преодоления требуется развитие новых методов и средств организации вычислений. К таким методам относятся, в частности, методы макроконвейерной обработки данных, использующие принцип макроконвейера, предложенный В.М. Глушковым в 1978 г.

Суть принципа макроконвейерной обработки данных состоит в том, что при распределении работы между процессорами каждому процессору на очередном шаге вычислений дается такое задание, которое может на

длительное время загрузить его работой, не требующей взаимодействия с другими процессорами. Последовательное применение принципа макроконвейерной обработки позволяет получить не менее, чем линейное ускорение, в зависимости от числа процессоров, используемых при решении задачи.

Действительно, предположим, что требуется решить задачу вычисления функции $y = f(x)$ некоторым заданным методом, где x и y – структуры данных достаточно большого объема. Время решения задачи зависит от числа операций, которое, в свою очередь, зависит от значения некоторого параметра или набора параметров $n = (n_1, n_2, ...)$, характеризующих исходные данные x. Пусть это время выражается функцией $\varphi(n)$. Параметр n обычно выбирается таким образом, что $\varphi(n)$ растет с ростом n; например, если $y = f(x) = f(x_1, x_2)$ – решение системы линейных алгебраических уравнений с матрицей коэффициенттов x_1 и вектором свободных членов x_2, которое вычисляется одним из прямых методов, то в качестве n можно взять порядок системы. Если же система решается итерационным методом, то в качестве n можно взять пару – порядок системы и число итераций.

Предположим, что удалось разделить работу по вычислению $f(x)$ между p процессорами равномерно так, что каждый процессор должен работать время $\varphi_p(n) = \varphi(n)/p$ при p, изменяющемя от 1 до $k(n)$. Величина $k(n)$ определяет верхнюю границу для числа процессоров, допускающих разумное распределение работы при заданном размере задачи. При совместной работе добавляются еще дополнительные действия по обмену информацией между процессорами. Предположим, что эта работа требует времени $\psi_p(n)$. В это время включается не только собственно время на передачу данных между двумя процессорами, но и время на синхронизацию (ожидание на очередном шаге одного процессора окончания работы другим), а также время на ожидание освобождения каналов при необходимости обмениваться одному процессору с несколькими другими или при ограниченном числе каналов в системе.

Если обозначить через $T_p(n)$ время решения задачи на системе из p процессоров, то ускорение $\alpha_p(n)$ при решении задачи с параметром n за счет использования p процессоров вместо одного выразится формулой

$$\alpha_p(n) = \frac{T_1(n)}{T_p(n)} = \frac{\varphi(n)}{\varphi_p(n) + \psi_p(n)} = \frac{1}{1 + \psi_p(n)/\varphi_p(n)} \cdot p = \beta_p(n)\,p.$$

Формула показывает, что если $\psi_p(n)/\varphi_p(n) < 1$, то при изменении числа процессоров от 1 до $k(n)$ производительность системы при фиксированном объеме обрабатываемых данных растет не медленней, чем линейно с коэффициентом линейности $\beta(n) > 1/2$, где $\beta(n) = \min\limits_{1 \le p \le k(n)} \beta_p(n)$. Коэффициент $\beta_p(n)$ или $\beta(n)$ называется коэффициентом эффективности многопроцессорной системы при решении данной задачи данным методом. Обычно этот коэффициент находится между нулем и единицей. Если время, затрачиваемое на обмены, растет медленней, чем время вычислений, то с ростом n коэффициент эффективности приближается к 1. Приведенные оценки достаточно грубы, однако уже эти оценки показывают направление поиска эффективных алгоритмов для решения задач на распределенных многопроцессорных системах.

Схему макроконвейерной организации вычислений в некоторых случаях можно получить, исходя из описания алгоритма решения задачи в виде последовательной программы. Для этого необходимо прежде всего проанализировать циклическую структуру программы. Можно предположить, что программа представлена в регулярной форме, т.е. представляет собой дерево, образованное условными операторами и состоящее из линейных участков (произведение базовых операторов) и циклов. Линейные участки, естественно, распараллеливать не имеет смысла, поскольку они достаточно быстро могут быть выполнены внутри одного процессора. Распараллеливания требуют многократно повторяющиеся циклы и особенно вложенные системы циклов. Используя соотношение $\{\underset{\alpha}{\ } P\} = \{\underset{\alpha\alpha}{\ } (\epsilon \vee P)^p\}$ в алгебре алгоритмов, можно попытаться распределить работу по выполнению укрупненного цикла между p процессорами $A_1, ..., A_p$, поручив каждому выполнять периодически одну и ту же прогрмму $\underset{\alpha}{(\epsilon \vee P)}$. При этом должна быть организована передача результатов от процессора A_i к A_{i+1} $(i = 1, ..., p-1)$ и от A_p к A_1. Для того чтобы процессоры могли работать параллельно, результаты должны передаваться из процессора A_i к A_{i+1} по частям так, что первая порция данных передается задолго до окончания работы A_i на очередном шаге вычислений, т.е. в очередном цикле. При этом получается работа с перекрытием, а при разумном выборе числа p — без ожиданий.

Полученная организация работы представляет собой линейный конвейер, чем оправдывается термин "макроконвейерная организация". Приставка "макро" подчеркивает, что речь идет о распараллеливании именно внешних циклов, а не внутренних, как это имеет место для машин с аппаратной конвейеризацией на уровне команд и микрокоманд. Начальный период работы линейного конвейера обычно связан с разгоном (ожидание процессором A_{i+1} момента, когда он сможет начать работу, получив первую порцию данных от A_i), а окончание — с торможением (после того как будет выполнено условие α, часть процессоров должна выполнять пустой оператор). Компенсация потерь может быть достигнута за счет перекрытия работы двух следующих друг за другом циклов в программе и последовательного подключения процессоров, составляющих конвейер, к работе в те моменты, когда для них готовы данные. Разумеется, при этом должны использоваться возможности динамической перестройки структуры, присущие распределенным многопроцессорным системам.

Таким образом, для организации параллельного выполнения последовательных программ необходим анализ информационных зависимостей и перестройка структур данных. После того как написана последовательная программа, выявление этих зависимостей и отыскание способов перестройки структур данных могут потребовать значительных усилий. Поэтому более естественным представляется путь построения параллельных программ непосредственно по математическому описанию метода решения задачи в терминах конструктивных определений функции над функциональными, теоретико-множественными или рекурсивными структурами данных.

§ 2. Макроконвейерные сети

Основной структурной моделью программы, выполняемой на распределенной системе, является асинхронная многоуровневая сеть из алгоритмических модулей, в которой модули нижнего уровня выполняются на различных процессорах системы. Сеть может иметь переменную структуру. Поскольку универсальная система связей обеспечивает практически наличие каналов обмена информацией между любыми двумя модулями, то обмены между компонентами по каналам с очередями удобно представлять операторами ПРИНЯТЬ, ПЕРЕДАТЬ, адресуя не каналы, а компоненты. Помимо парных связей в программе могут использоваться и связи по каналам, если это удобно с точки зрения алгоритмического описания функционирования сети.

Макроконвейерная организация вычислений накладывает определенные ограничения на программы компонент и организацию их взаимодействия. Здесь будут рассмотрены несколько типичных моделей макроконвейерных вычислений.

Наиболее простой моделью является статическая макроконвейерная сеть, или статический макроконвейер. Так будем называть простую асинхронную сеть из алгоритмических модулей с определенной структурой алгоритмов функционирования компонент, которая будет описана ниже. Таким образом, взаимодействие между компонентами в простой макроконвейерной сети реализуется с помощью простых очередей. Механизм очередей поддерживается операционной системой, которая может накладывать определенные ограничения на их длину. Сеть предполагается открытой. Входные каналы сети используются для ввода исходных данных, а выходные — для вывода результатов. В реальных программах внешние каналы сети могут соответствовать входным и выходным последовательным файлам либо каналам взаимодействия с компонентами сети более высокого уровня, в том числе с периферийными компонентами, обеспечивающими связь макроконвейерной сети с внешним миром.

В зависимости от структуры программ компонент различаются три основных вида статических макроконвейерных сетей — итеративные, волновые и смешанные макроконвейерные сети. Программа итеративного макроконвейера имеет следующий вид:

```
МОДУЛЬ имя;
    описание информационной среды;
НАЧАЛО
    Ввод исходных данных.
    ЦИКЛ
     обработка данных;
        обмен данными.
    КОНЕЦ ЦИКЛА.
    Вывод результатов.
КОНЕЦ.
```

Операторы "ввод исходных данных" и "вывод результатов" работают с входными и выходными каналами сети соответственно. При этом противоречий не возникает, поскольку в силу простоты каждая компонента имеет свои каналы. Выходы из цикла находятся внутри оператора обработки данных. Обмен данными представляет собой последовательность операторов ПРИНЯТЬ, ПЕРЕДАТЬ, обращенных к другим компонентам сети. Все алгоритмы согласованы таким образом, что число повторений основного цикла во всех модулях всегда одно и то же. Кроме того, компоненты должны быть согласованы по обменам. Это значит, что если в компоненте K есть оператор вида ПЕРЕДАТЬ x в K', то в компоненте K' должен быть оператор вида ПРИНЯТЬ y из K и наоборот. При этом внутри одного модуля не может быть более одного оператора передачи и более одного приема для каждого из других модулей. Внутри обработки данных операторы взаимодействия с другими модулями и с внешней компонентой отсутствуют.

Операторы обмена данными итеративного макроконвейера определяют связи между компонентами, которые могут быть представлены ориентированным графом обменов (M, S) с множеством вершин M, находящимся во взаимно однозначном соответствии с множеством компонент, и множеством дуг $S \subset M^2$. Компоненту, соответствующую вершине x, обозначим через $K(x)$. Вершины x и y графа M соединены дугой, если в программе компоненты $K(x)$ есть оператор передачи в компоненту $K(y)$. Граф M называют иногда также коммуникационным пространством, его вершины – точками, а дуги – связями. Относительно компоненты $K(x)$ говорят, что она расположена в точке x коммуникационного пространства M.

Волновой макроконвейер отличается структурой основного цикла, в котором прием данных всегда предшествует обработке. Цикл работы компоненты волнового макроконвейера имеет вид

ЦИКЛ

прием данных;

обработка данных;

передача данных

КОНЕЦ ЦИКЛА.

Все ограничения, сформулированные для итеративного макроконвейера, сохраняются также и для волнового, за исключением того, что в операторах обмена внутри основного цикла разрешается обращение к внешним компонентам. Очевидно, граф обменов волнового макроконвейера не должен иметь циклов, поскольку в противном случае неизбежны тупики. Наконец, в смешанном макроконвейере операторы обмена и обработки могут быть перемешаны произвольным образом. Рассмотрим каждый из видов статического макроконвейера более подробно.

Итеративный макроконвейер. Для этого вида макроконвейера можно достаточно просто вычислить коэффициенты ускорения и эффективности. Предположим, что рассматривается сеть, состоящая из p компонент. Для подсчета величин α_p и β_p следует различать два способа организации обменов – синхронный и асинхронный. Выбор того или иного способа обмена

зависит от операционной системы и возможностей аппаратуры. В случае синхронного обмена передача и прием данных начинаются лишь после того, как компоненты закончили обработку данных в очередном цикле, а обработка данных нового цикла начинается лишь после того, как закончились все обмены. При асинхронных обменах прием и передача в каждом из модулей начинается сразу после окончания обработки и выполняется в зависимости от ограничений на размеры очередей. Таким образом, в случае синхронного обмена макроконвейер работает как синхронизированная сеть с точками синхронизации перед началом и после окончания обменов между модулями.

Для синхронного обмена верхняя оценка эффективности имеет вид

$$\beta_p \geqslant \frac{\sigma}{u} \frac{1}{1+\lambda\frac{\upsilon}{u}}, \qquad (2.1)$$

где σ – среднее время обработки данных одной компонентой в одном цикле, u – максимальное время обработки данных одной компоненты в одном цикле, υ – максимальное время обмена между двумя компонентами по одной связи, усредненное по всем циклам, λ – хроматический класс графа M. Для того чтобы максимально распараллелить обмены между компонентами, их следует организовать следующим образом. Раскрасим дуги графа M так, чтобы все дуги, входящие в одну вершину и выходящие из нее, были раскрашены разными цветами. Хроматический класс λ – это минимальное число цветов, которое может понадобиться для такой раскраски. Упорядочим полученные цвета каким-либо образом и будем выполнять обмен так, что сначала выполняются параллельно все обмены, соответствующие дугам, выкрашенным в первый цвет, затем после окончания всех этих обменов выполняются обмены по дугам второго цвета и т.д. Для того чтобы реализовать такой обмен, операторы приема и передачи данных должны быть упорядочены в программе каждой компоненты в соответствии с нумерацией дуг, которую они получили при раскраске. Оценка (2.1) вычисляется без учета времени, потраченного на ввод и вывод. Это означает, что эффективность системы оценивается в период между окончанием ввода и началом вывода. Предполагается, что это время значительно больше, чем время ввода-вывода.

Для того чтобы убедиться в справедливости оценки (2.1), рассмотрим следующие величины:

t_{ij} – время, затрачиваемое на обработку i-й компонентой на j-м цикле ($i = 1, ..., p$, $j = 1, ..., N$);

N – общее число циклов (все величины зависят от исходных данных);

υ_{ijk} – время обмена данными i-й компоненты в j-м цикле по k-й дуге $i = 1, ..., p$, $j = 1, ..., N$, $k = 1, ..., \lambda$;

$$u = \frac{\sum_{j=1}^{N} \max_{1 \leqslant i \leqslant p} t_{ij}}{N}, \qquad \upsilon = \frac{\sum_{j=1}^{N} \max_{i,k} \upsilon_{ijk}}{N}, \qquad \sigma = \frac{\sum_{i=1}^{p} \sum_{j=1}^{N} t_{ij}}{Np}.$$

Если i-я вершина не имеет дуги, раскрашенной в k-й цвет, то $t_{ijk} = 0$. Теперь можно вычислить T_1 и T_p:

$$T_1 = \sum_{i=1}^{p} \sum_{j=1}^{N} t_{ij} = \sigma N p,$$

$$T_p = \sum_{j=1}^{N} \left(\max_i t_{ij} + \sum_{k=1}^{\lambda} \max_i v_{ijk} \right) \leqslant uN + \lambda v N,$$

откуда

$$\beta_p = \frac{T_1}{pT_p} \geqslant \frac{\sigma N p}{p(uN + \lambda v N)} = \frac{\sigma}{u} \frac{1}{1 + \lambda \dfrac{v}{u}}.$$

В качестве параметров σ, u и v можно рассматривать не времена, а объемы вычислений и передач, т.е. среднее и максимальное количества операций и максимальное количество единиц передаваемой информации. Тогда соответствующими временами будут $\sigma\tau_2$, $u\tau_2$, $v\tau_1$, где τ_1 — среднее время передачи единицы данных, τ_2 — среднее время выполнения одной операции. Формула (2.1) преобразуется в формулу

$$\beta_p \geqslant \frac{\sigma}{u} \frac{1}{1 + \lambda \dfrac{v}{u} \tau}, \qquad (2.2)$$

где $\tau = \tau_1/\tau_2$. Величины, входящие в формулу (2.1), имеют следующие названия:

σ/u — коэффициент равномерности;

v/u — коэффициент макроконвейерности;

λ — коэффициент локальности.

Коэффициент равномерности всегда находится между 0 и 1. При выборе алгоритма следует стремиться к тому, что бы он приближался к 1. Коэффициенты локальности и макроконвейерности должны быть по возможности меньше. Их уменьшение снижает влияние затрат на обмен.

При асинхронных обменах время T_p может уменьшиться за счет того, что при переходе к следующему циклу компоненты не должны ждать окончания цикла с максимальным временем выполнения, а начинают обмены сразу после окончания очередного шага обработки данных. Коэффициент эффективности при этом, вообще говоря, увеличивается. Количественно это может быть выражено оценкой

$$\beta_p \geqslant \frac{\sigma}{\nu} \frac{1}{1 + \lambda \dfrac{v}{\nu}}, \qquad (2.2)$$

где $\nu \leqslant u$. Отклонение величины ν от максимального времени обработки характеризует выигрыш, получаемый при переходе к асинхронному управлению обменами и может быть получено из следующих соображений. Если обозначить через T_{pi} время работы i-й компоненты, то для асинхронной

обработки имеет место:

$$T_{pi} = \sum_{j=1}^{N} (t_{ij} + w_{ij} + \sum_{k=1}^{\lambda} \upsilon_{ijk}),$$

где w_{ij} — общее время ожидания i-й компоненты на j-м цикле. Очевидно, что $T_p = \max_i T_{pi}$. Представим T_{pi} в виде $T_{pi} = (\nu_i + \lambda\upsilon)N$ и положим $\nu = \max_i \nu_i$. Тогда, поскольку $T_{pi} \leqslant (u + \lambda\upsilon)N$, имеет место $\nu_i \leqslant u$ и $\nu \leqslant u$. Величину ν_i можно выразить через средние величины

$$\nu_i = \sigma_i + w_i - \delta_i;$$

здесь $\sigma_i = (\sum_{j=1}^{N} t_{ij})/N$ — среднее время работы i-й компоненты, $w_i = (\sum_{j=1}^{N} w_{ij})/N$ — среднее ожидание i-й компоненты, а $\delta_i = \lambda\upsilon - \upsilon_i$, где $\upsilon_i = (\sum_{j=1}^{N} \sum_{k=1}^{\lambda} \upsilon_{ijk})/N$ — среднее время обменов i-й компоненты, т.е. δ_i есть отклонение среднего времени обмена i-й компоненты от среднего максимума. Из этого равенства можно получить нижнюю оценку для ν. Действительно, усреднив δ_i по всем компонентам и отбросив время ожидания (лучший случай), получим $\nu \leqslant \sigma - \delta$, где $\delta = \lambda\upsilon - (\sum_{i=1}^{p} \upsilon_i)/p$ — отклонение среднего времени обмена (по всем циклам и компонентам) от среднего максимума.

Волновой макроконвейер. Все компоненты волнового макроконвейера делятся на три типа — входные, выходные и внутренние. Входные компоненты те, которые принимают данные только из внешних каналов. Если все исходные данные уже находятся во входных очередях, то входные компоненты могут начинать работу с самого начала, не дожидаясь других компонент. Все остальные компоненты должны ждать момента передачи данных от смежных компонент. Выходные компоненты передают данные только во внешние выходные каналы. Внутренние — принимают от других компонент и передают другим компонентам. Характерной особенностью волнового макроконвейера является процесс распространения волны активности от входных компонент к выходным. Если $x_0, x_1, \ldots, x_k$ — путь в графе M, то компонента $K(x_{i+1})$, расположенная в точке x_{i+1}, не сможет начать i-й цикл обработки по крайней мере до тех пор, пока $K(x_i)$ не закончит этот цикл. В частности, любая выходная компонента сможет начать свою работу лишь после того, как активность распространится по всем путям, которые начинаются в точках расположения входных компонент и оканчиваются в точке расположения данной выходной компоненты.

Отрезок времени от начала работы макроконвейера до момента, когда начнут работать все выходные компоненты, называется периодом разгона. С другой стороны, первыми должны кончать свою работу входные компоненты. Переход от первого момента окончания работы какой либо из вход-

ных компонент до конца работы всего макроконвейера называется периодом торможения. Период от момента окончания разгона до начала торможения называется рабочим. Обычно при разумном использовании ресурсов многопроцессорной системы периоды разгона и торможения малы по сравнению с рабочим периодом. Поэтому при оценке эффективности работы волнового макроконвейера ими можно пренебречь и оценивать эффективность в рабочем периоде. Эта оценка тогда сведется к оценке эффективности итеративного макроконвейера соответствующими сдвигами номеров циклов: первым циклом компоненты K следует считать тот цикл, который она проходит в момент окончания разгона, а последним — цикл, соответствующий моменту начала торможения.

Смешанный макроконвейер характеризуется тем, что операторы обмена и обработки внутри основного цикла каждой компоненты перемешаны, но операторы обмена не могут находиться внутри циклов, содержащих операторы обработки или условные операторы. Это означает, что тело основного цикла представляет собой произведение операторов P_1 ... P_m, каждый из которых есть либо оператор обмена, либо оператор обработки данных. Фактически при большом числе процессоров большие группы обменов могут быть свернуты в циклы типа ДЛЯ i: = 1 ДО l ВЫПОЛНИТЬ ПЕРЕДАТЬ x В $K(x_i)$ КЦ, однако при фиксированном числе процессоров формально можно считать эти циклы развернутыми.

Операторы обработки данных, разумеется, могут быть сложными и содержать внутри себя как циклы, так и ветвления. Так же как и для других видов статического макроконвейера, требуется, чтобы две смежные компоненты были согласованы по обменам, но количество обменов по одной дуге может быть произвольным. Поэтому согласованность по обменам представляет собой более сложное условие: если в компоненте K содержится последовательность передач в K', то в K' должна быть соответствующая последовательность операторов приема, согласованная также по типам данных, которыми обмениваются компоненты K и K'. Для того чтобы последовательность обменов в каждом цикле была одной и той же, потребуем также, чтобы выход из цикла был возможен лишь из операторов P_1 и P_m. В частности, основной цикл может иметь вид ПОКА... или ДЛЯ... .

Необходимость перемешивания операторов обмена и обработки может объясняться различными причинами. Первая причина состоит в желании сократить ожидания за счет выгодного расположения операторов обмена. При наличии соответствующей поддержки со стороны операционной системы выгодно операторы передачи данных располагать так, чтобы они выполнялись как можно раньше, т.е. сразу после того, как получены данные, которые передаются. С другой стороны, операторы приема выгодно помещать как можно позже, т.е. непосредственно перед использованием данных, которые принимаются (по крайней мере, в предположении, что размеры очередей не ограничены). Другой причиной смешанного расположения операторов обмена может служить природа самого алгоритма.

В смешанном макроконвейере возможны тупиковые состояния. В общем случае при произвольных структурах алгоритмов функционирования компонент задача обнаружения тупиков трудна и общего решения не имеет. Однако ограничения, наложенные на алгоритмы функционирования компо-

нент смешанного макроконвейера, позволяют решить задачу обнаружения тупиков формальным образом. Следует различать тупики по вычислениям и тупики по обменам. Тупиковое состояние сети является тупиковым по обменам, если каждая компонента находится либо в заключительном состоянии, либо в состоянии ожидания завершения обмена. Для простых асинхронных сетей, если базовые операторы и условия всюду определены, то тупиковые состояния исчерпываются тупиками по обменам. В дальнейшем будем рассматривать только всюду определенные базисы и тупики по обменам. Поскольку в каждом цикле каждая компонента порождает одну и ту же последовательность операторов обмена, для того чтобы убедиться в отсутствии тупиков, достаточно проверить, что все операторы обмена, порожденные компонентами в цикле с одним и тем же номером, завершат свое выполнение в течение этого цикла. Условие завершения всех обменов является также необходимым, поскольку требуется согласованность по обменам.

Для обнаружения тупиков может быть полезной следующая модель смешанного макроконвейера. Рассмотрим смешанную макроконвейерную сеть S, расположенную в коммуникационном пространстве M, содержащем p точек. Заменим программы компонент сети S следующим образом. Отбросим операторы ввода данных и вывода результатов, отбросим начало и конец цикла, а в теле цикла оставим только операторы обмена между компонентами, упростив их следующим образом: оператор ПЕРЕДАТЬ x В K заменим на ПЕРЕДАТЬ 1 В K (вместо числа 1 можно было бы использовать любую другую константу), а оператор ПРИНЯТЬ y ИЗ K заменим на ПРИНЯТЬ a ИЗ K, где a — простая переменная типа "целое" (одна и та же для всех операторов приема). Полученная сеть $\overline{S}$ имеет конечное число состояний, и отсутствие тупиковых состояний в сети $\overline{S}$ (которое легко проверяется простым перебором) является необходимым и достаточным условием отсутствия тупиков сети S при условии неограниченных очередей. Если размеры очередей ограничены, то для обнаружения тупиков необходимо наложить соответствующие ограничения на множество допустимых последовательностей.

Модель $\overline{S}$ полезна также для изучения распространения волн активности в сети S. Эти волны могут распространяться более сложным образом, чем в волновом или итеративном макроконвейере. В частности, некоторые компоненты могут переходить из состояния ожидания в состояние активности несколько раз. Состояние активности компоненты сети $\overline{S}$ — это состояние, отличное от состояния ожидания. Для изучения активности в программы компонент следует добавить промежуточные состояния, соответствующие выброшенным операторам обработки, и при вычислении переходов не задерживать выполнимые переходы, соответствующие операторам обмена. Более того, если в некоторой компоненте выполняется оператор передачи, а в смежной — соответствующий этой передаче оператор приема при пустой очереди, то соответствующий обмен следует выполнять не за два, а за один такт. Переходы же, соответствующие операторам обработки, следует задерживать до окончания всех обменов. Если при таких условиях в каждом состоянии сети состояния всех компонент активны, то сеть принадлежит к итеративному типу; если же каждая ком-

понента переходит из состояния ожидания в активное состояние только один раз в течение цикла, то сеть принадлежит к волновому типу.

Рассмотрим несколько примеров. Наиболее ясные примеры статического макроконвейера дают итерационные методы решения уравнений математической физики. Представляя область интегрирования в виде области определения структуры данных x, расположенной на целочисленной решетке Z^m, получим, что алгоритм решения такой задачи сводится к многократному выполнению оператора присваивания $x := F(x)$, где F – функция над структурой данных x. Для параллельного вычисления функции F область определения структуры данных x разбивается на части и распределяется между компонентами статического макроконвейера. Связи между компонентами определяются зависимостями, которые порождаются сдвигами, используемыми при определении периодически определенной функции F. Для явных разностных схем типичными являются зависимости вида

$$x_{ij}^{(n+1)} = f(x_{i+\alpha_1 j+\beta_1}^{(n)}, \ldots, x_{i+\alpha_k j+\beta_k}^{(n)}),$$

где α_m, β_m – целые числа, малые по абсолютной величине. Для примера рассматривается двумерная область. Функция f и наборы сдвигов (α_m, β_m) могут быть различными для различных точек (i, j) области расположения структуры данных x. Предположим, например, что областью определения структуры данных x является прямоугольник $[1: m, 1: n]$, а сдвиги (α, β) по модулю не превосходят l. Тогда эту область можно распределить по процессорам, разбив на вертикальные (если $n \geqslant m$) или горизонтальные (если $m \geqslant n$) полосы. Граф обменов в этом случае будет иметь вид линии, в которой каждая вершина, кроме крайних, связана в обе стороны с двумя соседними. Коэффициент локальности λ равен в этом случае 4. Предположим, что область разбита на вертикальные полосы размером $m \times k$ и $n = pk$ (p – число процессоров). Естественно предположить, что среднее и максимальное числа операций, приходящиеся на одну компоненту, пропорциональны размеру области, выделенной для этой компоненты, т.е. $\sigma = amk\tau_2$, $u = bmk\tau_2$ (τ_2 – среднее время выполнения одной операции). Подставляя соответствующие значения в формулу (2.1), получим оценку

$$\beta_p \geqslant \frac{a}{b} \frac{1}{1 + 4\dfrac{lp}{bn}\tau}.$$

Из этой оценки легко получить оценку числа процессоров, которое возможно использовать при коэффициенте эффективности не ниже заданного уровня, скажем γ. Для того чтобы выполнялось условие $\beta_p \geqslant \gamma$, достаточно выполнения неравенства

$$p \leqslant \frac{n}{4l\tau} \frac{a - b\gamma}{\gamma}.$$

При $a = b$, т.е. при равномерном распределении работы между процессорами, получим условие

$$p \leqslant \frac{an}{4l\tau} \frac{1 - \gamma}{\gamma}.$$

При неравномерном распределении вычислений синхронная организация обменов не может дать коэффициент эффективности больше, чем a/b. Для того чтобы использовать возможности асинхронного взаимодействия, целесообразно перейти к схеме смешанного макроконвейера, организовав передачу данных по частям, например, следующим образом. Разобъем каждую из вертикальных полос на горизонтальные полосы, скажем на квадраты со стороной длины k, и будем в одной компоненте обрабатывать эти квадраты сверху вниз. Обработав очередной квадрат, передадим крайние l вертикалей этого квадрата соседям. Тогда, если вычисления в некоторой полосе на некоторой итерации закончились раньше, чем в соседних, можно будет начинать следующую итерацию, не дожидаясь соседей. Такая организация работы в некоторых случаях может приблизить ситуацию к случаю равномерного распределения вычислений.

Предположим, что $l = 2$, $\tau = 8$ (соотношение скоростей: 1 млн операций в секунду и 1 млн байт/с при обменах). Тогда при равномерном распределении вычислений для получения коэффициента эффективности 0.9 число процессоров не должно превосходить $an/576$. Таким образом, при $a = 10$, $n = 1000$ коэффициент 0.9 можно получить лишь при 17 процессорах (ускорение не менее чем в 15 раз). Существенное увеличение возможностей параллельной обработки можно получить, если разбивать прямоугольную область не на полосы, а на прямоугольники (лучше всего квадраты). Предположим для примера, что область $[1{:}\ n, 1{:}\ n]$ разбита на квадраты со стороной k, $n = kq$, $p = q^2$. Если в сдвигах (α, β) всегда либо $\alpha = 0$, либо $\beta = 0$, то каждая компонента, обрабатывающая один из p квадратов должна обмениваться не более чем с четырьмя соседями и $\lambda = 8$. Оценка (в равномерном случае) для числа процессоров, которые можно использовать для получения эффективности γ, получается из неравенства

$$q \leqslant \frac{an}{8l\tau} \cdot \frac{1-\gamma}{\gamma}.$$

Теперь при $l = 2$, $a = 10$, $n = 1000$ имеем, что коэффициент 0.9 можно получить с использованием 75 процессоров; ускорение получится не менее чем в 67 раз. Если же ограничимся коэффициентом 0.8, то можно будет на 381 процессоре получить ускорение в 304 раза.

Неявные разностные схемы различаются тем, что зависимости в схемах счета могут иметь такой вид:

$$x_{ij}^{(n+1)} = f(x_{i+\alpha j+\beta}^{(n)}, \ldots, x_{i+\gamma j+\delta}^{(n+1)}),$$

т.е. значение в точке (i, j) структуры данных x (а таких структур данных может быть и несколько) зависит от значений других структур в соседних точках, взятых не только на предыдущей, но также и на текущей итерации. Такие схемы естественно приводят к волновому макроконвейеру. Так же как и в случае итеративного макроконвейера, разбиение двумерной области на вертикальные (горизонтальные) полосы или прямоугольные блоки обеспечивает хорошую эффективность при достаточно высоком коэффициенте однородности σ/u. Явные методы решения систем линейных алгебраических уравнений большой размерности так же, как и вычисление других структурных функций с широким фронтом волны вычислений, могут быть сведены к статическому макроконвейеру. Естественным образом

к статическому макроконвейеру сводятся задачи моделирования сложных систем (дискретных или непрерывных). В этом случае сложную систему представляют в виде композиции подсистем, имеющих определенное функциональное назначение (в содержательном представлении), и располагают эти подсистемы в отдельных компонентах макроконвейера. Такой подход позволяет уменьшать количество связей между компонентами и, следовательно, важный коэффициент локальности λ.

Реальные практические задачи далеко не всегда в чистом виде могут быть выражены макроконвейерной схемой. Часто бывает, что на разных этапах решения задачи используются различные схемы. Например, при расчете сложной конструкции ее отдельные части могут рассчитываться разными методами; собственно расчету может предшествовать предварительная обработка или подготовка данных. В таких случаях возникает необходимость использования динамического макроконвейера. Одним из способов организации динамического макроконвейера будет управляемая сеть из алгоритмических модулей. Программа управляющей компоненты такой сети инициирует развертывание статических макроконвейерных структур на различных этапах обработки и организует взаимодействие между ними. Две статические макроконвейерные сети могут быть связаны информационно. Например, данные которые вырабатываются сетью Q_1, могут служить исходными данными для сети Q_2. Сеть Q_2 в этом случае может начинать свою работу еще до окончания работы сети Q_1 (работа с перекрытием). Особенно большой эффект получается, если обе сети работают по схеме волнового макроконвейера. Тогда в период торможения первого макроконвейера постепенно освобождающиеся процессоры передаются второму и включаются в работу по разгону второго макроконвейера.

Макроконвейерные сети с переменной структурой возникают как естественные модели, описывающие рекурсивные вычислительные процессы над теоретико-множественными структурами данных. Задачи, требующие большого перебора и обработки больших объемов неоднородной информации, также приводят к динамическим макроконвейерным сетям.

§ 3. Проектирование распределенных программ

Распределенные программы — это программы, предназначенные для выполнения на распределенной многопроцессорной системе. Основной особенностью распределенного программирования является необходимость учета особенностей структуры вычислительной системы уже на первых этапах проектирования распределенных программ. При этом следует учитывать не только структуру технических средств системы, но также и поддержку программных средств операционной системы и системы параллельного программирования. Другой особенностью распределенного параллельного программирования является необходимость в ряде случаев разрабатывать структурные модели программ, т.е. оперировать с многоуровневыми, многокомпонентными системами. Программы, учитывающие структуру распределенной многопроцессорной системы, семантика которых определяется в терминах сетей из алгоритмических модулей, будем называть мультимодульными программами (например, программы в языке типа А3, § 6 гл. 2).

Основные этапы проектирования распределенных программ следующие.

1. Разработка главной функциональной модели.
2. Преобразование функциональных моделей.
3. Разработка последовательной процедурной многоуровневой модели.
4. Разработка главной структурной модели и функциональных моделей компонент.
5. Разработка процедурных моделей компонент.
6. Преобразование структурных моделей.
7. Преобразование процедурных моделей.

Разработка главной функциональной модели, ее конструктивизация и преобразования, выполняемые на уровне предметной области, ничем не отличаются от соответствующих этапов проектирования последовательных программ, и мы объединяем эти этапы в один этап разработки главной функциональной модели и ее составляющих. Второй этап проектирования – преобразование функциональных моделей – имеет в виду преобразования, ориентированные на учет структуры распределенной системы и ее общесистемного программного обеспечения. В основном эти преобразования связаны с реализацией структуры данных предметной области структурами данных, удобными для распределенной обработки.

Этап разработки последовательной многоуровневой модели является альтернативным и может быть пропущен, если главная структурная модель разрабатывается непосредственно по функциональным моделям. Последовательная модель может возникнуть в одном из двух случаев. В первом случае последовательная модель используется в качестве промежуточной для получения параллельной структурной модели, во втором – как объект для динамического распараллеливания в процессе выполнения средствами операционной системы. В двухуровневой модели базовые операторы представляют собой обращения к программам (подпрограммам), которые могут выполняться на различных процессорах распределенной системы. Разумеется, эти программы должны удовлетворять принципу макроконвейера, т.е. каждая из них может загрузить процессор на достаточно большое время. Поэтому такие программы должны работать с большими порциями данных. Для того чтобы иметь возможность получать такие порции, все данные, обрабатываемые в системе, должны быть представлены как многоуровневые структуры данных, т.е. структуры данных, составленные из структур данных более низкого уровня. Если программа, вызываемая базовым оператором, сама может выполняться на нескольких процессорах, программы становятся многоуровневыми.

Главная структурная модель представляется как мультимодульная программа, описывающая функционирование статической или динамической макроконвейерной сети. Алгоритмы функционирования компонент или их части могут быть при этом представлены своими функциональными моделями и уточняться на следующих этапах проектирования. В этом случае в качестве отдельного этапа может выступать этап разработки процедурных моделей для компонент (модулей мультимодульной программы). Этапы преобразования структурных и процедурных моделей могут чередоваться. Их целью является улучшение тех или иных характеристик мультимодульной программы или учет все более конкретных особенностей вычислительной системы, на которой будет выполняться программа.

Рассмотрим некоторый набор языковых средств, полезных при разработке распределенных программ. Будем следовать при этом структуре языка МАЯК, разработанного Институтом кибернетики имени В.М. Глушкова АН УССР для макроконвейерного вычислительного комплекса. Мы не останавливаемся на средствах последовательной обработки, которые могут быть взяты из любого распространенного языка программирования общего назначения. В частности, МАЯК в этом плане следует традициям языка ПАСКАЛЬ с русскими ключевыми словами. Язык имеет три основных уровня представления программ, каждый из которых поддерживается различными средствами операционной системы и системы программирования.

Первый уровень языка ЯДРО составляет наименьшее подмножество и ориентирован на наиболее высокий уровень автоматизации программирования и выполнения программ, включая динамическое распараллеливание. ЯДРО-программа представляет собой систему последовательных программ (возможно, рекурсивных), каждая из которых может выполняться на отдельном процессоре. Все программы системы объединены программой верхнего уровня и обращаются друг к другу обычными операторами вызова ВЫЗВАТЬ $P(x)$ РЕЗУЛЬТАТ (y). Однако в отличие от последовательной семантики, описанной в § 5 гл. 3 предполагается возможность выполнения каждого вызова параллельно с основной программой в другом процессоре. При этом решение о параллельном или последовательном выполнении принимает операционная система в зависимости от наличия ресурсов и организации управляющих программ. Операционная система обеспечивает также взаимодействие и синхронизацию параллельных ветвей. Именно, после того, как инициировано параллельное выполнение программы, основная программа продолжает работать. Однако это продолжение возможно лишь до тех пор, пока основной программе не потребуется использовать результат работы вызванной программы. Если к этому моменту результат еще не готов, операционная система должна обеспечить ожидание результата. Программы работают с двухуровневой памятью. Память первой ступени является распределенной, т.е. программы не имеют общих переменных и работают только с переменными, локализованными в них, обмениваясь информацией через фактические параметры. Память второго уровня является общей для всех программ. Компоненты этой памяти называются внешними структурами данных. В языке МАЯК в качестве внешних структур данных используются внешние массивы, описание которых имеют следующий вид (пример):

ВНЕШНИЙ МАССИВ $A(m_1 : n_1, \ldots, m_k : n_k)$ БЛОКОВ ТИПА МАССИВ $(r_1 : s_1, \ldots, r_l : s_l)$ ВЕЩ

Такое описание определяет k-мерный внешний массив, элементами которого являются l-мерные массивы вещественных чисел. В отличие от памяти первой ступени, к элементам которой можно обращаться непосредственно в операторах присваивания, к внешней памяти можно обращаться только с помощью специальных операторов обмена:

ПРОЧИТАТЬ $A(i_1, \ldots, i_k)$ В X;

ЗАПИСАТЬ X В $A(i_1, \ldots, i_k)$.

Эти операторы обеспечивают обмен между двумя ступенями памяти. Опе-

ратор ПРОЧИТАТЬ передает блок внешнего массива внутренней переменной соответствующего типа, а оператор ЗАПИСАТЬ передает значение внутренней переменной в соответствующий блок внешнего массива.

При работе параллельных ветвей, порожденных вызовами программ, возможны конфликты, вызванные одновременным обращением к общей внешней памяти, и недетерминированность, вызванная порядком обращения различных ветвей к одному и тому же блоку внешнего массива. Соответствующую синхронизацию и управление обменами с памятью второй ступени также может взять на себя достаточно совершенная операционная система. Способ управления обменами определяется следующим принципом. Каждая ЯДРО-программа определяет некоторый последовательный алгоритм, который получается, если все вызовы выполнять последовательно. Параллельное выполнение ЯДРО-программы должно обеспечить эквивалентность результата тому, который получается при последовательном выполнении. При использовании более слабой операционной системы программист может взять на себя синхронизацию обменов. Для этого в языке МАЯК предусмотрены операторы

ЗАНЯТЬ $A(i_1, \ldots, i_k)$

ОСВОБОДИТЬ $A(i_1, \ldots, i_k)$,

которые позволяют организовать защиту отдельных блоков и целых массивов. После выполнения оператора ЗАНЯТЬ соответствующий блок будет доступен лишь программе, выполнившей этот оператор, и всем программам, которые она вызывает. После освобождения блок будет доступен всем.

В языке ЯДРО используется еще одно полезное средство синхронизации — параллельные составные операторы вида ПАРНАЧАЛО $P_1; \ldots; P_m$ ПАРКОНЕЦ. Такой оператор выполняется как обычный последовательный составной оператор, но выход из этого оператора (через ПАРКОНЕЦ) задерживается до тех пор, пока не завершатся все параллельные процессы, порожденные вызовами программ, выполненными после входа в этот оператор (через ПАРНАЧАЛО).

В качестве примера ЯДРО-программы рассмотрим умножение блочных матриц, представленных в виде внешних массивов с описаниями:

ВНЕШНИЕ МАССИВЫ А, В, С (1 : К, 1 : К) БЛОКОВ ТИПА МАТРИЦА (N) ВЕЩ.

Программа верхнего уровня:

```
ПРОГРАММА УМН МАТР (А, В, С : ИМЕНА ВНЕШ МАС, К, N; ЦЕЛ);
НАЧАЛО
    ЗАНЯТЬ А, В, С;
    ДЛЯ I, J : = 1 ДО К ВЫПОЛНИТЬ
      ВЫЗВАТЬ УМН БЛОКОВ (А, В, С, I, J, N);
    КЦ
    ОСВОБОДИТЬ А, В, С,
КОНЕЦ.
```

Подчиненная программа:

```
ПРОГРАММА УМН БЛОКОВ (X, Y, Z: ИМЕНА ВНЕШ МАС, I, J, N: ЦЕЛ);
ИМЕНА X1, Y1, Z1: МАТРИЦЫ (N) ВЕЩ;
```

```
НАЧАЛО.
   ДЛЯ К: = 1 ДО N ВЫП
     ПРОЧИТАТЬ X (1, К) В X1;
     ПРОЧИТАТЬ Y (К, J) В Y1;
     ЕСЛИ К = 1 ТО Z1: = X1 * Y1 ИНАЧЕ Z1: = Z1 + X1 *. Y1 КЕ
   КЦ;
   ЗАПИСАТЬ Z1 В Z (I, J)
КОНЕЦ.
```

Второй уровень языка МАЯК – это язык простых мультимодульных программ ПРОСТОР. Он представляет возможность программисту самому осуществить статическое распределение ресурсов вычислительной системы, явно указать необходимое число процессоров и организовать их взаимодействие в процессе решения задачи. При отсутствии параллельных вызовов простая мультимодульная программа эквивалентна простой асинхронной сети из алгоритмических модулей. Поэтому язык ПРОСТОР особенно удобен для описания статических макроконвейерных сетей. Общий вид ПРОСТОР-программы соответствует схеме языка А3 (с другими ключевыми словами). В качестве примера ПРОСТОР-программы рассмотрим снова алгоритм умножения блочных матриц.

```
ПРОГРАММА УМН МАТР П (А, В, С: ИМЕНА ВНЕШ МАС, К, N: ЦЕЛ);
КОМПОНЕНТЫ ЦЕПЬ (1 : К);
        КООРДИНАТА Q; ИСПОЛЬЗУЕТ К, N;
        ИМЕНА А1, В1, С1: МАТРИЦЫ (N) ВЕЩ;
НАЧАЛО.
      ДЛЯ I: = 1 ДО К ВЫПОЛНИТЬ
          ПРОЧИТАТЬ А(I, Q) В А1;
          ДЛЯ J: = 1 ДО К ВЫПОЛНИТЬ
               ПРОЧИТАТЬ В (Q, J) В В1;
               ЕСЛИ Q > 1 ТО
                    ПРИНЯТЬ ДАННЫЕ С1 ИЗ ЦЕПЬ (Q – 1);
                    С1: = С1 + А1 * В1;
               ИНАЧЕ     С1: = А1 * В1 КЕ;
               ЕСЛИ Q < К ТО
                    ПЕРЕДАТЬ ДАННЫЕ С1 В ЦЕПЬ (Q + 1);
               ИНАЧЕ ЗАПИСАТЬ С1 В С (I, J) КЕ
          КЦ
     КЦ
КОНЕЦ.
```

Программа УМН МАТР П описывает статический волновой макроконвейер, состоящий из К компонент, расположенных в виде одномерной цепи ЦЕПЬ (1), . . . , ЦЕПЬ (К). Для того чтобы в этом убедиться, достаточно двойной цикл по I и J преобразовать в одинарный и для каждой конкретной координаты Q компоненты макроконвейера снять условные операторы. Так, если, например, $1 < Q < K$, то программа компоненты имеет вид

```
НАЧАЛО.
    I: = 1, J: = 1;
    ЦИКЛ
      ЕСЛИ J = 1 то ПРОЧИТАТЬ A (I, Q) В A1 КЕ;
      ПРОЧИТАТЬ . . .
      ПРИНЯТЬ . . .
      . . .
      ПЕРЕДАТЬ . . .
      ЕСЛИ J > К то J: = J + 1 ИНАЧЕ
      ЕСЛИ I < К ТО I: = I + 1, J: = 1
      ИНАЧЕ ВЫЙТИ КЕ
    КЦ
КОНЕЦ.
```

В общем случае ПРОСТОР-программа состоит из заголовка и описаний инициализированных компонент, т.е. компонент вместе с программами их работы. Каждая компонента имеет имя, которое может снабжатся индексами, указывающими расположение компоненты в одной из точек целочисленной решетки соответствующей размерности. Для того чтобы вызвать ПРОСТОР-программу, в системе должно быть необходимое количество свободных в момент вызова процессоров. Если свободные ресурсы есть, то они распределяются по компонентам, в каждый из выделенных процессоров загружается программа соответствующей компоненты и система начинает работать. При загрузке компонентам сообщаются их координаты, если они образуют массивы.

В отличие от ЯДРО-программы УМН МАТР, которая работает в системе с произвольным числом свободных процессоров от 1 до К, программа УМН МАТР П может работать лишь тогда, когда система имеет не менее, чем К, свободных процессоров. Преимуществом ПРОСТОР-программы является, во-первых, отсутствие накладных расходов на повторный вызов программ и, во-вторых, некоторое снижение времени на обмены. Действительно, если предположить, что скорость обмена между процессорами выше, чем скорость обмена с внешними массивами, то выигрыш произойдет за счет того, что блоки из одной строки будут считываться каждым процессором лишь по одному разу. Впрочем, при хорошей операционной системе и благоприятных условиях эти преимущества могут быть небольшими. В частности, операционная система может сохранять программу, записанную ранее в некоторый процессор после его освобождения, и не перезагружать эту программу, если она уже загружена в процессор, которому дается новое задание. Операционная система может также использовать в качестве виртуальной памяти для хранения внешних массивов часть распределенной основной памяти процессоров. В этом случае обращение к внешним массивам может свестись к обменам между процессорами.

Таким образом, основные средства языка ПРОСТОР – описание компонент вместе с их программами, а средства для взаимодействия компонент – операторы

```
ПРИНЯТЬ ДАННЫЕ Х ИЗ К
ПЕРЕДАТЬ ДАННЫЕ Х В К
```

выполняемые как обращения к асинхронным очередям. Эти средства могут сочетаться со средствами ЯДРА. Тогда получаются более сложные структуры, в частности некоторые разновидности динамических макроконвейерных сетей. Сочетание ЯДРА и ПРОСТОРА позволяет описывать многоуровневые структуры, которые на одних уровнях описываются статическими, на других – динамическими моделями. Важной особенностью языка ПРОСТОР, отличающей его от ЯДРА (без синхронизации), является возможность тупиковых состояний в ПРОСТОР-программах, вызванных несогласованностью обменов. Для того чтобы их избегать, следует пользоваться хорошо обоснованными общими схемами типа макроконвейерных сетей либо проводить тщательное исследование конкретной программы с целью ее обоснования (отсутствие тупиков, оценка эффективности).

Следующий уровень языка МАЯК – это язык ММП (язык мультимодульных программ). Язык ММП обладает достаточно полным набором средств параллельного программирования. Он позволяет программисту взять на себя не только статическое, но и динамическое управление ресурсами. Для этой цели в языке допускаются описания неинициализированных компонент:

КОМПОНЕНТЫ $К_1, \ldots, К_m$ КОНЕЦ КОМП;
КОМПОНЕНТЫ К$(m_1 : n_1, \ldots, m_k : n_k)$ КОНЕЦ КОМП.

Если К – неинициализированная компонента, то программа, в которой действует ее описание, может вызвать в этой компоненте некоторую программу с помощью оператора управляемого вызова:

ВЫЗВАТЬ $P(x)$ В К;
ВЫЗВАТЬ $P(x)$ РЕЗУЛЬТАТ (y) В К.

Неинициализированные компоненты ММП-программы соответствуют процессорам распределений многопроцессорной системы. Соответствие между компонентами программы и процессорами может изменяться в процессе работы и, как правило, сохраняется в период выполнения программы. Для выполнения вызова система выделяет свободный процессор, устанавливает соответствие между этим процессором и компонентой, вызывает, если нужно, программу P, передает ей фактические параметры и инициализирует ее выполнение. Поскольку вызвавшей программе известно имя компоненты, в которой выполняется вызванная программа, обе компоненты могут теперь не только взаимодействовать через фактические параметры, но и обмениваться данными. Аналогично две программы, вызванные в различных компонентах, могут также взаимодействовать между собой, обмениваясь данными.

В языке ММП кроме обмена данными допускаются еще два вида взаимодействия – обмен сообщениями и прерывания. Сообщения различаются по типам. Сообщения одного и того же типа представляют собой структуры данных, принадлежащих одному и тому же типу данных. Иными словами, если типы сообщений совпадают, то совпадают и типы значений этих сообщений (но не наоборот). Определение типов сообщений и их значений выполняется в ММП с помощью описаний. Примером описания сообщений может служить следующее описание:

СТРУКТУРА СООБЩЕНИЙ,
СБОЙ →,
НОМЕР → ЦЕЛОЕ,
КОЛИЧЕСТВО → ЦЕЛОЕ,
СДВИГ → СТРУКТУРА (КУДА : ЦЕЛ, ДАЛЬШЕ :
ИМЯ КОМПОНЕНТЫ),
ГОТОВНОСТЬ → СТРУКТУРА(А : ЦЕЛ, В : СТРУКТ(С, D : ВЕЩ)).

В этом описании представлено пять типов сообщений. Первый тип (СБОЙ) не имеет значения, и сам тип несет всю передаваемую сообщением информацию. Значениями сообщений типа НОМЕР и КОЛИЧЕСТВО являются целые числа, а значения сообщений типа СДВИГ и ГОТОВНОСТЬ являются структурами (записями), каждая из которых состоит из двух полей: первая из полей КУДА типа "целое" и ДАЛЬШЕ типа "имя компоненты" и т.д.

Сообщения передаются с помощью оператора

ПЕРЕДАТЬ СООБЩЕНИЕ $t(x)$ В К,

где t — тип сообщения, x — значение сообщения, K — имя компоненты. Сообщения так же, как и данные, передаются асинхронно, однако в отличие от данных все сообщения, которые приходят к одной и той же компоненте, выстраиваются в одну очередь независимо от места отправления. Доступ к сообщениям осуществляется с помощью оператора обработки сообщения, который имеет следующий вид:

ОБРАБОТАТЬ СООБЩЕНИЕ:
$t_1(x_1) \to P_1$;
. . . . . .
$t_m(x_m) \to P_m$;
$\to P_{m+1}$
КОНЕЦ ОБРАБОТКИ

Выполняется этот оператор следующим образом. В очереди сообщений отыскивается первое сообщение одного из типов $t_1, \ldots, t_m$. Если такое сообщение есть и оно имеет тип t_i, то значение сообщения присваивается имени x_i и выполняется оператор P_i. Если такого сообщения нет, то выполняется оператор P_{m+1}. Последняя альтернатива ($\to P_{m+1}$) может отсутствовать. В этом случае оператор обработки ждет поступления нужного сообщения. Наконец, в заголовке оператора могут быть узказаны имена компонент:

ОБРАБОТАТЬ СООБЩЕНИЕ ОТ $K_1, \ldots, K_m$:
. . .

В этом случае для обработки выбирается сообщение, посланное одной из компонент $K_1, \ldots, K_m$. Вместо одного имени в левой части альтернативы может быть указано несколько имен. Эти имена соответствуют полям сообщения, если значение сообщения есть структура. Сообщения могут передаваться с прерываниями. Для этого используется оператор:

ПЕРЕДАТЬ СООБЩЕНИЕ $t(x)$ С ПРЕРЫВАНИЕМ В К.

Для обработки сообщений с прерыванием каждая программа может иметь в своем составе программу прерывания, состоящую из одного оператора

обработки сообщений. Эта программа помещается среди описаний информационной среды и имеет вид

ПРЕРЫВАНИЕ

ОБРАБОТАТЬ СООБЩЕНИЕ:

$t_1 \to P_1$;

...

$t_m \to P_m$;

$\to P_{m+1}$

КОНЕЦ ОБРАБОТКИ

При передаче сообщения с прерыванием компоненте K эта компонента прерывает свою работу, выполняет программу прерывания, которая обрабатывает посланное сообщение, и возобновляет свою работу с прерванного места.

Средства языка ММП могут быть реализованы многими различными способами с использованием базовых понятий взаимодействия параллельных процессов в сетях из алгоритмических модулей. С другой стороны, в терминах сообщений, прерываний и управляемых вызовов могут быть реализованы многие другие способы взаимодействия компонент. Так, например, если необходимо использовать обмен информацией между двумя компонентами по каналу z, который является выходным для компоненты K и входным для компоненты K', достаточно ввести тип сообщения z и вместо оператора ПЕРЕДАТЬ x в z использовать оператор ПЕРЕДАТЬ СООБЩ $z(x)$ в K', а в компоненте K' вместо оператора ПРИНЯТЬ y из z использовать оператор ОБРАБОТАТЬ СООБЩ $z(y) \to$ КОНЕЦ ОБРАБОТКИ. Основное назначение языка ММП – программирование операционной системы распределенной многопроцессорной ЭВМ и разработка других компонент общесистемного математического обеспечения, пакетов прикладных программ и других сложных алгоритмов преобразования информации с помощью многопроцессорных ЭВМ.

§ 4. Синтез макроконвейерных программ вычисления функций над структурами данных

Пусть $y = f(x)$ – система периодически определенных функций над двухуровневыми структурами данных, $y = (y_1, \ldots, y_m)$, $x = (x_1, \ldots, x_n)$. Это значит, что структуры данных $x_1, \ldots, x_n, y_1, \ldots, y_m$ содержатся в множестве $\Gamma(C', \Gamma(C'', D))$. Рассматривая $D' = \Gamma(C'', D)$ как базовую алгебру данных, можем синтезировать последовательную программу

ПРОГРАММА $A(x, H)$ РЕЗУЛЬТАТ (y),

которая вычисляет функции $y_i = f_i(x)$ $(i = 1, \ldots, m)$ на множествах $H_1, \ldots, H_m \subset C$, пользуясь одним из методов, описанных в § 3 гл. 6.

В программе A в качестве базовых операторов используются операторы присваивания вида

$$z(c) := \varphi(z_1(cg_1), \ldots, z_k(cg_k), c), \tag{4.1}$$

где $z, z_1, \ldots, z_k \in X = \{x_1, \ldots, x_n, y_1, \ldots, y_m\}$, c – выражение, принимающее значения в C', φ – функция над структурами данных, расположенными на C'' и принимающими значения в D. Поскольку области определения структур данных $z_i(cg_i)$ могут быть достаточно большими и выпол-

нение оператора (4.1) в этом случае может загрузить отдельный процессор на значительное время, целесообразно каждый из операторов вида (4.1) реализовать отдельной программой, предназначенной для выполнения на одном процессоре. Заменив присваивания вызовами соответствующих программ, получим макроконвейерную программу в языке ЯДРО, которая может выполняться с автоматическим распараллеливанием при поддержке соответствующей операционной системы. Структуры данных $z(c)$, $z \in X$, $c \in C'$ могут храниться как блоки внешних массивов. Если, например, $C' = Z^l$, то $z(c)$ может храниться как блок внешнего массива, имеющего описание

ВНЕШНИЙ МАССИВ $z(m_1 : m_1', \ldots, m_l : m_l')$ БЛОКОВ ТИПА $\Gamma(C'', D)$

Разумеется, тип $\Gamma(C'', D)$ должен быть представлен подходящими структурами данных используемого языка программирования. Поэтому в качестве фактического параметра программы выполнения присваивания (4.1) может передаваться только координата c, которая определяет также точки $cg_1, \ldots, cg_k$ и, возможно, имена внешних массивов, а также другие необходимые параметры. Границы внешнего массива z выбираются, исходя из возможных значений координат $c, cg_1, \ldots, cg_k$.

Если правая часть присваивания (4.1) (при фиксированном c) является периодически определенной функцией, то программа выполнения присваивания также может быть синтезирована методами § 3 гл. 6.

Если обозначить $u_i = z_i(cg_i)$ и привести φ к канонической форме, получим

$$\varphi(u_1, \ldots, u_k) = \bigsqcup_{i=1}^{q} \varphi_i(v_1^{h_1}, \ldots, v_l^{h_l}, c)/F_i,$$

где $v_1, \ldots, v_l \in \{u_1, \ldots, u_k\}$, $h_1, \ldots, h_l$ — сдвиги, определенные уже на C'', $F_i \subset C''$, φ_i — базовые операции алгебры D. Если найдутся такие множества F_i', что $F_i = F_i'/h_i$, то

$$\varphi(u_1, \ldots, u_k) = \bigsqcup_{i=1}^{q} \varphi_i((v_1/F_i')^{h_1}, \ldots, (v_l/F_i')^{h_l}, c).$$

Вырезки v_i/F_i' показывают, что для программы выполнения (4.1) могут понадобиться не все блоки $z_i(cg_i)$, а лишь некоторые их части. Поэтому для уменьшения объемов обменов с внешними массивами, возможно, следует предусмотреть более мелкое дробление информации на блоки или отдельное хранение тех частей, которые передаются между процессорами в своих внешних массивах. Выбор соответствующего дробления определяется структурой множеств F_i'.

После получения ЯДРО-программы для нее можно решить задачу синхронизации, проанализировав информационные связи и определив места, где следует поставить операторы ЗАНЯТЬ и ОСВОБОДИТЬ соответствующие блоки внешних массивов. В некоторых случаях программу в ЯДРЕ можно преобразовать в ПРОСТОР-программу. В частности, это может быть сделано в случае применения второго метода синтеза, когда выявлены простые законы изменения индексов для множества C', а количество процессоров совпадает с числом блоков внешних массивов.

Рассмотренный подход можно применить также и для случая, когда вместо системы функций задана программа над двухуровневыми структурами данных, базовыми операторами которой являются присваивания вида

$$y_1 := f_1(x, y), \ldots, y_m := f_m(x, y),$$

где $f_1, \ldots, f_m$ – периодически определенные функции над двухуровневыми структурами данных. Применяя к ним методы синтеза последовательных программ, получим программу с операторами присваивания вида (4.1), от которой снова можно переходить к ЯДРО-программе. В общем случае переход от программы над двухуровневыми структурами данных к ЯДРО-программе целесообразно выполнять, выделяя в качестве макрооператоров не отдельные присваивания, а группы присваиваний и даже более крупные операторы, содержащие циклы и ветвления. В целом задача выбора макрооператоров не простая и должна решаться с учетом конкретных свойств вычислительного процесса, определенного исходной программой. От ее решения в конечном счете зависит эффективность работы вычислительной системы при выполнении ЯДРО- или ПРОСТОР-программы.

Исходное описание системы функций $y = f(x)$ обычно задается как описание системы функций над одноуровневыми структурами данных:

$$f: \Gamma^n(C, D) \to \Gamma^m(C, D). \tag{4.2}$$

Поэтому для того, чтобы применить общий подход, описанный выше, следует прежде всего решить задачу перехода от одноуровневых к двухуровневым структурам данных. Рассмотрим решение этой задачи сначала в общем виде, затем для некоторых частных случаев. Основной метод перехода состоит в разложении области расположения C в композицию двух областей C' и C''. Такое разложение задается взаимно однозначным соответствием γ: $C \to C' \times C''$. Соответствие γ индуцирует взаимно однозначное отображение $\bar{\gamma}$: $\Gamma(C, D) \to \Gamma(C', \Gamma(C'', D))$ множества одноуровневых структур данных, расположенных на C, на множество двухуровневых структур данных с областями расположения C' (область второго уровня) и C'' (область расположения первого уровня). Если $\gamma(c) = (\gamma'(c), \gamma''(c))$, $c \in C$, то отображение $\bar{\gamma}$ получается с помощью формулы $\bar{\gamma}(x) = \bar{x}$, где

$$\bar{x}(c')(c'') = x(\gamma^{-1}(c', c'')).$$

Обратный переход получается по формуле

$$x(c) = \bar{x}(\gamma'(c))(\gamma''(c)).$$

Двухуровневую структуру данных $y \in \Gamma(C', \Gamma(C'', D))$ удобно иногда отождествлять со структурой данных $y' \in \Gamma(C' \times C'', D)$, расположенной на $C' \times C''$, полагая $y'(c', c'') = y(c')(c'')$. Это отождествление мы иногда будем делать для упрощения выкладок, не оговаривая его особо.

Рассмотрим примеры. Пусть $C = Z^m$ – целочисленная решетка размерности m, $I = (0 : k_1 - 1, \ldots, 0 : k_m - 1)$ – прямоугольный параллелепипед размером $k_1 \times \ldots \times k_m$. Определим разложение γ: $Z^m \to Z^m \times I$, полагая

$\gamma(c) = (c \div k,\ c \bmod k)$, где $c = (c_1, \ldots, c_m)$, $k = (k_1, \ldots, k_m)$, $c \div k = (c_1 \div k_1, \ldots, c_m \div k_m)$, $c \bmod k = (c_1 \bmod k_1, \ldots, c_m \bmod k_m)$, $\div$ есть целая часть частного, mod – остаток от деления. Обратное отображение γ^{-1} определяется формулой $\gamma^{-1}(c', c'') = kc' + c''$ (умножение и сложение снова покомпонентные). Такое разложение соответствует разбиению решетки на прямоугольные параллелепипеды. Действительно, координаты точки c решетки однозначно определяются координатами $c' = \gamma'(c) = c \div k$ параллелепипеда, в котором расположена эта точка, и координатами $c'' = c \bmod k$ этой точки внутри параллелепипеда. Отображение γ: $Z^2 \to Z \times I$ при $I = (-\infty : \infty, 0 : k-1)$ и $\gamma(c_1, c_2) = (c_1 \div k, (c_1, c_2 \bmod k))$ дает разбиение плоскости на горизонтальные полосы шириной k. Аналогичным образом можно получить и другие виды разбиений целочисленной решетки на области, каждая из которых является пересечением нескольких полупространств, любые два из которых либо параллельны, либо ортогональны. Эти примеры являются основными для синтеза макроконвейерных программ.

Вообще семейство множеств $Q(c') = \{\gamma^{-1}(c', c'') \mid c'' \in C''\}$ образует разбиение области C, классы которого называются блоками. Каждый из блоков находится во взаимно однозначном соответствии с точками области C''. Если $\gamma(c) = (c', c'')$, то говорят, что точка c находится внутри блока с координатой c', а c'' – координата точки c внутри блока. Таким образом, каждое разложение области C определяет ее разбиение на блоки.

Установив взаимно однозначное соответствие между структурами данных, его можно перенести на функции обычным образом. Если, скажем, α: $\Gamma(C_1, D) \to \Gamma(C_2, D)$ такое соответствие, то функции f: $\Gamma^n(C_1, D) \to \Gamma^m(C_1, D)$ будет соответствовать функция f' : $\Gamma^n(C_2, D) \to \Gamma^m(C_2, D)$, связанная с f соотношениями $f(x) = \alpha^{-1} f'(\alpha(x))$, $f'(z) = \alpha f(\alpha^{-1}(x))$. Если структуры данных, расположенные на C_2, рассматривать как структуры, реализующие те, которые расположены на C_1, то f' будет реализацией функции f.

В частности, если f есть функция типа (4.2), а α есть $\bar{\gamma}$, то функция $\bar{f}$, реализующая функцию f, определяется равенством $\bar{f}(\bar{x}) = \bar{\gamma}(f(x)) = \overline{f(x)}$. Предположим теперь, что f есть периодически определенная функция. Требуется определить $\bar{f}$ как периодически определенную функцию над структурами данных, расположенными на C'. Какие базовые операции следует рассматривать на области $D' = \Gamma(C'', D)$, для того чтобы это можно было сделать? Как периодически определенная функция f определяется канонической системой уравнений в алгебре структур данных. Поэтому для того, чтобы построить соответствующую систему уравнений для $\bar{f}$, достаточно уметь строить f в случае, когда f – элементарная функция алгебры структур данных, а это в свою очередь требует решения задачи для операций алгебры структур данных – базовых, наложения, вырезки и сдвигов.

Поскольку D' является множеством структур данных, то в качестве базовых операций алгебры D' можно рассматривать любые операции алгебры структур данных, расположенных на C''. При этом, однако, удобно рассматривать D' как компоненту пятиосновной алгебры $(D', C'', D, G'', \mathfrak{R}'')$, где G'' – полугруппа сдвигов множества C'', $\mathfrak{R}''$ – алгебра подмножеств множества C''. Таким образом, переходя к структурам данных, расположенных на C', можем рассматривать не только $\Gamma(C', D')$, но и $\Gamma(C', C'')$, $\Gamma(C', D)$, $\Gamma(C', G'')$, $\Gamma(C', \mathfrak{R}'')$. Поэтому, если, например, $H \in \Gamma(C', \mathfrak{R}'')$, а $z \in \Gamma(C', D')$, то $z/H \in \Gamma(C, D')$ и $(z/H)(c') = z(c')/H(c')$, а если $g \in \Gamma(C', G'')$, то $(z^g)(c') = z(c')^{g(c')}$.

Перейдем к решению задачи вычисления $\overline{f}$ для операций алгебры структур данных. Если ω – базовая операция алгебры D, то для любых $x_1, \ldots, x_n \in \Gamma(C, D)$ имеет место очевидное равенство

$$\overline{\omega(x_1, \ldots, x_n)} = \omega(\overline{x}_1, \ldots, \overline{x}_n). \tag{4.3}$$

Аналогично для наложения

$$\overline{x_1 \sqcup x_2} = \overline{x}_1 \sqcup \overline{x}_2. \tag{4.4}$$

Пусть теперь $H \subset C$. Построим структуру данных $\overline{H} \in \Gamma(C', \mathfrak{R}'')$, полагая $c'' \in H(c') \Leftrightarrow (c', c'') \in \gamma(H) \Leftrightarrow \gamma^{-1}(c', c'') \in H$. Тогда

$$\overline{x/H} = \overline{x}/\overline{H}. \tag{4.5}$$

Действительно, $\overline{x/H}(c', c'') = (x/H)(\gamma^{-1}(c', c'')) =$ (если $\gamma^{-1}(c', c'') \in H$, то $x(\gamma^{-1}(c', c''))$, иначе w) = (если $(c', c'') \in \gamma(H)$ то $\overline{x}(c', c'')$, иначе w) = (если $c'' \in \overline{H}(c')$, то $\overline{x}(c', c'')$, иначе w) = $(\overline{x}(c')/\overline{H}(c'))(c'') = (\overline{x}/\overline{H})(c', c'') \Rightarrow \overline{x/H} = \overline{x}/\overline{H}$.

Найдем теперь выражение для $\overline{x^g}$. Зафиксировав $c' \in C'$, определим на множестве C'' отношение эквивалентности, полагая $c_1'' = c_2''(c', g) \Leftrightarrow \gamma'((\gamma^{-1}(c', c_1''))g) = \gamma'((\gamma^{-1}(c', c_2''))g)$, т.е. точки $(\gamma^{-1}(c', c''))g$ и $(\gamma^{-1}(c', c''))g$ принадлежат одному и тому же блоку разбиения области C. Выберем достаточно большое множество индексов $I(g)$ и занумеруем этими индексами классы каждого из разбиений, определенных введенным отношением. Обозначим полученные классы через $H_{i,g}(c')$. Если индексов больше, чем классов, то часть из множеств $H_{i,g}(c')$ будут пустыми. Для каждого $i \in I(g)$ определим сдвиг $g_i' \subset C' \to C'$ области C', полагая

$$c'g_i' = \gamma'(\gamma^{-1}(c', c'')g), \quad c'' \in H_{i,g}(c'),$$

если множество $H_{i,g}(c')$ не пусто (иначе $c'g_i'$ не определено). Очевидно, определение не зависит от выбора c''. Определим теперь структуру данных

$g'' \in \Gamma(C', G'')$, полагая

$$c''g''(c') = \gamma''(\gamma^{-1}(c', c'')g).$$

Рассматривая $H_{i,g}$ как структуру данных из $\Gamma(C', \mathcal{R}'')$, получим

$$\overline{x^g} = \bigsqcup_{i \in I(g)} (\bar{x}^{g_i'})^{g''} / H_{i,g}. \tag{4.6}$$

Действительно, $\overline{x^g}(c', c'') = x(\gamma'((\gamma^{-1}(c', c''))g), \gamma''((\gamma^{-1}(c', c''))g))$. Пара (c', c'') однозначно определяет класс $H_{i,g}(c')$, которому принадлежит c''. Тогда $\gamma'((\gamma^{-1}(c', c''))g) = c'g_i'$, $\gamma''((\gamma^{-1}(c', c''))g) = c''g''(c')$, откуда следует, что $\overline{x^g}(c', c'') = (\bar{x}^{g_i'})^{g''}(c', c'')$, что и доказывает равенство (4.6).

Будем говорить, что сдвиг g имеет конечную степень относительно разложения γ: $C \to C' \times C''$ области C, если множество $I(g)$ может быть выбрано конечным. Наименьшее число элементов этого множества в конечном случае называется степенью сдвига g относительно данного разложения. Максимальная степень сдвигов, которые используются при определении периодически определенного преобразования, характеризует степень графа информационных зависимостей между блоками. Поэтому при выборе разложения следует стремиться к уменьшению этих степеней.

Оценим степень линейного сдвига g целочисленной решетки Z^m при разложении ее на прямоугольные параллелепипеды, как в рассмотренном выше примере. Пусть g – линейное аффинное преобразование: $g(c) = \beta(c) + \alpha$, гда β – линейное преобразование, заданное целочисленной матрицей (β_{ij}) порядка m, α – целочисленный m-мерный вектор. Вектор α единственным образом представим в виде $\alpha = k\alpha' + \alpha''$, где $0 \leqslant \alpha_i'' \leqslant k_i - 1$ $(i = 1, \ldots, m)$. Пусть $\overline{x^g}(c', c'') = \bar{x}(\gamma^{-1}g\gamma)(c', c'') = \bar{x}(c_1', c_1'')$. Найдем c_1' и c_1''. Имеем $(c', c'') \xrightarrow{\gamma^{-1}} kc' + c'' \xrightarrow{g} \beta(kc') + \beta(c'') + \alpha$, откуда $c_1' = (\beta(kc') + \beta(c'') + k\alpha' + \alpha'') \div k = (\beta(c'') + \alpha'') \div k + \beta(c') + \alpha'$, $c'' = (\beta(kc') + \beta(c'') + k\alpha' + \alpha'') \bmod k = \beta(c'') + \alpha'' \bmod k$. Обозначим $(\beta(c'') + \alpha'') \div k = (\delta_1, \ldots, \delta_m)$. Тогда $\delta_i = \left(\sum_{j=1}^{m} \beta_{ij}c_j'' + \alpha_i''\right) \div k_i$. Поскольку $0 \leqslant c_j'' \leqslant k_i - 1$, $0 \leqslant \alpha_i'' \leqslant k_i - 1$, то $0 \leqslant \delta_i \leqslant \mu_i$ или $-\mu_i \leqslant \delta_i \leqslant 0$, где $\mu_i = \left|\sum_{j=1}^{m} \beta_{ij}\right|$. Поэтому степень сдвига g не превосходит $(\mu_1 + 1) \ldots (\mu_m + 1) \leqslant (\mu + 1)^m$, где $\mu = \max_{1 \leqslant i \leqslant m} \left|\sum_{j=1}^{m} \beta_{ij}\right|$ – норма матрицы (β_{ij}).

Если, в частности, g есть движение, т.е. $\beta(c) = c$, то $\delta_i = (c_i'' + \alpha_i'') \div k_i$, откуда степень сдвига g не превосходит 2^m.

Таким образом, в процессе проектирования программы вычисления периодически определенной функции $y = f(x)$, заданной канонической системой уравнений $y = F(x, y)$, необходимо решить следующие задачи:

1. Реализовать одноуровневые структуры данных двухуровневыми, выбрав подходящее разложение γ: $C \to C' \times C''$ области расположения структур данных $x_1, ..., x_n, y_1, ..., y_m$.

2. Синтезировать программу верхнего уровня, вычисляющую $y_1, ..., y_m$ как структуры данных, расположенные на C'.

3. Синтезировать программы вычисления элементарных функций алгебры $D' = \Gamma(C'', D)$ как функций над структурами данных, расположенными на C''.

4. Представить полученные программы в подмножестве ЯДРО языка МАЯК.

Выбор разложения области расположения следует выполнять так, чтобы соотношения между объемами вычислений при вызове программ в отдельных процессорах и объемами обменов с блоками внешних массивов давали достаточно хорошую эффективность при динамическом распараллеливании. Оценку эффективности и коэффициент ускорения при заданном числе процессоров можно получить еще до построения окончательной программы, зная количественные характеристики задач, которые будут решаться с помощью проектируемой программы. Для этого следует проанализировать процессы вычислений, которые получаются при рассматриваемом разложении области расположения структур данных, построить математическую модель и исследовать ее характеристики.

Если исходное задание для синтеза макроконвейерной программы представлено в виде последовательной программы над структурами данных, использующей в качестве базовых операций периодически определенные функции, то в процессе проектирования решаются следующие задачи:

1. Переход к двухуровневым структурам данных.

2. Преобразование исходной программы в программу над двухуровневыми структурами данных.

3. Выделение макрооператоров.

4. Разработка ЯДРО- или ПРОСТОР-программ для выполнения макрооператоров.

5. Получение окончательной программы.

Для программ, имеющих простую структуру, основные решения достаточно очевидны. Для более сложно устроенных программ решения принимаются путем комбинирования решений, принятых для их частей. Рассмотрим некоторые типовые решения для простых программ. Пусть тело программы над структурами данных имеет вид следующего простого цикла:

ДЛЯ $t := 1$ ДО T ВЫП
  $x_1 := F_1(x, t), ..., x_n := F_n(x, t)$
КЦ,

где $F_1, ..., F_n$ – периодически определенные функции над структурами данных $x_1, ..., x_n$, расположенными на области C. Базовая алгебра данных может быть многоосновной. От параметра t могут зависеть базовые операции,

вырезки и сдвиги. Пусть при фиксированном t области определения структур данных $x_1, \ldots, x_n$ и области определения значений $y_1 = F_1(x, t), \ldots, y_n = F_n(x, t)$ содержатся в множествах $H_1(t), \ldots, H_n(t)$ соответственно. Пусть γ: $C \to C' \times C''$ – разложение области C. Тогда переход к двухуровневым структурам данных преобразует рассматриваемый цикл следующим образом:

ДЛЯ $t := 1$ ДО T ВЫП

ДЛЯ ВСЕХ $c' \in C'$ ТАКИХ, ЧТО $\bigvee_{i=1}^{n} H_i(t) \cap Q(c') \neq \phi$ ВЫП

$x_1(c') := \overline{F_1(x,t)}(c') / \overline{H_1(t)}(c'), \ldots, x_n(c') := \overline{F_n(x,t)}(c') / \overline{H_n(t)}(c')$

КЦ КЦ.

При проектировании ЯДРО-программы тело внутреннего цикла составляет один макрооператор, который оформляется как вызов программы $P(c', t, \ldots)$. Следующий шаг преобразования:

ДЛЯ $t := 1$ ДО T ВЫП
ДЛЯ ВСЕХ $c' \in C'$ ТАКИХ, ЧТО, ... ВЫП
ВЫЗВАТЬ $P(c', t, \ldots)$
КЦ КЦ.

Общая структура программы P:

НАЧАЛО
ЗАНЯТЬ $z_1(c'g_1), \ldots, z_k(c'g_k)$;
ПРОЧИТАТЬ $z_1(c'g_1), \ldots, z_k(c'g_k)$;
ОСВОБОДИТЬ ...
ДЛЯ ВСЕХ $c'' \in C''$ ВЫПОЛНИТЬ
. . . . . .
КЦ;
ЗАПИСАТЬ $x_1(c'), \ldots, x_m(c')$;
ОСВОБОДИТЬ $x_1(c'), \ldots, x_m(c')$
КОНЕЦ.

Среди занимаемых блоков $z_1(c'g_1)$, ... должны присутствовать не только блоки $x_1(c')$, ..., но и те, которые нужны для вычислений. Они освобождаются сразу после их чтения. Сдвиги $g_1, \ldots, g_k$ могут зависеть от параметра t. Следует обратить внимание, что по семантике языка МАЯК после вызова программы P до тех пор, пока не будет выполнен первый оператор этой программы (оператор ЗАНЯТЬ), новые вызовы не могут начаться. За этим следит операционная система и тем самым обеспечивается разрешение возможных конфликтов в случае, когда две последовательно вызываемые программы занимают одни и те же блоки. При автоматической синхронизации операционная система должна знать блоки $z_1(c'g_1), \ldots, z_k(c'g_k)$, которые используются и изменяются при вызове данной программы и следить за их освобождением. При этом вызов каждой следующей программы будет происходить только в моменты, когда свободны все используемые блоки, что обеспечит высокую эффективность. Информация

об обрабатываемых блоках может быть также использована для оптимизации управления памятью: блоки, которые понадобятся в ближайшем будущем, должны храниться в памяти процессоров.

Если количество доступных процессоров достаточно велико для того, чтобы одновременно вести обработку во всех точках области C', то весь цикл по t можно рассматривать как один макрооператор и реализовать его ПРОСТОР-программой следующего вида:

КОМПОНЕНТЫ $K(H)$; КООРДИНАТА c';
НАЧАЛО ПРОЧИТАТЬ $z_1(c'g_1), \ldots, z_k(c'g_k)$;
  ДЛЯ $t := 1$ ДО T ВЫПОЛНИТЬ
  ДЛЯ $c'' \in C''$ ВЫПОЛНИТЬ . . .
  КЦ;
  ОБМЕН ДАННЫМИ
  КЦ;
  ЗАПИСАТЬ $x_1(c'), \ldots, x_n(c')$
КОНЕЦ.

Здесь $H = \bigcup_{t=1}^{T} \bigcup_{t=1}^{n} H_t(t)$ и начальное чтение выполняется для сдвигов $g_1, \ldots, g_k$, соответствующих значению $t = 1$. Если сдвиги $g_1, \ldots, g_k$ не зависят от t, то получается схема статического итеративного макроконвейера. В противном случае может получиться макроконвейер с переменными связями. Обмен данными используется вместо обращения к внешним массивам.

Проблема получения достаточно высокой эффективности и учет конкретных особенностей периодически определенных функций $F_1, \ldots, F_n$ могут привести к необходимости определенных трансформаций рассматриваемых программ. В частности, может оказаться выгодным в один макрооператор ЯДРО-программы включить обработку нескольких точек множества C'. Это может оказаться выгодным, если число точек области C' подлежащих обработке, на много превосходит число доступных процессов. Например, пусть H есть прямоугольник на плоскости C'. Если число процессоров имеет порядок, равный числу строк или столбцов соответствующей матрицы, то выгодно внутри одного макрооператора обрабатывать все точки одной строки или столбца. При этом, если для обработки следующей точки нужна информация о предыдущей, то возможна экономия обменов, поскольку при переходе к обработке следующей точки информация о предыдущей остается внутри процессора. Формально для общего случая это можно представить с помощью дополнительного разложения области C' с помощью отображения $\gamma_1 \colon C' \to C_1' \times C_2'$ и преобразования цикла по t в такой:

ДЛЯ $t := 1$ ДО T ВЫПОЛНИТЬ
ДЛЯ ВСЕХ $c_1' \in C_1'$ ТАКИХ, ЧТО . . . ВЫПОЛНИТЬ
  ВЫЗВАТЬ $P'(c_1', t, \ldots)$
КЦ; КЦ.

При этом программа P преобразуется в P' следующего вида:

ДЛЯ $c_2' \in C_2'$ ВЫПОЛНИТЬ
ЗАНЯТЬ $z_1(\gamma_1^{-1}(c_1', c_2')g_1)$
ПРОЧИТАТЬ ...
ОСВОБОДИТЬ ...
ДЛЯ ВСЕХ $c'' \in C''$ ВЫП ...
КЦ
ЗАПИСАТЬ $x_1(\gamma_1^{-1}(c_1', c_2'))$, ...
ОСВОБОДИТЬ ...
КЦ.

Теперь, если среди блоков, записываемых в очередном цикле по c_2', имеются такие, которые нужны для следующего, можно будет их не читать, а использовать то, что уже находится в памяти, предусмотрев необходимые дополнительные массивы. В случае ПРОСТОР-программы может потребоваться введение дополнительного уровня для области C''. Разложение γ_2: $C_1'' \times C_2''$ позволяет перейти к смешанному макроконвейеру, некоторое увеличение обмена может снизить времена ожидания. Таким образом, в процессе проектирования может появиться необходимость перехода к трех- и четырехуровневым структурам данных.

Перейдем теперь к рассмотрению циклов по условиям. Пусть тело проектируемой программы имеет вид

ПОКА α ВЫПОЛНЯТЬ
$x_1 := F_1(x)/H_1, \ldots, x_n := F_n(x)/H_n$
КЦ.

Здесь $F_1, \ldots, F_n$ снова периодически определенные функции, а условие α пусть определяется с помощью аддитивного функционала

$$\alpha = \Phi_{H_0, \nu} F_0(x_1, \ldots, x_n),$$

принимающего значения 0, 1, где F_0 — периодически определенная функция, также принимающая значения 0, 1. Тогда ν — это конъюнкция, дизъюнкция, сложение по mod 2 или отрицание одной из этих функций. Цикл по α можно переписать тогда в следующем виде:

ЦИКЛ
$x_0 := F_0(x)/H_0$,
$u := \Phi_{H_0, \nu}(x_0)$;
ЕСЛИ u ТО ВЫЙТИ КЕ;
$x_1 := F_1(x)/H_1, \ldots, x_n := F_n(x)/H_n$
КЦ.

Перейдем к двухуровневым структурам данных. Получим:

ЦИКЛ
ДЛЯ ВСЕХ $c' \in C'$ ТАКИХ, ЧТО $Q(c') \cap H_0 \neq \phi$ ВЫП
$x_0(c') := \overline{F_0(x)}(c')/\overline{H}_0(c')$;
$z(c') := \Phi^{-}_{H_0(c'), \nu}(x_0(c'))$;

КЦ;

$u := \Phi_{H_0, \nu}(z)$;

ЕСЛИ u ТО ВЫЙТИ КЕ;

ДЛЯ ВСЕХ $c' \in C'$ ТАКИХ, ЧТО $\bigvee_{i=1}^{n} Q(c') \cap H_i \neq \phi$ ВЫП

$x_1(c') := \overline{F_1(x)}(c') / \overline{H}_1(c'), \ldots$

КЦ;

КЦ.

Здесь z – булева структура данных, расположенная на C'. Первый цикл лучше переписать в виде, который не требует лишних обращений к внешним массивам:

ДЛЯ ВСЕХ $c' \in C'$ ТАКИХ, ЧТО ... ВЫП

$z(c') := \Phi_{\overline{H}_0(c'), \nu}(\overline{F_0(x)}(c') / \overline{H}_0(c'))$

КЦ.

Выбирая тела каждого из циклов в качестве макрооператоров, получим ЯДРО-программу. Вычисление $\Phi_{H_0, \nu}(z)$ выполняется достаточно быстро и не требует распараллеливания. Недостатком построенной программы является то, что второй цикл не может начинаться прежде, чем окончится первый. Этот недостаток частично можно устранить, совместив вычисление условия α и значений x_i для следующего цикла. Такое преобразование, впрочем, потребует удвоения памяти внешних массивов. Преобразованный цикл:

ЦИКЛ

ДЛЯ ВСЕХ $c' \in C'$ ТАКИХ, ЧТО ... ВЫП

$x'(c') := x(c')$;

$z(c') := \Phi(x'(c'))$;

$x(c') := F(x'(c'))$;

КЦ;

$u := \Phi_{H_0, \nu}(z)$;

ЕСЛИ u ТО $x := x'$ ВЫЙТИ КЕ

КЦ.

Выполнение лишнего цикла компенсируется двухкратным уменьшением количества вызовов. Переменная z должна быть оформлена как внутренняя структура данных в основной программе с областью определения H_0, а сама программа после выделения макрооператора будет иметь следующий вид:

ЦИКЛ

ДЛЯ ВСЕХ $c' \in H_0$ ВЫП

ВЫЗВАТЬ $P(c', \ldots)$ РЕЗУЛЬТАТ $(z(c'))$

КЦ;

```
        u := Φ_{H_0, ν}(z);
        ЕСЛИ u ТО x := x' ВЫЙТИ КЕ
    КЦ
```

Если число доступных процессоров равно числу точек области H_0, то можно синтезировать ПРОСТОР-программу, в которой основная будет одной из компонент. Такая программа имеет следующий вид:

```
ПРОГ A(...);
  КОМПОНЕНТА K;
    ЦИКЛ.
      u := ξ;
      ДЛЯ c' ∈ H_0 ВЫПОЛНИТЬ
        ПРИНЯТЬ v ИЗ T(c');
        u := ν(u, v);
      КЦ;
      ДЛЯ c' ∈ H_0 ВЫП
        ПЕРЕДАТЬ u В T(c')
      КЦ;
    КЦ;
  КОНЕЦ КОМП;
  КОМПОНЕНТА T(H_0); КООРДИНАТА c';
      ЦИКЛ
        x'(c') := x(c');
        u := Φ(x');
        ПЕРЕДАТЬ u В K;
        x(c') := F(x'(c'))
        ПРИНЯТЬ u ИЗ K;
        ЕСЛИ u ТО x(c') := x'(c') ВЫЙТИ КЕ
      КЦ
  ККОМП
КОНЕЦ ПРОГ
```

Программа компоненты T не отображает такие детали, как обмен с внешними массивами. Они выполняются только в начале и конце работы программы. Элемент ξ — нейтральный элемент для операции ν. Если ν — сложение по mod 2 или дизъюнкция, то $\xi = 0$, если конъюнкция, то $\xi = 1$.

В качестве конкретного примера рассмотрим задачу решения системы линейных алгебраических уравнений методом Гаусса с выбором главного элемента в столбце. Предположим, что система представлена в виде прямоугольного массива a размером $(1:n, 1:n+1)$, в котором первые n столбцов представляют матрицу системы, а $n+1$-й столбец-вектор свободных членов. Результатом должен быть вектор x размером $(1:n)$. Начальное представление алгоритма в виде программы над структурами данных имеет следующий вид:

```
ПРОГРАММА ГАУСС
    НАЧАЛО
    ДЛЯ k := 1 ДО n ВЫПОЛНИТЬ
```

$l := \min\{k \leqslant i \leqslant n \mid a_{ik} = \max\{a_{i'k} \mid k \leqslant i' \leqslant n\}\}$;
$A/H_k := F_1(A, k, l, n)$;
КЦ;
$x/(1:n) := F_2(A, n)$
КОНЕЦ

Здесь F_1 и F_2 – периодически определенные функции, скалярное представление которых хорошо известно:

$$a'_{ij} = a_{ij}(1 - a_{lj}/a_{lk}),\quad k+1 \leqslant i \leqslant n,\quad i \neq l,\quad k \leqslant j \leqslant n+1,$$

$$a'_{lj} = a_{kj}(1 - a_{lj}/a_{ik}),\quad k \leqslant j \leqslant n+1,\quad k \neq l,$$

$$a'_{kj} = a_{lj},\qquad k \leqslant j \leqslant n+1.$$

Эти равенства определяют функцию $a' = F_1(A, k, l, n)$ при $k \neq l$. Если же $k = l$, то второе равенство следует отбросить. Множество $H_k = (k:n,\ k:n+1)$. Равенства для x:

$$x_i = (1/a_{ii})\Big(a_{in+1} - \sum_{j=i+1}^{n} a_{ij}x_j\Big),\qquad i = 1, \ldots, n.$$

Предположим, что размеры системы таковы, что в памяти каждого процессора могут полностью поместиться несколько строк или столбцов матрицы A. Поскольку сдвиги, используемые при вычислении функции F_1, направлены вертикально ($(i, j) \to (l, j)$), то при выполнении прямого хода удобно располагать матрицу, раскладывая ее на вертикальные полосы. С другой стороны, для обратного хода, использующего горизонтальные сдвиги ($i \to (i, j)$), лучше раскладывать матрицу на горизонтальные полосы. Удобным компромиссом в этом случае является разложение на прямоугольники, из которых можно набирать как горизонтальные, так и вертикальные полосы. Поэтому разложим плоскость Z^2 с помощью отображения γ: $Z^2 \to Z^2 \times (1:s,\ 1:s)$ с формулами преобразования $\gamma(i,j) = (((i-1) \div s + 1,\ (j-1) \div s + 1),\ ((i-1) \bmod s + 1,\ (j-1) \bmod s + 1)) = ((\xi(i)\xi(j)),\ (\eta(i),\ \eta(j)))$, $\gamma^{-1}((i', j'),\ (i'', j'')) = (s(i'-1) + i'',\ s(j'-1) + j'') = (\zeta(i', i''), \zeta(j', j''))$. При таком разложении блоки так же, как и элементы, нумеруются не с нуля, а с единицы. Аналогичным образом массив x разложим с помощью отображения γ_1: $Z \to Z \times (1:s)$, полагая $\gamma_1(i) = (\xi(i), \eta(i))$. Каждый из двух частей алгоритма (прямой и обратный ход) реализуем в виде отдельной ПРОСТОР-программы. Обе программы вызываются из основной:

ПРОГ ГАУСС
НАЧАЛО
ВЫЗВАТЬ ПРЯМОЙ ХОД (a, n, s);
ВЫЗВАТЬ ОБРАТНЫЙ ХОД (a, x, n, s)
КОНЕЦ.
ПРОГ ПРЯМОЙ ХОД (a, n, s);

```
КОМП Q(1 : ξ(n + 1)); КООРД j';
ИМЯ a' : МАС (1 : n, 1 : s) ВЕЩ;
   НАЧАЛО
      ПРОЧИТАТЬ a(1, j'), a(2, j'), . . . , B a'.
      ЕСЛИ j' = ξ(n + 1) ТО s := η(n + 1) КЕ;
      m := min(n, ζ(j', s));
      ДЛЯ k := 1 ДО m ВЫП
        k' := ξ(k); k'' := η(k);
        ЕСЛИ k' < j' ТО
          ПРИНЯТЬ (l, a1) ИЗ Q(k'); k'' := 1;
        ИНАЧЕ
        l := min{ k ⩽ i ⩽ n | a'ik'' = max{ a'i'k'' | k ⩽ i' ⩽ n}};
        a1 := a'lk'';
        ДЛЯ j := j' + 1 ДО ξ(n + 1) ВЫП
            ПЕРЕДАТЬ (l, a1) В Q(j);
        КЦ
      КЕ;
      ЕСЛИ l ≠ k ТО
        ДЛЯ j'' := k'' ДО s ВЫП a'kj'' := alj'' КЦ
      КЕ;
      ДЛЯ i := k + 1 ДО n ВЫП
      ДЛЯ j'' := k'' ДО s ВЫП
        ЕСЛИ i ≠ l ТО
        a'ij'' := a'ij''(1 − alj''/a1)
        ИНАЧЕ
        a'ij'' := a'kj''(1 − alj''/a1)
        КЕ
      КЦ; КЦ;
      КЦ;
      ЗАПИСАТЬ a' В a(1, j'), a(2, j'), . . .
   КОНЕЦ
КОНЕЦ КОМП.
ПРОГ ОБРАТНЫЙ ХОД (a, x, n, s);
КОМП Q(1 : ξ(n)); КООРД i';
ИМЯ a' : МАС (1 : s, 1 : n + 1) ВЕЩ;
ИМЯ x' : МАС (ζ(i', 1) : n) ВЕЩ;
   НАЧАЛО.
      ПРОЧИТАТЬ a(i', 1), a(i', 2), . . . В a';
      ПРОЧИТАТЬ x(n), x(n − 1), . . . В x.
      ЕСЛИ i' = ξ(n) ТО s := η(n) КЕ;
      ДЛЯ i := ζ(i', s) ШАГ − 1 ДО ζ(i', 1) ВЫП
          i'' := η(i);
          x_i := (1/a_{i''i}) (a_{i''n+1} − Σ_{j=i''+1}^{n} a_{i''j} x_j)
```

КЦ;
ЗАПИСАТЬ $x'(\zeta(i', 1) : \zeta(i', s))$ В $\underline{x}(i')$.
КОНЕЦ
КОНЕЦ КОМП.

В программах ПРЯМОЙ и ОБРАТНЫЙ ХОД операторы ПРОЧИТАТЬ и ЗАПИСАТЬ являются сокращенными обозначениями циклов, которые формируют из соответствующих блоков внешних массивов a и x массивы a' и x'. Программы прямого и обратного хода можно было бы объединить в одну ПРОСТОР-программу, однако это не целесообразно, поскольку коэффициент равномерности объединенной программы при большом числе процессоров и большом s равен примерно 2/3. Он же будет и предельным значением эффективности. В случае разделения процессоры, которые постепенно освобождаются в каждой из двух программ, передаются один за другим в распоряжение операционной системы и могут быть загружены другой работой, а те процессоры, которые остаются, работают с эффективностью, близкой к 1.

§ 5. Динамическое распараллеливание последовательных программ

Если в последовательной программе над двухуровневыми структурами данных выделены макрооператоры, которые целесообразно выполнять параллельно на отдельных процессорах, то возможно организовать параллельные макроконвейерные вычисления таким образом, что выяснение возможности параллельного выполнения макрооператоров будет происходить во время выполнения программы; другими словами, будет происходить динамическое распараллеливание последовательной программы. Динамическое распараллеливание обладает определенными преимуществами перед статическим распараллеливанием, поскольку динамический анализ информационной зависимости между операторами может быть проведен более тонко; динамическая информационная независимость операторов может иметь место при их статической зависимости. Кроме того, в динамике некоторые информационные связи можно разрывать, используя динамическую локализацию или переименование переменных.

В этом параграфе будет представлена общая теория динамического распараллеливания последовательных программ. Она может применяться не только для макроконвейерного распараллеливания, но и для распараллеливания на уровне команд и микрокоманд в системах с параллельными операционными устройствами, микроконвейерных системах и многопроцессорных системах с общей памятью.

Объектом динамического распараллеливания является U–Y-программа над памятью R, интерпретированная на области D. Состояния памяти $b \subset R \to D$ рассматриваются как частичные отображения. Типизация и косвенное именование переносятся на более конкретные уровни рассмотрения.

Предположим, что для каждого из базовых операторов $y \in Y$ и для каждого состояния памяти $b \in \gamma(R, D)$ заданы множества $\mathrm{In}(y, b)$ и $\mathrm{Out}(y, b)$ используемых и изменяемых в состоянии b переменных. Понятия "использование" и "изменение" зависят не только от функциональных характе-

ристик, но и от реализации базовых операторов. Поэтому, если $b(r) \neq by(r)$, то, безусловно, должно быть $r \in \mathrm{Out}(y, b)$, однако обратного, вообще говоря, не требуется. Аналогично, если $b(s) = b'(s)$ для всех $s \in R$ таких, что $s \neq r$, но $by \neq b'y$, то $r \in \mathrm{In}(y, b)$, но не наоборот. Кроме того, предполагается, что $\mathrm{In}(y, b)$ и $\mathrm{Out}(y, b)$ определены даже в том случае, когда by не определено (что не исключается). Понятия "используемые" и "изменяемые" переменные должны быть распространены на произвольные регулярные программы. При этом должны выполняться определенные условия. Если b и b' – состояния памяти, а $R' \subset R$, то $b = b'(R')$ означает, что $b(r) = b'(r)$ для всех $r \in R'$. Условия, которым должны удовлетворять множества In и Out:

V1. Если $b' = b(\mathrm{In}(P, b))$, то $\mathrm{In}(P, b') = \mathrm{In}(P, b')$, $\mathrm{Out}(P, b) = \mathrm{Out}(P, b')$.

V2. Если bP определено, то $b = bP(R \setminus \mathrm{Out}(P, b))$.

V3. Если bP определено и $b' = b(\mathrm{In}(P, b))$, то $b'P$ также определено и $bP = b'P(\mathrm{Out}(P, b))$.

Условия V2 и V3 требуют согласования понятий "использует" и "изменяет" с функциональными свойствами оператора P: в результате действия P на b могут измениться только выходные переменные и этот результат может зависеть только от входных. Условие V1 показывает, что сами множества In и Out могут зависеть также лишь от входных переменных оператора P. Будем предполагать, что множества In и Out, определенные для базовых операторов, удовлетворяют условиям V1–V3. Предположим также, что для каждого базового условия $\alpha \in U$ и каждого состояния памяти b задано множество $\mathrm{In}(\alpha, b) \subset R$, удовлетворяющее условию:

V4. Если $b = b'(\mathrm{In}(\alpha, b))$, то $\mathrm{In}(\alpha, b') = \mathrm{In}(\alpha, b)$ и $\alpha(b') = \alpha(b)$.

Пример. Пусть базовые операторы суть операторы присваивания над памятью с массивами. Тогда множества In и Out можно определить синтаксически, рассматривая вхождения переменных в левые и правые части простых присваиваний и вычисляя индексы в заданном состоянии памяти b. Для оператора присваивания $y = (r(i, j) := r(i, j) + s(i, k) * s'(k, j))$ естественно положить $\mathrm{In}(y, b) = \{i, j, k, r(b(i), b(j)), s(b(i), b(k)), s'(b(k), b(j))\}$, $\mathrm{Out}(y, b) = \{r(b(i), b(j))\}$, если только $b(i), b(j), b(k)$ определены. Если же, например, $b(i)$ и $b(j)$ не определены, можно положить $\mathrm{In}(y, b) = \{i, j, k\}$, $\mathrm{Out}(y, b) = \phi$.

При распространении определения множеств In и Out на произвольные регулярные программы должен быть найден компромисс между двумя противоречивыми требованиями. С одной стороны, должен существовать простой алгоритм вычисления In и Out для произвольного состояния b. С другой стороны, значения этих функций должны как можно больше приближаться к функциональному толкованию использования и изменения. В рассмотренном выше примере проще всего положить $\mathrm{In}(y, b) = \{i, j, k, r, s, s'\}$, $\mathrm{Out}(y, b) = \{r\}$, однако это будет слишком грубым приближением.

Для того чтобы изучить возможные способы распространения In и Out, рассмотрим сначала самое сильное определение этих функций. Результат этого определения обозначим через in и out. Определения in и out являются рекурсивными и основываются на таком представлении, что переменная

используется или изменяется при выполнении некоторой программы P тогда и только тогда, когда она используется или изменяется в процессе выполнения программы P некоторым базовым оператором. Полагаем

$\mathrm{in}(\epsilon, b) = \mathrm{out}(\epsilon, b) = \phi$;

$\mathrm{in}(y, b) = \mathrm{In}(y, b)$; $\mathrm{out}(y, b) = \mathrm{Out}(y, b)$, $y \in Y$;

$\mathrm{in}(PQ, b) = \mathrm{in}(P, b) \cup$ (если bP определено, то $\mathrm{in}(Q, bP) \setminus \mathrm{out}(P, b)$ иначе ϕ).

$\mathrm{in}(_\alpha(P \vee Q), b) = \mathrm{in}(\alpha, b) \cup$ (если $\alpha(b) = 1$, то $\mathrm{in}(P, b)$ иначе если $\alpha(b) = 0$, то $\mathrm{in}(Q, b)$ иначе ϕ);

$\mathrm{in}(_\alpha\{P\}, b) = \mathrm{in}(\alpha, b) \cup$ (если $\alpha(b) = 0$, то $\mathrm{in}(P\,{}_\alpha\{P\}, b)$ иначе ϕ);

$\mathrm{out}(PQ, b) = \mathrm{out}(P, b) \cup$ (если bP определено, то $\mathrm{out}(Q, bP)$ иначе ϕ);

$\mathrm{out}(_\alpha(P \vee Q), b) =$ если $\alpha(b) = 1$, то $\mathrm{out}(P, b)$ иначе, если $\alpha(b) = 0$, то $\mathrm{out}(Q, b)$ иначе ϕ;

$\mathrm{out}(_\alpha\{P\}, b) =$ если $\alpha(b) = 0$, то $\mathrm{out}(P\,{}_\alpha\{P\}, b)$ иначе ϕ;

$\mathrm{in}(\alpha \vee \beta, b) = \mathrm{in}(\alpha \wedge \beta, b) = \mathrm{in}(\alpha, b) \cup \mathrm{in}(\beta, b)$;

$\mathrm{in}(\overline{\alpha}, b) = \mathrm{in}(\alpha, b)$;

$\mathrm{in}(\alpha P, b) = \mathrm{in}(P, b) \cup$ (если bP определено, то $\mathrm{in}(\alpha, bP)$ иначе ϕ).

Несмотря на использование немонотонной операции (вычитание множеств), система уравнений, соответствующая приведенным равенствам, имеет наименьшее решение. Это решение можно выразить с помощью функции $l(\alpha, P, b)$, равной наименьшему целому m такому, что $0 \leqslant m \leqslant \infty$, и для всех k таких, что $0 \leqslant k < m$, имеет место $\alpha(bP^k) = 0$. Если $b\,{}_\alpha\{P\}$ определено, то ясно, что $b\,{}_\alpha\{P\} = bP^l$, где $l = l(\alpha, P, b)$. Для наименьшего решения (по включению множеств) имеют место равенства

$$\mathrm{in}(_\alpha\{P\}, b) = \bigcup_{k=1}^{l} (\mathrm{in}(P, bP^{k-1}) \setminus \mathrm{out}(P^{k-1}, b)) \cup \bigcup_{k=0}^{l'} (\mathrm{in}(\alpha, bP^k) \setminus \mathrm{out}(P^k, b)),$$

$$\mathrm{out}(_\alpha\{P\}, b) = \bigcup_{k=1}^{l} \mathrm{out}(P, bP^{k-1}),$$

где $l = l(\alpha, P, b)$, $l' =$ если $l < \infty$ и bP^l не определено, то $l - 1$ иначе l.

Теорема 5.1. *Множества* in *и* out *удовлетворяют условиям* V1–V4.

В доказательстве этого и других утверждений используются следующие, легко проверяемые свойства отношения $b = b'(R')$:

1. При фиксированном $R' \subset R$ отношение $b = b'(R')$ есть отношение эквивалентности на множестве всех состояний памяти.
2. Если $R' \subset R''$, то из $b = b'(R'')$ следует $b = b'(R')$.
3. Из $b = b'(R')$ и $b = b'(R'')$ следует $b = b'(R' \cup R'')$.

Полезны также следующие леммы, выражающие свойства множеств In и Out, вытекающих из V1–V4.

Лемма 5.1. *Если* $b = b'(R')$, bP *и* $b'P$ *определены и* $R'' = \mathrm{Out}(P, b) \cup \mathrm{Out}(P, b')$, *то* $bP = b'P(R' \setminus R'')$.

Действительно, из условия леммы и V3 следует:

$b = bP(R' \setminus R'')$,

$b' = b'P(R' \setminus R'')$,

$b = b'(R' \setminus R'')$;

далее идет транзитивность эквивалентности.

Лемма 5.2. *Если* $b = b'(R' \setminus \mathrm{Out}(P, b))$ *и* $b = b'(\mathrm{In}(P, b))$, *то* $bP = b'P(R')$.

Действительно, лемма 5.1 и V1 дают $bP = b'P(R' \setminus \mathrm{Out}(P, b) \setminus (\mathrm{Out}(P, b) \cup \mathrm{Out}(P, b'))) \Rightarrow bP = b'P(R' \setminus \mathrm{Out}(P, b))$, но поскольку $bP = b'P(\mathrm{Out}(P, b))$ в силу V3, то $bP = b'P(R')$.

Доказательства лемм показывают, что обе они имеют место, даже если V1–V3 выполняется только для оператора P.

Доказательство теоремы 5.1 проводится индукцией по числу операций в регулярном выражении P или условии α. Для базовых операторов и условий V1–V4 предполагаются выполненными; для тождественного оператора они очевидны. Доказательство шага индукции для произвольных операторов распадается на три случая – произведение, α-дизъюнкция и α-итерация.

Произведение.

V1. Пусть $b = b'(\mathrm{in}(PQ, b))$. Тогда $b = b'(\mathrm{in}(P, b))$ и по предположению индукции $\mathrm{in}(P, b) = \mathrm{in}(P, b')$, $\mathrm{out}(P, b) = \mathrm{out}(P, b')$. Пусть bP определено. Тогда по V3 $b'P$ также определено, а по предположению $b = b'(\mathrm{in}(Q, bP) \setminus \mathrm{out}(P, b))$. Применяя лемму 5.2, получаем $bP = b'P(\mathrm{in}(Q, bP))$, откуда по предположению индукции и V1 вытекают равенства $\mathrm{in}(Q, bP) = \mathrm{in}(Q, b'P)$ и $\mathrm{in}(PQ, b) = \mathrm{in}(PQ, b')$.

Пусть теперь bP не определено. Тогда по V3 $b'P$ также не определено и $\mathrm{in}(PQ, b) = \mathrm{in}(P, b) = \mathrm{in}(P, b') = \mathrm{in}(PQ, b')$. Равенство $\mathrm{out}(PQ, b) = \mathrm{out}(PQ, b')$ доказывается аналогично.

V2. Пусть bPQ определено. Тогда bP определено и по предположению индукции в силу V2 получаем

$$b = bP(R \setminus \mathrm{out}(P, b)),$$

$$bP = (bP)Q(R \setminus \mathrm{out}(Q, bP)).$$

Поскольку в этом случае $\mathrm{out}(PQ, b) = \mathrm{out}(P, b) \cup \mathrm{out}(Q, bP)$, то, увеличивая в двух предыдущих равенствах вычитаемое, получаем

$$b = bP(R \setminus \mathrm{out}(PQ, b)),$$

$$bP = (bP)Q(R \setminus \mathrm{out}(PQ, b)),$$

откуда окончательно

$$b = bPQ(R \setminus \mathrm{out}(PQ, b)).$$

V3. Пусть bPQ определено и $b = b'(\mathrm{in}(PQ, b))$. Тогда bP также определено и $b = b'(\mathrm{in}(P, b))$. Применяя лемму 5.2, получаем $bP = b'P(\mathrm{in}(Q, bP))$ и по предположению индукции $bPQ = b'PQ(\mathrm{out}(Q, bP))$. Применяя лемму 5.2 к условию $bP = b'P(\mathrm{out}(P, b))$, получаем $bPQ = b'PQ(\mathrm{out}(P, b))$, откуда окончательно $bPQ = b'PQ(\mathrm{out}(PQ, b))$.

α-дизъюнкция.

Пусть $S = {}_{\alpha}(P \vee Q)$, $b = b'(S', b)$. Рассмотрим случаи:

1) $\alpha(b) = 1 \Rightarrow \mathrm{in}(S, b) = \mathrm{in}(\alpha) \cup \mathrm{in}(P, b) \Rightarrow \alpha(b') = 1 \Rightarrow \mathrm{in}(S, b) = \mathrm{in}(S, b')$, $\mathrm{out}(S, b) = \mathrm{out}(S, b')$;

2) $\alpha(b) = 0$ аналогично;

3) $\alpha(b)$ – не определено $\Rightarrow \mathrm{in}(S, b) = \mathrm{in}(\alpha) \Rightarrow \alpha(b) = \alpha(b') \Rightarrow \mathrm{in}(S, b) = \mathrm{in}(S, b')$, $\mathrm{out}(S, b) = \mathrm{out}(S, b')$.

Условия V2–V3 доказываются аналогично разбором случаев $\alpha(b) = 0$ и $\alpha(b) = 1$.

Итерация.

Лемма 5.3. *Если* $b = b'(\mathrm{in}_\alpha\{P\})$, $l = l(\alpha, P, b)$, *то для всех* $k - 1 < l$ $(k = 1, 2, \ldots)$ *имеет место* $b = b'(\mathrm{in}(P^k, b))$.

Доказательство проводится индукцией по k. При $k = 1$ очевидно. Пусть $b = b'(\mathrm{in}(P^{k-1}, b))$ и $k - 1 < l$. Тогда, поскольку $b = b'(\mathrm{in}(P, bP^{k-1}))\setminus \mathrm{out}(P^{k-1}, b))$, то $b = b'(\mathrm{in}(P^{k-1}, b) \cup (\mathrm{in}(P, bP^{k-1})\setminus\mathrm{out}(P^{k-1}, b)))$, что эквивалентно $b = b'(\mathrm{in}(P^k, b))$. Лемма доказана.

V1. Пусть $b = b'(\mathrm{in}(_\alpha\{P\}, b))$. Достаточно показать, что для всех $k = 1, 2, \ldots$ таких, что $k - 1 < l$, имеют место равенства

$$\mathrm{in}(P^k, bP^{k-1}) = \mathrm{in}(P, b'P^{k-1}),$$
$$\mathrm{out}(P^k, b) = \mathrm{out}(P^k, b'),$$
$$\mathrm{in}(\alpha, bP^{k-1}) = \mathrm{in}(\alpha, b'P^{k-1}),$$
$$\mathrm{out}(P, bP^{k-1}) = \mathrm{out}(P, b'P^{k-1}).$$

Предполагая $k - 1 < l$, по лемме 5.3 получаем

$$b = b'(\mathrm{in}(P^k, b)),$$

откуда следует второе равенство. Из $b = b'(\mathrm{in}(P^{k-1}, b))$ и $b = b'(\mathrm{in}(P, bP^{k-1})\setminus\mathrm{out}(P^{k-1}, b))$ по лемме 5.2 следует $bP^{k-1} = b'P^{k-1}(\mathrm{in}(P, bP^{k-1}))$, а из $b = b'(\mathrm{in}(\alpha, bP^{k-1})\setminus\mathrm{out}(P^{k-1}, b))$ по той же лемме $bP^{k-1} = b'P^{k-1}(\mathrm{in}(\alpha, bP^{k-1}))$, откуда в силу V1 и V4 соответственно получаются первое и третье равенства. Четвертое доказывается индукцией по k с использованием второго равенства.

V2. Если $b\,_\alpha\{P\}$ определено, то $b\,_\alpha\{P\} = bP^l$. Поскольку V2 верно для P, то, пользуясь доказательством V2 для произведения, получаем, что оно верно для $P^2, \ldots, P^l$. Поэтому $b = b'P(R\setminus\mathrm{out}(P^l, b))$.

Если при тех же условиях $b = b'(\mathrm{in}(_\alpha\{P\}, b))$, то по уже доказанному $\mathrm{in}(\alpha, bP^{k-1}) = \mathrm{in}(\alpha, b'P^{k-1})$ для всех k таких, что $k - 1 < l$, а если bP^l определено, то $\mathrm{in}(\alpha, bP^l) = \mathrm{in}(\alpha, b'P^l)$, откуда $\alpha(bP^l) = \alpha(b'P^l) = 1$ и, следовательно, $b'P^l = b'\,_\alpha\{P\}$ определено. Применяя доказательство V3 для степеней оператора P, получаем, что $bP^l = b'P^l(\mathrm{out}(P^l, b))$, что и дает требуемый результат.

Докажем теперь утверждение шага индукции теоремы 5.1 для условий. Если условие представляет собой дизъюнкцию, конъюнкцию или отрицание, то доказательство V4 следует из определений. Рассмотрим условие вида αP. Пусть $b = b'(\mathrm{in}(\alpha P, b))$. Рассмотрим два случая.

1) bP определено. Тогда $\mathrm{in}(\alpha P, b) = \mathrm{in}(P, b) \cup \mathrm{in}(\alpha, bP)$, $b = b'(\mathrm{in}(P, b))$, $b = b'(\mathrm{in}(\alpha, bP))$. В силу V1 для P (индуктивное предположение) $\mathrm{in}(P, b) = \mathrm{in}(P, b')$. С другой стороны, $b = b'(\mathrm{in}(\alpha, bP)) \Rightarrow b = b'(\mathrm{in}(\alpha, bP)\setminus\mathrm{out}(P, b)) \Rightarrow bP = b'P(\alpha, bP)$ (лемма 5.2) $\Rightarrow \mathrm{in}(\alpha, bP) = \mathrm{in}(\alpha, b'P) \Rightarrow \mathrm{in}(\alpha P, b) = \mathrm{in}(\alpha P, b')$.

Кроме того, $(\alpha P)(b) = \alpha(bP) = \alpha(b'P) = (\alpha P)(b')$ (индуктивное предположение для α).

2) bP не определено. Тогда $\mathrm{in}(\alpha P, b) = \mathrm{in}(P, b) = \mathrm{in}(P, b') = \mathrm{in}(\alpha P, b')$, $(\alpha P)(b)$ и $(\alpha P)(b')$ оба не определены. Теорема 5.1 доказана.

Функции in и out трудно вычислимы, и для динамического распараллеливания их следует аппроксимировать с помощью более просто вычислимых функций In и Out, которые выбираются так, что бы выполнялись аксиомы V1–V4. При этом может быть полезен следующий критерий правильности выбора аппроксимации.

Теорема 5.2. *Пусть для любой регулярной программы P и состояния памяти b имеют место включения* $\mathrm{in}(P, b) \subset \mathrm{In}(P, b)$, $\mathrm{out}(P, b) \subset \subset \mathrm{Out}(P, b) \subset \mathrm{out}(P, b) \cup \mathrm{In}(P, b)$. *Тогда, если* In *и* Out *для любых регулярных программ удовлетворяют условию* V1, *то они также удовлетворяют условиям* V2 *и* V3.

Пусть условия теоремы имеют место, и пусть bP определено. Тогда $b = bP(R \setminus \mathrm{out}(P, b)) \Rightarrow b = bP(R \setminus \mathrm{out}(P, b))$. Если же $b = b'(\mathrm{In}(P, b))$, то $b = b'(\mathrm{in}(P, b))$ и в силу леммы 5.2 $bP = b'P(\mathrm{In}(P, b))$. Поскольку $bP = b'P(\mathrm{out}(P, b))$, то $bP = b'P(\mathrm{In}(P, b) \cup \mathrm{out}(P, b)) \Rightarrow bP = b'P(\mathrm{Out}(P, b))$. Теорема доказана.

Предположим, что функции In и Out, удовлетворяющие аксиомам V1–V4, зафиксированы. Программы (операторы) P и Q называются информационно независимыми относительно состояния b, если выполняется условие:

$$(\mathrm{In}(P, b) \cup \mathrm{Out}(P, b)) \cap \mathrm{Out}(Q, b) = (\mathrm{In}(Q, b) \cup \mathrm{Out}(Q, b)) \cap \mathrm{Out}(P, b) = \phi.$$

Условие информационной независимости обозначим через $\mathrm{Ind}(P, Q, b)$.

Теорема 5.3. $\mathrm{Ind}(P, Q, b) \Rightarrow bPQ = bQP$.

Предположим сначала, что bPQ и bQP оба определены. Из условий теоремы непосредственно вытекает, что $\mathrm{Out}(P, b) = \mathrm{Out}(P, bQ)$, $\mathrm{Out}(Q, b) = \mathrm{Out}(Q, bP)$. Используя этот факт, получаем $bPQ = bP(R \setminus \mathrm{Out}(Q, bP)) \Rightarrow bPQ = bP(R \setminus \mathrm{Out}(Q, b)) \Rightarrow bPQ = b(R \setminus \mathrm{Out}(Q, b) \cup \mathrm{Out}(P, b))$. Аналогично $bQP = b(R \setminus \mathrm{Out}(P, b) \cup \mathrm{Out}(Q, b))$, откуда $bPQ = bQP(R \setminus \mathrm{Out}(P, b) \cup \cup \mathrm{Out}(Q, b))$. Поскольку $bP = b(\mathrm{In}(Q, b))$, то $bPQ = bQ(\mathrm{Out}(Q, b))$. С другой стороны, $bQP = bQ(R \setminus \mathrm{Out}(P, b))$, откуда $bQP = bQ(\mathrm{Out}(Q, b))$; следовательно, $bPQ = bQP(\mathrm{Out}(Q, b))$. Аналогично доказывается $bPQ = bQP(\mathrm{Out}(P, b))$. Таким образом, имеем $bPQ = bQP$.

Пусть теперь bPQ не определено. Если bP не определено, то из $\mathrm{In}(P, b) = \mathrm{In}(P, bQ)$ следует, что и bQP не определено. Если же bP определено, но bPQ не определено, то поскольку $\mathrm{In}(P, bP) = \mathrm{In}(Q, bP)$, то bQ, а следовательно, и bQP снова не определены. Теорема доказана.

Наряду с операторами можно рассматривать информационную независимость условий. Будем, в частности, говорить, что условие α и оператор P независимы, если независимы операторы $(\underset{\alpha}{\epsilon} \vee \epsilon)$ и P. Выполнение регулярной программы, в том числе параллельное, можно описать с помощью автоматной дискретной динамической системы, состояние которой включает в себя состояние информационной среды, выполняемые операторы и остаточную программу, т.е. часть программы, которую осталось выполнить на данный момент времени. Различные способы вы-

полнения удобно получать из системы, описывающей последовательное выполнение программы оператор за оператором. Состояниями такой системы служат пары (b, P), где b – состояние информационной среды, $b \in B$, P – регулярная U–Y-программа, интерпретированная на B и рассматриваемая с точностью до соотношений ассоциативности для умножения и соотношений для тождественного оператора ϵ. Иными словами, $(b, P) = (b', P') \Leftrightarrow b = b'$, а P может быть получено из P' применением соотношений $P_1(P_2P_3) = (P_1P_2)P_3$ и $P\epsilon = \epsilon P = P$. Отношение непосредственных переходов задается следующими правилами:

B1. Если $y \in Y$, то $(b, yP) \to (by, P)$,

B2. Если $\alpha(b) = 1$, то $(b, \underset{\alpha}{(}P \vee P')Q) \to (b, PQ)$,

B3. Если $\alpha(b) = 0$, то $(b, \underset{\alpha}{(}P \vee P')Q) \to (b, P'Q)$,

B4. Если $\alpha(b) = 1$, то $(b, \underset{\alpha}{\{}P\}Q) \to (b, Q)$,

B5. Если $\alpha(b) = 0$, то $(b, \underset{\alpha}{\{}P\}Q) \to (b, P\underset{\alpha}{\{}P\}Q)$.

Систему, определенную правилами B1–B5, обозначим через Γ_0. Настроим систему Γ_0, выбрав в качестве множества начальных состояний все Γ_0, а в качестве заключительных – множество $\{(b, \epsilon) \mid b \in B\}$. Тогда функция F, которую вычисляет система Γ_0, совпадает с отображением $F(b, P) = (bP, \epsilon)$, что вытекает из следующей теоремы.

Теорема 5.4. $bP = b' \Leftrightarrow (b, P) \underset{\Gamma_0}{\Longrightarrow} (b', \epsilon)$. Вспомогательное утверждение:

Лемма 5.4. $(b, P) \underset{\Gamma_0}{\Longrightarrow} (b', P') \Rightarrow (b, PQ) \underset{\Gamma_0}{\Longrightarrow} (b', P'Q)$.

Справедливость этого утверждения проверяется на отношении непосредственных переходов.

Прямая импликация теоремы 5.4 доказывается индукцией по числу операций в программе P. Базис индукции представляет собой тривиальное утверждение, которое соответствует случаю $P = \epsilon$. Шаг индукции доказывается разбором случаев:

1) $P = yQ \Rightarrow b' = byQ$, $(b, P) \xrightarrow{B1} (by, Q) \underset{\Gamma_0}{\Longrightarrow} (byQ, \epsilon) = (b', \epsilon)$;

2) $P = \underset{\alpha}{(}P' \vee P'')Q$, $\alpha(b) = 1 \Rightarrow b' =$

$= bP$, $(b, P) \xrightarrow{B2} (b, P'Q) \underset{\Gamma_0}{\Longrightarrow} (bP'Q, \epsilon) = (b', \epsilon)$.

Аналогично для $\alpha(b) = 0$:

3) $P = \underset{\alpha}{\{}Q\}Q'$, $\alpha(b) = 1 \Rightarrow bP = bQ'$, $(b, P) \xrightarrow{B4} (b, Q') \underset{\Gamma_0}{\Longrightarrow}$

$\underset{\Gamma_0}{\Longrightarrow} (bQ', \epsilon) = (b', \epsilon)$;

4) $P = \underset{\alpha}{\{}Q\}Q'$, $\alpha(b) = 0$. Тогда $b' = bP^l$, где $l = l(\alpha, Q, b)$, $l > 0$.

По лемме 5.4 и предположению индукции для Q получаем, что для произвольного состояния $b'' \in B$ имеет место $(b'', Q\underset{\alpha}{\{}Q\}Q') \underset{\Gamma_0}{\Longrightarrow}$

$\underset{\Gamma_0}{\Longrightarrow}(b''Q, \underset{\alpha}{\{}\ Q\}Q')$, откуда $(b, \underset{\alpha}{\{}\ Q\}Q') \xrightarrow{B5} (b, Q\ \underset{\alpha}{\{}\ Q\}Q') \underset{\Gamma_0}{\Longrightarrow}$

$\underset{\Gamma_0}{\Longrightarrow}(bQ, \underset{\alpha}{\{}Q\}Q') \xrightarrow{B5} (BQ, Q\underset{\alpha}{\{}\ Q\}Q') \underset{\Gamma_0}{\Longrightarrow} (bQ^2, \underset{\alpha}{\{}\ Q\}Q') \to \ldots \underset{\Gamma_0}{\Longrightarrow}$

$\underset{\Gamma_0}{\Longrightarrow}(bQ^l, \underset{\alpha}{\{}\ Q\}Q') \xrightarrow{B4} (bQ^l, \epsilon) = (b', \epsilon).$

Обратная импликация доказывается проверкой того, что $(b, P) \underset{\Gamma_0}{\Longrightarrow}$ $\underset{\Gamma_0}{\Longrightarrow}(b', P') \Rightarrow bP = b'P'$. Теорема доказана.

Система Γ_0 является детерминированной, и процесс, подтверждающий переход $(b, P) \underset{\Gamma_0}{\Longrightarrow} (bP, \epsilon)$, определяется единственным образом. Каждый шаг функционирования системы Γ_0 состоит в выполнении базового оператора или проверке условия. Систему можно обогатить, факторизуя множество ее состояний с помощью произвольного отношения эквивалентности ρ такого, что из $(b, P) = (b', P')\,(\rho)$ вытекает $bP = b'P'$. Состояния факторизованной системы Γ_0/ρ представляются так же, как и состояния Γ_0 парами (b, P), но равенство состояний в Γ_0/ρ означают их эквивалентность по отношению ρ. Если нужно отличить состояние системы Γ_0/ρ от соответствующего состояния системы Γ_0, будем писать $(b, P) \bmod \rho$. Отношение переходов системы Γ_0/ρ определяется теми же самыми правилами B1–B5, что и для системы Γ_0. Отображение $(b, P) \to (b, P) \bmod \rho$ является реализующим и при соответствующей настройке согласованным с настройкой. Таким образом, Γ_0/ρ есть гомоморфная модель системы Γ_0, и для нее справедливы как лемма 5.4, так и теорема 5.4. Система Γ_0/ρ недетерминированна и допускает различные процессы выполнения одной и той же программы. В то же время она, очевидно, глобально детерминированна.

Система Γ_1, которая расширяет Γ_0/ρ (для некоторого отношения ρ), использует свойство информационной независимости программ для организации параллельных вычислений. Состояния системы Γ_1 суть тройки (b, Z, P), где $b \in B = \Gamma(R, D), Z \subset Y$ – конечное множество попарно независимых относительно b базовых операторов, P – регулярная $U - Y$-программа, рассматриваемая с точностью до соотношений ассоциативности и соотношений для ϵ. Если $Z = \{y_1, \ldots, y_m\}$, то через bZ обозначим состояние $by_1 \ldots y_m$. В силу информационной независимости значение bZ не зависит от порядка выполнения операторов. Это вытекает из того, что если P_1, P_2, P_3 попарно информационно независимы относительно b, то P_1, P_2 и P_3 также информационно независимы относительно b. Состояния системы Γ_1 рассматриваются с точностью до отношения эквивалентности ρ, удовлетворяющего следующим условиям:

$$(b, Z, P) = (b', Z', P')\,(\rho) \Rightarrow bZP = b'Z'P', \tag{5.1}$$

$$\mathrm{Ind}(P, P', b) \Rightarrow (b, Z, PP'Q) = (b, Z, P'PQ)\,(\rho), \tag{5.2}$$

$$(b, Z, P) = (b', Z', P')\,(\rho) \Rightarrow Z = Z'. \tag{5.3}$$

Правила, определяющие отношение непосредственных переходов, имеют следующий вид:

С1. Если $y \in Y$ и y' информационно независимы относительно состояния b для любого $y' \in Z$, то

$(b, Z, yP) \to (b, Z \cup \{y\}, P)$.

С2. Если α и yQ информационно независимы относительно b для любого $y \in Z$ и $\alpha(b) = 1$, то

$(b, Z, Q \underset{\alpha}{(} P \vee P')Q') \to (b, Z, QPQ')$.

С3. При тех же условиях, если $\alpha(b) = 0$, то

$(b, Z, Q \underset{\alpha}{(} P \vee P')Q') \to (b, Z, QP'Q')$.

С4. При тех же условиях, если $\alpha(b) = 1$, то

$(b, Z, Q \underset{\alpha}{\{} P\}Q') \to (b, Z, QQ')$.

С5. При тех же условиях, если $\alpha(b) = 0$, то

$(b, Z, Q \underset{\alpha}{\{} P\}Q') \to (b, Z, QP \underset{\alpha}{\{} P\}Q')$.

С6. Если $y \in Z$, то

$(b, Z, P) \to (by, Z \setminus \{y\}, P)$.

С7. Если $s \xrightarrow{\text{С1}} s_1 \xrightarrow{\text{С6}} s_2 \xrightarrow{\text{С6}} \ldots \xrightarrow{\text{С6}} s_m = s'$ $(m \geqslant 1)$, то $s \to s'$.

Отождествляя пару (b, P) с тройкой (b, ϕ, P) и перенося отношение ρ на множество пар (b, P), получаем, что система Γ_0/ρ вкладывается в Γ_1 как подсистема. Это значит, что из $s \underset{\Gamma_0/\rho}{\Longrightarrow} s'$ вытекает $s \underset{\Gamma_1}{\Longrightarrow} s'$. Проверив, что $(b, Z, P) \to (b', Z', P') \Rightarrow bZP = b'Z'P'$, получим теорему.

Теорема 5.5. $bP = b' \Longleftrightarrow (b, \phi, P) \underset{\Gamma_1}{\Longrightarrow} (b', \phi, \epsilon)$.

Множество Z в состоянии (b, Z, P) системы Γ_1 – это множество выполняемых параллельно базовых операторов. Переходы по правилу С1 предоставляют собой инициализацию выполнения одного из операторов, который может быть выполнен, переходы по правилу С6 – завершение одного из выполняемых операторов. Правило С7 показывает, что инициализация одного и завершение нескольких других базовых операторов могут происходить одновременно. Система Γ_1 является недетерминированной и среди различных процессов вычислений, подтверждающих переход $(b, \phi, P) \Rightarrow (bP, \phi, \epsilon)$, содержит много процессов, допускающих параллельное выполнение нескольких операторов. При этом допускается, вообще говоря, неограниченное распараллеливание, т.е. множество Z может быть сколь угодно большим. Для примера рассмотрим программу умножения матриц:

ДЛЯ i := 1 ДО n ВЫПОЛНИТЬ
ДЛЯ j := 1 ДО n ВЫПОЛНИТЬ
    c_{ij} := 0;
    $P(i, j)$
КЦ
КЦ

где $P(i, j)$ – базовый оператор, эквивалентный оператору

ДЛЯ k := 1 ДО n ВЫПОЛНИТЬ
    $c_{ij} := c_{ij} + a_{ik} \cdot b_{kj}$
КЦ

Для любых конкретных значений d_1 и d_2 переменных i и j обозначим через $P(d_1, d_2)$ оператор, который получается подстановкой этих значений вместо i и j в оператор $P(i, j)$. Переменная k рассматривается как локализованная внутри $P(i, j)$. В качестве ρ возьмем отношение, которое включает в себя соотношение

$$(b, Z, P(i, j)Q) = (b, Z, P(b(i), b(j))Q),$$

при условии что операторы из Z не изменяют переменные i и j. Цикл вида ДЛЯ $i := 1$ ДО n ВЫПОЛНИТЬ P КЦ рассматривается как сокращение регулярной программы

НАЧАЛО
  $i := 1$;
  ПОКА $i \leqslant n$ ВЫПОЛНИТЬ
    P; $i := i + 1$;
  КЦ
КОНЕЦ

Поскольку для различных значений пары (i, j) операторы $P(i, j)$ информационно независимы и могут выполняться одновременно, система Γ_1 допускает в этом случае неограниченное распараллеливание.

Реализация системы Γ_1 требует определения функций In и Out отношения ρ и выбора некоторых допустимых процессов, которые будут реализовываться. При этом должна быть обеспечена полнота, т.е. для любых b и P, если $bP = b'$, то хотя бы один процесс, подтверждающий переход $(b, \phi, P) \underset{\Gamma_1}{\Longrightarrow} (b', \phi, \epsilon)$, должен иметь реализацию. Практически функции In и Out и отношение ρ выбираются с учетом требований быстрой вычислимости, а выбор реализующих процессов — с учетом имеющихся в наличии ресурсов (процессоров). Базовые операторы могут быть представлены программами, которые выполняются в различных процессорах. Управляющая программа в этом случае выполняет только действия, связанные с проверкой условий и вычислениями, соответствующими заголовкам циклов. Кроме того, в задачи управляющей программы входит распределение данных между процессорами.

Некоторое, практически оправданное усложнение системы дает возможность увеличить степень распараллеливаемости за счет расчленения на отдельные переходы выполнения базовых операторов. Это достигается добавлением к переходу по правилу С6 дополнительной возможности:

$$(b, Z, P) \to (by', (Z \setminus \{y\}) \cup y'', P),$$

где $y = y'y''$, $\mathrm{In}(y'', b) \subset \mathrm{In}(y, b)$, $\mathrm{Out}(y'', b) \subset \mathrm{Out}(y, b)$. В реализации выполнение подобного перехода соответствует, например, передаче в управляющую программу сообщения о том, что часть y' оператора y завершила свою работу. Уменьшение множеств входных и выходных переменных разрывает некоторые информационные зависимости.

Комментарии к главе 7

Текст главы написан по работам [22, 52, 64, 69, 70].

СПИСОК ЛИТЕРАТУРЫ

1. Автоматы. – М.: Мир, 1956. – 404 с.
2. *Агафонов В.Н.* Спецификация программ: понятийные средства и их организация. – Новосибирск: Наука, 1987. – 238 с.
3. *Арнольд В.И.* Обыкновенные дифференциальные уравнения. – М.: Наука, 1975. – 240 с.
4. *Ахо А., Ульман Дж.* Теория синтаксического анализа, перевода и компиляции. – М.: Мир, 1978. – Т. 1. – 612 с.; Т. 2. – 488 с.
5. *Ахо А., Хопкрофт Дж., Ульман Дж.* Построение и анализ вычислительных алгоритмов. – М.: Мир, 1979. – 536 с.
6. *Байцер Б.* Архитектура вычислительных комплексов. – М.: Мир, 1974. – Т. 1. – 468 с.; Т. 2. – 566 с.
7. *Бублик В.В., Гороховский С.С.* Алгебраическая трактовка структур данных// Кибернетика. – 1978. – № 2. – С. 10–15.
8. *Глушков В.М.* Абстрактная теория автоматов//УМН. – 1961. – № 5. – С. 3–62.
9. *Глушков В.М.* Алгебра алгоритмов и динамическое распараллеливание последовательных программ//Кибернетика. – 1982. – № 5.
10. *Глушков В.М.* Вопросы теории практики. Кибернетика. – М.: Наука, 1986. – 488 с.
11. *Глушков В.М.* К вопросу о минимизации микропрограмм и схем алгоритмов// Кибернетика. – 1966. – № 5. – С. 1–3.
12. *Глушков В.М.* О применении абстрактной теории автоматов для минимизации микропрограмм//Изв. АН СССР. Технич. кибернетика. – 1964. – № 1. – С. 3–8.
13. *Глушков В.М.* Основные архитектурные принципы повышения производительности ЭВМ. Проблемы ВТ. – М.: МЦНТИ, 1981. – С. 6–21.
14. *Глушков В.М.* USSR, Computing in – In // Encyclopedia of Computer Sience and Technology. – N–Y: Dekker, 1979. – V. 13. – P. 498–507.
15. *Глушков В.М.* Синтез цифровых автоматов. – М.: Физматгиз, 1962. – 476 с.
16. *Глушков В.М.* Теория автоматов и вопросы проектирования структуры цифровых машин//Кибернетика. – 1965. – № 1. – С. 3–11.
17. *Глушков В.М.* Теория автоматов и формальные преобразования микропрограмм// Кибернетика. – 1965. – № 5. – С. 1–9.
18. *Глушков В.М., Игнатьев М.Б.* и др. Рекурсивная машина. – Препринт/ИК АН УССР. – Киев, 1974. – № 74–57.
19. *Глушков В.М., Капитонова Ю.В., Летичевский А.А.* Автоматизация проектирования вычислительных машин. – Киев: Наукова думка, 1975. – 232 с.
20. *Глушков В.М., Капитонова Ю.В., Летичевский А.А.* Инструментальные средства проектирования программ обработки математических текстов//Кибернетика. – 1970. – № 2.
21. *Глушков В.М., Капитонова Ю.В., Летичевский А.А.* К теории проектирования схемного и программного оборудования многопроцессорных ЭВМ//Кибернетика. – 1978. – № 6. – С. 1–15.

22. *Глушков В.М., Капитонова Ю.В., Летичевский А.А., Горлач С.П.* Макроконвейерные вычисления функций над структурами данных//Кибернетика. – 1981. – № 4. – С. 13–21.
23. *Глушков В.М., Капитонова Ю.В., Летичевский А.А.* О применении метода формализованных технических заданий к проектированию программ обработки структур данных//Программирование. – 1978. – № 6.
24. *Глушков В.М., Капитонова Ю.В., Летичевский А.А.* и др. ПРОЕКТ–ЕС. Программирование на языке L2B. – Препринт / ИК АН УССР. – Киев, 1979. – № 79–23.
25. *Глушков В.М., Капитонова Ю.В., Летичевский А.А.* Теоретические основы проектирования дискретных систем//Кибернетика. – 1977. – № 6. – С. 5–20.
26. *Глушков В.М., Капитонова Ю.В., Летичевский А.А.* Теория структур данных и синхронные параллельные вычисления//Кибернетика. – 1976. – № 6. – С. 2–15.
27. *Глушков В.М., Летичевский А.А.* Теория автоматов и программирование//Пленарные докл. 1 Всес. конф. по программированию. – Киев: ИК АН УССР, 1968. – С. 3–19.
28. *Глушков В.М., Летичевский А.А.* Теория дискретных преобразователей//Избранные вопросы алгебры и логики. – Новосибирск: Наука, 1973. – С. 5–39.
29. *Глушков В.М., Летичевский А.А., Годлевский А.Б.* Методы синтеза дискретных моделей биологических систем//Методы математической биологии. Кн. 6. – Киев: Высшая школа, 1983. – 264 с.
30. *Глушков В.М., Рабинович З.Л., Барабанов А.А.* и др. Вычислительные машины с развитыми системами интерпретации. – Киев: Наукова думка, 1970. – 259 с.
31. *Глушков В.М., Цейтлин Г.Е., Ющенко Е.Л.* Алгебра, языки, программирование. – Киев: Наукова думка, 1974. – 328 с.
32. *Головкин Б.А.* Параллельные вычислительные системы. – М.: Наука, 1980. – 519 с.
33. *Гончаров С.С., Свириденко Д.И.* Σ-программирование//Логико-математические проблемы МОЗ (Вычислительные системы. Вып. 107). – Новосибирск, 1985. – С. 3–29.
34. *Гороховский С.С., Капитонова Ю.В., Летичевский А.А.* О средствах программирования и решения логических задач в системах математического обеспечения (основные понятия языка L2) //Кибернетика. – 1973. – № 4.
35. *Горшков В.П.* О соотношениях в алгебрах структур данных//Кибернетика. – 1978. – № 3. – С. 24–32.
36. *Горшков В.П., Горлач С.П.* Об определяющей совокупности соотношений в многоосновной алгебре структур данных//Кибернетика. – 1982. – № 1. – С. 124–126.
37. *Дал У., Дейкстра Э., Хоор К.* Структурное программирование. – М.: Мир, 1978. – 275 с.
38. Данные в языках программирования. – М.: Мир, 1982. – 328 с.
39. *Дейкстра Э.* Дисциплина программирования. – М.: Мир, 1975. – 247 с.
40. *Ершов А.П.* Научные основы доказательного программирования. – М.: Вестник АН СССР, 1984. – № 10. – С. 9–19.
41. *Ершов А.П.* Операторные алгоритмы 1//Проблемы кибернетики. Вып. 3. – М.: Физматгиз, 1960. – С. 5–48.
42. *Ершов А.П.* Современное состояние схем программ//Проблемы кибернетики. Вып. 27. – М.: Наука, 1973. – С. 87–110.
43. *Ершов Ю.Л.* Теория *A*-пространств//Алгебра и логика. – 1973. – № 4. – С. 369–416.
44. *Ершов А.П.* Трансформационный подход в технологии программирования//Технология программирования. – Киев: ИК АН УССР, 1979. – С. 12–26. – (Тезисы докл. 1 Всес. конф.)
45. *Ершов А.П., Сабельфельд В.К.* Очерк схемной теории рекурсивных программ// Трансляция и модели программ. – Новосибирск, 1980. – С. 23–43.
46. *Ершов А.П.* и др. Методика разработки многоязыковых трансляторов н примере системы БЕТА//Математическая теория и практика систем программного обеспечения. – Новосибирск, 1982. – С. 64–80.
47. *Калман Р., Фалб П., Арбиб М.* Очерки по математической теории систем. – М.: Мир, 1971. – 400 с.
48. *Капитонова Ю.В.* Дискретные системы и задачи их реализации//Кибернетика. – 1975. – № 4, 5.

49. *Капитонова Ю.В.* Об аналитических преобразованиях с помощью ЭВМ//Кибернетика. – 1985. – № 1.
50. *Капитонова Ю.В.* Теоретические модели вычислений//Энциклопедия по вычислительной науке и технике. – Нью-Йорк, 1970.
51. *Капитонова Ю.В., Гороховский С.С., Летичевский А.А.* и др. Алгоритмический язык МАЯК//Кибернетика. – 1984. – № 3. – С. 54–74.
52. *Капитонова Ю.В., Летичевский А.А., Горлач С.П.* Алгебра структур данных и синтез параллельных программ//Всесоюзная конф. по прикладной логике. – Новосибирск, 1985.
53. *Капитонова Ю.В., Летичевский А.А., Гороховский С.С.* и др. ПРОЕКТ–ЕС. Базовый инструментальный язык программирования. – Препринт/ИК АН УССР. – Киев, 1979. – № 79–22.
54. *Касьянов В.Н.* Введение в теорию оптимизации программ. – Новосибирск: ВЦ СО АН СССР, 1985. – 258 с.
55. *Клини С.К.* Ввдение в метаматематику. – М.: ИЛ, 1957. – 526 с.
56. *Королев Л.Н.* Структура ЭВМ и их математического обеспечения. – М.: Наука, 1978. – 351 с.
57. *Котов В.Е.* Введение в теорию схем программ. – Новосибирск: Наука, 1978. – 258 с.
58. *Котов В.Е.* Сети Петри. – М.: Наука, 1984. – 160 с.
59. *Кудрявцев В.Б., Алешин С.В., Подколзин А.С.* Введение в теорию автоматов. – М.: Наука, 1985. – 320 с.
60. *Летичевский А.А.* Алгебры с аппроксимацией и рекурсивные структуры данных // Кибернетика. – 1987. – № 5.
61. *Летичевский А.А.* Об ускорении итерации монотонных операторов // Кибернетика. – 1976. – № 4. – С. 1–7.
62. *Летичевский А.А., Годлевский А.Б., Кривой С.Л.* Об эффективном алгоритме построения базиса подгруппы свободной группы // Кибернетика. – 1981. – № 3. – С. 107–116.
63. *Летичевский А.А., Горлач С.П.* Многоосновная алгебра структур данных // Математическое обеспечение систем логического вывода и дедуктивных построений на ЭВМ. – Киев: ИК АН УССР, 1983. – С. 18–31.
64. *Летичевский А.А., Горлач С.П.* Укрупнение структур данных и синтез параллельных программ // Анализ и обработка математических текстов. – Киев: ИК АН УССР, 1984. – С. 34–43.
65. Логическое программирование. – М.: Мир, 1986.
66. *Ляпунов А.А.* О логических схемах программ // Проблемы кибернетики. Вып. 1. – М.: Физматгиз, 1958. – С. 46–74.
67. *Мальцев А.И.* Алгоритмы и рекурсивные функции. – М.: Наука, 1965. – 392 с.
68. *Манна З.* Теория неподвижной точки программ // Кибернетический сб. – Новая серия. Вып. 15. – С. 38–100.
69. *Михалевич В.С., Капитонова Ю.В., Летичевский А.А.* О методах организации макроконвейерных вычислений // Кибернетика. – 1986. – № 3. – С. 3–10.
70. *Михалевич В.С., Капитонова Ю.В., Летичевский А.А., Молчанов И.Н., Погребинский С.Б.* Организация вычислений в многопроцессорных вычислительных системах // Кибернетика. – 1984. – № 3. – С. 1–10.
71. *Непомнящий В.А.* Практические методы верификации программ // Кибернетика. – 1984. – № 2. – С. 21–28, 43.
72. *Плюшкявичус Р.А., Плюшкявиченс А.Ю., Саклаускайте Ю.В., Юкна С.П.* О программных логиках // Кибернетика. – 1979. – № 2. – С. 12–19.
73. *Попов Э.В., Фирдман Г.Р.* Алгоритмические основы интеллектуальных роботов. – М.: Наука, 1977. – 456 с.
74. *Скотт Д.* Теория решеток, типы данных и семантика // Данные в языках программирования. – М.: Мир, 1982. – С. 25–53.
75. Электронные вычислительные машины. – Киев: Наукова думка, 1974.
76. *Ющенко Е.Л.* Адресное программирование. – Киев: Техиздат, 1963. – 288 с.
77. *Ющенко Е.Л., Касаткина И.В.* Современные методы доказательства правильности программ // Кибернетика. – 1980. – № 6. – С. 37.
78. *Яблонский С.В.* Введение в дискретную математику. – М.: Наука, 1979. – 272 с.

79. *Яблонский С.В.* Основные понятия кибернетики // Проблемы кибернетики. Вып. 2. – М.: Наука, 1959. – С. 7–38.
80. *Янов Ю.И.* О логических схемах алгоритмов // Проблемы кибернетики. Вып. 1. – М.: Физматгиз, 1958. – С. 75–127.
81. *Bauer F.L.* Program development by stepwise transformations. The project CIP // Lecture Notes on Comp. Sci, 1979. – V. 69. – P. 237–272.
82. *Böhm C., Jacopini G.* Flow Diagrams, Turing Machines and Languages with only Two Formation Rules. – CACM, May 1966. – P. 366–371.
83. *Dijkstra E.W.* Go–To Statement Considered Harmful Letter to Editor. – CACM, March 1968.
84. *Floyd R.W.* Assing meaning to programs // Proc. Symp. in Appl. Math., 1967. – P. 19–32.
85. *Glushkov V.M., Letichevsky A.A.* Theory of algorithms and discrete processors in Advances in information system sciences. – 1969. – N 1. – P. 1–58.
86. *Goguen G.A.* Gr. On homomorphisms, correctness termination unfoldments and equivalence of flow programms diagram. Journal of computer and system sciences. – 1974. – N 2. – P. 333–365.
87. *Hoare C.A.R.* An Axiomatic basic for computer programming. – CACM, 1969. – N 10. – P. 576–580, 583.
88. *Keller R.M.* A fundamental theorem of asynchronous parallel computation. – LNCS, 24 Parallel processing, 1975.
89. *Martelli A., Rossi G.* Efficient unifaction with Enfinite terms in logic programming in Proc. of the Int Conf. on 5th Generation Computer Systems. – Tokyo: Japan Nov., 1984. – N 6–9.
90. *McCarthy J.* Recursive functions of symbolic expressions. – CAMC, April 1960. – P. 184–185.
91. *Pawlak Z.* Maszyny programowane "Algorytmy". – 1969. – N 10. – P. 7–22.
92. *Schwartz I.T.* On programming: an interin report on the SETL projet Caurant. – N–Y.: Inst. of Math. Sci., 1975.
93. *Wegner P.* Programming languages – the first 25 years IEEE Transactions on Computers Dec. – 1976. – P. 1207–1225.

Капитонова Юлия Владимировна
Летичевский Александр Адольфович

МАТЕМАТИЧЕСКАЯ ТЕОРИЯ ПРОЕКТИРОВАНИЯ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

Редактор *Т.В. Шароватова*
Художественный редактор *Т.Н. Кольченко*
Технические редакторы: *С.Н. Баронина, О.Б. Черняк*
Корректоры: *Н.П. Круглова, Т.В. Обод*

Набор осуществлен в издательстве
на наборно-печатающих автоматах

ИБ № 32353

Сдано в набор 23.11.87. Подписано к печати 08.04.88. Т-09551
Формат 60 X 90/16. Бумага для множительных аппаратов
Гарнитура Пресс-Роман. Печать офсетная
Усл.печ.л. 18,5. Усл. кр.-отт. 18,5. Уч.-изд.л. 22,32
Тираж 7600 экз. Тип. зак. 1184 Цена 2 р. 50 к.

Ордена Трудового Красного Знамени
издательство "Наука"
Главная редакция физико-математической литературы
117071 Москва В-71, Ленинский проспект, 15

Четвертая типография издательства "Наука"
630077 г. Новосибирск-77, ул. Станиславского, 25