А. П. Ершов,

член-корреспондент АН СССР

ТЕОРИЯ ПРОГРАММИРОВАН И ВЫЧИСЛИТЕЛЬН СИСТЕМЫ

ИЗДАТЕЛЬСТВО «ЗНАНИЕ»
МОСКВА 1972

Ершов Андрей Петрович

Е 80 Теория программирования и вычислительные системы. М., «Знание», 1972

64 стр. (Новое в жизни, науке и технике). Серия «Математика, кибернетика», 6.

В брошюре приведен обзор основного раздела теоретического программирования — теории схем программ, обсуждается фундаментальная задача теории программирования и анализируются основные теории параллельного программирования. В качестве вывода высказывается гипотеза о целесообразной структуре ЭВМ четвертого поколения. В заключительной главе изложен подход к реализации алгоритмических языков нового поколения — Алгола 68, ПЛ/1, Симулы 67.

Брошюра рассчитана на широкий круг читателей, занимающихся программированием и интересующихся перспективами его развития.

3—3—14 6ф7.3

ВВЕДЕНИЕ

Изучая некоторую техническую дисциплину, связанную с тем или иным видом созидательной деятельности человека, обычно различают теорию и технологию изучаемой деятельности. Теория исследует фундаментальные компоненты явлений и предметов, вовлекаемых в этот вид деятельности, рассматривая их в связи с общими законами природы и их математическим выражением. Технология, опираясь на доставляемые теорией сведения, берет те же компоненты в связи с конечным результатом изучаемой деятельности, придавая ей характер устойчивого, целеустремленного и эффективного процесса.

Теория и технология развиваются каждая по своим законам и часто — до поры до времени — независимо друг от друга. Технология возникает сначала как совокупность эмпирических наблюдений и рецептурных приемов; теория рождается иногда на базе спекулятивных идей, случайных гипотез и слишком грубых моделей и не оказывает поначалу никакого влияния на технологию, в свою очередь, не получая от нее достаточной пищи для размышлений. Иногда разрыв между теорией и технологией затягивается во времени просто потому, что этими разделами занимаются разные люди с разными «интеллектуальными биографиями». Однако в развитии неизбежно наступает момент, когда наука и техника объединяются, взаимно оплодотворяя друг друга, после чего в данной отрасли техники наступает бурный прогресс, который быстро превращает молодую техническую дисциплину в зрелую и развитую промышленную отрасль, эффективно обслуживающую человечество в течение долгого времени. В качестве примера таких видов человеческой деятельности, где достигнута гармоничная и плодотворная связь теории и технологии, автор мог бы назвать авиацию, некоторые виды машиностроения, электротехнику и энергетику.

Программирование для электронных вычислительных ма-

шин (ЭВМ), которому, как и самим ЭВМ, недавно минуло 25 лет, уже в недалеком будущем станет одним из массовых видов человеческой деятельности, притом не только профессиональной. В настоящее время, однако, невзирая на свое бурное развитие и популярность, программирование еще далеко от того, чтобы называться зрелой наукой или тщательно разработанной и стабильной технологией.

Одним из выражений этой незрелости до последнего времени являлся уже упоминавшийся разрыв между теорией и технологией; при этом теория программирования страдала от худосочия и неразвитости математического аппарата, а технология носила случайный характер, закрепляя зачастую привычки и вкусы вместо знания и опыта.

Сейчас, по мнению автора, в программировании созревают предпосылки для его быстрого превращения в развитую и стабильную отрасль человеческой деятельности. Одной из этих предпосылок является установление прочных связей между теорией и технологией программирования. В теории программирования завершается строительный период, и воздвигаемое здание теории сможет предоставить свои площади и «коммунальные услуги» для объективного и полного обоснования методики программирования. Со своей стороны практика программирования выдвигает актуальные и трудные задачи, которые в силу своей сложности просто не могут быть решены чисто эмпирическим путем. Обоснованию и обсуждению складывающейся ситуации в программировании и посвящена эта брошюра.

За последнее время автору довелось несколько раз выступать с общими докладами, имеющими прямое отношение к обсуждаемой проблеме. В этих докладах делался обзор некоторых результатов теории программирования, и с позиций этих результатов делалась попытка обоснованной постановки актуальных задач программирования. Эти выступления были благожелательно встречены слушателями, что и побудило автора рискнуть выступить с ними перед более широкой читательской аудиторией.

В первой главе сделан обзор основных работ по теории схем программ — основного аппарата теоретического программирования. В конце главы дается постановка фундаментальной задачи теории программирования и кратко обсуждаются предпосылки к ее решению и возможные следствия.

Во второй главе проведен анализ существующих теорий параллельного программирования. Под этим названием имеется в виду методика программирования для вычислительных систем, состоящих из нескольких или даже многих параллельно работающих вычислителей, способных объединяться для решения одной задачи. На базе сделанного анализа высказывается гипотеза о целесообразной структуре ЭВМ четвертого поколения и предлагается определенная этапность в работах по параллельному программированию.

В третьей главе излагается подход к решению актуальной и сложной задачи — реализации алгоритмических языков нового поколения, таких, как Алгол 68, ПЛ/1, Симула 67. Высказывается идея построения универсального программирующего процессора, настраиваемого автоматически на тот или иной входной язык с помощью языковых таблиц. Такой подход сможет быть эффективен только в том случае, если в процессоре будут реализованы мощные универсальные алгоритмы оптимизации, создаваемые теорией программирования. Показывается, что этого можно достичь, введя в качестве посредника между входными и машинными языками так называемый внутренний язык. Некоторые особенности внутреннего языка также обсуждаются в заключительной главе.

Объединенные некоторой общей идеей, главы были написаны в разное время и по разным поводам. Автор приносит свои извинения читателям за вызванные этим некоторые неровности стиля, устранение которых потребовало бы коренной переработки текста. В брошюре значительное место занимает обсуждение математических вопросов, однако автор старался вести изложение на уровне, не требующем от читателя специальных знаний, за исключением некоторых начальных понятий дискретной математики (граф, логическая функция, предикат, алгоритм и т. п.).

ГЛАВА 1.

ТЕОРИЯ СХЕМ ПРОГРАММ

В настоящей главе[1] будет рассмотрено некоторое направление теории программирования, которое восходит к классической работе Ю. И. Янова [10], положившей начало исследованию эквивалентности и формальных преобразований схем программ. Это направление рассматривается не только потому, что оно близко личным научным интересам автора, и не только потому, что вклад советских математиков в него особенно существенен. Это направление рассматривается прежде всего потому, что, по мнению автора, в его недрах созревает решение некоторых фундаментальных проблем теории программирования, что не только увеличит степень нашего знания природы вещей, но и через создание некоторого нового формального алгоритмического языка сможет непосредственно повлиять на технологию разработки систем программирования, а также дать в руки программистам средства программирования машин последующих поколений — многопроцессорных параллельных вычислительных систем. Сделать эту ситуацию явной и привлечь к ней внимание — вот цель этого рассмотрения. Для тех, кто следит за работами в области теории программирования, автор рекомендовал бы сравнить главу с обзором, относящимся к 1967 [3]. Сравнение покажет, как многое изменилось за прошедшие четыре года.

---

[1] Материалом для первой главы послужил доклад автора, сделанный на Конгрессе Международной федерации по обработке информации (Любляна, Югославия, август 1971 г.)

§ 1. Предмет теории программирования

В настоящее время теория программирования выросла в довольно развитую математическую дисциплину, имеющую некоторую внутреннюю структуру. Рассмотрим основные понятия теории программирования.

Программы. Определение программ начинается с описания некоторых конструктивных объектов, представляющих программы. Такие конструктивные объекты — слова (тексты), помеченные графы, некоторые конечные множества и системы множеств. Принято говорить о языке описания программы; в этом случае способ конструирования объектов, представляющих программы, называют синтаксисом данного языка описания.

Выполнение. Неотъемлемой частью формального определения программ является понятие выполнения программы. Это понятие вводится путем задания некоторого универсального алгоритма, применяемого к программам. Процесс, осуществляемый при применении этого алгоритма к некоторой программе, и называется выполнением программы.

Процесс выполнения программы может иметь некоторое количество свободных параметров. Часть этих свободных параметров относится к выполняемой программе. Обычно эти параметры являются аргументами программы, задающими ее входные данные. В этом случае выполнение некоторой фиксированной программы становится функцией ее входных данных. Иногда свободные параметры может иметь сам универсальный алгоритм. В этом случае фиксированная программа, даже при фиксированных своих аргументах задает целое множество выполнений, получаемое варьированием параметров универсального алгоритма. Такой подход характерен для некоторых задач теории параллельного программирования.

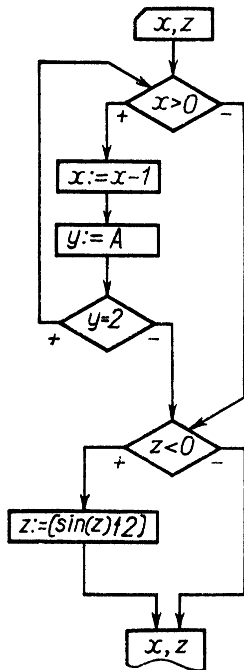
Выполнение программы рассматривается как процесс, протекающий во времени. Процесс представляет собой последовательность некоторых событий. Обычно считается, что события происходят мгновенно на стыках временных интервалов. Всегда существует заключительное событие, наступление которого прекращает выполнение программы. Среди заключительных событий выделяются результативные события, которые позволяют объявить некоторый объект результатом выполнения программ. Тем самым с программой может быть связана некоторая функция, вычисляемая (или реализуемая) данной программой. Аргументами этой функции являются входные данные программы. Результатом функции для данного аргумента является результат выполнения программы, соответствующего этому аргументу.

Рассмотрим некоторые общие черты разных определений

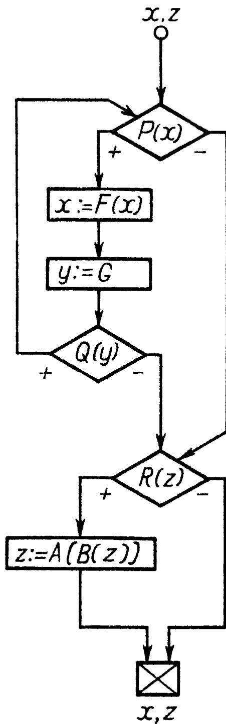
программ и их выполнения. Выполнение программы представляет собой последовательность преобразований информации. Тем самым понятие программы обычно состоит из двух частей — операционной и информационной. Эти части не обязательно различимы на уровне синтаксиса (например, в памяти машины нет разницы между командами и перерабатываемой информацией), однако на уровне выполнения программы всегда рассматривается некоторая единица действия по переработке некоторых единиц информации. Объекты, задающие единицы действия, называются командами, или операторами. Способность программы к хранению информации выражается в наличии в ней памяти. Память в программах обычно имеет некоторую структуру, позволяющую операторам программы действовать не на всю память, а лишь на некоторые ее компоненты, называемые ячейками, или величинами. Существенное место в формализме программ занимают средства идентификации компонент памяти.

Выполнение оператора в общем случае состоит в переработке информации, хранящейся в ячейках памяти, и в определении оператора, выполняющегося вслед за данным. Информация, выработанная данным оператором, может быть использована некоторыми операторами, выполняемыми позднее данного. Тем самым между операторами устанавливаются два типа бинарных отношений (оператор B выполняется непосредственно вслед за оператором A; оператор B воспринимает в качестве аргументов результаты оператора A). Важным обобщением этих отношений по передаче управления и передаче информации являются их «транзитивные замыкания» (оператор B может выполняться позже оператора A; аргументы оператора B функционально зависят от результатов оператора A). Расчленение программы на операторы, действующие над памятью, и рассмотрение системы управляющих и информационных связей между операторами — вот, пожалуй, наиболее общие и характерные черты существующих формализмов в теории программирования.

На рис. 1, *а* показан пример конкретной программы. Операторы программы образуют так называемый управляющий граф. В графе имеются входная и выходная вершины. В них могут находиться операторы ввода аргументов и вывода результатов выполнения программы. Каждая вершина, кроме выходной, имеет одного или двух преемников. В первом случае она называется, следуя Калужнину, преобразователем, во втором — распознавателем. Преобразователем может быть оператор типа оператора присваивания, строящийся над некоторой базисной системой элементарных операций. Распознавателем является предикат — логическая функция, имеющая вид условия (в языках типа Алгол и Фортран). Аргументы и результаты операторов замещены переменными — ячейками памяти, хранящими результаты вычислений и реализующими информационные связи между операторами.

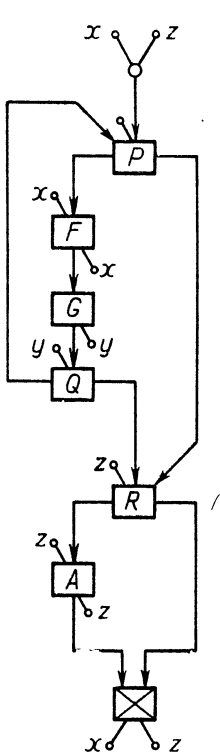


а) программа

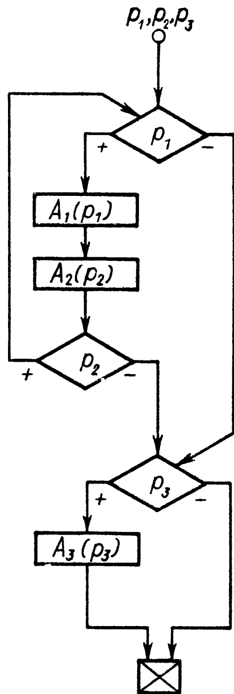


б) стандартная схема

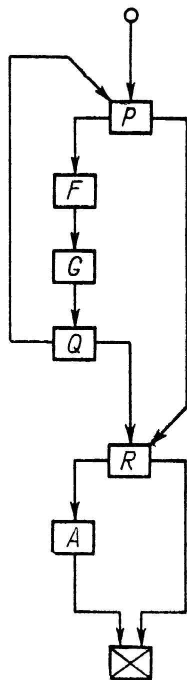
Рис. 1. Примеры схем программ.



в) схема Лаврова



г) схема Янова



д) схема Мартынюка

Эквивалентность. Третьей существенной частью формализма понятия программы является эквивалентность программ. В том случае, когда программа реализует некоторую функцию (а это, как правило, так), имеет место естественное и наиболее общее определение эквивалентности. Мы говорим, что две программы, имеющие общее множество аргументов, функционально эквивалентны, если реализуемые ими функции совпадают.

Однако на пути к созданию теории такой степени общности неустранимым препятствием стоит следующий отрицательный результат теории алгоритмов. Некоторое свойство программ называется внутренним, если оно присуще всем функционально эквивалентным между собой программам. Райс доказал, что каково бы ни было внутреннее свойство программ, не существует алгоритма, который среди всех программ отличал бы программы, обладающие данным свойством (естественно, что класс программ должен быть достаточно содержательным, например, он должен вычислять любые рекурсивные функции).

Основным методом сужения понятия эквивалентности программ является сравнение не только значений функций, реализуемых программами, но и некоторой истории их вычисления в процессе выполнения. Формально понятие истории вводится следующим образом. Дополнительно к универсальному алгоритму выполнения программ вводится еще один общий алгоритм (назовем его следящим алгоритмом), который по программе и набору ее входных данных строит некоторый объект, называемый историей реализации программы и содержащий ту или иную информацию о ходе ее выполнения. История может быть любой степени детальности, лишь бы по ней однозначно восстанавливался результат выполнения программ. Тем самым программы с совпадающими историями автоматически имеют совпадающие результаты. Вырожденным случаем истории является сама программа (тождественный следящий алгоритм). Эта история является, естественно, наиболее детальной историей, так как из программы можно получить любую информацию, применяя к ней алгоритм выполнения. Эквивалентность, определяемая на основе этой истории, оказывается наиболее узкой: программа эквивалентна только самой себе. Между этими двумя полюсами лежит множество различных определений эквивалентности, отличающихся степенью детальности рассматриваемых историй программ.

Приводим основные типы историй, рассматриваемые в теории:

— операционная (*О*) история программы. Под этим подразумевается запись последовательности преобразователей, выполненных при работе программы. Иногда эта последовательность сопровождается последовательностью состояний памяти после выполнения очередного оператора;

— операционно-логическая (*ОЛ*) история программы допол-

нительно указывает на все распознаватели, пройденные при работе программы;

— информационный граф (*ИГ*) реализации программы. Вершинами графа являются преобразователи. Вершина *A* соединяется дугой с вершиной *B*, если *B* использует результат *A* в качестве аргумента;

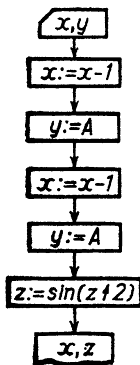
— информационно-логический граф (*ИЛГ*) реализации программы. К информационному графу добавляются вершины проходимых распознавателей. Их аргументы соединяются с соответствующими результатами информационными связями. Кроме того, каждый распознаватель соединяется связью особого рода с каждой вершиной информационно-логического графа, прохождение или непрохождение которой непосредственно зависит от выполнения данного распознавателя;

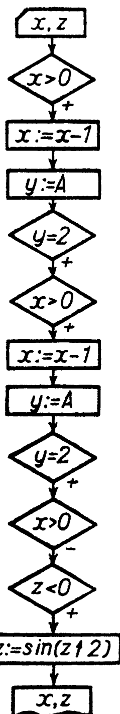
— термальное (*T*) значение результативных переменных. Очевидно, что информационный граф — это граф, не содержащий циклов. Это позволяет для каждого аргумента *a* любой вершины *V* графа построить некоторое дерево, которое включает в себя все вершины, достижимые из *V* при движении навстречу дугам информационных связей. При этом если некоторая вершина достижима по двум путям, то она расклеивается на две. Аргумент становится при этом корнем получившегося дерева. Это дерево, которое алгебраически можно записать в виде выражения, или терма, называется, следуя Иткину, термальным значением аргумента *a*. Продвигаясь «вниз» по информационному графу, мы получим в конце концов термальные значения результативных переменных;

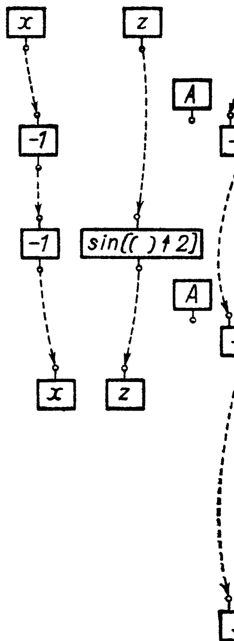
— логико-термальная (*ЛТ*) история реализации получается добавлением к термальному значению результативных переменных цепочки пройденных распознавателей вместе с термальными значениями их аргументов. Логико-термальная история строится по информационно-логическому графу.

На рис. 2 показаны примеры указанных историй для некоторой реализации программы из рис. 1, *а*. Мы видим, что эти истории группируются парами. Каждая последующая пара содержит меньше информации о реализации, чем предыдущая. Каждая вторая история в паре некоторым образом фиксирует историю работы распознавателей. Для восстановления результатов по исходным данным достаточно только информационной истории. Однако, как вытекает из многочисленных результатов, которые будут обсуждаться ниже, попытка ограничиться только информационной историей, даже самой детальной, дает алгоритмически неразрешимую эквивалентность. Кроме того, в ряде случаев порядок выполнения распознавателей в программе является существенной характеристикой реализации программы.

Схемы программ. Учет более подробной истории вычислений

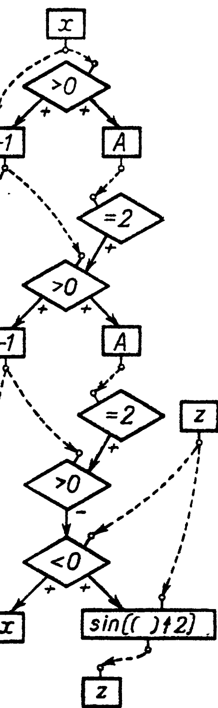






I, а) О–история I, б) ОЛ–история II, а) ИГ–история

Рис. 2. Примеры историй.



II, б) ИЛГ-история

III, а) Т-история

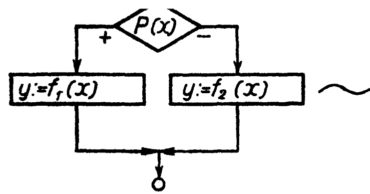
III, б) ЛТ-история

сам по себе является еще недостаточным для построения эффективных методов распознавания эквивалентности и формального преобразования программ. Упомянутые результаты приводят к еще более суровым выводам: можно построить такой реальный класс программ, в котором ни одна из известных нетривиальных эквивалентностей не будет алгоритмически разрешима. Поэтому для получения более конструктивных и конкретных результатов приходится жертвовать понятием программы во всей его всеобщности и рассматривать более простые и абстрактные объекты. Такими объектами в первую очередь являются так называемые схемы программ, или абстрактные программы. Они сохраняют значительную часть структурных свойств программ, в частности, членение на операторы с указанием информационных и управляющих связей между ними. Это позволяет для схем/программ строить многие из характеристик, присущих конкретным программам, например, те или иные реализации схем программ. В схемах программ вместо переменных, операций и предикатов выступают формальные символы, не имеющие конкретных внутренних свойств. В этих формальных объектах удерживается только та информация, которая нужна для построения реализаций или их историй; например, для формальных операций указываются число формальных аргументов и имена подставляемых туда формальных переменных, для формального оператора передачи управлений указываются метки тех операторов, куда может быть передано управление, и т. п. На рис. 1, *б* — *д* показаны примеры схем программ с рис. 1, *а*.

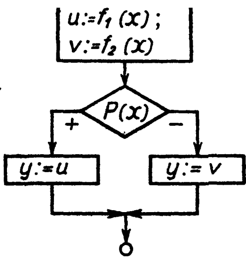
Конкретные программы получаются из схем той или иной интерпретацией, которая состоит в сопоставлении формальным переменным и операциям некоторых конкретных переменных и операций. Очень важным понятием является понятие множества I всех интерпретаций схемы программы. Теория строится так, что бы факт, установленный для некоторой схемы, был бы справедлив для любой интерпретирующей ее программы. В частности, так вводится отношение эквивалентности двух схем программ: две схемы программ S_1 и S_2 эквивалентны в смысле истории H, если для любой интерпретации i из I конкретные программы $S_1(i)$ и $S_2(i)$ будут эквивалентны в смысле этой же истории.

Для того чтобы подчеркнуть глубину подхода к определению эквивалентности как инварианта любой интерпретации схем, на рис. 3 приведены два примера четырех пар схем программ, в которых схемы *б* эквивалентны в любой интерпретации, а схемы *а* — лишь в некоторых. Там же указаны примеры интерпретаций, в которых нарушается даже функциональная экивалентность.

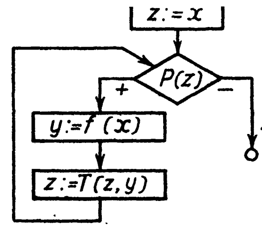
Другой подход к определению эквивалентности схем программ состоит в следующем. Для схемы не вводятся понятия выполнения и вычисляемой функции. Вместо универсального алгоритма



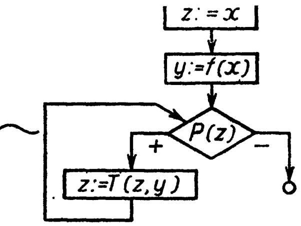
~



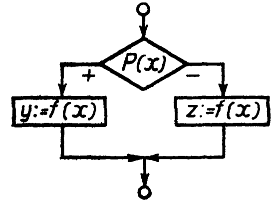
I, а



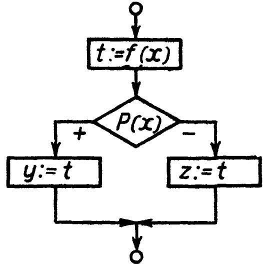
~



II, а

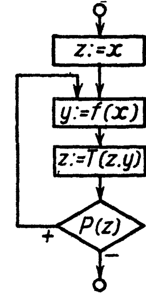


~

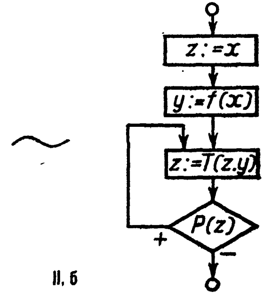


I, б

I. Упрощение ветвлений



~



II, б

II. Чистка циклов

I. Для ветвлений	II. Для циклов
$P(x): x<0$	$P(z): z \neq 0$
$f(x): sqrt(x)$	$f(x): 1/x$
$f(x): sqrt(-x)$	

„Плохие“ интерпретации

Рис. 3. Примеры преобразований, эквивалентных:
а — для некоторых;
б — для всех интерпретаций.

выполнения и следящего алгоритма вводятся «покрывающие» их порождающие процессы, которые порождают множество реализаций, заведомо содержащее в себе множество реализаций любой интерпретации данной схемы. Тот факт, что вместо алгоритма выполнения рассматривается недетерминированный порождающий процесс, в котором на каждом шагу его применения производится свободный выбор одной из нескольких альтернатив без учета предыстории развития процесса, приводит к тому, что множество реализаций, порождаемых схемой, становится более простым, т. е. допускающим его эффективное описание. То, что реализации схемы программы S заведомо включают в себя реализации соответствующей интерпретации $S(i)$, гарантирует, что некоторый факт, установленный для S, будет автоматически иметь место для $S(i)$. Это прежде всего относится к установлению факта эквивалентности схем программ.

Ниже перечисляются основные характерные черты схем программ:

— совокупность операторов, образующих схему программы, изображается в схеме явно;

— при построении реализаций преемник оператора обычно выбирается произвольно, без учета истории движения к данному оператору;

— если в рассмотрение вовлекается величина, «вырабатываемая» некоторым оператором, то она трактуется как независимая переменная, т. е. считается, что после «выполнения» данного оператора он может принимать любое значение независимо от предыдущей истории.

Необходимость двух разных подходов к определению эквивалентности схем программ вызвана некоторыми методологическими причинами: первый подход прокладывает более прямой путь к практическим приложениям, в то время как второй подход позволяет создать более «чистую» теорию, не использующую понятия интерпретации. Для некоторых классов операторных схем эти два подхода дают, по существу, одно и тоже определение эквивалентности. Однако для других классов определение эквивалентности через множество всех интерпретаций приводит снова к алгоритмически неразрешимым проблемам. В этом случае теория строится таким образом, чтобы эквивалентность по множеству историй, образуемых порождающим процессом, была бы достаточным условием эквивалентности, определяемой через интерпретации. Надо заметить при этом, что требование такой согласованности налагает некоторые дополнительные условия на выбор истории, по которой мы судим об эквивалентности программ, поскольку не для любой истории удается построить подходящий порождающий процесс, обеспечивающий разрешимость эквивалентности и дающий достаточное условие для эквивалентности, определяемой через интерпретации.

Наличие двух подходов к определению эквивалентности не случайно; более подробно о различиях между ними будет сказано в конце главы.

§ 2. Основные результаты теории программирования

Рассмотрим основные результаты по теории схем программ. Для того чтобы различать разные типы схем программ, мы будем идентифицировать их по фамилии автора работы, в которой этот класс схем был рассмотрен впервые.

Схемы Мартынюка были введены в литературу в 1961 г. Они не содержат никакой информации о программе, кроме управляющего графа. Между вершинами управляющего графа переходов может быть введено отношение тождества. Для схем Мартынюка, естественно, вводится порождающий процесс построения цепочек — путей в управляющем графе от входной вершины до выходной. Две схемы Мартынюка считаются эквивалентными, если они порождают одно и то же множество цепочек. Проблема распознавания эквивалентности здесь разрешима; это было доказано Тузовым, однако, по существу, было известно и раньше в теории автоматов, так как множество цепочек ориентированного графа с выделенными входной и выходной вершинами является регулярным событием. Для схем Мартынюка были рассмотрены отдельные виды преобразований схем, главным образом расщепление вершин. Они применялись Купером, Коком и Миллером для регуляризации структуры управляющего графа.

Для схем Мартынюка была исследована и практически полностью решена одна важная методическая проблема. Рассмотрение схемы программы становится существенно более простым, если любая цепочка в ней является допустимой, т. е. существует такая интерпретация, которая реализует эту цепочку. В то же время на практике чаще встречаются программы, в которых реализуются не все последовательности команд, предписанные управляющим графом. Поэтому возникает следующая проблема: пусть некоторыми средствами, внешними по отношению к схеме G, описаны ограничения на множество цепочек схемы. Требуется построить другую схему G^1, полное множество цепочек которой совпадает с множеством цепочек схемы G, подчиненным указанным ограничениям. Здесь возникает задача нахождения наиболее общего языка для описания ограничений на цепочки, а также методики построения схемы G^1. Первые шаги в решении этой проблемы сводимости были предприняты Смирновым, затем весьма общий результат был получен Мартынюком.

Схемы Мартынюка оказались удобной моделью для описания ряда алгоритмов построения некоторых множеств вершин схем

(так называемых транзитивных замыканий бинарных отношений между вершинами) и их декомпозиций на подсхемы. Эти алгоритмы имеют большое значение для оптимизации программ.

Схемы Янова были введены в литературу Ляпуновым и Яновым в 1956 г. Полное изложение результатов было опубликовано в 1958 г. [10]. Эта работа стала классической благодаря своей полноте и законченности: все основные компоненты теории преобразований программ в ней были явно сформулированы и в рамках построенной системы понятий почти до конца проработаны. Этими компонентами явились: формализация понятия схемы программы, задание отношения эквивалентности, нахождение алгоритма, распознающего эквивалентность схем, и, наконец, построение системы преобразований, полной в том смысле, что любую пару эквивалентных программ можно трансформировать друг в друга последовательным применением этих преобразований, сохраняющих эквивалентность. Для доказательства полноты преобразований Яновым был успешно применен аппарат теории логических исчислений, при котором правило преобразования трактуется как аксиома, постулирующая сохранение эквивалентности при выполнении преобразования.

Схема программы, по Янову, представляет собой линейную последовательность операторов $A_1, \ldots, A_n$ и распознавателей, имеющих вид произвольных логических функций от переменных $p_1, \ldots, p_k$. Операторы и условия могут быть помечены. Для схемы задается так называемое распределение сдвигов, состоящее в сопоставлении каждому оператору сдвига — некоторого подмножества переменных $p_1, \ldots, p_k$. Переменные, входящие в сдвиг некоторого оператора, интерпретируются как его результаты.

В качестве реализации схемы рассматривается последовательность выполняемых операторов при прохождении по схеме от начала до конца. Множество реализаций задается следующим порождающим процессом. Берется произвольный набор Δ значений переменных $p_1, \ldots, p_k$, и управление передается на начало схемы. Построение делается по индукции. Если с набором Δ перешли к распознавателю a, то при истинном a (Δ) управление передается по указанной в распознавателе метке, а при ложном— к следующему за a оператору или распознавателю. Если с набором Δ перешли к оператору A (T), где T — сдвиг оператора A, то оператор A выписывается в реализацию, а Δ может быть произвольно трансформирован в набор Δ^1, отличающийся от Δ не более чем значениями переменных, входящих в T. Процесс построения реализации обрывается при выходе на конец схемы или при бесконечном «зацикливании» на одних логических условиях.

Две схемы считаются эквивалентными, если совпадают их множества реализаций. Яновым был найден алгоритм распознавания эквивалентности любых двух схем и построена полная

схема преобразований, содержащая 14 аксиом и три правила вывода.

В работе [2] теория схем Янова была перенесена на язык графов. Пример схемы Янова в графовой форме приведен на рис. 1, *г*. Оказалось, что новое определение является более адекватным рассматриваемой проблеме. Это нашло свое отражение, во-первых, в упрощении аксиоматики преобразований: 14 аксиомам в [10] соответствовало только шесть аксиом в [2]. Кроме того, новый аппарат позволил эффективизировать правило вывода, использующее понятие логической подчиненности. Некоторый распознаватель a подчинен логической функции f, если проверка a при «выполнении» схемы производится только для тех значений логических переменных, на которых f истинна. Функция F_a такая, что $F_a \to f$ ($\to$ означает знак импликации) для любой f, подчиняющей a, называется полным условием работы a. Применение правила вывода логической подчиненности позволяет заменять a на af для любого f, в том числе и для F_a. Неэффективность этого правила состояла в том, что проверка логической подчиненности производилась в [10] по сложному алгоритму преобразования схемы средствами, лежащими вне аксиоматики.

Вместо этого алгоритма в [2] было введено четыре аксиомы, которые задают обратимое преобразование схемы Янова, состоящее в разметке дуг управляющего графа логическими функциями. Свободное применение этих аксиом в конце концов создает в схеме так называемую стационарную разметку, момент наступления которой для каждой вершины обнаруживается путем проверки некоторых тождественных соотношений между функциями, метящими входные и выходные дуги вершины. При стационарной разметке $F_a = f_1 \vee f_2$, где f_1 и f_2 — функции, метящие выходы распознавателя a. Таким образом соответствующее правило вывода получило принципиально более простое условие своей применимости.

Упрощение аксиоматики позволило исследовать некоторые более глубокие свойства правил преобразований. Янов исследовал, какие содержательные свойства его схем ответственны за необходимость иметь нелокальные правила преобразования. Оказалось, что этими свойствами являются допущение несущественных (невыполняемых) операторов и наличие нетривиальных распределений сдвигов. Удалось установить, что для схем без несущественных операторов и с универсальным распределением сдвигов (каждый оператор может менять все логические переменные схемы) существует полная система чисто локальных преобразований, а также найти общий критерий локальности правил преобразований схем Янова.

Работы [10] и [2] создали определенный стиль подхода к задачам теории преобразований программ, который можно проследить на ряде последующих работ.

Горель доказала логическую независимость преобразований из [2]. Иткин построил полную систему преобразований для схем Янова, допускающих отношение тождества между операторами; им же была найдена система преобразований для простого обобщения схем Янова на параллельные программы. Горель построила систему преобразований для схем Янова, эквивалентность которых рассматривается только на конечных последовательностях операторов.

Во всех этих работах при построении систем преобразований существенно использовалась методика разметки; она оказалась удобным средством для формализации преобразований, требующих для своего применения предварительного сбора некоторой информации о схеме в целом. В работе Вальковского методика разметки дуг схем Янова логическими функциями была рассмотрена с общих позиций и был описан класс информации, который может быть получен применением различных разметок.

Проблематика схем Янова еще не исчерпана. Наиболее актуальными задачами, решение которых поможет существенно расширить применимость теории, являются следующие.

а) Необходимо изменить определение схем Янова таким образом, чтобы можно было бы сравнивать схемы, содержащие разные логические переменные и в разном количестве. Кроме того, слишком обременительным является требование рассматривать любой распознаватель как функцию всех логических переменных. Другими словами, необходимо более подобно описывать информационные связи между операторами, вырабатывающими значения логических переменных, и распознавателями, их использующими.

б) Сейчас схемы считаются эквивалентными, если соответствующие последовательности операторов в сравниваемых схемах строго совпадают, возможно, с учетом отношения тождества между операторами. Очень интересно допустить существование отношения перестановочности между операторами и сравнивать последовательности операторов с точностью до перестановок перестановочных операторов.

Решение этой задачи имеет большое значение, так как в значительной степени доведет до логического завершения программу исследований, открывшуюся работой Янова, поскольку перестановки и отождествления операторов являются главными средствами преобразования программ. Тузов решил задачу распознавания эквивалентности схем Янова с учетом перестановочности операторов, но при весьма искусственных предположениях. Летичевский также решил задачу распознавания эквивалентности с учетом перестановочности для частного случая схем в рамках его исследований по полугрупповой эквивалентности (см. ниже).

Схемы Лаврова были введены в литературу в 1961 г. [7] в связи с задачей экономии памяти и сейчас являются наиболее

употребляемой моделью для решения прикладных задач теории программирования — главным образом для алгоритмов оптимизации программ. В схемах Лаврова исследуются информационные связи между операторами в условиях неизменности логической структуры программ. Формально они определяются следующим образом: берется схема Мартынюка, в которой каждому оператору приписывается некоторое неотрицательное число информационных входов и выходов (можно считать, что оператор имеет не более одного выхода). Входам и выходам произвольным образом сопоставляются символы переменных величин, или ячеек памяти.

Как ни странно, общая теория схем Лаврова (в смысле нахождения наиболее общего определения распознаваемой эквивалентности, построения канонической формы и полной системы преобразований) до сих пор еще не разработана, все конкретные работы базируются на некоторых допущениях или ограничениях, играющих роль достаточных условий, обеспечивающих нужную автору эквивалентность. В наиболее общей форме схемы Лаврова, по-видимому, рассматривались у Котова в его работах по параллельному программированию. Фактически роль истории в схемах Лаврова играет информационно-логический граф. Есть и другой, в некотором смысле двойственный подход для определения инвариантов схем Лаврова. Информационный граф представляет собой совокупность всех информационных связей, реализуемых некоторой конкретной операционно-логической историей. В ряде случаев можно, наоборот, рассматривать для заданной информационной связи между операторами A и B множество всех путей в схеме от A до B, реализующих эту связь. Такие пути называются маршрутами информационной связи. Множество всех маршрутов всех информационных связей схемы Лаврова является, однако, более узким инвариантом, чем совокупность информационно-логических графов.

Перечислим основные результаты, полученные для схем Лаврова. Лавровым [7] и Ершовым была решена задача минимизации числа переменных в схеме Лаврова. Эти вопросы в более общей постановке были также рассмотрены Никитиным. Мартынюк в рамках эквивалентности по информационно-логическому графу и допуская отношения тождества между операторами нашел достаточное и необходимое условие допустимости перемещения оператора из одного места схемы в другое. Поттосин рассмотрел эту задачу применительно к «чистке циклов» в трансляторах. В группе, концентрирующейся вокруг д-ра Дж. Кока (США), выполнена целая серия работ по исследованию информационных связей в схемах Лаврова в интересах задач оптимизации (экономия совпадающих выражений, оптимизация ветвлений, чистка циклов и т. п.). Котов рассмотрел серию преобразований схем Лаврова (переименование переменных, расщепление и перемещение операторов), позволяющих из данной схемы получить

другую, обладающую большей внутренней параллельностью.

Ершов рассмотрел вариант схем Лаврова, в которых информационные связи между входами и выходами операторов задаются не с помощью имен переменных, а явно — в виде дополнительного информационного графа схемы (следует различать информационный граф схемы и информационный граф реализации). Это позволяет построить теорию, не зависящую от разных вариантов распределения памяти. Полезность такого подхода обнаружилась в задачах оптимизации программ и в параллельном программировании. Оказалось, что не всякий информационный граф, наложенный на управляющий граф схемы, может быть реализован в виде схемы Лаврова. Иткин нашел необходимое и достаточное условие для реализуемости заданного информационного графа схемой Лаврова.

Полугрупповая эквивалентность. Вернемся к обсуждению различных видов историй реализаций программы. Заметим, что все истории ведут свое начало от операционной истории — последовательности выполненных операторов, которая в случае схем программ выглядит как некоторое слово

$$S_{i_1}S_{i_2}\ldots S_{i_1}$$

в алфавите $S = \{S_1, \ldots, S_n\}$, где $S_1, \ldots, S_n$ — символы операторов. Отличие менее детальных историй от операционной истории можно рассматривать как учет некоторых соотношений между операторами схемы. Этот учет состоит в том, что вместо требования буквального совпадения операционных историй мы разрешаем сравниваемым историям отличаться, но в пределах априорно заданных соотношений между операторами. Рассмотрим ряд примеров:

а) Две операционные истории могут совпадать с точностью до несовпадения операторов, между которыми введены отношения тождества

$$S_i \equiv S_j.$$

б) Две операционные истории имеют общий информационный граф. Это можно выразить тем, что между некоторыми парами операторов имеются отношения коммутативности вида

$$S_iS_j \equiv S_jS_i,$$

которые используются так, что операторы, соединенные путем в информационном графе, являются некоммутирующими, а не соединенные — коммутирующими.

в) Если сложный оператор, например $y := x + z\uparrow 2$, сводится к выполнению серии более простых, в данном случае $t := z\uparrow 2$; $y := x + t$, то это может быть учтено допущением определяющих соотношений вида

$$S_i \equiv S_kS_l.$$

Таким образом, каждый вид эквивалентности можно описать, задав над множеством слов W в алфавите S символов операторов некоторое количество G определяющих отношений вида

$$\left.\begin{array}{l} w_1 \equiv w_1' \\ w_2 \equiv w_2' \\ \cdot \ \cdot \ \cdot \\ w_n \equiv w_n' \end{array}\right\} G,$$

где w и w' — конкретные слова из W. Будем называть два слова из W равными над G, если они могут быть преобразованы одно в другое последовательными заменами входящих в них подслов в соответствии с соотношениями G. Тогда множество W, разбитое на множество классов равных друг другу слов, называется полугруппой $P(G, S)$ над алфавитом S, заданной определяющими соотношениями G.

Вводимая таким образом эквивалентность называется полугрупповой эквивалентностью над полугруппой $P(G, S)$; две схемы A_1 и A_2 эквивалентны в этом смысле, если для каждой операционной истории h_1 схемы A_1 схема A_2 содержит историю h_2, равную h_1 над G; или две схемы A_1 и A_2 эквивалентны над полугруппой $P(G, S)$, если для любой интерпретации i операционная история программы $A_1(i)$ равна над G операционной истории программы $A_2(i)$. Различие между определениями состоит в том, привлекается или нет интерпретация к рассмотрению эквивалентности. На рис. 4 показан пример двух схем, для которых выписаны определяющие соотношения для полугрупповой эквивалентности, соответствующей истории в виде термального значения.

В связи с таким единым подходом к определению эквивалентности можно поставить общую проблему: какой должна быть полугруппа $P(G, S)$, чтобы соответствующая полугрупповая эквивалентность была бы разрешимой.

Полугрупповая эквивалентность схем была введена Летичевским; им же получены сильные и весьма общие результаты в этом направлении. Читателям, знакомым с теорией полугрупп, без сомнения, представит интерес формулировка наиболее общего результата, полученного Летичевским: для разрешимости полугрупповой эквивалентности достаточно, чтобы в полугруппе

а) не было бы разложимой единицы;

б) было бы справедливо правило левого сокращения.

Эти интересные результаты явились результатом обширной программы исследований, инициированной академиком В. М. Глушковым и связанной с приложением методов алгебраической теории автоматов к задачам теории программирования.

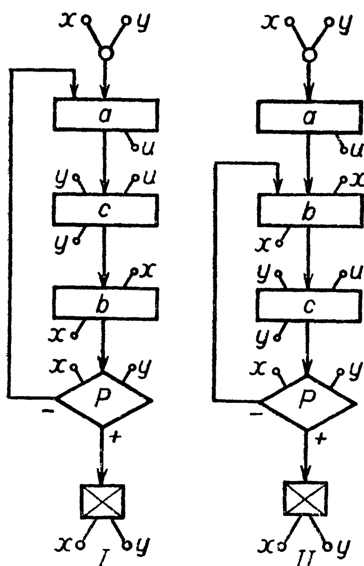
I: a c b a c b a c b
II: a b c b c b c

Операционные истории

x: $b(b(b(x)))$

y: $c(c(c(y,a),a),a)$

Термальные значения



Эквивалентные схемы

1. $aa = a$
2. $ab = ba$
3. $cb = bc$
4. $ac = aca$

} G

Полугруппа

$a\,\underbrace{c\,b}_{3}\,a\,c\,b\,a\,c\,b =$

$a\,b\,c\,a\,c\,\underbrace{b\,a}_{2}\,c\,b =$

$a\,b\,c\,\underbrace{a\,c\,a}_{4}\,b\,c\,b =$

$\underbrace{a\,b}_{2}\,c\,a\,c\,b\,c\,b =$

$b\,\underbrace{a\,c\,a}_{4}\,c\,b\,c\,b =$

$\underbrace{b\,a}_{2}\,c\,c\,b\,c\,b =$

$a\,b\,c\,\underbrace{c\,b}_{3}\,c\,b =$

$a\,b\,c\,b\,c\,\underbrace{c\,b}_{3} =$

$a\,b\,c\,b\,c\,b\,c$

Доказательство равенства операционных историй I и II над G

Рис. 4. Пример полугрупповой эквивалентности.

Следует, однако, отметить, что непосредственное использование полученных результатов еще затруднительно, так как техника полугрупповой эквивалентности еще плохо справляется с введением в историю зависимостей от логической структуры схемы. Некоторый прогресс, но еще не вполне достаточный, был

в этом отношении осуществлен Непомнящим в отношении схем Янова. Рамки брошюры не позволяют останавливаться подробнее на этом важном вопросе.

§ 3. Проблематика

Фундаментальная проблема теории программирования. Такой проблемой автор считает построение полной теории схем программ, в которой был бы объединен анализ управляющих связей, сделанный в схемах Янова, с анализом информационных связей, сделанным в схемах Лаврова. Под полной теорией понимается следующее:

1) нахождение алгоритма распознавания эквивалентных схем программ;

2) построение полной системы эквивалентных преобразований;

3) нахождение канонической формы схем программ;

4) построение рабочей системы преобразований, позволяющих решать задачи оптимизации.

Опишем более подробно класс схем программ, для которых имеет смысл эта постановка вопроса. Схемы из этого класса будут называться стандартными.

Имеются формальные символы и выражения:
переменные (x, y, z и т. д.), операции определенной «местности» (f, $f(\ )$, $f(\ ,\ )$ и т. д.), термы (выражения, которые строятся из переменных и операций по обычным правилам), предикатные символы некоторой «местности» (**истина**, **ложь**—тождественные предикаты; $P(\ )$, $P(\ ,\ )$ и т. д.), предикатные термы (предикатные символы, аргументы которых заполнены некоторыми термами),а также произвольные логические функции этих предикатных символов. В качестве преобразователей берутся операторы присваивания, имеющие вид $y := T$, где y — переменная, а T — терм. Из этих операторов присваивания выделяются операторы пересылки, имеющие вид $y := x$, где x — переменная. В качестве распознавателей берутся предикатные термы.

Рассмотрим теперь определение эквивалентности таких схем программ.

Ситуация такова, что сейчас еще невозможно дать точное определение эквивалентности, в рамках которого следует искать решение фундаментальной проблемы для стандартных схем. Соответствующая история должна быть существенно детальнее, чем термальные значения результатов, но может быть, по-видимому, несколько слабее логико-термальной истории. Поиски ослабления необходимы хотя бы для того, чтобы приспособить определение логико-термальной истории для параллельных программ, когда каждый результат должен иметь свою серию предикатов, от которых он зависит, а сами предикаты должны об-

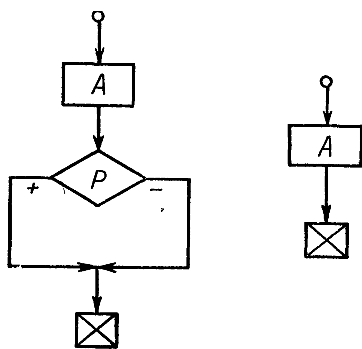


Рис. 5. Схемы, не эквивалентные по отношению к ЛТ-истории.

разовывать не цепочку, а, по-видимому, дерево зависимостей. Кроме того, нужно научиться не учитывать в логико-термальной истории зависимости от предикатов, которые, присутствуя в цепочке операторов, фактически не влияют на вычисление результата. Тривиальный пример такого рода показан на рис. 5.

Рассмотрим теперь, что сделано в направлении решения фундаментальной проблемы.

Не все знают, что первая попытка атаковать эту проблему была предпринята Криницким в 1959 г., почти вслед за появлением работы Янова. Она была известна лишь узкому кругу лиц по диссертации. Отдельные фрагменты исследования появились через четыре года в [6] и лишь недавно вышло полное изложение этой работы [5]. Работа выглядит сейчас несколько архаично, однако там эта задача была полностью решена для случая схем программ, не содержащих циклов. Это, конечно, очень частный случай, что, однако, никак не умаляет достоинств работы.

В работе была фактически рассмотрена функциональная эквивалентность, т. е. две схемы считаются эквивалентными, если соответствующие функции в любой интерпретации одинаковы. В рамках этой эквивалентности был найден алгоритм ее распознавания, построена каноническая форма и найдена полная система эквивалентных преобразований.

Через несколько лет эти результаты практически в том же объеме были независимо получены Игараши и описаны тоже сначала в диссертации, относящейся к 1964 г, а затем модернизированы и опубликованы.

С другой стороны, фундаментальная проблема подпирается серией отрицательных результатов, устанавливающих неразрешимость распознавания эквивалентности для ряда историй. Патерсоном и чуть позднее, но независимо Летичевским была доказана неразрешимость распознавания функциональной эквивалентности. Полные изложения этого важного результата появились в обратной последовательности в 1969 и 1970 гг. (Летичевский, Лукам, Парк и Патерсон). Летичевский, кроме того, показал, что эквивалентность по термальным значениям равносильна функциональной эквивалентности. В этих же работах были описаны некоторые частные виды схем, для которых функциональная эквивалентность разрешима. Из них наибольший интерес представляет случай схем, в которых все информацион-

ные связи могут быть реализованы через одну ячейку памяти.

Результат Патерсона и Летичевского был существенно усилен Иткиным и Звиногродским, которые показали, что практически любое разумное определение эквивалентности, использующее понятие множества всех интерпретаций и применимое ко всем схемам программ, приводит к алгоритмической неразрешимости. «Разумность» определения эквивалентности выражается следующими свойствами:

1) отношение эквивалентности схем программ называется существенно интерпретационным, если из того, что S_1 не эквивалентна S_2, следует, что существует такая интерпретация i, в которой при некоторых исходных данных логико-операционная история реализации программы $S_1(i)$ не совпадает с соответствующей логико-операционной историей программы $S_2(i)$;

2) отношение эквивалентности является невырожденным, если для неэквивалентности двух схем S_1 и S_2 достаточно существования такого оператора, который был бы существен при некоторой интерпретации i для программы, скажем, $S_1(i)$ и не существен для $S_2(i)$.

Теорема Иткина и Звиногродского гласит, что если отношение эквивалентности, рассматриваемое для всех схем программы, является существенно интерпретационным и невырожденным, то распознавание эквивалентности алгоритмически неразрешимо. Эту теорему можно рассматривать как формализацию и доказательство тезиса Патерсона, высказанного им в 1968 г. «Поистине, мы можем одним ударом доказать неразрешимость любого отношения, лежащего между общей и слабой эквивалентностями».

Главный вывод из этого результата состоит в том, что любая эквивалентность, которая базируется на множествах таких путей в схеме программы, которые подтверждаются хотя бы одной интерпретацией, неизбежно будет неразрешима.

Таким образом, построение содержательной теории эквивалентности возможно только с помощью рассмотрения множества историй, порождаемого некоторым недетерминированным процессом. В этом отношении имеет большое значение совсем свежий результат, полученный Иткиным. Им рассмотрены стандартные схемы программ, но допускающие в позиции распознавателя только предикатный терм, а не их логическую функцию. Определение эквивалентности вводится как требование совпадения множества логико-термальных историй, строящихся по каждой логико-операционной истории, которые, в свою очередь, создаются порождающим процессом, аналогичным тому же в схемах Янова или Лаврова. Для этого определения эквивалентности им доказана ее распознаваемость для любой пары схем. Этот важный результат позволяет выразить некоторый оптимизм в отношении решения фундаментальной проблемы в целом.

Алгебра программирования. Рассмотрим, что может дать программированию решение фундаментальной задачи.

Во-первых, ее решение обеспечит интеграцию большего числа существующих, но разрозненных методов реализации языков программирования и трансляторов.

Система преобразований, сохраняющая логико-термальную историю, сможет стать единым математическим базисом для таких алгоритмов трансляции, как:

— декомпозиция выражений и условных операторов;

— оптимизация использования выделенных регистров и сверхбыстродействующей памяти;

— экономия совпадающих участков вычислений;

— статическая экономия памяти;

— оптимизация процедур;

— все виды оптимизации, связанные с перемещением операторов в программе;

— многие виды оптимизации циклов.

Такая интеграция методики и строгое обоснование алгоритмов трансляции особенно необходимы для реализации универсальных программирующих процессоров, управляемых описанием языка. Это вызвано необходимостью иметь некоторую единую модель программы, для того чтобы обеспечить во время трансляции сохранение семантической правильности любой программы на любом внешнем языке. Это вызвано также тем, что трансляторы, управляемые описанием языка, смогут конкурировать с трансляторами, специально скроенными под данный язык, только в том случае, если будут содержать мощные оптимизирующие алгоритмы, компенсирующие внутреннюю неэффективность декомпозированной входной программы.

Во-вторых, решение фундаментальной проблемы является, по нашему мнению, единственным путем построения конструктивной теории параллельного программирования, о чем более подробно будет сказано в следующей главе.

Последовательные программы, написанные в существующих алгоритмических языках, несут на себе отпечаток экономии мышления человека и поэтому очень часто могут выполняться в точности так, как они написаны. Для параллельных программ это в принципе не так. Дело не только в том, что интуиция и алгоритмический стиль мышления человека подводят его, когда он сталкивается с большим количеством параллельных и взаимодействующих процессов. Получение параллельной программы для нескольких десятков или даже сотен обрабатывающих элементов большой вычислительной системы является априори сложной вычислительной процедурой, которая должна выполняться динамически и с большой скоростью. Создать алгоритмы генерирования параллельной программы и планирования работы многих процессоров можно только на основе точной формальной теории схем программ. Вопросы теории параллельного прог-

раммирования рассматриваются в следующей главе, однако здесь следует заметить, что две наиболее важные, по мнению автора, работы по параллельному программированию (Карп и Миллер, Котов и Нариньяни) целиком базируются на некотором фрагменте теории схем программ.

Наконец, значение решения фундаментальной проблемы не ограничивается возможностью построить новые или обосновать существующие механические процедуры трансляции. Успех этой проблемы позволит нам приблизиться к построению АЛГЕБРЫ ПРОГРАММИРОВАНИЯ — к универсальному символизму, позволяющему манипулировать в математике с процессами и алгоритмами так же, как мы сейчас с помощью символизма математического анализа манипулируем с функциями и другими традиционными объектами математики. Автор, естественно, не первый, кто привлекает внимание к этой широкой научной проблеме; достаточно сослаться на известные работы Маккарти и Глушкова. Хотелось, однако, подчеркнуть, что только решение фундаментальной проблемы даст первый адекватный материал для алгебры программирования.

Внутренний язык. Мост обычно строят с двух сторон. Это полностью относится и к связи теории и технологии программирования. Схемы программ являются абстрактной моделью реальных конструкций, применяемых в программировании. Довольно давно автор исследовал вопрос о связи схем программ с алгоритмическими языками и системами команд ЭВМ. Без сомнения, многие другие также исследовали этот вопрос. В трансляторе системы АЛЬФА существовал специальный просмотр, на котором по программе строилась схема Лаврова для анализа информационных связей в интересах глобальной экономии памяти. Опыт показывает, что непосредственное применение алгоритмов анализа или преобразований схем программ довольно затруднительно. Необходимо вводить в технологию написания трансляторов специальные формы представления программы, которые были бы адекватны теоретическим конструкциям.

Автору кажется, что совокупность проблем, связанная с обоснованием методов трансляции и использованием в них результатов теории программирования, фокусируется в проблеме создания внутреннего, или промежуточного, алгоритмического языка. Речь идет о конструировании еще одного универсального алгоритмического языка, который, однако, не предназначался бы для составления программ человеком, а использовался главным образом для внутреннего представления программ на некотором промежуточном уровне.

Надо сказать, что концепция внутреннего, или, лучше сказать, промежуточного, языка, является сегодня для программирования рабочей. Имеется разнообразие систем программирования и проектов, в которых в явном виде присутствует

эта концепция. Внутренний язык в таких системах служит одной или нескольким целям из следующего списка:

— логический этап в многофазной схеме трансляции;

— семантический язык для трансляторов, управляемых описанием входного языка;

— средство записи декомпозированной программы (трехадресный символический код);

— промежуточный язык для многоязыковых систем программирования;

— промежуточный язык для систем программирования с многообразием выходных машин;

— уровень анализа и оптимизации транслируемой программы.

Внутренние языки, однако, в таких системах еще слишком специальны и носят на себе следы одностороннего или даже случайного подхода. Представляет большой интерес выделение внутреннего языка на уровень, независимый от какого-либо частного проекта, и придание ему черт, позволяющих использовать его для большинства перечисленных выше применений. Некоторые соображения по поводу такого универсального подхода будут обсуждаться в третьей главе брошюры.

В заключение следует отметить, что теория схем программ далеко не исчерпывает тематику теоретического программирования, хотя безусловно занимает в ней сейчас центральное место, являясь точкой роста. Ранее разрозненные группы ученых, упражнявшихся в формализации отдельных понятий программирования, сейчас вступают в период интенсивного научного соревнования, атакуя принципиальные вопросы теории. Это приведет к тому, что теория схем программ станет через пять — десять лет стандартным университетским курсом, и новое поколение программистов, вооруженное этими и другими знаниями, преодолеет нынешние методологические трудности программирования.

ГЛАВА 2

ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ

Автор глубоко убежден, что проблемы программирования являются решающими в общей проблеме создания и использования параллельных вычислительных систем. Действительно, воображение инженера может довольно быстро методом аналогии или, наоборот, методом выбора противоположного подхода предложить и зримо воплотить те или иные конструкции вычислительных систем. Этот подход, однако, вынужденно приводит к такому положению, когда проблемы программирования приобретают вторичный характер и степени свободы при их решении автоматически ограничены предписанной структурой, которая далеко не всегда может отвечать сути дела.

Проблема программирования, по существу, является вторичной, когда речь идет о частной задаче, которую необходимо решить на данной машине. Однако проблема программирования является первичной при вдумчивом подходе к созданию вычислительной системы в том смысле, что разработчик должен силой своего проникновения в суть дела охватить мысленно все множество возможных программ и алгоритмов и на основе анализа их внутренних свойств, структуры и внешних характеристик сформулировать такие принципы построения вычислительной системы, которые были бы адекватны рассмотренным свойствам программ, при которых, что особенно важно, задача программирования могла бы быть сведена к формальной процедуре и тем самым автоматизирована.

Автоматизация программирования для параллельных вычислительных систем является принципиально необходимой в силу ряда причин.

Во-первых, написание параллельной программы не свойственно природе алгоритмического мышления человека. Правда, этот тезис верен лишь постольку, поскольку мы говорим о ре-

ализации на вычислительной системе задачи, сформулированной в алгоритмической форме. Сейчас это наиболее реальная постановка проблемы, однако в дальнейшем положение может измениться, так как на самом деле параллелизм в задаче зачастую скрыт в ее исходной постановке, а при ее алгоритмизации искусственно изгоняется. Ниже об этом будет сказано более подробно.

Во-вторых, независимо от того, мыслит человек «последовательно» или «параллельно», он склонен мыслить лаконично и стремиться к компактной формулировке задачи. В то же время исследования по построению параллельных программ показывают, что во многих случаях совершенного параллелизма можно добиться только при потенциально неограниченном «размножении» некоторых исходных конструкций программы, сопровождаемом их систематической модификацией. Таким образом, некоторая часть процесса программирования становится органической частью процесса вычисления и поэтому в принципе должна быть передана машине.

В этой главе[1] будет сделана попытка осуществить такой примат программирования над конструкцией вычислительной системы. Будут рассмотрены сложившиеся к настоящему времени концепции параллельного программирования и на их основе дана оценка тех или иных подходов к структуре вычислительных систем. В силу общей неразработанности вопроса этот анализ будет сделан поверхностно и местами даже спекулятивно. С другой стороны, будет сделана попытка высказать некоторые рекомендации с позиций реальности, учитывающие современные технические возможности, исторически сложившиеся рамки и другие ограничения текущего момента.

§ 1. Основные концепции параллельного программирования

В любой теории параллельного программирования, претендующей на полноту, должны быть представлены:

— способ описания параллельной программы (средства параллельного программирования);

— способ получения параллельной программы из обычной алгоритмической записи (десеквенция последовательных алгоритмов);

— способ выполнения параллельной программы (определение вычислительной системы, назначение процессоров на работы, управление параллельным процессом).

Рекомендуемая структура вычислительной системы должна быть производной от основных принципов, отражающих на-

[1] Материалом для второй главы послужили доклады автора на Второй Всесоюзной конференции по однородным вычислительным системам и средам (Москва, ноябрь 1969 г.) и в ряде американских университетов (ноябрь, 1970 г.)

иболее существенные стороны каждой из этих компонент. Можно рассмотреть три подхода к параллельному программированию, которые условно будут названы следующим образом:

— природный параллелизм;
— совместные вычисления в алгоритмических языках;
— асинхронное программирование.

Природный параллелизм далек от того, чтобы его можно было бы считать сколько-нибудь развернутой теорией. Однако лежащие в его основе соображения носят достаточно четкий и во многом весьма привлекательный и своеобразный характер, что дает право выделять его в особый подход. Эта теория не имеет индивидуального носителя, хотя и неявно используется во многих работах. Поэтому конкретно указать на ее источники трудно.

Природный параллелизм. Основной тезис этой теории состоит в том, что в природе на самом деле все происходит параллельно, и чем буквальнее описан некоторый реальный процесс, тем ближе это описание к параллельной программе. Общая проблема состоит в том, что нужно уметь удачно и просто улавливать этот природный параллелизм и описывать его в некоторой универсальной системе понятий, базирующихся на таких принципах, как одновременность многих событий, близкодействие и т. п., т. е. представляющих явление как совокупность многих элементарных процессов, которые осуществляются в пространстве одновременно, с взаимодействием только соседствующих процессов. В основе такого подхода лежит принцип детерминизма, гласящий в данном случае, что если явление в какой-то момент описано как совокупность начальных состояний и механизмов взаимодействия, то вся последующая история предопределена и может быть наблюдаема, измеряема и вычисляема. Пространство, о котором идет речь, не обязательно должно быть физическим и континуальным, оно может быть дискретным, конечным и т. д.; важно только, чтобы в нем было определено понятие непосредственного соседства.

Этот принцип достаточно хорошо известен и фактически применяем во многих случаях (на нем, в частности, основаны все методы описания явлений в дифференциальной форме). Истинная и уже далеко не тривиальная проблема состоит в том, чтобы конструктивно сформулировать этот принцип в универсальной форме, представляющей все сущности, которые должны быть воплощены в параллельных программах, в виде своеобразного вычислительного универсума, в которой может быть вложена информационная модель любого интересующего нас явления.

При чистой постановке вопроса проблема перевода «обычной» записи задачи в параллельную тоже не возникает, так как природный параллелизм является первичным и понятие «обычной» записи теряет смысл. Однако чистая постановка вопроса здесь невозможна, так как вся вычислительная сторона, связанная с построением арифметической модели явления, несет на себе

отпечаток сугубо последовательного процесса логического рассуждения, лежащего в основе любого алгоритма. Поэтому любая систематическая попытка проложить прямой мост от природного параллелизма к его воспроизведению в вычислительной системе потребует существенной ревизии всего численного анализа и аппарата математической физики. Другой дополнительной и не менее грандиозной проблемой является построение такой системы понятий, при которой любое взаимодействие, происходящее в природе, описывается как процесс обмена информацией и в которой взаимодействующий элемент пространства представляет собой миниатюрный универсальный вычислитель-многополюсник, производящий необходимую переработку информации.

Если рассмотреть третью компоненту природного параллелизма, то сразу нужно сделать гипотезу о подходящей вычислительной системе. Совершенно очевидно, что в качестве первого кандидата такой системы напрашивается вычислительная среда. Отложив рассуждение собственно о среде на конец главы, следует отметить, что при достаточном запасе вычислительной мощности, позволяющем сопоставить каждому элементу пространства отдельную ячейку среды, все управление и реализация вычислительного процесса состоят в настройке и запуске среды. Среда воспроизводит описываемое явление почти «один к одному» и тем самым избавляет нас от забот по организации вычислительного процесса.

Задача, однако, существенно усложняется, если параллельный процесс содержит хотя бы на один элемент больше, чем доступное число ячеек среды. Возникают очень сложные задачи учета краевых эффектов и проблемы перенастройки среды во время воспроизведения исследуемого процесса.

Заключая рассмотрение природного параллелизма, можно заметить, что, начав с утверждения о привлекательности этого подхода и обращая внимание на то, что некоторые проблемы исчезают при таком подходе, все последующие рассуждения были направлены на дискредитацию этого подхода. Дискредитация состоит в том, что развиваемый в качестве универсального этот подход является слишком революционным, ломающим многие привычные представления, выводящим некоторые проблемы на уровень фантастики, а посему не могущим быть рекомендуемым в качестве руководства к немедленному действию.

В то же время автор надеется, что эта дискредитация не приведет к тому, что поиски универсальной схемы для введения в программирование природного параллелизма ослабнут. Это не должно произойти, в частности, потому, что успешный синтез природного параллелизма и вычислительной среды в эффективно действующих устройствах должен оказаться важным вкладом в исследование структуры и механизма работы мозга — наиболее совершенной вычислительной среды, созданной природой.

Совместные вычисления в алгоритмических языках в противоположность только что рассмотренному подходу спускают нас с небес на землю. Этот подход является некоторой минимальной мерой внедрения параллелизма в алгоритмические языки, полностью учитывающим современную архитектуру машин и организацию операционных систем. Многие вопросы параллельного программирования этот подход оставляет в стороне, но зато позволяет хотя бы отчасти использовать те возможности, которые представляются сегодняшней техникой.

Изобразительные средства для указания параллелизма, о которых идет речь в этом разделе, имеются в таких языках, как языки моделирования Симскрипт [8], Симула [1] и алгоритмические языки «нового поколения» — ПЛ/1 и Алгол 68. Их общая сущность состоит в том, что в алгоритме явно выписываются параллельные, или совместные, ветви вычислений. Ветви, параллельные друг другу, обычно имеют общее начало и конец. Каждая ветвь, в свою очередь, может содержать подветви. Таким образом, весь параллелизм программы выписан явно, точки ветвления и схода могут использовать общие переменные и содержать средства прерывания или приостановки ветвей в зависимости от значения переменной величины. Тем самым осуществляется синхронизация совместных вычислений, явное программирование которой является задачей программиста.

Никакого способа выделения параллельных ветвей из обычной записи рассматриваемый подход пока не предлагает, так как использование изобразительных средств параллельности не формализовано и оставлено полностью за человеком.

В большинстве случаев введение параллельных ветвей разрешает, но не предписывает им выполняться фактически параллельно. При этом программа позволяет в случае отклонения от полного параллелизма перебирать ветви в произвольном порядке, что облегчает работу операционной системе. Правда, опять-таки в существующих языках формальных правил для определения «хорошо написанной» программы не дается.

Организация вычислений по программе с параллельными ветвями осуществляется по принципу многопрограммной работы, когда программа решения большой задачи в «глазах» диспетчера операционной системы представляет собой поток взаимосвязанных мелких работ. Одна параллельная ветвь представляет собой одну работу. На эту работу составляется паспорт, хранящийся в операционной системе. Когда контроль доходит до точки ветвления, выполняется поставленное туда транслятором обращение к супервизору, которое активизирует паспорта ветвей, начинающихся в этой точке, и делает их претендентами на счет. Супервизор осуществляет некоторое планирование назначения ветвей на работу, стремясь загрузить устройства (или процессоры), могущие работать вместе, и стараясь как можно быстрее добиться завершения всех параллельных ветвей, так

как только совместный выход на точку схода позволяет продвинуться дальше в решении задачи.

Такой метод практичен лишь тогда, когда прохождение точек ветвления и схода происходит относительно редко. Это сразу налагает серьезные ограничения на степень параллелизма и на число допустимых ветвей. Другое ограничение состоит в том, что параллельная структура программы статична и фактически не допускает, например, развертывания циклов.

Асинхронное программирование, с точки зрения автора, в наибольшей степени имеет право называться теорией параллельного программирования и именно поэтому будет рассмотрено более подробно. Основы этой теории являются первым итогом шестилетней работы молодых математиков Котова и Нарииньяни. К этому же направлению примыкает работа американских математиков Карпа и Миллера. Асинхронное программирование с самого начала развивается как математическая теория, строящаяся на некоторой аксиоматике. В этом его достоинство и недостаток. По принципам асинхронного программирования не было еще создано ни вычислительной системы, ни транслятора. Зато теория асинхронного программирования позволяет:

— дать точное понятие параллельной (асинхронной) программы;

— описать класс вычислительных систем;

— строго определить эквивалентность программ;

— сформулировать признаки правильности параллельной программы для данной системы;

— дать определение степени параллелизма и максимального параллелизма;

— конструктивно доказать возможность формального перевода обычной программы в параллельную программу, обладающую максимальным параллелизмом;

— исследовать некоторые внутренние проблемы параллельного программирования.

Дополнительной особенностью асинхронного программирования является то, что оно в своей теоретической части смыкается с теорией схем программ. Эта смычка еще не осуществлена полностью, но перспективы ее очевидны.

Рассмотрим основные понятия асинхронного программирования. Исходным является понятие квазипрограммы, представляющей собой произвольный набор операторов, работающих над памятью, состоящей из переменных величин. Оператор представляет собой многополюсник, который задает значения выходов оператора как функцию его входов. Входами и выходами оператора сопоставлены переменные величины из памяти, поставляющие аргументы оператора на входы и принимающие результаты оператора из выходов. Каждый оператор может быть в одном из трех состояний: спящем, бодрствующем и работающем.

Выполнение квазипрограммы некоторой вычислительной си-

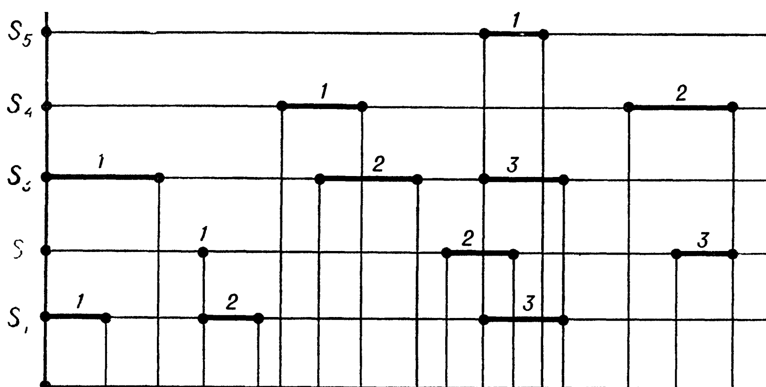


Рис. 6. Вычислительный процесс.

стемой состоит в следующем. В начальный момент задается исходное состояние памяти. Все операторы спящие. Работа системы состоит в последовательности изменений состояний вычислений в отдельные дискретные моменты времени. В каждый такой момент вычислительная система может производить перевод операторов из одного состояния в другое, причем работающим стать может только бодрствующий оператор.

Если оператор становится работающим, он в этот момент воспринимает из памяти значения аргументов. Если оператор перестает быть работающим, он в этот момент передает в память вычисленные значения результатов.

Ход вычислений может быть изображен графически в виде вычислительного процесса, представляющего собой временну́ю диаграмму включения и выключения операторов, понимая под включением переход в работающее состояние и под выключением — выход из рабочего состояния. Отрезок на временно́й диаграмме, соединяющий моменты включения и выключения, изображает одно из срабатываний оператора (рис. 6).

По вычислительному процессу можно однозначно построить его важную характеристику — информационно-логический граф. Это ориентированный граф без циклов, вершинами которого являются срабатывания операторов. От вершины A ведет стрелка к вершине B, если при срабатывании B какой-нибудь его аргумент принял информацию, заданную каким-нибудь результатом срабатывания A, или же если срабатывание A непосредственным образом влияет на срабатывание B. Понятие непосредственного влияния одного оператора на другой определяется довольно сложно, но некоторое представление о его природе дает пример, изображенный на рис. 7. Здесь оператор S_1 непосредственно влияет на оператор S_2, так как именно от S_1 непосредственно

зависит, выполнится S_2 или нет, в то время как S_1 уже не влияет на S_2. Оператор S_0 влияет, но косвенно, на S_3 и непосредственно на S_1 и S_3.

Информационно-логический граф, с одной стороны, сохраняет некоторый необходимый минимум сведений, позволяющий восстановить способ переработки входных данных вычислительного процесса и конечные результаты. С другой стороны, он игнорирует менее существенные подробности хода вычислительного процесса. Заметим, что не случайно информационно-логический граф для квазипрограмм определяется так же, как и для последовательных программ. Являясь их общим инвариантом, он позволяет объединить одной теорией последовательные и параллельные программы.

Принципиальной особенностью асинхронного программирования является допущение неоднозначности работы вычислительной системы при выполнении квазипрограммы. Предполагается, что, вообще говоря, система имеет некоторые не фиксируемые квазипрограммой параметры, или степени свободы. Это приводит к тому, что при заданной квазипрограмме для одного и того же начального состояния памяти система может осуществить некоторое множество, в том числе и бесконечное, вычислительных процессов.

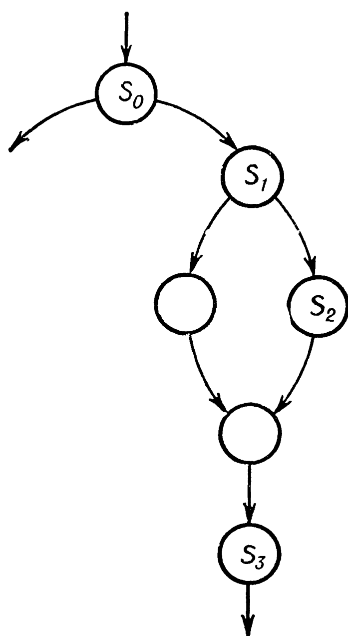


Рис. 7. Примеры зависимости операторов.

Квазипрограмма называется программой по отношению к данной вычислительной системе, если для любого заданного начального состояния памяти все вычислительные процессы, осуществляемые для него системой, имеют один и тот же информационно-логический граф. Таким образом, информационно-логический граф является инвариантом, обеспечивающим правильность преобразования информации при любом поведении вычислительной системы, выполняющей данную программу.

Более подробно степени свободы вычислительной системы состоят в следующем. Предполагается, что каждый оператор программы снабжен предикатом, имеющим некоторые входы из памяти и называемым спусковой функцией. Оператор и его спусковая функция называются блоком. При выполнении про-

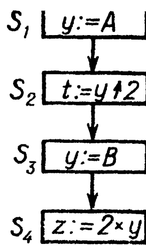
граммы система непрерывно вычисляет значения всех спусковых функций блоков программы. Блоки, спусковые функции которых равны единице, считаются бодрствующими, а остальные — спящими. В любой момент времени система может включить любой из бодрствующих блоков или выключить любой из работающих. Вычислительные системы такого рода называются асинхронными, или *А*-системами, а программы, выполняемые на них,— асинхронными программами.

Мы видим, что асинхронные программы обладают очень важным качеством: защищенностью от произвола, которым может обладать система в отношении моментов включения бодрствующих операторов, времени их выполнения и объема вычислительных средств, которым можно передать бодрствующие операторы для работы. Таким образом, асинхронная программа не предъявляет системе никаких специфических требований к ее временным характеристикам или вычислительным мощностям (числу процессоров). Кроме того, характеристики системы могут меняться динамически, не нарушая правильности работы программы.

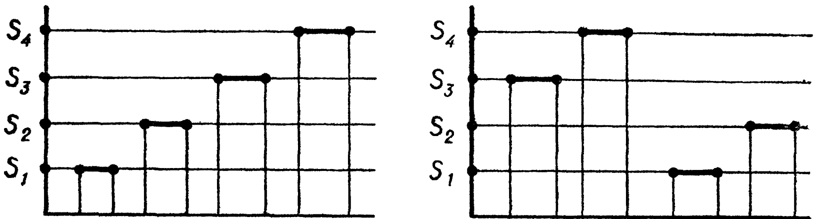
Интересным образом вводятся в асинхронном программировании понятия степени параллельности, или степени асинхронности программы. Количественной меры асинхронности не вводится, однако можно определить, какая программа является более асинхронной, чем другая, а также определить наиболее асинхронную программу среди множества программ, сравниваемых по асинхронности.

Рассмотрим две программы P_1 и P_2, эквивалентные в том смысле, что для одинаковых начальных состояний памяти они порождают вычислительные процессы с одинаковыми информационно-логическими графами. Для каждого информационно-логического графа *ИЛ* имеются соответственно множества M_1 и M_2 вычислительных процессов, обладающих этим графом. Тогда если для любого *ИЛ* множество M_1 включает M_2, то P_1 обладает большей асинхронностью. Таким образом, более асинхронной признается та программа, которая предоставляет вычислительной системе больше возможностей в разнообразии вычислительных процессов, реализующих данный информационно-логический граф. Здесь нет явного указания на больший параллелизм, и это правильно, так как степень параллельности программы фактически определяется не только ею самой, но и возможностями системы. Однако если программа P_1 может выполняться всеми способами, что и P_2, а также, быть может, и еще некоторыми способами, то шансов на более параллельное выполнение она имеет по крайней мере не меньше.

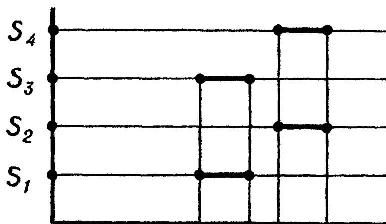
Отношение степени асинхронности имеет и более конструктивную форму определения. Степень разнообразия реализаций программы в конечном итоге упирается в совокупность бинарных отношений между операторами программы, определяющих,



а) программа



б) возможные вычислительные процессы



в) невозможный вычислительный процесс

Рис. 8. Неудачное распределение памяти с точки зрения параллелизма.

есть ли ограничения на взаимный порядок выполнения операторов. Совокупность этих бинарных отношений образует граф зависимости. Чем меньше ребер имеет граф зависимости, тем более асинхронна программа. Если программа имеет такой граф зависимости, что удаление любого ребра из него заведомо нарушает информационно-логические связи какого-либо выполнения программы, то такая программа имеет максимальную асинхронность.

Как уже указывалось, теория асинхронного программирования предлагает конструктивный метод формального перехода от последовательных программ, рассматриваемых в виде схем

программ, в асинхронные программы максимальной параллельности. Этот переход осуществляется в два этапа. На первом этапе производятся некоторые эквивалентные преобразования самой схемы программы, а затем осуществляется разовое преобразование схемы в асинхронную программу, состоящее в «рассыпании» схемы на отдельные операторы и в задании для каждого оператора его спусковой функции.

Оказывается, «барьером» на пути перехода от последовательной программы к максимально асинхронной программе лежит внутренняя асинхронность последовательной программы, которая может оказаться существенно меньше ее потенциальной асинхронности. Эта разница заключается в неудачном распределении памяти, налагающем искусственные связи между операторами, и в недостаточной расщепленности некоторых операторов на отдельные экземпляры, которые, выполняясь одновременно, могут увеличить разнообразие выполнений программы. Примером различия между внутренней и потенциальной асинхронностями служат две пары операторов, изображенные на рис. 8. Из-за того, что для передачи информации в каждой паре использована одна и та же величина y, эти две пары операторов не могут выполняться совместно, что было бы допустимо при использовании в каждой паре своей величины.

В работах Котова предложена методика преобразования любой схемы программы в эквивалентную схему, внутренняя асинхронность которой достигает ее потенциальный асинхронности.

При развитии теории параллельного программирования заранее было неясно, можно ли, не жертвуя общностью результатов, исключить количественные показатели течения времени, нанизывая на ось времени только факты появления событий, а не их длительность. Крайним выражением такой абстракции является допущение, что все изменения состояний происходят мгновенно, в том числе и срабатывания операторов. Нариньяни показал, что для любого информационно-логического графа существует реализующий его вычислительный процесс, в котором все операторы срабатывают мгновенно (так называемый приведенный процесс). Более того, им показано, что если квазипрограмма является программой для вычислительной системы с мгновенным срабатыванием операторов, то она остается программой и для вычислительной системы с любыми длительностями операторов.

В выполнении асинхронных программ существенное место занимает непрерывное вычисление спусковых функций. Если реализовать этот процесс в реальной системе, становится очень важным максимальное упрощение и ускорение этого вычисления. С этой точки зрения весьма интересно, что для любой асинхронной программы можно конструктивно найти ей эквивалентную, такую, что в этой программе все спусковые функции имеют вид логических функций элементарных условий, выражающих

лишь факт срабатывания того или иного оператора. Это позволяет придать без потери общности вычислению спусковых функций специфический вид, допускающий реализацию с использованием регистров прерывания и другой быстродействующей схемной техники.

§ 2. Обсуждение структуры вычислительных систем

Выполнение двоичных переключений и осуществление операций алгебры логики являются в предвидимом будущем непоколебимым фундаментом электроники, в наиболее непосредственной форме реализующим принцип дискретности действия. Однако одним из основных принципов осуществления примата программирования над оборудованием должно быть «заглубление» элементарных двоичных структур внутрь более крупных структур, отражающих априорный алгоритмический базис программирования. Автор уже отмечал, что сложившийся алгоритмический базис (арифметические действия и отношения над числами, ассоциативный и адресный поиск информации) носит, возможно, исторический характер и может быть подвергнут ревизии с позиций природного параллелизма. Тем не менее сейчас мы не имеем возможности выйти за рамки этого алгоритмического базиса, который, кстати, полностью учитывается видом современных алгоритмических языков и аксиоматикой теории программирования.

Именно поэтому автор не рекомендует двоичные однородные вычислительные среды с динамической настройкой в качестве первоочередного претендента на вычислительный универсум. Любая такая среда из-за своего универсализма будет проигрывать в производительности вычислительной системе, в которую структура алгоритмического базиса (сумматоры, множители, индекс-регистры, адресные коммутаторы и т. п.) внесена априорно в момент конструирования и решена с помощью богатого арсенала специальных приемов.

В частности, автор считает целесообразным сохранение сложившегося разделения между организацией преобразования информации в активных процессорах и ее хранением в пассивной памяти. Большой массив общей памяти с произвольной выборкой является наиболее приемлемым для техники программирования устройством, «поглощающим» и развязывающим все трудности, связанные с ограниченностью числа процессоров в системе и вытекающей из нее необходимостью быстрой коммутации процессоров на ту или иную работу в непредсказуемой заранее последовательности.

Распределение памяти между процессорами приводит к необходимости установления более жесткой синхронизации. Дей-

ствительно, если процессор *A* передает результаты своей работы другому процессору *B*, то это значит, что результаты составляют только часть той информации, которая нужна для работы *B* (если бы это было не так, то *A* мог бы продолжить работу сам, не передавая ее *B*). Но согласованное получение информации процессором *B* от разных частей системы может войти в противоречие с желанием как можно более полно использовать продуктивность процессора и не создавать простоев, так как эту продуктивность можно обеспечить, вообще говоря, только при «равнодоступности» любой работы любому процессору, что противоречит принципу распределенности памяти. Другая угроза производительности при распределенной памяти состоит в том, что для обеспечения дальних связей по передаче информации слишком много процессоров должны работать только лишь как пункты транзита. Таким образом, однородная система многих процессоров с распределенной памятью станет экономически оправданной, по мнению автора, лишь тогда, когда коэффициент использования процессоров будет сравнительно мал, измеряясь примерно теми же значениями, что и коэффициент использования ячеек оперативной памяти ЭВМ, измеряемый как число обращений к ячейке, отнесенное к длительности хранения информации.

Итак, в качестве первоочередного претендента на вычислительную систему ближайшего будущего автор выдвигает однородную вычислительную систему над общей памятью с достаточно развитыми рабочими процессорами, содержащими в качестве заложенной структуры сложившийся алгоритмический базис. Это соображение, однако, нуждается в некотором углублении.

Во-первых, программирование для такой системы должно базироваться на принципах асинхронности, так как такая архитектура системы наиболее адекватна абстрактной модели вычислительной системы, употребляемой в асинхронном программировании. Кроме того, асинхронность программы делает очень простой проблему назначения процессоров на работы, в том числе и в динамике, если только решена проблема быстрого сканирования спусковых функций программы.

Во-вторых, в системе должен быть реализован принцип разделения собственно вычислительного процесса (выполнения операторов программы) от процесса управления (сканирование спусковых функций или сигналов внешних прерываний и назначение процессоров на работу).

Очень заманчивым является концентрация процесса управления в специальном процессоре, который будет здесь называться монитором. Монитору, с одной стороны, должна быть доступна регистровая память процессоров, с другой стороны, он должен иметь в своей памяти всю «логическую схему» задачи, которая представляется монитору как генератор потока заявок, обработка которых путем назначения процессоров на ту или иную работу и составляет сущность управления.

Такой подход представляет интерес еще и потому, что он, как кажется автору, позволяет объединить в рамках одной архитектуры системы такие, казалось бы, противоположные виды работ, как организация коллективного пользования, т. е. распределение вычислительных ресурсов между потоком слабосвязанных заявок на обслуживание, и как параллельная многопроцессорная работа, т. е. концентрация вычислительных ресурсов для решения одной задачи. Единство же достигается тем, что если рассмотреть «анатомическую структуру» процесса управления решением большой задачи, записанной в виде асинхронной программы, то эта структура окажется как бы предельным случаем заявок на работы в режиме коллективного пользования.

Выделение управления в отдельный процесс, обслуживаемый монитором, имеет и свои слабые стороны. Одна из них состоит в предъявлении критических требований к скорости сканирования спусковых функций, назначения процессоров на работу и загрузки их регистровой памяти. Другое опасение состоит в том, что централизация управления понижает живучесть системы. При этом выдвигается другой подход, состоящий в том, что каждый процессор сам ищет для себя новую работу, как только он разделался с текущей. Это требует создания в общей памяти своеобразной «биржи труда», доступ в которую открыт ка́ждому процессору. Эта биржа может быть задублирована или иметь «филиалы» в участках памяти, преимущественно относящихся к тому или иному процессору.

Признавая преждевременным делать окончательные суждения в пользу централизованного или распределенного управления, автор тем не менее считает необходимым осуществить до конца проработку методов централизованного управления. Рассмотрение управления как самостоятельной функции, допускающей также и специальную техническую реализацию, имеет особое значение для объективного решения проблемы соотношения программных и аппаратных средств в осуществлении управления вычислительной системой. Опыт работы с развитыми операционными системами как для пакетной обработки, так и для коллективного пользования, показал, что в процессах управления складывается свой алгоритмический базис, структура которого может и должна быть воплощена в особых устройствах. Не исключено, что этот пока что специфический базис сольется впоследствии с «общеалгоритмическим» базисом, однако для этого он должен сначала возникнуть на основе рассмотрения четко поставленной и проработанной проблемы.

§ 3. Заключение

Каковы же должны быть, с точки зрения автора, основные направления работ в области программирования для параллельных вычислительных систем?

В качестве первого направления необходимо реализовать такие средства параллельного программирования, как параллельные ветви совместных вычислений в современных алгоритмических языках.

На основе методики асинхронного программирования, по-видимому, можно будет пополнить «ручные» методы способами автоматического выделения параллельных ветвей, разумного их сокращения и проверки асинхронности параллельной программы. Эта техника вполне может быть реализована уже в первых операционных системах для многопроцессорных конфигураций ЭВМ третьего поколения к середине 70-х годов. Некоторые основы такой техники можно найти в уже существующих у нас операционных системах.

В качестве второго направления, которое, как кажется автору, даст наиболее богатый выход еще до конца 70-х годов, должно быть развернутое построение асинхронного программирования как законченной теории, сомкнутой с формальной теорией схем программ, с тем чтобы оно стало рабочим инструментом параллельного программирования.

При этом следует сконцентрировать усилия на решении следующих вопросов:

— механизм вычисления спусковых функций;

— получение «логической схемы» параллельной программы в процессе трансляции и его пополнение или изменение в динамике работы;

— алгоритмы динамического и статического назначения процессоров на работу;

— методика быстрой коммутации процессоров при разворачивании циклических вычислений;

— организация буферизации и подкачки более быстрой памяти из более медленной;

— изучение возможностей распределенного и централизованного управлений;

— нахождение специального алгоритмического базиса процессов управления в вычислительных системах.

Успешное движение по этим направлениям поможет, по мнению автора, выйти к концу 70-х годов с вычислительной системой четвертого поколения с производительностью, превышающей 100 млн. операций в секунду.

В качестве третьего направления нужно поддерживать широкий фронт исследований по вычислительным средам с особым изучением следующих вопросов:

— поиск наиболее целесообразной универсальной ячейки среды и определение степени ее связности со средой;

— изучение граничных эффектов в ограниченных средах и проблемы динамической настройки среды;

— исследование возможностей и целесообразности сооружения «кратных», пронизывающих друг друга сред с разным функ-

циональным назначением (например, управляющая и исполнительная, вычисляющая и проводящая среды);

— разработка реальных устройств специального применения в таких задачах, структура которых сама по себе адекватна выбранной структуре среды;

— развитие принципов природного параллелизма и пересмотр на его основе нашего алгоритмического базиса.

По мнению автора, вычислительные среды в 80-х годах смогут стать побеждающим конкурентом в проблеме повышения потолка мощности вычислительных средств и в создании искусственного интеллекта.

ГЛАВА 3

УНИВЕРСАЛЬНЫЙ ПРОГРАММИРУЮЩИЙ ПРОЦЕССОР

В этой главе[1] дается постановка задачи и общая идеология создания универсального программирующего процессора, реализующего в одной системе программирования несколько входных алгоритмических языков с возможностью выполнения рабочих программ на вычислительных системах, состоящих из нескольких процессоров над общей памятью.

С одной стороны, идеология программирующего процессора следует линии разработки оптимизирующих трансляторов восходящей к программирующим программам, созданным в Вычислительном центре Академии наук СССР во второй половине 50-х годов [4]. С другой стороны, при реализации описываемого процессора делается попытка ввести в технологию программирования новые компоненты, пришедшие в основном из теории, такие, как управление трансляцией, осуществляемое описанием языка, автоматическое формирование параллельных ветвей решения одной задачи и их распределение по процессорам, алгоритмы глобальной оптимизации и ряд других. Некоторые из этих компонент существуют, но применяются только в экспериментальных системах или работают неэффективно, некоторые ставят перед программированием новые задачи.

Идеи, здесь изложенные, послужили отправной точкой для развертывания работ в Вычислительном центре Сибирского отделения АН СССР по созданию системы Бета — многоязыковой системы программирования для машин третьего поколения. Само изложение носит, однако, общий характер и не привязано к какой-либо конкретной машине.

---

[1] В основу настоящей главы положены доклады автора на Международной конференции по реализации Алгола 68 (Мюнхен, июнь 1970 г.) и на юбилейной конференции Комиссии по эксплуатации вычислительных машин (Москва, март 1971 г.).

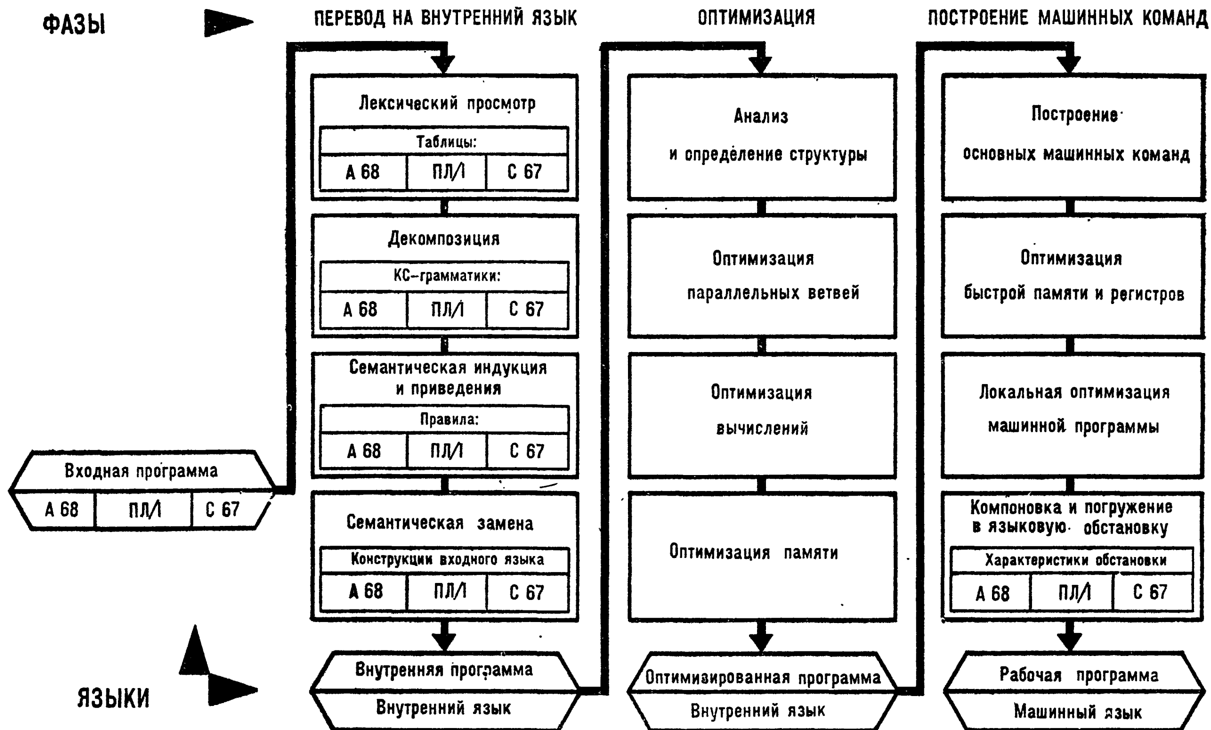


Рис. 9. Схема совместной реализации языков Алгол 68, ПЛ/I и Симула 67.

Процессор должен транслировать программы, написанные, в частности, в любом из следующих языков: Алгол 68, ПЛ/1 [9] и Симула 67. Ориентация процессора на тот или иной язык производится вставкой в некоторые места транслятора таблиц, составляемых для каждого языка по общим правилам.

Ключевым моментом в организации транслятора является специально разрабатываемый алгоритмический язык, который мы будем называть внутренним языком.

Его можно было бы назвать также промежуточным языком, потому что он является промежуточной стадией на пути перевода программы с входного языка на язык рабочей машины, или семантическим языком, потому что семантика входного языка будет задаваться в виде конструкций этого языка, выполняющих то или иное понятие входного языка. Нам кажется, однако, наиболее важным подчеркнуть внутренний характер языка в том смысле, что именно программирующий процессор, а не человек работает с текстами на этом языке.

Внутренний язык является также тем медиумом, в котором производится оптимизация транслируемой программы. Общетеоретические предпосылки к созданию такого языка рассматривались в конце первой главы.

На рис. 9 показана общая схема работы транслятора.

В последующих разделах главы будут более подробно обсуждаться этапы работы транслятора.

§ 1. Трансляция на внутренний язык

Перевод на внутренний язык включает в себя лексический анализ, контекстно-свободный разбор и декомпозицию некоторых рекурсивных конструкций входных языков, семантический анализ программ, «протаскивание» информации из описаний во все использующие вхождения идентификаторов, экспозицию приведений и буквальную замену понятий входных языков на подходящие конструкции Внутреннего языка.

Лексический анализ. Здесь было бы уместно отметить некоторые особенности Внутреннего языка. Как и в любом алгоритмическом языке, его грамматика строится с помощью совокупности понятий, которые не просто являются промежуточными инстанциями при разборе программы, но обозначают некоторые важные и логически независимые единицы языка. Естественно предполагать, что совокупность понятий Внутреннего языка не строится независимо от рассматриваемых входных языков. Язык машин Тьюринга вполне мог бы быть Внутренним языком, если бы мы заботились только о том, чтобы иметь принципиальную возможность точно описать процесс выполнения программ, написанных на входных языках. Однако этого недостаточно.

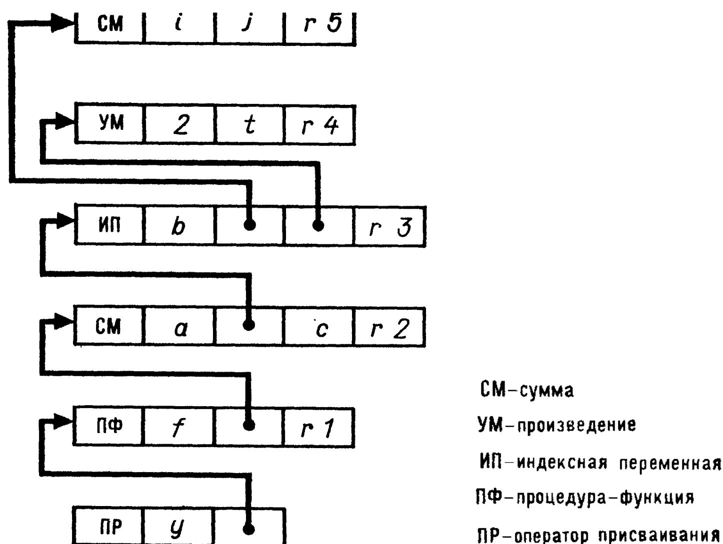


Рис. 10. Списочная структура программы после декомпозиции.

Мы хотим средствами Внутреннего языка найти и выделить то общее, что есть во всех этих входных языках. Мы ищем эту общность не только в их алгоритмической равномощности, но и в деталях их структуры, в типах операторов и в видах данных. Одной из форм материализации этой общности будет наличие большого числа понятий, общих как для Внутреннего языка, так и для входных языков (константы разных видов, идентификаторы, индексы, операторы присваивания, переходы, форматы, процедуры и т. п.).

В частности, предполагается, что во Внутреннем языке будет иметься общее для всех входных языков представление «лексем», то есть объектов, являющихся точками приложения лексического анализа (например, идентификаторов, чисел и строк). Замена лексем на их стандартное представление и заполнение соответствующих таблиц составляют содержание лексического просмотра.

Декомпозиция представляет собой контекстно-свободный разбор программы. В основе этого разбора, проводимого с помощью универсального разбирающего процессора, будут лежать контекстно-свободные грамматики входных языков.

Стандартным результатом разбора будет представление рекурсивных и вложенных структур в виде списочной структуры, элементы которой будут линейно упорядочены, следуя логическому порядку их исполнения. Так, например,

$$y := f(a + b[i + j,\ 2 \times t] + c)$$

перейдет в конструкцию, изображенную на рис. 10.

Вместо лексем $+$, i, j, 2 и т. д. здесь будут стоять их стандартные представления во Внутреннем языке. Эти лексемы, а также идентификаторы *r1*,..., *r5*, введенные для обозначения результатов выполнения промежуточных понятий языка *СМ*, *УМ*, *ИП* и т. д., пока что не содержат никакой семантической информации, однако расставленные ссылки позволят в дальнейшем найти и запомнить эту информацию.

Разбор описаний состоит в индуктивном конструировании информации, содержащейся в описаниях, и в перенесении этой информации в таблицы идентификаторов и операций. Формат этих таблиц определяется Внутренним языком.

Контекстные условия и приведения. Реализация контекстных условий и приведений представляет собой индуктивный процесс распределения предписанной информации, содержащейся в таблицах, по всем вхождениям лексем в программу. Этот процесс начинается с определяющих вхождений лексем, а затем по правилам идентификации распространяется на их использующие вхождения. Индуктивные шаги распространения информации появляются тогда, когда вовлекаются в дело промежуточные лексемы, появляющиеся при декомпозиции. Направление этой индукции указывается введенной системой ссылок.

В общем случае снабжение промежуточной лексемы семантической информацией происходит неоднозначно. При такой трактовке приведение — это сопоставление разных вариантов, предоставляемых шагом индукции для лексемы-операнда, единственной возможности, допускаемой данной позицией. Если позиция также допускает несколько возможностей, возникает неоднозначность; если варианты не совпадают с возможностями, возникает противоречие. Фактически же приведение состоит во вставке некоторого промежуточного действия, приводящего фактический вид операнда к виду, требуемому позицией.

Можно надеяться построить общую теорию идентификации и приведений, в которой понятия идентификатора, знака операции, области действия, промежуточных лексем, операнда и позиции будут трактоваться некоторым общим для всех языков методом. Эта общая трактовка позволит определить единый формат и единые правила обработки «семантических таблиц», в которых будут помещены специфические для каждого языка правила идентификации, индукции и приведения. Эти правила будут выражены в виде некоторых конкретных действий пополнения или изменения таблицы видов лексем, управляемых предикатами такого типа, как «Имеет ли лексема (L) вид (M)»? «Являются ли виды (M_1) и (M_2) родственными?»

Весь этот механизм будет, весьма вероятно, близок к соответствующему механизму Алгола 68, в котором он описан наиболее полно и явно.

Семантическая замена. Преобразование декомпозированных конструкций входных языков в программу на Внутреннем языке естественно назвать семантической заменой, так как это преобразование состоит в сопоставлении каждому исполнительному понятию N входного языка, некоторой программы на Внутреннем языке, в точности описывающей исполнение данного понятия. В общем случае такая программа состоит из нескольких вариантов, каждый из которых выбирается в зависимости от конкретных видов операндов понятия N. Если эти виды определены статически на предыдущем этапе, соответствующие предикаты могут быть вычислены при трансляции, и понятие заменяется наиболее соответствующей данному случаю программой. В некоторых случаях редукция общей программы исполнения понятия N к частной программе может быть сделана «автоматически» с помощью универсальных алгоритмов оптимизации.

§ 2. Внутренний язык

В этом разделе резюмируются установочные данные для Внутреннего языка. Их можно рассматривать как конкретизацию тех общих требований, изложенных в конце первой главы, которые должны обеспечить стыковку аппарата теории схем программ с реальными конструкциями алгоритмических языков.

Изобразительные средства. Используя арифметическую аналогию, Внутренний язык должен быть набором наибольших общих делителей основных понятий входных языков.

Эту аналогию мы могли бы сделать гораздо более глубокой, если бы уже сейчас могли ответить на вопросы, что такое в данном случае «наибольший», «делитель» и «основные». Однако эта фраза уже указывает на определенный подход к проектированию Внутреннего языка: мы отказываемся от построения априорной системы элементарных понятий для алгоритмических языков, а постараемся найти ее после изучения языков-кандидатов. Мы верим в то, что за этими языками нового поколения действительно стоит реальный мир программ, операционных систем и архитектур машин. Мы верим в то, что такие важные аспекты этих входных языков, как параллелизм и синхронизация, работа с каналами и форматами, действия с повышенной точностью, организация циклических процессов, механизм распределения памяти, могут быть выражены адекватными средствами на некотором абстрактном уровне, более высоком, чем язык машинных команд и битов информации.

Под абстрактным уровнем здесь понимается сохранение во Внутреннем языке той же степени неопределенности, которая существует во входных языках в отношении, например, деталей арифметики вещественных чисел, количественных ограничений памяти, диапазона чисел, времен выполнения, порядка исполнения параллельных процессов и так далее.

Внутренний язык можно понимать как систему команд абстрактного вычислителя, выполняющего программы на входных языках. Введя аксиоматически ту или иную интерпретацию команд Внутреннего языка, мы получим тем самым точное описание семантики входных языков, которое могло бы представлять интерес само по себе. На стадии перевода на Внутренний язык эта семантика уже имеет определенное значение, поскольку некоторые стадии выполнения программы могут иметь место уже в момент трансляции. Остальные же конструкции Входного языка формально вводятся в программу по правилам замены безотносительно к их смыслу.

Говоря об изобразительных средствах Внутреннего языка, необходимо подчеркнуть наличие в нем достаточного ассортимента «управляющих» действий, организующих надлежащее размещение данных и вычислений во времени и пространстве. Это не значит, что будет окончательно фиксироваться модель вычислителя (например, конкретный способ реализации косвенной адресации), однако концептуально все эти действия (формирование массивов, копирование значений, резервирование памяти, движение по ссылкам, подстановка фактических параметров, запуск параллельных ветвей) должны вещественно присутствовать в программе в виде явных команд.

Связь с рабочей машиной. Хотелось бы, чтобы Внутренний язык был машинно независимым языком. Этой независимости можно придать совершенно точный смысл, потребовав, например, чтобы смена рабочей машины ничего не меняла бы в первых двух фазах транслятора. Даже если это будет достигнуто, это, однако не значит, что отсутствует какая бы то ни была связь между Внутренним языком и машиной. Наоборот, Внутренний язык должен быть построен при вполне определенных гипотезах о возможной рабочей машине. Продолжая однажды использованную арифметическую аналогию, можно сказать, что со стороны машины Внутренний язык должен представлять набор наименьших общих кратных для машинных конструкций, которые должны использоваться при реализации выбранных входных языков.

Удовлетворить такому требованию было бы весьма просто, если быть уверенным, что все конструкции Внутреннего языка крупнее машинных команд. Обычно машина имеет ядро в своей системе команд, которое позволяет реализовать такой случай, однако ограничиваться только им — значит потерять эффективность, потенциально представляемую сложными командами машин (например, сложные команды управления циклами, объединение индексной арифметики с основными вычислениями и т. п.). Это значит, что Внутренний язык должен предоставлять специфические возможности для обнаружения таких рассеянных конструкций, которые могут быть объединены в одну машинную команду.

Структура языка и программы. Основное рабочее назначе-

ние Внутреннего языка — это быть удобным средством оптимизационных преобразований программы. Оптимизация основана прежде всего на анализе информационных и логических связей между операторами и на изучении структуры программы. Это налагает некоторые специфические требования на язык. Операторы должны быть простыми, легко отождествляемыми друг с другом, не содержать операндов с рекурсивным строением. Информационные и логические связи между операторами должны быть легко обнаруживаемы. Это означает, что преемники и предшественники должны указываться ссылками, а не обнаруживаться сканированием. То же относится к поиску операторов, снабжающих аргументами операнды данного оператора. Эти требования довольно обременительны, но зато они сокращают на порядки время работы алгоритмов оптимизации. Еще одним требованием является как можно более слабая контекстная зависимость операторов, что облегчает их перестановку.

Менее тривиальными являются требования к тем средствам Внутреннего языка, которые позволяют выявлять и изучать структуру программы.

Под структурой программы понимается разбиение множества операторов и переменных, образующих программу, на некоторые специальные подмножества, имеющие особое значение при оптимизации, а именно:

— потенциальные подпрограммы и линейные участки;

— участки повторяемости;

— области действия;

— параллельные ветви.

Потенциальная подпрограмма, или «гамак»,— это связное множество G операторов программы, в котором есть две вершины — вход EN и выход EX — такие, что войти в G можно только через оператор EN и выйти из G можно только через EX. Важным частным случаем G являются линейные участки, когда все операторы из G, кроме, быть может, EX, имеют в точности по одному преемнику.

Сейчас еще нельзя дать точного определения участка повторяемости, и мы лишь скажем, что это такое связное множество операторов программы, о которых можно заключить, что они в совокупности выполняются не чаще или не реже, чем какая-либо другая совокупность операторов. Можно сделать следующие частные, но важные наблюдения:

— тело цикла выполняется не реже, чем непосредственные преемник и предшественник соответствующего оператора цикла;

— любой оператор внутри тела процедуры выполняется не реже, чем вход и выход процедуры;

— любой конкретный вызов процедуры выполняется не чаще, чем тело этой процедуры;

— транслятор выполняется не чаще, чем любая транслируемая им программа.

Как будет показано ниже, участки повторяемости играют фундаментальную роль в построении универсальных алгоритмов оптимизации.

Точное определение области действия S некоторой переменной величины v программы требует слишком больших конструкций, чтобы быть точно сформулированным здесь. Для наших целей достаточно будет сказать, что с каждой переменной v можно связать некоторое множество *Out* операторов, инициирующих переменную v, и соответствующее множество *In* операторов, воспринимающих v. Тогда областью действия S будет множество операторов, проходимых при «транспортировке» значений v от операторов из *Out* к операторам из *In*. Очевидно, что если для пары переменных v_1 и v_2 множество *Out* для одного из них не пересекается с множеством S другого, то v_1 и v_2 могут быть размещены в одном участке памяти.

Наконец, мы скажем, что программа имеет параллельные ветви $P_1, \ldots, P_n$, если задано разбиение операторов программы на множества $P_1, \ldots, P_n$, при этом для каждой пары P_i и P_j известно, могут ли операторы из P_i и P_j выполняться совместно или нет.

Вероятно, для каждой из структур указанного типа можно разработать универсальные алгоритмы их нахождения. Это верно по крайней мере для потенциальных подпрограмм и областей действия. Однако полагаться только на общие алгоритмы и рассматривать перед оптимизацией программу на Внутреннем языке просто как «однородный» граф операторов над полем некоторых переменных было бы неразумно. Входные языки содержат, по своей сути, очень полезные сведения о структуре своих программ, которые должны быть полностью сохранены после перевода на Внутренний язык.

Перечислим те структуры, которые вносятся в программы большинством современных алгоритмических языков:

гамаки: любой блок или тело процедуры, не содержащее выходов по внешней метке;

линейные участки: арифметическое выражение, не содержащее обращений к описанным процедурам;

участки повторяемости: тела циклов, тела процедур, блоки;

области действия: тела процедур, блоки, тела циклов для своего параметра, программы арифметических выражений для своих рабочих ячеек, любые гамаки для их внутренних величин;

параллельные ветви: компоненты совместных фраз, программы вычисления фактических параметров, аргументы любой бинарной операции в Алголе 68, ветви в ПЛ/1, блоки в Симуле 67.

Очевидно, что Внутренний язык должен сохранить разнообразие ограничителей, позволяющих явно указать границы соответствующих структур. Заметим, что эти ограничители должны быть уже одними и теми же, независимо от того, к какому входному языку относится транслируемая программа. Например,

независимо от того, каковы различия синтаксических средств указания тела процедуры в языках ПЛ/1, Симуле 67 или Алголе 68, способ выделения соответствующей группы операторов должен быть один и тот же во Внутреннем языке.

Кроме введения подходящих ограничителей, дополнительным средством сохранения структуры программы является классификация переменных или операторов по их специфическим свойствам, облегчающим нахождение связанных с ними структур. К таким средствам относятся, например:

отдельные обозначения для статических и динамических массивов;

отдельные обозначения для рабочих ячеек арифметических выражений;

отдельные обозначения для переменных, которым имеется только одно присваивание.

Унификация всех таких средств на уровне Внутреннего языка может оказаться существенным вкладом в выявление тех универсальных внутренних структур программ, которые имеют значение для эффективной оптимизации программы.

§ 3. Оптимизация

Из предыдущего стало очевидным, что сохранение структурных свойств программы, заложенных в нее средствами входного языка, является основой для оптимизирующих алгоритмов. Мало того, обычно до сих пор оптимизирующие алгоритмы очень тесно связывались с конкретными свойствами входного языка и были настолько от них неотделимы, что даже мало кому приходило в голову описывать их независимо, вне связи с конкретной реализацией данного языка.

Экспозиция оптимизирующих структур программ на уровне Внутреннего языка в форме, не связанной с индивидуальными особенностями входных языков, позволяет поставить вопрос о нахождении универсальных алгоритмов оптимизации, обеспечивающих получение качественной рабочей программы безотносительно к ее происхождению. К этим универсальным алгоритмам оптимизации относятся:

оптимизация вычислений (разгрузка участков повторяемости, исключение избыточных вычислений, исключение тождественных вычислений, исключение неиспользуемых вычислений);

оптимизация памяти;

оптимизация параллельных вычислений.

Каждому применению алгоритма оптимизации соответствует некоторый участок оптимизации, а именно та часть программы, к которой относится данное применение.

Следует отличать: а) локальные, б) квазилокальные и в) глобальные алгоритмы, для которых участком оптимизации явля-

ется соответственно: а) один или ограниченное число близких операторов, б) некоторый участок программы регулярной структуры, в) вся программа.

Разгрузка участков повторяемости в своем универсальном применении является в высшей степени эффективным алгоритмом. Существуют универсальные алгоритмы, которые позволяют для любого оператора и любой точки программы определить, может ли оператор быть перенесенным в эту точку без потери необходимых информационных связей. Можно, далее, разработать алгоритмы, позволяющие для любого оператора S, находящегося на некотором участке повторяемости R, определить, является ли действие оператора S статическим или динамическим по отношению к R, т. е. вырабатывает ли S одно и то же или разные значения при разных повторениях участка R. Тогда разгрузкой участка повторяемости R является систематический перенос всех его статических операторов на участки программы, повторяемость которых не выше, чем повторяемость участка R. В общем случае это глобальный алгоритм, хотя для некоторых его частных применений он может быть квазилокальным (чистка циклов) или даже локальным (выполнение действий над константами в процессе трансляции).

Если описать работу транслятора на том же языке, что и язык транслируемой программы, то, применяя систематически разгрузку участков повторяемости к таким участкам, как сам транслятор и транслируемая программа, можно разработать целую философию программирующих процессоров с непрерывным спектром режимов работы — от чистой интерпретации до чистой компиляции.

Исключение избыточных вычислений состоит из двух частей — обнаружения совпадающих выражений (имеется в виду либо текстуальное совпадение, либо сводимость с помощью некоторой системы соотношений — например, коммутативности операндов) и выяснения, какие повторные вхождения совпадающих выражений являются избыточными, с тем чтобы в последующем удалить их. Это — типичный пример комбинаторной оптимизации, давно атакуемой системными программистами.

Имеется разработанная техника экономии выражений на линейных участках и простых ветвлениях; недавно стали появляться работы с описанием глобальных алгоритмов экономии. Реализация алгоритма тесно связана с экономией памяти, в частности с построением областей действия.

Исключение тождественных и неиспользуемых вычислений по крайней мере по отношению к программам, написанным рукой человека, самостоятельной роли не играет. Их основная роль — чистка «мусора», который появляется в результате работы других алгоритмов оптимизации.

Оптимизация памяти является крайне важным видом оптимизации из-за постоянной нехватки быстродействующей памяти.

Имеет два аспекта: статический, связанный с экономным распределением ячеек памяти, и динамический, связанный с понятием и реализацией виртуальной памяти.

Статический аспект требует априорного знания областей действия различных величин. Существуют универсальные алгоритмы построения областей действия и некоторая разумная комбинаторика распределения переменных с учетом отношения их несовместимости. Более реальными, однако, представляются систематические алгоритмы статического или динамического — по принципу магазина — распределения памяти для переменных, областями действия которых являются регулярные вложенные компоненты программы (блоки, гамаки, линейные участки — для рабочих ячеек).

Динамический аспект здесь не рассматривается, так как сильно связан с конкретными особенностями рабочей машины и ее оперативной памяти.

Оптимизация параллельных вычислений. В соответствии с выводом второй главы в качестве рабочего варианта реализации параллельных вычислений принимается методика мультипрограммирования. Она исходит из того, что вычислительная система имеет сравнительно небольшое число процессоров (исчисляемое единицами), а назначение процессора на работу является сравнительно редким событием.

Параллельные ветви в программе не обнаруживаются автоматически, а задаются явными средствами входных языков. Каждая параллельная ветвь представляет собой выделенную часть программы, которая для операционной системы является «единицей работы». Таким образом, программа в целом является собранием работ $T_1, \ldots, T_n$. Любая параллельная ветвь начинается с некоторой точки ветвления и кончается точкой схода.

В любой точке ветвления B, например на ветви T' и T'', ставится обращение к операционной системе, которое гласит о завершении вычисления, предшествовавшего точке B, и о появлении двух новых работ T' и T''. Некоторые характеристики этих работ (адреса начала, длительность и т. д.) также передаются операционной системе.

Окончание всех работ, имеющих общую точку схода, приводит к выполнению в этой точке обращения к операционной системе, инициирующего появление новой работы, расположенной непосредственно за точкой схода.

Такой подход не навязывает операционной системе какой бы то ни было стратегии назначения процессоров на работы, но лишь, через динамические обращения к ней, сообщает ей непротиворечивые сведения о работах — возможных кандидатах на вычисление.

В этом случае, однако, важно, чтобы число параллельных ветвей не превосходило бы существенно числа рабочих процессоров, а сами ветви были бы как можно более длинными. Это со-

кратит число взаимодействий программы с операционной системой.

В то же время правила, например, Алгола 68 приводят к тому, что буквальное следование семантике создает в программе слишком большое количество слишком коротких параллельных ветвей. Как известно, источниками совместных исполнений в Алголе 68 являются собственно совместные фразы, а также списки операндов процедур и компоненты многомерных массивов. Можно в связи с этим вспомнить критику в адрес Алгола 68, состоящую в том, что в этом языке концептуально не различается внутренний параллелизм, вносимый в программу автоматически (компоненты массивов и операнды процедур), и параллелизм, сознательно вводимый в программу (совместные фразы).

Этот недостаток языка можно, однако, обратить в его достоинство, если разработать универсальные алгоритмы сокращения количества и увеличения длины параллельных ветвей программы по отношению к той структуре, которая будет создана при разборе исходного текста.

§ 4. Фаза построения машинных команд

В основной своей части эта фаза не выдвигает никаких существенно новых проблем и поэтому здесь не обсуждается. Имеет смысл отметить только одну особенность, связанную с погружением рабочей программы в конкретную обстановку входного языка. К элементам этой обстановки относятся: стандартные программы, вызываемые в процессе работы; специфические административные системы; подпрограммы исполнения форматов при операциях ввода — вывода; специальные средства отладки; редакторы печати и т. п.

Любая транслируемая программа имеет точки соприкосновения с этой обстановкой. На первой стадии построения машинных команд эти точки соприкосновения имеют некоторый условный вид, обозначая, но не являясь фактическими командами вызова и загрузки параметров. Таблицы языковой обстановки в этом смысле имеют вид словарей, которые по условному коду точки соприкосновения выдают реальные команды, которые надо включить в программу, а также некоторую дополнительную информацию, которую нужно использовать для правильной ассимиляции этих команд.

§ 5. Заключение

В этой главе описана попытка системного подхода к разработке многоязыковой системы программирования, когда общие контуры системы и надлежащая философия фиксируются раньше,

чем будут накоплены решения частных проблем. Многоязыковой характер системы обеспечивается ориентацией на описание языка, которое управляет работой транслятора на начальной фазе.

Описание языка состоит из пяти частей:

— таблиц лексического анализа;

— контекстно-свободной грамматики для разбора;

— таблиц семантической индукции и правил приведений;

— таблиц семантической замены конструкциями Внутреннего языка;

— таблиц языковой обстановки.

Транслятор должен обеспечить высокое качество трансляции благодаря применению универсальных алгоритмов оптимизации. Эффективность этих алгоритмов существенно зависит от того, насколько хорошо будет использована информация о структуре программы, заложенная конструкциями входных языков Алгол 68, ПЛ/1 и Симула 67.

В случае успеха этот подход сможет внести вклад в решение следующих проблем:

— экономия усилий и затрат по сравнению с разработкой нескольких независимых трансляторов;

— разработка универсальной техники оптимизации;

— нахождение базисных концепций алгоритмических языков;

— объективное сравнение языков Алгола 68, ПЛ/1 и Симула 67.

ЛИТЕРАТУРА

1. Дал О. И., Мюрхаут Б., Нюгард К. Симула 67. Универсальный язык программирования. М., «Мир», 1968.
2. Ершов А. П. Об операторных схемах Янова. «Проблемы кибернетики», 20. М., «Наука», 1967.
3. Ершов А. П., Ляпунов А. А. О формализации понятия программы. «Кибернетика» [Киев], № 5, 1967.
4. Ершов А. П. Программирующая система для БЭСМ. М., Изд-во АН СССР, 1958.
5. Криницкий Н. А. Равносильные преобразования алгоритмов и программирование. М., «Советское радио», 1970.
6. Криницкий Н. А., Миронов Г. А., Фролов Г. Д. Программирование. М., «Наука», 1966.
7. Лавров С. С. Об экономии памяти в замкнутых операторных схемах. «Журнал вычислительной математики и математической физики», I, № 4, 1961.
8. Марковиц Г. и др. Симскрипт — алгоритмический язык для моделирования. М., «Советское радио», 1966.
9. Универсальный язык программирования PL/1. М., «Мир», 1968.
10. Янов Ю. И. О логических схемах алгоритмов. «Проблемы кибернетики», I. М., Физматгиз, 1968.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 3

Глава 1 6
ТЕОРИЯ СХЕМ ПРОГРАММ

Глава 2 30
ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ

Глава 3 46
УНИВЕРСАЛЬНЫЙ ПРОГРАММИРУЮЩИЙ ПРОЦЕССОР

ЛИТЕРАТУРА 60

ЕРШОВ Андрей Петрович
ТЕОРИЯ ПРОГРАММИРОВАНИЯ
И ВЫЧИСЛИТЕЛЬНЫЕ СИСТЕМЫ

Редактор В. Ю. Иваницкий
Обложка Л. П. Ромасенко
Худож. редактор Т. И. Добровольнова
Техн. редактор Т. В. Самсонова
Корректор Г. П. Ефименко

А01606 Сдано в набор 29/III 1972 г. Подписано к печати 17/V 1972 г. Формат бумаги 60×90[1]/16 Бумага типографская № 3 Бум. л. 2,0 Печ. л. 4,0 Уч.-изд. л. 3,44 Тираж 48000 экз. Издательство «Знание». Москва, Центр, Новая пл., д. 3/4. Заказ 581 Цена 12 коп.

Чеховский полиграфкомбинат Главполиграфпрома
Комитета по печати при Совете Министров СССР
г. Чехов Московской области