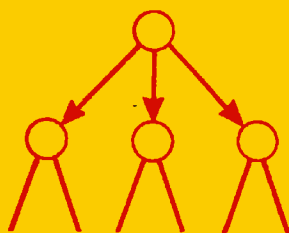


Ч.Хоар

Взаимодействующие последовательные процессы



Издательство «Мир»

Взаимодействующие последовательные процессы

Ч.Хоар

**Взаимодействующие
последовательные
процессы**

Перевод с английского
А. А. Бульонковой
под редакцией
А. П. Ершова



Москва «Мир» 1989

Prentice-Hall International Series in Computer Science

**Communicating Sequential
Processes**

C. A. R. Hoare
Professor of Computation
Oxford University

Prentice-Hall
Englewood Cliffs, New Jersey London Mexico
New Delhi Rio de Janeiro Singapore
Sydney Tokyo Toronto Wellington

ББК 22.18
X68
УДК 681.3

Хоар Ч.

X68 Взаимодействующие последовательные процессы:
Пер. с англ. — М.: Мир, 1989. — 264 с., ил.

ISBN 5-03-001043-2

Книга известного системного программиста и теоретика информатики (Великобритания), последовательно излагающая теорию взаимодействующих процессов; эта тематика тесно связана с такими реальными понятиями, как операционные системы, мультипроцессорные комплексы и сети ЭВМ. Автор рассматривает параллелизм в языках высокого уровня АДА, Симула 67, Паскаль.

Для специалистов в области системного программирования, теоретической информатики, математической логики, аспирантов и студентов вузов.

X $\frac{1702050000-480}{041(01)-89}$ 22-88, ч. 1

ББК 22.18

Редакция литературы по математическим наукам

ISBN 5-03-001043-2 (русск.) © 1985 by Prentice-Hall International,
UK, LTD
ISBN 0-13-153271-5 (англ.) © перевод на русский язык, «Мир»,
1989

От редактора перевода

Эта книга написана выдающимся ученым, лауреатом премии Тьюринга, внесшим большой вклад в теорию и практику программирования. Автор написал к книге подробное предисловие, в котором обстоятельно изложил свой подход к предлагаемому материалу.

Другой, не менее выдающийся ученый, тоже лауреат премии Тьюринга, написал изящное вступление, воздающее должное автору и подогревающее интерес к книге.

В этом контексте намерение писать третье предисловие, хотя бы и по праву редактора перевода, легко может быть воспринято как желание «войти в компанию». Мою решимость поддерживает сознание того, что мне хочется не столько вписать еще пару абзацев в панегирик автору (хотя и тут мне было бы что сказать), сколько побудить читателя к критическому прочтению этой книги.

Здесь уместно сделать одно пояснение. Мы зачастую несколько упрощенно представляем себе, что критиковать — это значит выискивать недостатки. Не желая спорить с этим расхожим представлением, хочу добавить, что критический разбор произведения — это способность приобщиться к мысли и видению автора, сохраняя запас широты восприятия, независимость мышления и самостоятельность собственной отправной позиции.

Предупредить об этом меня побуждает благоприятное смещение стилей, случившееся в этой книге: это монография, написанная с убедительностью учебника. Вспомним словарь иностранных слов: «Монография — научный труд, углубленно разрабатывающий одну тему, ограниченный круг вопросов». Автор «Взаимодействующих последовательных процессов» специально подчеркивает монографический характер книги и свой авторский подход к посылкам, границам и целям исследования. В то же время глубина и скрупулезность проработки, редкая для нашей программистской «лохматости» элегантность обозначений и конструкций, прекрасно подобранные примеры и крепко сбитая структура книги создают

комфортабельное ощущение убедительности и своего рода однозначности материала книги.

С этой книгой удобно работать, потому что она написана как отличный учебник. Однако ее нельзя изучать по-школярски, потому что это монографическое исследование в новом и труднейшем разделе программирования. Изложенный материал не исчерпывает всех проблем на долгом пути жизненного цикла разработки программ, а сам подход к решению этих проблем ограничивается особенностями выбранной математической модели.

Академик С. Л. Соболев как-то сказал нашему студенческому курсу: «Готовиться к экзамену по плохим лекциям даже лучше. Пробелы в конспектах заставят вас их восстановить». Эта книга не дает читателю подобного шанса. Важно, однако, сказать, что законченность изложения и другие неординарные качества книги являются результатом двадцатилетнего труда автора и неоднократной переработки вариантов. Если читатель примет на себя хотя бы малую долю этого труда во время работы над книгой, он сможет удовлетворенно заметить, что не только приобщился к предмету изложения, но и продвинул еще на один шаг развитие теории и методов параллельного программирования.

Академгородок,
январь 1988

А. П. Ершов

Предисловие

По многим причинам эту книгу с интересом ждали все, кто знал о планах ее создания; сказать, что их терпение вознаграждено, означало бы не сказать почти ничего.

Причина проста: это первая книга Тони Хоара. Многие знают его по лекциям, с которыми он неустанно выступает по всему свету; гораздо большему числу он знаком как искусный и вдумчивый автор изрядного количества (и разнообразия!) статей, которые становились классикой раньше, чем успевала высохнуть типографская краска. Но книга — это нечто другое: здесь автор свободен от стеснительных ограничений сроков и объема; книга предоставляет ему возможность глубже выразить себя и охватить тему более широко. Эти возможности Тони Хоар использовал лучше, чем мы могли себе представить.

Более серьезная причина кроется в непосредственном содержании книги. Когда примерно четверть века назад сообщество программистов столкнулось с параллелизмом, это вызвало бесконечную путаницу, частично из-за крайнего разнообразия источников параллелизма, частично из-за того, что волею судеб это совпало по времени с началом изучения проблем недетерминизма. Для развязки этой путаницы требовался тяжелый труд зрелого и целеустремленного ученого, которому при некотором везении удалось бы прояснить ситуацию. Тони Хоар посвятил значительную часть своей научной жизни этой работе, и у нас есть тысяча причин быть ему благодарными за это.

Основная причина, однако, наиболее остро была осознана теми, кому довелось увидеть ранние варианты рукописи книги и кто с удивительной ясностью вдруг увидел в новом свете, какой могла бы — или даже должна бы — быть вычислительная наука. Провозгласить или почувствовать, что главная задача специалиста в области информатики состоит в том, чтобы не оказаться погребенным под сложностью своих же собственных конструкций, — это одно, и совсем другое дело — обнаружить и пока ать. как стр г е п дчинени сов р-

шенно явной и осязаемой элегантности небольшого числа математических законов позволяет достичь этой высокой цели. Именно в этом, смею думать, признательные читатели в наибольшей степени извлекут пользу из научной мудрости, нотационной безупречности и аналитического искусства Чарльза Энтони Ричарда Хоара.

Эдсгер В. Дейкстра

От автора

Эта книга — для ищущего программиста, программиста, который стремится глубже понять и лучше овладеть практическим искусством своей наукоёмкой профессии. Прежде всего книга вызывает к естественному любопытству, возникающему благодаря новому подходу к хорошо знакомым вещам. Этот подход иллюстрируется большим числом практических примеров, взятых из самого широкого круга приложений: от торговых автоматов, игр и сказочных историй до операционных систем ЭВМ. В основе изложения лежит математическая теория, описанная в виде систематического набора алгебраических законов. Конечной целью является выработка у читателя особой проницательности, которая даст ему возможность увидеть как текущие, так и будущие проблемы в новом свете, что позволит решать эти проблемы экономнее и надежнее и, что еще важнее, порой избегать их.

Наиболее очевидной сферой применения новых идей служит спецификация, разработка и реализация вычислительных систем, которые непрерывно действуют и взаимодействуют со своим окружением. Основная идея заключается в том, что эти системы без труда можно разложить на параллельно работающие подсистемы, взаимодействующие как друг с другом, так и со своим общим окружением. Параллельная композиция подсистем ничуть не сложнее последовательного сочетания строк или операторов в обычных языках программирования.

Такой подход обладает целым рядом преимуществ. Во-первых, он позволяет избежать многих традиционных для параллельного программирования проблем, таких, как взаимное влияние и взаимное исключение, прерывания, семафоры, многопоточная обработка и т. д. Во-вторых, он включает в себя в виде частных случаев многие из актуальных идей структурирования, которые используются в современных исследованиях по языкам и методологии программирования; мониторы, классы, модули, пакеты, критические участки, конверты, формы и даже заурядные подпрограммы. И наконец, этот подход является надежной основой для избежания таких

ошибок, как расходимость, тупиковые ситуации, заикливание, а также для доказательства правильности при проектировании и разработке вычислительных систем.

Я стремился изложить свои мысли в определенной логической и психологической последовательности, начиная с простых элементарных операторов и постепенно переходя к более сложным приложениям. Усердный читатель, возможно, прочтет всю книгу целиком, от корки до корки. Но у многих одни разделы вызовут больший интерес, чем другие; в помощь им, чтобы облегчить разумный выбор, материал каждой главы тщательно структурирован.

(1) Каждая новая идея предварена ее неформальным описанием и проиллюстрирована рядом небольших примеров, полезных, вероятно, всем читателям.

(2) Алгебраические законы, описывающие основные свойства различных операций, будут интересны читателям, ценящим в математике красоту. Они окажутся полезными также тем, кто хочет оптимизировать разработку своих систем с помощью преобразований, сохраняющих корректность.

(3) Необычность предложенных способов реализации состоит в том, что в них используется очень простое и чисто функциональное подмножество языка программирования ЛИСП. Это доставит особенное удовольствие тем, кто работает с ЛИСПом, и благодаря этому получит возможность реализовать и испытать свои замыслы.

(4) Определения протоколов и спецификаций заинтересуют специалистов-системщиков, которым приходится специфицировать требования заказчика, прежде чем приступить к исполнению. Эти понятия пригодятся также ведущим программистам, перед которыми стоит задача проектирования системы путем разбиения ее на подсистемы с четко описанными интерфейсами.

(5) Правила доказательств будут интересны тем, кто серьезно относится к стоящей перед ними задаче написания надежных программ по заданной спецификации в установленные сроки и с фиксированными затратами.

(6) И наконец, эта математическая теория дает строгие определения понятия процесса и способов построения процессов. Эти определения лежат в основе алгебраических законов, реализаций и правил доказательства.

Читатель может, либо придерживаясь своей системы, либо выборочно, опускать или откладывать на потом те из перечисленных разделов, которые или менее интересны, или вызывают трудности в понимании.

Структура книги позволяет взыскательному читателю либо ограничиться беглым просмотром, либо определить порядок

чтения и выбор глав. Первые разделы гл. 1 и 2 представляют собой введение, необходимое для всех читателей, тогда как их последующие разделы допускают более поверхностное знакомство или же могут быть отложены до следующего прочтения. Главы 3, 4 и 5 не связаны друг с другом, и знакомство с ними может происходить в любом порядке и комбинации в зависимости от интересов и намерений читателя. Поэтому, если на некотором этапе возникнут трудности с пониманием, желательно продолжить чтение со следующего раздела или даже главы, так как, вполне вероятно, пропущенный материал не потребует немедленно. Если же потребность в нем возникает, то, как правило, дается явная ссылка на предыдущий материал, которой при необходимости можно воспользоваться. Я надеюсь, что в конце концов все в этой книге окажется интересным и полезным, однако порядок чтения и проработки может быть индивидуальным.

Примеры, выбранные для иллюстрации заключенных в книге идей, покажутся очень простыми. Это сделано сознательно. Первые примеры, иллюстрирующие каждую новую идею, должны быть просты настолько, чтобы сложность и необычность самого примера не могли затемнить идею. Некоторые из последующих примеров отличаются большей тонкостью; стоящие за ними проблемы способны породить немало путаницы и сложностей. Простоту их решения следует отнести за счет мощности используемых понятий и элегантности выражающей их нотации.

Тем не менее каждый читатель знаком (возможно, до боли знаком) с проблемами куда более сложными, обширными и важными, нежели те, которые служат примерами во вступительной части. Может показаться, что эти проблемы не под силу никакой математической теории. Мой совет — не поддаваться ни гневу, ни отчаянию, а попробовать испытать на этих проблемах новый метод. Начните с самого упрощенного случая какой-нибудь выбранной вами стороны проблемы и постепенно усложняйте ее в меру необходимости. Просто удивительно, как часто изначально переупрощенная модель позволяет увидеть путь к решению всей проблемы в целом. Возможно, ваша модель послужит структурой, на которую в дальнейшем без риска будут накладываться разные усложнения. И главным сюрпризом будет то, что многие из этих усложнений, вероятно окажутся попросту ненужными. В этом случае усилия по освоению нового метода вознаграждаются особенно щедро.

Обозначения часто служат причиной недовольства. Студент, начинающий изучение русского языка, жалуется на трудности запоминания незнакомых букв кириллицы, осо-

бенно из-за того, что многие из них имеют непривычное произношение. В порядке утешения скажу, что эта проблема — далеко не главная. Вслед за алфавитом вам предстоит изучить грамматику, накопить словарный запас, затем овладеть знанием идиом и стиля, а после этого — обрести способность свободно выражать на этом языке свои мысли. Все это требует старания, тренировки и времени, и поспешности здесь не место. Так же и в математике. Поначалу символы могут оказаться серьезным препятствием, однако истинная проблема состоит в постижении смысла и свойств этих символов, того, как можно и как нельзя их использовать, в умении свободно обращаться с ними при описании новых задач, решений и доказательств. В конце концов у вас выработается вкус к хорошему математическому стилю. К тому времени символы станут невидимыми; сквозь них вы будете видеть именно то, что они означают. Большое преимущество математики в том, что правила ее значительно проще, а словарь — гораздо скуднее, чем в естественных языках. Вследствие этого, сталкиваясь с незнакомыми вещами, путем логической дедукции или удачного приема вы сами сможете найти решение, не обращаясь к справочникам и специалистам.

Вот почему математика, как и программирование, доставляет порой истинное наслаждение. Но достичь этого бывает нелегко. Даже математик испытывает трудности при изучении новых направлений в своей науке. Теория взаимодействующих процессов — новое направление в математике; программист, приступающий к изучению этой теории, не окажется в невыгодном положении по сравнению с математиком; в конце же он обретет явное преимущество, имея возможность найти полученным знаниям практическое применение.

Материал этой книги неоднократно апробировался как на рабочих семинарах, так и на основных курсах лекций. Сначала он был задуман в качестве семестрового спецкурса по технологии программирования, хотя большая его часть может быть изложена на последнем и даже на втором году общего курса по информатике. Перечень исходных знаний ограничивается некоторым знакомством со школьной алгеброй, понятиями теории множеств и символикой исчисления предикатов. Книга может быть использована также в качестве интенсивного недельного курса для профессиональных программистов. В таком курсе следует делать упор на примеры и определения, оставляя математические обоснования для последующей самостоятельной работы. Материал первых двух глав вполне пригоден и для более компактного курса, и даже часовой обзор книги может быть полезен при условии тща-

тельного отбора материала, достаточного для разбора весьма поучительной истории о пяти обедающих философах.

Надо сказать, что чтение лекций и проведение семинаров по теории взаимодействующих процессов доставляет огромное удовольствие, а рассмотрение примеров дает широкие возможности для развития сценического искусства лектора. Каждый пример — это маленькая драма, которая может быть поставлена с должным учетом эмоций действующих лиц. Особую реакцию аудитории вызывают шуточные сценки на темы дедлока. Однако нужно постоянно предупреждать слушателей об опасности антропоморфизма. Математические формулы сознательно абстрагированы от всех мотивов, предпочтений и эмоциональных реакций, к которым прибегает лектор «для придания художественного правдоподобия повествованию бесцветному и малоубедительному»¹⁾. Так что нужно учиться концентрировать внимание на сухом тексте математических формул и развивать умение восхищаться прелестью их абстрактности. В частности, это относится к некоторым рекурсивно определенным алгоритмам, чья красота в чем-то сродни захватывающей дух красоте фуг И. С. Баха.

¹⁾ Гильберт В. Микадо, Кибернетика, 6, 1969 (пер. А. Ф. Пара). — *Прим. перев.*

Краткое содержание

В гл. 1 вводится общее понятие процесса как математической абстракции взаимодействия системы и ее окружения. Показано, как с помощью известного механизма рекурсии можно описывать протяженные во времени и бесконечные процессы. Вначале идея иллюстрируется с помощью примеров и рисунков; более полное истолкование дают алгебраические законы, а также машинная реализация на функциональном языке программирования. Вторая часть главы посвящена тому, как можно представить поведение процесса в виде протокола последовательности его действий. Определены многие полезные операции над протоколами. Еще до реализации процесс может быть специфицирован путем описания свойств его протоколов. Даны правила, помогающие получить реализации процессов, сопровождаемые доказательством их соответствия исходным спецификациям.

Вторая глава описывает способы построения из отдельных процессов систем, компоненты которых взаимодействуют друг с другом и с общим внешним окружением. Введение параллелизма здесь не влечет за собой появления какого бы то ни было недетерминизма. Основным примером в этой главе служит трактовка известной притчи о пяти обедающих философах. Во второй части главы мы показываем, что процесс можно легко использовать в новых целях, изменив названия составляющих его событий. Главу завершает описание математической теории детерминированных процессов, включающей простой очерк теории неподвижной точки.

В третьей главе дается одно из простейших известных решений наболевшей проблемы недетерминизма. Показано, что недетерминизм представляет собой полезный механизм достижения абстрактности, так как он естественным образом возникает из желания не принимать во внимание не интересующие нас аспекты поведения системы. Кроме того, он позволяет сохранить своеобразную симметрию в определении операторов в математической теории. Методы доказательства для недетерминированных процессов несколько сложнее, чем для детерминированных, вследствие необходимости

показывать, что в результате любого недетерминированного выбора поведение процесса будет отвечать заданной спецификации. К счастью, существуют прекрасные способы избежать недетерминизма, и они широко используются в гл. 4 и 5. Отсюда следует, что изучение гл. 3 можно отложить непосредственно до гл. 6, где знакомство с недетерминизмом становится необходимым.

В последнем разделе третьей главы дается исчерпывающее определение недетерминированного процесса. Оно представляет интерес для чистого математика, который хочет исследовать основы предмета или доказательно проверить справедливость алгебраических законов и других свойств процессов. Прикладные математики (включая программистов) могут воспринять эти законы как самоочевидные или оправданные их практической применимостью и в связи с этим спокойно пропустить наиболее теоретические разделы.

Только в гл. 4 мы, наконец, точно определяем, что такое взаимодействие: это особый способ взаимосвязи двух процессов, один из которых передает сообщение, а другой в то же время его принимает. Это взаимодействие синхронизовано. Если канал требует буферизации, это достигается вставкой буферного процесса между двумя процессами. Важной целью в проектировании параллельных систем является повышение скорости вычислений при решении практических задач. Это иллюстрируется на примере построения некоторых простых систолических (или итеративных) матричных алгоритмов. Простым примером является транспортер, представляющий собой последовательность процессов, в которой каждый процесс принимает сообщения только у своего предшественника и передает только своему преемнику. Транспортеры полезны при реализации протоколов однонаправленной связи, представленных в виде иерархии слоев. Наконец, важное понятие абстрактного типа данных моделируется с помощью подчиненного процесса, каждое вхождение которого взаимодействует только с тем блоком, в котором он описан.

В гл. 5 мы показываем, как совокупность обычных операторов последовательного программирования может быть взята за основу структуры взаимодействующих последовательных процессов. Опытному программисту, возможно, покажется удивительным, что эти операторы обладают такими же красивыми алгебраическими свойствами, что и операторы, знакомые ему из математики, и что способы доказательства соответствия программ своим спецификациям для последовательных и для параллельных программ во многом схожи. Даже внешние прерывания могут быть точно определены и полезно использованы с соблюдением элегантных законов,

Гл. 6 описывает структуру и способ построения системы, в которой ограниченное число физических ресурсов, таких, как диски и печатающие устройства, разделено между большим количеством процессов с переменной потребностью в этих ресурсах. Каждый ресурс представлен одним процессом. Как только в нем возникает потребность у некоторого процесса-пользователя, создается новый виртуальный ресурс. Виртуальный ресурс — это процесс, который ведет себя как подчиненный по отношению к процессу-пользователю, а кроме того, при необходимости взаимодействует с реальным ресурсом. Эти взаимодействия чередуются с взаимодействиями с другими одновременно активными виртуальными процессами. Таким образом, реальные и виртуальные процессы играют ту же роль, что и мониторы и конверты в языке PASCAL PLUS. Содержание главы иллюстрируется на примерах построения законченных, но очень простых операционных систем; это самые крупные примеры во всей книге.

В гл. 7 приводится ряд альтернативных подходов к понятиям параллелизма и взаимодействия; поясняются технические, исторические и личные мотивы, которые привели к появлению теории, изложенной в предыдущих главах. Здесь же я воздаю должное другим авторам и даю рекомендации по дальнейшему чтению в этой области.

Благодарности

Мне доставляет огромное удовольствие выразить признательность Робину Милнеру за его глубокую и творческую работу (Milner R. A Calculus of Communicating Systems), заложившую основы исчисления взаимодействующих систем. Его оригинальные идеи, дружеское отношение наряду с профессиональным соперничеством были для меня постоянными источниками вдохновения и поддержки в работе, завершившейся публикацией этой книги.

Последние двадцать лет я занимался проблемами программирования для параллельных вычислений и разработкой языка программирования для решения этих проблем. В течение этого периода мне было в высшей степени полезно сотрудничество со многими учеными, в том числе с П. Б. Хансеном, С. Бруксом, Д. Бастардом, Чжоу Чао Ченем, О.-И. Далом, Э. Дейкстрой, Д. Элдером, Д. Якобом, Й. Хейесом, Д. Кобишем, Д. Кеннауэйем, Т. И. Конгом, П. Лоэром, М. Маккигом, К. Морганом, Э.-Р. Олдерогом, Р. Райнеке, Б. Роско, А. Теруелом, О. Токером, Д. Уэлшем.

И, наконец, особую благодарность я приношу О.-И. Далу, Э. Дейкстре, Лесли М. Голдшлагеру, Д. Сандерсу и остальным, кто тщательно изучил предыдущие варианты этого текста и указал на ошибки и неточности. Это также относится к слушателям Уоллонгонской летней¹⁾ школы по теоретическому программированию (январь 1983 г.), участникам моего семинара в Школе аспирантов Академии наук Китая (апрель 1983 г.) и студентам кафедры вычислений Оксфордского университета выпуск 1979—1984 гг.

¹⁾ Наблюдательный читатель да не забудет о существовании Южного полушария. — *Прим. ред.*

Глава 1. Процессы

1.1. ВВЕДЕНИЕ

Забудем на время о компьютерах и программировании и вместо этого подумаем о вещах, которые нас окружают. Они действуют и вступают во взаимодействие с нами и друг с другом в соответствии со своими особенностями. Вспомним о часах, телефонах, настольных играх, торговых автоматах. Чтобы описывать поведение (работу) этих объектов, надо решить, какого рода события или действия нас интересуют, и выбрать для каждого из них подходящее название, или *имя*. В случае простого торгового автомата существуют два вида событий:

мон — опускание монеты в щель автомата,

шок — появление шоколадки из выдающего устройства.

В случае более сложного торгового автомата различных событий может быть больше:

n1 — опускание 1 пенни,

n2 — опускание монеты в 2 пенни,

мал — появление маленького коржика или булочки,

бол — появление большого коржика или булочки,

сd1 — появление 1 пенни сдачи.

Заметим, что имя каждого события обозначает целый *класс* событий; отдельные вхождения события внутри одного класса разделены во времени. Аналогичное различие между классом и вхождением можно проследить на примере буквы «а», многочисленные вхождения которой в текст этой книги разделены в пространстве.

Множество имен событий, выбранных для конкретного описания объекта, называется его *алфавитом*. Алфавит считается постоянным, заранее определенным свойством объекта. Участвовать в событии, выходящем за рамки его алфавита, для объекта логически невозможно — автомат по продаже шоколадок никогда не выдаст вам неожиданно игрушечный линкор. Обратное, впрочем, не обязательно. Машина, предназначенная для продажи шоколадок, может на самом деле этого не делать — возможно, потому что ее не загрузили, или

она сломалась, или же просто никто не хочет шоколада. Но если уж решено, что «шок» входит в алфавит машины, то это имя останется там, несмотря на то, что соответствующее событие фактически никогда не происходит.

Обычно выбор алфавита влечет за собой сознательное упрощение: не рассматриваются многие действия и свойства, представляющие меньший интерес. Например, не описываются цвет, вес и форма торгового автомата, равно как и такие весьма важные события в его жизни, как пополнение запаса шоколадок или разгрузка монетоприемника. Возможно, это делается из тех соображений, что эти детали не имеют (или не должны иметь) прямого отношения к пользователям автомата.

Считается, что конкретное событие в жизни объекта происходит мгновенно, т. е. является элементарным действием, не имеющим протяженности во времени. Протяженное, т. е. требующее времени, действие следует рассматривать как пару событий, первое из которых отмечает начало действия, а второе — его завершение. Продолжительность действия определяется интервалом между наступлением его события-начала и наступлением его события-завершения; в течение всего этого времени могут происходить другие события. Два протяженных действия могут перекрываться по времени, если начало каждого из них предшествует завершению другого.

Еще одно обстоятельство, от которого мы сознательно абстрагируемся, — это точная привязка событий ко времени. Преимуществом такого подхода является упрощение построений и рассуждений, которые к тому же можно применить к физическим и вычислительным системам любой скорости и производительности. В тех же случаях, когда временная привязка событий необходима по существу, соответствующее рассмотрение может быть проведено дополнительно к доказательству логической правильности построения. Независимость от времени всегда была решающим условием успешного применения языков программирования высокого уровня.

Следствием исключения времени является то, что мы отказываемся не только от ответов, то даже и от вопросов о том, происходит ли одно событие строго одновременно с другим. Когда совместность событий существенна (например, при синхронизации), мы отмечаем это, сводя их в одно событие, или же позволяем совместным событиям происходить в любом относительно друг друга порядке.

При выборе алфавита не обязательно делать различия между событиями, инициированными самим объектом (например, «шок»), или некоторым внешним по отношению

к объекту фактором (например, «мон»). Неупотребление понятия причинности ведет к значительному упрощению нашей теории и ее приложений.

Будем отныне употреблять слово *процесс* для обозначения поведения объекта постольку, поскольку оно может быть описано в терминах ограниченного набора событий, выбранного в качестве его алфавита. Введем следующие соглашения:

1. Имена событий будем обозначать словами, составленными из строчных букв, например:

мон, шок, n1, n2,

а также буквами *a, b, c, d, e.*

2. Имена конкретных процессов будем обозначать словами, составленными из прописных букв, например:

ТАП — простой торговый автомат,

ТАС — сложный торговый автомат,

а буквами *P, Q, R* (в законах) будем обозначать произвольные процессы.

3. Буквы *x, y, z* используются для переменных, обозначающих события.

4. Буквы *A, B, C* используются для обозначения множества событий.

5. Буквы *X, Y* используются для переменных, обозначающих процессы.

6. Алфавит процесса *P* обозначается αP , например:

$\alpha \text{ТАП} = \{\text{мон, шок}\}$

$\alpha \text{ТАС} = \{n1, n2, \text{мал, бол, cd1}\}$

Процесс с алфавитом *A*, такой, что в нем не происходит ни одно событие из *A*, назовем *СТОП_A*. Этот процесс описывает поведение сломанного объекта: имея физическую способность участвовать в событиях из *A*, тот никогда ее не использует. Мы различаем объекты с различными алфавитами, даже если эти объекты ничего не делают. Так, *СТОП_{αТАП}* мог бы выдать шоколадку, тогда как *СТОП_{αТАС}* никогда бы не выдал шоколадку, а только коржик. Покупатель знает это даже в том случае, когда ему неизвестно, что обе машины сломаны.

В заключение определим простую систему обозначений, которая поможет при описании объектов, уже обладающих способностью к некоторым действиям.

1.1.1. Префиксы

Пусть x — событие, а P — процесс. Тогда

$(x \rightarrow P)$ (читается как “ P за x ”)

описывает объект, который вначале участвует в событии x , а затем ведет себя в точности как P . Процесс $(x \rightarrow P)$ имеет по определению тот же алфавит, что и P ; поэтому это обозначение можно использовать только при условии, что x принадлежит тому же алфавиту. Более формально:

$$\alpha(x \rightarrow P) = \alpha P, \text{ если } x \in \alpha P.$$

Примеры

X1. Простой торговый автомат, который поглощает монету и ломается:

$$(мон \rightarrow СТОП_{атап})$$

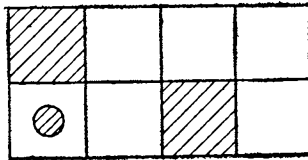
X2. Простой торговый автомат, который благополучно обслуживает двух покупателей и затем ломается:

$$(мон \rightarrow (шок \rightarrow (мон \rightarrow (шок \rightarrow СТОП_{атап}))))$$

В начальном состоянии автомат позволяет опустить монету, но не позволяет вынуть шоколадку. После же приема монеты щель автомата закрывается до тех пор, пока не будет вынута шоколадка. Таким образом, эта машина не позволяет ни опустить две монеты подряд, ни получить подряд две шоколадки.

В дальнейшем мы будем опускать скобки в случае линейной последовательности событий, как в примере **X2**, условившись, что операция \rightarrow ассоциативна справа.

X3. Фишка находится в левом нижнем углу поля и может двигаться только вверх или вправо на соседнюю белую клетку:



$$\alpha \text{ФИШ} = \{\text{вверх, вправо}\}$$

$$\begin{aligned} \text{ФИШ} = & (\text{вправо} \rightarrow \text{вверх} \rightarrow \text{вправо} \rightarrow \text{вправо} \\ & \rightarrow \text{СТОП}_{\alpha \text{Фиш}}) \end{aligned}$$

Заметим, что в записи с операцией \rightarrow справа всегда стоит процесс, а слева — отдельное событие. Если P и Q — процессы, то запись $P \rightarrow Q$ синтаксически неверна. Правильный способ описания процесса, который ведет себя сначала как P , а затем как Q , будет дан в гл. 4. Аналогично синтаксически неверной будет запись $x \rightarrow y$, где x и y — события. Такой процесс надо было бы записать как $x \rightarrow (y \rightarrow \text{СТОП})$. Таким образом, мы тщательно разделяем понятия «событие» и «процесс», состоящий из событий — может быть, из многих, а может быть, ни из одного.

1.1.2. Рекурсия

Префиксную запись можно использовать для полного описания поведения процесса, который рано или поздно останавливается. Но было бы чрезвычайно утомительно полностью выписывать поведение торгового автомата, рассчитанного на долгую службу; поэтому нам необходим способ более сжатого описания повторяющихся действий. Было бы желательно, чтобы этот способ не требовал знать заранее срок жизни объекта; это позволило бы нам описывать объекты, которые продолжали бы действовать и взаимодействовать с их окружением до тех пор, пока в них есть необходимость.

Рассмотрим простейший долговечный объект — часы, у которых только одна забота — тикать (их заводом мы сознательно пренебрегаем):

$$\alpha \text{ЧАСЫ} = \{\text{тик}\}$$

Теперь рассмотрим объект, который вначале издает единственный «тик», а затем ведет себя в точности как ЧАСЫ

$$(\text{тик} \rightarrow \text{ЧАСЫ})$$

Поведение этого объекта неотличимо от поведения исходных часов. Следовательно, один и тот же процесс описывает поведение обоих объектов. Эти рассуждения позволяют сформулировать равенство

$$\text{ЧАСЫ} = (\text{тик} \rightarrow \text{ЧАСЫ})$$

Его можно рассматривать как неявное задание поведения часов, точно так же как корень квадратный из двух может быть найден как положительное решение для x в уравнении

$$x = x^2 + x - 2.$$

Уравнение для часов имеет некоторые очевидные следствия, которые получаются простой заменой членов на равные:

$$\begin{aligned} ЧАСЫ &= (тик \rightarrow ЧАСЫ) && \text{исходное уравнение} \\ &= (тик \rightarrow (тик \rightarrow ЧАСЫ)) && \text{подстановкой} \\ &= (тик \rightarrow тик \rightarrow тик \rightarrow ЧАСЫ) && \text{аналогично} \end{aligned}$$

Это уравнение можно разворачивать столько раз, сколько нужно, при этом возможность для дальнейшего разворачивания сохраняется. Мы эффективно описали потенциально бесконечное поведение объекта ЧАСЫ как

$$тик \rightarrow тик \rightarrow тик \rightarrow \dots$$

точно так же, как корень квадратный из двух можно представить в виде предела последовательности десятичных дробей 1.414

Такой метод «самоназывания», или рекурсивное определение процесса, будет правильно работать, только если в правой части уравнения рекурсивному вхождению имени процесса предшествует хотя бы одно событие. Например, рекурсивное «определение» $X = X$ не определяет ничего, так как решением этого уравнения может служить все что угодно. Описание процесса, начинающееся с префикса, называется *предваренным*. Если $F(X)$ — предваренное выражение, содержащее имя процесса X , а A — алфавит X , то мы утверждаем, что уравнение $X = F(X)$ имеет единственное решение в алфавите A . Иногда удобнее обозначать это решение выражением $\mu X : A.F(X)$. Здесь X является локальным именем (связанной переменной) и может произвольно изменяться, поскольку

$$\mu X : A.F(X) = \mu Y : A.F(Y).$$

Справедливость этого равенства следует из того, что решение для X уравнения $X = F(X)$ является также решением для Y уравнения $Y = F(Y)$.

В дальнейшем мы будем задавать рекурсивные определения процессов либо уравнениями, либо с помощью μ -оператора в зависимости от того, что удобнее. В записи $\mu X : A.F(X)$ мы часто будем опускать явное упоминание алфавита A , если это понятно из контекста или из содержания процесса.

Примеры

X1. Вечные часы

$$ЧАСЫ = \mu X : \{тик\}. (тик \rightarrow X)$$

X2. Наконец-то простой торговый автомат, полностью удовлетворяющий спрос на шоколадки:

$$ТАП = (мон \rightarrow (шок \rightarrow ТАП))$$

Как уже пояснялось, это уравнение является лишь альтернативой более формального определения

$$ТАП = \mu X : \{мон, шок\}. (мон \rightarrow (шок \rightarrow X))$$

X3. Автомат по размеру монеты в 5 пенни:

$$\alpha PA3M5A = \{n5, cd2, cd1\}$$

$$PA3M5A = (n5 \rightarrow cd2 \rightarrow cd1 \rightarrow cd2 \rightarrow PA3M5A)$$

X4. Другой автомат по размену монет с тем же алфавитом:

$$PA3M5B = (n5 \rightarrow cd1 \rightarrow cd1 \rightarrow cd1 \rightarrow cd2 \rightarrow PA3M5B)$$

Утверждение о том, что предваренное уравнение имеет решение, и это решение единственное, можно неформально доказать методом подстановки. Всякий раз, когда в правую часть уравнения производится подстановка на место каждого вхождения имени процесса, выражение, определяющее поведение процесса, становится длиннее, а значит, описывает больший начальный отрезок этого поведения. Таким путем можно определить любой конечный отрезок поведения процесса. А так как два объекта, ведущие себя одинаково вплоть до любого момента времени, ведут себя одинаково всегда, то они представляют собой один и тот же процесс. Те, кто считает, что такие рассуждения непонятны или неубедительны, могут принять это утверждение как аксиому; со временем его важность и уместность станут очевидными. Более строгое доказательство невозможно без точного математического определения процесса. Это будет сделано в разд. 2.8.3. Приведенное здесь описание рекурсии существенно опирается на понятие предваренности рекурсивного уравнения. Смысл не-предваренной рекурсии мы обсудим в разд. 3.8.

1.1.3. Выбор

Используя префиксы и рекурсию, можно описывать объекты, обладающие только одной возможной линией поведения. Однако поведение многих объектов зависит от окружающей их обстановки. Например, торговый автомат может иметь различные щели для 1- и 2-пенсовых монет; выбор одного из двух событий в этом случае предоставлен покупателю.

Если x и y — различные события, то $(x \rightarrow P | y \rightarrow Q)$ описывает объект, который сначала участвует в одном из собы-

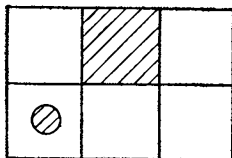
тий x, y . Последующее же поведение объекта описывается процессом P , если первым произошло событие x , или Q , если первым произошло событие y . Поскольку x и y — различные события, то выбор между P и Q определяется тем, какое из событий в действительности наступило первым. Требование неизменности алфавитов по-прежнему остается в силе, т. е.

$$\alpha(x \rightarrow P \mid y \rightarrow Q) = \alpha P$$

если $\{x, y\} \subseteq \alpha P$ и $\alpha P = \alpha Q$. Вертикальную черту $|$ следует читать как “или”: “ P за x или Q за y ”.

Примеры

X1. Возможные перемещения фишки на поле описываются процессом



(*вверх* \rightarrow *СТОП* $|$ *вправо* \rightarrow *вправо* \rightarrow *вверх* \rightarrow *СТОП*)

X2. Автомат с двумя вариантами размена монеты в 5 пенни (ср. с примерами 1.1.2 **X3** и **X4**, где выбора нет):

$$\begin{aligned} \text{РАЗМ5С} = n5 \rightarrow & (cd1 \rightarrow cd1 \rightarrow cd1 \rightarrow cd2 \rightarrow \text{РАЗМ5С} \\ & | cd2 \rightarrow cd1 \rightarrow cd1 \rightarrow cd2 \rightarrow \text{РАЗМ5С}) \end{aligned}$$

Выбор осуществляется покупателем.

X3. Автомат, предлагающий на выбор шоколадку или ириску:

$$\text{ТАШИ} = \mu X. \text{мон} \rightarrow (\text{шок} \rightarrow X \mid \text{ирис} \rightarrow X)$$

X4. Более сложный автомат, предлагающий различные товары различной стоимости и дающий сдачу:

$$\begin{aligned} \text{ТАС} = & (n2 \rightarrow (\text{бол} \rightarrow \text{ТАС} \\ & | \text{мал} \rightarrow cd1 \rightarrow \text{ТАС}) \\ & | n1 \rightarrow (\text{мал} \rightarrow \text{ТАС} \\ & | n1 \rightarrow (\text{бол} \rightarrow \text{ТАС} \\ & | n1 \rightarrow \text{СТОП}))) \end{aligned}$$

Как и многие сложные механизмы, этот автомат имеет в своей конструкции изъян. Часто бывает проще изменить инструкцию для пользователя, чем вносить исправления в готовую разработку, и поэтому мы снабдим наш автомат пре-

дупреждением:

«ВНИМАНИЕ: не опускайте три монеты подряд!»

X5. Автомат, доверяющий покупателю сначала попробовать шоколад, а заплатить потом. Нормальная последовательность событий тоже допускается:

$$\begin{aligned} \text{ТАДОВЕР} = \mu X. (& \text{мон} \rightarrow \text{шок} \rightarrow X \\ & | \text{шок} \rightarrow \text{мон} \rightarrow X) \end{aligned}$$

X6. Чтобы не возникало убытков, за право пользоваться *ТАДОВЕР* взимается предварительная плата

$$\text{ТАП2} = (\text{мон} \rightarrow \text{ТАДОВЕР})$$

Эта машина позволяет опустить последовательно до двух монет, после чего последовательно получить до двух шоколадок; она не выдает ни на шоколадку больше заранее оплаченного количества.

X7. Процесс копирования состоит из следующих событий:

- вв.0* — считывание нуля из входного канала,
- вв.1* — считывание единицы из входного канала,
- выв.0* — запись нуля в выходной канал,
- выв.1* — запись единицы в выходной канал.

Поведение процесса состоит из повторяющихся пар событий. На каждом такте он считывает, а затем записывает один бит.

$$\begin{aligned} \text{КОПИБИТ} = \mu X. (& \text{вв.0} \rightarrow \text{выв.0} \rightarrow X \\ & | \text{вв.1} \rightarrow \text{выв.1} \rightarrow X) \end{aligned}$$

Заметим, что этот процесс предоставляет своему окружению выбор того, какое значение следует ввести, но сам решает, какое значение вывести. В этом будет состоять основное различие между считыванием и записью в нашей трактовке взаимодействия в гл. 4.

Определение выбора легко обобщить на случай более чем двух альтернатив, например:

$$(x \rightarrow P | y \rightarrow Q | \dots | z \rightarrow R)$$

Заметим, что символ выбора $|$ не является операцией над процессами; для процессов P и Q запись $P|Q$ будет синтаксически неверной. Мы вводим это правило, чтобы избежать необходимости выяснять смысл записи $(x \rightarrow P) | (x \rightarrow Q)$, в которой безуспешно предлагается выбор первого события. Эта проблема решается в разд. 3.3 ценой введения недетерминизма. Между тем, если x, y, z — различные события, запись

$$(x \rightarrow P | y \rightarrow Q | z \rightarrow R)$$

следует рассматривать как один оператор с тремя аргументами P, Q, R . При этом она не будет эквивалентна записи

$$(x \rightarrow P \mid (y \rightarrow Q \mid z \rightarrow R))$$

которая синтаксически неверна.

В общем случае если B — некоторое множество событий, а $P(x)$ — выражение, определяющее процесс для всех различных x из B , то запись

$$(x: B \rightarrow P(x))$$

определяет процесс, который сначала предлагает на выбор любое событие y из B , а затем ведет себя как $P(y)$. Запись следует читать как « P от x за x из B ». В этой конструкции x играет роль локальной переменной, и поэтому

$$(x: B \rightarrow P(x)) = (y: B \rightarrow P(y))$$

Множество B определяет начальное меню процесса, потому что в нем предлагается набор действий, из которого осуществляется первоначальный выбор.

Пример

Х8. Процесс, который в каждый момент времени может участвовать в любом событии из своего алфавита A :

$$\alpha \text{ИСП}_A = A$$

$$\text{ИСП}_A = (x: A \rightarrow \text{ИСП}_A)$$

В особом случае, когда в начальном меню содержится лишь одно событие e ,

$$(x: \{e\} \rightarrow P(x)) = (e \rightarrow P(e))$$

потому что e является единственным возможным начальным событием. В еще более специальном случае, когда начальное меню пусто, вообще ничего не происходит, и поэтому

$$(x: \{ \} \rightarrow P(x)) = (y: \{ \} \rightarrow Q(y)) = \text{СТОП}$$

Двуместный оператор выбора \mid можно определить и в более общих обозначениях:

$$(a \rightarrow P \mid b \rightarrow Q) = (x: B \rightarrow R(x))$$

где $B = \{a, b\}$, а $R(x) = \text{if } x = a \text{ then } P \text{ else } Q$.

Аналогичным образом можно выразить выбор из трех и более альтернатив. Итак, мы сумели сформулировать выбор, префиксы и **СТОП** как частные случаи записи обобщенного опера-

тора выбора. Это нам очень пригодится в разд. 1.3, где будут сформулированы общие законы, которым подчиняются процессы, и в разд. 1.4, посвященном реализации процессов.

1.1.4. Взаимная рекурсия

Рекурсия позволяет определить единственный процесс как решение некоторого единственного уравнения. Эта техника легко обобщается на случай решения систем уравнений с более чем одним неизвестным. Для достижения желаемого результата необходимо, чтобы правые части всех уравнений были предваренными, а каждый неизвестный процесс входил ровно один раз в правую часть одного из уравнений.

Пример

X1. Автомат с газированной водой имеет рукоятку с двумя возможными положениями — *ЛИМОН* и *АПЕЛЬСИН*. Действия по установке рукоятки в соответствующее положение назовем *устлимон* и *устапельсин*, а действия автомата по наливаю напитка — *лимон* и *апельсин*. Вначале рукоятка занимает некоторое нейтральное положение, к которому затем уже не возвращается. Ниже приводятся уравнения, определяющие алфавит и поведение трех процессов:

$$\alpha A G A Z = \alpha G = \alpha W = \{устлимон, устапельсин, мон, лимон, апельсин\}$$

$$A G A Z = (устлимон \rightarrow G \mid устапельсин \rightarrow W)$$

$$G = (мон \rightarrow лимон \rightarrow G \mid устапельсин \rightarrow W)$$

$$W = (мон \rightarrow апельсин \rightarrow W \mid устлимон \rightarrow G)$$

Неформально, после того как произошло первое событие, поведение автомата описывается одним из двух состояний *G* и *W*. В каждом из этих состояний автомат либо наливает соответствующий напиток, либо может быть переключен в другое состояние.

Использование переменных с индексами позволяет описывать бесконечные системы уравнений.

Пример

X2. Объект начинает движение с земли и может двигаться *вверх*. После этого в любой момент времени он может сдвинуться на один шаг *вверх* или *вниз*, за исключением того, что, находясь на земле, он больше не может двигаться вниз. Однако, находясь на земле, объект может совершать движение *вокруг*. Пусть n — некоторое число из натурального ряда $\{0, 1, 2, \dots\}$. Для каждого такого n введем пронумерованное

имя CT_n , с помощью которого будем описывать поведение объекта, находящегося в n шагах от земли. Начальное поведение определяется уравнением

$$CT_0 = (\text{вверх} \rightarrow CT_1 \mid \text{вокруг} \rightarrow CT_0)$$

а остальные уравнения образуют бесконечную систему

$$CT_{n+1} = (\text{вверх} \rightarrow CT_{n+2} \mid \text{вниз} \rightarrow CT_n)$$

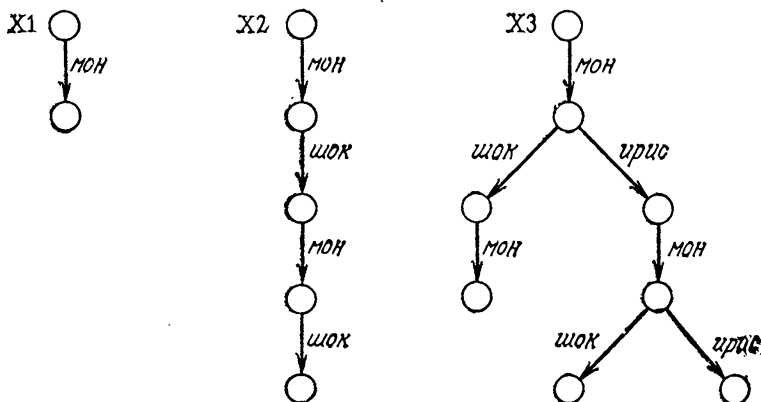
где n пробегает натуральный ряд $0, 1, 2, \dots$.

Содержательность обычного индуктивного определения основана на том, что индексы, используемые в правой части каждого уравнения, меньше, чем индексы левой части. Здесь же CT_{n+1} определяется в терминах CT_{n+2} , и поэтому нашу систему можно рассматривать только как бесконечный набор взаимно рекурсивных определений, содержательность которых вытекает из того факта, что правая часть каждого уравнения является предваренной.

1.2. РИСУНКИ

Иногда процесс полезно представить графически в виде древовидной структуры с помощью кружков-вершин, соединенных стрелками-дугами. Следуя традиционной терминологии машин, обладающих внутренними состояниями, вершины представляют состояния процесса, а дуги — переходы между ними. Корневая вершина дерева (изображаемая обычно вверху страницы) соответствует начальному состоянию; от нее процесс движется вниз по стрелкам. Каждая дуга помечена именем события, которое соответствует данному переходу. Все дуги, ведущие из одной вершины, должны иметь различные метки.

Примеры (1.1.1 X1, X2; 1.1.3 X3)



Чтобы нарисовать эту картину целиком, никогда не хватит места. Да и на рисунок для объекта, обладающего «всего» 65 536 состояниями, потребуется немало времени.

1.3. ЗАКОНЫ

Несмотря на ограниченность набора введенных нами обозначений, одно и то же поведение процесса можно описывать по-разному. Так, например, не имеет значения порядок, в котором записаны члены в операторе выбора:

$$(x \rightarrow P | y \rightarrow Q) = (y \rightarrow Q | x \rightarrow P)$$

С другой стороны, процесс, способный что-либо выполнять, и процесс, который не может делать ничего, — это, очевидно, не одно и то же: $(x \rightarrow P) \neq \text{СТОП}$. Чтобы правильно понимать и эффективно использовать обозначения, надо научиться узнавать, какие выражения определяют один и тот же объект, а какие — нет, точно так же как каждый, кто знаком с арифметикой, знает, что $(x + y)$ — это то же число, что и $(y + x)$. Тожественность процессов с одинаковыми алфавитами можно устанавливать с помощью алгебраических законов, очень похожих на законы арифметики.

Первый закон (L1 ниже) касается оператора выбора. Он гласит, что два процесса, определенные с помощью оператора выбора, различны, если на первом шаге они предлагают различные альтернативы или после одинакового первого шага ведут себя по-разному. Если же множества начального выбора оказываются равными и для каждой начальной альтернативы дальнейшее поведение процессов совпадает, то очевидно, что процессы тождественны.

$$\text{L1. } (x : A \rightarrow P(x)) = (y : B \rightarrow Q(y)) \equiv (A = B \ \& \ \forall x \in A. P(x) = Q(x))$$

Здесь и далее мы неявно предполагаем, что алфавиты процессов в обеих частях уравнения совпадают.

Закон L1 имеет ряд следствий:

$$\text{L1A. } \text{СТОП} \neq (d \rightarrow P)$$

Доказательство:

$$\begin{aligned} \text{ЛЧ} &= (x ; \{ \} \rightarrow P) && \text{по определению (конец 1.1.3)} \\ &\neq (x : \{d\} \rightarrow P) && \text{так как } \{ \} \neq \{d\} \\ &= \text{ПРЧ} && \text{по определению (конец 1.1.3)} \end{aligned}$$

$$\text{L1B. } (c \rightarrow P) \neq (d \rightarrow Q), \text{ если } c \neq d.$$

Доказательство: $\{c\} \neq \{d\}$

$$\text{L1C. } (c \rightarrow P | d \rightarrow Q) = (d \rightarrow Q | c \rightarrow P)$$

Доказательство:

Определим $R(x) = P$, если $x = c$,
 $= Q$, если $x = d$.

$ЛЧ = (x : \{c, d\} \rightarrow R(x))$ по определению
 $= (x : \{d, c\} \rightarrow R(x))$ так как $\{c, d\} = \{d, c\}$
 $= \text{ПРЧ}$ по определению

L1D. $(c \rightarrow P) = (c \rightarrow Q) \equiv P = Q$

Доказательство: $\{c\} = \{c\}$

С помощью этих законов можно доказывать простые теоремы.

Примеры

X1. $(\text{мон} \rightarrow \text{шок} \rightarrow \text{мон} \rightarrow \text{шок} \rightarrow \text{СТОП}) \neq (\text{мон} \rightarrow \text{СТОП})$

Доказательство: следует из **L1D** и **L1A**.

X2. $\mu X. (\text{мон} \rightarrow (\text{шок} \rightarrow X \mid \text{ирис} \rightarrow X))$
 $= \mu X. (\text{мон} \rightarrow (\text{ирис} \rightarrow X \mid \text{шок} \rightarrow X))$

Доказательство: следует из **L1C**.

Для доказательства более общих теорем о рекурсивно определенных процессах необходимо ввести закон, гласящий, что всякое должным образом предваренное рекурсивное уравнение имеет единственное решение.

L2. Если $F(X)$ — предваренное выражение, то $(Y = F(Y)) \equiv (Y = \mu X. F(X))$.

Как прямое, но важное следствие получаем, что $\mu X. F(X)$ в действительности является решением соответствующего уравнения:

L2A. $\mu X. F(X) = F(\mu X. F(X))$

Пример

X3. Пусть $TA1 = (\text{мон} \rightarrow TA2)$, а $TA2 = (\text{шок} \rightarrow TA1)$. Требуется доказать, что $TA1 = \text{ТАП}$.

Доказательство:

$TA1 = (\text{мон} \rightarrow TA2)$ по определению $TA1$
 $= (\text{мон} \rightarrow (\text{шок} \rightarrow TA1))$ по определению $TA2$

Таким образом, $TA1$ является решением того же рекурсивного уравнения, что и ТАП . Так как это уравнение предваренное, оно имеет единственное решение. Значит, $TA1$ и

ТАП — это просто разные имена одного и того же процесса.

Эта теорема может казаться настолько очевидной, что ее формальное доказательство ничего не прибавит к ее убедительности. Единственная цель этого доказательства — показать на примере, что наши законы достаточно мощны для установления фактов подобного рода. В частности, при доказательстве очевидных фактов с помощью менее очевидных законов важно проследить справедливость каждого шага доказательства, чтобы избежать порочного круга.

Закон **L2** можно распространить на случай взаимной рекурсии. Систему рекурсивных уравнений можно записать в общем виде, используя индексы:

$$X_i = F(i, X) \text{ для всех } i \text{ из } S,$$

где S — множество индексов, взаимно однозначно соответствующих множеству уравнений, X — массив процессов с индексами из S , а $F(i, X)$ — предваренное выражение.

Закон **L3** гласит, что при соблюдении этих условий существует единственный массив X , элементы которого удовлетворяют всем уравнениям.

L3. При соблюдении описанных выше условий
если $(\forall i : S. (X_i = F(i, X) \ \& \ Y_i = F(i, Y)))$, то $X = Y$.

1.4. РЕАЛИЗАЦИЯ ПРОЦЕССОВ

Любой процесс P , записанный с помощью уже введенных обозначений, может быть представлен в виде

$$(x : B \rightarrow F(x))$$

где F — функция, ставящая в соответствие множеству символов множество процессов, множество B может быть пустым (в случае **СТОП**), может содержать только один элемент (в случае префикса) или более одного элемента (в случае выбора). Если процесс задан рекурсивным уравнением, то мы требуем, чтобы рекурсия была предваренной и, значит, решение уравнения могло быть записано как

$$\mu X. (x : B \rightarrow F(x, X))$$

Используя закон **L2A**, можно привести эту запись к требуемому виду

$$(x : B \rightarrow F(x, \mu X. (x : B \rightarrow F(x, X))))$$

Таким образом, каждый процесс можно рассматривать как функцию F с областью определения B , выделяющей множе-

ство событий, которые могут служить начальными событиями процесса. Если первым произошло событие x из B , то $F(x)$ определяет дальнейшее поведение процесса.

Такой подход позволяет представить любой процесс как функцию в некотором подходящем функциональном языке программирования, например в ЛИСПе. Каждое событие из алфавита процесса представлено атомом, например "МОН, "ИРИС. Процесс — это функция которая может использовать эти символы в качестве аргументов. При этом если символ не может быть начальным событием процесса, то результатом функции будет специальный символ "BLEEP, который используется только для этой цели. Например, так как в процессе СТОП не происходит никаких событий, то "BLEEP — единственный возможный результат соответствующей функции, и можно определить

$$\text{СТОП} = \lambda x. \text{"BLEEP}$$

Если же фактический аргумент является событием, возможным для процесса, результатом функции будет другая функция, определяющая последующее поведение процесса. Так, процесс (мон \rightarrow СТОП) можно представить функцией

$$\lambda x. \text{if } x = \text{"МОН then СТОП} \\ \text{else "BLEEP}$$

В последнем примере мы используем возможность ЛИСПа получать в качестве результата функции снова функцию (в нашем случае СТОП). ЛИСП также позволяет задавать функцию в качестве аргумента — средство, которое мы используем для представления общей операции префиксации ($c \rightarrow P$):

$$\text{префикс}(c, P) = \lambda x. \text{if } x = c \text{ then } P \\ \text{else "BLEEP}$$

Для представления общего двуместного выбора ($c \rightarrow P \mid d \rightarrow Q$) нам потребуется функция с четырьмя параметрами:

$$\text{выбор2}(c, P, d, Q) = \lambda x. \text{if } x = c \text{ then } P \\ \text{else if } x = d \text{ then } Q \\ \text{else "BLEEP}$$

Для представления рекурсивно определенных процессов можно использовать функцию ЛИСПа LABEL. Например, простой торговый автомат ($\mu X. \text{мон} \rightarrow \text{шок} \rightarrow X$) определяется функцией

$$\text{LABEL } X. \text{префикс("МОН, префикс("ШОК, X))}$$

С помощью *LABEL* можно описать и взаимную рекурсию. Например, *CT* из примера 1.1.4 **X2** можно рассматривать как функцию, ставящую в соответствие множеству натуральных чисел множество процессов (которые сами являются функциями — но нас это не должно тревожить). Поэтому можно определить *CT* как

```
CT = LABEL X.λn.
  if n ≠ 0 then выбор2("ВОКРУГ,X(0),"ВВЕРХ,X(1))
  else выбор2("ВВЕРХ,X(n + 1),
              "ВНИЗ,X(n - 1))
```

Объект, стартующий с земли, описывается процессом *CT*(0).

Если *P* — функция, описывающая процесс, а *A* — список символов, составляющих его алфавит, то *ЛИСП*-функция *меню*(*A*, *P*) выдает список всех символов из *A*, которые могут обозначать первое событие в жизни *P*:

```
меню(A,P) = if A = NIL then NIL
             else if P(car(A)) = "BLEEP then меню (cdr(A), P)
             else cons(car(A), меню (cdr(A),P))
```

Если *x* принадлежит *меню*(*A*, *P*), то значение функции *P*(*x*) отлично от "BLEEP и, значит, определяет дальнейшее поведение процесса *P* после выполнения события *x*. Поэтому если *y* принадлежит *меню*(*A*, *P*(*x*)), то *P*(*x*)(*y*) описывает последующее поведение *P* после того, как произошли *x* и *y*. Это наблюдение подсказывает удобный способ исследования поведения процесса. Напишите программу, которая выводит на экран содержимое *меню*(*A*, *P*), а затем воспринимает символ, вводимый с клавиатуры. Если символ отсутствует в меню, программа реагирует на это слышимым сигналом «bleep», а символ в дальнейшем игнорирует. В противном случае символ воспринимается, а процесс повторяется, но *P* в нем заменяется на результат применения *P* к указанному символу. Процесс завершается появлением на экране символа "END. Таким образом, если *k* — последовательность вводимых с клавиатуры символов, то последовательность выводимых на экран ответов описывается функцией

```
взаимодействие(A,P,k) = cons (меню(A,P),
                               if car(k) = "END then NIL
                               else if P(car(k)) = "BLEEP then
                                   cons("BLEEP,
                                       взаимодействие(A,P,cdr(k)))
                               else взаимодействие(A,P(car(k)),cdr(k))
```

Обозначения, которые мы использовали для описания ЛИСП-функций, очень нестрогие и нуждаются в трансляции в специальные общепринятые *S*-выражения какой-либо конкретной реализации ЛИСПа. В ЛИСПките, например, функция *префикс* запишется как

```
(префикс
  lambda
    (a p)
    (lambda (x) (if(eq x a) p (quote BLEEP))))
```

К счастью, мы будем пользоваться только очень небольшим подмножеством чисто функционального ЛИСПа, что сведет к минимуму трудности при трансляции и исполнении наших процессов на множестве диалектов языка на различных машинах.

Если в вашем распоряжении несколько версий ЛИСПа, выберите ту, где должным образом представлено статическое связывание переменных. Наличие в ЛИСПе отложенных вычислений также предпочтительно, поскольку оно делает возможным прямое кодирование рекурсивных уравнений без помощи механизма LABEL; так, например,

ТАП = *префикс*("МОН, *префикс*("ШОК, *ТАП*))

Если ввод и вывод реализованы с помощью отложенных вычислений, функцию *взаимодействие* можно вызвать, подставив в качестве третьего параметра символы, вводимые с клавиатуры, и получить на экране меню для процесса *P*. Так, отбирая и вводя символы следующих друг за другом меню, пользователь имеет возможность интерактивно исследовать поведение процесса *P*. В других версиях ЛИСПа для достижения этой цели функцию *взаимодействие* необходимо переписать, явным образом используя ввод и вывод. После того как это сделано, можно наблюдать машинное исполнение процесса, представленного ЛИСП-функцией. В этом смысле функцию ЛИСПа можно рассматривать как *реализацию* соответствующего процесса. Более того, такую ЛИСП-функцию, как *префикс*, обрабатывающую представленные таким образом процессы, можно рассматривать как реализацию соответствующей операции над процессами.

1.5. ПРОТОКОЛЫ

Протоколом поведения процесса называется конечная последовательность символов, фиксирующая события, в которых процесс участвовал до некоторого момента времени. Представьте себе наблюдателя с блокнотом, который следит за

процессом и записывает имя каждого происходящего события. У нас есть все основания не учитывать возможности одновременного наступления двух событий; даже если это случится, наблюдателю все равно придется записать их одно за другим, и порядок, в котором записаны события, не будет иметь значения.

Мы будем обозначать протокол последовательностью символов, разделенной запятыми и заключенной в угловые скобки:

$\langle x, y \rangle$ состоит из двух событий — x и следующего за ним y .

$\langle x \rangle$ состоит из единственного события x .

$\langle \rangle$ пустая последовательность, не содержащая событий.

Примеры

X1. Протокол простого торгового автомата *ТАП* (1.1.2 **X2**) в момент завершения обслуживания первых двух покупателей:

$\langle \text{мон}, \text{шок}, \text{мон}, \text{шок} \rangle$

X2. Протокол того же автомата перед тем, как второй покупатель вынул свою шоколадку:

$\langle \text{мон}, \text{шок}, \text{мон} \rangle$

Понятие завершенной сделки недоступно ни процессу, ни его наблюдателю. Голод ожидающего клиента и готовность автомата его удовлетворить не входит в алфавит процесса и не могут быть зафиксированы и занесены в протокол.

X3. Перед началом работы процесса блокнот наблюдателя пуст. Это изображается пустым протоколом $\langle \rangle$ — самым коротким из всех возможных протоколов любого процесса.

X4. Сложный торговый автомат *ТАС* (1.1.3. **X4**) имеет семь различных протоколов длины два и менее:

$\langle \rangle$

$\langle n2 \rangle$ $\langle n1 \rangle$

$\langle n2, \text{бол} \rangle$ $\langle n2, \text{мал} \rangle$ $\langle n1, n1 \rangle$ $\langle n1, \text{мал} \rangle$

Из четырех протоколов длины два для данного устройства фактически может произойти только один. Его выбор по своему желанию осуществляет первый покупатель.

X5. Протокол *ТАС*, если первый покупатель нарушил инструкцию: $\langle n1, n1, n1 \rangle$. Сам протокол не фиксирует поломку автомата. На нее указывает лишь тот факт, что среди всех возможных протоколов этой машины не существует такого,

который являлся бы продолжением данного, т. е. не существует такого x , что $\langle n1, n1, n1, x \rangle$ является возможным протоколом ТАС. Покупатель может волноваться, негодовать; наблюдатель — нетерпеливо ожидать, держа карандаш наготове, но отныне не произойдет ни одно событие, и ни один символ не будет записан в блокнот. Окончательное право распорядиться судьбой покупателя и машины не входит в выбранный нами алфавит.

1.6. ОПЕРАЦИИ НАД ПРОТОКОЛАМИ

Протоколам принадлежит основная роль в фиксировании, описании и понимании поведения процессов. В этом разделе мы исследуем некоторые общие свойства протоколов и операций над ними. Введем следующие соглашения:

s, t, u обозначают протоколы,
 S, T, U обозначают множества протоколов,
 f, g, h обозначают функции.

1.6.1. Конкатенация

Наиболее важной операцией над протоколами является конкатенация, которая строит новый протокол из пары операндов s и t , просто соединяя их в указанном порядке; результат будем обозначать $s \hat{ } t$.

Например,

$$\begin{aligned} \langle \text{мон,шок} \rangle \hat{ } \langle \text{мон,улис} \rangle &= \langle \text{мон,шок,мон,улис} \rangle \\ \langle n1 \rangle \hat{ } \langle n1 \rangle &= \langle n1, n1 \rangle \\ \langle n1, n1 \rangle \hat{ } \langle \rangle &= \langle n1, n1 \rangle \end{aligned}$$

Самые важные свойства конкатенации — это ее ассоциативность и то, что пустой протокол $\langle \rangle$ служит для нее единицей:

$$\begin{aligned} \text{L1. } s \hat{ } \langle \rangle &= \langle \rangle \hat{ } s = s \\ \text{L2. } s \hat{ } (t \hat{ } u) &= (s \hat{ } t) \hat{ } u \end{aligned}$$

Следующие законы и очевидны, и полезны:

$$\begin{aligned} \text{L3. } s \hat{ } t &= s \hat{ } u \equiv t = u \\ \text{L4. } s \hat{ } t &= u \hat{ } t \equiv s = u \\ \text{L5. } s \hat{ } t &= \langle \rangle \equiv s = \langle \rangle \& t = \langle \rangle \end{aligned}$$

Пусть f — функция, отображающая протоколы в протоколы. Мы говорим, что функция *строгая*, если она отображает пустой протокол в пустой протокол:

$$f(\langle \rangle) = \langle \rangle$$

Будем говорить, что функция *дистрибутивна*, если

$$f(s \hat{~} t) = f(s) \hat{~} f(t)$$

Все дистрибутивные функции являются строгими.

Если n — натуральное число, то t^n будет обозначать конкатенацию n копий протокола t . Ее легко определить индукцией по n :

$$\text{L6. } t^0 = \langle \rangle$$

$$\text{L7. } t^{n+1} = t \hat{~} t^n$$

Это определение дает нам два полезных закона; и еще два можно доказать как их следствия:

$$\text{L8. } t^{n+1} = t^n \hat{~} t$$

$$\text{L9. } (s \hat{~} t)^{n+1} = s \hat{~} (t \hat{~} s)^n \hat{~} t$$

1.6.2. Сужение

Выражение $(t \upharpoonright A)$ обозначает протокол t , суженный на множество символов A ; он строится из t отбрасыванием всех символов, не принадлежащих A . Например,

$$\begin{aligned} & \langle \text{вокруг, вверх, вниз, вокруг} \rangle \upharpoonright \{ \text{вверх, вниз} \} \\ &= \langle \text{вверх, вниз} \rangle \end{aligned}$$

Сужение дистрибутивно и поэтому строго.

$$\text{L1. } \langle \rangle \upharpoonright A = \langle \rangle$$

$$\text{L2. } (s \hat{~} t) \upharpoonright A = (s \upharpoonright A) \hat{~} (t \upharpoonright A)$$

Эффект сужения на одноэлементных последовательностях очевиден:

$$\text{L3. } \langle x \rangle \upharpoonright A = \langle x \rangle \quad \text{если } x \in A.$$

$$\text{L4. } \langle y \rangle \upharpoonright A = \langle \rangle \quad \text{если } y \notin A.$$

Дистрибутивная функция однозначно определяется своими результатами на одноэлементных последовательностях, поскольку ее значения на последовательностях большей длины могут быть вычислены конкатенацией ее результатов на отдельных элементах последовательности. Например, если $x \neq y$, то

$$\begin{aligned} \langle x, y, x \rangle \upharpoonright \{x\} &= (\langle x \rangle \hat{~} \langle y \rangle \hat{~} \langle x \rangle) \upharpoonright \{x\} \\ &= (\langle x \rangle \upharpoonright \{x\}) \hat{~} (\langle y \rangle \upharpoonright \{x\}) \hat{~} (\langle x \rangle \upharpoonright \{x\}) \quad \text{по L2} \\ &= \langle x \rangle \hat{~} \langle \rangle \hat{~} \langle x \rangle \quad \text{по L3, L4} \\ &= \langle x, x \rangle \end{aligned}$$

Приведенные ниже законы раскрывают взаимосвязь сужения и операций над множествами. От протокола, суженного на пустое множество, не остается ничего, а последовательное сужение на два множества равнозначно одному сужению на пересечение этих множеств. Эти законы можно доказать индукцией по длине s .

$$\text{L5. } s \vdash \{ \} = \langle \rangle$$

$$\text{L6. } (s \vdash A) \vdash B = s \vdash (A \cap B)$$

1.6.3. Голова и хвост

Если s — непустая последовательность, обозначим ее первый элемент s_0 , а результат, полученный после его удаления, — s' . Например:

$$\begin{aligned} \langle x, y, x \rangle_0 &= x \\ \langle x, y, x \rangle &= \langle y, x \rangle \end{aligned}$$

Обе эти операции не определены для пустой последовательности.

$$\text{L1. } (\langle x \rangle^{\wedge} s)_0 = x$$

$$\text{L2. } (\langle x \rangle^{\wedge} s)' = s$$

$$\text{L3. } s = (\langle s_0 \rangle^{\wedge} s') \text{ если } s \neq \langle \rangle.$$

Следующий закон предоставляет удобный метод доказательства равенства или неравенства двух протоколов:

$$\text{L4. } s = t \equiv (s = t = \langle \rangle) \vee (s_0 = t_0 \ \& \ s' = t')$$

1.6.4. Звездочка

Множество A^* — это набор всех конечных протоколов (включая $\langle \rangle$), составленных из элементов множества A . После сужения на A такие протоколы остаются неизменными. Отсюда следует простое определение:

$$A^* = \{s \mid s \vdash A = s\}$$

Приведенные ниже законы являются следствиями этого определения:

$$\text{L1. } \langle \rangle \in A^*$$

$$\text{L2. } \langle x \rangle \in A^* \equiv x \in A$$

$$\text{L3. } (s^{\wedge} t) \in A^* \equiv s \in A^* \ \& \ t \in A^*$$

Они обладают достаточной мощностью, чтобы определить, принадлежит ли протокол множеству A^* . Например, если

$x \in A$, а $y \in A$, то

$$\begin{aligned} \langle x, y \rangle \in A^* &= (\langle x \rangle \wedge \langle y \rangle) \in A^* \\ &= (\langle x \rangle \in A^* \& (\langle y \rangle \in A^*)) \quad \text{по L3} \\ &= \text{истина} \& \text{ложь} \quad \text{по L2} \\ &= \text{ложь} \end{aligned}$$

Еще один полезный закон может служить рекурсивным определением A^* :

$$\text{L4. } A^* = \{t \mid t = \langle \rangle \vee (t_0 \in A \& t' \in A^*)\}$$

1.6.5. Порядок

Если s — копия некоторого начального отрезка t , то можно найти такое продолжение u последовательности s , что $s \hat{=} u = t$. Поэтому мы определим отношение порядка

$$s \leq t = (\exists u. s \hat{=} u = t)$$

и будем говорить, что s является *префиксом* t . Например:

$$\begin{aligned} \langle x, y \rangle &\leq \langle x, y, x, w \rangle \\ \langle x, y \rangle &\leq \langle z, y, x \rangle \equiv x = z \end{aligned}$$

Отношение \leq является частичным упорядочением и имеет своим наименьшим элементом $\langle \rangle$. Об этом говорят законы **L1—L4**:

L1. $\langle \rangle \leq s$	наименьший элемент
L2. $s \leq s$	рефлексивность
L3. $s \leq t \& t \leq s \Rightarrow s = t$	антисимметричность
L4. $s \leq t \& t \leq u \Rightarrow s \leq u$	транзитивность

Следующий закон вместе с **L1** позволяет определить, является ли справедливым отношение $s \leq t$:

$$\text{L5. } (\langle x \rangle \wedge s) \leq t \equiv t \neq \langle \rangle \& x = t_0 \& s \leq t'$$

Множество префиксов данной последовательности вполне упорядочено:

$$\text{L6. } s \leq u \& t \leq u \Rightarrow s \leq t \vee t \leq s$$

Если s является подпоследовательностью t (не обязательно начальной), мы говорим, что s находится **в** t ; это можно определить так:

$$s \text{ в } t = (\exists u, v. t = u \hat{=} s \hat{=} v)$$

Это отношение также задает частичный порядок в том смысле, что оно удовлетворяет законам **L1—L4**. Кроме того,

для него справедливо следующее утверждение:

$$\text{L7. } (\langle x \rangle^{\wedge} s) \text{ в } t \equiv t \neq \langle \rangle \& ((t_0 = x \& s \leq t') \vee (\langle x \rangle^{\wedge} s) \text{ в } t')$$

Будем говорить, что функция f из множества протоколов во множество протоколов *монотонна*, если она сохраняет отношение порядка \leq , т. е.

$$f(s) \leq f(t) \quad \text{всякий раз, когда } s \leq t$$

Все дистрибутивные функции монотонны, например:

$$\text{L8. } s \leq t \Rightarrow (s \upharpoonright A) \leq (t \upharpoonright A)$$

Двуместная функция может быть монотонной по каждому из аргументов. Например, конкатенация монотонна по второму (но не по первому) аргументу:

$$\text{L9. } t \leq u \Rightarrow (s^{\wedge} t) \leq (s^{\wedge} u)$$

Функция, монотонная по всем аргументам, называется просто монотонной.

1.6.6. Длина

Длину протокола t будем обозначать $\#t$. Например,

$$\# \langle x, y, x \rangle = 3$$

Следующие законы определяют операцию $\#$:

$$\text{L1. } \# \langle \rangle = 0$$

$$\text{L2. } \# \langle x \rangle = 1$$

$$\text{L3. } \# \langle s^{\wedge} t \rangle = (\#s) + (\#t)$$

Число вхождений в t символов из A вычисляется выражением $\#(t \upharpoonright A)$.

$$\text{L4. } \#(t \upharpoonright (A \cup B)) = \#(t \upharpoonright A) + \#(t \upharpoonright B) - \#(t \upharpoonright (A \cap B))$$

$$\text{L5. } s \leq t \Rightarrow \#s \leq \#t$$

$$\text{L6. } \#(t^n) = n \times (\#t)$$

Число вхождений символа x в протокол s определяется как

$$s \downarrow x = \#(s \upharpoonright \{x\})$$

1.7. РЕАЛИЗАЦИЯ ПРОТОКОЛОВ

Для машинного представления протоколов и реализации операций над ними нам потребуется язык высокого уровня с развитыми средствами работы со списками. К счастью, ЛИСП как нельзя лучше подходит для этой цели. Протоколы

в нем очевидным образом представляются списками атомов, соответствующих его событиям:

$$\begin{aligned}\langle \rangle &= NIL \\ \langle \text{мон} \rangle &= \text{cons}(\text{"МОИ"}, NIL) \\ \langle \text{мон, шок} \rangle &= \text{"(МОИ, ШОК)}\end{aligned}$$

что означает $\text{cons}(\text{"МОИ"}, \text{cons}(\text{"ШОК"}, NIL))$.

Операции над протоколами легко реализовать как функции над списками. Так, например, голова и хвост непустого списка задаются примитивами *car* и *cdr*:

$$\begin{aligned}t_0 &= \text{car}(t) \\ t' &= \text{cdr}(t) \\ \langle x \rangle \hat{\sim} s &= \text{cons}(x, s)\end{aligned}$$

Конкатенация в общем виде реализуется с помощью известной функции *append*, которая задается рекурсивно:

$$s \hat{\sim} t = \text{append}(s, t)$$

где $\text{append}(s, t) = \text{if } s = NIL \text{ then } t$
else $\text{cons}(\text{car}(s), \text{append}(\text{cdr}(s), t))$

Корректность этого определения вытекает из законов

$$\begin{aligned}\langle \rangle \hat{\sim} t &= t \\ s \hat{\sim} t &= \langle s_0 \rangle \hat{\sim} (s' \hat{\sim} t) \text{ когда } s \neq \langle \rangle.\end{aligned}$$

Завершение ЛИСП-функции *append* гарантируется тем, что при каждом рекурсивном вызове список, подставляемый в качестве первого аргумента, короче, чем на предыдущем уровне рекурсии. Подобное рассуждение позволяет установить корректность и других, приведенных ниже, реализаций.

Для реализации сужения представим конечное множество списком его элементов. Проверка $(x \in B)$ выполняется вызовом функции

$$\begin{aligned}\text{принадлежит}(x, B) &= \text{if } B = NIL \text{ then } \text{ложь} \\ &\text{else if } x = \text{car}(B) \text{ then } \text{истина} \\ &\text{else } \text{принадлежит}(x, \text{cdr}(B))\end{aligned}$$

после чего сужение $(s \upharpoonright B)$ может быть реализовано функцией

$$\begin{aligned}\text{сужение}(s, B) &= \text{if } s = NIL \text{ then } NIL \\ &\text{else if } \text{принадлежит}(\text{car}(s), B) \\ &\text{then } \text{cons}(\text{car}(s), \text{сужение}(\text{cdr}(s), B)) \\ &\text{else } \text{сужение}(\text{cdr}(s), B)\end{aligned}$$

Проверка $(s \leq t)$ реализуется функцией, дающей ответ *истина* или *ложь*; реализация основана на законах 1.6.5 L1

и L5:

```

префикс_ли (s,t) = if s = NIL then истина
                  else if t = NIL then ложь
                  else car(s) = car(t) &
                     префикс_ли (cdr(s), cdr(t))

```

1.8. ПРОТОКОЛЫ ПРОЦЕССА

В разд. 1.5 мы ввели понятие протокола как последовательной записи поведения процесса вплоть до некоторого момента времени. До начала процесса неизвестно, какой именно из возможных протоколов будет записан: его выбор зависит от внешних по отношению к процессу факторов. Однако полный набор всех возможных протоколов процесса P может быть известен заранее, и мы введем функцию *протоколы*(P) для обозначения этого множества.

Примеры

X1. Единственным возможным протоколом процесса *СТОП* является $\langle \rangle$. Блокнот наблюдателя этого процесса всегда остается чистым:

$$\text{протоколы}(\text{СТОП}) = \{ \langle \rangle \}$$

X2. Автомат, поглощающий монету, прежде чем сломаться, имеет всего два протокола:

$$\text{протоколы}(\text{мон} \rightarrow \text{СТОП}) = \{ \langle \rangle, \langle \text{мон} \rangle \}$$

X3. Часы, которые только тикают:

$$\begin{aligned} \text{протоколы}(\mu X. \text{тик} \rightarrow X) &= \{ \langle \rangle, \langle \text{тик} \rangle, \langle \text{тик}, \text{тик} \rangle, \dots \} \\ &= \{ \text{тик} \}^* \end{aligned}$$

Здесь, как и во всех наиболее содержательных процессах, множество протоколов бесконечно, хотя, разумеется, каждый отдельный протокол конечен.

X4. Простой торговый автомат:

$$\text{протоколы}(\mu X. \text{мон} \rightarrow \text{шок} \rightarrow X) = \{ s \mid \exists n. s \leq \langle \text{мон}, \text{шок} \rangle^n \}$$

1.8.1. Законы

В этом разделе мы покажем, как вычислить множество протоколов любого процесса, определенного с помощью уже введенных обозначений. Как отмечалось выше, процесс *СТОП*

имеет только один протокол:

$$\mathbf{L1.} \text{ протоколы}(\text{СТОП}) = \{t \mid t = \langle \rangle\} = \{\langle \rangle\}$$

Протокол процесса $(c \rightarrow P)$ может быть пустым, поскольку $\langle \rangle$ является протоколом поведения любого процесса до момента наступления его первого события. Каждый непустой протокол этого процесса начинается с c , а его хвост должен быть возможным протоколом P :

$$\mathbf{L2.} \text{ протоколы}(c \rightarrow P) = \{t \mid t = \langle \rangle \vee (t_0 = c \& t' \in \text{протоколы}(P))\} \\ = \{\langle \rangle \cup \{\langle c \rangle \wedge t \mid t \in \text{протоколы}(P)\}\}$$

Протокол процесса, содержащего выбор начального события, должен быть протоколом одной из альтернатив:

$$\mathbf{L3.} \text{ протоколы}(c \rightarrow P \mid d \rightarrow Q) = \\ \{t \mid t = \langle \rangle \vee (t_0 = c \& t' \in \text{протоколы}(P)) \\ \vee (t_0 = d \& t' \in \text{протоколы}(Q))\}$$

Эти три закона можно объединить в один общий закон, которому подчиняется конструкция выбора:

$$\mathbf{L4.} \text{ протоколы}(x : B \rightarrow P(x)) = \\ = \{t \mid t = \langle \rangle \vee (t_0 \in B \& t' \in \text{протоколы}(P(t_0)))\}$$

Несколько сложнее найти множество протоколов рекурсивно определенного процесса. Рекурсивно определенный процесс является решением уравнения $X = F(X)$. Сначала определим по индукции итерацию функции F :

$$F^0(X) = X \\ F^{n+1}(X) = F(F^n(X)) \\ = F^n(F(X)) \\ = \underbrace{F(\dots(F(F(X)))\dots)}_{n \text{ раз}}$$

Затем, при условии, что функция F — предваренная, можно определить

$$\mathbf{L5.} \text{ протоколы}(\mu X : A.F(X)) = \bigcup_{n \geq 0} \text{протоколы}(F^n(\text{СТОП}_A))$$

Примеры

X1. Вспомним, что в примере 1.1.3 X8 мы определили ИСП_A как

$\mu X : A.F(X)$, где $F(X) = (x : A \rightarrow X)$. Мы хотим доказать, что $\text{протоколы}(\text{ИСП}_A) = A^*$.

Доказательство: $A^* = \bigcup_{n \geq 0} \{s \mid s \in A^* \& \# s \leq n\}$

Поэтому, согласно **L5**, достаточно для всех n доказать, что

$$\text{протоколы}(F^n(\text{СТОП}_A)) = \{s \mid s \in A^* \& \#s \leq n\}$$

Это делается индукцией по n :

(1) Для $n = 0$

$$\begin{aligned} \text{протоколы}(\text{СТОП}_A) &= \{\langle \rangle\} \\ &= \{s \mid s \in A^* \& \#s \leq 0\} \end{aligned}$$

(2)

$$\begin{aligned} &\text{протоколы}(F^{n+1}(\text{СТОП}_A)) \\ &= \text{протоколы}(x: A \rightarrow F^n(\text{СТОП}_A)) \text{ по определению } F, F^{n+1} \\ &= \{t \mid t = \langle \rangle \vee (t_0 \in A \& t' \in \text{протоколы}(F^n(\text{СТОП}_A)))\} \quad \mathbf{L4} \\ &= \{t \mid t = \langle \rangle \vee (t_0 \in A \& (t' \in A^* \& \#t' \leq n))\} \quad \text{предположение} \\ &\quad \text{индукции} \\ &= \{t \mid (t = \langle \rangle \vee (t_0 \in A \& t' \in A^*)) \& \#t \leq n + 1\} \quad \text{свойство } \# \\ &= \{t \mid t \in A^* \& \#t \leq n + 1\} \quad 1.6.4 \quad \mathbf{L4} \end{aligned}$$

X2. Докажем 1.8 **X4**, т. е.

$$\text{протоколы}(\text{ТАП}) = \bigcup_{n \geq 0} \{s \mid s \leq \langle \text{мон}, \text{шок} \rangle^n\}$$

Доказательство:

Согласно предположению индукции

$$\text{протоколы}(F^n(\text{ТАП})) = \{t \mid t \leq \langle \text{мон}, \text{шок} \rangle^n\}$$

где $F(X) = (\text{мон} \rightarrow \text{шок} \rightarrow X)$

$$(1) \text{ протоколы}(\text{СТОП}) = \{\langle \rangle\} = \{s \mid s \leq \langle \text{мон}, \text{шок} \rangle^0\} \quad 1.6.1 \quad \mathbf{L6}$$

$$\begin{aligned} (2) \text{ протоколы}(\text{мон} \rightarrow \text{шок} \rightarrow F^n(\text{СТОП})) \\ &= \{\langle \rangle, \langle \text{мон} \rangle\} \cup \{\langle \text{мон}, \text{шок} \rangle^t \mid t \in \\ &\quad \in \text{протоколы}(F^n(\text{СТОП}))\} \quad \mathbf{L2} \text{ (дважды)} \\ &= \{\langle \rangle, \langle \text{мон} \rangle\} \cup \{\langle \text{мон}, \text{шок} \rangle^t \mid t \leq \langle \text{мон}, \text{шок} \rangle^n\} \quad \text{предположение} \\ &\quad \text{индукции} \\ &= \{s \mid s = \langle \rangle \vee s = \langle \text{мон} \rangle \vee \exists t. s = \langle \text{мон}, \text{шок} \rangle^t \& t \leq \langle \text{мон}, \text{шок} \rangle^n\} \\ &= \{s \mid s \leq \langle \text{мон}, \text{шок} \rangle^{n+1}\} \end{aligned}$$

Заключительная часть следует из **L5**.

Как упоминалось в разд. 1.5, протокол — это последовательность символов, фиксирующих события, в которых процесс P участвовал до некоторого момента времени. Отсюда следует, что $\langle \rangle$ является протоколом любого процесса до момента наступления его первого события. Кроме того, если (s^t) — протокол процесса до некоторого момента, то s должен быть протоколом того же процесса до некоторого более раннего момента времени. Наконец, каждое происходящее событие должно содержаться в алфавите процесса. Три этих

факта находят свое формальное выражение в законах:

L6. $\langle \rangle \in \text{протоколы}(P)$

L7. $s \hat{=} t \in \text{протоколы}(P) \Rightarrow s \in \text{протоколы}(P)$

L8. $\text{протоколы}(P) \subseteq (\alpha P)^*$

Существует тесная взаимосвязь между протоколами процесса и изображением его поведения в виде дерева. Для каждой вершины дерева протоколом процесса до момента достижения этой вершины будет просто последовательность меток,

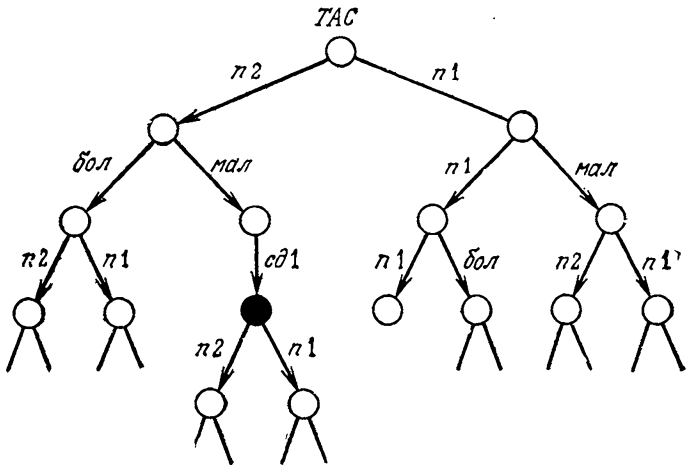


Рис. 1.1

встреченных на пути из корня дерева в эту вершину. Например, в дереве для *TAC*, приведенном на рис. 1.1, пути из корня в черную вершину соответствует протокол $\langle \text{п2, мал, сд1} \rangle$.

Очевидно, что все начальные отрезки путей в дереве сами являются путями в этом дереве; более строго это было сформулировано в законе **L7**. Пустой протокол определяет путь нулевой длины из корня в самого себя, что подтверждает закон **L6**. Протоколы процесса представляют собой множество всех путей, ведущих из корня в различные вершины дерева. И наоборот, так как все дуги, ведущие из каждой вершины, помечены различными событиями, каждый протокол процесса однозначно определяет путь из корня дерева в некоторую вершину. Таким образом, любое множество протоколов, удовлетворяющих законам **L6** и **L7**, является удобным математическим представлением для дерева без повторяющихся меток на дугах, исходящих из одной вершины.

1.8.2. Реализация

Предположим, что процесс реализован ЛИСП-функцией P , и пусть s — некоторый протокол. Определить, является ли s возможным протоколом P , можно с помощью функции

$протокол_ли(s, P) = \text{if } s = NIL \text{ then истина}$
 $\text{else if } P(s_0) = "BLEEP" \text{ then ложь}$
 $\text{else } протокол_ли(s', P(s_0))$

Так как протокол s конечен, то содержащаяся в этом определении рекурсия завершится после исследования лишь конечного по длине начального отрезка поведения процесса P . Вследствие того что мы избегаем бесконечного исследования поведения процесса, можно с уверенностью определить процесс как бесконечный объект, т. е. функцию, результат которой есть функция, результат которой есть функция, результат которой...

1.8.3. После

Если $s \in \text{протоколы}(P)$, то P/s (P после s) — это процесс, ведущий себя так, как ведет себя P с момента завершения всех действий, записанных в протоколе s . Если s не является протоколом P , то (P/s) не определено.

Примеры

X1. $(ТАП/\langle мон \rangle) = (шок \rightarrow ТАП)$

X2. $(ТАП/\langle мон., шок \rangle) = ТАП$

X3. $(ТАС/\langle n1 \rangle^3) = СТОП$

X4. Во избежание убытка от установки **ТАДОВЕР** (1.1.3 **X5**, **X6**) первую шоколадку владелец автомата решает съесть сам:

$(ТАДОВЕР/\langle шок \rangle) = ТАП2$

В древовидном представлении P (рис. 1.1) конструкции (P/s) соответствует все поддерево с корнем в конечной вершине пути, помеченного символами из s . Таким образом, в нашем примере поддерево, исходящее из черной вершины, описывается как $ТАС/\langle n2, мал, сd1 \rangle$

Следующие законы раскрывают значение операции $/$. Ничего не сделав, процесс остается неизменным:

L1. $P/\langle \rangle = P$

Поведение процесса P после завершения событий из $s^{\wedge}t$ совпадает с поведением (P/s) после завершения событий из t :

$$\text{L2. } P/(s^{\wedge}t) = (P/s)/t$$

Если процесс участвовал в событии c , его дальнейшее поведение определяется именно этим начальным выбором:

$$\text{L3. } (x : B \rightarrow P(x))/\langle c \rangle = P(c) \quad \text{при условии, что } c \in B$$

В качестве следствия мы получаем, что оператор $/\langle c \rangle$ является обратным по отношению к оператору префиксации $c \rightarrow$:

$$\text{L3A. } (c \rightarrow P)/\langle c \rangle = P$$

Протоколы (P/s) определяются как

$$\text{L4. } \text{протоколы}(P/s) = \{t \mid s^{\wedge}t \in \text{протоколы}(P)\} \quad \text{при условии, что } s \in \text{протоколы}(P)$$

Для доказательства того, что процесс P работает бесконечно, достаточно доказать, что

$$P/s \neq \text{СТОП} \quad \text{для всех } s \in \text{протоколы}(P)$$

Другим желательным свойством процесса является цикличность; по определению процесс P — *циклический*, если при любых обстоятельствах он может вернуться в начальное состояние, т. е.

$$\forall s : \text{протоколы}(P). \exists t. (P/(s^{\wedge}t) = P)$$

СТОП — это тривиальный циклический процесс; если же любой другой процесс циклический, он также обладает полезным свойством никогда не останавливаться.

Примеры

X1. Следующие процессы являются циклическими (1.1.3 X8, 1.1.2 X2, 1.1.3 X3, 1.1.4 X2):

$$\text{ИСП}_A, \text{ТАП}, (\text{шок} \rightarrow \text{ТАП}), \text{ТАШИ}, \text{СТ}_7$$

X2. Следующие процессы не являются циклическими, потому что они не могут вернуться в начальное состояние (1.1.2 X2, 1.1.3 X3, 1.1.3 X2):

$$(\text{мон} \rightarrow \text{ТАП}), (\text{шок} \rightarrow \text{ТАШИ}), (\text{вокруг} \rightarrow \text{СТ}_7)$$

Так, например, в начальном состоянии $(\text{шок} \rightarrow \text{ТАШИ})$ можно получить только шоколадку, тогда как в дальнейшем наряду с шоколадкой всегда можно выбрать ириску; следова-

тельно, никакое из этих последующих состояний не эквивалентно начальному.

Предостережение. Использование / в рекурсивно определенных процессах, к сожалению, лишает их свойства предваренности, и поэтому возникает опасность получения неоднозначных решений. Например, уравнение $X = (a \rightarrow (X / \langle a \rangle))$ не является предваренным и имеет решением *любой* процесс вида $a \rightarrow P$ для любого P .

Доказательство:

$$(a \rightarrow ((a \rightarrow P) / \langle a \rangle)) = (a \rightarrow P) \quad \text{согласно L3A}$$

По этой причине мы никогда не будем использовать операцию «/» в рекурсивных определениях процесса.

1.9. ДАЛЬНЕЙШИЕ ОПЕРАЦИИ НАД ПРОТОКОЛАМИ

Этот раздел посвящен некоторым дальнейшим операциям над протоколами. На данной стадии его можно пропустить, потому что в следующих главах, где используются эти операции, будут даны соответствующие ссылки.

1.9.1. Замена символа

Пусть f — функция, отображающая символы из множества A в множество B . С помощью f можно построить новую функцию f^* , отображающую последовательность символов из A^* в последовательность символов из B^* , путем применения f к каждому элементу последовательности. Например, если *удвоить* — функция, удваивающая свой целый аргумент, то

$$\text{удвоить}^*(\langle 1, 5, 3, 1 \rangle) = \langle 2, 10, 6, 2 \rangle$$

Очевидно, что функция со звездочкой является дистрибутивной и, следовательно, строгой.

$$\text{L1. } f^*(\langle \rangle) = \langle \rangle$$

$$\text{L2. } f^*(\langle x \rangle) = \langle f(x) \rangle$$

$$\text{L3. } f^*(s \hat{\ } t) = f^*(s) \hat{\ } f^*(t)$$

Остальные законы являются очевидными следствиями:

$$\text{L4. } f^*(s)_0 = f(s_0) \quad \text{если } s \neq \langle \rangle$$

$$\text{L5. } \# f^*(s) = \# s$$

Однако следующий «очевидный» закон, к сожалению, справедлив не всегда:

$$f^*(s \upharpoonright A) = f^*(s) \upharpoonright f(A), \quad \text{где } f(A) = \{f(x) \mid x \in A\}$$

Простейший контрпример представляет собой функция f , такая, что

$$f(b) = f(c) = c, \quad \text{где } b \neq c$$

Так как $b \neq c$, то, следовательно,

$$\begin{aligned} f^*(\langle b \rangle \uparrow \{c\}) &= f^*(\langle \rangle) \\ &= \langle \rangle && \text{по L1} \\ &\neq \langle c \rangle \\ &= \langle c \rangle \uparrow \{c\} \\ &= f^*(\langle c \rangle) \uparrow f(\{c\}) && \text{так как } f(c) = c \end{aligned}$$

Если же функция f взаимно однозначна, то данный закон выполняется:

L6. $f^*(s \uparrow A) = f^*(s) \uparrow f(A)$ при условии, что f — инъективная функция.

1.9.2. Конкатенация

Пусть s — последовательность, каждый элемент которой в свою очередь является последовательностью. Тогда \wedge/s получается из s конкатенацией всех ее элементов в их исходном порядке, например:

$$\begin{aligned} \wedge/\langle \langle 1,3 \rangle, \langle \rangle, \langle 7 \rangle \rangle &= \langle 1,3 \rangle \wedge \langle \rangle \wedge \langle 7 \rangle \\ &= \langle 1,3,7 \rangle \end{aligned}$$

Эта операция дистрибутивна:

$$\text{L1. } \wedge/\langle \rangle = \langle \rangle$$

$$\text{L2. } \wedge/\langle s \rangle = s$$

$$\text{L3. } \wedge/(s \wedge t) = (\wedge/s) \wedge (\wedge/t)$$

1.9.3. Чередование

Последовательность s является чередованием последовательностей t и u , если ее можно разбить на серию подпоследовательностей, которые, чередуясь, представляют собой подпоследовательности из t и u . Например,

$$s = \langle 1,6,3,1,5,4,2,7 \rangle$$

является чередованием t и u , где

$$t = \langle 1,6,5,2,7 \rangle, \quad \text{а } u = \langle 3,1,4 \rangle$$

Рекурсивное определение чередования можно задать следующими законами:

L1. $\langle \rangle$ чередование(t, u) $\equiv (t = \langle \rangle \& u \neq \langle \rangle)$

L2. s чередование(t, u) $\equiv s$ чередование(u, t)

L3. $\langle \langle x \rangle^s \rangle$ чередование(t, u) \equiv

$(t \neq \langle \rangle \& t_0 = x \& s \text{ чередование}(t', u))$

$(u \neq \langle \rangle \& u_0 = x \& s \text{ чередование}(t, u'))$

1.9.4. Индекс

Если $0 \leq i \leq \#s$, то мы используем привычную запись $s[i]$ для обозначения i -го элемента последовательности s , как это описано в законе **L1**:

L1. $s[0] = s_0$ & $s[i+1] = s'[i]$, при условии, что $s \neq \langle \rangle$

L2. $(f^*(s))[i] = f(s[i])$ для $i < \#s$

1.9.5. Обратный порядок

Если s — последовательность, то \bar{s} получается из s взятием ее элементов в обратном порядке. Например,

$$\langle \bar{3, 5, 37} \rangle = \langle 37, 5, 3 \rangle$$

Полностью обратный порядок задают следующие законы:

L1. $\overline{\langle \rangle} = \langle \rangle$

L2. $\overline{\langle x \rangle} = \langle x \rangle$

L3. $\overline{s \hat{t}} = \bar{t} \hat{\bar{s}}$

Обратный порядок обладает рядом простых алгебраических свойств, в том числе:

L4. $\bar{\bar{s}} = s$

Исследование остальных свойств мы оставляем читателю. Полезным является тот факт, что \bar{s}_0 есть *последний* элемент последовательности, а в общем случае

L5. $\bar{s}[i] = s[\#s - i - 1]$ для $i < \#s$

1.9.6. Выборка

Если s — последовательность пар, определим $s \downarrow x$ как результат выборки из s всех пар, первый элемент которых есть x , и замены каждой такой пары на ее второй элемент. Пер-

вый и второй элементы пары будем разделять точкой. Так, если

$$s = \langle a.7, b.9, a.8, c.0 \rangle, \text{ то } s \downarrow a = \langle 7, 8 \rangle, \text{ а } s \downarrow d = \langle \rangle.$$

$$\text{L1. } \langle \rangle \downarrow x = \langle \rangle$$

$$\text{L2. } \langle \langle y.z \rangle^{\wedge} t \rangle \downarrow x = t \downarrow x \quad \text{если } y \neq x$$

$$\text{L3. } \langle \langle x.z \rangle^{\wedge} t \rangle \downarrow x = \langle z \rangle^{\wedge} (t \downarrow x)$$

Если s не является последовательностью пар, то $s \downarrow a$ обозначает число вхождений a в s (как это определялось в разд. 1.6.6).

1.9.7. Композиция

Пусть символ \surd означает успешное завершение процесса. Значит, этот символ может появиться только в конце протокола. Пусть t — протокол, отражающий последовательность событий с того момента, как s успешно завершился. Композицию s и t обозначим $(s; t)$. Если в s отсутствует символ \surd , то t не может начаться:

$$\text{L1. } s; t = s \quad \text{если } \neg(\langle \surd \rangle \text{ в } s)$$

Если же символ \surd присутствует в конце s , то он удаляется а t присоединяется к результату:

$$\text{L2. } (s^{\wedge} \langle \surd \rangle); t = s^{\wedge} t \quad \text{если } \neg(\langle \surd \rangle \text{ в } s)$$

Символ \surd можно рассматривать как своего рода “клей”, склеивающий s и t ; при отсутствии клея t не может прилипнуть (L1). Если символ \surd встречается (что ошибочно) в середине протокола, то из соображений общности условимся, что все символы после его первого вхождения следует считать ошибочными и опустить.

$$\text{L2A. } (s^{\wedge} \langle \surd \rangle^{\wedge} u); t = s^{\wedge} t \quad \text{если } \neg(\langle \surd \rangle \text{ в } s)$$

Эта необычная операция композиции обладает рядом обычных алгебраических свойств. Как и конкатенация, она ассоциативна. В отличие от конкатенации она монотонна как по первому, так и по второму аргументу с $\langle \surd \rangle$ в качестве левой, единицы.

$$\text{L3. } s; (t; u) = (s; t); u$$

$$\text{L4A. } s \leq t \Rightarrow ((u; s) \leq (u; t))$$

$$\text{L4B. } s \leq t \Rightarrow ((s; u) \leq (t; u))$$

$$\text{L5. } \langle \rangle; t = \langle \rangle$$

$$\text{L6. } \langle \surd \rangle; t = t$$

Если символ \surd не встречается нигде, кроме конца протокола, то $\langle \surd \rangle$ является также и правой единицей:

L7. $s; \langle \surd \rangle = s$ при условии, что $\neg \langle \langle \surd \rangle \rangle$ в $(\bar{s})'$

1.10. СПЕЦИФИКАЦИИ

Спецификация изделия — это описание его предполагаемого поведения. Это описание представляет собой предикат, содержащий свободные переменные, каждая из которых соответствует некоторому обозримому аспекту поведения изделия. Например, спецификация электронного усилителя с входным диапазоном в один вольт и с усилением приблизительно в 10 раз задается предикатом

$$УСИЛ10 = (0 \leq v \leq 1 \Rightarrow |v' - 10 \times v| \leq 1)$$

Из этой спецификации ясно, что v обозначает входное, а v' — выходное напряжения. Без понимания физического смысла переменных вообще невозможно успешное применение математики в других науках и инженерном деле.

В случае процесса наиболее естественно в качестве результата наблюдения за его поведением рассматривать протокол событий, произошедших вплоть до данного момента времени. Для обозначения произвольного протокола процесса мы будем использовать специальную переменную nr , точно так же как в предыдущем примере v и v' используются для произвольных значений данных наблюдений за напряжением.

Примеры

X1. Владелец торгового автомата не желает терпеть убытков от его установки. Поэтому он оговаривает, что число выданных шоколадок не должно превышать числа опущенных монет:

$$НЕУБЫТ = (\#(nr \upharpoonright \{шок\}) \leq \#(nr \upharpoonright \{мон\}))$$

В дальнейшем мы будем использовать введенное в разд. 1.6.6 сокращение

$$nr \downarrow c = \#(nr \upharpoonright \{c\})$$

для обозначения числа вхождений символа c в nr .

X2. Пользователь автомата хочет быть уверенным в том, что машина не будет поглощать монеты, пока не выдаст уже оплаченный шоколад:

$$ЧЕСТН1 = ((nr \downarrow мон) \leq (nr \downarrow шоко) + 1)$$

X3. Изготовитель простого торгового автомата должен учесть требования как владельца, так и клиента:

$$\begin{aligned} \text{ТАПВЗАИМ} &= \text{НЕУБЫТ\&ЧЕСТН1} \\ &= (0 \leq ((\text{пр} \downarrow \text{мон}) - (\text{пр} \downarrow \text{шок})) \leq 1) \end{aligned}$$

X4. Спецификация исправления сложного торгового автомата не позволяет последнему принимать три монеты подряд:

$$\text{ТАСРЕМ} = (\neg \langle n1 \rangle^3 \text{ в пр})$$

X5. Спецификация исправленной машины

$$\text{ИСПРТАС} = (\text{пр} \in \text{протоколы}(\text{ТАС}) \& \text{ТАСРЕМ})$$

X6. Спецификация ТАП2 (1.1.3 X6)

$$0 \leq ((\text{пр} \downarrow \text{мон}) - (\text{пр} \downarrow \text{шок})) \leq 2$$

1.10.1. Соответствие спецификации

Если P — объект ¹⁾, отвечающий спецификации S , мы говорим, что P удовлетворяет S , сокращенно

P уд S

Это означает, что S описывает все возможные результаты наблюдения за поведением P , или, другими словами, S истинно всякий раз, когда его переменные принимают значения, полученные в результате наблюдения за объектом P , или, более формально, $\forall \text{пр. пр} \in \text{протоколы}(P) \Rightarrow S$. В следующей таблице, например, приведены некоторые результаты наблюдений за свойствами усилителя:

	1	2	3	4	5
ν	0	0,5	0,5	2	0,1
ν'	0	5	4	1	3

Все наблюдения, кроме последнего, описываются УСИЛ10. Вторая и третья колонки иллюстрируют тот факт, что выход усилителя не зависит целиком от его входа. Четвертая колонка показывает, что, если входное напряжение выходит за пределы указанного диапазона, на выходе мы можем получить что угодно, и это не будет нарушением спецификации.

¹⁾ В оригинале автор называет специфицируемый предмет продуктом (product); однако использование русского слова «продукт» вносит нежелательные смысловые аберрации. Поэтому в качестве русского эквивалента взяты неспецифические термины «объект» или «изделие», выбор между которыми определяется контекстом. — Прим. перев.

(В этом простом примере мы не учитываем того, что слишком высокое входное напряжение может привести к поломке объекта.)

В следующих законах приводятся наиболее общие свойства отношения *удовлетворяет*. Спецификации *истина*, не накладывающей никаких ограничений на поведение, будут удовлетворять все объекты; такой слабой и нетребовательной спецификации удовлетворяет даже неисправный объект:

L1. P уд *истина*

Если объект удовлетворяет двум различным спецификациям, он удовлетворяет также и их конъюнкции:

L2A. Если P уд S и P уд T , то P уд $(S \& T)$

Закон L2A можно обобщить на бесконечное число конъюнкций, т. е. на предикаты, содержащие квантор всеобщности. Пусть $S(n)$ — предикат, содержащий переменную n .

L2. Если $\forall n. (P \text{ уд } S(n))$, то P уд $(\forall n. S(n))$

при условии что P не зависит от n .

Если из спецификации S логически следует другая спецификация T , то всякое наблюдение, описываемое S , описывается также и T . Следовательно, каждый объект, удовлетворяющий S , должен удовлетворять также и более слабой спецификации T :

L3. Если $P \text{ уд } S$ и $S \Rightarrow T$, то $P \text{ уд } T$

В свете этого закона мы будем иногда записывать доказательства в виде цепочки, т. е., если $S \Rightarrow T$, мы запишем

$$\begin{array}{l} P \text{ уд } S \\ \Rightarrow T \end{array}$$

как сокращение более полного доказательства:

$$\begin{array}{l} P \text{ уд } S \\ S \Rightarrow T \\ P \text{ уд } T \end{array}$$

по L3

Приведенные выше законы и пояснения к ним применимы ко всем видам объектов и ко всем видам спецификаций. В следующем разделе будут приведены дополнительные законы, применимые к процессам.

1.10.2. Доказательства

При проектировании изделия разработчик несет ответственность за то, чтобы оно соответствовало своей спецификации; эта ответственность может быть реализована посредством

обращения к методам соответствующих разделов математики, например геометрии или дифференциального и интегрального исчисления. В этом разделе мы приводим набор законов, позволяющих с помощью математических рассуждений убедиться в том, что процесс P соответствует своей спецификации S .

Иногда мы будем записывать спецификацию как $S(np)$, предполагая, что спецификация обычно содержит np в качестве свободной переменной. В действительности же мы явно записываем np , чтобы указать, что ее можно заменить на более сложное выражение, как, например, в $S(np'')$. Важно отметить, что как S , так и $S(np)$ кроме np могут иметь и другие свободные переменные.

Результатом наблюдения за процессом *СТОП* всегда будет пустой протокол, так как этот процесс никогда ничего не делает:

L4A. *СТОП* уд $(np = \langle \rangle)$

Протокол процесса $(c \rightarrow P)$ вначале пуст. Каждый последующий протокол начинается с c , а его хвост является протоколом P . Следовательно, хвост может быть описан любой спецификацией для P :

L4B. Если P уд $S(np)$, то $(c \rightarrow P)$ уд $(np = \langle \rangle \vee (np_0 = c \ \& \ S(np')))$

В качестве следствия получаем закон для двойной префиксации:

L4C. Если P уд $S(np)$, то $(c \rightarrow d \rightarrow P)$ уд $(np \leq \langle c, d \rangle \vee (np \geq \langle c, d \rangle \ \& \ S(np'')))$

Двуместный выбор аналогичен префиксации с той разницей, что протокол может начинаться с любого из двух событий-альтернатив, а хвост должен описываться спецификацией выбранной альтернативы:

L4D. Если P уд $S(np)$, а Q уд $T(np)$, то $(c \rightarrow P \mid d \rightarrow Q)$ уд $(np = \langle \rangle \vee (np_0 = c \ \& \ S(np')) \vee (np_0 = d \ \& \ T(np')))$

Все приведенные выше законы являются частными случаями закона для обобщенного оператора выбора:

L4. Если $\forall x \in B. (P(x) \text{ уд } S(np, x))$, то $(x : B \rightarrow P(x))$ уд $(np = \langle \rangle \vee (np_0 \in B \ \& \ S(np', np_0)))$

Закон для оператора *после* удивительно прост. Если np — протокол (P/s) , то $s \hat{~} np$ является протоколом P и поэтому

должен описываться любой спецификацией, которой удовлетворяет P :

L5. Если P уд $S(nr)$, а $s \in \text{протоколы}(P)$, то (P/s) уд $S(s \hat{\ } nr)$

И наконец, нам нужен закон, чтобы устанавливать корректность рекурсивно определенного процесса.

L6. Если $F(X)$ — предваренная, **СТОП** уд S , а $((X \text{ уд } S) \Rightarrow (F(X) \text{ уд } S))$, то $(\mu X.F(X))$ уд S

Из левой части этого закона по индукции следует, что

$$F^n(\text{СТОП}) \text{ уд } S \text{ для всех } n$$

Так как F предварена, то $F^n(\text{СТОП})$ полностью описывает как минимум n первых шагов в поведении $\mu X.F(X)$. Поэтому каждый протокол $\mu X.F(X)$ является протоколом $F^n(\text{СТОП})$ для некоторого n . Значит, этот протокол должен удовлетворять той же спецификации, что и $F^n(\text{СТОП})$, которая (для всех n) есть S . Более строгое доказательство можно дать в терминах математической теории из разд. 2.8.

Пример

X1. Мы хотим доказать (1.1.2 **X2**, 1.10 **X3**), что

ТАП уд **ТАПВЗАИМ**

Доказательство:

$$\begin{aligned} (1) \text{ СТОП уд } nr = \langle \rangle & \quad \text{L4A} \\ \Rightarrow 0 \leq (nr \downarrow \text{мон} - nr \downarrow \text{шок}) \leq 1, & \\ \text{так как } (\langle \rangle \downarrow \text{мон}) = (\langle \rangle \downarrow \text{шок}) = 0. & \end{aligned}$$

Это заключение сделано на основании неявного применения закона **L3**.

$$(2) \text{ Предположим, что } X \text{ уд } (0 \leq ((nr \downarrow \text{мон}) - (nr \downarrow \text{шок})) \leq 1).$$

$$\begin{aligned} \text{Тогда } (\text{мон} \rightarrow \text{шок} \rightarrow X) \text{ уд } (nr \leq \langle \text{мон,шок} \rangle) & \\ \vee (nr \geq \langle \text{мон,шок} \rangle) & \\ \& 0 \leq ((nr'' \downarrow \text{мон}) - (nr'' \downarrow \text{шок})) \leq 1)) & \text{L4C} \\ \Rightarrow 0 \leq ((nr \downarrow \text{мон}) - (nr \downarrow \text{шок})) \leq 1, & \end{aligned}$$

$$\text{так как } \langle \rangle \downarrow \text{мон} = \langle \rangle \downarrow \text{шок} = \langle \text{мон} \rangle \downarrow \text{шок} = 0,$$

$$\text{а } \langle \text{мон} \rangle \downarrow \text{мон} = \langle \text{мон,шок} \rangle \downarrow \text{мон} = \langle \text{мон,шок} \rangle \downarrow \text{шок} = 1$$

$$\text{и } nr \geq \langle \text{мон,шок} \rangle \Rightarrow$$

$$\Rightarrow (nr \downarrow \text{мон} = nr'' \downarrow \text{мон} + 1 \& nr \downarrow \text{шок} = nr'' \downarrow \text{шок} + 1).$$

Применяя теперь закон **L3**, а затем — **L6**, получим требуемый результат.

Тот факт, что процесс P удовлетворяет спецификации, еще не означает, что он будет нормально функционировать. Например, так как

$$pr = \langle \rangle \Rightarrow 0 \leq (pr \downarrow мон - pr \downarrow шок) \leq 1$$

то с помощью законов **L3** и **L4** можно доказать, что

$$СТОП \text{ уд } 0 \leq (pr \downarrow мон - pr \downarrow шок) \leq 1$$

Однако *СТОП* вряд ли соответствует требованиям торгового автомата, с точки зрения как владельца, так и клиента. Он, несомненно, не сделает ничего плохого, но только благодаря тому, что он не делает ничего вообще. По этой же причине *СТОП* удовлетворяет любой спецификации, которой может удовлетворять процесс.

К счастью, по независимым соображениям очевидно, что *ТАП* никогда не останавливается. На самом деле, любой процесс, определенный только с помощью префикса, выбора и предваренной рекурсии, никогда не останавливается. Единственный способ записать останавливающийся процесс — это явно включить в него *СТОП* или эквивалентный процесс $(x: B \rightarrow P(x))$, где B — пустое множество. Избегая таких элементарных ошибок, можно с гарантией записывать бесконечные процессы. Однако после введения в следующей главе параллелизма такие простые меры предосторожности становятся недостаточными. Более общий способ спецификации и доказательства свойства бесконечности процесса описывается в разд. 3.7.

Глава 2. Параллельные процессы

2.1. ВВЕДЕНИЕ

Процесс определяется полным описанием его потенциального поведения. При этом часто имеется выбор между несколькими различными действиями, например опусканием большой или маленькой монеты в щель торгового автомата ТАС (1.1.3 Х4). В каждом таком случае выбор того, какое из событий произойдет в действительности, может зависеть от окружения, в котором работает процесс. Так, например, покупатель сам решает, какую монету ему опустить. К счастью, само окружение процесса может быть описано как процесс, поведение которого определяется в уже знакомых нам терминах. Это позволяет исследовать поведение целой системы, состоящей из процесса и его окружения, взаимодействующих по мере их параллельного исполнения. Всю систему также следует рассматривать как процесс, поведение которого определяется в терминах поведения составляющих его процессов; эта система в свою очередь может быть помещена в еще более широкое окружение. На самом деле, лучше всего забыть разницу между процессами, окружениями и системами; все они — всего лишь процессы, поведение которых может быть предписано, описано, зафиксировано и проанализировано простым и единообразным способом.

2.2. ВЗАИМОДЕЙСТВИЕ

Объединяя два процесса для совместного исполнения, мы, как правило, хотим, чтобы они взаимодействовали друг с другом. Эти взаимодействия можно рассматривать как события, требующие одновременного участия обоих процессов. Давайте на некоторое время сосредоточимся лишь на таких событиях и забудем обо всех остальных. Будем поэтому предполагать, что алфавиты обоих процессов совпадают. Следовательно, каждое происходящее событие является допустимым в независимом поведении каждого процесса в отдельности. Шоколадку, например, можно извлечь, только когда этого хочет покупатель и только когда автомат готов ее выдать. Если P и Q — процессы с одинаковым алфавитом,

обозначим через $P \parallel Q$ процесс, ведущий себя как система, составленная из процессов P и Q , взаимодействие между которыми пошагово синхронизовано, как это описано выше.

Примеры

X1. Жадный покупатель был бы не прочь получить ириску или даже шоколадку бесплатно. Однако, если это не вышло, он с неохотой готов заплатить, но при этом настаивает на получении шоколадки:

$$\begin{aligned} \text{ЖАДН} = & (\text{ирис} \rightarrow \text{ЖАДН} \\ & | \text{шок} \rightarrow \text{ЖАДН} \\ & | \text{мон} \rightarrow \text{шок} \rightarrow \text{ЖАДН}) \end{aligned}$$

Когда такой покупатель сталкивается с автоматом *ТАШИ* (1.1.3 X3), его алчные попытки оказываются несостоятельными, потому что этот торговый автомат не позволяет получить товар прежде, чем он будет оплачен:

$$(\text{ЖАДН} \parallel \text{ТАШИ}) = \mu X. (\text{мон} \rightarrow \text{шок} \rightarrow X)$$

Этот пример показывает, как можно описать процесс, заданный в виде композиции двух подпроцессов, без использования параллельного оператора \parallel .

X2. Рассеянный покупатель хочет большой коржик и поэтому опускает монету в торговый автомат *ТАС*. Он не обратил внимания, какую монету он опустил — большую или маленькую, однако настаивает только на большом коржике:

$$\text{ТАС} = (n2 \rightarrow \text{бол} \rightarrow \text{ТАС} \mid n1 \rightarrow \text{бол} \rightarrow \text{ТАС})$$

К сожалению, торговый автомат не предусматривает выдачу большого коржика за маленькую монету:

$$(\text{ТАС} \parallel \text{ТАС}) = \mu X. (n2 \rightarrow \text{бол} \rightarrow X \mid n1 \rightarrow \text{СТОП})$$

Событие *СТОП*, следующее за первым же $n1$, известно как тупиковая ситуация или *дедлок*. Хотя каждый из составляющих процессов готов к некоторым дальнейшим действиям, действия эти различны. И так как процессы не могут прийти к соглашению о том, какое действие будет следующим, ничего в дальнейшем произойти не может.

Сюжеты, сопровождающие эти примеры, являют собой образец полной измены принципу научной абстракции и объективности. Важно помнить, что, согласно нашему предположению, события являются нейтральными переходами, которые могут наблюдаться и фиксироваться некоторым

бесстрастным пришельцем с другой планеты, не ведающим ни радости поедания коржика, ни голода незадачливого покупателя, тщетно пытающегося добыть себе пропитание. При выборе алфавита соответствующих событий мы сознательно не включили в него эти внутренние эмоциональные состояния; при желании же дополнительные события, моделирующие изменения «внутреннего» состояния, могут быть введены, как показано в 2.3 X1.

2.2.1. Законы

Законы, управляющие поведением ($P \parallel Q$), исключительно просты и регулярны. Первый закон выражает логическую симметрию между процессом и его окружением:

$$\text{L1. } P \parallel Q = Q \parallel P.$$

Следующий закон показывает, что при совместной работе трех процессов неважно, в каком порядке они объединены оператором параллельной композиции:

$$\text{L2. } P \parallel (Q \parallel R) = (P \parallel Q) \parallel R$$

В-третьих, процесс, находящийся в тупиковой ситуации, приводит к дедлоку всей системы: композиция же с $ИСП_{\alpha P}$ (1.1.3 X8) ничего не меняет:

$$\text{L3A. } P \parallel \text{СТОП}_{\alpha P} = \text{СТОП}_{\alpha P}$$

$$\text{L3B. } P \parallel ИСП_{\alpha P} = P$$

Следующий закон гласит, что пара процессов либо одновременно выполняет одно и то же действие, либо попадает в состояние дедлока, если начальные события процессов не совпадают:

$$\text{L4A. } (c \rightarrow P) \parallel (c \rightarrow Q) = (c \rightarrow (P \parallel Q))$$

$$\text{L4B. } (c \rightarrow P) \parallel (d \rightarrow Q) = \text{СТОП}, \text{ если } c \neq d$$

Эти законы легко обобщить на случаи, когда у одного или обоих процессов имеется выбор начального события: при комбинации процессов остаются возможными лишь те события, которые содержатся во множествах выбора обоих процессов:

$$\text{L4. } (x : A \rightarrow P(x)) \parallel (y : B \rightarrow Q(y)) = (z : (A \cap B) \rightarrow (P(z) \parallel Q(z)))$$

Именно этот закон позволяет описать систему, определенную в терминах параллельного исполнения без использования параллелизма.

дают, то этот же алфавит имеет и их результирующая комбинация, и в этом случае у операции $(P \parallel Q)$ в точности тот же смысл, что и у операции, описанной в предыдущем разделе.

Примеры

X1. Пусть $\alpha\text{ТАШУМ} = \{\text{мон}, \text{шок}, \text{звяк}, \text{щелк}, \text{ирис}\}$, где «звяк» — звук монеты, попадающей в монетоприемник шумного торгового автомата, а «щелк» — щелчок, издаваемый автоматом при завершении каждой торговой операции. Если у шумного торгового автомата кончился запас ирисков, то

$$\text{ТАШУМ} = (\text{мон} \rightarrow \text{звяк} \rightarrow \text{шок} \rightarrow \text{щелк} \rightarrow \text{ТАШУМ})$$

Покупатель у этого автомата определенно предпочитает ириску: не сумев получить ее, он восклицает «черт», после чего берет шоколадку:

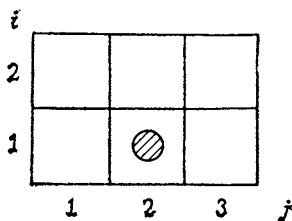
$$\begin{aligned} \alpha\text{КЛИЕНТ} &= \{\text{мон}, \text{шок}, \text{черт}, \text{ирис}\} \\ \text{КЛИЕНТ} &= (\text{мон} \rightarrow (\text{ирис} \rightarrow \text{КЛИЕНТ} \\ &\quad | \text{черт} \rightarrow \text{шок} \rightarrow \text{КЛИЕНТ})) \end{aligned}$$

Результатом параллельной работы этих двух процессов является

$$\begin{aligned} (\text{ТАШУМ} \parallel \text{КЛИЕНТ}) &= \\ &= \mu X. (\text{мон} \rightarrow (\text{звяк} \rightarrow \text{черт} \rightarrow \text{шок} \rightarrow \text{щелк} \rightarrow X \\ &\quad | \text{черт} \rightarrow \text{звяк} \rightarrow \text{шок} \rightarrow \text{щелк} \rightarrow X)) \end{aligned}$$

Заметим, что событие *звяк* может как предшествовать событию *черт*, так и наоборот. Они могут произойти и одновременно, и порядок, в котором они будут зафиксированы, не имеет значения. Заметим, что настоящая математическая формула никак не отражает того факта, что клиент предпочитает получить ириску, нежели выругаться. Формула есть абстракция реальной действительности, не учитывающая людские эмоции и сосредоточенная лишь на описании возможности наступления или ненаступления событий из алфавита процесса независимо от того, желательны они или нет.

X2. Фишка стартует со средней нижней клетки поля и может



двигаться по нему *вверх, вниз, влево* или *вправо*. Пусть

$$\begin{aligned}\alpha P &= \{\text{вверх, вниз}\} \\ P &= (\text{вверх} \rightarrow \text{вниз} \rightarrow P) \\ \alpha Q &= \{\text{влево, вправо}\} \\ Q &= (\text{вправо} \rightarrow \text{влево} \rightarrow Q \\ &\quad | \text{влево} \rightarrow \text{вправо} \rightarrow Q)\end{aligned}$$

Поведение фишки можно определить как $P \parallel Q$. В этом примере алфавиты αP и αQ не имеют общих событий. Перемещения фишки, следовательно, представляют собой произвольное чередование действий процессов P и Q . Без помощи параллелизма описать такое чередование очень сложно. Обычно требуется взаимная рекурсия с уравнением для каждого состояния системы. Пусть, например, R_{ij} отвечает поведению фишки (X2), находящейся в i -строке и j -м столбце поля, где $i \in \{1, 2\}$, $j \in \{1, 2, 3\}$.

Тогда $(P \parallel Q) = R_{12}$, где

$$\begin{aligned}R_{21} &= (\text{вниз} \rightarrow R_{11} | \text{вправо} \rightarrow R_{22}) \\ R_{11} &= (\text{вверх} \rightarrow R_{21} | \text{вправо} \rightarrow R_{12}) \\ R_{22} &= (\text{вниз} \rightarrow R_{12} | \text{влево} \rightarrow R_{21} | \text{вправо} \rightarrow R_{23}) \\ R_{12} &= (\text{вверх} \rightarrow R_{22} | \text{влево} \rightarrow R_{11} | \text{вправо} \rightarrow R_{13}) \\ R_{23} &= (\text{вниз} \rightarrow R_{13} | \text{влево} \rightarrow R_{22}) \\ R_{13} &= (\text{вверх} \rightarrow R_{23} | \text{влево} \rightarrow R_{12})\end{aligned}$$

2.3.1. Законы

Первые три закона для расширенной операции параллельной композиции совпадают с аналогичными законами для оператора взаимодействия (2.2.1).

L1.2. Операция \parallel симметрична и ассоциативна

L3A. $P \parallel \text{СТОП}_{\alpha P} = \text{СТОП}_{\alpha P}$

L3B. $P \parallel \text{ИСП}_{\alpha P} = P$

Пусть $a \in (\alpha P - \alpha Q)$, $b \in (\alpha Q - \alpha P)$, а $\{c, d\} \subseteq (\alpha P \cap \alpha Q)$. Следующие законы показывают, каким образом процесс P один участвует в событии a , Q один участвует в b , а c и d требуют одновременного участия P и Q .

L4A. $(c \rightarrow P) \parallel (c \rightarrow Q) = c \rightarrow (P \parallel Q)$

L4B. $(c \rightarrow P) \parallel (d \rightarrow Q) = \text{СТОП}$, если $c \neq d$

L5A. $(a \rightarrow P) \parallel (c \rightarrow Q) = a \rightarrow (P \parallel (c \rightarrow Q))$

L5B. $(c \rightarrow P) \parallel (b \rightarrow Q) = b \rightarrow ((c \rightarrow P) \parallel Q)$

L6. $(a \rightarrow P) \parallel (b \rightarrow Q) = (a \rightarrow (P \parallel (b \rightarrow Q))) \mid b \rightarrow ((a \rightarrow P) \parallel Q)$.

Эти законы можно обобщить для общего случая оператора выбора:

L7. Пусть $P = (x : A \rightarrow P(x))$, $Q = (y : B \rightarrow Q(y))$.

Тогда $(P \parallel Q) = (z : C \rightarrow (P' \parallel Q'))$,

где $C = (A \cap B) \cup (A - \alpha Q) \cup (B - \alpha P)$,

$P' = P(z)$, если $z \in A$

$= P$ иначе,

а $Q' = Q(z)$, если $z \in B$

$= Q$ иначе.

С помощью этих законов можно переопределить процесс, удалив из его описания параллельный оператор, как это показано в следующем примере.

Пример

X1. Пусть $\alpha P = \{a, c\}$, $\alpha Q = \{b, c\}$, $P = (a \rightarrow c \rightarrow P)$,
 $Q = (c \rightarrow b \rightarrow Q)$.

Тогда $P \parallel Q = (a \rightarrow c \rightarrow P) \parallel (c \rightarrow b \rightarrow Q)$ по определению
 $= a \rightarrow ((c \rightarrow P) \parallel (c \rightarrow b \rightarrow Q))$ по **L5**
 $= a \rightarrow c \rightarrow (P \parallel (b \rightarrow Q))$ по **L4... (1)**
 а $P \parallel (b \rightarrow Q) = (a \rightarrow (c \rightarrow P) \parallel (b \rightarrow Q))$
 $\quad | b \rightarrow (P \parallel Q))$ по **L6**
 $= (a \rightarrow b \rightarrow ((c \rightarrow P) \parallel Q))$
 $\quad | b \rightarrow (P \parallel Q))$ по **L5**
 $= (a \rightarrow b \rightarrow c \rightarrow (P \parallel (b \rightarrow Q)))$ { по **L4**
 $\quad | b \rightarrow a \rightarrow c \rightarrow (P \parallel (b \rightarrow Q)))$ { и (1) выше
 $= \mu X. (a \rightarrow b \rightarrow c \rightarrow X$ { в силу
 $\quad | b \rightarrow a \rightarrow c \rightarrow X)$ { предваренности

Отсюда следует, что

$(P \parallel Q) = (a \rightarrow c \rightarrow \mu X. (a \rightarrow b \rightarrow c \rightarrow X$
 $\quad | b \rightarrow a \rightarrow c \rightarrow X))$ по (1) выше

2.3.2. Реализация

Реализация операции \parallel выводится непосредственно из закона **L7**. Алфавиты операндов представляются конечными списками символов A и B . В проверке на принадлежность используется функция *принадлежит*(x, A), определенная в разд. 1.7.

$(P \parallel Q)$ реализуется вызовом функции

параллельно($P, \alpha P, \alpha Q, Q$)

определенной как

$$\text{параллельно}(P, A, B, Q) = \text{вспом}(P, Q)$$

где

```

вспом(P, Q) =
λx. if P = "BLEEP or Q = "BLEEP then "BLEEP
    else if принадлежит(x, A) & принадлежит(x, B)
        then вспом(P(x), Q(x))
    else if принадлежит(x, A) then вспом(P(x), Q)
    else if принадлежит(x, B) then вспом(P, Q(x)) else "BLEEP

```

2.3.3. Протоколы

Пусть t — протокол ($P \parallel Q$). Тогда все события из t , принадлежащие алфавиту P , являлись событиями в жизни P , а все события из t , не принадлежащие αP , происходили без участия P . Таким образом, $(t \upharpoonright \alpha P)$ — это протокол всех событий, в которых участвовал процесс P , и поэтому он является протоколом P . По тем же соображениям $(t \upharpoonright \alpha Q)$ является протоколом Q . Более того, каждое событие из t должно содержаться либо в αP , либо в αQ . Эти рассуждения позволяют сформулировать закон

$$\begin{aligned} \mathbf{L1.} \text{ протоколы}(P \parallel Q) = \{t \mid (t \upharpoonright \alpha P) \in \text{протоколы}(P) \\ \& (t \upharpoonright \alpha Q) \in \text{протоколы}(Q) \\ \& t \in (\alpha P \cup \alpha Q)^*\} \end{aligned}$$

Следующий закон демонстрирует дистрибутивность операции $/s$ относительно оператора параллельной композиции:

$$\mathbf{L2.} (P \parallel Q)/s = (P/(s \upharpoonright \alpha P)) \parallel (Q/(s \upharpoonright \alpha Q))$$

Если $\alpha P = \alpha Q$, то $s \upharpoonright \alpha P = s \upharpoonright \alpha Q = s$, и тогда законы **L1**, **L2** совпадают с аналогичными из разд. 2.2.3.

Пример

X1. См. 2.3 **X1**.

Пусть $t1 = \langle \text{мон, звяк, черт} \rangle$.

Тогда $t1 \upharpoonright \alpha \text{ТАШУМ} = \langle \text{мон, звяк} \rangle$, что принадлежит множеству $\text{протоколы}(\text{ТАШУМ})$,

а $t1 \upharpoonright \alpha \text{КЛИЕНТ} = \langle \text{мон, черт} \rangle$, что принадлежит множеству $\text{протоколы}(\text{КЛИЕНТ})$.

Следовательно, $t1 \in \text{протоколы}(\text{ТАШУМ} \parallel \text{КЛИЕНТ})$. Аналогичные рассуждения показывают, что

$$\langle \text{мон, черт, звяк} \rangle \in \text{протоколы}(\text{ТАШУМ} \parallel \text{КЛИЕНТ})$$

Отсюда видно, что события *черт* и *звяк* могут происходить и фиксироваться в любом порядке. Они могут произойти и одновременно, но мы решили никак не отражать этот факт.

В общих словах, протокол $(P \parallel Q)$ представляет собой некоторую разновидность чередования протокола P с протоколом Q , в котором события, содержащиеся в обоих алфавитах, записаны по одному разу. Если $\alpha P \cap \alpha Q = \{\}$, то этот протокол в чистом виде является чередованием (разд. 1.9.3), как показано в примере 2.3 X2. В другом крайнем случае, когда $\alpha P = \alpha Q$, каждое событие принадлежит обоим алфавитам, и $(P \parallel Q)$ имеет в точности тот же смысл, что и взаимодействие, определенное в разд. 2.2.

- L3A.** Если $\alpha P \cap \alpha Q = \{\}$, то
 $\text{протоколы}(P \parallel Q) = \{s \mid \exists t: \text{протоколы}(P). \exists u: \text{протоколы}(Q). s \text{ чередование}(t, u)\}$
- L3B.** Если $\alpha P = \alpha Q$, то
 $\text{протоколы}(P \parallel Q) = \text{протоколы}(P) \cap \text{протоколы}(Q)$

2.4. РИСУНКИ

Процесс P с алфавитом $\{a, b, c\}$ изображается прямоугольником с меткой P , из которого исходят линии, помеченные различными событиями из его алфавита (рис. 2.1). Аналогично процесс Q с алфавитом $\{b, c, d\}$ можно изобразить, как показано на рис. 2.2. Когда эти два процесса объединяются для параллельного исполнения, полученную систему можно

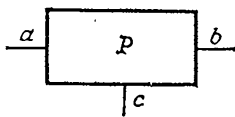


Рис. 2.1

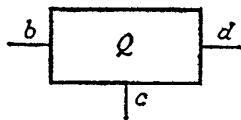


Рис. 2.2

изобразить в виде сети, в которой линии с одинаковыми метками соединены, а линии, помеченные событиями, принадлежащими алфавиту только одного процесса, оставлены свободными (рис. 2.3). Сюда можно добавить третий процесс R с алфавитом $\alpha R = \{c, e\}$, как показано на рис. 2.4. Из этой диаграммы видно, что событие c требует участия всех трех процессов, b требует участия P и Q , тогда как все остальные события имеют отношение только к отдельным процессам.

Рисунки эти, однако, легко могут ввести в заблуждение. Система, построенная из трех процессов, все же представляет собой один процесс и должна поэтому изображаться одним

прямоугольным (рис. 2.5). Число 60 тоже можно получить как произведение трех других чисел ($3 \times 4 \times 5$), но после того, как оно было таким образом получено, это всего лишь

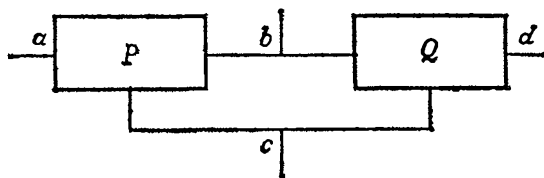


Рис. 2.3

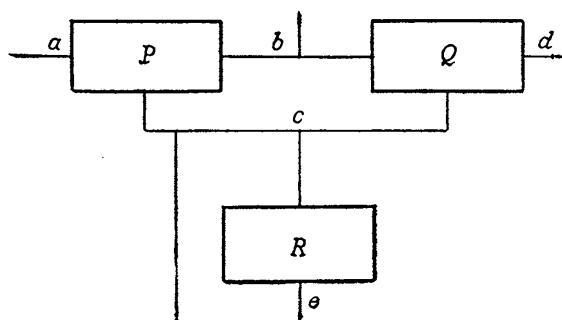


Рис. 2.4

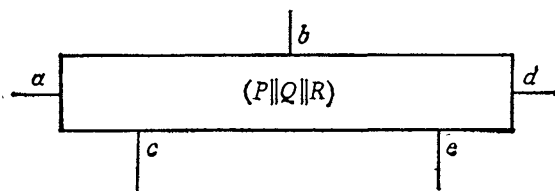


Рис. 2.5

отдельное число, и способ его получения перестает быть достойным внимания.

2.5. ПРИМЕР: ОБЕДАЮЩИЕ ФИЛОСОФЫ

В давние времена один богатый филантроп пожертвовал свой капитал на основание некоего пансиона, чтобы дать пристанище пяти знаменитым философам. У каждого философа была своя комната, где он мог предаваться размышлениям.

Была у них и общая столовая с круглым столом, вокруг которого стояли пять стульев, каждый помеченный именем того философа, которому он предназначался. Звали философов $\Phi ИЛ_0$, $\Phi ИЛ_1$, $\Phi ИЛ_2$, $\Phi ИЛ_3$ и $\Phi ИЛ_4$, и за столом они располагались в этом же порядке против часовой стрелки. Слева от каждого философа лежала золотая вилка, а в центре стола стояла большая миска спагетти, содержимое которой постоянно пополнялось.

Предполагалось, что большую часть времени философ проводил в размышлениях, а почувствовав голод, шел в столовую, садился на свой стул, брал слева от себя вилку и приступал к еде. Но такова уж сложная природа спагетти, что их не донести до рта без помощи второй вилки. Поэтому философу приходилось брать вилку и справа от себя. Закончив трапезу, он клал на место обе вилки, выходил из-за стола и возвращался к своим размышлениям. Разумеется, одной вилкой философы могли пользоваться только по очереди. Если вилка требовалась другому философу, ему приходилось ждать, пока она освободится.

2.5.1. Алфавиты

Теперь построим математическую модель этой системы. Сначала надо выбрать подходящие множества событий. Для $\Phi ИЛ_i$ определим это множество как

$$\alpha\Phi ИЛ_i = \{i.садится, i.встает, \\ i.берет\ вилку.i, i.берет\ вилку.(i \oplus 1), \\ i.кладет\ вилку.i, i.кладет\ вилку.(i \oplus 1)\}$$

где \oplus — сложение по модулю 5, т. е. $i \oplus 1$ указывает правого соседа i -го философа. Заметим, что алфавиты философов друг с другом не пересекаются. Ни в одном из событий философы не участвуют совместно, и поэтому возможность какого бы то ни было общения или взаимодействия между ними исключена — весьма реалистическое отражение поведения философов тех дней.

Другие «действующие лица» в нашей маленькой драме — это пять вилок, каждая из которых имеет тот же номер, что и философ, которому она принадлежит. Вилку берет и кладет на место или сам этот философ, или его сосед слева. Алфавит i -й вилки определим как

$$\alphaВИЛКА_i = \{i.берет\ вилку.i, (i \ominus 1).берет\ вилку.i, \\ i.кладет\ вилку.i, (i \ominus 1).кладет\ вилку.i\}$$

где \ominus обозначает вычитание по модулю 5.

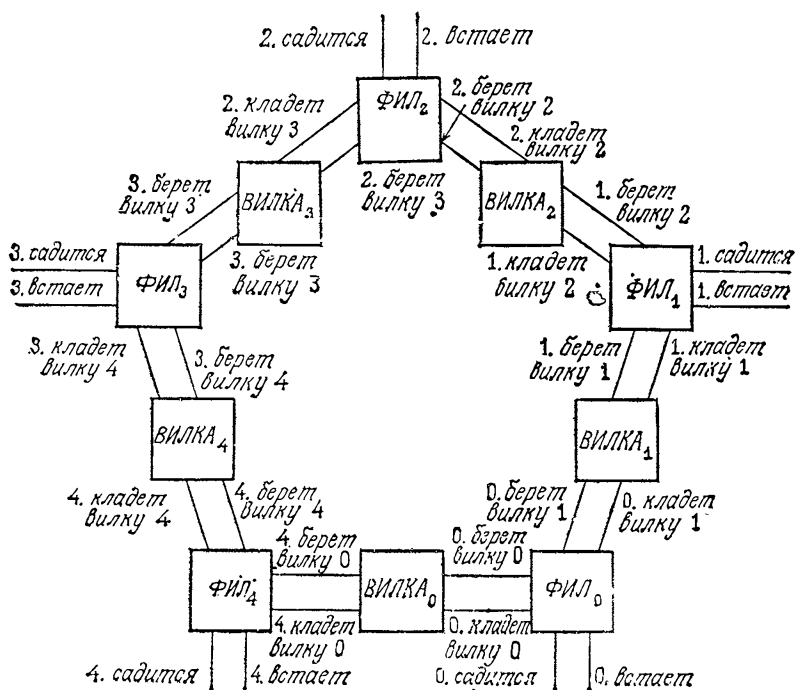


Рис. 2.6

Таким образом, каждое событие, кроме *садится* и *встает*, требует участия ровно двух соседних действующих лиц — философа и вилки, как показано на диаграмме связей на рис. 2.6.

2.5.2. Поведение

Помимо процессов размышления и еды, которыми мы решили пренебречь, жизнь каждого философа представляет собой повторение цикла из шести событий:

$$\begin{aligned} \text{ФИЛ}_i = & (i.\text{садится} \rightarrow i.\text{берет вилок}.i \rightarrow \\ & i.\text{берет вилок}.(i \oplus 1) \rightarrow i.\text{кладет вилок}.i \rightarrow \\ & i.\text{кладет вилок}.(i \oplus 1) \rightarrow i.\text{встает} \rightarrow \text{ФИЛ}_i) \end{aligned}$$

Роль вилки проста: ее неоднократно берет и кладет на место кто-нибудь из соседних с ней философов:

$$\begin{aligned} \text{ВИЛКА}_i = & (i.\text{берет вилок}.i \rightarrow i.\text{кладет вилок}.i \rightarrow \text{ВИЛКА}_i \\ & | (i \ominus 1).\text{берет вилок}.i \rightarrow \\ & \rightarrow (i \ominus 1).\text{кладет вилок}.i \rightarrow \text{ВИЛКА}_i) \end{aligned}$$

Поведение всего пансиона можно описать как параллельную комбинацию поведения следующих компонент:

$$\begin{aligned}\text{ФИЛОСОФЫ} &= (\text{ФИЛ}_0 \parallel \text{ФИЛ}_1 \parallel \text{ФИЛ}_2 \parallel \text{ФИЛ}_3 \parallel \text{ФИЛ}_4) \\ \text{ВИЛКИ} &= (\text{ВИЛКА}_0 \parallel \text{ВИЛКА}_1 \parallel \text{ВИЛКА}_2 \parallel \text{ВИЛКА}_3 \parallel \text{ВИЛКА}_4) \\ \text{ПАНСИОН} &= (\text{ФИЛОСОФЫ} \parallel \text{ВИЛКИ})\end{aligned}$$

В одном из интересных вариантов этой истории философы могут брать и класть обе вилки в любом порядке. Рассмотрим поведение каждой руки философа в отдельности. Каждая рука может самостоятельно взять соответствующую вилку, но для того, чтобы сесть или встать, требуются обе руки:

$$\begin{aligned}a\text{ЛЕВАЯ}_i &= \{i.\text{берет ви́лку}.i, i.\text{кладет ви́лку}.i, \\ &\quad i.\text{садится}, i.\text{встает}\} \\ a\text{ПРАВАЯ}_i &= \{i.\text{берет ви́лку}.(i \oplus 1), i.\text{кладет ви́лку}.(i \oplus 1), \\ &\quad i.\text{садится}, i.\text{встает}\} \\ \text{ЛЕВАЯ}_i &= (i.\text{садится} \rightarrow i.\text{берет ви́лку}.i \rightarrow \\ &\quad i.\text{кладет ви́лку}.i \rightarrow i.\text{встает} \rightarrow \text{ЛЕВАЯ}_i) \\ \text{ПРАВАЯ}_i &= (i.\text{садится} \rightarrow i.\text{берет ви́лку}.(i \oplus 1) \rightarrow \\ &\quad i.\text{кладет ви́лку}.(i \oplus 1) \rightarrow i.\text{встает} \rightarrow \text{ПРАВАЯ}_i) \\ \text{ФИЛ}_i &= \text{ЛЕВАЯ}_i \parallel \text{ПРАВАЯ}_i\end{aligned}$$

Синхронизацией процессов ЛЕВАЯ_i и ПРАВАЯ_i в момент, когда философ садится или встает, достигается то, что вилка не может быть поднята, если соответствующий философ не сидит за столом. Помимо этого, операции с обеими вилками произвольно чередуются.

В еще одном варианте этой истории философ, находясь за столом, может поднимать и опускать каждую вилку многократно. Для этого придется изменить поведение обеих рук, введя итерацию, например:

$$\begin{aligned}\text{ЛЕВАЯ}_i &= (i.\text{садится} \rightarrow \\ &\quad \mu X.(i.\text{берет ви́лку}.i \rightarrow i.\text{кладет ви́лку}.i \rightarrow X \\ &\quad \mid i.\text{встает} \rightarrow \text{ЛЕВАЯ}_i))\end{aligned}$$

2.5.3. Дедлок!

Построенная математическая модель позволяет обнаружить серьезную опасность. Предположим, что все философы проголодаются примерно в одно и то же время. Все они сядут, все возьмут в левую руку вилку, все потянутся за второй — которой уже нет на месте. В столь недостойной ситуации всех их неизбежно ждет голодная смерть. Несмотря на то, что каждый из участников способен к дальнейшим действиям, ни одна пара участников не в состоянии договориться о том, какое действие совершить следующим.

Конец нашей истории, однако, не столь печален. Как только опасность была обнаружена, было предложено множество способов ее избежать. Один из философов, например, всегда может брать сначала правую вилку — если бы только они сумели договориться, кто из них будет это делать! По тем же причинам исключается покупка одной дополнительной вилки; покупка же пяти новых вилок обошлась бы слишком дорого.

Окончательным решением проблемы явилось появление слуги, в чьи обязанности входило сопровождать каждого философа за стол и из-за стола. Алфавит его был определен как

$$\bigcup_{i=0}^4 \{i.\text{садится}, i.\text{встает}\}$$

Слуге было дано секретное указание следить за тем, чтобы за столом никогда не оказывалось больше четырех философов одновременно. Его поведение проще всего описать с помощью взаимной рекурсии.

Пусть

$$B = \bigcup_{i=0}^4 \{i.\text{встает}\}, C = \bigcup_{i=0}^4 \{i.\text{садится}\}$$

$СЛУГА_j$ определяет поведение слуги, когда за столом сидят j философов:

$$СЛУГА_0 = (x : C \rightarrow СЛУГА_1)$$

$$СЛУГА_j = (x : C \rightarrow СЛУГА_{j+1} \mid y : B \rightarrow СЛУГА_{j-1}) \quad \text{для } j \in \{1, 2, 3\}$$

$$СЛУГА_4 = (y : B \rightarrow СЛУГА_3)$$

Пансион, свободный от дедлока, определяется как

$$НОВПАНСИОН = (ПАНСИОН \parallel СЛУГА_0)$$

Эта поучительная история об обедающих философах принадлежит Эдсгеру В. Дейкстре. Слуга появился благодаря Карелу С. Шолтену.

2.5.4. Доказательство беступиковости

В первоначальном $ПАНСИОНе$ риск тупиковой ситуации был далеко не очевиден, и поэтому утверждение, что $НОВПАНСИОН$ свободен от дедлоков, следует доказывать с известной аккуратностью. То, что мы хотим доказать, строго можно сформулировать как

$$(НОВПАНСИОН/s) \neq \text{СТОП} \quad \text{для всех} \\ s \in \text{протоколы}(НОВПАНСИОН)$$

Для доказательства возьмем произвольный протокол s и покажем, что в любом случае найдется хотя бы одно событие, которым можно продолжить s и по-прежнему получить протокол *НОВПАНСИОН*. Сначала определим число сидящих философов:

$$\text{сидят}(s) = \#(s \upharpoonright C) - \#(s \upharpoonright B), \text{ где}$$

C и B определены ранее. Так как, согласно закону 2.3.3,

$$s \upharpoonright (B \cup C) \in \text{протоколы}(\text{СЛУГА}_0)$$

мы знаем, что

$$\text{сидят}(s) \leq 4$$

Если $\text{сидят}(s) \leq 3$, то мы знаем, что еще по крайней мере один философ может сесть и это не приведет к дедлоку. В оставшемся случае, когда $\text{сидят}(s) = 4$, рассмотрим число философов, которые едят (т. е. обе их вилки подняты). Если это число не равно нулю, то такой философ всегда может положить левую вилку. Если же никто из философов не ест, рассмотрим число поднятых вилок. Если оно меньше или равно трем, то один из сидящих философов по-прежнему может поднять левую вилку. Если поднято четыре вилки, это значит, что философ, сидящий рядом со свободным местом, уже поднял левую вилку и может взять правую. Если же поднятых вилок пять, это означает, что по крайней мере один из философов уже ест.

Это доказательство состоит из рассмотрения множества различных случаев, неформально описанных в терминах нашего конкретного примера. Рассмотрим другой способ доказательства: составление машинной программы для исследования поведения системы в поисках дедлока. В общем случае мы никогда не сможем узнать, достаточно ли проработала программа, чтобы гарантировать отсутствие тупиковых ситуаций. Но если система имеет конечное число состояний, как в нашем случае *ПАНСИОН*, достаточно рассмотреть только те протоколы, длина которых не превышает известной верхней границы числа состояний. Число состояний $(P \parallel Q)$ не превышает произведения числа состояний P и числа состояний Q . Так как каждый философ имеет шесть, а каждая вилка — три состояния, общее число состояний системы *ПАНСИОН* не превышает $6^5 \times 3^5$, или приблизительно 1,8 млн. Так как алфавит слуги содержится в алфавите *ПАНСИОН*, число состояний *НОВПАНСИОНа* не может превышать числа состояний *ПАНСИОНа*. Поскольку почти каждое состояние содержит два или более возможных события, то число протоколов, которые надо проверить, будет превышать два в сте-

пени 1,8 млн. Весьма маловероятно, что компьютер когда-нибудь сумеет исследовать все возможные случаи. Доказательство отсутствия тупиковых ситуаций даже для весьма простых конечных процессов, таким образом, по-прежнему будет входить в обязанности разработчика параллельных систем.

2.5.5. Бесконечный перехват

Помимо дедлока обедающего философа подстерегает другая опасность, а именно опасность бесконечного перехвата инициативы соседом. Предположим, что левая рука у сидящего философа весьма медлительна, в то время как его левый сосед в высшей степени проворен. Прежде чем философ дотянется до своей левой вилки, его левый сосед быстро вступает в дело, садится, хватает обе вилки и долго ест. Рано или поздно наевшись, он кладет обе вилки на стол и покидает свое место. Но стоит ему встать, как он снова ощущает голод, возвращается к столу, садится, мгновенно хватается обе вилки — и все это прежде, чем его медлительный, долгосидящий и изнывающий от голода правый сосед дотянется до их общей вилки. Поскольку такого рода цикл может повторяться неограниченно, может оказаться, что сидящий философ никогда не приступит к еде.

Похоже, что при строгом подходе эта проблема неразрешима, поскольку если каждый философ столь жаден и проворен, как было описано, то неизбежно кто-нибудь (или он, или его сосед) очень долгое время будет оставаться голодным. В такой ситуации не видно эффективного пути к достижению общего удовлетворения, кроме как приобрести достаточное количество вилок и побольше спагетти.

Однако, если все-таки важно гарантировать, чтобы сидящий философ смог в конце концов поесть, нужно изменить поведение слуги: проводив философа до его места, он ждет, пока философ не возьмет обе вилки, и только после этого позволяет сесть его соседям.

Однако в отношении бесконечного перехвата остается более философская проблема. Предположим, что слуга испытывает иррациональную неприязнь к одному из философов и постоянно затягивает исполнение своей обязанности проводить его к столу, даже если философ к этому готов. Это ситуация, которую мы не можем описать в нашей концептуальной схеме, поскольку она неотличима от той, когда философу требуется бесконечно длительное время, чтобы проголодаться. Таким образом, возникает проблема, аналогичная проблемам детальной синхронизации, которыми, как вы помните, мы

сознательно решили пренебречь или, более точно, отложить на другие стадии проектирования и реализации. Таким образом, добиваться, чтобы любое желаемое и возможное событие обязательно наступило в пределах разумного интервала времени, становится задачей realizатора. Это аналогично требованию, предъявляемому realizатору обычного языка программирования высокого уровня, а именно: не создавать произвольных задержек в выполнении программы, хотя программист не имеет возможности ни обеспечить, ни даже описать это требование.

2.6. ПЕРЕИМЕНОВАНИЕ

В примере из предыдущего раздела мы имели дело с двумя группами процессов — философами и вилками; внутри каждой группы процессы вели себя очень похоже с той разницей, что события, в которых они участвовали, имели различные имена. В настоящем разделе мы описываем удобный способ задания группы процессов, обладающих сходным поведением. Пусть f — взаимно однозначная (инъективная) функция, отображающая алфавит процесса P во множество символов A :

$$f : \alpha P \rightarrow A$$

Определим $f(P)$ как процесс, участвующий в событии $f(c)$ всякий раз, когда P по определению участвует в событии c . Из определения следует, что

$$\begin{aligned} \alpha f(P) &= f(\alpha P) \\ \text{протоколы}(f(P)) &= \{f^*(s) \mid s \in \text{протоколы}(P)\} \end{aligned}$$

(Определение f^* см. в разд. 1.9.1.)

Примеры

X1. Как известно, с годами стоимость жизни растет. Функцию f , отражающую последствия инфляции, зададим с помощью уравнений

$$\begin{aligned} f(n2) &= n10 & f(\text{бол}) &= \text{бол} \\ f(n1) &= n5 & f(\text{мал}) &= \text{мал} \\ f(c\delta 1) &= c\delta 5 \end{aligned}$$

Новый торговый автомат

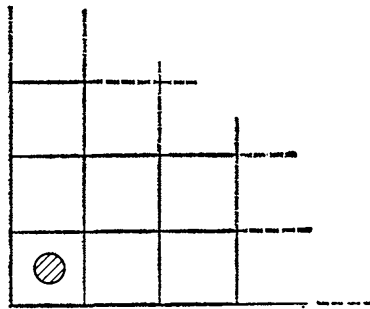
$$\text{НОВТАС} = f(\text{ТАС})$$

X2. Фишка ведет себя как CT_0 (1.1.4 X2), только вместо *вверх* и *вниз* она совершает движения *вправо* и *влево*.

$$f(\text{вверх}) = \text{вправо}, f(\text{вниз}) = \text{влево}, f(\text{вокруг}) = \text{вокруг} \\ \text{ЛП}_0 = f(CT_0)$$

В основном подобное переименование событий делается для того, чтобы использовать полученные процессы в параллельной комбинации друг с другом.

X3. Фишка двигается *вверх*, *вниз*, *влево* и *вправо* по бесконечному полю, ограниченному слева и снизу.



Движение начинается из левого нижнего угла поля. Только из этой клетки фишка может двигаться *вокруг*. Как и в примере 2.3 X3, вертикальные и горизонтальные перемещения можно промоделировать независимыми событиями различных процессов, однако движение *вокруг* требует одновременного участия обоих процессов.

$$\text{ЛПВН} = \text{ЛП}_0 \parallel CT_0.$$

X4. Мы хотим соединить два экземпляра процесса *КОПИБИТ* (1.1.3 X7) в цепь, чтобы каждый выходной бит первого процесса одновременно был входом для второго. Сначала нам придется переименовать события, использующиеся для внутреннего сообщения; введем поэтому два новых события, *средн. 0* и *средн. 1*, и определим функции f и g для изменения выхода одного процесса и входа другого:

$$\begin{aligned} f(\text{выв.0}) &= g(\text{вв.0}) = \text{средн.0} \\ f(\text{выв.1}) &= g(\text{вв.1}) = \text{средн.1} \\ f(\text{вв.0}) &= \text{вв.0}, f(\text{вв.1}) = \text{вв.1} \\ g(\text{выв.0}) &= \text{выв.0}, g(\text{выв.1}) = \text{выв.1} \end{aligned}$$

Требуемый результат получаем как

$$\text{ЦЕПЬ2} = f(\text{КОПИБИТ}) \parallel g(\text{КОПИБИТ})$$

Заметим, что по определению f и g каждый вывод 0 или 1 левым операндом и ввод тех же 0 и 1 правым операндом представляют собой единое событие *средн. 0* или *средн. 1*. Так моделируется синхронизованный обмен двоичными цифрами по каналу, соединяющему два операнда, как показано на рис. 2.7,

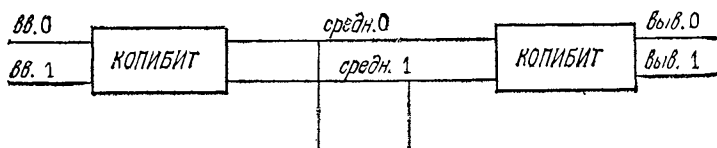


Рис. 2.7

Левый операнд не выбирает значение, которое передается по соединяющему каналу, тогда как правый операнд готов и к событию *средн. 0*, и к событию *средн. 1*. Поэтому лишь процесс вывода в каждом случае определяет, которое из этих двух событий произойдет. Этот способ взаимодействия между параллельными процессами будет обобщен в гл. 4.

Заметим, что внутренние сообщения *средн. 0* и *средн. 1*, оставаясь в алфавите составного процесса, доступны для наблюдения (и, возможно, контроля) со стороны его окружения. Иногда такие внутренние события желают не принимать во внимание или делать недоступными; в общем случае это может приводить к недетерминизму, и поэтому мы отложим этот вопрос до разд. 3.5.

Х5. Мы хотим представить поведение логической переменной, использующейся в программе. Ее алфавит содержит события

- присв0* присваивание переменной значения 0,
- присв1* присваивание переменной значения 1,
- выб0* взятие нулевого значения переменной,
- выб1* взятие значения переменной, равного единице.

Поведение этой переменной удивительно похоже на поведение автомата с газировкой (пример 1.1.4 **X1**), и поэтому определим

$$\text{ЛОГ} = f(\text{АГАЗ}), \text{ где}$$

определение f мы оставляем в качестве элементарного упражнения. Заметим, что значение нашей переменной не может быть считано прежде, чем произойдет первое присваивание этой переменной. Попытка взятия значения неопределенной переменной приведет к тупиковой ситуации—пожалуй, луч-

шему, что может произойти с неверной программой, ибо простейшая «посмертная» выдача точно укажет на эту ошибку.

Древовидное представление $f(P)$ можно построить по древовидному представлению P , применив функцию f к меткам на всех дугах. Так как функция f взаимно однозначна, это преобразование сохраняет структуру дерева и существенное

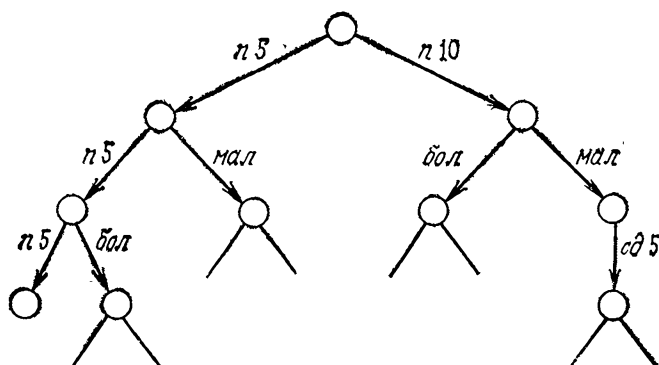


Рис. 2.8

свойство — различие всех меток на дугах, исходящих из одной вершины. Дерево для *НОВТАС*, к примеру, выглядит, как показано на рис. 2.8.

2.6.1. Законы

Переименование путем применения взаимно однозначной функции не меняет структуры поведения процесса. Это отражает тот факт, что применяемая функция дистрибутивна относительно всех остальных операций, о чем гласят приведенные ниже законы. Будем использовать следующие вспомогательные определения:

$$f(B) = \{f(x) \mid x \in B\}$$

$$f^{-1} \text{ функция, обратная к } f$$

$$f \circ g \text{ композиция функций } f \text{ и } g$$

$$f^* \text{ определена в разд. 1.9.1}$$

(использование в законах f^{-1} является важной причиной нашего требования инъективности f).

После переименования *СТОП* по-прежнему не участвует ни в одном из событий своего нового алфавита:

$$L1. f(СТОП_A) = СТОП_{f(A)}$$

В конструкции выбора меняются символы из множества первоначального выбора, и аналогично изменяется последующее поведение:

$$\mathbf{L2.} \quad f(x : B \rightarrow P(x)) = (y : f(B) \rightarrow f(P(f^{-1}(y))))$$

Могут понадобиться разъяснения по поводу использования в правой части функции f^{-1} . Вспомним, что P — это функция, выдающая процесс в зависимости от выбора некоторого x из множества B . Но переменная y в правой части выбрана из множества $f(B)$. Соответствующим событием для P будет $f^{-1}(y)$, принадлежащее B (так как $y \in f(B)$). После этого события поведение P есть $P(f^{-1}(y))$, и ко всем действиям, которые совершает этот процесс, по-прежнему должна применяться функция f .

Переименование дистрибутивно относительно оператора параллельной композиции:

$$\mathbf{L3.} \quad f(P \parallel Q) = f(P) \parallel f(Q)$$

Дистрибутивность относительно оператора рекурсии осуществляется чуть сложнее; при этом соответственно изменяется алфавит:

$$\mathbf{L4.} \quad f(\mu X : A. F(X)) = (\mu Y : f(A). f(F(f^{-1}(Y))))$$

Опять может вызвать недоумение использование f^{-1} в правой части. Вспомним, что для того, чтобы рекурсивное определение в левой части было корректным, требуется, чтобы аргументом функции F был процесс с алфавитом A , а результатом — процесс с тем же алфавитом. В правой части переменная Y принимает значения процессов с алфавитом $f(A)$ и не может поэтому использоваться в качестве аргумента F , пока ее алфавит не будет заменен снова на A . Это делается применением обратной функции f^{-1} . Теперь $F(f^{-1}(Y))$ имеет алфавит A , и поэтому применение функции f изменит этот алфавит на $f(A)$, обеспечив таким образом справедливость рекурсии в правой части.

Композиция двух переименований определяется как композиция двух переименовующих функций:

$$\mathbf{L5.} \quad f(g(P)) = (f \circ g)(P)$$

Протоколы процесса после переименования получаются простой заменой отдельных символов во всех протоколах исходного процесса:

$$\mathbf{L6.} \quad \text{протоколы}(f(P)) = \{f^*(s) \mid s \in \text{протоколы}(P)\}$$

Пояснение к следующему и последнему закону аналогично пояснению к L6.

$$L7. f(P)/f^*(s) = f(P/s)$$

2.6.2. Помеченные процессы

Переименование особенно полезно при создании групп сходных процессов, которые в режиме параллельной работы предоставляют некоторые идентичные услуги их общему окружению, но никак не взаимодействуют друг с другом. Это означает, что все они должны иметь различные и взаимно непересекающиеся алфавиты. Чтобы этого достичь, снабдим каждый процесс отдельным именем; каждое событие помеченного процесса помечено тем же именем. Помеченное событие — это пара $l.x$, где l — метка, а x — имя события.

Процесс P с меткой l обозначают $l:P$. Он участвует в событии $l.x$, когда по определению P участвует в x . Процесс $l:P$ задается функцией

$$f_l(x) = l.x \text{ для всех } x \text{ из } P,$$

и пометкой

$$l:P = f_l(P).$$

Примеры

X1. Два работающих рядом торговых автомата

$$(лев : ТАП) \parallel (прав : ТАП)$$

Алфавиты этих процессов не пересекаются, и каждое происходящее событие помечено именем того устройства, на котором оно происходит. Если перед их параллельной установкой устройства не получили имен, каждое событие будет требовать участия их обоих, и тогда пара машин будет неотличима от единственного устройства; это является следствием того, что $(ТАП \parallel ТАП) = ТАП$.

Пометка позволяет использовать процессы наподобие локальных переменных в языке высокого уровня, описанных в том блоке программы, где они используются.

X2. Поведение логической переменной моделируется процессом ЛОГ (2.6 X5). Поведение блока программы представимо процессом ПОЛБЗ. Этот процесс присваивает и считывает значения двух логических переменных b и c . Поэтому его алфавит $\alpha ПОЛБЗ$ состоит из таких событий, как

$b.присво$ присвоить b значение 0

$c.выб1$ выбрать текущее значение c , когда оно равно 1

Процесс *ПОЛЪЗ* и обе его логические переменные работают параллельно:

$$b : \text{ЛОГ} \parallel c : \text{ЛОГ} \parallel \text{ПОЛЪЗ}$$

В программе *ПОЛЪЗ* можно задать следующие действия:

$$\begin{aligned} b &:= \text{ложь}; P \text{ — посредством } (b.\text{присв}0 \rightarrow P), \\ b &:= \top c; \quad P \text{ — посредством } (c.\text{выб}0 \rightarrow b.\text{присв}1 \rightarrow P \\ &\quad | c.\text{выб}1 \rightarrow b.\text{присв}0 \rightarrow P) \end{aligned}$$

Заметим, что для того, чтобы выяснить текущее значение переменной, используется выбор между событиями *выб0* и *выб1*; результат этого выбора соответствующим образом влияет на последующее поведение процесса *ПОЛЪЗ*.

В *X2* и последующих примерах было бы удобнее определять эффект одного присваивания, например:

$$b := \text{ложь},$$

нежели пары команд

$$b := \text{ложь}; P$$

где имеется явное упоминание остатка программы *P*. Это может быть достигнуто с помощью средств, описанных в гл. 5.

X3. Процессу *ПОЛЪЗ* требуются два счетчика — переменные *l* и *m*. Их начальные значения равны 0 и 3 соответственно. Процесс *ПОЛЪЗ* может увеличивать эти переменные с помощью событий *l.вверх* и *m.вверх* и уменьшать (когда они положительны) с помощью *l.вниз* и *m.вниз*. Проверка на нуль осуществляется событиями *l.вокруг* и *m.вокруг*. Отсюда следует, что мы можем использовать процесс *СТ* (1.1.4 *X2*), пометив его соответственно *l* и *m*:

$$(l : \text{СТ}_0 \parallel m : \text{СТ}_3 \parallel \text{ПОЛЪЗ})$$

В ходе процесса *ПОЛЪЗ* могут быть осуществлены следующие действия (выраженные в обычных обозначениях):

$$\begin{aligned} (m &:= m + 1; P) \text{ — посредством } (m.\text{вверх} \rightarrow P), \\ \text{if } l = 0 \text{ then } P \text{ else } Q &\text{ — применением } (l.\text{вокруг} \rightarrow P | \\ &\quad l.\text{вниз} \rightarrow l.\text{вверх} \rightarrow Q) \end{aligned}$$

Обратите внимание, как работает проверка на нуль: *l.вниз* пытается уменьшить счетчик на единицу, и одновременно производится попытка выполнить *l.вокруг*. Счетчик выбирает одно из этих событий: если его величина равна нулю — событие *l.вокруг*, иначе — другую альтернативу. Но в последнем случае величина счетчика уменьшается на еди-

ницу и поэтому должна быть немедленно восстановлена с помощью *l. вверх*. В последующем примере восстановление начального значения более трудоемко.

Конструкция $(m := m + 1; P)$ реализуется рекурсивно определенным процессом *СЛОЖ*:

$$\text{СЛОЖ} = \text{ВНИЗ}_0$$

где $\text{ВНИЗ}_i = (l.\text{вниз} \rightarrow \text{ВНИЗ}_{i+1} \mid l.\text{вокруг} \rightarrow \text{ВВЕРХ}_i)$
 $\text{ВВЕРХ}_0 = P$

а $\text{ВВЕРХ}_{i+1} = l.\text{вверх} \rightarrow m.\text{вверх} \rightarrow \text{ВВЕРХ}_i$

Процесс ВНИЗ_i выясняет начальное значение *l*, уменьшая его до нуля. Затем процесс ВВЕРХ_i прибавляет полученное значение к *m* и к *l*, восстанавливая тем самым исходное значение *l* и прибавляя это значение к *m*.

Переменную-массив можно представить набором параллельных процессов, каждый из которых помечен его индексом в массиве.

Х4. Задача процесса *НД* состоит в том, чтобы регистрировать, происходило ли ранее событие *v* или нет. При первом наступлении *v* процесс отвечает *нет*, а при всех последующих — *да*.

$$\alpha \text{НД} = \{v, \text{нет}, \text{да}\}$$

$$\text{НД} = v \rightarrow \text{нет} \rightarrow \mu X. (v \rightarrow \text{да} \rightarrow X)$$

Массив из этих процессов можно использовать для имитации поведения цифрового множества:

$$\text{МНОЖЗ} = (0 : \text{НД}) \parallel (1 : \text{НД}) \parallel (2 : \text{НД}) \parallel (3 : \text{НД})$$

Перед использованием весь массив также может быть помечен:

$$m : \text{МНОЖЗ} \parallel \text{ПОЛЬЗ}$$

Каждое событие из алфавита $\alpha(m : \text{МНОЖЗ})$ представляет собой тройку, например *m.2.v*. В ходе процесса *ПОЛЬЗ* эффект конструкции

$$\text{if } 2 \in m \text{ then } P \text{ else } (m := m \cup \{2\}; Q)$$

может быть достигнут посредством

$$m.2.v \rightarrow (m.2.\text{да} \rightarrow P \mid m.2.\text{нет} \rightarrow Q)$$

2.6.3. Реализация

Для реализации переименования в общем случае требуется знать функцию *g* — обратную к переименовающей функции *f*. Кроме того, необходимо обеспечить, чтобы *g* выдавала

специальный символ "BLEEP в случае, когда ее аргумент выходит за область значений f . Реализация основана на законе 2.6.1 L4.

$$\begin{aligned} \text{переименование}(g, P) = \lambda x. & \text{ if } g(x) = \text{"BLEEP then "BLEEP} \\ & \text{ else if } P(g(x)) = \text{"BLEEP} \\ & \quad \text{then "BLEEP} \\ & \quad \text{else переименова-} \\ & \quad \text{ние}(g, P(g(x))) \end{aligned}$$

Пометка процессов реализуется проще. Составное событие $l.x$ представляется парой атомов $\text{cons}(\text{"}l, \text{"}x)$. Помеченный процесс $(l : P)$ реализуется функцией

$$\begin{aligned} \text{пометка}(l, P) = \lambda y. & \text{ if } \text{null}(y) \vee \text{atom}(y) \text{ then "BLEEP} \\ & \text{ else if } \text{car}(y) \neq l \text{ then "BLEEP} \\ & \text{ else if } P(\text{cdr}(y)) = \text{"BLEEP then "BLEEP} \\ & \quad \text{else пометка} \\ & \quad (l, P(\text{cdr}(y))) \end{aligned}$$

2.6.4. Множественная пометка

Определение пометки можно расширить, позволив пометить каждое событие любой меткой l из некоторого множества L . Если P — процесс, определим $(L : P)$ как процесс, ведущий себя в точности как P с той разницей, что он участвует в событии $l.c$ (где $l \in L$, а $c \in \alpha P$), если по определению P участвует в c . Выбор метки l каждый раз независимо осуществляется окружением процесса $(L : P)$.

Пример

X1. Лакей — это младший слуга, который имеет одного хозяина, провожает его к столу и из-за стола и прислуживает ему, пока тот ест:

$$\begin{aligned} \alpha \text{ЛАКЕЙ} &= \{\text{садится, встает}\} \\ \text{ЛАКЕЙ} &= (\text{садится} \rightarrow \text{встает} \rightarrow \text{ЛАКЕЙ}) \end{aligned}$$

Чтобы научить лакея обслуживать всех пятерых философов (но только по очереди), определим

$$\begin{aligned} L &= \{0, 1, 2, 3, 4\} \\ \text{ОБЩИЙ ЛАКЕЙ} &= (L : \text{ЛАКЕЙ}) \end{aligned}$$

Общего лакея можно нанимать на период отпуска слуги (2.5.3) для предохранения обедающих философов от тупиковой ситуации. Конечно, в течение этого времени философам

придется поголодать, ибо находиться за столом они смогут только по очереди.

Если множество L содержит более одной метки, древовидное представление $L:P$ похоже на древовидное представление P , однако оно гораздо гуще в том смысле, что из каж-



Рис. 2.9

дой вершины исходит гораздо больше дуг. Дерево процесса ЛАКЕЙ, например, представляет собой ствол без ветвей

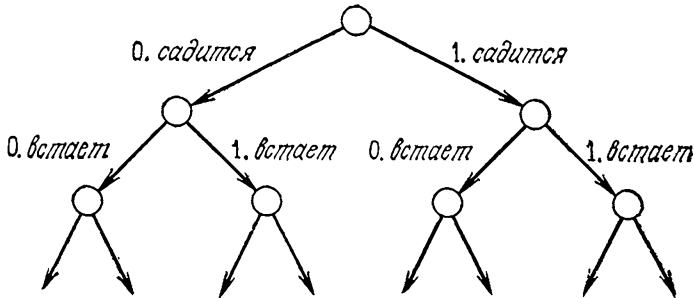


Рис. 2.10

(рис. 2.9). Представлением же процесса $\{0, 1\} : \text{ЛАКЕЙ}$ будет полное двоичное дерево (рис. 2.10). Дерево для процесса ОБЩИЙ ЛАКЕЙ будет еще гуще.

В общем случае множественную пометку можно использовать для распределения услуг одного процесса между некоторым числом других помеченных процессов при условии, что множество меток известно заранее. Более полно эта техника развивается в гл. 6.

2.7. СПЕЦИФИКАЦИИ

Пусть P и Q — процессы, которые мы хотим исполнять параллельно, и предположим, что мы доказали, что P уд $S(nr)$, а Q уд $T(nr)$. Пусть nr — протокол процесса $(P \parallel Q)$. Из закона 2.3.3 L1 следует, что $(nr \upharpoonright \alpha P)$ является протоколом P и, следовательно, удовлетворяет S , т. е. $S(nr \upharpoonright \alpha P)$. Аналогично $(nr \upharpoonright \alpha Q)$ является протоколом Q и, значит, $T(nr \upharpoonright \alpha Q)$. Это справедливо для каждого протокола $(P \parallel Q)$. Следовательно, можно заключить, что

$$(P \parallel Q) \text{ уд } (S(nr \upharpoonright \alpha P) \& T(nr \upharpoonright \alpha Q))$$

Как следствие этих нестрогих рассуждений можно сформулировать закон

L1. Если P уд $S(nr)$, а Q уд $T(nr)$,
то $(P \parallel Q) \text{ уд } (S(nr \upharpoonright \alpha P) \& T(nr \upharpoonright \alpha Q))$.

Пример

X1. (см. 2.3.1 X1)

Пусть $\alpha P = \{a, c\}$, $\alpha Q = \{b, c\}$, $P = (a \rightarrow c \rightarrow P)$, $Q = (c \rightarrow b \rightarrow Q)$.

Мы хотим доказать, что

$$(P \parallel Q) \text{ уд } (0 \leq nr \downarrow a - nr \downarrow b \leq 2).$$

Чтобы показать, что

$$P \text{ уд } (0 \leq nr \downarrow a - nr \downarrow c \leq 1),$$

$$\text{а } Q \text{ уд } (0 \leq nr \downarrow c - nr \downarrow b \leq 1),$$

можно очевидным образом использовать доказательство из примера 1.10.2 X1. Отсюда по L1 следует, что

$$\begin{aligned} (P \parallel Q) \text{ уд } (0 \leq (nr \upharpoonright \alpha P) \downarrow a - (nr \upharpoonright \alpha P) \downarrow c \leq 1 \& 0 \leq \\ (nr \upharpoonright \alpha Q) \downarrow c - (nr \upharpoonright \alpha Q) \downarrow b \leq 1) \\ \Rightarrow 0 \leq nr \downarrow a - nr \downarrow b \leq 2, \text{ так как } (nr \upharpoonright A) \downarrow a = nr \downarrow a, \\ \text{если } a \in A \end{aligned}$$

Поскольку, согласно законам для отношения уд, *СТОП* удовлетворяет любой выполнимой спецификации, рассуждениями, основанными на этих законах, доказать отсутствие дедлока нельзя. В разд. 3.7 будут даны более сильные законы. Пока же единственным способом исключить риск остановки является тщательное доказательство, как и в разд. 2.5.4. Другой способ — это показать, что процесс, определенный с помощью оператора параллельной комбинации, эквивалентен бесконечному процессу, определенному без этого оператора, как делалось в примере 2.3.1 X1. Такие доказательства, однако, требуют очень трудоемких алгебраиче-

ских преобразований. Где это возможно, следует прибегать к помощи какого-либо общего закона, как, например,

L2. Если процессы P и Q бесконечны, а $(\alpha P \cap \alpha Q)$ содержит не более одного события, то процесс $(P \parallel Q)$ бесконечен.

Пример

X2. Процесс $(P \parallel Q)$, определенный в **X1**, никогда не останавливается, потому что

$$\alpha P \cap \alpha Q = \{c\}.$$

Правило доказательства для переименования имеет вид

L3. Если P уд $S(np)$, то $f(P)$ уд $S(f^{-1*}(np))$.

Использование функции f^{-1*} в последующем члене этого закона может потребовать разъяснений. Пусть np — протокол $f(P)$. Тогда $f^{-1*}(np)$ — протокол P . Предыдущий член **L3** утверждает, что всякий протокол P удовлетворяет S . Следовательно, $f^{-1*}(np)$ удовлетворяет S , что в точности соответствует утверждению последующего члена **L3**.

2.8. МАТЕМАТИЧЕСКАЯ ТЕОРИЯ ДЕТЕРМИНИРОВАННЫХ ПРОЦЕССОВ

При описании процессов мы сформулировали множество законов и время от времени использовали их в доказательствах. Если мы и давали этим законам некоторое обоснование, то только в форме нестрогих разъяснений, почему мы предполагаем или хотим, чтобы эти законы выполнялись. Для читателя, обладающего чутьем математика-прикладника или инженера, этого может быть достаточно. Но возникает вопрос: а действительно ли справедливы эти законы? Является ли их набор хотя бы непротиворечивым? Должно ли их быть больше, или они полны в том смысле, что с их помощью можно доказать любой истинный факт о процессах? Можно ли обойтись меньшим числом более простых законов? Ответы на эти вопросы следует искать в более глубоком математическом исследовании.

2.8.1. Основные определения

При создании математической модели некоторой физической системы хороша стратегия, при которой определение основных понятий ведется в терминах свойств и признаков, поддающихся прямому или косвенному наблюдению или

измерению. Для детерминированного процесса P нам известны два таких свойства:

- αP множество событий, в которых процесс в принципе может участвовать;
протоколы (P) множество всех последовательностей событий, в которых процесс действительно может участвовать, если потребуется.

На эти два множества налагаются условия, которые уже были сформулированы в законах 1.8.1 L6, L7, L8. Рассмотрим теперь произвольную пару множеств (A, S) , удовлетворяющую этим трем законам. По этой паре можно единственным образом построить процесс P с множеством протоколов S согласно следующим определениям:

Пусть $P^0 = \{x \mid \langle x \rangle \in S\}$, а $P(x)$ — процесс с множеством протоколов $\{t \mid \langle x \rangle \hat{=} t \in S\}$ для всех x из P^0 .

Тогда $\alpha P = A$, а $P = (x : P^0 \rightarrow P(x))$.

Кроме того, уравнения

$$A = \alpha P$$

$$S = \text{протоколы}(x : P^0 \rightarrow P(x))$$

позволяют по P восстановить пару (A, S) . Таким образом, между каждым процессом P и парой множеств $(\alpha P, \text{протоколы}(P))$ существует взаимно однозначное соответствие. В математике такое использование одного понятия для определения другого служит достаточным основанием для их последующего отождествления.

D0. Детерминированный процесс — это пара (A, S) , где A — произвольное множество символов, а S — любое подмножество A , удовлетворяющее двум условиям:

$$\text{C0. } \langle \rangle \in S$$

$$\text{C1. } \forall s, t. s \hat{=} t \in S \Rightarrow s \in S.$$

Простейшим примером процесса, отвечающего этому определению, служит *СТОП* — процесс, который ничего не делает:

$$\text{D1. } \text{СТОП}_A = (A, \{\langle \rangle\})$$

В другом крайнем случае — это процесс, всегда готовый выполнить любое действие:

$$\text{D2. } \text{ИСП}_A = (A, A^*)$$

Теперь мы можем дать формальные определения различных операций над процессами, описав, как по алфавиту и

протоколам операндов строятся алфавит и протоколы результата.

D3. $(x : B \rightarrow (A, S(x))) = (A, \{\langle \rangle\} \cup \{\langle x \rangle^s \mid x \in B \ \& \ s \in S(x)\})$
при условии, что $B \subseteq A$;

D4. $(A, S)/s = (A, \{t \mid (s^{\wedge} t) \in S\})$ при условии, что $s \in S$;

D5. $\mu X : A.F(X) = (A, \bigcup_{n \geq 0} \text{протоколы } (F^n(\text{СТОП}_A)))$
при условии, что выражение F — предваренное;

D6. $(A, S) \parallel (B, T) = (A \cup B, \{s \mid s \in (A \cup B)^* \ \& \ (s \upharpoonright A) \in S \ \& \ (s \upharpoonright B) \in T\})$.

D7. $f(A, S) = (f(A), \{f^*(s) \mid s \in S\})$
при условии, что f взаимно однозначна.

Разумеется, надо доказать, что правые части этих определений действительно являются процессами, т. е. удовлетворяют условиям **C0** и **C1** в определении **D0**. К счастью, сделать это несложно.

В гл. 3 станет очевидно, что **D0** — не вполне достаточное определение понятия процесса, поскольку в нем никак не представлена возможность недетерминированного поведения. Следовательно, нам потребуется более общее и более сложное определение. Все законы для недетерминированных процессов справедливы также и для детерминированных. Но детерминированные процессы подчиняются, кроме того, некоторым дополнительным законам, например $P \parallel P = P$. Во избежание путаницы такие законы мы в книге не помечаем; таким образом, все помеченные законы применимы как к детерминированным, так и к недетерминированным процессам.

2.8.2. Теория неподвижной точки

Целью этого раздела является доказательство в общих чертах основополагающей теоремы о рекурсии, т. е. о том, что рекурсивно определенный процесс (2.8.1 **D5**) в действительности является решением соответствующего рекурсивного уравнения

$$\mu X.F(X) = F(\mu X.F(X))$$

Наш подход будет основан на теории неподвижной точки по Скотту.

Прежде всего мы должны ввести на множестве процессов отношение порядка:

D1. $(A, S) \sqsubseteq (B, T) = (A = B \ \& \ S \subseteq T)$

При таком упорядочении два процесса сравнимы, если они имеют общий алфавит, и один из них может делать все то же, что и другой, и, возможно, больше. Это отношение является частичным порядком в том смысле, что

$$L1. P \sqsubseteq P$$

$$L2. P \sqsubseteq Q \ \& \ Q \sqsubseteq P \Rightarrow P = Q$$

$$L3. P \sqsubseteq Q \ \& \ Q \sqsubseteq R \Rightarrow P \sqsubseteq R$$

Цепью в частичном упорядочении называется бесконечная последовательность элементов $\{P_0, P_1, P_2, \dots\}$, такая, что $P_i \sqsubseteq P_{i+1}$ для всех i . *Предел* (наименьшую верхнюю грань) такой цепи определим как

$$\bigsqcup_{i \geq 0} P_i = (\alpha P_0, \bigsqcup_{i \geq 0} \text{протоколы}(P_i))$$

В дальнейшем мы будем применять предельный оператор \bigsqcup только к тем последовательностям процессов, которые образуют цепь.

Будем говорить, что частичный порядок *полный*, если в нем имеется наименьший элемент, а все цепи имеют наименьшую верхнюю грань. Множество всех процессов над данным алфавитом A образует полный частичный порядок (п. ч. п.), так как оно удовлетворяет следующим законам:

$$L4. \text{СТОП}_A \sqsubseteq P \text{ при условии, что } \alpha P = A$$

$$L5. P_i \sqsubseteq \bigsqcup_{i \geq 0} P_i$$

$$L6. (\forall i \geq 0. P_i \sqsubseteq Q) \Rightarrow \left(\bigsqcup_{i \geq 0} P_i \right) \sqsubseteq Q$$

Кроме того, в терминах предела можно переформулировать и определение μ -оператора (2.8.1 D5):

$$L7. \mu X : A. F(X) = \bigsqcup_{i \geq 0} F^i(\text{СТОП}_A)$$

Говорят, что функция F из одного п. ч. п. в другой (или в тот же) *непрерывна*, если она дистрибутивна относительно пределов всех цепей, т. е.

$$F\left(\bigsqcup_{i \geq 0} P_i\right) = \bigsqcup_{i \geq 0} F(P_i), \text{ если } \{P_i \mid i \geq 0\} \text{ образует цепь.}$$

(Все непрерывные функции монотонны в том смысле, что $P \sqsubseteq Q \Rightarrow F(P) \sqsubseteq F(Q)$ для всех P и Q , и поэтому правая часть предыдущего равенства является пределом возрастающей цепи.) По определению функция G от нескольких аргументов

непрерывна, если она непрерывна по каждому аргументу в отдельности, например:

$$G\left(\left(\bigsqcup_{i \geq 0} P_i\right), Q\right) = \bigsqcup_{i \geq 0} G(P_i, Q) \text{ для всех } Q \text{ и}$$

$$G\left(Q, \left(\bigsqcup_{i \geq 0} P_i\right)\right) = \bigsqcup_{i \geq 0} G(Q, P_i) \text{ для всех } Q.$$

Композиция непрерывных функций также непрерывна; и, конечно же, любое выражение, полученное применением любого числа непрерывных функций к любому числу и комбинации переменных, непрерывно по каждой из этих переменных. Например, если G , F и H непрерывны, то $G(F(X), H(X, Y))$ непрерывна по X , т. е.

$$G\left(F\left(\bigsqcup_{i \geq 0} P_i\right), H\left(\left(\bigsqcup_{i \geq 0} P_i\right), Y\right)\right) = \bigsqcup_{i \geq 0} G(F(P_i), H(P_i, Y))$$

для всех Y .

Все определенные в **D3** — **D7** операции (кроме $/$) непрерывны в вышеописанном смысле:

$$\text{L8. } (x : B \rightarrow \left(\bigsqcup_{i \geq 0} P_i(x)\right)) = \bigsqcup_{i \geq 0} (x : B \rightarrow P_i(x))$$

$$\text{L9. } \mu X : A. F(X, \left(\bigsqcup_{i \geq 0} P_i\right)) = \bigsqcup_{i \geq 0} \mu X : A. F(X, P_i) \text{ при условии, что } F \text{ непрерывна}$$

$$\text{L10. } \left(\bigsqcup_{i \geq 0} P_i\right) \parallel Q = Q \parallel \left(\bigsqcup_{i \geq 0} P_i\right) = \bigsqcup_{i \geq 0} (Q \parallel P_i)$$

$$\text{L11. } f\left(\bigsqcup_{i \geq 0} P_i\right) = \bigsqcup_{i \geq 0} f(P_i)$$

Таким образом, если выражение $F(X)$ построено в терминах только этих операций, оно будет непрерывным по X . Теперь можно доказать основную теорему о неподвижной точке:

$$\begin{aligned} F(\mu X : A. F(X)) &= F\left(\bigsqcup_{i \geq 0} (F^i(\text{СТОП}_A))\right) \text{ по определению } \mu \\ &= \bigsqcup_{i \geq 0} F(F^i(\text{СТОП}_A)) \text{ непрерывность } F \\ &= \bigsqcup_{i \geq 0} F^i(\text{СТОП}_A) \text{ по определению } F^{i+1} \\ &= \bigsqcup_{i \geq 0} F^i(\text{СТОП}_A) \text{ } \text{СТОП}_A \equiv F(\text{СТОП}_A) \\ &= \mu X : A. F(X) \text{ по определению } \mu \end{aligned}$$

Это доказательство основано на том, что F непрерывна. Предваренность F необходима лишь для доказательства единственности решения.

2.8.3. Единственность решения

В этом разделе мы несколько формализуем наши рассуждения из разд. 1.1.2, показавшие, что уравнение, задающее процесс с помощью предваренной рекурсии, имеет единственное решение. При этом нам удастся сформулировать более общие условия единственности таких решений. Для простоты мы будем рассматривать процессы, заданные только одним рекурсивным уравнением; эти рассуждения можно легко обобщить на случай систем взаимно рекурсивных уравнений.

Если P — процесс, а n — натуральное число, то определим $(P \upharpoonright n)$ как процесс, который ведет себя как P в течение первых n событий, а затем останавливается; более строго:

$$(A, S) \upharpoonright n = (A, \{s \mid s \in S \ \& \ \# s \leq n\})$$

Отсюда следует, что

$$L1. P \upharpoonright 0 = \text{СТОП}$$

$$L2. P \upharpoonright n \sqsubseteq P \upharpoonright (n + 1) \sqsubseteq P$$

$$L3. P = \bigsqcup_{n \geq 0} P \upharpoonright n$$

$$L4. \bigsqcup_{n \geq 0} P_n = \bigsqcup_{n \geq 0} (P_n \upharpoonright n)$$

Пусть F — монотонная функция из множества процессов во множество процессов. Будем говорить, что F *конструктивна*, если

$$F(X) \upharpoonright (n + 1) = F(X \upharpoonright n) \upharpoonright (n + 1) \quad \text{для всех } X$$

Это означает, что поведение $F(X)$ на $n + 1$ первом шаге определяется поведением X только на n первых шагах; таким образом, если $s \neq \langle \rangle$, то

$$s \in \text{протоколы}(F(X)) \iff s \in \text{протоколы}(F(X \upharpoonright (\# s - 1)))$$

Простейшим примером конструктивной функции служит префиксация, поскольку

$$(c \rightarrow P) \upharpoonright (n + 1) = (c \rightarrow (P \upharpoonright n)) \upharpoonright (n + 1)$$

Обобщенный выбор также конструктивен:

$$(x : B \rightarrow P(x)) \upharpoonright (n + 1) = (x : B \rightarrow (P(x) \upharpoonright n)) \upharpoonright (n + 1)$$

Тождественная функция I не конструктивна, поскольку

$$\begin{aligned} I(c \rightarrow P) \upharpoonright 1 &= c \rightarrow \text{СТОП} \\ &\neq \text{СТОП} \\ &= I((c \rightarrow P) \upharpoonright 0) \upharpoonright 1 \end{aligned}$$

Теперь можно сформулировать основную теорему.

L5. Пусть F — конструктивная функция. Тогда уравнение $X = F(X)$ имеет единственное решение для X .

Доказательство. Пусть X — произвольное решение. Сначала докажем по индукции лемму о том, что

$$X \upharpoonright n = F^n(\text{СТОП}) \upharpoonright n.$$

$$\begin{aligned} \text{Основание индукции. } X \upharpoonright 0 &= \text{СТОП} = \text{СТОП} \upharpoonright 0 = \\ &= F^0(\text{СТОП}) \upharpoonright 0 \end{aligned}$$

Шаг индукции.

$$\begin{aligned} X \upharpoonright (n+1) &= F(X) \upharpoonright (n+1) && \text{так как } X = F(X) \\ &= F(X \upharpoonright n) \upharpoonright (n+1) && F \text{ конструктивна} \\ &= F(F^n(\text{СТОП}) \upharpoonright n) \upharpoonright (n+1) && \text{предположение} \\ & && \text{индукции} \\ &= F(F^n(\text{СТОП})) \upharpoonright (n+1) && F \text{ конструктивна} \\ &= F^{n+1}(\text{СТОП}) \upharpoonright (n+1) && \text{по определению } F^n \end{aligned}$$

Теперь вернемся к основной теореме.

$$\begin{aligned} X &= \bigsqcup_{n \geq 0} (X \upharpoonright n) && \mathbf{L3} \\ &= \bigsqcup_{n \geq 0} F^n(\text{СТОП}) \upharpoonright n && \text{согласно лемме} \\ &= \bigsqcup_{n \geq 0} F^n(\text{СТОП}) && \mathbf{L4} \\ &= \mu X.F(X) && 2.8.2 \mathbf{L7} \end{aligned}$$

Таким образом, все решения уравнения $X = F(X)$ совпадают и равны $\mu X.F(X)$; другими словами, $\mu X.F(X)$ является единственным решением уравнения.

Полезное значение этой теоремы сильно возрастет, если мы научимся четко определять, какие функции являются конструктивными, а какие — нет. Назовем функцию G неструктивной, если она удовлетворяет условию

$$G(P) \upharpoonright n = G(P \upharpoonright n) \upharpoonright n \quad \text{для всех } n \text{ и } P.$$

Переименование в этом смысле неструктивно, поскольку

$$f(P) \upharpoonright n = f(P \upharpoonright n) \upharpoonright n$$

Это же справедливо и для тождественной функции. Любая монотонная конструктивная функция является также и неструктивной. Операция *после*, однако, деструктивна, поскольку

$$\begin{aligned} ((c \rightarrow c \rightarrow \text{СТОП}) / \langle c \rangle) \upharpoonright 1 &= c \rightarrow \text{СТОП} \\ &\neq \text{СТОП} \\ &= (c \rightarrow \text{СТОП}) / \langle c \rangle \\ &= (((c \rightarrow c \rightarrow \text{СТОП}) \upharpoonright 1) / \langle c \rangle) \upharpoonright 1. \end{aligned}$$

Любая композиция недеструктивных функций (G и H) также недеструктивна, потому что

$$G(H(P)) \vdash n = G(H(P) \vdash n) \vdash n = G(H(P \vdash n) \vdash n) \vdash n = \\ = G(H(P \vdash n)) \vdash n$$

Что еще важнее, любая композиция конструктивной функции с недеструктивными функциями является конструктивной. Поэтому если F, G, \dots, H — недеструктивные функции и хотя бы одна из них конструктивна, то $F(G(\dots(H(X))\dots))$ — конструктивная функция от X .

Приведенные рассуждения легко распространить на функции более чем одного аргумента. Так, например, параллельная композиция недеструктивна (по обоим аргументам), потому что

$$(P \parallel Q) \vdash n = ((P \vdash n) \parallel (Q \vdash n)) \vdash n$$

Пусть E — выражение, содержащее в качестве переменной процесс P . Говорят, что E конструктивно по X , если к каждому вхождению X в E применяется некоторая конструктивная функция и не применяется никакая деструктивная функция. Так, следующее выражение является конструктивным по X :

$$(c \rightarrow X \mid d \rightarrow f(X \parallel P) \mid e \rightarrow (f(X) \parallel Q)) \parallel ((d \rightarrow X) \parallel R)$$

Важным следствием этого определения является то, что теперь конструктивность можно синтаксически определить с помощью следующих условий предваренности:

D0. Будем говорить, что выражение сохраняет предваренность, если оно построено только с помощью операторов параллелизма, переименования и обобщенного выбора.

D1. Будем говорить, что выражение, не содержащее X , предварено по X .

D2. Обобщенный выбор $(x: B \rightarrow P(X, x))$ предварен по X , если $P(X, x)$ сохраняет предваренность для всех x .

D3. Переименование $f(P(X))$ предварено по X , если $P(X)$ предварено по X .

D4. Параллельная система $P(X) \parallel Q(X)$ предварена по X , если как $P(X)$, так и $Q(X)$ предварены по X .

Окончательно мы заключаем, что

L6. Если E предварено по X , то уравнение $X = E$ имеет единственное решение.

Глава 3. Недетерминизм

3.1. ВВЕДЕНИЕ

Оператор выбора ($x: B \rightarrow P(x)$) используется для задания процесса, обладающего целым спектром возможного поведения; параллельный оператор \parallel позволяет некоторому другому процессу делать выбор между альтернативами из множества B . Так, например, автомат по размену денег РАЗМ5С (1.1.3 X2) позволяет получить на выбор либо три маленькие монеты и одну большую, либо две большие монеты и одну маленькую. Такие процессы называются *детерминированными*, потому что в том случае, когда возможно наступление более чем одного события, выбор между ними определяется внешней по отношению к процессу обстановкой. Это происходит либо в том смысле, что окружение процесса может в действительности осуществлять этот выбор, либо в более слабом смысле, а именно: выбор становится известен окружению в тот самый момент, когда он происходит.

Иногда процесс обладает некоторым спектром возможного поведения, но его окружение не имеет возможности ни влиять на выбор между различными альтернативами, ни даже наблюдать его. Так, например, некоторый отличный от нашего автомат по размену денег может давать сдачу в виде любой из описанных выше комбинаций; однако клиент не может ни повлиять на этот выбор, ни даже предсказать его. Выбор осуществляется самой машиной произвольным или недетерминированным образом. Этот выбор нельзя ни проконтролировать, ни пронаблюдать; нельзя даже в точности выяснить, когда он был сделан; однако заключение о том, какое именно событие произошло, можно сделать позже на основании дальнейшего поведения процесса.

В этой разновидности недетерминизма нет ничего загадочного: он возникает вследствие сознательного решения не принимать во внимание факторы, определяющие выбор. Вариант комбинации сдачи, выдаваемой автоматом, может определяться, например, тем, как автомат был заполнен большими и маленькими монетами; эти события, однако, не были включены в алфавит. Таким образом, недетерминизм полезен для достижения высокого уровня абстракции при описании поведения физических систем и машин.

3.2. НЕДЕТЕРМИНИРОВАННЫЙ ВЫБОР

Если P и Q — процессы, введем обозначение

$$P \sqcap Q \quad (P \text{ или } Q)$$

для процесса, который ведет себя или как P , или как Q , причем выбор между ними осуществляется произвольно и без ведома или контроля со стороны внешнего окружения. Предполагается, что алфавиты операндов совпадают:

$$\alpha(P \sqcap Q) = \alpha P = \alpha Q$$

Примеры

X1. Автомат по размену денег, дающий сдачу в виде одной из двух комбинаций:

$$\begin{aligned} \text{РАЗМ5D} = & (n5 \rightarrow ((c\partial 1 \rightarrow c\partial 1 \rightarrow c\partial 1 \rightarrow c\partial 2 \rightarrow \text{РАЗМ5D}) \\ & \sqcap (c\partial 2 \rightarrow c\partial 1 \rightarrow c\partial 2 \rightarrow \text{РАЗМ5D}))) \end{aligned}$$

X2. При различных обращениях к автомату РАЗМ5D он может давать различные варианты комбинации сдачи. Приведем пример машины, которая всегда дает один и тот же вариант сдачи, первоначально, однако, неизвестный (см. 1.1.2 X3, X4):

$$\text{РАЗМ5E} = \text{РАЗМ5A} \sqcap \text{РАЗМ5B}$$

Разумеется, что после того, как эта машина сдаст первую же монету, дальнейшее ее поведение полностью предсказуемо. Поэтому

$$\text{РАЗМ5D} \neq \text{РАЗМ5E}$$

С помощью двуместного оператора \sqcap мы ввели недетерминизм в его наиболее чистом и простом виде. Мы, конечно, не предполагаем, что оператор \sqcap может быть полезным при реализации процесса. Было бы крайне неразумно, построив оба процесса P и Q , спрятать их в черный ящик, затем тянуть наугад один из них и выбросить оставшийся! Как правило, преимущества недетерминизма используются при спецификации процесса. Процесс, описанный как $(P \sqcap Q)$, может быть реализован или как P , или как Q . Выбор может быть сделан разработчиком заранее и из соображений, не имеющих никакого отношения к спецификации (и сознательно в ней не указанных), таких, как низкая стоимость, хорошее время реакции или высокая скорость передачи сообщения.

На самом деле, даже в спецификациях оператор \sqcap будет использоваться не часто; более естественно недетерминизм возникает при употреблении других операторов, которые будут описаны в этой главе.

3.2.1. Законы

Алгебраические законы, управляющие недетерминированным выбором, исключительно просты и очевидны. Выбор между P и P несуществен:

$$\text{L1. } P \sqcap P = P \quad (\text{идемпотентность})$$

Порядок записи членов не имеет значения:

$$\text{L2. } P \sqcap Q = Q \sqcap P \quad (\text{симметричность})$$

Выбор между тремя альтернативами можно разбить на два последовательных двуместных выбора. Порядок, в котором это делается, не имеет значения:

$$\text{L3. } P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R \quad (\text{ассоциативность})$$

Момент недетерминированного выбора тоже несуществен. Процесс, сначала выполняющий x , а затем делающий выбор, неотличим от процесса, который сначала делает выбор, а затем выполняет x :

$$\text{L4. } x \rightarrow (P \sqcap Q) = (x \rightarrow P) \sqcap (x \rightarrow Q) \quad (\text{дистрибутивность})$$

Закон L4 утверждает, что операция префиксации дистрибутивна относительно операции недетерминированного выбора. Такие операции будем называть просто *дистрибутивными*. Двуместная операция называется дистрибутивной, если она дистрибутивна относительно \sqcap по каждому аргументу в отдельности. Большинство уже введенных операций над процессами в этом смысле дистрибутивно:

$$\text{L5. } (x : B \rightarrow (P(x) \sqcap Q(x))) = (x : B \rightarrow P(x)) \sqcap (x : B \rightarrow Q(x))$$

$$\text{L6. } P \parallel (Q \sqcap R) = (P \parallel Q) \sqcap (P \parallel R)$$

$$\text{L7. } (P \sqcap Q) \parallel R = (P \parallel R) \sqcap (Q \parallel R)$$

$$\text{L8. } f(P \sqcap Q) = f(P) \sqcap f(Q)$$

Рекурсивный оператор, однако, не дистрибутивен, за исключением тривиального случая, когда операнды \sqcap совпадают. Это легко проиллюстрировать на примере разницы двух процессов:

$$P = \mu X. ((a \rightarrow X) \sqcap (b \rightarrow X))$$

$$Q = (\mu X. (a \rightarrow X)) \sqcap (\mu X. (b \rightarrow X))$$

Процесс P может выбирать между a и b независимо на каждой итерации, и поэтому множество его протоколов вклю-

чает и такой:

$$\langle a, b, b, a, b \rangle$$

Процесс Q должен раз и навсегда сделать выбор между a и b и поэтому не может иметь приведенного выше протокола. Однако P тоже может всегда выбирать только a или только b , и поэтому

$$\text{протоколы}(Q) \subseteq \text{протоколы}(P)$$

В некоторых теориях недетерминизм обязан быть *продуктивным* в том смысле, что если некоторое событие *может* произойти бесконечное число раз, то рано или поздно оно *должно* произойти (хотя не налагается никаких ограничений на то, как долго может откладываться его наступление). В нашей теории мы не придерживаемся этой концепции. Так как мы рассматриваем только конечные протоколы поведения процесса и если наступление события можно откладывать бесконечно, то мы не можем знать, наступит оно когда-нибудь или нет. Если же мы хотим, чтобы данное событие рано или поздно обязательно наступило, надо ввести требование о существовании такого числа n , что все протоколы длины более чем n содержат это событие. В этом случае при разработке процесса надо в явном виде включить в него описание этого ограничения. Например, в процессе P_0 , описанном ниже, событие a должно наступить в течение первых n шагов после своего предыдущего вхождения:

$$P_i = (a \rightarrow P_0) \sqcap (b \rightarrow P_{i+1}) \text{ для } i < n$$

$$P_n = (a \rightarrow P_0)$$

Позже мы убедимся, что как Q , так и P_0 являются допустимыми реализациями P .

Если требуется соблюсти условие продуктивности недетерминизма, его следует специфицировать и реализовать независимо, например приписав ненулевые вероятности альтернативам недетерминированного выбора. Нам кажется, что было бы очень полезно отделить сложные вероятностные рассуждения от соображений относительно логической правильности поведения процесса.

В свете законов L1—L3 полезно ввести операцию многократного выбора. Пусть $S = \{i, j, \dots, k\}$ — конечное непустое множество. Тогда положим

$$\bigsqcap_{x:S} P(x) = P(i) \sqcap P(j) \sqcap \dots \sqcap P(k).$$

Запись $\bigsqcap_{x:S}$ не имеет смысла, если множество S пусто или бесконечно.

3.2.2. Реализации

Как уже указывалось, одна из главных причин введения недетерминизма — это желание абстрагироваться от конкретных деталей реализации. Дело в том, что недетерминированный процесс P может иметь много реализаций, каждая из которых демонстрирует те или иные особенности поведения P . Эти различия порождаются разнообразием способов разрешения недетерминизма, присущего P . Соответствующий выбор может быть сделан еще до инициирования процесса, но может быть и отложен до стадии исполнения.

Например, одна из реализаций процесса $(P \sqcap Q)$ могла бы состоять в выборе первого операнда:

$$\text{или1}(P, Q) = P$$

Другую реализацию можно было бы получить, избрав второй операнд — возможно, из соображений большей эффективности на данной конкретной машине:

$$\text{или2}(P, Q) = Q$$

В третьей реализации принятие решения откладывалось бы до стадии исполнения процесса; выбор в этом случае предоставлялся бы окружению, выбирающему событие, возможное для одного процесса и невозможное для другого. Если же событие возможно для обоих процессов, то принятие решения снова откладывалось бы:

$$\begin{aligned} \text{или3}(P, Q) = \lambda x. \text{ if } P(x) = \text{"BLEEP"} \text{ then } Q(x) \\ \text{ else if } Q(x) = \text{"BLEEP"} \text{ then } P(x) \\ \text{ else } \text{или3}(P(x), Q(x)) \end{aligned}$$

Здесь мы привели только три возможные реализации одного и того же оператора. На самом деле их гораздо больше; например, реализация может вести себя в течение первых пяти шагов как *или3*, и если все эти шаги возможны как для P , так и для Q , в дальнейшем она произвольно выбирает P .

Поскольку разработчик процесса $(P \sqcap Q)$ не может предписать, какой процесс выбрать из P и Q , он должен гарантировать, что его система будет работать правильно в обоих случаях. Если же есть какой бы то ни было риск, что *либо* P , *либо* Q войдут в дедлок со своей обстановкой, то $(P \sqcap Q)$ несет в себе ту же самую степень риска. Реализация *или3* минимизирует риск дедлока, откладывая выбор до тех пор, пока его не сделает обстановка, и выбирая тот процесс из P и Q , который не создает дедлока. Благодаря этому свойству определение *или3* иногда называют *ангельским недетерминизмом*. Однако цена этому ангельскому поведению в терминах эф-

фективности высока: если выбор между P и Q не сделан на первом же шаге, P и Q должны исполняться параллельно до тех пор, пока обстановка не выберет событие, возможное для одного процесса и невозможное для другого. В простом, но крайнем случае $или3(P, P)$ такого события просто не найдется, что приведет к опять-таки крайней неэффективности. В отличие от $или3$ реализации $или1$ и $или2$ являются асимметричными в том смысле, что

$$или1(P, Q) \neq или1(Q, P)$$

Это может показаться нарушением закона 3.2.1 L1, однако это не так. Все законы применяются к процессам, а не к их конкретным реализациям. Фактически они представляют собой утверждения о равенстве множеств всех реализаций правых и левых частей. Например, так как $или3$ симметрична, то

$$\begin{aligned} \{или1(P, Q), или2(P, Q), или3(P, Q)\} &= \{P, Q, или3(P, Q)\} \\ &= \{или2(Q, P), или1(Q, P), \\ &\quad или3(Q, P)\} \end{aligned}$$

Одно из преимуществ, которые дает нам введение недетерминизма, — это возможность избегать потери симметрии, неизбежной в случае выбора одной из двух простых реализаций, и в то же время не прибегать к неэффективной симметричной реализации $или3$.

3.2.3. Протоколы

Если s является протоколом P , то он будет также возможным протоколом $(P \sqcap Q)$ (в случае если будет избран P). Аналогично если s — протокол Q , то он является и протоколом $(P \sqcap Q)$. И наоборот, каждый протокол процесса $(P \sqcap Q)$ должен быть протоколом одной или обеих альтернатив. Поведение $(P \sqcap Q)$ после s определяется тем, какой из процессов, P или Q , может выполнить s ; если же s — возможный протокол обоих процессов, то выбор остается недетерминированным.

$$\text{L1. } протоколы(P \sqcap Q) = протоколы(P) \cup протоколы(Q)$$

$$\begin{aligned} \text{L2. } (P \sqcap Q)/s &= Q/s && \text{если } s \in (протоколы(Q) - \\ & && \quad - протоколы(P)) \\ &= P/s && \text{если } s \in (протоколы(P) - \\ & && \quad - протоколы(Q)) \\ &= (P/s) \sqcap (Q/s) && \text{если } s \in (протоколы(P) \cap \\ & && \quad \cap протоколы(Q)) \end{aligned}$$

3.3. ГЕНЕРАЛЬНЫЙ ВЫБОР

Как уже говорилось, окружение процесса $(P \sqcap Q)$ не может не только влиять на выбор процесса, но даже не знает ни результата выбора между P и Q , ни времени, когда был сделан этот выбор. В связи с этим $(P \sqcap Q)$ оказывается не самым лучшим способом объединения процессов, поскольку обстановка должна быть готова к тому, чтобы работать как с P , так и с Q , в то время как работа лишь с одним из них в отдельности была бы, скорее всего, более простой. Для этого мы вводим еще одну операцию $(P \amalg Q)$, где обстановка уже *может* управлять выбором между P и Q при условии, что этот выбор будет сделан на первом же шаге. Если самое первое действие *не* возможно для P , то выбирается Q ; если Q не может выполнить первое действие, то выбирается P . Если же первое действие возможно как для P , так и для Q , то выбор между ними остается недетерминированным. (Разумеется, что если событие невозможно и для P , и для Q , то оно просто не может произойти.) Как обычно,

$$\alpha(P \amalg Q) = \alpha P = \alpha Q$$

В случае когда ни одно начальное событие P не является также возможным начальным событием Q , оператор генерального выбора аналогичен оператору, который до сих пор использовался для представления выбора между различными событиями:

$$(c \rightarrow P \amalg d \rightarrow Q) = (c \rightarrow P \mid d \rightarrow Q) \quad \text{если } c \neq d$$

При одинаковых же начальных событиях $(P \amalg Q)$ упрощается до недетерминированного выбора:

$$(c \rightarrow P \amalg c \rightarrow Q) = (c \rightarrow P) \sqcap (c \rightarrow Q)$$

Здесь мы ввели неявное соглашение, что операция \rightarrow связывает аргументы сильнее, чем \amalg .

3.3.1. Законы

Алгебраические законы для операции \amalg похожи на законы для \sqcap и справедливы по тем же причинам:

L1—L3. Операция \amalg идемпотентна, симметрична и ассоциативна.

L4. $P \amalg \text{СТОП} = P$

Следующие законы формализуют нестрогое определение операции:

$$\begin{aligned} \text{L5. } (x: A \rightarrow P(x)) \amalg (y: B \rightarrow Q(y)) = \\ = (z: (A \cup B) \rightarrow (\text{if } z \in (A - B) \text{ then } P(z) \\ \text{else if } z \in (B - A) \text{ then } Q(z) \\ \text{else if } z \in (A \cap B) \text{ then } (P(z) \sqcap Q(z))) \end{aligned}$$

Как и все уже введенные операции (кроме рекурсии), \amalg дистрибутивна относительно \sqcap :

$$\text{L6. } P \amalg (Q \sqcap R) = (P \amalg Q) \sqcap (P \amalg R)$$

Может показаться удивительным, но и \sqcap дистрибутивна относительно \amalg :

$$\text{L7. } P \sqcap (Q \amalg R) = (P \sqcap Q) \amalg (P \sqcap R)$$

В этом законе утверждается, что недетерминированный выбор и выбор, сделанный обстановкой, независимы в том смысле, что результат одного из них не влияет на результат другого. Пусть, например, Джон — это исполнитель, совершающий недетерминированный выбор, а Мэри — его окружение. В левой части закона Джон решает (\sqcap), выбрать ли ему P или же предоставить Мэри выбор (\amalg) между Q и R . В правой части Мэри решает:

- (1) предоставить ли Джону выбор между P и Q ,
- (2) или предоставить Джону выбор между P и R .

В обеих частях уравнения если Джон выбирает P , то это и будет окончательным результатом. Но если Джон не выбрал P , то выбор между Q и R совершает Мэри. Таким образом, обе стратегии выбора всегда приводят к одинаковым результатам. Эти же рассуждения, конечно, применимы и к закону L6.

Приведенные рассуждения являются довольно тонкими, и, пожалуй, было бы лучше вывести этот закон как неожиданное, но неизбежное следствие других, более очевидных определений и законов, которые будут даны в этой главе.

3.3.2. Реализация

Реализация операции выбора непосредственно вытекает из закона L5. В предположении симметричности или она тоже симметрична:

$$\begin{aligned} \text{выбор}(P, Q) = \lambda x. \text{ if } P(x) = \text{"BLEEP"} \text{ then } Q(x) \\ \text{else if } Q(x) = \text{"BLEEP"} \text{ then } P(x) \\ \text{else или } (P(x), Q(x)) \end{aligned}$$

3.3.3. Протоколы

Каждый протокол $(P \amalg Q)$ должен быть протоколом P или протоколом Q , и наоборот:

$$\text{L1. } \text{протоколы}(P \amalg Q) = \text{протоколы}(P) \cup \text{протоколы}(Q)$$

Следующий закон немного отличается от соответствующего закона для \sqcap :

$$\begin{aligned} \text{L2. } (P \amalg Q)/s &= P/s && \text{если } s \in \text{протоколы}(P) - \\ & && - \text{протоколы}(Q) \\ &= Q/s && \text{если } s \in \text{протоколы}(Q) - \\ & && - \text{протоколы}(P) \\ &= (P/s) \sqcap (Q/s) && \text{если } s \neq \langle \rangle \text{ и } s \in \text{протоколы}(P) \cap \\ & && \cap \text{протоколы}(Q) \end{aligned}$$

3.4. ОТКАЗЫ

Разница между процессами $(P \sqcap Q)$ и $(P \amalg Q)$ является весьма тонкой. Их нельзя различить по множествам протоколов, ибо протокол одного из них всегда возможен и для другого. Однако можно поместить эти процессы в такое окружение, с которым $(P \sqcap Q)$ на первом шаге войдет в дедлок, а $(P \amalg Q)$ нет. Пусть, например, $x \neq y$, а

$$P = (x \rightarrow P), \quad Q = (y \rightarrow Q), \quad \alpha P = \alpha Q = \{x, y\}.$$

$$\text{Тогда } (P \amalg Q) \parallel P = (x \rightarrow P) \\ = P$$

$$\text{а } (P \sqcap Q) \parallel P = (P \parallel P) \sqcap (Q \parallel P) \\ = P \sqcap \text{СТОП}$$

Отсюда видно, что в окружении P процесс $(P \sqcap Q)$ может прийти в тупиковое состояние, а $(P \amalg Q)$ нет. Конечно, о наступлении дедлока мы не можем знать наверняка даже относительно $(P \sqcap Q)$, и если он не произойдет, мы никогда не узнаем о его угрозе. Однако сама возможность наступления дедлока достаточна, чтобы различать $(P \amalg Q)$ и $(P \sqcap Q)$.

В общем случае, пусть X — множество событий, предлагаемых на первом шаге окружением процесса P , относительно которых в данном контексте мы предполагаем, что их алфавиты совпадают с алфавитом процесса P . Если при помещении P в эту обстановку существует риск дедлока на первом же шаге, мы говорим, что X — это *отказ* процесса P . Множество всех таких отказов обозначим $\text{отказы}(P)$. Заметим, что отказы процесса образуют семейство множеств символов. Это довольно неприятная сложность, но в данной концепции недетерминизма нам она представляется неизбежной. Может

показаться более естественным рассматривать вместо отказов множества символов, которые процесс *готов* принять; однако отказы оказываются несколько проще, поскольку они подчиняются законам **L9** и **L10** из разд. 3.4.1 (ниже), тогда как соответствующие законы для множеств готовности P были бы более сложными.

Введение понятия отказа позволяет четко определить формальное различие между детерминированными и недетерминированными процессами. Процесс называется *детерминированным*, если он никогда не отказывается от события, в котором может участвовать. Другими словами, множество является отказом детерминированного процесса, только если это множество не содержит событий, в которых процесс может участвовать на первом же шаге; более строго:

P детерминированный $\Rightarrow (X \in \text{отказы}(P) = (X \cap P^0 = \{ \})),$
где $P^0 = \{x \mid \langle x \rangle \in \text{протоколы}(P)\}$

Это условие применимо не только к начальному шагу процесса P , но и ко всем возможным последовательностям его действий. Поэтому можно определить:

P детерминированный \equiv
 $\equiv \forall s : \text{протоколы}(P). (X \in \text{отказы}(P/s) = (X \cap (P/s)^0 = \{ \}))$

Недетерминированным называется процесс, не обладающий этим свойством, т. е. может найтись такое событие, в котором процесс мог бы участвовать, но в то же время (в результате некоторого внутреннего недетерминированного выбора) он может отказаться от исполнения этого события, хотя обстановка ему не препятствует.

3.4.1. Законы

В приведенных ниже законах определяются отказы различных простых процессов. Процесс *СТОП* ничего не делает и отказывается ото всего:

L1. $\text{отказы}(\text{СТОП}_A) = \text{все подмножества множества } A \text{ (включая само } A)$

Процесс $(c \rightarrow P)$ отказывается от любого множества, не содержащего события c :

L2. $\text{отказы}(c \rightarrow P) = \{X \mid X \subseteq (\alpha P - \{c\})\}$

L3 является обобщением этих двух законов:

L3. $\text{отказы}(x : B \rightarrow P(x)) = \{X \mid X \subseteq (\alpha P - B)\}$

Если P отказывается от X , то так же поступает и $(P \sqcap Q)$ в случае выбора P . Аналогично каждый отказ процесса Q яв-

ляется возможным отказом процесса $(P \sqcap Q)$. Других отказов этот процесс не имеет, и поэтому

$$\text{L4. } \text{отказы}(P \sqcap Q) = \text{отказы}(P) \cup \text{отказы}(Q)$$

К процессу $(P \amalg Q)$ можно применить обратные рассуждения. Если X не является отказом процесса P , то P не может отказаться от X и, следовательно, не может отказаться от X и $(P \amalg Q)$. Аналогично если X не является отказом процесса Q , то он не является и отказом процесса $(P \amalg Q)$. Если же оба процесса P и Q могут отказаться от X , то и $(P \amalg Q)$ может отказаться от X :

$$\text{L5. } \text{отказы}(P \amalg Q) = \text{отказы}(P) \cap \text{отказы}(Q)$$

Сравнение законов L5 и L4 демонстрирует различие между \amalg и \sqcap .

Если P может отказаться от X , а Q может отказаться от Y , то их комбинация $(P \parallel Q)$ может отказаться как от X , так и от Y , т. е. от объединения множеств X и Y :

$$\text{L6. } \text{отказы}(P \parallel Q) = \{X \cup Y \mid X \in \text{отказы}(P) \ \& \ Y \in \text{отказы}(Q)\}$$

Закон для переименования очевиден:

$$\text{L7. } \text{отказы}(f(P)) = \{f(X) \mid X \in \text{отказы}(P)\}$$

Существует ряд общих законов, касающихся отказов. Процесс может отказываться только от событий из своего алфавита. Если обстановка не предлагает никаких событий, процесс приходит в тупиковое состояние; если же процесс отказывается от непустого множества, то он может отказаться и от любого его подмножества. И наконец, любое событие x , которое не может быть начальным, можно добавить к любому отказу X .

$$\text{L8. } X \in \text{отказы}(P) \Rightarrow X \subseteq \alpha P$$

$$\text{L9. } \{ \} \in \text{отказы}(P)$$

$$\text{L10. } (X \cup Y) \in \text{отказы}(P) \Rightarrow X \in \text{отказы}(P)$$

$$\text{L11. } X \in \text{отказы}(P) \Rightarrow (X \cup \{x\}) \in \text{отказы}(P) \vee \langle x \rangle \in \text{протоколы}(P)$$

3.5. СОКРЫТИЕ

Вообще говоря, алфавит процесса содержит лишь те события, которые мы сочли уместными и чье наступление требует одновременного участия обстановки. При описании внутреннего поведения механизма часто приходится иметь дело с событиями, отвечающими внутренним переходам этого механизма. Такие события могут отражать взаимодействия

и связи между параллельно работающими компонентами, из которых построен данный механизм, например ЦЕПЬ2 (2.6 X4) или пример 2.6.2 X3.

Создав механизм, мы делаем недоступной структуру его компонент; кроме того, мы хотим скрыть все внутренние действия этого механизма. Фактически мы хотим, чтобы эти действия происходили автоматически и в тот самый момент, когда для этого появляется возможность, и при этом были бы недоступны для контроля и наблюдения со стороны окружения процесса. Если C — конечное множество событий, которые мы хотим таким образом скрыть, то $P \setminus C$ обозначает процесс, который ведет себя как P с той разницей, что все события из C скрыты. Нашим очевидным намерением является равенство $\alpha(P \setminus C) = (\alpha P) - C$.

Примеры

X1. Шумный торговый автомат (2.3 X1) можно поместить в звуконепроницаемый кожух:

$$ТАШУМ \setminus \{\text{звук, щелк}\}$$

Неосуществленную способность этого автомата выдавать ириску тоже можно исключить из алфавита устройства, не изменив при этом его реального поведения. Полученный в результате процесс эквивалентен простому торговому автомату:

$$ТАП = ТАШУМ \setminus \{\text{звук, щелк, ирис}\}$$

Когда два процесса объединяются для параллельного исполнения, взаимодействия между ними обычно рассматриваются как внутренние действия результирующей системы; предполагается, что они происходят автономно, в первый же возможный момент и без ведома и вмешательства со стороны внешнего окружения системы. Таким образом, сокрытию подлежат все символы из пересечения алфавитов двух компонент.

X2. Пусть $\alpha P = \{a, c\}$, $\alpha Q = \{b, c\}$, $P = (a \rightarrow c \rightarrow P)$,
 $Q = (c \rightarrow b \rightarrow Q)$ (2.3.1 X1)

Действие c из алфавитов процессов P и Q рассматривается теперь как внутреннее и подлежит сокрытию:

$$\begin{aligned} (P \parallel Q) \setminus \{c\} &= (a \rightarrow c \rightarrow \mu X. (a \rightarrow b \rightarrow c \rightarrow X \\ &\quad | b \rightarrow a \rightarrow c \rightarrow X)) \setminus \{c\} \\ &= a \rightarrow \mu X. (a \rightarrow b \rightarrow X \\ &\quad | b \rightarrow a \rightarrow X) \end{aligned}$$

3.5.1. Законы

В первых законах утверждается, что сокрытие пустого множества символов ничего не дает и что безразлично, в каком порядке скрываются элементы множества. Остальные законы этой группы демонстрируют особенности дистрибутивности сокрытия относительно других операторов.

Если ничего не скрывается, то все остается явным:

$$L1. P \setminus \{ \} = P$$

Если сначала скрыть одно множество символов, а затем еще одно, то это равносильно их одновременному сокрытию:

$$L2. (P \setminus B) \setminus C = P \setminus (B \cup C)$$

Сокрытие дистрибутивно относительно недетерминированного выбора:

$$L3. (P \sqcap Q) \setminus C = (P \setminus C) \sqcap (Q \setminus C)$$

Сокрытие влияет только на алфавит процесса *СТОП* и не влияет на его поведение:

$$L4. \text{СТОП}_A \setminus C = \text{СТОП}_{A-C}$$

Цель сокрытия — дать событиям возможность происходить автоматически и мгновенно, но сделать их наступление полностью невидимым. Нескрытые события остаются без изменений:

$$L5. \begin{aligned} (x \rightarrow P) \setminus C &= x \rightarrow (P \setminus C) && \text{если } x \approx C \\ &= P \setminus C && \text{если } x \in C \end{aligned}$$

Если множество C содержит только такие события, в которых P и Q участвуют независимо, сокрытие C дистрибутивно относительно их параллельной композиции:

$$L6. \text{Если } \alpha P \sqcap \alpha Q \sqcap C = \{ \}, \text{ то } (P \parallel Q) \setminus C = (P \setminus C) \parallel (Q \setminus C).$$

Практическое значение этого закона невелико, потому что обычно требуется скрыть именно взаимодействия между параллельными процессами, т. е. события из $\alpha P \sqcap \alpha Q$, в которых процессы участвуют совместно.

Дистрибутивность сокрытия относительно взаимно однозначной функции переименования очевидна:

$$L7. f(P \setminus C) = f(P) \setminus f(C)$$

Если в конструкции выбора ни одно возможное начальное событие не скрывается, то множество начального выбора

остается прежним:

L8. Если $B \cap C = \{ \}$, то $(x : B \rightarrow P(x)) \setminus C = (x : B \rightarrow (P(x) \setminus C))$.

Так же как и оператор выбора \amalg , сокрытие событий может привести к появлению недетерминизма. Когда одновременно возможно наступление нескольких скрытых событий, не определяется, какое именно произойдет; в любом случае, однако, оно останется скрытым.

L9. Если $B \subseteq C$ и B конечно и непусто, то

$$(x : B \rightarrow P(x)) \setminus C = \prod_{x:B} (P(x) \setminus C)$$

В промежуточном случае, когда некоторые из начальных событий скрыты, а некоторые — нет, ситуация заметно осложняется. Рассмотрим процесс $(c \rightarrow P \mid d \rightarrow Q) \setminus C$, где $c \in C$, $d \notin C$. Скрытое событие c может произойти немедленно. В этом случае все поведение будет определяться процессом $(P \setminus C)$, и возможность наступления d сведется к нулю. Но достоверно предполагать, что d не произойдет, мы все-таки не можем. Если обстановка к этому готова, d вполне может произойти и раньше скрытого события, после чего уже скрытое событие c больше произойти не может. Но может случиться и так, что d произойдет, но будет исполнено процессом $(P \setminus C)$ уже после скрытого наступления c . В этом случае все поведение процесса можно описать как

$$(P \setminus C) \amalg (d \rightarrow (Q \setminus C))$$

Выбор между ним и $(P \setminus C)$ недетерминирован. Эти весьма закрученные рассуждения служат обоснованием довольно сложного закона:

$$(c \rightarrow P \mid d \rightarrow Q) \setminus C = (P \setminus C) \sqcap ((P \setminus C) \amalg (d \rightarrow (Q \setminus C)))$$

Аналогичными рассуждениями подкрепляется справедливость и более общего закона:

L10. Если $C \cap B \neq \{ \}$ и конечно, то

$$(x : B \rightarrow P(x)) \setminus C = Q \sqcap (Q \amalg (x : (B - C) \rightarrow P(x)))$$

где $Q = \prod_{x:B \cap C} P(x) \setminus C$.

Эти законы графически иллюстрируются в разд. 3.5.4.

Заметим, что оператор Π не дистрибутивен относительно $\setminus C$. В качестве контрпримера рассмотрим процесс

$$\begin{aligned}
 (c \rightarrow \text{СТОП} \Pi d \rightarrow \text{СТОП}) \setminus \{c\} &= \text{СТОП} \sqcap (\text{СТОП} \Pi (d \rightarrow \text{СТОП})) & \text{L10} \\
 &= \text{СТОП} \sqcap (d \rightarrow \text{СТОП}) & 3.3.1 \text{ L4} \\
 &\neq d \rightarrow \text{СТОП} \\
 &= \text{СТОП} \Pi (d \rightarrow \text{СТОП}) \\
 &= ((c \rightarrow \text{СТОП}) \setminus \{c\}) \Pi \\
 &\quad ((d \rightarrow \text{СТОП}) \setminus \{c\})
 \end{aligned}$$

Сокрытие уменьшает алфавит процесса. Но можно определить и такую операцию, которая расширяет алфавит процесса P , добавляя к нему символы из множества B :

$$\begin{aligned}
 \alpha(P_{+B}) &= \alpha P \cup B \\
 P_{+B} &= (P \parallel \text{СТОП}_B) \quad \text{при условии, что } B \cap \alpha P = \{ \}
 \end{aligned}$$

На самом деле ни одно из новых событий никогда не происходит, и поэтому поведение процесса P_{+B} в действительности совпадает с поведением P :

$$\text{L11. } \text{протоколы}(P_{+B}) = \text{протоколы}(P)$$

Сокрытие множества B обратно расширению алфавита множеством B :

$$\text{L12. } (P_{+B}) \setminus B = P$$

Здесь будет уместно затронуть одну проблему, решению которой посвящен разд. 3.8. В простейших случаях сокрытие дистрибутивно относительно рекурсии:

$$\begin{aligned}
 (\mu X: A.(c \rightarrow X)) \setminus \{c\} &= \mu X: (A - \{c\}).((c \rightarrow X_{+ \{c\}}) \setminus \{c\}) \\
 &= \mu X: (A - \{c\}).X & \text{по L12, L5}
 \end{aligned}$$

Таким образом, попытка сокрытия *бесконечной* непрерывной последовательности событий приводит к такому же неудачному результату, как и бесконечный цикл или непредваренная рекурсия. Все эти явления объединяются общим термином *расходимость*.

Та же проблема возникает, даже если расходящийся процесс может бесконечно часто выполнять некоторое нескрытое действие, например:

$$\begin{aligned}
 (\mu X.(c \rightarrow X \Pi d \rightarrow P)) \setminus \{c\} \\
 &= \mu X.((c \rightarrow X \Pi d \rightarrow P) \setminus \{c\}) \\
 &= \mu X.(X \setminus \{c\}) \sqcap ((X \setminus \{c\}) \Pi d \rightarrow (P \setminus \{c\})) & \text{по L10}
 \end{aligned}$$

Здесь мы опять видим, что рекурсия не предварена и приводит к расходимости. Даже несмотря на то, что обстановке,

казалось бы, бесконечно часто предоставляется шанс выбрать событие d , невозможно помешать процессу бесконечно часто выполнять вместо этого скрытое событие. Эта возможность, как нам представляется, позволяет достигать наивысшей производительности при реализации. Кроме того, она, похоже, имеет отношение к нашему решению не настаивать на продуктивности недетерминизма, что затрагивалось в разд. 3.2.1. Более строго явление расходимости рассматривается в разд. 3.8.

В некотором и немаловажном смысле упрятывание в действительности продуктивно. Пусть $d \in \alpha R$, и рассмотрим процесс

$$\begin{aligned} & ((c \rightarrow a \rightarrow P \mid d \rightarrow \text{СТОП}) \setminus \{c\}) \parallel (a \rightarrow R) \\ &= ((a \rightarrow P \setminus \{c\}) \sqcap (a \rightarrow P \setminus \{c\} \parallel d \rightarrow \text{СТОП})) \parallel (a \rightarrow R) \quad \text{L10} \\ &= (a \rightarrow P \setminus \{c\}) \parallel (a \rightarrow R) \sqcap (a \rightarrow P \setminus \{c\} \parallel d \rightarrow \text{СТОП}) \parallel (a \rightarrow R) \\ &= a \rightarrow ((P \setminus \{c\}) \parallel R) \end{aligned}$$

Этот пример показывает, что процесс, предлагающий на выбор спрятанное событие c и неспрятанное d , не может настаивать на том, чтобы произошло неспрятанное событие. Если обстановка (в нашем примере $a \rightarrow R$) не готова к d , то должно произойти спрятанное событие, чтобы дать обстановке возможность взаимодействовать с полученным в результате процессом (например, $(a \rightarrow P \setminus \{c\})$).

3.5.2. Реализация

Для простоты мы реализуем операцию, упрятывающую символы по одному:

$$\text{спрячь}(P, c) = P \setminus \{c\}$$

Множество из двух и более символов можно упрятать, пряча его символы по очереди, поскольку

$$P \setminus \{c_1, c_2, \dots, c_n\} = (\dots((P \setminus \{c_1\}) \setminus \{c_2\}) \dots) \setminus \{c_n\}$$

В простейшей реализации спрятанное событие происходит незаметно, всякий раз и в тот самый момент, когда это возможно:

$$\begin{aligned} \text{спрячь}(P, c) = & \text{if } P(c) = \text{"BLEEP"} \text{ then} \\ & (\lambda x. \text{if } P(x) = \text{"BLEEP"} \text{ then "BLEEP"} \\ & \quad \text{else спрячь}(P(x), c)) \\ & \text{else спрячь}(P(c), c) \end{aligned}$$

Давайте исследуем, что происходит, когда функция *спрячь* применяется к процессу, способному к участию в бесконечной

последовательности событий, например:

$$\text{спрячь}(\mu X.(c \rightarrow X \amalg d \rightarrow P), c)$$

В этом случае результатом проверки $(P(c) = \text{"BLEEP"})$ всегда будет значение *ложь*, и поэтому функция *спрячь* будет выполнять свою *else*-часть, т. е. немедленно вызывать себя рекурсивно. Выхода из этой рекурсии нет, и, таким образом, никакого взаимодействия с внешним миром не происходит. Это и есть расплата за попытку реализации расходящегося процесса.

Такая реализация сокрытия не подчиняется закону **L2**; действительно, как показывает приведенный ниже пример, в этом случае порядок, в котором упрятываются события, имеет значение.

Пусть $P = (c \rightarrow \text{СТОП} \mid d \rightarrow a \rightarrow \text{СТОП})$. Тогда

$$\begin{aligned} \text{спрячь}(\text{спрячь}(P, c), d) &= \text{спрячь}(\text{спрячь}(\text{СТОП}, c), d) \\ &= \text{СТОП} \end{aligned}$$

$$\begin{aligned} \text{а } \text{спрячь}(\text{спрячь}(P, d), c) &= \text{спрячь}(\text{спрячь}(a \rightarrow \text{СТОП}), d), c) \\ &= (a \rightarrow \text{СТОП}) \end{aligned}$$

Но, как уже говорилось в разд. 3.2.2, конкретная реализация недетерминированного оператора не обязана подчиняться законам. Достаточно, чтобы оба полученных нами результата были допустимыми реализациями одного и того же процесса

$$P \setminus \{c, d\} = (\text{СТОП} \sqcap (a \rightarrow \text{СТОП}))$$

3.5.3. Протоколы

Если t — протокол P , то соответствующий протокол $P \setminus C$ получается из t удалением всех вхождений символов из C . В свою очередь каждый протокол $P \setminus C$ должен был получиться из некоторого протокола P . Поэтому мы утверждаем, что

$$\mathbf{L1.} \text{ протоколы}(P \setminus C) = \{t \upharpoonright (\alpha P - C) \mid t : \text{протоколы}(P)\}$$

при условии, что $\forall s : \text{протоколы}(P). \neg \text{расходится}(P/s, C)$

Условие $\text{расходится}(P, C)$ означает, что при сокрытии C процесс P расходится, т. е. может участвовать в неограниченной последовательности скрытых действий. Поэтому определим

$$\text{расходится}(P, C) = \forall n. \exists s : \text{протоколы}(P) \cap C^*. \# s > n$$

Одному протоколу s процесса $P \setminus C$ могут соответствовать несколько протоколов t возможного поведения P , неразличимого после сокрытия, т. е. $t \upharpoonright (\alpha P - C) = s$. Следующий закон

утверждает, что выбор варианта поведения P , определяющего поведение $(P \setminus C)$ после s , не определен:

$$\text{L2. } (P \setminus C)/s = \left(\bigcap_{t:T} P/t \right) \setminus C,$$

где $T = \text{протоколы}(P) \cap \{t \mid t \uparrow (\alpha P - C) = s\}$,

при условии, что T конечно, а $s \in \text{протоколы}(P \setminus C)$.

Эти законы не распространяются на случаи расходящихся процессов. Ограничение это нельзя считать серьезным, поскольку при попытке определения процесса расхождимость никогда не является желаемым результатом. Более подробно вопросы расхождимости обсуждаются в разд. 3.8.

3.5.4. Рисунки

Графически недетерминированный выбор можно представить вершиной, из которой исходят две или более немеченные дуги; достигая этой вершины, процесс незаметно совершает переход по одной из этих дуг, причем выбор дуги осуществляется недетерминированно.

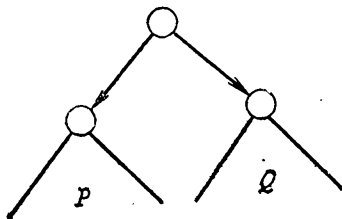


Рис. 3.1

На рис. 3.1 изображен процесс $P \sqcap Q$. Равенство графически представленных процессов устанавливается на основании алгебраических законов для недетерминизма; так, например, рис. 3.2 иллюстрирует ассоциативность операции \sqcap .

Соккрытие символов можно рассматривать как операцию, которая просто удаляет скрываемые символы со всех помеченных ими дуг, тем самым превращая их в немеченные дуги. При этом, естественно, возникает недетерминизм, как показано на рис. 3.3.

Каков же в таком случае смысл вершины, если некоторые из ее дуг помечены, а некоторые — нет? Ответ на это дает закон 3.3.1 L10. Такую вершину можно удалить, изменив графическое представление, как показано на рис. 3.4.

Достаточно очевидно, что для конечных деревьев такая замена всегда возможна. Она возможна и для бесконечных

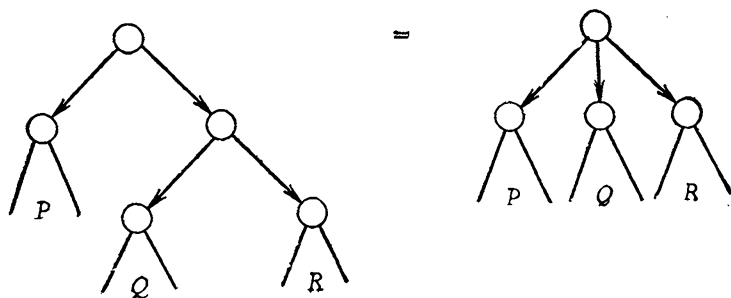


Рис. 3.2

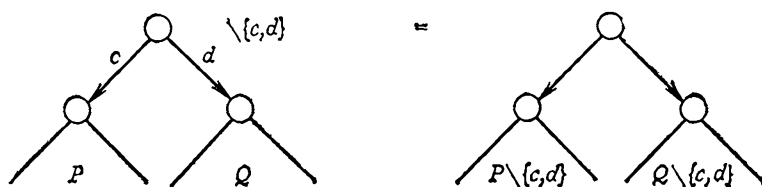


Рис. 3.3

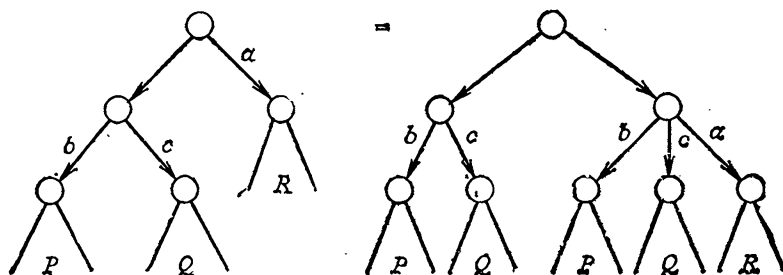


Рис. 3.4

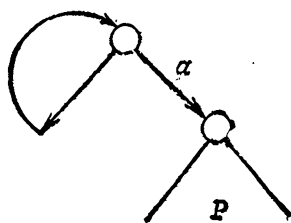


Рис. 3.5

графов при условии, что граф не содержит бесконечного пути, состоящего из последовательных непомеченных дуг, как, например, на рис. 3.5. Такая картина, однако, может возникнуть только в случае расходимости, который мы решили считать ошибочным.

В результате преобразования L10 может получиться, что вершина будет иметь две одинаково помеченные исходящие дуги. Такие вершины можно удалить в соответствии с законом, приведенным в конце разд. 3.3 (рис. 3.6).

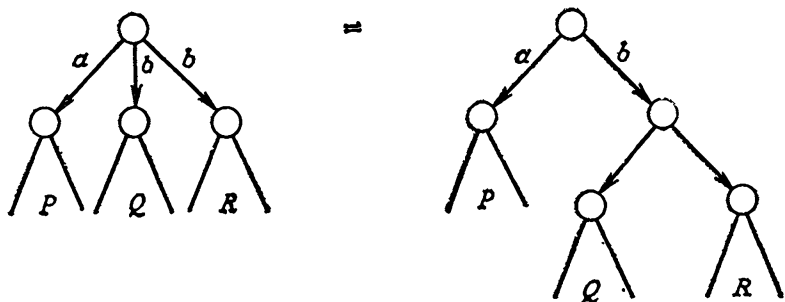


Рис. 3.6

Графическое представление процессов и управляющие ими законы приведены здесь для лучшего понимания и запоминания; при этом не предполагается их практическое использование для преобразований и манипуляций с большими процессами.

3.6. ЧЕРЕДОВАНИЕ

Во второй главе оператор \parallel был определен таким образом, что действия, содержащиеся в алфавите обоих процессов, требуют их одновременного участия, в то время как остальные действия системы произвольно чередуются. Используя этот оператор, можно объединять взаимодействующие процессы с разными алфавитами в параллельно работающие системы, не прибегая при этом к недетерминизму.

Однако бывает полезно объединять процессы с одинаковыми алфавитами для параллельного исполнения, в ходе которого не происходит непосредственного взаимодействия или синхронизации процессов друг с другом. В этом случае каждое действие системы — это действие в точности одного из процессов. Если один процесс не может выполнить действие, это должен сделать другой; если же одно действие готовы выполнить оба процесса, то выбор между ними недетермини-

рован. Такая форма комбинации процессов обозначается

$$P \parallel Q \quad (P \text{ чередуется с } Q),$$

а ее алфавит определяется в соответствии с обычным соглашением

$$\alpha(P \parallel Q) = \alpha P = \alpha Q.$$

Примеры

X1. Торговый автомат, принимающий до двух монет подряд, прежде чем выдать до двух шоколадок (1.1.3 **X6**):

$$(ТАП \parallel ТАП) = ТАП2$$

X2. Слуга, образованный четырьмя лакеями, каждый из которых обслуживает одновременно только одного философа (см. разд. 2.5.3 и 2.6.4 **X1**):

$$L \parallel L \parallel L \parallel L$$

где L — ОБЩИЙ ЛАКЕЙ.

3.6.1. Законы

L1—3. Операция \parallel ассоциативна, симметрична и дистрибутивна относительно \sqcap .

L4. $P \parallel \text{СТОП} = P$

L5. $P \parallel \text{ИСП} = \text{ИСП}$ при условии, что P не расходится

L6. $(x \rightarrow P) \parallel (y \rightarrow Q) = (x \rightarrow (P \parallel (y \rightarrow Q))) \parallel y \rightarrow ((x \rightarrow P) \parallel Q)$

L7. Если $P = (x: A \rightarrow P(x))$, а

$Q = (y: B \rightarrow P(y))$, то

$$P \parallel Q = (x: A \rightarrow (P(x) \parallel Q) \parallel y: B \rightarrow (P \parallel Q(y)))$$

Заметим, что \parallel не дистрибутивна относительно \sqcup . Это можно показать с помощью контрпримера (где $b \neq c$):

$$\begin{aligned} ((a \rightarrow \text{СТОП}) \parallel (b \rightarrow Q \parallel c \rightarrow R)) / \langle a \rangle \\ &= (b \rightarrow Q \parallel c \rightarrow R) \\ &\neq ((b \rightarrow Q) \sqcap (c \rightarrow R)) \\ &= ((a \rightarrow \text{СТОП} \parallel b \rightarrow Q) \parallel (a \rightarrow \text{СТОП} \parallel c \rightarrow R)) / \langle a \rangle \end{aligned}$$

В левом конце этой цепочки наступление события a связано только с левым операндом \parallel и, следовательно, не приводит к недетерминизму. Левый операнд останавливается, а выбор между b и c предоставляется обстановке. В правом конце цепочки событие a может произойти в обоих операндах \parallel , причем выбор недетерминирован. Таким образом, обстановка

уже не может выбирать, какое событие будет следующим — b или c .

В законах **L6** и **L7** утверждается, что выбор начального события из предложенных операндами \parallel делает обстановка. Недетерминизм возникает, только если выбранное событие является возможным для обоих операндов.

Пример

Пусть $R = (a \rightarrow b \rightarrow R)$.

$$(R \parallel R) = (a \rightarrow ((b \rightarrow R) \parallel R) \parallel a \rightarrow (R \parallel (b \rightarrow R))) \quad \mathbf{L6}$$

$$\begin{aligned} &= (a \rightarrow ((b \rightarrow R) \parallel R) \sqcap (R \parallel (b \rightarrow R))) \\ &= a \rightarrow ((b \rightarrow R) \parallel R) \quad \mathbf{L2} \end{aligned}$$

В свою очередь

$$(b \rightarrow R) \parallel R = (a \rightarrow ((b \rightarrow R) \parallel (b \rightarrow R)) \parallel (b \rightarrow (R \parallel R))) \quad \mathbf{L6}$$

$$\begin{aligned} &= (a \rightarrow (b \rightarrow ((b \rightarrow R) \parallel R))) \quad \left\{ \begin{array}{l} \text{как показано} \\ \text{выше} \end{array} \right. \\ &\parallel b \rightarrow (a \rightarrow ((b \rightarrow R) \parallel R))) \end{aligned}$$

$$\begin{aligned} &= \mu X. (a \rightarrow b \rightarrow X \\ &\parallel b \rightarrow a \rightarrow X) \quad \text{так как рекурсия предварена} \end{aligned}$$

Таким образом, мы видим, что процесс $(R \parallel R)$ совпадает с процессом из примера 3.5 **X2**. Аналогично можно показать, что $(TAP \parallel TAP) = TAP2$.

3.6.2. Протоколы и отказы

Протокол процесса $(P \parallel Q)$ представляет собой произвольное чередование протокола P с протоколом Q . Определение чередования протоколов дано в разд. 1.9.3.

$$\mathbf{L1.} \text{ протоколы}(P \parallel Q) = \{s \mid \exists t \in \text{протоколы}(P). \\ \exists u \in \text{протоколы}(Q). s \text{ чередование}(t, u)\}$$

Процесс $(P \parallel Q)$ может участвовать в любом начальном событии, возможном для P или для Q ; поэтому отказаться он может только от множества, являющегося отказом обоих процессов P и Q :

$$\mathbf{L2.} \text{ отказы}(P \parallel Q) = \text{отказы}(P \parallel Q)$$

Поведение $(P \parallel Q)$ после участия в событиях протокола s описывается довольно сложной формулой:

$$\mathbf{L3.} (P \parallel Q)/s = \bigcap_{(t, u) \in T} (P/t) \parallel (Q/u),$$

где $T = \{(t, u) \mid t \in \text{протоколы}(P) \ \& \ u \in \text{протоколы}(Q) \ \& \ s \text{ чередование}(t, u)\}$.

Этот закон отражает тот факт, что нам неизвестно, как именно чередуются протоколы процессов P и Q в протоколе s процесса $(P \parallel Q)$; следовательно, поведение $(P \parallel Q)$ после s может отражать любое из этих возможных чередований. Выбор между ними неизвестен и неопределен.

3.7. СПЕЦИФИКАЦИИ

В разд. 3.4. мы пришли к необходимости ввести множества отказов как один из важных косвенно обозримых аспектов поведения процесса. При спецификации процесса, таким образом, наряду со свойствами его протоколов мы должны описывать желаемые свойства его отказов. Для обозначения произвольного множества отказов (или просто отказа) процесса мы будем использовать переменную $отк$ точно так же, как переменная $пр$ использовалась для обозначения произвольного протокола процесса. В результате если P — недетерминированный процесс, то запись

$$P \text{ уд } S(пр, отк)$$

имеет смысл

$$\forall пр, отк. пр \in \text{протоколы}(P) \& отк \in \text{отказы}(P/пр) \Rightarrow S(пр, отк)$$

Примеры

X1. Если количество монет, поглощенных торговым автоматом, превышает количество выданных шоколадок, то покупатель требует, чтобы торговый автомат не отказывался от выдачи шоколадки.

$$\text{ЧЕСТН} = (пр \downarrow \text{шок} < пр \downarrow \text{мон} \Rightarrow \text{шок} \tilde{=} отк)$$

Неявно подразумевается, что каждый протокол $пр$ и каждый отказ $отк$ специфицируемого процесса в любой момент времени удовлетворяет этой спецификации.

X2. Если выдано ровно столько шоколадок, сколько было оплачено, владелец требует, чтобы автомат не отказывался от приема монеты:

$$\text{ВЫГОД1} = (пр \downarrow \text{шок} = пр \downarrow \text{мон} \Rightarrow \text{мон} \tilde{=} отк)$$

X3. Простой торговый автомат должен удовлетворять составной спецификации

$$\text{НОВТАПВЗАИМ} = \text{ЧЕСТН} \& \text{ВЫГОД1} \\ \& (пр \downarrow \text{шок} \leq пр \downarrow \text{мон})$$

Этой спецификации удовлетворяет *ТАП*, а также автомат типа *ТАП2* (1.1.3 **X6**), который принимает подряд несколько монет, а затем выдает несколько шоколадок.

X4. При желании можно наложить ограничение на количество опускаемых подряд монет:

$$НЕБОЛЕЕ2 = (pr \downarrow мон - pr \downarrow шок \leq 2)$$

X5. Можно потребовать, чтобы автомат принимал не менее двух монет подряд, если этого хочет покупатель:

$$НЕМЕНЕЕ2 = (pr \downarrow мон - pr \downarrow шок < 2 \Rightarrow мон \hat{=} отк)$$

X6. Процесс *СТОП* отказывается от всех событий своего алфавита. Следующий предикат описывает требование, чтобы процесс с алфавитом *A* не останавливался:

$$НЕСТОП = (отк \neq A)$$

Если *P* уд *НЕСТОП*, а его обстановка допускает любое событие из *A*, то *P* должен исполнять одно из них. Так как (см. **X3** выше)

$$НОВТАПВЗАИМ \Rightarrow отк \neq \{мон, шок\}$$

то, следовательно, любой процесс, удовлетворяющий спецификации *НОВТАПВЗАИМ*, бесконечен.

Эти примеры демонстрируют, как введение в спецификацию процесса переменной *отк* позволяет выразить ряд тонких, но важных свойств; наиболее важное из них — это, пожалуй, свойство процесса работать бесконечно (**X6**). Преимущества эти, однако, получены ценой некоторого увеличения сложности правил доказательства и самих доказательств.

Другой факт, который хотелось бы уметь доказывать, — это то, что процесс не расходится. В разд. 3.8 дается описание расходящегося процесса как процесса, который может как выполнять, так и отказываться от любого действия. Значит, если существует множество, которое не может быть отказом процесса, то процесс не расходится. На этом основана формулировка достаточного условия нерасходимости:

$$НЕРАСХ = (отк \neq A)$$

К счастью,

$$НЕСТОП \equiv НЕРАСХ$$

и поэтому доказательство нерасходимости не более трудоемко, чем доказательство отсутствия тупиков.

3.7.1. Доказательства

В следующих правилах доказательств спецификации будут записываться в виде S , $S(np)$ или $S(np, отк)$ в зависимости от того, как будет удобнее. В любом случае надо понимать, что спецификация может содержать np и $отк$ в числе своих свободных переменных.

По определению недетерминизма $(P \sqcap Q)$ ведет себя либо как P , либо как Q . Поэтому всякий результат наблюдений за его поведением будет результатом, возможным для P , для Q или для них обоих. Значит, результат этого наблюдения будет описываться спецификацией P , спецификацией Q или обеими спецификациями. Следовательно, правило доказательства для недетерминизма имеет исключительно простой вид:

L1. Если P уд S
 а Q уд T
 то $(P \sqcap Q)$ уд $(S \vee T)$

Правило доказательства для **СТОП** утверждает, что этот процесс ничего не делает и от всего отказывается:

L2A. $СТОП_A$ уд $(np = \langle \rangle \& отк \subseteq A)$

Так как отказы никогда не выходят за рамки алфавита (3.4.1 **L8**), условие $отк \subseteq A$ можно опустить. Если же мы вообще не упоминаем алфавит (что мы будем делать в дальнейшем), закон **L2A** совпадает с аналогичным законом для детерминированных процессов (1.10.2 **L4A**):

$СТОП$ уд $np = \langle \rangle$

Предыдущий закон для префиксации (1.10.2 **L4B**) тоже остается в силе, однако он недостаточно силен для доказательства того, что процесс не может остановиться еще до наступления начального события. Этот закон необходимо усилить, упомянув тот факт, что в начальном состоянии, когда $np = \langle \rangle$, процесс не может отказаться от начального действия.

L2B. Если P уд $S(np)$,
 то $(c \rightarrow P)$ уд $((np = \langle \rangle \& c \not\subseteq отк) \vee (np_0 = c \& S(np')))$

Закон для общего выбора нуждается в аналогичном усилении:

L2. Если $\forall x \in B. P(x)$ уд $S(np, x)$, то
 $(x : B \rightarrow P(x))$ уд $((np = \langle \rangle \& (B \cap отк) = \{ \}) \vee (np_0 \in B \& S(np', np_0)))$

Закон для параллельной композиции (2.7 **L1**) остается в силе при условии, что в спецификациях никак не упоминают,

ся множества отказов. Для корректного рассмотрения отказов требуется несколько более сложный закон:

L3. Если P уд $S(nr, отк)$, а Q уд $T(nr, отк)$ и ни P , ни Q не расходятся, то

$$(P \parallel Q) \text{ уд } (\exists X, Y. отк = (X \cup Y) \ \& \ S(nr \upharpoonright \alpha P, X) \ \& \ T(nr \upharpoonright \alpha Q, Y))$$

Закон для переименования требует аналогичной адаптации:

L4. Если P уд $S(nr, отк)$, то

$f(P)$ уд $S(f^{-1*}(nr), f^{-1}(отк))$ при условии, что f взаимно однозначна.

Закон для Π удивительно прост:

L5. Если P уд S , а Q уд T и ни P , ни Q не расходятся, то $(P \Pi Q)$ уд (if $nr = \langle \rangle$ then $(S \ \& \ T)$ else $(S \ \vee \ T)$).

Первоначально, когда $nr = \langle \rangle$, множество является множеством отказов процесса $(P \Pi Q)$, только если от него отказываются и P , и Q . Значит, это множество должно описываться *обеими* спецификациями. В дальнейшем, когда $nr \neq \langle \rangle$, каждый результат наблюдений процесса $(P \Pi Q)$ должен быть результатом наблюдения либо P , либо Q и потому описываться одной из спецификаций (или обеими одновременно).

В законе для чередования вообще не требуется упоминания множеств отказов:

L6. Если P уд $S(nr)$, а Q уд $T(nr)$ и ни P , ни Q не расходятся, то $(P \parallel\!\!\!\parallel Q)$ уд $(\exists s, t. (nr \text{ чередование}(s, t) \ \& \ S(s) \ \& \ T(t)))$

Закон для сокрытия сложен ввиду необходимости защиты от расходимости:

L7. Если P уд $(НЕРАСХ \ \& \ S(nr, отк))$, то

$$(P \setminus C) \text{ уд } \exists s. nr = s \upharpoonright (\alpha P - C) \ \& \ S(s, отк \cup C),$$

где $НЕРАСХ$ утверждает, что число спрятанных событий, которые могут произойти, ограничено некоторой функцией от уже произошедших спрятанных событий:

$$НЕРАСХ = \#(nr \upharpoonright C) \leq f(nr \upharpoonright (\alpha P - C)),$$

где f — некоторая всюду определенная функция из множества протоколов во множество натуральных чисел.

Выражение $отк \cup C$ в последующем члене закона **L7** требует пояснений. Здесь отражен тот факт, что $P \setminus C$ может отказаться от множества X , только если P может отказаться от всего множества $X \cup C$, т. е. от множества X вместе со *всеми* спрятанными событиями. Процесс $P \setminus C$ не может от-

казаться от взаимодействия со своим внешним окружением до тех пор, пока не достигнет состояния, в котором он не может больше совершать никаких скрытых внутренних действий. Продуктивность такого рода является важнейшей чертой любого разумного определения сокрытия, о чем уже говорилось в разд. 3.5.1.

Метод доказательства для рекурсии (1.10.2 L6) также нуждается в усилении. Пусть $S(n)$ — предикат, содержащий переменную n , принимающую значения на множестве натуральных чисел $0, 1, 2, \dots$.

L8. Если $S(0)$ и $(X \text{ уд } S(n)) \Rightarrow (F(x) \text{ уд } S(n+1))$, то
 $(\mu X.F(X)) \text{ уд } (\forall n.S(n))$

Этот закон справедлив даже для непредваренной рекурсии, хотя самой сильной спецификацией, которую можно доказать для такого процесса, будет бессодержательная спецификация *истина*.

3.8. РАСХОДИМОСТЬ

В предыдущих главах мы сформулировали ограничение, состоящее в том, что уравнения, рекурсивно определяющие процесс, должны быть *предварены* (разд. 1.1.2). Это ограничение гарантировало, что уравнения имеют только одно решение (1.3 L2), а также избавляло нас от необходимости придавать смысл бесконечной рекурсии $\mu X.X$.

К сожалению, введение сокрытия (разд. 3.5) приводит к тому, что явно предваренная рекурсия оказывается неконструктивной. Рассмотрим, например, уравнение

$$X = c \rightarrow (X \setminus \{c\})_{+(c)}$$

Его решением являются как $(c \rightarrow \text{СТОП})$, так и $(c \rightarrow a \rightarrow \text{СТОП})$, в чем можно убедиться, сделав подстановку.

Следовательно, любое рекурсивное уравнение с рекурсией, заключенной внутри упорядочивающего оператора, является потенциально *непредваренным* и может поэтому иметь более одного решения. Какое же из них считать верным? Договоримся, что правильным решением следует считать наименее детерминированное, потому что это позволяет производить недетерминированный выбор между всеми остальными решениями. Придя к такому соглашению, мы вполне можем отбросить ограничение, связанное с предваренностью рекурсии, и придать (возможно, недетерминированный) смысл *любому* выражению вида $\mu X.F(X)$, где F определено в терминах любых операторов, введенных в этой книге (кроме $/$), и соблюдены все условия, налагаемые на алфавит.

Чтобы сделать этот смысл более понятным, рассмотрим сначала простейший случай, который (как это часто бывает) является при этом и наихудшим, а именно бесконечную рекурсию $\mu X.X$. Решением уравнения $X = X$ может быть любой процесс; следовательно, $\mu X.X$ может вести себя совершенно произвольно. Это наиболее недетерминированный, наименее предсказуемый, наименее контролируемый и, словом, наихудший из всех процессов. Дадим ему подходящее имя и определим

$$XAO C_A = \mu X : A.X$$

Ненамного лучше его и рекурсия, приведенная выше:

$$\mu X.(c \rightarrow (X \setminus \{c\})) = c \rightarrow XAO C$$

Этот процесс отличен от $XAO C_A$, поскольку прежде, чем обратиться в $XAO C$, он по крайней мере заведомо выполнит начальное действие c .

Помимо придания смысла бесконечной рекурсии $XAO C$ является также результатом в случае, когда процесс может участвовать в бесконечной непрерывной последовательности скрытых событий. Простейшим и худшим случаем является немедленно расходящийся процесс, описанный в конце разд. 3.5.1.

$$\begin{aligned} (\mu X : A.(c \rightarrow X)) \setminus \{c\} &= \mu X : (A - \{c\}).(c \rightarrow X) \setminus \{c\} \\ &= \mu X : A - \{c\}. \{X \setminus \{c\}\} \quad \text{по 3.5.1 L5} \\ &= \mu X : A - \{c\}.X \\ &= XAO C_{A - \{c\}} \quad \text{по определению } XAO C \end{aligned}$$

3.8.1. Законы

Поскольку $XAO C$ является наиболее недетерминированным процессом, он не меняется при добавлении к нему дальнейшего недетерминированного выбора; таким образом, он служит нулем для оператора \sqcap :

$$L1. P \sqcap XAO C = XAO C$$

Функция над процессами, принимающая значение $XAO C$ в случае, когда одним из ее аргументов является $XAO C$, называется *строгой*. Предыдущий закон (плюс свойство симметричности) утверждает, что операция \sqcap является строгой. $XAO C$ так ужасен, что почти каждый процесс, определенный в его терминах, оказывается ему эквивалентен.

L2. Следующие операции являются строгими:

$$/s, \parallel, f, \Pi, \setminus C, \parallel, \mu X$$

Префиксация, однако, не является строгой:

$$L3. XAO C \neq (a \rightarrow XAO C)$$

потому что правая часть этого неравенства, прежде чем стать абсолютно недостоверной, с достоверностью выполнит a .

Как уже упоминалось, $ХАОС$ — это наиболее непредсказуемый и наиболее неконтролируемый из всех процессов. Нет такого события, которое он не мог бы выполнить: более того, нет такого события, от которого он не мог бы отказаться!

L4. $протоколы(ХАОС_A) = A^*$

L5. $отказы(ХАОС_A) = \text{все подмножества } A$

3.8.2. Расходимости

Расходимость процесса определяется как любой протокол, после которого процесс начинает вести себя хаотически. Множество всех расходимостей определяется как

$$расходимости(P) = \{s \mid s \in протоколы(P) \text{ \& } (P/s) = ХАОС_{\alpha P}\}$$

Отсюда немедленно следует, что

L1. $расходимости(P) \subseteq протоколы(P)$

Поскольку операция $/t$ строга,

$$ХАОС/t = ХАОС$$

и, значит, множество расходимостей процесса замкнуто относительно расширения в том смысле, что

L2. $s \in расходимости(P) \text{ \& } t \in (\alpha P)^* \Rightarrow (s \hat{ } t) \in расходимости(P)$

Поскольку $ХАОС_A$ может отказаться от любого подмножества своего алфавита A , то

L3. $s \in расходимости(P) \text{ \& } X \subseteq \alpha P \Rightarrow X \in отказы(P/s)$

Три приведенных выше закона формулируют основные свойства множества расходимостей любого процесса. Следующие законы описывают, как расходимости составных процессов определяются множествами расходимостей и протоколов их составляющих. Во-первых, процесс *СТОП* никогда не расходится:

L4. $расходимости(СТОП) = \{ \}$

В другом крайнем случае любой протокол $ХАОС_\alpha$ приводит к $ХАОС_\gamma$:

L5. $расходимости(ХАОС_A) = A^*$

Процесс, определенный с помощью оператора выбора, на первом шаге не расходится. Следовательно, его расходимости определяются тем, что происходит после первого шага:

$$\text{L6. } \text{расходимости}(x : B \rightarrow P(x)) = \\ = \{\langle x \rangle^{\wedge} s \mid x \in B \text{ \& } s \in \text{расходимости}(P(x))\}$$

Любая расходимость P является также расходимостью процессов $(P \sqcap Q)$ и $(P \sqcup Q)$:

$$\text{L7. } \text{расходимости}(P \sqcap Q) = \text{расходимости}(P \sqcup Q) \\ = \text{расходимости}(P) \cup \text{расходимости}(Q)$$

Поскольку операция \parallel строгая, расходимость процесса $(P \parallel Q)$ начинается с протокола нерасходящегося поведения процессов P и Q , приводящего к расходимости либо P , либо Q (или обоих вместе):

$$\text{L8. } \text{расходимости}(P \parallel Q) = \\ \{s^{\wedge} t \mid t \in (\alpha P \cup \alpha Q)^* \text{ \& } ((s \upharpoonright \alpha P \in \text{расходимости}(P) \text{ \& } \\ s \upharpoonright \alpha Q \in \text{протоколы}(Q)) \vee \\ \vee (s \upharpoonright \alpha P \in \text{протоколы}(P) \text{ \& } \\ s \upharpoonright \alpha Q \in \text{расходимости}(Q)))\}$$

Аналогичные рассуждения поясняют закон для операции $\parallel\parallel$:

$$\text{L9. } \text{расходимости}(P \parallel\parallel Q) = \\ \{u \mid \exists s, t. \text{ и чередование}(s, t) \text{ \& } ((s \in \text{расходимости}(P) \text{ \& } \\ t \in \text{протоколы}(Q)) \vee \\ \vee (s \in \text{протоколы}(P) \text{ \& } \\ t \in \text{расходимости}(Q)))\}$$

Множество расходимостей процесса, полученного в результате сокрытия, состоит из протоколов, соответствующих исходному множеству расходимостей, и протоколов, полученных в результате попытки скрыть бесконечную последовательность символов:

$$\text{L10. } \text{расходимости}(P \setminus C) = \\ \{(s \upharpoonright (\alpha P - C))^{\wedge} t \mid t \in (\alpha P - C)^* \\ \text{ \& } (s \in \text{расходимости}(P) \vee \\ \vee (\forall n. \exists u \in C^*. \# u > n \text{ \& } (s^{\wedge} u) \in \\ \text{протоколы}(P)))\}$$

Процесс, определенный с помощью переименования, расходится только в том случае, когда расходится исходный процесс:

$$\text{L11. } \text{расходимости}(f(P)) = \{f^*(s) \mid s \in \text{расходимости}(P)\} \text{ при } \\ \text{условии, что } f \text{ — взаимно однозначная,}$$

Конечно, обидно уделять столько внимания расходимости, когда расходимость — это нечто всегда нежелательное. К сожалению, она представляется нам неизбежным следствием любого эффективного и даже просто вычислимого способа реализации. Она может возникать как вследствие сокрытия, так и вследствие непредваренной рекурсии, и доказательство того, что в некоторой конкретной разработке эта проблема не возникает, остается задачей системного проектировщика. Для доказательства того, что нечто никогда не случится, мы не можем обойтись без математической теории, которая допускает, что это происходит.

3.9. МАТЕМАТИЧЕСКАЯ ТЕОРИЯ НЕДЕТЕРМИНИРОВАННЫХ ПРОЦЕССОВ

Законы, приведенные в этой главе, заметно сложнее, чем законы двух предыдущих глав; нестрогие же пояснения и примеры к ним соответственно менее убедительны. Поэтому еще более важно дать точное математическое определение понятия недетерминированного процесса и доказать справедливость законов, исходя из определений операторов.

Как и в разд. 2.8.1, математическая модель основана на подходящих, прямо или косвенно обозримых свойствах процесса. Среди них, конечно, алфавит процесса и его протоколы; но для недетерминированного процесса это еще и множества его отказов (разд. 3.4) и расходимостей (разд. 3.8). В дополнение к отказам на первом шаге процесса P надо принять во внимание и те множества, от которых P может отказаться после участия в событиях произвольного протокола s . Поэтому определим *неудачи* процесса как отношение (множество пар)

$$\text{неудачи}(P) = \{(s, X) \mid s \in \text{протоколы}(P) \ \& \ X \in \text{отказы}(P/s)\}$$

Если (s, X) — неудача процесса P , это значит, что сначала P участвует в последовательности событий, отраженной в s , после чего может отказаться выполнять что бы то ни было еще, несмотря на то, что обстановка готова участвовать в любом событии из X . Неудачи процесса дают больше информации о его поведении, чем его отказы и протоколы, которые, кстати, могут быть выражены в терминах неудач:

$$\begin{aligned} \text{протоколы}(P) &= \{s \mid \exists X. (s, X) \in \text{неудачи}(P)\} \\ &= \text{область}(\text{неудачи}(P)) \\ \text{отказы}(P) &= \{X \mid (\langle \rangle, X) \in \text{неудачи}(P)\} \end{aligned}$$

Различные свойства протоколов (1.8.1 L6, L7, L8) и отказов (3.4 L8, L9, L10, L11) легко переформулировать в терминах

неудач (см. ниже условия **C0**, **C1**, **C2**, **C3** после определения **D0**).

Теперь мы можем смело утверждать, что процесс однозначно определяется тремя множествами, задающими его алфавит, его неудачи и его расходимости; и наоборот, любые три множества, удовлетворяющие соответствующим условиям, однозначно определяют процесс. Сначала определим множество-степень A как множество всех его подмножеств:

$$PA = \{X \mid X \subseteq A\}$$

D0. Процесс — это тройка (A, F, D) , где A — произвольное множество символов (для простоты — конечное), F — отношение между A^* и PA , D — подмножество A^* , удовлетворяющие следующим условиям:

$$\mathbf{C0.} \ (\langle \rangle, \{ \}) \in F$$

$$\mathbf{C1.} \ (s \hat{ } t, X) \in F \Rightarrow (s, \{ \}) \in F$$

$$\mathbf{C2.} \ (s, Y) \in F \ \& \ X \subseteq Y \Rightarrow (s, X) \in F$$

$$\mathbf{C3.} \ (s, X) \in F \ \& \ x \in A \Rightarrow (s, X \cup \{x\}) \in F \vee (s \hat{ } \langle x \rangle, \{ \}) \in F$$

$$\mathbf{C4.} \ D \subseteq \text{область}(F)$$

$$\mathbf{C5.} \ s \in D \ \& \ t \in A^* \Rightarrow s \hat{ } t \in D$$

$$\mathbf{C6.} \ s \in D \ \& \ X \subseteq A \Rightarrow (s, X) \in F$$

(три последних условия отражают законы 3.8.2 **L1**, **L2**, **L3**).

Простейший процесс, удовлетворяющий этому определению, одновременно является и наихудшим:

$$\mathbf{D1.} \ \text{ХАОС}_A = (A, (A^* \times PA, A^*))$$

где $A^* \times PA$ — декартово произведение $\{s, X \mid s \in A^* \ \& \ X \in PA\}$.

Это наибольший процесс с алфавитом A , ибо каждый элемент A является и его протоколом, и расходимостью, а каждое подмножество A является отказом после любого протокола.

Другой простой процесс — это

$$\mathbf{D2.} \ \text{СТОП}_A = (A, \{ \} \times PA, \{ \})$$

Этот процесс ничего не делает, от всего отказывается и не имеет расходимостей.

Операция над процессами задается описанием того, как получить три множества, определяющие ее результат, из соответствующих множеств ее операндов. Конечно, необходимо показать, что результат операции удовлетворяет шести условиям в определении **D0**; доказательство этого обычно основано на предположении, что эти условия выполняются для всех операндов.

Проще всего определяется операция недетерминированного «или» (\sqcap). Как и многие другие операции, она определена только над процессами с одинаковыми алфавитами:

$$\mathbf{D3.} (A, F1, D1) \sqcap (A, F2, D2) = (A, F1 \cup F2, D1 \cup D2)$$

Результирующий процесс может терпеть неудачу или расходиться во всех тех же случаях, что и любой из операндов. Законы 3.2.1 L1, L2, L3 являются прямыми следствиями этого определения.

Аналогично можно определить все остальные операции; нам же кажется, что будет изящнее дать отдельные определения для алфавитов, неудач и расходимостей. Определения для расходимостей уже приводились в разд. 3.8.2, и поэтому нам осталось определить только алфавиты и неудачи:

$$\mathbf{D4.} \text{ Если } \alpha P(x) = A \text{ для всех } x \text{ и } B \subseteq A, \text{ то } \alpha(x : B \rightarrow P(x)) = A.$$

$$\mathbf{D5.} \alpha(P \parallel Q) = (\alpha P \cup \alpha Q)$$

$$\mathbf{D6.} \alpha(f(P)) = f(\alpha P)$$

$$\mathbf{D7.} \alpha(P \amalg Q) = \alpha(P \parallel Q) = \alpha P \quad \text{при условии, что } \alpha P = \alpha Q$$

$$\mathbf{D8.} \alpha(P \setminus C) = \alpha P - C$$

$$\mathbf{D9.} \text{ неудачи}(x : B \rightarrow P(x)) = \{ \langle \rangle, X \mid X \subseteq (\alpha P - B) \} \cup \{ \langle x \rangle^{\sim} s, X \mid x \in B \vee (s, X) \in \text{неудачи}(P(x)) \}$$

$$\begin{aligned} \mathbf{D10.} \text{ неудачи}(P \parallel Q) = \{ s, (X \cup Y) \mid s \in (\alpha P \cup \alpha Q)^* \\ \& (s \upharpoonright \alpha P, X) \in \text{неудачи}(P) \\ \& (s \upharpoonright \alpha Q, Y) \in \text{неудачи}(Q) \} \\ \cup \{ s, X \mid s \in \text{расходимости}(P \parallel Q) \} \end{aligned}$$

$$\mathbf{D11.} \text{ неудачи}(f(P)) = \{ f^*(s), f(X) \mid (s, X) \in \text{неудачи}(P) \}$$

$$\begin{aligned} \mathbf{D12.} \text{ неудачи}(P \amalg Q) = \{ s, X \mid (s, X) \in \text{неудачи}(P) \cap \text{неудачи}(Q) \\ \vee (s \neq \langle \rangle \& (s, X) \in (\text{неудачи}(P) \cup \\ \cup \text{неудачи}(Q))) \} \\ \cup \{ s, X \mid s \in \text{расходимости}(P \amalg Q) \} \end{aligned}$$

$$\begin{aligned} \mathbf{D13.} \text{ неудачи}(P \parallel Q) = \{ s, X \mid \exists t, u. s \text{ чередование}(t, u) \\ \& (t, X) \in \text{неудачи}(P) \\ \& (u, X) \in \text{неудачи}(Q) \} \\ \cup \{ s, X \mid s \in \text{расходимости}(P \parallel Q) \} \end{aligned}$$

$$\mathbf{D14.} \text{ неудачи}(P \setminus C) = \{ s \upharpoonright (\alpha P - C), X \mid (s, X \cup C) \in \text{неудачи}(P) \} \cup \{ s, X \mid s \in \text{расходимости}(P \setminus C) \}$$

Пояснением к этим законам могут служить пояснения к соответствующим законам о протоколах и отказах для операции $/$.

Остается дать определение процесса, рекурсивно определенного с помощью μ -оператора. В основе будет лежать та

же теория неподвижной точки, что и в разд. 2.8.2, но с другим определением упорядочения:

$$\mathbf{D15.} (A, F1, D1) \equiv (A, F2, D2) \equiv (F2 \subseteq F1 \ \& \ D2 \subseteq D1)$$

Запись $P \equiv Q$ теперь означает, что Q равен P или лучше его в том смысле, что он с меньшей вероятностью расходится и с меньшей вероятностью терпит неудачу. Процесс Q более предсказуем и более контролируем, чем P , потому что если Q может сделать что-либо нежелательное, то это же может сделать и P , а если Q может отказаться делать что-либо желательное, то и P может от этого отказаться. Процесс ХАОС в любой момент может выполнить любое действие и в любой момент может отказаться от любого множества. Вполне оправдывая свое название, он является наименее предсказуемым и контролируемым, т. е., одним словом, наихудшим из процессов:

$$\mathbf{L1.} \text{ХАОС} \equiv P$$

Очевидно, что это упорядочение задает частичный порядок. На самом деле это полный частичный порядок с предельной операцией, определяемой в терминах пересечений убывающих цепей неудач и расходимостей:

$$\mathbf{D16.} \bigsqcup_{n \geq 0} (A, F_n, D_n) = \left(A, \bigcap_{n \geq 0} F_n, \bigcap_{n \geq 0} D_n \right) \\ \text{при условии, что } (\forall n \geq 0. F_{n+1} \subseteq F_n \ \& \ D_{n+1} \subseteq D_n).$$

μ -оператор определяется так же, как и для детерминированных процессов (2.8.2 L7), с учетом разницы в определении упорядочения, требующей, чтобы на месте *СТОП* находился ХАОС:

$$\mathbf{D17.} \mu X : A. F(X) = \bigsqcup_{i \geq 0} F^i(\text{ХАОС}_A).$$

Доказательство того, что это — решение (фактически самое недетерминированное решение) соответствующего уравнения, не отличается от доказательства из разд. 2.8.2.

Как и прежде, справедливость доказательства решающим образом зависит от того факта, что все операции, используемые в правой части рекурсии, должны быть непрерывными в подходящем упорядочении. К счастью, все определенные в этой книге операции (кроме /) непрерывны, и поэтому непрерывной будет любая формула, построенная из них. В случае операции сокрытия требование непрерывности явилось одним из основных соображений в пользу рассмотрения такого весьма сложного понятия, как расходимость.

Глава 4. Взаимодействие

4.1. ВВЕДЕНИЕ

В предыдущих главах мы ввели и проиллюстрировали общее понятие события как действия, не имеющего протяженности во времени, наступление которого может требовать одновременного участия более чем одного независимо описанного процесса. В этой главе мы сосредоточим внимание на одном специальном классе событий, называемых *взаимодействиями*. Взаимодействие¹⁾ состоит в передаче сообщений и является событием, описываемым парой $c.v$, где c — имя канала, по которому происходит взаимодействие, а v — значение передаваемого сообщения. Пример использования такого обозначения уже приводился при описании процессов КОПИБИТ (1.1.3 X7) и ЦЕПЬ2 (2.6 X4).

Множество всех сообщений, с помощью которых P может осуществлять взаимодействие по каналу c , определяется как

$$ac(P) = \{v \mid c.v \in \alpha P\}$$

Кроме того, определены функции, выбирающие имя канала и значение сообщения:

$$\text{канал}(c.v) = c, \text{ сообщ}(c.v) = v$$

Все вводимые в этой главе операции можно определить в терминах более примитивных понятий, введенных в предыдущих главах, а большинство законов являются частными случаями уже известных. Специальные же обозначения вводятся из тех соображений, что они могут подсказать удобные способы реализации и применения, а также потому, что в некоторых случаях их введение позволяет применять более мощные методы рассуждений.

¹⁾ Заметим, что в оригинале взаимодействующие процессы называются буквально «сообщающиеся процессы» (communicating processes). В русской литературе, однако, уже установился термин «взаимодействующие процессы». — *Прим. ред.*

4.2. ВВОД И ВЫВОД

Пусть v — элемент $\alpha c(P)$. Процесс, который сначала выводит v по каналу c , а затем ведет себя как P , обозначим

$$(c!v \rightarrow P) = (c.v \rightarrow P)$$

Единственное событие, к которому этот процесс готов в начальном состоянии, — это взаимодействие $c.v$.

Процесс, который в начальном состоянии готов ввести любое значение x , передаваемое по каналу c , а затем ведет себя как $P(x)$, обозначим

$$(c?x \rightarrow P(x)) = (y: \{y \mid \text{канал}(y) = c\} \rightarrow P(\text{сообщ}(y)))$$

Пример

X0. Используя новые определения ввода и вывода, можно переписать пример 1.1.3 **X7**:

$$\text{КОПИБИТ} = \mu X. (av?x \rightarrow (vv!x \rightarrow X)),$$

где $avv(\text{КОПИБИТ}) = avviv(\text{КОПИБИТ}) = \{0,1\}$

Напомним принятое нами соглашение, что каналы используются для передачи сообщений только в одном направлении

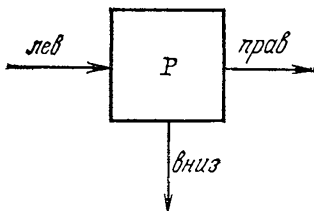


Рис. 4.1

и только между двумя процессами. Канал, используемый процессом только для вывода сообщений, будем называть выходным каналом этого процесса, используемый только для ввода — входным каналом. В обоих этих случаях мы будем позволять себе неточность и говорить, что имя канала принадлежит алфавиту процесса.

На коммутационной схеме процесса (разд. 2.4) каналы изображаются стрелками, направленными в соответствующую сторону и помеченными именем канала (рис. 4.1).

Пусть P и Q — процессы, а c — выходной канал P и входной канал Q . Если P и Q объединены в параллельную систему $(P \parallel Q)$, то взаимодействие по каналу c будет происходить всякий раз, когда P выводит сообщение, а Q в тот же

самый момент вводит его. Выводящий процесс однозначно определяет значение передаваемого сообщения, тогда как вводящий процесс готов принять любое поступающее значение. Поэтому в действительности происходящим событием будет взаимодействие $c.v$, где v — значение, определяемое выводящим процессом¹⁾. Отсюда вытекает очевидное ограничение, что алфавиты на обоих концах канала должны совпадать, т. е.

$$\alpha c(P) = \alpha c(Q)$$

В дальнейшем мы будем предполагать, что это ограничение соблюдается, и там, где это не вызывает недоразумений, вместо $\alpha c(P)$ писать αc . Пример работы такой модели взаимодействия представляет собой ЦЕПЬ2 (2.6 X4); более интересные примеры будут приведены в разд. 4.3 и последующих разделах.

В общем случае значение, вводимое процессом, определяется выражением, содержащим переменные, которым было присвоено значение некоторым предыдущим вводом, что иллюстрируется следующими примерами.

Примеры

X1. Процесс, немедленно копирующий каждое сообщение, поступившее слева, выводя его направо:

$$\begin{aligned} \alpha_{лев}(КОПИР) &= \alpha_{прав}(КОПИР) \\ КОПИР &= \mu X. (лев?x \rightarrow прав!x \rightarrow X) \end{aligned}$$

Если $\alpha_{лев} = \{0, 1\}$, то КОПИР почти полностью совпадает с КОПИБИТ (1.1.3 X7).

X2. Процесс, похожий на КОПИР, с той разницей, что каждое вводимое число перед выводом удваивается:

$$\begin{aligned} \alpha_{лев} &= \alpha_{прав} = \mathbb{N} \\ УДВ &= \mu X. (лев?x \rightarrow прав!(x + x) \rightarrow X) \end{aligned}$$

X3. Значение перфокарты — это последовательность из восьмидесяти литер, которая может считываться как единое значение по левому каналу. Процесс, считывающий перфокарты

¹⁾ Хотя передача сообщения по каналу происходит только в одну сторону, процессы ведут себя согласованно в том смысле, что сообщение, выводимое процессом P , обязательно вводится процессом Q . Именно это обстоятельство позволяет говорить о передаче сообщений как о взаимодействии. — *Прим. ред.*

и выводящий их литеры одну за другой:

$$\begin{aligned} \text{алев} &= \{s \mid s \in \text{аправ}^* \& \# s = 80\} \\ \text{РАСПАК} &= P_{\langle \rangle} \end{aligned}$$

где $P_{\langle \rangle} = \text{лев?}s \rightarrow P_s$, $P_{\langle x \rangle} = \text{прав!}x \rightarrow P_{\langle \rangle}$, $P_{\langle x \rangle \sim s} = \text{прав!}x \rightarrow P_s$.

Х4. Процесс, вводящий слева литеры по одной и komponующий их в строки длиной 125 литер. Каждая заполненная строка выводится направо как единое сообщение, имеющее значение массива.

$$\begin{aligned} \text{аправ} &= \{s \mid s \in \text{алев}^* \& \# s = 125\} \\ \text{УПАК} &= P_{\langle \rangle} \end{aligned}$$

где $P_s = \text{прав!}s \rightarrow P_{\langle \rangle}$, если $\# s = 125$, $P_s = \text{лев?}x \rightarrow P_{s \sim \langle x \rangle}$, если $\# s < 125$.

Здесь P_s описывает поведение процесса, считавшего и упаковавшего литеры в последовательность s ; их вывод осуществляется, когда строка достигает требуемой длины.

Х5. Процесс, осуществляющий копирование слева направо, но заменяющий каждую пару последовательных символов "*" на один символ "↑".

$$\begin{aligned} \text{алев} &= \text{аправ} - \{ " \uparrow " \} \\ \text{СЖИМ} &= \mu X. \text{лев?}x \rightarrow \\ &\quad \text{if } x \neq "*" \text{ then } (\text{прав!}x \rightarrow X) \\ &\quad \text{else } \text{лев?}y \rightarrow (\text{if } y = "*" \text{ then } (\text{прав!"} \uparrow " \rightarrow X) \\ &\quad \quad \text{else } (\text{прав!"*"} \rightarrow \text{прав!}y \rightarrow X)) \end{aligned}$$

Первоначально процесс может быть готов взаимодействовать по любому из некоторого множества каналов, оставляя выбор между ними другим сообщаемым с ним процессам. С этой целью мы изменим форму записи оператора выбора из гл. 1. Если c и d — два различных имени каналов, то

$$(c?x \rightarrow P(x) \mid d?y \rightarrow Q(y))$$

обозначает процесс, который сначала вводит x по каналу c , а затем ведет себя как $P(x)$ или сначала вводит y по каналу d , а затем ведет себя как $Q(y)$. Выбор определяется тем, какой из соответствующих выходов будет готов первым, что поясняется ниже.

Так как мы решили абстрагироваться от временной привязки событий и скорости участвующих в них процессов, последняя фраза предыдущего абзаца может потребовать пояснения. Рассмотрим случай, когда c и d являются выходными каналами двух различных процессов, независимых в том смысле, что они ни прямо, ни косвенно не взаимодействуют

друг с другом. Значит, действия этих процессов произвольно чередуются. Таким образом, если вывод по каналу c происходит в ходе работы одного процесса, а вывод по каналу d — в ходе работы другого, то неизвестно, какое из этих двух событий произойдет первым. Предполагается, что разработчик разрешит этот недетерминизм в пользу того вывода, который будет готов первым (хотя он и не вынужден поступать именно так). Такая политика защищает от дедлока, возникающего в случае, если второй вывод не наступает вовсе или может наступить только после первого вывода. Это возможно, например, когда оба канала c и d соединены с одним и тем же параллельным процессом, который выводит сообщение сначала по одному из них, а затем по другому: $(c!2 \rightarrow d!4 \rightarrow P)$.

Определенный таким образом выбор между входами не только является защитой от дедлока, но и позволяет достигнуть большей эффективности и сократить время реакции на предложенное взаимодействие. Пассажиру, ожидающему автобус номер 127, в общем случае приходится ждать дольше, чем тому, кто может воспользоваться как 19-м, так и 127-м маршрутом автобуса в зависимости от того, который из них раньше подойдет к остановке. В предположении что автобусы подходят в случайном порядке, у пассажира, имеющего выбор, время ожидания оказывается вдвое короче — парадоксально, но получается, что он ждет вдвое «быстрее»! Единственный способ достигнуть этого — ожидать именно первого из многих возможных событий; приобретение же более быстросредействующего компьютера здесь не поможет.

Х6. Процесс, принимающий сообщение по одному из двух каналов $лев1$ и $лев2$ и немедленно передающий его направо:

$$\begin{aligned} \alpha_{лев1} &= \alpha_{лев2} = \alpha_{прав}, \\ \text{СЛИЯНИЕ} &= (лев1?x \rightarrow прав!x \rightarrow \text{СЛИЯНИЕ} \\ &\quad | лев2?x \rightarrow прав!x \rightarrow \text{СЛИЯНИЕ}) \end{aligned}$$

Вывод, осуществляемый этим процессом, представляет собой чередование сообщений, введенных по каналам $лев1$ и $лев2$.

Х7. Процесс, всегда готовый или ввести значение, поступающее слева, или вывести направо последнее введенное значение:

$$\begin{aligned} \alpha_{лев} &= \alpha_{прав} \\ \text{ПЕРЕМ} &= лев? : \rightarrow \text{ПЕРЕМ}_x \\ \text{где } \text{ПЕРЕМ}_x &= (лев?y \rightarrow \text{ПЕРЕМ}_y \\ &\quad | прав!x \rightarrow \text{ПЕРЕМ}_x) \end{aligned}$$

$ПЕРЕМ_x$ ведет себя здесь как программная переменная с текущим значением x . Новые значения присваиваются ей передачами по левому каналу, а взятие текущего значения осуществляется передачей по правому каналу. Если $\alpha_{лев} = \{0, 1\}$, поведение $ПЕРЕМ$ почти полностью совпадает с поведением $ЛОГ$ (2.6 X5).

X8. Процесс вводит сообщения, поступающие по каналам *верх* и *лев*, и выводит функцию от введенных данных по каналу *вниз*, после чего все повторяется:

$$ВЕРШИНА(v) = \mu X. (верх?сумма \rightarrow лев?произв \rightarrow \\ \text{вниз}!(сумма + v \times произв) \rightarrow X)$$

X9. Процесс, в любой момент времени готовый принять сообщение слева и вывести направо первое значение, которое он принял, но еще не передавал: $БУФЕР = P_{\langle \rangle}$, где $P_{\langle \rangle} = лев?x \rightarrow P_{\langle x \rangle}$, а

$$P_{\langle x \rangle} \sim s = (лев?y \rightarrow P_{\langle x \rangle} \sim s \sim \langle y \rangle) \\ | \text{прав}!x \rightarrow P)$$

$БУФЕР$ ведет себя по принципу очереди; сообщения попадают в правый конец очереди, а покидают ее с левого конца в порядке их поступления, но после возможной задержки, в течение которой очередь может пополняться новыми сообщениями.

X10. Процесс, ведущий себя по принципу стека сообщений. Будучи пустым, он реагирует на сигнал *пусто*. В любой момент он готов принять слева новое сообщение и поместить его в верхушку стека; будучи непустым, он всегда готов вывести и удалить элемент, находящийся в верхушке стека:

$$СТЕК = P_{\langle \rangle}, \text{ где } P_{\langle \rangle} = (\text{пусто} \rightarrow P_{\langle \rangle} | лев?x \rightarrow P_{\langle x \rangle}), \text{ а } P_{\langle x \rangle} \sim s = \\ = (\text{прав}!x \rightarrow P_s | лев?y \rightarrow P_{\langle y \rangle} \sim \langle x \rangle \sim s).$$

Этот процесс очень похож на предыдущий с той разницей, что, будучи пустым, он участвует в событии *пусто* и что вновь поступающие сообщения он помещает в тот же конец хранимой последовательности, с которого и удаляет. Таким образом, если y — вновь введенное сообщение, а x — сообщение, готовое в текущий момент к выдаче, $СТЕК$ помещает их в порядке $\langle y \rangle \sim \langle x \rangle \sim s$, а $БУФЕР$ — в порядке $\langle x \rangle \sim s \sim \langle y \rangle$.

4.2.1. Реализация

В ЛИСП-реализации взаимодействующих процессов событие $c.v$ естественно представить парой $(c.v)$, построенной как $cons("c, v)$. Команды ввода и вывода удобно представлять

функциями, получающими в качестве аргумента имя канала. Если процесс не готов к взаимодействию по этому каналу, он выдает ответ *"BLEEP"*. Значение же передаваемого сообщения обрабатывается отдельно на следующей стадии, как описывается ниже.

Если Q — команда ввода $(c?x \rightarrow Q(x))$, то $Q(c) \neq \text{"BLEEP"}$; напротив ее результатом будет функция, ожидающая в качестве аргумента входное значение x и передающая его в качестве результата процессу $Q(x)$. Поэтому Q можно реализовать вызовом ЛИСП-функции *ввод*(" $c, \lambda x.Q(x)$ "), определенной как

$$\text{ввод}(c, F) = \lambda y. \text{if } y \neq c \text{ then "BLEEP" else } F$$

Отсюда вытекает, что $Q/\langle c.v \rangle$ представляется в ЛИСПе как $Q("c)(v)$, при условии, что $\langle c.v \rangle$ является протоколом Q .

Если P — команда вывода $(c!v \rightarrow P')$, то $P("c) \neq \text{"BLEEP"}$; напротив, ее результатом будет пара $\text{cons}(v, P')$. Таким образом, P реализуется вызовом ЛИСП-функции *вывод*(" c, v, P "), определенной как

$$\text{вывод}(c, v, P) = \lambda y. \text{if } y \neq c \text{ then "BLEEP" else } \text{cons}(v, P)$$

Отсюда вытекает, что $v = \text{car}(P("c))$, а $P/\langle c.v \rangle$ представляется в ЛИСПе конструкцией $\text{cdr}(P("c))$ при условии, что $\langle c.v \rangle$ — протокол P .

Теоретически, если αc конечен, можно было бы рассматривать $c.v$ как единое событие, передаваемое в качестве параметра командам ввода и вывода. Но это было бы чудовищно неэффективно, потому что тогда единственным способом обнаружить значение выводимого сообщения была бы проверка $P(c.v) \neq \text{"BLEEP"}$ для всех значений v из αc до тех пор, пока не будет найдено верное. Одним из соображений подтверждающих правомерность введения специальных обозначений, служит поддержка возможности гораздо более эффективных способов реализации. Недостаток же этого метода состоит в том, что в свете этой оптимизации требуется перекодировка реализации почти всех остальных операций.

Примеры

X1. $\text{КОПИР} = \text{LABEL } X. \text{ввод}(\text{"лев"}, \lambda x. \text{вывод}(\text{"прав"}, x, X))$

X2. $\text{УПАК} = P(\text{NIL})$, где $P = \text{LABEL } X.$

$$\lambda s. \text{if } \text{length}(s) = 125 \text{ then } \text{вывод}(\text{"прав"}, s, X(\text{NIL})) \\ \text{else } \text{ввод}(\text{"лев"}, \lambda x. X(\text{append}(s, \text{cons}(x, \text{NIL}))))$$

4.2.2. Спецификации

При спецификации поведения взаимодействующих процессов удобно отдельно описывать последовательности сообщений, передаваемых вдоль каждого из каналов. Если c — имя канала, то определим (см. разд. 1.9.6)

$$pr \downarrow c = \text{сообщ}^*(pr \upharpoonright ac).$$

Для удобства записи член $pr \downarrow$ можно опускать и вместо $pr \downarrow \text{прав} \leq pr \downarrow \text{лев}$ писать $\text{прав} \leq \text{лев}$.

Другое полезное определение ограничивает снизу длину префикса:

$$s \overset{n}{\leq} t = (s \leq t \ \& \ \#t \leq \#s + n)$$

Эта запись означает, что s является префиксом t и короче его не более чем на n элементов. Приведем несколько очевидных и полезных законов:

$$s \overset{0}{\leq} t \equiv (s = t)$$

$$s \overset{n}{\leq} t \ \& \ t \overset{m}{\leq} u \Rightarrow s \overset{n+m}{\leq} u$$

$$s \leq t \equiv \exists n. s \overset{n}{\leq} t$$

Примеры

X1. КОПИР уд $\text{прав} \overset{1}{\leq} \text{лев}$

X2. УДВ уд $\text{прав} \overset{1}{\leq} \text{удвоить}^*(\text{лев})$

X3. РАСПАК уд $\text{прав} \leq \hat{\ }/\text{лев},$

где $\hat{\ }/\langle s_0, s_1, \dots, s_{n-1} \rangle = s_0 \hat{\ } s_1 \hat{\ } \dots \hat{\ } s_{n-1}$ (см. 1.9.2)

Эта спецификация утверждает, что значение, выводимое по правому каналу, получается распаковкой и сцеплением последовательности последовательностей, вводимых слева.

X4. УПАК уд $((\hat{\ }/\text{прав} \overset{125}{\leq} \text{лев}) \ \& \ (\#^* \text{прав}) \in \{125\}^*)$

Эта спецификация утверждает, что каждый выводимый направо элемент представляет собой последовательность длины 125, а конкатенация всех этих последовательностей является начальной подпоследовательностью ввода слева.

Если \oplus — двуместный оператор, то его удобно дистрибутивно применять к соответствующим элементам двух после-

довательностей. Длина результирующей последовательности равна длине более короткого из операндов:

$$s \oplus t = \begin{cases} \langle \rangle & \text{если } s = \langle \rangle \text{ или } t = \langle \rangle \\ \langle s_0 \oplus t_0 \rangle \wedge (s' \oplus t') & \text{иначе} \end{cases}$$

Ясно, что $(s \oplus t)[i] = s[i] \oplus t[i]$ для $i < \min(\#s, \#t)$,

$$\text{а } s \stackrel{n}{\leq} t \Rightarrow (s \oplus u \stackrel{n}{\leq} t \oplus u) \& (u \oplus s \stackrel{n}{\leq} u \oplus t)$$

Х5. Последовательность Фибоначчи $\langle 1, 1, 2, 3, 5, 8 \dots \rangle$ определяется рекуррентным соотношением

$$\begin{aligned} \text{фиб}[0] &= \text{фиб}[1] = 1 \\ \text{фиб}[i+2] &= \text{фиб}[i+1] + \text{фиб}[i] \end{aligned}$$

Вторую строчку можно переписать, используя операцию ', сдвигающую всю последовательность на один элемент влево:

$$\text{фиб}'' = \text{фиб}' + \text{фиб}.$$

Исходное определение последовательности Фибоначчи можно восстановить по этой менее явной форме записи, взяв i -ю компоненту от обеих частей уравнения:

$$\text{фиб}''[i] = (\text{фиб}' + \text{фиб})[i],$$

значит,

$$\text{фиб}'[i+1] = \text{фиб}'[i] + \text{фиб}[i], \quad (1.9.4 \text{ L1})$$

откуда следует, что

$$\text{фиб}[i+2] = \text{фиб}[i+1] + \text{фиб}[i].$$

По-другому смысл этого уравнения можно объяснить, представив его в виде бесконечной суммы, где сдвиг влево представлен в явном виде:

$$\begin{array}{rcl} 1, 1, 2, 3, 5 \dots & & \text{фиб} \\ // \quad // \quad // \quad // & & \\ 1, 2, 3, 5, \dots & + & \text{фиб}' \\ // \quad // \quad // & & \\ 2, 3, 5, \dots & = & \text{фиб}'' \end{array}$$

До сих пор мы рассматривали *фиб* как бесконечную последовательность. Если s — конечный начальный отрезок *фиб* (и $\#s \geq 2$), то вместо уравнения мы имеем неравенство $s'' \leq \leq s' + s$. Эту формулу можно использовать для спецификации процесса *ФИБ*, выдающего направо последовательность чисел Фибоначчи:

$$\text{ФИБ уд} (n\text{рав} \leq \langle 1, 1 \rangle \vee \langle 1, 1 \rangle \leq n\text{рав} \& n\text{рав}'' \leq n\text{рав}' + n\text{рав}).$$

Х6. Переменная, имеющая значение x , выводит направо или последнее введенное слева значение, или x , если такого значения введено не было. Более строго, если последним действием был вывод, то выведенное значение равно последнему члену последовательности $\langle x \rangle^{\sim \text{лев}}$:

ПЕРЕМ_x уд $(\text{канал}(\overline{p r_0}) = \text{прав} \Rightarrow \overline{\text{прав}_0} = \overline{(\langle x \rangle^{\sim \text{лев}})_0})$,

где \bar{s}_0 — последний элемент s (разд. 1.9.5).

Это пример процесса, не поддающегося адекватной спецификации только в терминах последовательностей сообщений, передаваемых по его отдельным каналам. Здесь необходимо также знать порядок, в котором чередуются взаимодействия по различным каналам, например то, что последнее взаимодействие происходило по правому каналу. Такая дополнительная сложность будет главным образом необходима для процессов, использующих оператор выбора.

Х7. Процесс *СЛИЯНИЕ* выдает в качестве результата чередование (разд. 1.9.3) двух последовательностей, вводимых по каналам *лев1* и *лев2*, буферизуя их и объединяя (сливая) в одно сообщение:

СЛИЯНИЕ уд $\exists r. \text{прав} \stackrel{1}{\leq} r \ \& \ r \text{ чередование}(\text{лев1}, \text{лев2})$.

Х8. *БУФЕР* уд $\text{прав} \leq \text{лев}$

Процесс, удовлетворяющий спецификации $(\text{прав} \leq \text{лев})$, описывает поведение транспортного протокола¹⁾ связей, гарантирующего передачу направо только тех сообщений, которые были получены слева, и только в том же порядке. Протокол достигает этого, несмотря на тот факт, что место, откуда передаются сообщения, может находиться на значительном удалении от места, где они принимаются, и что передающая среда, связывающая эти два места, вообще говоря, до некоторой степени ненадежна.

Примеры будут приведены в разд. 4.4.5.

4.3. ВЗАИМОДЕЙСТВИЯ

Пусть P и Q — процессы, а c — канал, используемый P для вывода, а Q для ввода. Тогда множество, состоящее из событий-взаимодействий вида $c.v$, содержится в пересечении алфавита P с алфавитом Q . Если процессы объединяются в параллельную систему $(P \parallel Q)$, взаимодействие $c.v$ может

¹⁾ Заметим, что здесь «протокол» используется не как понятие из теории процессов, а как понятие из теории связи. С этой омонимией читателю придется встретиться еще несколько раз, — *Прим. ред.*

произойти, только когда в этом событии одновременно участвуют оба процесса, т. е. в тот момент, когда P выводит значение v по каналу c , а Q одновременно вводит это значение. Вводящий процесс готов принять *любое* возможное поступающее сообщение, и поэтому то, какое именно значение передается в каждом случае, определяет, как и в примере 2.6 **X4**, выводящий процесс.

Таким образом, вывод можно рассматривать как специальный случай операции префиксации, а ввод — как специальный случай выбора; это позволяет сформулировать закон

$$\mathbf{L1.} \quad (c!v \rightarrow P) \parallel (c?x \rightarrow Q(x)) = c!v \rightarrow (P \parallel Q(v))$$

Заметим, что в правой части этого уравнения событие $c!v$ остается наблюдаемым действием в поведении системы. На практике это соответствует физической возможности сделать ответвления проводов, соединяющих компоненты системы, и регистрировать таким образом внутренние взаимодействия этих компонент. Это же помогает в рассуждениях относительно системы. При желании такие внутренние взаимодействия можно скрыть, применив снаружи к параллельной композиции двух процессов, взаимодействующих по общему каналу, оператор сокрытия, описанный в разд. 3.5:

$$\mathbf{L2.} \quad ((c!v \rightarrow P) \parallel (c?x \rightarrow Q(x))) \setminus C = (P \parallel Q(v)) \setminus C,$$

где $C = \{c.v \mid v \in \alpha c\}$.

Примеры будут приведены в разд. 4.4 и 4.5.

Если использовать имена каналов для обозначения последовательностей проходящих по ним сообщений, то спецификации параллельной композиции взаимодействующих процессов приобретают особенно простой вид. Пусть c — имя канала, по которому взаимодействуют P и Q . В спецификации P имя c соответствует последовательности сообщений, с помощью которых P взаимодействует по каналу c . Аналогично в спецификации Q имя c обозначает последовательность сообщений, с помощью которых взаимодействует Q . К счастью, сама природа взаимодействия такова, что при взаимодействии P и Q по каналу c последовательности принятых и полученных сообщений в любой момент должны совпадать. Следовательно, эта последовательность должна удовлетворять как спецификации P , так и спецификации Q . Это же справедливо для всех каналов из пересечения алфавитов процессов.

Рассмотрим теперь канал d , принадлежащий алфавиту P , но не принадлежащий алфавиту Q . Этот канал не может упоминаться в спецификации Q , и потому ограничения на передаваемые по нему сообщения налагаются только специфика-

цией P . Аналогично Q определяет свойства взаимодействий по своим каналам. Поэтому спецификацию поведения $(P \parallel Q)$ можно построить просто как конъюнкцию спецификаций P и Q . Однако такое упрощение правомерно только в том случае, когда спецификации P и Q выражены исключительно в терминах имен каналов, что, как показывает пример 4.2.2 X6, не всегда возможно.

Примеры

X1. Пусть $P = (\text{лев} ? x \rightarrow \text{средн} ! (x \times x) \rightarrow P)$

$Q = (\text{средн} ? y \rightarrow \text{прав} ! (173 \times y) \rightarrow Q)$

Очевидно, что P уд $(\text{средн} \stackrel{1}{\leq} \text{квадрат}^*(\text{лев}))$, а Q уд $(\text{прав} \stackrel{1}{\leq} 173 \times \text{средн})$, где $(173 \times \text{средн})$ умножает каждое сообщение средн на 173. Отсюда следует, что

$(P \parallel Q)$ уд $(\text{прав} \stackrel{1}{\leq} 173 \times \text{средн}) \& (\text{средн} \stackrel{1}{\leq} \text{квадрат}^*(\text{лев}))$

Эта спецификация влечет за собой неравенство

$\text{прав} \leq 173 \times \text{квадрат}^*(\text{лев})$

которое, как можно предположить, и являлось ее исходной целью.

Формулы, полученные при объединении взаимодействующих процессов параллельным оператором \parallel , сразу же наводят на мысль о физическом способе реализации, когда электронные компоненты соединяются проводами, по которым они взаимодействуют. Целью такой реализации является повышение скорости получения ожидаемых результатов. Эта техника особенно эффективна, когда одно и то же вычисление повторяется для каждого члена потока входных данных, а результаты должны выводиться с той же скоростью, с которой происходит ввод, но, возможно, после некоторой начальной задержки. Такие системы называются потоковыми сетями.

Графическое изображение системы взаимодействующих процессов весьма точно отражает их физическую реализацию. Выходной канал процесса соединен с одноименным входным каналом другого процесса, а каналы, входящие в алфавит только отдельных процессов, оставлены свободными. Пример X1, таким образом, можно изобразить, как показано на рис. 4.2.

X2. Два потока чисел вводятся по каналам лев1 и лев2 . Для каждого значения x , считанного с лев1 , и каждого y , считан-

ного с *лев2*, направо выводится число $(a \times x + b \times y)$. Условие быстродействия требует, чтобы умножения производились параллельно. Определим для этого два процесса, *X21* и *X22*, и возьмем их композицию:

$$X21 = (\text{лев1?}x \rightarrow \text{средн!}(a \times x) \rightarrow X21)$$

$$X22 = (\text{лев2?}y \rightarrow \text{средн?}z \rightarrow \text{прав!}(z + b \times y) \rightarrow X22)$$

$$X2 = (X21 \parallel X22)$$

Очевидно, что

$$\begin{aligned} X2 \text{ уд } (\text{средн} \stackrel{1}{\leq} a \times \text{лев1} \ \& \ \text{прав} \stackrel{1}{\leq} \text{средн} + b \times \text{лев2}) \\ \Rightarrow (\text{прав} \leq a \times \text{лев1} + b \times \text{лев2}). \end{aligned}$$

X3. Слева поступает поток чисел, а направо выдаются взвешенные суммы последовательных пар входных чисел с ве-

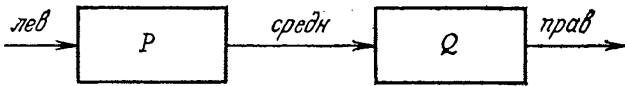


Рис. 4.2

сами a и b . Более точно, мы требуем, чтобы $\text{прав} \leq a \times \text{лев} + b \times \text{лев}'$. Решение можно построить, добавив к решению *X2* новый процесс *X23*:

$X3 = (X2 \parallel X23)$, где $X23 \text{ уд } (\text{лев1} \stackrel{1}{\leq} \text{лев} \ \& \ \text{лев2} \stackrel{1}{\leq} \text{лев}')$. Процесс *X23* можно определить как

$$X23 = (\text{лев?}x \rightarrow \text{лев1!}x \rightarrow (\mu X. \text{лев?}x \rightarrow \text{лев2!}x \rightarrow \text{лев1!}x \rightarrow X))$$

Он копирует из *лев* в *лев1* и *лев2*, но в случае *лев2* пропускает первый элемент.

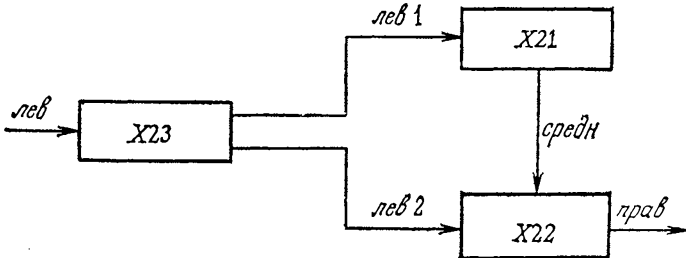


Рис. 4.3

Графическое представление сети *X3* дано на рис. 4.3.

Когда два параллельных процесса взаимодействуют друг с другом вводом и выводом по единственному каналу, дедлок между ними невозможен (ср. с 2.7 L2). Поэтому любая сеть

бесконечных процессов, не содержащая циклов, не может прийти в тупиковое состояние, поскольку ациклический граф можно разбить на подграфы, соединенные единственной дугой. Сеть $X3$, однако, содержит неориентированный цикл и, следовательно, не может быть разбита на подсети, соединенные менее чем двумя каналами; в этом случае убедиться в отсутствии возможного дедлока так просто не удастся. Если, например, в цикле процесса $X23$ переставить местами два вывода $лев2!x \rightarrow лев1!x \rightarrow$, то дедлок наступит немедленно. При доказательстве беступиковости часто удается не принимать во внимание содержание сообщений и рассматривать каждое взаимодействие по каналу c как отдельное событие с именем c . Взаимодействия по неподсоединенным каналам можно не рассматривать. В терминах этих событий $X3$ можно записать как

$$\begin{aligned} & (\mu X. лев1 \rightarrow средн \rightarrow X \\ & \quad \parallel (\mu Y. лев2 \rightarrow средн \rightarrow Y) \\ & \quad \parallel (лев1 \rightarrow (\mu Z. лев2 \rightarrow лев1 \rightarrow Z))) \\ & \Leftarrow \mu X3. (лев1 \rightarrow лев2 \rightarrow средн \rightarrow X3) \end{aligned}$$

Теперь, используя алгебраические методы, как в примере 2.3 **X1**, можно доказать, что $X3$ не может прийти в тупиковое состояние.

Эти примеры показывают, как с помощью потоковых сетей по одному или более потоку входных данных вычисляется один или более поток результатов. Форма сети тесно соответствует структуре операндов и операций, встречающихся в вычисляемых выражениях. Когда эти структуры обширны, но регулярны, удобно воспользоваться индексированными именами каналов и ввести итеративную запись для параллельной композиции:

$$\parallel_{i < n} P(i) = (P(0) \parallel P(1) \parallel \dots \parallel P(n-1))$$

Регулярная сеть такого вида известна как *итеративный массив*. Если схема коммутаций не содержит ориентированных циклов (контуров), часто используют термин *систолический массив*, так как прохождение данных через такую систему очень напоминает прохождение крови через сердечные полости.

X4. Каналы $\{лев_j | j < n\}$ используются для ввода координат последовательных точек в n -мерном пространстве. Каждый набор координат требуется умножить на фиксированный вектор V длины n , а полученное скалярное произведение выве-

сти направо; более строго, $прав \leq \sum_{j=0}^{n-1} V_j \times лев_j$. Кроме того, требуется, чтобы каждую микросекунду вводилось n координат очередной точки и выводилось одно скалярное произведение. Быстродействие каждого процесса в отдельности позволяет примерно за одну микросекунду выполнить ввод, умножение, сложение и вывод. Поэтому ясно, что потребуется по крайней мере n параллельных процессов. Значит, решение этой задачи должно строиться в виде итеративного вектора как минимум с n элементами.

Заменим в спецификации \sum -нотацию на обычное индуктивное определение:

$$\begin{aligned} средн_0 &= 0^* \\ средн_{j+1} &= V_j \times лев_j + средн_j \quad \text{для } j < n \\ прав &= средн_n \end{aligned}$$

Таким образом, мы разбили спецификацию на систему из $n+1$ уравнений, каждое из которых содержит не более одного умножения. Все, что теперь требуется, — это построить процесс для каждого уравнения.

$$\begin{aligned} УМН_0 &= (\mu X. средн_0!0 \rightarrow X) \\ УМН_{j+1} &= (\mu X. лев_j?x \rightarrow средн_j?y \rightarrow \\ &\quad \rightarrow средн_{j+1}!(V_j \times x + y) \rightarrow X) \quad \text{для } j < n \\ УМН_{n+1} &= (\mu X. средн_n?x \rightarrow прав!x \rightarrow X) \\ СЕТЬ &= \prod_{i < n+2} УМН_i \end{aligned}$$

Коммутационная схема изображена на рис. 4.4.

Х5. Этот пример похож на **Х4** с той разницей, что m различных скалярных произведений прежних наборов координат требуются практически одновременно. В целях эффективной реализации канал $лев_j$ (для $j < n$) используется для ввода j -го столбца бесконечного массива; затем он умножается на $(n \times m)$ -матрицу M , а i -й столбец результата выводится по каналу $прав_i$, где $i < m$. Это можно представить формулой $прав_i = \sum_{j < n} M_{ij} \times лев_j$. Координаты результата запрашиваются с прежней скоростью, и поэтому требуется как минимум $m \times n$ процессов.

Это решение могло бы найти практическое применение в графическом дисплее, автоматически преобразующем или даже вращающем двумерное представление трехмерного объекта. Форма объекта задается последовательностью точек

в абсолютном пространстве; итеративный массив применяется в линейных преобразованиях, вычисляющих вертикальное и горизонтальное отклонения электронного луча в кинескопе; третья выходная координата может, например, регулировать интенсивность луча.

Решение основано на схеме, изображенной на рис. 4.5. Каждый столбец этого массива (за исключением последнего) строится по принципу решения примера X4; но каждое вход-

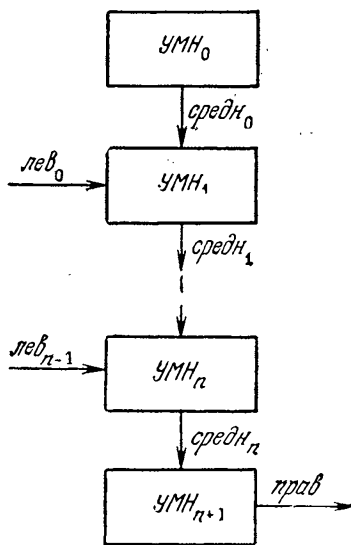


Рис. 4.4

ное значение, поступающее к нему по горизонтальному входному каналу, копируется и передается соседу по горизонтальному выходному каналу. Крайние правые процессы попросту сбрасывают получаемые ими значения. В целях экономии можно было бы переложить функции этих процессов на их левых соседей.

Подробности решения мы оставляем в качестве упражнения.

X6. Ввод по каналу c интерпретируется как последовательность цифр натурального числа C , выраженного через основание b , начинающаяся с самого младшего разряда. Величину вводимого числа определим как $C = \sum_{i \geq 0} c[i] \times b^i$, где $c[i] < b$ для всех i . Пусть M — фиксированный множитель и требуется, чтобы выход по каналу d представлял собой по-

Определим процесс $УМН1(z)$, имеющий перенос z в качестве параметра:

$$УМН1(z) = c?x \rightarrow d!(M \times x + z) \bmod b \rightarrow УМН1((M \times x + z) \div b)$$

Начальное значение z равно нулю, и поэтому искомым решением будет

$$УМН = УМН1(0).$$

Х7. Рассмотрим задачу, аналогичную **Х6**, но где M — многозначное число $M = \sum_{i < n} M_i \times b^i$. Один процессор может перемножать только однозначные числа. Вывод, однако, должен осуществляться со скоростью, не позволяющей тратить время на более чем одно умножение для каждой цифры. Следовательно, требуется по меньшей мере n процессоров. Для каждой цифры множителя M_i введем свой процесс $УЗЕЛ_i$.

В основе решения лежит традиционный алгоритм ручного перемножения многозначных чисел с тем, однако, отличием, что частичные суммы в нем прибавляются к следующему ряду таблицы немедленно:

....153091	C	входное число
253	M	множитель
....306182	$M_2 \times C$	вычисляется $УЗЕЛ_2$
....765455	$M_1 \times C$	} вычисляется $УЗЕЛ_1$
....827275	$25 \times C$	
....459273	$M_0 \times C$	} вычисляется $УЗЕЛ_0$
....732023	$M \times C$	

Узлы соединены, как показано на рис. 4.6. Начальное значение вводится по каналу c_0 и продвигается влево по c -ка-

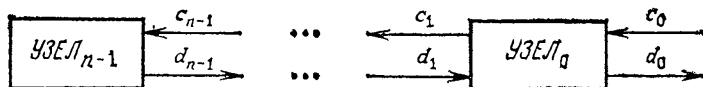


Рис. 4.6

налам. Частичные результаты продвигаются вправо по d -каналам, а окончательный результат выводится по каналу d_0 . К счастью, прежде, чем вступить во взаимодействие со своим левым соседом, каждый узел может выдать одну цифру результата. Более того, поведение самого левого узла можно описать как решение задачи **Х6**:

$$\begin{aligned} УЗЕЛ_{n-1}(z) &= c_{n-1}?x \rightarrow d_{n-1}!(M_{n-1} \times x + z) \bmod b \\ &\rightarrow УЗЕЛ_{n-1}((M_{n-1} \times x + z) \div b) \end{aligned}$$

Остальные узлы описываются аналогично, но только каждый из них передает входную цифру левому соседу, а результат левого соседа добавляет к собственному переносу. Для $k < n - 1$

$$\begin{aligned} \text{УЗЕЛ}_k(z) &= c_k ? x \rightarrow d_k ! (M_k \times x + z) \bmod b \rightarrow c_{k+1} ! x \rightarrow d_{k+1} ? y \\ &\rightarrow \text{УЗЕЛ}_k(y + (M_k \times x + z) \div b) \end{aligned}$$

Вся сеть целиком определяется как $\big\|_{i < n} \text{УЗЕЛ}_i(0)$.

Пример X7 — это простой пример из класса тех замысловатых сетевых алгоритмов, в основе которых лежит цикл в ориентированном графе коммуникационных каналов. Постановка задачи была сильно упрощена предположением, что множитель известен заранее и все время остается неизменным. На практике же гораздо чаще бывает так, что такого рода параметры вводятся по тому же каналу, что и последовательность данных, и должны вводиться заново всякий раз, когда их требуется изменить. При реализации этого нужна большая аккуратность, но не требуется особой изобретательности.

Простой способ реализации состоит во введении специального символа — скажем, *перезагр* (перезагрузить), указывающего, что следующее число следует рассматривать как смену параметра; если же число параметров является переменной величиной, можно ввести символ *конперезагр* (конец перезагрузки).

X8. Этот пример похож на X4 с той разницей, что параметры V_j требуется перезагружать значениями, следующими за символом *перезагр*. Определение УМН_{j+1} придется изменить, введя в него множитель в качестве параметра:

$$\begin{aligned} \text{УМН}_{j+1}(v) &= \text{лев}_j ? x \rightarrow \\ &\quad \text{if } x = \text{перезагр then } (\text{лев}_j ? y \rightarrow \text{УМН}_{j+1}(y)) \\ &\quad \text{else } (\text{средн}_j y \rightarrow \text{средн}_{j+1} ! (v \times x + y) \rightarrow \text{УМН}_{j+1}(v)) \end{aligned}$$

4.4. ТРАНСПОРТЕРЫ

В этом разделе мы сосредоточим внимание на процессах, имеющих в алфавите только два канала, а именно, входной канал *лев* и выходной канал *прав*. Такие процессы называются *транспортёрами* и графически могут быть представлены, как на рис. 4.7.

Процессы P и Q можно объединить так, чтобы правый канал P соединялся с левым каналом Q , а последовательность сообщений, выводимых P и вводимых Q по этому внут-

ренному каналу, была скрыта от их общего окружения. Результат такого объединения обозначается

$$P \gg Q$$

и изображается, как указано на рис. 4.8.

На этой коммутационной схеме соединяющий P и Q канал не имеет имени, что отражает его сокрытие. Кроме того,

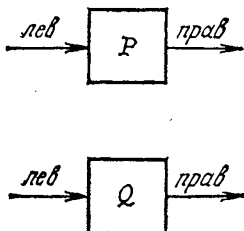


Рис. 4.7

из этой схемы видно, что все сообщения, принимаемые процессом $(P \gg Q)$ по правому каналу, вводятся P , а все сообщения, выдаваемые $(P \gg Q)$ по левому каналу, выводятся



Рис. 4.8

Q . И наконец, $(P \gg Q)$ сам по себе является транспортером и поэтому снова может быть объединен в цепочку с другими транспортерами:

$$(P \gg Q) \gg R, (P \gg Q) \gg (R \gg S) \text{ и т. д.}$$

По закону 4.4.1 **L1** операция \gg ассоциативна, и поэтому в дальнейшем при записи таких цепочек скобки мы будем опускать.

Чтобы применение операции сцепления процессов было корректным, необходимо соблюдать очевидное ограничение на алфавиты

$$\alpha(P \gg Q) = \alpha_{лев}(P) \cup \alpha_{прав}(Q)$$

Дальнейшее ограничение связано с утверждением, что по соединенным каналам может передаваться одна и та же информация:

$$\alpha_{прав}(P) = \alpha_{лев}(Q)$$

Примеры

X1. Транспортёр, выводящий каждое входное значение, умноженное на четыре (4.2 X2):

УЧЕТВ = УДВ ≫ УДВ

X2. Процесс, вводящий перфокарты размером 80 литер и выводящий их текст, упакованный в строки длиной 125 литер (4.2 X3, X4):

РАСПАК ≫ УПАК

Этот процесс довольно сложно записать в обычных терминах структурированного программирования, поскольку неясно, как организовать итерацию основного цикла — по числу входных перфокарт или по числу выходных строк. Джексон дал этой проблеме название *структурного конфликта*. Наше решение содержит отдельный цикл для каждого из двух процессов, что прекрасно соответствует структуре исходной задачи.

X3. Задача, аналогичная X2, но дополнительно каждая пара последовательных символов «*» заменяется символом «↑» (4.2 X5):

РАСПАК ≫ СЖИМ ≫ УПАК

В обычном последовательном программировании даже незначительное изменение спецификации может вызвать серьезные проблемы. Хорошо, что эти проблемы можно обойти, просто вставив дополнительный процесс. Эта разновидность модульности была введена и используется разработчиками операционных систем.

X4. Задача, аналогичная X2, но имеющая отличие в том, что считывание карт может продолжаться во время приостановки работы выводющего устройства, а позже вывод может продолжаться во время приостановки считывающего устройства (4.2 X9):

РАСПАК ≫ БУФЕР ≫ УПАК

Буфер накапливает литеры, выработанные процессом *РАСПАК*, но еще не потребленные процессом *УПАК*. Процессу *УПАК* они доступны для ввода в те промежутки времени, когда процесс *РАСПАК* временно приостанавливается. Таким образом, буфер сглаживает временные колебания скорости выработки значений и их потребления. Однако он не способен решить проблему длительного несоответствия между этими скоростями. Если устройство считывания пер-

фокарт в среднем медленнее, чем печатающее устройство, то буфер всегда будет пустым, и никакого сглаживающего эффекта достигнуто не будет. Если же считывающее устройство работает быстрее, то буфер будет бесконечно расти, пока не займет всю доступную ему свободную память.

Х5. Чтобы избежать нежелательного роста размеров буфера, обычно накладывают ограничение на число буферизованных сообщений. Даже одностороннего буфера, полученного с помощью процесса *КОПИР* (4.2 **Х1**), может оказаться достаточно. Приведем другой вариант примера **Х4**, где при вводе считывается одна лишняя карта, а при выводе буферизуется одна лишняя строка:

КОПИР \gg *РАСПАК* \gg *УПАК* \gg *КОПИР*

Заметим, что алфавиты двух вхождений процесса *КОПИР* различны — это должно быть понятно из контекста, в котором они употребляются.

Х6. Двойной буфер, принимающий до двух сообщений, прежде чем требовать вывода одного из них:

КОПИР \gg *КОПИР*

Его поведение очень напоминает поведение процесса *ЦЕПЬ2* (2.6 **Х4**) и даже *ТАП2* (1.1.3 **Х6**).

4.4.1. Законы

Самое полезное алгебраическое свойство сцепления — его ассоциативность:

L1. $P \gg (Q \gg R) = (P \gg Q) \gg R$

Остальные законы показывают, как в транспортерах реализуются ввод и вывод сообщений; они позволяют упрощать описание процессов за счет введения разновидности символических вычислений. Например, если процесс, стоящий слева от операции \gg , начинает с выдачи направо сообщения v , а процесс, стоящий справа от операции \gg , начинает с ввода слева, то сообщение v передается от первого процесса ко второму; действительное же взаимодействие является скрытым, как показывает следующий закон:

L2. $(\text{прав!}v \rightarrow P) \gg (\text{лев?}y \rightarrow Q(y)) = P \gg Q(v)$

Если один из процессов хочет взаимодействовать с другим, в то время как тот приготовился взаимодействовать с внешней обстановкой, то сначала происходит внешнее взаимодей-

ствие, а внутреннее откладывается до следующей возможности:

$$\mathbf{L3.} \quad (прав!v \rightarrow P) \gg (прав!w \rightarrow Q) = прав!w \rightarrow ((прав!v \rightarrow P) \gg Q)$$

$$\mathbf{L4.} \quad (лев?x \rightarrow P(x)) \gg (лев?y \rightarrow Q(y)) = лев?x \rightarrow (P(x) \gg (лев?y \rightarrow Q(y)))$$

Если оба процесса приготовились к внешним взаимодействиям, то первым может произойти любое из них:

$$\mathbf{L5.} \quad (лев?x \rightarrow P(x)) \gg (прав!w \rightarrow Q) = (лев?x \rightarrow (P(x) \gg (прав!w \rightarrow Q))) \\ | прав!w \rightarrow ((лев?x \rightarrow P(x)) \gg Q))$$

Закон **L5** справедлив и в том случае, когда оператор \gg заменяется на $\gg R$, поскольку транспортёры внутри цепочки не могут непосредственно взаимодействовать с обстановкой:

$$\mathbf{L6.} \quad (лев?x \rightarrow P(x)) \gg R \gg (прав!w \rightarrow Q) = \\ (лев?x \rightarrow (P(x) \gg R \gg (прав!w \rightarrow Q))) \\ | прав!w \rightarrow ((лев?x \rightarrow P(x)) \gg R \gg Q))$$

Аналогично можно обобщить и другие законы.

L7. Если R — цепочка процессов, каждый из которых начинается с вывода направо, то

$$R \gg (прав!w \rightarrow Q) = прав!w \rightarrow (R \gg Q)$$

L8. Если R — цепочка процессов, каждый из которых начинается с ввода слева, то

$$(лев?x \rightarrow P(x)) \gg R = лев?x \rightarrow (P(x) \gg R)$$

Примеры

X1. Определим

$$R(y) = (прав!y \rightarrow КОПИР) \gg КОПИР$$

$$\text{Тогда } R(y) = (прав!y \rightarrow КОПИР) \gg (лев?x \rightarrow прав!x \rightarrow КОПИР) \\ \text{по опр. КОПИР} \\ = КОПИР \gg (прав!y \rightarrow КОПИР) \quad \mathbf{L2}$$

X2. $КОПИР \gg КОПИР$

$$= (лев?x \rightarrow прав!x \rightarrow КОПИР) \gg КОПИР \\ \text{по определению КОПИР} \\ = лев?x \rightarrow ((прав!x \rightarrow КОПИР) \gg КОПИР) \quad \mathbf{L4} \\ = лев?x \rightarrow R(x) \quad \text{по определению } R(x)$$

X3. Из последней строки **X1** можно вывести, что

$$R(y) = (лев?x \rightarrow прав!x \rightarrow КОПИР) \gg (прав!y \rightarrow КОПИР) \\ = (лев?x \rightarrow ((прав!x \rightarrow КОПИР) \gg (прав!y \rightarrow КОПИР))) \\ | прав!y \rightarrow (КОПИР \gg КОПИР) \quad \mathbf{L5} \\ \Leftarrow (лев?x \rightarrow прав!y \rightarrow R(x)) \quad \mathbf{L3} \\ | прав!y \rightarrow лев?x \rightarrow R(x) \quad \mathbf{X2}$$

Этот пример показывает, что после принятия первого сообщения двойной буфер готов либо выдать его, либо ввести перед этим еще одно сообщение. Рассуждения в этих доказательствах очень напоминают 2.3.1 X1.

4.4.2. Реализация

При реализации ($P \gg Q$) различают три случая:

(1) Если процессы готовы к взаимодействию по соединяющему их каналу, то это взаимодействие происходит немедленно без учета внешней обстановки. Если возможна бесконечная последовательность таких событий, то процесс расходится (разд. 3.5.2).

(2) Иначе, если обстановка заинтересована во взаимодействии по левому каналу, она взаимодействует с P .

(3) Или же, если обстановку интересует правый канал, она взаимодействует с Q .

Разъяснение смысла операций ввода и вывода можно найти в разд. 4.2.1.

```

цепь( $P, Q$ ) =
  if  $P("прав") \neq "BLEEP \& Q("лев") \neq "BLEEP$ 
    then  $цепь(cdr(P("прав")), Q("лев")(car(P("прав"))))$ 
                                                    случай (1)
  else  $\lambda x. \text{ if } x = "прав$ 
    then if  $Q("прав") = "BLEEP$  then  $"BLEEP$ 
          else  $cons(car(Q("прав")),$ 
                     $цепь(P, cdr(Q("прав"))))$ 
                                                    случай (2)
    else if  $x = "лев$ 
      then if  $P(x) = "BLEEP$  then  $"BLEEP$ 
            else  $\lambda y. цепь(P("лев")(y), Q)$ 
                                                    случай (3)
    else  $"BLEEP$ 

```

4.4.3. Замыкание

Операция сцепления соединяет процессы только одним каналом и поэтому не угрожает системе дедлоком. Если P и Q — бесконечные процессы, то ($P \gg Q$) также не останавливается. К сожалению, возникает новая опасность, а именно что P и Q замкнутся друг на друга, т. е. будут проводить все время во взаимодействии друг с другом, и, значит, ($P \gg Q$) не будет сообщаться с внешним миром. Этот вариант расходимости (разд. 3.5.1, 3.8) иллюстрируется тривиальным примером

```

 $P = (прав!1 \rightarrow P)$ 
 $Q = (лев?x \rightarrow Q)$ 

```

Очевидно, что $(P \gg Q)$ совершенно бесполезный процесс; он даже хуже, чем *СТОП*, так как, представляя собой бесконечный цикл, он может потреблять неограниченное количество машинных ресурсов, ничего не производя взамен. Менее тривиальный пример — это $(P \gg Q)$, где

$$P = (\text{прав!}1 \rightarrow P \mid \text{лев?}x \rightarrow P1(x))$$

$$Q = (\text{лев?}x \rightarrow Q \mid \text{прав!}1 \rightarrow Q1)$$

В этом примере расходимость возникает вследствие одной лишь возможности бесконечного внутреннего взаимодействия; она существует несмотря на то, что выбор внешнего взаимодействия по правому и по левому каналам предлагается при каждом возможном случае, и даже на то, что после такого внешнего взаимодействия дальнейшее поведение $(P \gg Q)$ исключает расходимость.

Простой способ доказать, что $(P \gg Q)$ свободен от замыкания, — это показать, что P *предварен слева* в том смысле, что он не может выдать направо бесконечный ряд сообщений, не прерываемый вводом по левому каналу. Чтобы в этом убедиться, мы должны доказать, что длина выводимой направо последовательности все время ограничена сверху некоторой хорошо определенной функцией f , зависящей от последовательности значений, вводимых слева; или, более формально, мы определяем:

$$P \text{ предварен слева} \equiv \exists f. P \text{ уд } (\# \text{ прав} \leq f(\text{лев}))$$

Свойство предваренности слева часто бывает очевидным из самого текста P .

L1. Если каждая рекурсия в определении P предварена вводом слева, то P предварен слева.

L2. Если P предварен слева, то $(P \gg Q)$ свободен от замыкания. В точности такие же рассуждения применимы к предваренности справа второго операнда \gg .

L3. Если Q предварен справа, то $(P \gg Q)$ свободен от замыкания.

Примеры

X1. Следующие процессы согласно закону **L1** предварены слева (4.1 **X1**, **X2**, **X5**, **X9**):

КОПИР, УДВ, СЖИМ, БУФЕР.

Х2. Следующие процессы предварены слева, что непосредственно следует из их определения, потому что

$$\begin{aligned} \text{РАСПАК уд } \# \text{ прав} &\leq \# (\wedge / \text{лев}) \\ \text{УПАК уд } \# \text{ прав} &\leq \# \text{ лев} \end{aligned}$$

Х3. *БУФЕР* не предварен справа, потому что он может вводить сколь угодно много сообщений слева, не выдавая ничего направо.

4.4.4. Спецификации

Часто спецификацию транспортера удастся выразить как отношение $S(\text{лев}, \text{прав})$ между последовательностью сообщений, вводимых с левого канала, и последовательностью сообщений, выводимых направо. Когда два транспортера соединяются в цепочку, производимая левым операндом последовательность *прав* отождествляется с потребляемой правым операндом последовательностью *лев*, и эта общая последовательность затем скрывается. Все, что известно о скрытой последовательности, — это лишь то, что она существует. Но мы должны предупредить риск замыкания, и поэтому вводим правило

Л1. Если P уд $S(\text{лев}, \text{прав})$, а Q уд $T(\text{лев}, \text{прав})$ и P предварен слева или Q предварен справа, то $(P \gg Q)$ уд $\exists s. S(\text{лев}, s) \& T(s, \text{прав})$.

Этот закон утверждает, что отношение между *лев* и *прав*, устанавливаемое процессом $(P \gg Q)$, представляет собой обычную композицию отношений для P и для Q . Так как операция \gg не приводит к дедлоку в транспортерах, мы можем позволить себе опустить рассуждения, связанные с отказами.

Примеры

Х1. $\text{УДВ уд прав} \stackrel{1}{\leq} \text{удвоить}^*(\text{лев})$
 УДВ предварен слева и справа

и поэтому

$$\begin{aligned} (\text{УДВ} \gg \text{УДВ}) \text{ уд } \exists s. (s &\stackrel{1}{\leq} \text{удвоить}^*(\text{лев}) \& \text{прав} \stackrel{1}{\leq} \text{удвоить}^*(s)) \\ &\equiv \text{прав} \stackrel{2}{\leq} \text{удвоить}^*(\text{удвоить}^*(\text{лев})) \\ &\equiv \text{прав} \stackrel{2}{\leq} \text{учетвер}^*(\text{лев}) \end{aligned}$$

X2. Переопределим БУФЕР, используя рекурсию и операцию \gg :

$$\text{БУФ} = \mu X. (\text{лев?}x \rightarrow (X \gg (\text{прав!}x \rightarrow \text{КОПИР})))$$

Мы хотим доказать, что

$$\text{БУФ} \text{ уд } \text{прав} \leq \text{лев}$$

Предположим, что

$$X \text{ уд } \# \text{лев} \geq n \vee \text{прав} \leq \text{лев}$$

Мы знаем, что

$$\text{КОПИР} \text{ уд } \text{прав} \ll \text{лев}$$

$$\begin{aligned} \text{Поэтому } (\text{прав!}x \rightarrow \text{КОПИР}) \text{ уд } ((\text{прав} = \text{лев} = \langle \rangle \\ \vee (\text{прав} \geq \langle x \rangle \& \text{прав}' \leq \text{лев})) \Rightarrow \text{прав} \leq \langle x \rangle^{\sim} \text{лев}) \end{aligned}$$

Поскольку правый операнд предварен справа, из **L1** и нашего предположения следует, что

$$\begin{aligned} (X \gg (\text{прав!}x \rightarrow \text{КОПИР})) \text{ уд } (\exists s. (\# \text{лев} \geq n \vee s \leq \text{лев}) \& \\ \& \text{прав} \leq \langle x \rangle^{\sim} s) \\ \Rightarrow (\# \text{лев} \geq n \vee \text{прав} \leq \langle x \rangle^{\sim} \text{лев}) \end{aligned}$$

$$\begin{aligned} \text{Поэтому } \text{лев?}x \rightarrow (\dots) \text{ уд } \text{прав} = \text{лев} = \langle \rangle \\ \vee (\text{лев} > \langle \rangle \& (\# \text{лев}' \geq n \vee \\ \vee \text{прав} \leq \langle \text{лев}_0 \rangle^{\sim} \text{лев}')) \\ \Rightarrow \# \text{лев} \geq n + 1 \vee \text{прав} \leq \text{лев} \end{aligned}$$

Требуемое заключение следует из правила доказательства для рекурсивных процессов (3.7.1 **L8**). Более простой закон (1.10.2 **L6**) здесь не подойдет, поскольку предваренность рекурсии неочевидна.

4.4.5. Буферы и протоколы

Буфер — это такой процесс, который выводит направо в точности ту последовательность сообщений, которую он ввел слева, но, возможно, после некоторой задержки; более того, будучи непустым, он всегда готов к выводу направо. Более строго, определим буфер как процесс P , который не останавливается, свободен от замыкания и удовлетворяет спецификации

$$P \text{ уд } (\text{прав} \leq \text{лев}) \& (\text{if } \text{прав} = \text{лев} \text{ then } \text{лев} \approx \text{отк} \text{ else } \text{прав} \approx \text{отк})$$

Условие $s \approx \text{отк}$ означает здесь, что процесс не может отказаться от взаимодействия по каналу s (разд. 3.7, 3.4). Отсюда следует, что все буферы предварены слева.

Пример

X1. Следующие процессы являются буферами:

КОПИР, (КОПИР \gg КОПИР), БУФ, БУФЕР.

Очевидно, что буферы полезны для хранения информации, ожидающей обработки. Но еще полезнее они для спецификации желаемого поведения протокола связей, чье назначение — передавать сообщения в том же порядке, в котором они поступают. Такой протокол состоит из двух процессов — передатчика T и приемника R , соединенных в цепочку ($T \gg R$). Ясно, что если протокол правильный, то ($T \gg R$) должен быть буфером.

На практике приемник и передатчик соединяются достаточно длинным проводом, что приводит к разрушению и утере посылаемых сообщений. Поэтому поведение самого провода можно промоделировать процессом *ПРОВОД*, ведущим себя не совсем как буфер. В задачу разработчика протокола входит обеспечить, чтобы система в целом вела себя как буфер, несмотря на возможные неисправности в работе провода, т. е. ($T \gg \text{ПРОВОД} \gg R$) был бы буфером.

Обычно транспортный протокол строится как некоторое количество слоев (T_1, R_1) , (T_2, R_2) , ..., (T_n, R_n) , каждый из которых использует предыдущий слой как среду для взаимодействия:

$$T_n \gg \dots \gg (T_2 \gg (T_1 \gg \text{ПРОВОД} \gg R_1) \gg R_2) \gg \dots \gg R_n.$$

Конечно, в практической реализации протокола все передатчики собраны в единый передатчик на одном конце, а все приемники — на другом, что соответствует иной расстановке скобок:

$$(T_n \gg \dots \gg T_2 \gg T_1) \gg \text{ПРОВОД} \gg (R_1) \gg R_2 \gg \dots \gg R_n).$$

Закон ассоциативности операции \gg гарантирует, что эта перегруппировка не меняет поведения системы.

На практике протоколы должны быть сложнее, ибо однонаправленный поток сообщений недостаточен для достижения надежной связи по ненадежному проводу: необходимы дополнительные каналы в обратном направлении, по которым приемник мог бы посылать подтверждение благополучного приема сигнала, с тем чтобы неподтвержденные сигналы передавались повторно.

Следующие законы полезны при доказательстве правильности транспортных протоколов. Они принадлежат А. У. Роско,

- L1.** Если P и Q — буферы, то $(P \gg Q)$ и $(\text{лев?}x \rightarrow (P \gg (\text{прав!}x \rightarrow Q)))$ также являются буферами.
- L2.** Если $T \gg R = (\text{лев?}x \rightarrow (T \gg (\text{прав!}x \rightarrow R)))$, то $(T \gg R)$ является буфером.
Следующий закон является обобщением **L2**:
- L3.** Если для некоторой функции f и для всех z
 $(T(z) \gg R(z)) = (\text{лев?}x \rightarrow (T(f(x, z)) \gg (\text{прав!}x \rightarrow R(f(x, z))))$,
 то $T(z) \gg R(z)$ является буфером для всех z .

Примеры

- X1.** Согласно закону **L1** следующие процессы являются буферами:
 $\text{КОПИР} \gg \text{КОПИР}$, $\text{БУФЕР} \gg \text{КОПИР}$,
 $\text{КОПИР} \gg \text{БУФЕР}$, $\text{БУФЕР} \gg \text{БУФЕР}$.
- X2.** В примерах 4.4.1 **X1** и **X2** было показано, что
 $(\text{КОПИР} \gg \text{КОПИР}) = (\text{лев?}x \rightarrow (\text{КОПИР} \gg (\text{прав!}y \rightarrow \text{КОПИР})))$

Значит, по закону **L2** он является буфером.

X3. Фазовое кодирование

Устройство фазового кодирования — это процесс T , который вводит поток битов и выводит $\langle 0, 1 \rangle$ для каждого входного 0 и $\langle 1, 0 \rangle$ для каждой входной 1. Дешифратор R производит обратный перевод:

$$\begin{aligned} T &= \text{лев?}x \rightarrow \text{прав!}x \rightarrow \text{прав!}(1 - x) \rightarrow T \\ R &= \text{лев?}x \rightarrow \text{лев?}y \rightarrow \text{if } y = x \text{ then НЕОПР else } (\text{прав!}x \rightarrow R) \end{aligned}$$

где **НЕОПР** — процесс, неопределенный слева.

Мы хотим с помощью **L2** доказать, что $(T \gg R)$ является буфером.

$$\begin{aligned} (T \gg R) &= \text{лев?}x \rightarrow ((\text{прав!}x \rightarrow \text{прав!}(1 - x) \rightarrow T) \\ &\quad \gg (\text{лев?}x \rightarrow \text{лев?}y \rightarrow \text{if } y = x \text{ then НЕОПР} \\ &\quad \text{else } (\text{прав!}x \rightarrow R))) \\ &= \text{лев?}x \rightarrow (T \gg \text{if } (1 - x) = x \text{ then НЕОПР} \\ &\quad \text{else } (\text{прав!}x \rightarrow R)) \\ &= \text{лев?}x \rightarrow (T \gg (\text{прав!}x \rightarrow R)) \end{aligned}$$

Значит, по **L2**, $(T \gg R)$ является буфером.

X4. Вставка битов

Передачик T исправно передает входные биты слева направо, но после каждой тройки следующих друг за другом единичных битов он дополнительно вставляет один нулевой

бит. Так, получив на входе 01011110, он выводит 010111010. Приемник R удаляет эти лишние нули. Требуется доказать, что $(T \gg R)$ является буфером. Построение T , R и доказательство корректности мы оставляем в качестве упражнения.

Х5. Общая линия

Мы хотим копировать данные с канала *лев1* в *прав1* и с *лев2* в *прав2*. Проще всего достигнуть этого с помощью двух протоколов, каждый из которых использует свой провод. Но, к сожалению, в нашем распоряжении находится лишь один канал *средн*, по которому и должна осуществляться передача обоих потоков данных, как показано на рис. 4.9.



Рис. 4.9

Прежде чем передать сообщение по каналу *средн*, T снабжает его специальным признаком (тегом), а R удаляет этот признак и выводит сообщение по соответствующему правому каналу:

$$\begin{aligned}
 T &= (\text{лев1?}x \rightarrow \text{средн!тег1}(x) \rightarrow T \\
 &\quad \mid \text{лев2?}y \rightarrow \text{средн!тег2}(y) \rightarrow T) \\
 R &= \text{средн?}z \rightarrow \text{if } \text{тег}(z) = 1 \text{ then } (\text{прав1!растег}(z) \rightarrow R) \\
 &\quad \text{else } (\text{прав2!растег}(z) \rightarrow R)
 \end{aligned}$$

Но это решение не вполне удовлетворительно. Если два сообщения вводятся по *лев1*, а получатель еще не готов к их приему, то всей системе приходится ждать, и сообщения между *лев2* и *прав2* могут надолго задержаться. Введение буферов откладывает эту проблему лишь очень ненадолго. Правильным решением будет введение еще одного канала в обратном направлении, по которому R посылал бы T сигнал приостановить тот поток сообщений, потребность в котором в настоящий момент невелика. Это называется управлением потоками.

4.5. ПОДЧИНЕНИЕ

Пусть P и Q — процессы, такие что $\alpha P \subseteq \alpha Q$. В комбинации $(P \parallel Q)$ каждое действие P может произойти, только если это позволяет Q , тогда как Q может независимо осуществлять действия из $(\alpha Q - \alpha P)$ без разрешения и даже без ведома своего партнера P . Таким образом, P служит по отношению

к Q подчиненным процессом, тогда как Q выступает как основной или главный процесс. Когда отношения основного и подчиненного процессов требуется скрыть от их общего окружения, мы будем применять асимметричное обозначение $P//Q$. В терминах оператора сокрытия это можно определить как

$$P//Q = (P \parallel Q) \setminus \alpha P$$

Это обозначение используется, только если $\alpha P \subseteq \alpha Q$, и тогда

$$\alpha(P//Q) = (\alpha Q - \alpha P)$$

Обычно бывает удобно давать подчиненному процессу имя, скажем m , которое используется основным процессом для всех взаимодействий с подчиненным. Способы именования процессов, описанные в разд. 2.6.2, легко расширить для взаимодействующих процессов, введя составные имена каналов. Они будут иметь вид $m.c$, где m — имя процесса, а c — имя одного из его каналов. Каждое взаимодействие по этому каналу представляет собой тройку $m.c.v$, где $am.c(m:P) = ac(P)$, а $v \in ac(P)$.

В конструкции $(m:P//Q)$ процесс Q взаимодействует с P по каналам с составными именами вида $m.c$ и $m.d$, тогда как P для этих же взаимодействий использует соответствующие простые каналы c и d . Так, например,

$$(m:(c!v \rightarrow P) // (m.c?x \rightarrow Q(x))) = (m:P//Q(v))$$

Так как все эти взаимодействия скрыты от обстановки, имя m недоступно снаружи; следовательно, оно служит локальным именем подчиненного процесса.

Подчинение может быть вложенным, например $(n:(m:P//Q)//R)$. В этом случае все вхождения событий, содержащих имя m , скрываются прежде, чем имя n приписывается оставшимся событиям, каждое из которых входит в алфавит Q и не входит в алфавит P . Процесс R не имеет возможности ни непосредственно взаимодействовать с P , ни даже знать о существовании P и его имени m .

Примеры

X1. удв: $УДВ//Q$

(описание $УДВ$ см. в 4.2 **X2**)

Подчиненный процесс ведет себя как обыкновенная подпрограмма, вызываемая внутри основного процесса Q . Внутри Q значение $2 \times e$ может быть получено поочередными выводом

аргумента e по левому каналу процесса $удв$ и вводом результата по правому каналу:

$$удв.лев!e \rightarrow (удв.прав?x \rightarrow \dots)$$

X2. Одна подпрограмма может использовать другую как подчиненную и делать это неоднократно:

$$\begin{aligned} УЧЕТВ = \\ (удв : УДВ // (\mu X. лев?x \rightarrow удв.лев!x \rightarrow \\ удв.прав?y \rightarrow удв.лев!y \rightarrow \\ удв.прав?z \rightarrow прав!z \rightarrow X)) \end{aligned}$$

Сама она также может использоваться как подпрограмма:

$$учетв: УЧЕТВ // Q$$

Эта версия $УЧЕТВ$ напоминает 4.4 **X1**, но не имеет эффекта двойной буферизации.

X3. Обычную программную переменную с именем t можно промоделировать подчиненным процессом $t : ПЕРЕМ // Q$. Внутри основного процесса Q значение t может задаваться, считываться и обновляться с помощью ввода и вывода, как описано в 2.6.2 **X2**:

$$\begin{aligned} t := 3; P & \quad \text{реализуется выражением } (t.лев!3 \rightarrow P) \\ x := t; P & \quad \text{реализуется выражением } (t.прав?x \rightarrow P) \\ t := t + 3; P & \quad \text{реализуется выражением} \\ & \quad (t.прав?y \rightarrow t.лев!(y + 3) \rightarrow P) \end{aligned}$$

Подчиненный процесс может использоваться для реализации структур данных, обладающих более сложным поведением, чем простая переменная.

X4. ($q : БУФЕР // Q$) (см. 4.2 **X9**)

Подчиненный процесс служит здесь бесконечной очередью с именем q . Внутри Q вывод $q.лев!v$ добавляет v в один конец очереди, а $q.прав?u$ удаляет элемент с другого конца, а его значение присваивает u . Если очередь пуста, она никак не реагирует на эти команды, и в системе может наступить дедлок.

X5. Стек с именем $ст$ (см. 4.2 **X10**)
описывается как $ст : СТЕК // Q$

Внутри основного процесса Q $ст.лев!v$ используется для проталкивания v в верхушку стека, а $ст.прав?x$ выталкивает верхнее значение. Использование конструкции выбора позволяет рассматривать ситуацию, когда стек пуст:

$$(ст.прав?x \rightarrow Q1(x) \mid ст.пуст \rightarrow Q2)$$

Если стек не пуст, то выбирается первая альтернатива; если пуст, то выбор второй альтернативы позволяет избежать дедлока.

Подчиненный процесс, имеющий несколько каналов, может использоваться несколькими параллельными процессами при условии, что никакие два из них не пользуются одним и тем же каналом.

Х6. Процесс Q используется для передачи потока значений процессу R ; для того чтобы вывод Q не приостанавливался, когда R не готов к приему, эти значения буферизуются подчиненным буферным процессом b . Процесс Q использует канал $b.лев$ для вывода, а R использует $b.прав$ для ввода:

$$(b : БУФЕР // (Q \parallel R))$$

Заметим, что если R пытается вводить из пустого буфера, то это не обязательно приводит к дедлоку; R будет просто приостановлен до тех пор, пока Q не передаст буферу следующее значение. (Если Q и R взаимодействуют с буфером по одному и тому же каналу, то тогда этот канал должен быть в алфавите обоих процессов; при этом по определению операции \parallel ввод и вывод передаваемого значения должны осуществляться ими одновременно, что было бы совершенно неправильно.)

Оператор подчинения может использоваться для рекурсивного определения подпрограмм. Каждый уровень рекурсии (кроме последнего) задает *новую* локальную подпрограмму, работающую с рекурсивным вызовом (вызовами).

Х7. Факториал

$$\begin{aligned} \Phi АКТ = \mu X. лев?n \rightarrow & (\text{if } n = 0 \text{ then } (прав!1 \rightarrow X) \\ & \text{else } (f : X // (f.лев!(n - 1) \rightarrow f.прав?y \rightarrow \\ & \hspace{10em} \rightarrow прав!(n \times y) \rightarrow X)) \end{aligned}$$

Подпрограмма $\Phi АКТ$ использует каналы $лев$ и $прав$ для обмена результатами и параметрами с вызывающим ее процессом, а каналы $f.лев$ и $f.прав$ — для взаимодействия со своим подчиненным процессом f . В этом отношении она аналогична подпрограмме $УЧЕТВ(Х2)$. Единственное ее отличие состоит в том, что подчиненный процесс изоморфен самому процессу $\Phi АКТ$.

Этот знакомый и изрядно надоевший пример рекурсии выражен нами в непривычной и весьма громоздкой системе обозначений. Еще менее привычна идея использования рекурсии и подчинения для реализации бесконечной структуры дан-

ных. На каждом уровне рекурсии хранится лишь отдельная компонента этой структуры, а также описывается *новая* локальная подчиненная структура данных для работы с остальной ее частью.

Х8. Неограниченное конечное множество

Процесс, реализующий множество, вводит его элементы по левому каналу. После каждого ввода он выводит *ДА*, если такое значение уже вводилось, и *НЕТ* в противном случае. Это множество очень похоже на множество из примера 2.6.2 **Х4** с той разницей, что оно может хранить сообщения любого вида:

$$\text{МНОЖ} = \text{лев?}x \rightarrow \text{прав!НЕТ} \rightarrow (\text{ост} : \text{МНОЖ} // \text{ЦИКЛ}(x))$$

где $\text{ЦИКЛ}(x) =$

$$\begin{aligned} \mu X. \text{лев?}y \rightarrow (\text{if } y = x \text{ then } \text{прав!ДА} \rightarrow X \\ \text{else } (\text{ост.лев!}y \rightarrow \text{ост.прав?}z \rightarrow \\ \rightarrow \text{прав!}z \rightarrow X)) \end{aligned}$$

Вначале множество пусто, и потому на ввод первого элемента x оно немедленно отвечает *НЕТ*. Затем заводится подчиненный процесс *ост*, в котором должны храниться все элементы множества, кроме x . *ЦИКЛ* предназначен для ввода последовательных элементов множества. Если вновь введенный элемент равен x , по правому каналу сразу же посылается сообщение *ДА*. В противном случае новый элемент передается на хранение процессу *ост*. Затем *ост* передает назад ответ (*ДА* или *НЕТ*), и *ЦИКЛ* повторяется.

Х9. Двоичное дерево

Наиболее эффективным представлением множества служит двоичное дерево, основанное на некотором заданном на его элементах полном порядке \leq . В каждой вершине хранится первый помещенный туда элемент, а также определены два подчиненных дерева, одно для хранения элементов меньших, чем первый, а другое для хранения больших элементов. Внешняя спецификация дерева похожа на **Х8**:

$$\begin{aligned} \text{ДЕРЕВО} = \text{лев?}x \rightarrow \text{прав!НЕТ} \rightarrow \\ (\text{меньш} : \text{ДЕРЕВО} // (\text{больш} : \text{ДЕРЕВО} // \text{ЦИКЛ})) \end{aligned}$$

Построение процесса *ЦИКЛ* мы оставляем в качестве упражнения.

4.5.1. Законы

Взаимодействия между процессом и его подчиненными происходят согласно следующим очевидным законам. Первый из них описывает скрытые взаимодействия в обоих направ-

лениях между основным и подчиненным процессом:

$$\mathbf{L1A.} \quad (m : (c?x \rightarrow P(x))) // (m.c!v \rightarrow Q) = (m : P(v)) // Q$$

$$\mathbf{L1B.} \quad (m : (d!v \rightarrow P)) // (m.d?x \rightarrow Q(x)) = (m : P) // Q(v)$$

Если канал b не помечен именем m , то взаимодействия основного процесса по b никак не влияют на подчиненный процесс:

$$\mathbf{L2.} \quad (m : P // (b!e \rightarrow Q)) = (b!e \rightarrow (m : P // Q))$$

Единственный, кто может делать выбор за подчиненный процесс, — это его основной процесс:

$$\mathbf{L3.} \quad (m : (c?x \rightarrow P1(x) | d?y \rightarrow P2(y))) // (m.c!v \rightarrow Q) = (m : P1(v) // Q)$$

Если у двух подчиненных процессов одинаковые имена, то один из них недоступен:

$$\mathbf{L4.} \quad m : P // (m : Q // R) = (m : Q // R)$$

Обычно не имеет значения порядок, в котором записаны подчиненные процессы:

L5. Если имена m и n различны, то

$$m : P // (n : Q // R) = n : Q // (m : P // R)$$

Использование при определении подчиненных процессов рекурсии достаточно удивительно, чтобы вызвать сомнения в том, что это будет работать. Частично сомнения можно рассеять, продемонстрировав, как эта комбинация будет разворачиваться. В приведенном ниже примере рассматривается некоторый протокол поведения процесса и показано, как этот протокол получается. Еще более важно то, что из этого примера видно, как не могут получаться другие, слегка отличные от него протоколы.

Пример

X1. Типичный протокол процесса *МНОЖ* — это

$$s = \langle \text{лев.1, прав.НЕТ, лев.2, прав.НЕТ} \rangle$$

Значение *МНОЖ/s* можно вычислить последовательностью шагов, используя **L1** и **L2**:

$$\text{МНОЖ} / \langle \text{лев.1} \rangle = \text{прав!НЕТ} \rightarrow (\text{ост} : \text{МНОЖ} // \text{ЦИКЛ(1)})$$

поэтому

$$\begin{aligned}
 & \text{МНОЖ} / \langle \text{лев.1, прав.НЕТ} \rangle = (\text{ост} : \text{МНОЖ} // \text{ЦИКЛ}(1)) \\
 \text{а } & \text{МНОЖ} / \langle \text{лев.1, прав.НЕТ, лев.2} \rangle \\
 & = (\text{ост} : \text{МНОЖ} // (\text{ост.лев!2} \rightarrow \text{ост.прав?z} \rightarrow \text{прав!z} \rightarrow \\
 & \quad \rightarrow \text{ЦИКЛ}(1))) \\
 & = (\text{ост} : (\text{прав!НЕТ} \rightarrow (\text{ост} : \text{МНОЖ} // \text{ЦИКЛ}(2)))) \\
 & \quad // (\text{ост.прав?z} \rightarrow \text{прав!z} \rightarrow \text{ЦИКЛ}(1))) \\
 & = \text{ост} : (\text{ост} : \text{МНОЖ} // \text{ЦИКЛ}(2)) // (\text{прав!НЕТ} \rightarrow \text{ЦИКЛ}(1))
 \end{aligned}$$

Следовательно,

$$\text{МНОЖ} / s = \text{ост} : (\text{ост} : \text{МНОЖ} // \text{ЦИКЛ}(2)) // \text{ЦИКЛ}(1)$$

Отсюда очевидно, что

$$\langle \text{лев.1, прав.НЕТ, лев.2, прав.ДА} \rangle$$

не является протоколом МНОЖ.

Читатель может проверить, что

$$\begin{aligned}
 & \text{МНОЖ} / s \sim \langle \text{лев.2, прав.ДА} \rangle = \text{МНОЖ} / s, \\
 \text{а } & \text{МНОЖ} / s \sim \langle \text{лев.5, прав.НЕТ} \rangle = \text{ост} : (\text{ост} : (\text{ост} : \text{МНОЖ} \\
 & \quad // \text{ЦИКЛ}(5)) // \text{ЦИКЛ}(2)) // \text{ЦИКЛ}(1)
 \end{aligned}$$

4.5.2. Схемы коммутаций

Подчиненный процесс можно изображать внутри прямоугольника, соответствующего использующему его процессу, как показано на рис. 4.10 для примера 4.5 X1.

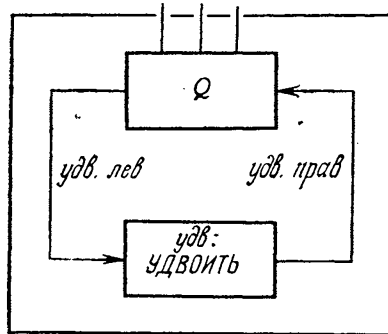


Рис. 4.10

Вложенным подчиненным процессам соответствует система вложенных прямоугольников, как на рис. 4.11 для примера 4.5 X2. Рекурсивный процесс вложен сам в себя, что можно представить себе как известный пример мастерской

художника, где на мольберте стоит законченное полотно, изображающее стоящее на мольберте законченное полотно, изображающее... Такую картину никогда не удалось бы завершить. К счастью, рекурсивному процессу и не требуется «завершать картину» — в ходе исполнения он автоматически разворачивается до нужной глубины. Поэтому последователь-

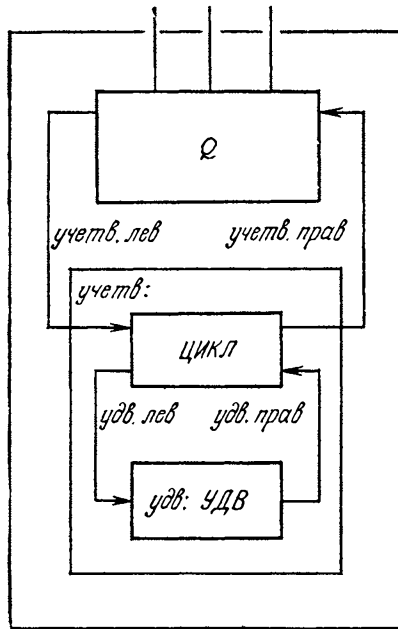


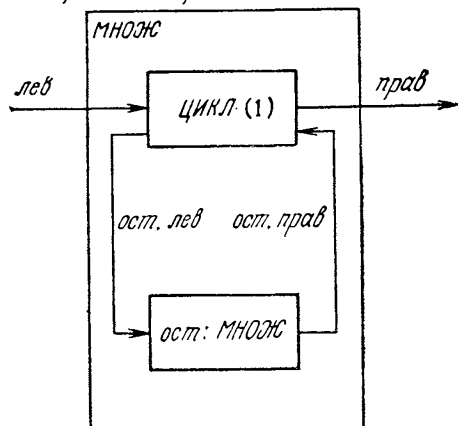
Рис. 4.11

ные стадии процесса *МНОЖ* на раннем этапе его работы (см. 4.5.1 *X1*) можно изобразить, как на рис. 4.12.

Если не учитывать вложенность прямоугольников, то это же можно изобразить в виде линейной структуры, как на рис. 4.13. Аналогично *ДЕРЕВО* (пример 4.5 *X9*) можно изобразить, как на рис. 4.14.

Коммутационные схемы подсказывают способ построения соответствующих сетей из аппаратных компонентов, где прямоугольники соответствуют интегральным схемам, а стрелки — соединяющим их проводам. Разумеется, в каждой практической реализации, прежде чем сеть сможет начать нормально функционировать, рекурсия должна быть развернута до некоторого конечного предела, и если в процессе выполнения этот предел окажется превышен, дальнейшая нормальная ра-

МНОЖ/⟨лев. 1, прав. НЕТ⟩ =



МНОЖ/s =

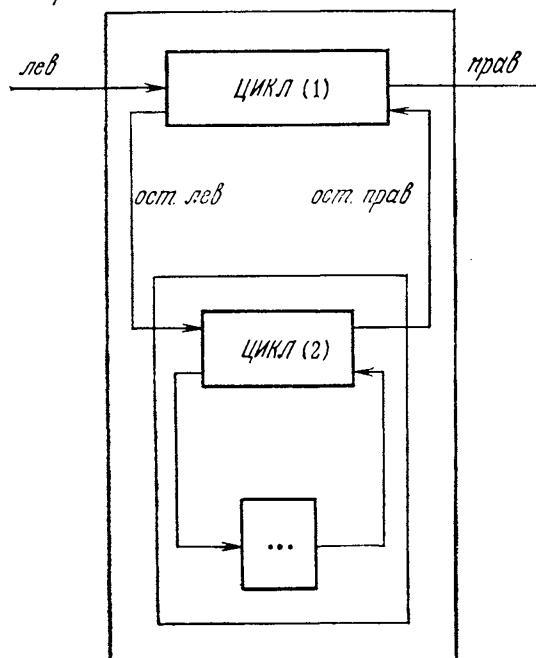


Рис. 4.12

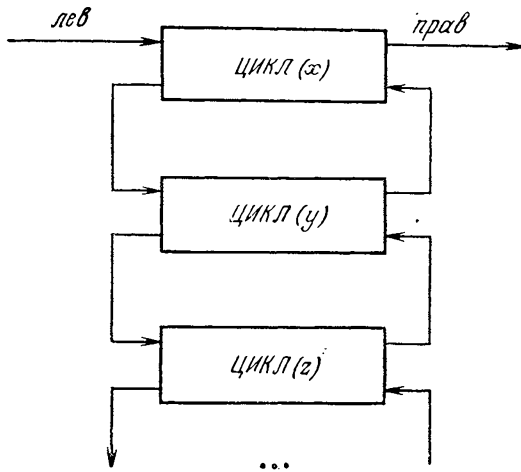


Рис. 4.13

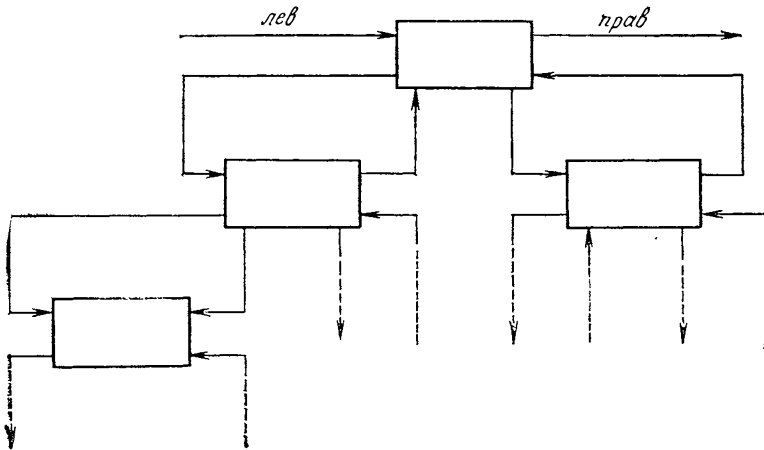


Рис. 4.14

бота сети становится невозможной. Динамические перераспределения и изменения конфигурации в аппаратных сетях значительно сложнее, чем распределение памяти на основе стека, делающее рекурсию в обычных последовательных программах столь эффективной. Но, несмотря на это, рекурсия несомненно оправдывает себя теми возможностями, которые она дает при проектировании и разработке алгоритмов, а если и не этим, то хотя бы тем интеллектуальным наслаждением, которое испытывает человек, понимающий и применяющий ее.

Глава 5. Последовательные процессы

5.1. ВВЕДЕНИЕ

Мы определили *СТОП* как процесс, не выполняющий никаких действий. Его нельзя назвать полезным процессом, и возникает он, скорее, в результате дедлока или других ошибок проектирования, нежели по замыслу разработчика. Существует, однако, и другая причина, по которой процесс может прекратить работу, а именно — если он выполнил все, что ему полагалось. О таком процессе говорят, что он успешно завершился. Чтобы отличать такое завершение от *СТОП*, удобно рассматривать его как специальное событие, обозначаемое символом \surd («успех»). Последовательным называется процесс, имеющий в алфавите символ \surd ; естественно, что это событие может быть только последним в работе процесса. По этой причине мы ставим условием, что \surd не может служить альтернативой в конструкции выбора:

$(x : B \rightarrow P(x))$ неверно, если $\surd \in B$

Определим *ПРОПУСК_A* как процесс, который ничего не делает, но благополучно завершается:

$$\alpha\text{ПРОПУСК}_A = A \cup \{\surd\}$$

Как обычно, мы часто будем опускать индекс A .

Примеры

X1. Торговый автомат, который обслуживает одного покупателя шоколадкой или ириской, на чем успешно завершает работу:

$$\text{ТАОДИН} = (\text{мон} \rightarrow (\text{шок} \rightarrow \text{ПРОПУСК} \mid \text{ирис} \rightarrow \text{ПРОПУСК}))$$

При проектировании процесса для решения некоторой сложной задачи часто бывает полезно разбить ее на две подзадачи, одна из которых успешно завершается до начала другой. Если P и Q — последовательные процессы с одним и тем же алфавитом, их последовательная композиция $P; Q$ представляет собой процесс, ведущий себя сначала как P , а после успешного завершения P продолжающий вести себя

как Q . Если успешного завершения P не происходит, то не завершается и $(P; Q)$.

Х2. Торговый автомат, предназначенный для обслуживания по очереди ровно двух покупателей:

$$ТАДВА = ТАОДИН; ТАОДИН$$

Процесс, с требуемой частотой повторяющий одни и те же действия, известен как цикл; его можно определить как особый случай рекурсии:

$$\begin{aligned} *P &= \mu X.(P; X) \\ &= P; P; P; \dots \\ \alpha(*P) &= \alpha P - \{\sqrt{}\} \end{aligned}$$

Очевидно, что такой цикл никогда успешно не завершится, и поэтому имеет смысл убрать из его алфавита символ $\sqrt{}$.

Х3. Торговый автомат, обслуживающий любое число покупателей:

$$ТАШИ = *ТАОДИН$$

тождествен $ТАШИ$ из примера 1.1.3 **Х3**.

Последовательность символов называется *предложением* процесса P , если после выполнения соответствующей последовательности действий P успешно завершается. Множество всех таких предложений называется *языком*, допускаемым P . Таким образом, обозначения, введенные для описания последовательных процессов, можно использовать и для определения грамматики простого языка типа тех, что применяются при взаимодействии человека с машиной.

Х4. Предложение языка Пиджингол состоит из группы подлежащего, за которым следует сказуемое. Сказуемое — это глагол, за которым следует группа подлежащего. Глаголом может быть либо *bites*, либо *scratches*. Строгое определение группы подлежащего дается ниже:

$$\begin{aligned} \alpha \text{ПИДЖИНГОЛ} &= \{a, the, cat, dog, bites, scratches\} \\ \text{ПИДЖИНГОЛ} &= \text{ГРУППАПОД}; \text{СКАЗУЕМОЕ} \\ \text{СКАЗУЕМОЕ} &= \text{ГЛАГОЛ}; \text{ГРУППАПОД} \\ \text{ГЛАГОЛ} &\rightarrow (bites \rightarrow \text{ПРОПУСК} \mid scratches \rightarrow \text{ПРОПУСК}) \\ \text{ГРУППАПОД} &= \text{Артикль}; \text{СУЩЕСТВИТЕЛЬНОЕ} \\ \text{Артикль} &= (a \rightarrow \text{ПРОПУСК} \mid the \rightarrow \text{ПРОПУСК}) \\ \text{СУЩЕСТВИТЕЛЬНОЕ} &= (cat \rightarrow \text{ПРОПУСК} \\ &\quad \mid dog \rightarrow \text{ПРОПУСК}) \end{aligned}$$

Примеры предложений на Пиджинголе:

the cat scratches a dog
a dog bites the cat

Для описания языков с неограниченным количеством предложений необходимо воспользоваться тем или иным видом итерации или рекурсии.

X5. Группа подлежащего, которая может содержать любое число прилагательных *furry* и *prize*:

ГРУППАПОД = АРТИКЛЬ; $\mu X. (furry \rightarrow X \mid prize \rightarrow X$
 $\mid cat \rightarrow \text{ПРОПУСК}$
 $\mid dog \rightarrow \text{ПРОПУСК})$

Примеры группы подлежащего:

the furry furry prize dog
a dog

X6. Процесс, который допускает цепочки, состоящие из любого числа символов «a», следующего за ними символа «b», а затем — того же числа символов «c», после чего процесс успешно завершается.

$A^n BC^n = \mu X. (b \rightarrow \text{ПРОПУСК}$
 $\mid a \rightarrow (X; (c \rightarrow \text{ПРОПУСК})))$

Если первым допускается символ «b», процесс завершается; в этом случае цепочка не содержит ни «a», ни «c»; значит, их количество совпадает. Если выбрана вторая альтернатива, допускаемые предложения начинаются с «a» и заканчиваются «c», а между ними заключено предложение, допускаемое рекурсивным вызовом процесса X. Если мы предположим, что рекурсивный вызов допускает равное число символов «a» и «c», то это же будет справедливо и для нерекурсивного вызова процесса $A^n BC^n$, поскольку он допускает ровно на один символ «a» больше в начале и на один символ «c» — в конце.

Этот пример показывает, как использование последовательной композиции в совокупности с рекурсией позволяет определить машину с бесконечным числом состояний.

X7. Процесс, ведущий себя как $A^n BC^n$, а затем допускающий символ «d», за которым следует то же число символов «e»:

$A^n BC^n DE^n = ((A^n BC^n); d \rightarrow \text{ПРОПУСК} \parallel C^n DE^n,$

где $C^n DE^n = f(A^n BC^n)$ для f , отображающей a в c , b в d , а c в e .

В этом примере левый операнд \parallel отвечает за соблюдение равенства числа символов «a» и «c» (разделенных b). Он не

допускает символа «*d*» до тех пор, пока не наберется нужного числа «*c*»; символы «*e*» (не принадлежащие его алфавиту) процесс просто игнорирует. Правый операнд \parallel отвечает за соблюдение равенства числа «*e*» и «*c*». Он игнорирует символы «*a*» и «*b*», не принадлежащие его алфавиту. Пара этих процессов завершается одновременно, выполнив порученные им задания.

Система обозначений, задающая язык посредством допускающего процесса, не менее мощна, чем регулярные выражения. Использование рекурсии делает ее мощность близкой к контекстно-свободным грамматикам, но все же не до конца. Процесс может задавать лишь языки, поддающиеся разбору слева направо без возвратов и предварительных просмотров. Это происходит потому, что при использовании оператора выбора требуется, чтобы первое событие каждой альтернативы отличалось от всех остальных возможных первых событий. Следовательно, определяя, например, группу подлежащего, конструкцию из примера X5 нельзя использовать так, чтобы слово *prize* могло выполнять в нем роль как существительного, так и прилагательного, скажем *the prize dog, the furry prize*. Использование операции Π (разд. 3.3) тоже не поможет, потому что в этом случае возникает недетерминизм, и выбор члена, анализирующего остаток ввода, может осуществляться произвольно. Если выбор будет сделан неверно, процесс придет в состояние дедлока, не достигнув конца входного текста. Решить эту проблему можно с помощью новой разновидности оператора выбора, который бы обеспечил ангельский недетерминизм типа *или3* (разд. 3.2.2). Такой оператор требует, чтобы обе альтернативы исполнялись параллельно до тех пор, пока обстановка не сделает выбор; его определение мы оставляем в качестве упражнения.

Без ангельского недетерминизма описанный метод задания языков не так мощен, как контекстно-свободные грамматики, поскольку он требует анализируемости слева направо без возвратов. Однако введение операции \parallel позволяет задавать языки, не являющиеся контекстно-свободными, как, например, X7.

X8. Процесс, допускающий любое чередование событий *вверх* и *вниз* с тем ограничением, что он успешно завершается, как только число событий *вниз* превзойдет число событий *вверх*:

$$ПОЗ = (вниз \rightarrow ПРОПУСК \mid вверх \rightarrow (ПОЗ; ПОЗ))$$

Если первый символ — *вниз*, то задание *ПОЗ* немедленно оказывается выполненным. Но если первый символ — *вверх*,

то тогда требуется допустить на два символа *вниз* больше, чем *вверх*. Достигнуть этого можно, только если сначала допустить на один символ *вниз* больше, чем *вверх*, а затем снова допустить на один символ *вниз* больше, чем *вверх*. Таким образом, потребуются два рекурсивных вызова ПОЗ, следующих друг за другом.

Х9. Процесс C_0 ведет себя как $CT_0(1.1.4 \text{ Х2})$:

$$\begin{aligned} C_0 &= (\text{вокруг} \rightarrow C_0 \mid \text{вверх} \rightarrow C_1) \\ C_{n+1} &= \text{ПОЗ}; C_n \qquad \text{для всех } n \geq 0. \\ &= \underbrace{\text{ПОЗ}; \dots; \text{ПОЗ}; \text{ПОЗ}; C_0}_{n \text{ раз}} \end{aligned}$$

Мы получили возможность решить проблему, упомянутую в примере 2.6.2 **Х3** и вновь возникшую в 4.5 **Х3**, а именно, что в каждой операции над подчиненным процессом явно упоминается остаток следующего за ней главного процесса. Теперь того же эффекта гораздо проще достигнуть средствами последовательной композиции и процесса ПРОПУСК.

Х10. Процесс ПОЛЬЗ оперирует двумя переменными-счетчиками l и m (см. 2.6.2 **Х3**):

$$l : CT_0 \parallel m : CT_3 \parallel \text{ПОЛЬЗ}$$

Следующий подпроцесс (внутри ПОЛЬЗ) зобавляет текущее значение l к m :

$$\begin{aligned} \text{СЛОЖ} &= (l.\text{вокруг} \rightarrow \text{ПРОПУСК} \\ &\quad \mid l.\text{вниз} \rightarrow (\text{СЛОЖ}; (m.\text{вверх} \rightarrow l.\text{вверх} \rightarrow \\ &\qquad \qquad \qquad \rightarrow \text{ПРОПУСК}))) \end{aligned}$$

Если первоначально значение l равно нулю, делать ничего не надо. В противном случае l уменьшается на единицу, а полученное значение добавляется к m (рекурсивным вызовом СЛОЖ). Затем m и l увеличиваются на единицу, чтобы компенсировать вычитание единицы из l и вернуть l его начальное значение.

5.2. ЗАКОНЫ

Законы для последовательной композиции аналогичны законам для конкатенации протоколов (разд. 1.6.1), а ПРОПУСК играет роль единицы:

$$\text{Л1. } \text{ПРОПУСК}; P = P; \text{ПРОПУСК} = P$$

$$\text{Л2. } (P; Q); R = P; (Q; R)$$

$$\text{Л3. } (x : B \rightarrow P(x)); Q = (x : B \rightarrow (P(x); Q))$$

Закон для оператора выбора имеет следствия:

$$\text{L4. } (a \rightarrow P); Q = a \rightarrow (P; Q)$$

$$\text{L5. } \text{СТОП}; Q = \text{СТОП}$$

Когда последовательные процессы объединяются параллельно, их комбинация успешно завершается лишь при успешном завершении *обеих* компонент:

$$\text{L6. } \text{ПРОПУСК}_A \parallel \text{ПРОПУСК}_B = \text{ПРОПУСК}_{A \cup B}$$

Успешно завершившийся процесс не участвует ни в каких дальнейших событиях, предлагаемых его параллельным партнером:

$$\begin{aligned} \text{L7. } ((x : B \rightarrow P(x)) \parallel \text{ПРОПУСК}_A) = \\ = (x : (B - A) \rightarrow (P(x) \parallel \text{ПРОПУСК}_A)) \end{aligned}$$

Когда же успешно завершается параллельная комбинация последовательного и непоследовательного процессов? Если алфавит последовательного процесса полностью содержит алфавит своего партнера, завершение их композиции определяется последовательным процессом, поскольку после его завершения другой процесс ничего не может делать:

$$\text{L8. } \text{СТОП}_A \parallel \text{ПРОПУСК}_B = \text{ПРОПУСК}_B \quad \text{если } \surd \cong A \ \& \ A \subseteq B,$$

Условие корректности этого закона несет в себе глубокий смысл, который всегда следует иметь в виду, когда событие \surd принадлежит алфавиту лишь одного из параллельно работающих процессов. Таким путем мы избегаем проблемы процесса, продолжающегося после события \surd .

Законы **L1—L3** можно использовать при доказательстве сделанного в **5.1 X9** утверждения, что C_0 ведет себя как ST_0 (**1.1.4 X2**). Сделать это можно, показав, что C удовлетворяет системе предваренных рекурсивных уравнений, использованных для определения ST . Уравнения для ST_0 в точности совпадают с уравнением для C_0 :

$$C_0 = (\text{вокруг} \rightarrow C_0 \mid \text{вверх} \rightarrow C_1) \quad \text{по определению } C_0$$

Для $n > 0$ надо доказать, что

$$C_n = (\text{вверх} \rightarrow C_{n+1} \mid \text{вниз} \rightarrow C_{n-1})$$

Доказательство:

$$\begin{aligned}
 ЛЧ &= ПОЗ; C_{n-1} && \text{по определению } C_n \\
 &= (вниз \rightarrow ПРОПУСК \mid вверх \rightarrow ПОЗ; ПОЗ); C_{n-1} \\
 &&& \text{по определению } ПОЗ \\
 &= (вниз \rightarrow (ПРОПУСК; C_{n-1}) \mid вверх \rightarrow (ПОЗ; ПОЗ); C_{n-1}) \\
 &= (вниз \rightarrow C_{n-1} \mid вверх \rightarrow ПОЗ; (ПОЗ; C_{n-1})) && L1, L2 \\
 &= (вниз \rightarrow C_{n-1} \mid вверх \rightarrow ПОЗ; C_n) && \text{по определению } C_n \\
 &= ПРЧ && \text{по определению } C_n
 \end{aligned}$$

Поскольку C_n удовлетворяет той же системе предваренных рекурсивных уравнений, что и ST_n , эти процессы совпадают.

Мы полностью привели текст этого доказательства, чтобы проиллюстрировать использование законов и, кроме того, рассеять подозрения о наличии в доказательстве порочного круга. Подозрительнее всего то, что в доказательстве не используется индукция по n . На самом же деле любая попытка применить индукцию по n потерпит неудачу, поскольку само определение ST_n включает процесс ST_{n+1} . К счастью, здесь и с легкостью и с пользой можно применить закон о единственном решении.

5.3. МАТЕМАТИЧЕСКАЯ ТРАКТОВКА

Математическое определение последовательной композиции необходимо сформулировать таким образом, чтобы обеспечить истинность законов, введенных в предыдущем разделе. Особого внимания требует равенство $P; ПРОПУСК = P$. Как обычно, трактовка детерминированных процессов значительно проще; с нее мы и начнем.

5.3.1. Детерминированные процессы

Операции над детерминированными процессами определяются в терминах протоколов их результатов. Первым и единственным действием процесса $ПРОПУСК$ является успешное завершение, и поэтому он имеет только два протокола:

$$L0 \text{ протоколы}(ПРОПУСК) = \{ \langle \rangle, \langle \surd \rangle \}$$

При определении последовательной композиции процессов удобно сначала определить последовательную композицию их отдельных протоколов. Если s и t — протоколы и s не содержит символа \surd , то

$$\begin{aligned}
 (s; t) &= s \\
 (s \frown \langle \surd \rangle); t &= s \frown t
 \end{aligned}$$

(подробности см. в разд. 1.9.7). Протокол $(P;Q)$ состоит из протокола P , и если этот протокол оканчивается символом \surd , то \surd заменяется на протокол Q :

L1. $\text{протоколы}(P;Q) = \{s; t \mid s \in \text{протоколы}(P) \text{ \& } t \in \text{протоколы}(Q)\}$

Эквивалентное определение:

L1A. $\text{протоколы}(P;Q) = \{s \mid s \in \text{протоколы}(P) \text{ \& } \neg \langle \surd \rangle \text{ в } s\} \cup \{s \hat{\sim} t \mid s \hat{\sim} \langle \surd \rangle \in \text{протоколы}(P) \text{ \& } t \in \text{протоколы}(Q)\}$

Это определение может быть понятнее, но его сложнее использовать.

Назначение события \surd целиком состоит в том, чтобы завершать участвующий в нем процесс. Поэтому нам требуется закон

L2. $P/s = \text{ПРОПУСК}$ если $s \hat{\sim} \langle \surd \rangle \in \text{протоколы}(P)$

Этот закон существенно используется при доказательстве того, что $P/\text{ПРОПУСК} = P$. К сожалению, это не всегда верно. Если, например,

$$P = (\text{ПРОПУСК}_{\{c\}} \parallel c \rightarrow \text{СТОП}_{\{c\}})$$

то $\text{протоколы}(P) = \{\langle \rangle, \langle \surd \rangle, \langle c \rangle, \langle c, \surd \rangle, \langle \surd, c \rangle\}$ и $P/\langle \rangle \neq \text{ПРОПУСК}$, даже несмотря на то, что $\langle \surd \rangle \in \text{протоколы}(P)$. Поэтому нам потребуется ввести алфавитные ограничения на операцию параллельной композиции. Конструкцию $(P \parallel Q)$ следует считать неверной, если не выполняется соотношение

$$\alpha P \subseteq \alpha Q \vee \alpha Q \subseteq \alpha P \vee \surd \in (\alpha P \cap \alpha Q \cup \overline{\alpha P} \cap \overline{\alpha Q})$$

По сходным причинам требуется, чтобы переименование не затрагивало символа \surd , т. е. $f(P)$ неверно, если не выполняется условие $f(\surd) = \surd$. Более того, мы должны условиться, что если m — имя процесса, то $m.\surd = \surd$. И наконец, нельзя использовать символ \surd в конструкции выбора $(\surd \rightarrow P \mid c \rightarrow Q)$. Это ограничение также исключает ИСП_A , если $\surd \in A$.

5.3.2. Недетерминированные процессы

При описании последовательной композиции недетерминированных процессов возникает ряд проблем. Первая из них состоит в том, что недетерминированный процесс типа $\text{ПРОПУСК} \sqcap (c \rightarrow \text{ПРОПУСК})$ не удовлетворяет закону **L2** из предыдущего раздела. Решить эту проблему можно, ослабив 5.3.1 **L2** до

L2A. $s \hat{\sim} \langle \surd \rangle \in \text{протоколы}(P) \Rightarrow (P/s) \subseteq \text{ПРОПУСК}$

Это значит, что всякий раз, когда P может завершиться, он не предлагает при этом своему окружению никакого альтернативного события. Для поддержания истинности **L2A** необходимо соблюдение всех ограничений из предыдущего раздела, а кроме того: **ПРОПУСК** не должен использоваться как непредваренный операнд Π , символ \surd не должен содержаться в алфавите обоих операндов III . (Возможно, что небольшие изменения в определении операций Π и III позволяют ослабить эти ограничения.)

В дополнение к законам, приведенным ранее в этой главе, последовательная композиция недетерминированных процессов удовлетворяет следующим законам. Во-первых, расходящийся процесс остается расходящимся независимо от того, что, согласно спецификации, происходит после его успешного завершения:

L1. $\text{XАОС}; P = \text{XАОС}$

Последовательная композиция дистрибутивна относительно недетерминированного выбора:

L2A. $(P \sqcap Q); R = (P; R) \sqcap (Q; R)$

L2B. $R; (P \sqcap Q) = (R; P) \sqcap (R; Q)$

Для определения $(P; Q)$ в математической модели недетерминированных процессов (разд. 3.9) необходимо рассмотреть его неудачи и расходимость. Но сначала мы опишем его отказы (разд. 3.4). Если P может отказаться от X и не может успешно завершиться, то это значит, что $X \cup \{\surd\}$ также является отказом P (3.4.1 L11). В этом случае X является отказом процесса $(P; Q)$. Но если в поведении P присутствует вариант успешного завершения, то тогда в $(P; Q)$ этот переход может произойти автономно, т. е. его наступление будет скрытым, и каждый отказ процесса Q будет также отказом процесса $(P; Q)$. В определении также рассмотрен случай, когда успешное завершение P недетерминировано:

D1. $\text{отказы}(P; Q) = \{X \mid (X \cup \{\surd\}) \in \text{отказы}(P)\} \cup \{X \mid \surd \in \text{протоколы}(P) \ \& \ X \in \text{отказы}(Q)\}$

Протоколы $(P; Q)$ определяются так же, как и для детерминированных процессов. Расходимость процесса $(P; Q)$ можно определить, если заметить, что он расходится всякий раз, когда расходится P или когда P успешно завершился, а Q расходится.

D2. $\text{расходимость}(P; Q) = \{s \mid s \in \text{расходимость}(P) \ \& \ \surd \in s\} \cup \{s \hat{=} t \mid s \hat{=} \surd \in \text{протоколы}(P) \ \& \ \surd \in s \ \& \ t \in \text{расходимость}(Q)\}$

Любая неудача процесса $(P; Q)$ — это либо неудача процесса P , прежде чем P успел завершиться, либо неудача процесса Q после успешного завершения P :

$$\begin{aligned} \text{Д3. } \text{неудачи}(P; Q) = & \{(s, X) \mid (s, X \cup \{\checkmark\}) \in \text{неудачи}(P)\} \\ & \cup \{(s \hat{\sim} t, X) \mid s \hat{\sim} \langle \checkmark \rangle \in \text{протоколы}(P) \text{ \& } (t, X) \\ & \qquad \qquad \qquad \in \text{неудачи}(Q)\} \\ & \cup \{(s, X) \mid s \in \text{расходимости}(P; Q)\} \end{aligned}$$

5.3.3. Реализация

ПРОПУСК реализуется как процесс, допускающий только один символ "УСПЕХ. Что он делает после — не имеет значения.

$$\text{ПРОПУСК} = \lambda x. \text{ if } x = \text{"УСПЕХ"} \text{ then } \text{СТОП} \text{ else "BLEEP"}$$

Если первый операнд завершается, последовательная композиция ведет себя как второй операнд; в противном случае первый операнд участвует в первом событии, а его оставшаяся часть последовательно объединяется со вторым операндом:

$$\begin{aligned} \text{послед}(P, Q) = & \text{if } P(\text{"УСПЕХ"}) \neq \text{"BLEEP"} \text{ then } Q \\ & \text{else } \lambda x. \text{ if } P(x) = \text{"BLEEP"} \text{ then "BLEEP"} \\ & \qquad \qquad \qquad \text{else } \text{послед}(P(x), Q) \end{aligned}$$

5.4. ПРЕРЫВАНИЯ

В данном разделе мы определим разновидность последовательной композиции $(P \hat{\sim} Q)$, не зависящей от успешного завершения P . Вместо этого при наступлении первого события в Q исполнение P просто прерывается; P затем уже не возобновляется. Из этого определения следует, что протоколом процесса $(P \hat{\sim} Q)$ будет протокол процесса P до некоторого произвольного места, где произошло прерывание, и следующий за ним протокол процесса Q :

$$\alpha(P \hat{\sim} Q) = \alpha P \cup \alpha Q$$

$$\begin{aligned} \text{протоколы}(P \hat{\sim} Q) = & \{s \hat{\sim} t \mid s \in \text{протоколы}(P) \text{ \& } t \in \\ & \qquad \qquad \qquad \text{протоколы}(Q)\} \end{aligned}$$

Во избежание проблем потребуем, чтобы символ \checkmark не принадлежал αP .

Следующий закон гласит, что момент наступления Q определяется обстановкой, выбирающей начальное событие, возможное для Q и невозможное для P :

$$\text{Л1. } (x : B \rightarrow P(x)) \hat{\sim} Q = Q \amalg (x : B \rightarrow P(x) \hat{\sim} Q)$$

Если $(P \hat{Q})$ может быть прерван процессом R , то это то же самое, как если бы P прерывался процессом $(Q \hat{R})$:

$$\text{L2. } (P \hat{Q}) \hat{R} = P \hat{(Q \hat{R})}$$

Поскольку *СТОП* не имеет возможных начальных событий, он не может быть запущен обстановкой. Аналогично если *СТОП* прерывается, то лишь прерывание и может в действительности произойти. Таким образом, *СТОП* служит для операции $\hat{\quad}$ единицей:

$$\text{L3. } P \hat{\text{СТОП}} = P = \text{СТОП} \hat{P}$$

В операции прерывания оба ее операнда исполняются не более чем по одному разу, и, значит, прерывание дистрибутивно относительно недетерминированного выбора:

$$\text{L4A. } P \hat{(Q \sqcap R)} = (P \hat{Q}) \sqcap (P \hat{R})$$

$$\text{L4B. } (Q \sqcap R) \hat{P} = (Q \hat{P}) \sqcap (R \hat{P})$$

И наконец, прерывание не спасает расходящийся процесс; точно так же небезопасно описывать расходящийся процесс после прерывания:

$$\text{L5. } \text{ХАОС} \hat{P} = \text{ХАОС} = P \hat{\text{ХАОС}}$$

В оставшейся части этого раздела мы потребуем, чтобы возможные начальные события прерывающего процесса не входили в алфавит прерываемого. Поскольку наступление прерывания обозримо и контролируемо обстановкой, это ограничение сохраняет детерминизм и упрощает рассуждения об операторах. Чтобы подчеркнуть сохранение детерминированности, мы расширим определение оператора выбора. При условии, что $c \in B$, $(x : B \rightarrow P(x) \mid c \rightarrow Q)$ будет сокращенной записью конструкции $(x : (B \cup \{c\}) \rightarrow (\text{if } x = c \text{ then } Q \text{ else } P(x)))$ и аналогично для большего числа операндов.

5.4.1. Катастрофы

Пусть символ \mathfrak{c} означает катастрофическое прерывание процесса P , вызванное, как естественно предполагать, не самим P ; более строго:

$$\mathfrak{c} \hat{=} \alpha P$$

Тогда процесс, ведущий себя до катастрофы как P , а после нее — как Q , определим следующим образом:

$$P \hat{\mathfrak{c}} Q = P \hat{(\mathfrak{c} \rightarrow Q)}$$

Здесь Q может играть роль процесса, ликвидирующего последствия катастрофы. Заметим, что символ инфиксного оператора $\hat{!}$ отличается от символа события $!$ значком $\hat{}$.

Первый закон представляет собой всего лишь очевидную формализацию нестрогого описания оператора:

$$L1. (P \hat{!} Q) / (s \hat{!} \langle ! \rangle) = Q \quad \text{для } s \in \text{протоколы}(P)$$

В детерминированной модели одного этого закона достаточно для однозначного определения оператора $\hat{!}$. В области недетерминированных процессов для однозначности потребуются дополнительные законы, касающиеся строгости и дистрибутивности по обоим аргументам.

Второй закон дает более явное представление о первом и последующих шагах процесса. Он показывает, как осуществляется «обратная» дистрибутивность операции $\hat{!}$ относительно \rightarrow :

$$L2. (x : B \rightarrow P(x)) \hat{!} Q = (x : B \rightarrow (P(x) \hat{!} Q \mid ! \rightarrow Q))$$

Этот закон также однозначно задает оператор на множестве детерминированных процессов.

5.4.2. Перезапуск

Одной из возможных реакций на катастрофу является перезапуск исходного процесса. Пусть процесс P таков, что $! \in \alpha P$. Определим \hat{P} как процесс, ведущий себя как P до наступления $!$, а после каждого события $!$ ведущий себя снова как P , но с самого начала. Такой процесс называется *перезапускаемым* и определяется простой рекурсией:

$$\begin{aligned} \alpha \hat{P} &= \alpha P \cup \{!\} \\ \hat{P} &= \mu X. (P \hat{!} X) \\ &= P \hat{!} (P \hat{!} (P \hat{!} \dots)) \end{aligned}$$

Эта рекурсия предварена, поскольку вхождением X предшествуют события $!$. Разумеется, что \hat{P} — циклический процесс (разд. 1.8.3), даже если P таковым не является.

Катастрофа — это не единственный повод для перезапуска. Рассмотрим игровой процесс, взаимодействие с которым осуществляется его партнером, человеком, путем выбора клавиш на клавиатуре (см. описание функции *взаимодействия* из разд. 1.4). Бывает, что человек недоволен ходом игры и хочет начать игру заново. Для этой цели на клавиатуре может быть специальная клавиша ($!$), нажатие которой на любом этапе игры приводит к ее повторному запуску. Игру P удобно описывать независимо от возможности пере-

запуска, а затем, используя описанный выше оператор, преобразовать ее в повторно запускаемую игру \hat{P} . Эта идея принадлежит Алексу Тэруелу.

Неформальное определение \hat{P} дается законом

$$\mathbf{L1.} \quad \hat{P}/s \hat{\langle \cdot \rangle} = \hat{P} \quad \text{для } s \in \text{протоколы}(P)$$

Правда, этот закон неоднозначно определяет \hat{P} хотя бы потому, что ему удовлетворяет ИСП. Однако \hat{P} является наименьшим детерминированным процессом, удовлетворяющим **L1**.

5.4.3. Чередование

Предположим, что P и Q — игровые процессы наподобие описанных в разд. 5.4.2 и человек хочет играть в обе игры одновременно, делая ходы поочередно, как это делает шахматный гроссмейстер, ведущий сеанс одновременной игры сразу с несколькими более слабыми противниками. Для этого введем новую клавишу \otimes , вызывающую смену игры. Это чем-то напоминает прерывание, поскольку текущая игра прерывается в произвольном месте; но в отличие от прерывания в этом случае текущее состояние прерываемой игры сохраняется, и позже, когда будет прервана следующая игра, игра возобновляется именно с этого места. Играющий таким образом в P и Q процесс обозначается $(P \otimes Q)$ и с предельной ясностью определяется законами

$$\mathbf{L1.} \quad \otimes \in (\alpha(P \otimes Q) - \alpha P - \alpha Q)$$

$$\mathbf{L2.} \quad (P \otimes Q)/s = (P/s) \otimes Q \quad \text{если } s \in \text{протоколы}(P)$$

$$\mathbf{L3.} \quad (P \otimes Q)/\langle \otimes \rangle = (Q \otimes P)$$

Нас интересует наименьший процесс, удовлетворяющий **L2** и **L3**. Из этих законов можно вывести более конструктивное определение операции \otimes ; в нем показано, каким образом осуществляется обратная дистрибутивность \otimes относительно \rightarrow :

$$\mathbf{L4.} \quad (x : B \rightarrow P(x)) \otimes Q = (x : B \rightarrow (P(x) \otimes Q) \mid \otimes \rightarrow (Q \otimes P))$$

Операция чередования полезна не только в игровых процессах. Аналогичным средством чередования системных утилит должна обладать ориентированная на пользователя операционная система, если, к примеру, вы не хотите терять место в редакторе, обращаясь к программе «help», и наоборот.

5.4.4. Контрольные точки

Пусть P — процесс, описывающий поведение системы управления долговечной базой данных. Одной из наихудших реакций на удар молнии (\imath) будет перезапуск P с самого начала, при котором теряются все накопленные системой данные. Намного лучше было бы вернуться к некоторому недавнему состоянию системы, о котором известно, что оно является удовлетворительным. Такое состояние называется контрольной точкой. Введем для этого новую клавишу \odot , которую следует нажимать только в том случае, когда известно, что текущее состояние системы является удовлетворительным. Когда наступает событие \imath , восстанавливается состояние в последней контрольной точке; в случае отсутствия контрольных точек восстанавливается начальное состояние. Согласно нашему предположению, \odot и \imath не принадлежат алфавиту P , и поэтому мы определим $Ch(P)$ как процесс, ведущий себя как P , но должным образом реагирующий на эти два события.

Наиболее сжато нестрогое определение $Ch(P)$ формализуют следующие законы:

$$\mathbf{L1.} \quad Ch(P)/(s^{\wedge}(\imath)) = Ch(P) \quad \text{для } s \in \text{протоколы}(P)$$

$$\mathbf{L2.} \quad Ch(P)/s^{\wedge}(\odot) = Ch(P/s) \quad \text{для } s \in \text{протоколы}(P)$$

В более явном виде $Ch(P)$ можно определить, используя двуместный оператор $Ch2(P, Q)$, где P — текущий процесс, а Q — самая последняя контрольная точка, ожидающая восстановления. Если катастрофа наступает еще до первой контрольной точки, то система перезапускается:

$$\mathbf{L3.} \quad Ch(P) = Ch2(P, P)$$

$$\mathbf{L4.} \quad \text{Если } P = (x : B \rightarrow P(x)), \text{ то } Ch2(P, Q) = \\ = (x : B \rightarrow Ch2(P(x), Q) \mid \imath \rightarrow Ch2(Q, Q) \mid \odot \rightarrow Ch2(P, P)).$$

Закон **L4** наводит на мысль о практическом способе реализации, при котором контрольное состояние системы хранится на некотором дешевом, но надежном носителе — таком, как диск или лента. При наступлении события \odot текущее состояние копируется как новая контрольная точка; если же наступает событие \imath , контрольная точка рекопируется как новое текущее состояние. Из соображений экономии системный разработчик стремится, чтобы текущее и контрольное состояния совместно использовали как можно больше данных. Эта оптимизация является весьма машинно- и системно-зависимой, и поэтому особенно приятна простота ее математической основы,

Оператор копирования в контрольных точках полезен не только в больших системах баз данных. Играя в сложную игру, человек может захотеть испробовать различные возможные линии поведения, не останавливаясь пока окончательно ни на одной из них. Для этого он нажимает клавишу ©, сохраняя текущее состояние, и если его дальнейшие действия безуспешны, нажатие клавиши (ι) восстанавливает прежнее положение.

Идеи контрольных точек исследовались Йеном Хейесом.

5.4.5. Множественные контрольные точки

При использовании системы с контрольными точками может случиться, что контрольная точка была объявлена ошибочно. В таких случаях желательно отменить последнюю контрольную точку и вернуться к предыдущей. Для этого система должна хранить два или более последних контрольных состояния. В принципе нам ничто не мешает определить систему $Mch(P)$, хранящую все контрольные точки вплоть до начальной. Каждое событие ι приводит систему к состоянию, *предшествующему* последнему событию ©, а не следующему за ним. Как всегда, мы требуем, чтобы

$$\alpha Mch(P) - \alpha P = \{\odot, \iota\}$$

Событие ι, наступившее до ©, возвращает систему к начальному состоянию:

$$L1. Mch(P)/s \hat{\sim} \langle \iota \rangle = Mch(P) \quad \text{для } s \in \text{протоколы}(P)$$

Событие ι, произошедшее после ©, аннулирует результат всего, что происходило после и включая последнее событие ©:

$$L2. Mch(P)/s \hat{\sim} \langle \odot \rangle \hat{\sim} t \hat{\sim} \langle \iota \rangle = Mch(P/s) \quad \text{для } (s \upharpoonright (\alpha P - \{\odot\})) \hat{\sim} t \in \text{протоколы}(P)$$

Более явное определение $Mch(P)$ можно дать в терминах двуместного оператора $Mch(P, Q)$, где P — текущий процесс, а Q — стек контрольных точек, ожидающих восстановления в случае необходимости. Начальным содержимым стека служит бесконечная последовательность копий P :

$$\begin{aligned} L3. Mch(P) &= \mu X. Mch2(P, X) \\ &= Mch2(P, Mch(P)) \\ &= Mch2(P, Mch2(P, Mch2(P, \dots))) \end{aligned}$$

При наступлении \odot в верхушку стека проталкивается текущее состояние, а при наступлении \lrcorner весь стек восстанавливается:

L4. Если $P = (x : B \rightarrow P(x))$, то

$$\begin{aligned} Mch2(P, Q) = & (x : B \rightarrow Mch2(P(x), Q)) \\ & | \odot \rightarrow Mch2(P, Mch2(P, Q)) \\ & | \lrcorner \rightarrow Q \end{aligned}$$

Структура возникающей в **L4** рекурсии достаточно замысловата, но и сам инструмент множественных контрольных точек оказывается при практической реализации весьма дорогостоящим, особенно когда число контрольных точек становится большим.

5.4.6. Реализация

Реализация различных вариантов прерывания основана на законах, описывающих дистрибутивность этого оператора относительно \rightarrow . Рассмотрим для примера операцию чередования (5.4.3 **L4**):

$$\begin{aligned} черед(P, Q) = & \\ & \lambda x. \text{ if } x = \otimes \text{ then } черед(Q, P) \\ & \text{ else if } P(x) = \text{"BLEEP"} \text{ then "BLEEP"} \\ & \text{ else } черед(P(x), Q) \end{aligned}$$

Гораздо более удивительна реализация Mch (5.4.5 **L3, L4**):

$$Mch(P) = Mch2(P, Mch(P))$$

где $Mch2(P, Q) =$

$$\begin{aligned} & \lambda x. \text{ if } x = \lrcorner \text{ then } Q \\ & \text{ else if } x = \odot \text{ then } Mch2(P, Mch2(P, Q)) \\ & \text{ else if } P(x) = \text{"BLEEP"} \text{ then "BLEEP"} \\ & \text{ else } Mch2(P(x), Q) \end{aligned}$$

При исполнении этой функции количество используемой памяти растет пропорционально числу контрольных точек, и доступная память очень быстро исчерпывается. Конечно, при каждом наступлении \lrcorner можно производить чистку памяти, но эффект этого все равно будет очень невелик. Как и в остальных случаях рекурсии, соображения, связанные с практической применимостью, приводят к необходимости ограничения глубины вложенности. В нашем случае разработчику следует наложить ограничение на число хранимых контрольных точек и уничтожить наиболее ранние. Но это решение не имеет такого изящного рекурсивного выражения,

5.5. ПРИСВАИВАНИЕ

В этом разделе мы представим наиболее важные аспекты обычного последовательного программирования, а именно: присваивания, условные операторы и циклы. Для упрощения формулировок полезных законов введем некоторые необычные обозначения.

Одной из важнейших особенностей традиционного программирования для вычислительных машин является оператор присваивания. Если x — программная переменная, e — выражение, а P — процесс, то $(x := e; P)$ — это процесс, ведущий себя как P , но только начальное значение x задается равным начальному значению выражения e . Начальные значения всех остальных переменных остаются без изменений. Смысл присваивания как такового можно задать следующим определением:

$$(x := e) = (x := e; \text{ПРОПУСК})$$

Простое присваивание легко обобщить до множественного присваивания. Пусть x представляет собой список различных переменных $x = x_0, x_1, \dots, x_{n-1}$, а e — список выражений $e = e_0, e_1, \dots, e_{n-1}$. Если длина этих двух списков совпадает, то $x := e$ присваивает начальное значение e_i переменной x_i для всех i . Заметим, что все e_i вычисляются прежде, чем будет сделано хоть одно присваивание, и потому если в выражении g встречается переменная y , то результаты фрагментов

$$y := f; z := g$$

и

$$y, z := f, g$$

будут различными.

Пусть выражение b вычисляет истинность логической функции (значение его может быть либо *истина*, либо *ложь*). Если P и Q — процессы, то

$$P \nless b \nless Q \quad (P \text{ if } b \text{ else } Q)$$

— это процесс, ведущий себя как P , если начальное значение b истинно, или как Q , если начальное значение b ложно. Это обозначение необычно, но менее громоздко, чем традиционное

$$\text{if } b \text{ then } P \text{ else } Q$$

По этим же причинам традиционный цикл

$$\text{while } b \text{ do } Q$$

будет записываться как

$$b * Q$$

Рекурсивно это можно определить как

$$D1. \ b * Q = \mu X. ((Q; X) \nleftarrow b \rhd \text{ПРОПУСК})$$

Примеры

X1. Процесс, ведущий себя как CT_n (1.1.4 X2):

$$\begin{aligned} X1 &= \mu X. (\text{вокруг} \rightarrow X \mid \text{вверх} \rightarrow (n := 1; X)) \\ &\nleftarrow n = 0 \rhd \\ &\quad (\text{вверх} \rightarrow (n := n + 1; X) \mid \text{вниз} \rightarrow (n := n - 1; X)) \end{aligned}$$

Текущее значение счетчика поддерживается в переменной n .

X2. Процесс, ведущий себя как CT_0 :

$$n := 0; X1$$

Начальное значение счетчика устанавливается равным нулю.

X3. Процесс, ведущий себя как $ПОЗ$ (5.1 X8):

$$n := 1; (n > 0) * (\text{вверх} \rightarrow n := n + 1 \mid \text{вниз} \rightarrow n := n - 1)$$

Рекурсия здесь заменена на обычный цикл.

X4. Процесс, который делит натуральное число x на положительное число y и присваивает значение частного переменной q , а остатка — переменной r :

$$\text{ЧАСТН} = (q := x \div y; r := x - q \times y)$$

X5. Процесс, решающий ту же задачу, что и **X4**, но вычисляющий результат медленным методом повторного вычитания:

$$\begin{aligned} \text{МЕДЛЧАСТН} &= (q := 0; r := x; \\ &\quad ((r \geq y) * (q := q + 1; r := r - y))) \end{aligned}$$

В предыдущем примере (4.5 X3) мы показали, как можно промоделировать поведение переменной подчиненным процессом, который связан с использующим его процессом посредством своего значения. В этой главе мы сознательно отказались от этой техники, потому что она не обладает рядом желательных для нас математических свойств. Так, например, мы хотим, чтобы

$$(m := 1; m := 1) = (m := 1)$$

но, к сожалению,

$$(m.\text{лев}!1 \rightarrow m.\text{лев}!1 \rightarrow \text{ПРОПУСК}) \neq (m.\text{лев}!1 \rightarrow \text{ПРОПУСК})$$

5.5.1. Законы

В законах для присваивания x и y обозначают списки различных переменных; e , $f(x)$, $f(e)$ обозначают списки выражений, возможно, содержащих вхождения переменных из x или y ; и если $f(x)$ содержит x_i , то $f(e)$ содержит e_i для всех индексов i . Для простоты будем считать, что все выражения в законах имеют результат для любых значений содержащихся в них переменных.

L1. $(x := x) = \text{ПРОПУСК}$

L2. $(x := e; x := f(x)) = (x := f(e))$

L3. Если x, y — список различных переменных, то $(x := e) = (x, y := e, y)$

L4. Если x, y, z той же длины, что и e, f, g соответственно, то $(x, y, z := e, f, g) = (x, z, y := e, g, f)$

Используя эти законы, любую последовательность присваиваний можно преобразовать в единое присваивание списку всех встречающихся переменных.

Если рассматривать $\nless b \nless$ как двуместную инфиксную операцию, оказывается, что она обладает некоторыми знаковыми алгебраическими свойствами:

L5—6. Операция $\nless b \nless$ идемпотентна, ассоциативна и дистрибутивна относительно $\nless c \nless$

L7. $P \nless \text{истина} \nless Q = P$

L8. $P \nless \text{ложь} \nless Q = Q$

L9. $P \nless \neg b \nless Q = Q \nless b \nless P$

L10. $P \nless b \nless (Q \nless b \nless R) = P \nless b \nless R$

L11. $P \nless (a \nless b \nless c) \nless Q = (P \nless a \nless Q) \nless b \nless (P \nless c \nless Q)$

L12. $x := e; (P \nless b(x) \nless Q) = (x := e; P) \nless b(e) \nless (x := e; Q)$

L13. $(P \nless b \nless Q); R = (P; R) \nless b \nless (Q; R)$

Для эффективного использования присваивания в параллельных процессах необходимо наложить ограничение, что ни одна переменная из тех, которым было присвоено значение внутри одного процесса, не может использоваться в другом. Чтобы соблюсти это ограничение, введем в алфавит последовательного процесса две новые категории символов: $пер(P)$ — множество переменных, которым можно присваивать значения внутри P , $дост(P)$ — множество переменных, доступных в выражениях внутри P .

Все переменные, которые можно изменять, являются также и доступными:

$$\text{пер}(P) \subseteq \text{дост}(P) \subseteq \alpha P$$

Определим по аналогии $\text{дост}(e)$ как множество переменных, встречающихся в e . Если P и Q объединяются оператором \parallel , мы требуем, чтобы

$$\text{пер}(P) \cap \text{дост}(Q) = \text{пер}(Q) \cap \text{дост}(P) = \{ \}$$

При выполнении этого условия уже неважно, произошло ли присваивание до разбиения на параллельные подсистемы или же в ходе работы одного из процессов уже после начала их параллельного исполнения.

$$\text{L14. } ((x := e; P) \parallel Q) = (x := e; (P \parallel Q))$$

при условии, что $x \subseteq \text{пер}(P) - \text{дост}(Q)$ и $\text{дост}(e) \cap \text{пер}(Q) = \{ \}$.

Из этого закона непосредственно следует, что

$$(x := e; P) \parallel (y := f; Q) = (x, y := e, f; (P \parallel Q))$$

при условии, что $x \subseteq \text{пер}(P) - \text{дост}(Q) - \text{дост}(f)$, а $y \subseteq \text{пер}(Q) - \text{дост}(P) - \text{дост}(e)$. Отсюда видно, каким образом ограничения на алфавиты позволяют гарантировать, что присваивания внутри одного процесса из параллельной пары не влияют на присваивания внутри другого. При реализации последовательности присваиваний могут либо чередоваться, либо выполняться совместно, что не имеет никакого значения по отношению к наблюдаемым извне действиям системы.

И наконец, параллельная комбинация дистрибутивна относительно условного оператора:

$$\text{L15. } P \parallel (Q \triangleleft b \triangleright R) = (P \parallel Q) \triangleleft b \triangleright (P \parallel R)$$

при условии, что $\text{дост}(b) \cap \text{пер}(P) = \{ \}$.

Этот закон утверждает также, что момент вычисления условия b (до или после распараллеливания) не имеет значения.

Теперь рассмотрим проблему, возникающую, когда выражения не определены для некоторых значений содержащихся в них переменных. Если e — список выражений, определим $\mathcal{D}e$ как логическое выражение, истинное, когда все операнды из e принадлежат области определения своих операторов. Так, например, для натуральной арифметики

$$\begin{aligned} \mathcal{D}(x \div y) &= (y > 0) \\ \mathcal{D}(y + 1, z + y) &= \text{истина} \\ \mathcal{D}(e + f) &= \mathcal{D}e \& \mathcal{D}f \\ \mathcal{D}(r - y) &= y \leq r \end{aligned}$$

Имеет смысл потребовать, чтобы $\mathcal{D}e$ было всегда определено, т. е. $\mathcal{D}(\mathcal{D}e) = \text{истина}$.

Мы сознательно никак не определили результат попытки вычислить неопределенное выражение — в этом случае может произойти все что угодно. С помощью процесса ХАОС это можно выразить в следующих законах:

$$\mathbf{L16'}. (x := e) = (x := e \nleftarrow \mathcal{D}e \nrightarrow \text{ХАОС})$$

$$\mathbf{L17'}. P \nleftarrow b \nrightarrow Q = ((P \nleftarrow b \nrightarrow Q) \nleftarrow \mathcal{D}b \nrightarrow \text{ХАОС})$$

Небольшой модификации требуют и законы **L2**, **L4** и **L12**

$$\mathbf{L2'}. (x := e; x := f(x)) = (x := f(e) \nleftarrow \mathcal{D}e \nrightarrow \text{ХАОС})$$

$$\mathbf{L4'}. (P \nleftarrow b \nrightarrow P) = (P \nleftarrow \mathcal{D}b \nrightarrow \text{ХАОС})$$

5.5.2. Спецификации

Спецификация последовательного процесса описывает не только протоколы происходящих событий, но и отношения между протоколами, а также начальные и конечные значения программных переменных. Для обозначения начального значения программной переменной x мы используем просто само ее имя x . Конечное значение переменной будем обозначать ее именем с верхним индексом \surd — x^\surd . Величина x^\surd недоступна до тех пор, пока процесс не завершится, т. е. до наступления последнего события его протокола \surd . Мы представим это следующим образом: если $np_0 \neq \surd$, то величина x^\surd не специфицирована.

Примеры

X1. Процесс, не совершающий никаких действий, но прибавляющий единицу к величине x и успешно завершающийся, оставив величину y без изменений:

$$np = \langle \rangle \vee (np = \langle \surd \rangle \& x^\surd = x + 1 \& y^\surd = y)$$

X2. Процесс, выполняющий действие с именем, равным начальному значению x , а затем успешно завершающийся, оставляя конечные значения x и y равными их начальным значениям:

$$np = \langle \rangle \vee np = \langle x \rangle \vee (np = \langle x, \surd \rangle \& x^\surd = x \& y^\surd = y)$$

X3. Процесс, хранящий имя своего первого события как конечное значение переменной x :

$$\# np \leq 2 \& (\# np = 2 \Rightarrow (np = \langle x^\surd, \surd \rangle \& y^\surd = y))$$

Корректная работа процесса часто зависит от некоторого предусловия $s(x)$, налагаемого на начальные значения программных переменных x . Это можно выразить, введя $S(x)$ в спецификацию в качестве предыдущего члена (антецедента).

X4. Процесс, который делит неотрицательное число x на положительное число y и присваивает частное переменной q , а остаток — переменной r :

$$\begin{aligned} \text{ДЕЛ} = (y > 0 \Rightarrow np = \langle \rangle \vee (np = \langle \sqrt{\rangle} \& q^{\vee} = (x \div y) \& r^{\vee} = \\ = x - (q^{\vee} \times y) \& y^{\vee} = y \& x^{\vee} = x)) \end{aligned}$$

Без предусловия было бы невозможно добиться соответствия этой спецификации во всей ее полноте.

X5. Приведем некоторые более сложные спецификации, которыми мы воспользуемся позже:

$$\begin{aligned} \text{ДЕЛЦИКЛ} = (np = \langle \rangle \vee (np = \langle \sqrt{\rangle} \& r = (q^{\vee} - q) \times y + \\ + r^{\vee} \& r^{\vee} < y \\ \& x^{\vee} = x \& y^{\vee} = y)) \end{aligned}$$

$$T(n) = r < n \times y$$

Предполагается, что в этих и последующих спецификациях все переменные обозначают натуральные числа, и поэтому, если второй операнд больше первого, то вычитание неопределено.

Теперь сформулируем закон, лежащий в основе доказательства того, что процесс удовлетворяет своей спецификации. Пусть $S(x, np, x^{\vee})$ — некоторая спецификация. Чтобы доказать, что **ПРОПУСК** удовлетворяет этой спецификации, надо, очевидно, показать, что она истинна, когда протокол пуст; более того, она должна быть истинна, когда протокол равен $\langle \sqrt{\rangle}$, а конечные значения всех переменных x^{\vee} равны их начальным значениям. Два этих условия, как утверждает следующий закон, являются также и достаточными:

L1. Если $S(x, \langle \rangle, x^{\vee})$ и $S(x, \langle \sqrt{\rangle}, x)$, то **ПРОПУСК** уд $S(x, np, x^{\vee})$.

X6. Самая сильная спецификация, которой удовлетворяет **ПРОПУСК** — это

$$\text{ПРОПУСК}_A \text{ уд } (np = \langle \rangle \vee (np = \langle \sqrt{\rangle} \& x^{\vee} = x)),$$

где x — список всех переменных из A , а x^{\vee} — список этих же переменных, но помеченных символом $\sqrt{}$. Пример **X6** непосредственно следует из **L1**, и наоборот.

X7. **ПРОПУСК** уд $(r < y \Rightarrow (T(n+1) \Rightarrow \text{ДЕЛЦИКЛ}))$

Доказательство:

(1) Замена nr на $\langle \rangle$ в спецификации дает нам

$$r < y \& T(n+1) \Rightarrow \langle \rangle = \langle \rangle \vee \dots,$$

что является тавтологией.

(2) Замена nr на $\langle \sqrt{\ } \rangle$, а конечных значений на начальные дает нам

$$r < y \& T(n+1) \Rightarrow (\langle \sqrt{\ } \rangle = \langle \rangle \vee (\langle \sqrt{\ } \rangle = \langle \sqrt{\ } \rangle \& x = x \& y = y \& r = ((q - q) \times y + r) \& r < y))$$

что также является тривиальным утверждением. Доказанный результат будет использован в **X10**.

Предусловие успешного присваивания $x := e$ заключается в том, что выражение e в правой части должно быть определено. В этом случае если P удовлетворяет спецификации $S(x)$, то $(x := e; P)$ удовлетворяет той же спецификации, но начальное значение x равно e .

L2. Если P уд $S(x)$, то $(x := e; P)$ уд $(\mathcal{D}e \Rightarrow S(e))$

Закон для простого присваивания можно вывести из **L2**, заменив P на **ПРОПУСК** и используя **X6** и 5.2 **L1**:

L2A. $x_0 := e$ уд $(\mathcal{D}e \& nr \neq \langle \rangle \Rightarrow nr = \langle \sqrt{\ } \rangle \& x_0^{\sqrt{\ }} = e \& x_1^{\sqrt{\ }} = x_1 \& \dots)$

Из **L2** следует, что самым сильным результатом, который можно доказать о $(x := 1/0; P)$, будет

$(x := 1/0; P)$ уд истина для любого P .

Какую бы содержательную цель мы себе ни ставили, ее не удастся достигнуть, начав с незаконного присваивания.

Примеры

X8. **ПРОПУСК** уд $(nr \neq \langle \rangle \Rightarrow nr = \langle \sqrt{\ } \rangle \& q^{\sqrt{\ }} = q \& r^{\sqrt{\ }} = r \& y^{\sqrt{\ }} = y \& x^{\sqrt{\ }} = x)$

следовательно, $(r := x - q \times y; \text{ПРОПУСК})$

уд $(x \geq q \times y \& nr \neq \langle \rangle \Rightarrow$

$nr = \langle \sqrt{\ } \rangle \& q^{\sqrt{\ }} = q \& r^{\sqrt{\ }} = (x - q \times y) \& y^{\sqrt{\ }} = y \& x^{\sqrt{\ }} = x)$

и $(q := x \div y; r := x - q \times y)$ уд $(y > 0 \& x \geq (x \div y) \times y \& nr \neq \langle \rangle \Rightarrow$

$nr = \langle \sqrt{\ } \rangle \& q^{\sqrt{\ }} = (x \div y) \& r^{\sqrt{\ }} =$
 $= (x - (x \div y) \times y) \& y^{\sqrt{\ }} = y \& x^{\sqrt{\ }} = x)$

Последняя строка спецификации эквивалентна ДЕЛ, определенной в Х4.

Х9. Предположим, что X уд $(T(n) \Rightarrow \text{ДЕЛЦИКЛ})$. Значит, $(r := r - y; X)$ уд $(y \leq r \Rightarrow (r - y < n \times y \Rightarrow (np = \langle \rangle \vee np = \langle \surd \rangle \& (r - y) = \dots)))$ и $(q := q + 1; r := r - y; X)$ уд $(y \leq r \Rightarrow (r < (n + 1) \times y \Rightarrow \text{ДЕЛЦИКЛ}'))$

где $\text{ДЕЛЦИКЛ}' = (np = \langle \rangle \vee (np = \langle \surd \rangle \& (r - y) = (q^\surd - (q + 1)) \times y + r^\surd \& r^\surd < y \& x^\surd = x \& y^\surd = y))$

Элементарная алгебра натуральных чисел позволяет заключить, что

$$y \leq r \Rightarrow (\text{ДЕЛЦИКЛ}' \equiv \text{ДЕЛЦИКЛ})$$

Следовательно, $(q := q + 1; r := r - y; X)$ уд $(y \leq r \Rightarrow (T(n + 1) \Rightarrow \text{ДЕЛЦИКЛ}))$. Этот результат будет использован в Х10.

Общий случай последовательной композиции требует более сложного закона, в котором протоколы компонент последовательно объединяются, причем начальное состояние второй компоненты совпадает с конечным состоянием первой компоненты. Значения переменных в этом промежуточном состоянии, однако, недоступны; гарантируется лишь сам факт их существования.

Л3. Если P уд $S(x, np, x^\surd)$, а Q уд $T(x, np, x^\surd)$ и P не расходуется, то

$$(P; Q) \text{ уд } (\exists y, s, t. np = (s; t) \& S(x, s, y) \& T(y, t, x^\surd))$$

В этом законе x обозначает список всех переменных из алфавитов процессов P и Q , x^\surd — список этих же переменных, помеченных галочкой, а y — список того же числа новых переменных.

Спецификация условного оператора совпадает со спецификацией первой компоненты, если условие истинно, и со спецификацией второй компоненты, если условие ложно:

Л4. Если P уд S и Q уд T , то $(P \nless b \nless Q)$ уд $((b \& S) \vee (\neg b \& T))$.

Иногда бывает удобнее использовать этот закон в другой форме:

Л4А. Если P уд $(b \Rightarrow S)$ и Q уд $(\neg b \Rightarrow S)$, то $(P \nless b \nless Q)$ уд S .

Пример

X10. Пусть $УСЛ = (q := q + 1; r := r - y; X) \nLeftarrow r \geq y \nrightarrow ПРОПУСК$,
а X уд $(T(n) \Rightarrow ДЕЛЦИКЛ)$.
Тогда $УСЛ$ уд $(T(n + 1) \Rightarrow ДЕЛЦИКЛ)$.

Два достаточных для такого заключения условия были доказаны в **X7** и **X9**, результат же следует из **L4A**.

В доказательстве для циклов используется рекурсивное определение 5.5 **D1** и закон для непредваренной рекурсии (3.7.1 **L8**). Если мы хотим доказать, что цикл удовлетворяет спецификации R , то мы должны найти такую спецификацию $S(n)$, что $S(0)$ истинна и, кроме того, $(\forall n. S(n)) \Rightarrow R$. Общим способом построения $S(n)$ является нахождение предиката $T(n, x)$, описывающего условия на начальных состояниях x , при которых известно, что цикл завершится менее чем за n повторений. После этого определяют $S(n) = (T(n, x) \Rightarrow R)$. Очевидно, что ни один цикл не может завершиться прежде, чем он сделает нуль повторений, и поэтому, если $T(n, x)$ был определен корректно, $T(0, x)$ будет ложным и, следовательно, $S(0)$ будет истинным. Результатом доказательства для цикла будет $\forall n. S(n)$, т. е. $\forall n. (T(n, x) \Rightarrow R)$. Так как n была выбрана как переменная, не входящая в R , эта запись эквивалентна следующей: $(\exists n. T(n, x)) \Rightarrow R$. Добиться соответствия какой-либо более сильной спецификации, вероятно, не удастся, поскольку $\exists n. T(n, x)$ — это предусловие, при выполнении которого цикл завершается после некоторого конечного числа повторений.

И наконец, мы должны доказать, что тело цикла соответствует своей спецификации. Так как рекурсивное уравнение для цикла содержит условное выражение, наша задача разбивается на две. Таким образом, мы приходим к общему закону:

L5. Если $\neg T(0, x)$ и $T(n, x) \Rightarrow \mathcal{D}b$ и $ПРОПУСК$ уд $(\neg b \Rightarrow (T(n, x) \Rightarrow R))$, а
 $(X \text{ уд } (T(n, x) \Rightarrow R)) \Rightarrow ((Q; X) \text{ уд } (b \Rightarrow (T(n + 1, x) \Rightarrow R)))$,
то $(b * Q) \text{ уд } ((\exists n. T(n, x)) \Rightarrow R)$.

Пример

X11. Мы хотим доказать, что программа медленного деления методом повторного вычитания (5.5 **X5**) соответствует спецификации **ДЕЛ**. Эта задача естественным образом разбивается на две. Вторая, и более трудная, ее часть — это доказательство того, что цикл удовлетворяет некоторой подхо-

дящим образом сформулированной спецификации, а именно:

$$(r \geq y) * (q := q + 1; r := r - y) \text{ уд } (y > 0 \Rightarrow \text{ДЕЛЦИКЛ})$$

Сначала необходимо сформулировать условие, при котором цикл завершается менее чем за n итераций: $T(n) = r < n \times y$. Очевидно, что условие $T(0)$ здесь ложно, а выражение $\exists n. T(n)$ эквивалентно $y > 0$, что и является предусловием, при выполнении которого цикл завершается. Заключительные шаги доказательства для цикла уже предпринимались в **X7** и **X5**, так что остаток доказательства представляет собой простое упражнение.

Согласно нашему замыслу, законы, приведенные в этом разделе, представляют собой исчисление полной корректности для чисто последовательных программ, не содержащих ввода и вывода. Если Q — такая программа, то, доказав, что

$$Q \text{ уд } (P(x) \ \& \ nr \neq \langle \rangle \Rightarrow nr = \langle \surd \rangle \ \& \ R(x, x^\surd)) \quad (1)$$

мы можем сделать вывод, что если в момент начала работы Q предикат $P(x)$ истинен на начальных значениях переменных, то Q обязательно завершится, а $R(x, x^\surd)$ будет описывать отношение между начальными данными x и конечными данными x^\surd . Таким образом, $(P(x), R(x, x^\surd))$ образуют пару предусловие/постусловие в смысле Клиффа Джоунса. Если $R(x^\surd)$ не содержит начальные значения x , утверждение (1) эквивалентно

$$P(x) \Rightarrow \wp r(Q, R(x))$$

где $\wp r$ — слабейшее предусловие Дейкстры.

Таким образом, в частном случае невзаимодействующих между собой программ методы доказательства математически эквивалентны уже известным, хотя явно встречающиеся условия " $nr = \langle \rangle$ " и " $nr = \langle \surd \rangle$ " делают их запись более громоздкой. Однако при расширении этих методов на случай взаимодействующих процессов эта дополнительная сложность становится необходимой и потому уже не столь обременительной.

5.5.3. Реализация

Начальное и конечное состояния последовательного процесса можно представить как функцию, отображающую имя каждой переменной в ее значение. Последовательный процесс определяется как функция, отображающая его начальное состояние в его дальнейшее поведение. Успешное завершение (\surd) представляется атомом "УСПЕХ, Готовый к заверше-

нию процесс допускает этот символ и отображает его не в другой процесс, а в конечное состояние своих переменных.

Таким образом, процесс *ПРОПУСК* берет в качестве параметра начальное состояние, в качестве единственного действия допускает "*УСПЕХ*" и выдает начальное состояние в качестве конечного:

$$\text{ПРОПУСК} = \lambda s. \lambda y. \text{ if } y \neq \text{"УСПЕХ"} \text{ then "BLEEP" else } s$$

Присваивание работает аналогично, лишь слегка меняется его конечное состояние:

$$\text{присв}(x, e) = \lambda s. \lambda y. \text{ if } y \neq \text{"УСПЕХ"} \text{ then "BLEEP"} \\ \text{else обновл}(s, x, e)$$

где $\text{обновл}(s, x, e) = \lambda y. \text{ if } y = x \text{ then вычисл}(e, s) \text{ else } s(y)$, а $\text{вычисл}(e, s)$ — результат вычисления выражения e в состоянии s . Если в состоянии s выражение e не определено, происходящее нас не интересует. Здесь, для простоты, мы реализовали только простое присваивание. Реализация множественного присваивания немного сложнее.

При реализации последовательной композиции вначале необходимо проверить, не завершился ли первый операнд. Если это так, то его конечное состояние передается второму операнду. Если нет, выполняется первое действие первого операнда.

$$\text{послед}(P, Q) = \\ \lambda s. \text{ if } P(s) \text{ ("УСПЕХ")} \neq \text{"BLEEP"} \text{ then } Q(P(s) \text{ ("УСПЕХ")}) \\ \text{else } \lambda y. \text{ if } P(s)(y) = \text{"BLEEP"} \text{ then "BLEEP"} \\ \text{else послед}(P(s)(y), Q)$$

Условный оператор реализуется условным выражением

$$\text{услов}(P, b, Q) = \lambda s. \text{ if вычисл}(b, s) \text{ then } P(s) \text{ else } Q(s)$$

Реализацию цикла мы оставляем в качестве упражнения.

Заметим, что приведенное выше определение *послед* сложнее, чем в разд. 5.3.3, поскольку состояние s является его первым аргументом, а кроме того, передается в качестве первого аргумента его операндам. Аналогичная сложность возникает, к сожалению, и при определении всех остальных операторов, описанных в предыдущих главах. Другой и более простой возможностью было бы моделирование переменных с помощью подчиненных процессов; но, скорее всего, это было бы гораздо менее эффективно, нежели использование обычной памяти с произвольной выборкой. Когда к соображениям математического удобства добавляются соображения эффективности, то не лишено основания трактовать переменную с присваиваемыми ей значениями как примитивное понятие, а не определять ее в терминах понятий, введенных ранее.

Глава 6. Разделяемые ресурсы

6.1. ВВЕДЕНИЕ

В разд. 4.5 мы ввели понятие именованного подчиненного процесса $(m:R)$, единственной обязанностью которого было удовлетворение потребностей одного главного процесса S ; для этого мы использовали обозначение $(m://S)$. Предположим теперь, что S состоит из двух параллельных процессов $(P\parallel Q)$ и они *оба* нуждаются в услугах одного и того же подчиненного процесса $(m:R)$. К сожалению, P и Q не могут взаимодействовать с R по одним и тем же каналам, потому что тогда эти каналы должны содержаться в алфавитах обоих процессов, и, значит, согласно определению \parallel , взаимодействия с $(m:R)$ могут происходить, только когда P и Q одновременно посылают одно и то же сообщение, что (как мы видели в 4.5 X6) далеко не соответствует желаемому результату. Нам же требуется своего рода чередование взаимодействий между P и $(m:R)$ с взаимодействиями между Q и $(m:R)$. В этом случае $(m:R)$ служит как ресурс, разделяемый P и Q ; каждый из них использует его независимо, и их взаимодействия с ним чередуются.

Когда все процессы-пользователи известны заранее, можно организовать работу так, чтобы каждый процесс-пользователь имел свой набор каналов для взаимодействий с совместно используемым ресурсом. Эта техника применялась в задаче об обедающих философах (разд. 2.5): каждая вилка совместно использовалась двумя соседними философами, а слуга пользовался всеми пятью. В качестве другого примера можно привести 4.5 X6, где буфер совместно использовался двумя процессами, один из которых пользовался только левым, а другой — только правым каналом. Общий метод разделения ресурсов дает множественная пометка (разд. 2.6.4), которая является эффективным средством создания достаточного числа каналов для независимого взаимодействия с каждым процессом-пользователем. Отдельные взаимодействия по этим каналам произвольно чередуются. Однако при таком методе необходимо знать заранее имена всех процессов-пользователей, и поэтому он не подходит для подчиненного процесса, предназначенного для обслуживания основ-

ного процесса, который разбивается на произвольное число параллельных подпроцессов. Данная глава знакомит с техникой разделения ресурсов между многими процессами, имена и количество которых могут быть заранее неизвестны. Эта техника иллюстрируется примерами, взятыми из разработки операционных систем.

6.2. ПООЧЕРЕДНОЕ ИСПОЛЬЗОВАНИЕ

Проблема, описанная в разд. 6.1, возникает вследствие использования при описании параллельного поведения процессов комбинирующего оператора \parallel , и часто этой проблемы можно избежать, используя параллелизм в форме чередования ($P \parallel Q$). Здесь P и Q имеют одинаковые алфавиты, а их взаимодействия с внешними (совместно используемыми) процессами произвольно чередуются. Это, конечно, делает невозможным прямое взаимодействие между P и Q ; но не прямые сообщения можно поддерживать с помощью соответствующим образом построенного разделяемого подчиненного процесса, как показано в примере 4.5 X6 и в примере X2, приведенном ниже.

Примеры

X1. Совместно используемая подпрограмма

удв : $УДВ \parallel (P \parallel Q)$

Здесь и P , и Q могут содержать вызов подчиненного процесса

$(удв.лев!v \rightarrow удв.прав?x \rightarrow ПРОПУСК)$

Несмотря на то что пары взаимодействий P и Q произвольно чередуются, опасности, что один из процессов случайно получит ответ, предназначенный для другого, быть не должно. Чтобы это гарантировать, все подпроцессы основного процесса должны строго чередовать взаимодействия по левому каналу с взаимодействиями по правому каналу совместно используемой подпрограммы. По этой причине мы считаем целесообразным введение специального обозначения, которое будет использоваться исключительно в целях соблюдения требуемой дисциплины. Предложенное обозначение напоминает традиционный вызов процедуры в языках высокого уровня, за исключением того, что значениям параметров предшествует символ $!$, а результирующим параметрам — символ $?$. Таким образом,

$удв!x?y = (удв.лев!x \rightarrow удв.прав?y \rightarrow ПРОПУСК)$

Предполагаемый эффект поочередного использования иллюстрируется следующим рядом алгебраических преобразований. Когда два процесса одновременно пытаются использовать разделяемую подпрограмму, согласованные пары взаимодействий чередуются в произвольном порядке, но компоненты пары взаимодействий с одним процессом никогда не разделяются взаимодействием с другим процессом. Для удобства мы используем следующие сокращения:

$$\left. \begin{array}{l} d!v \text{ для } d.\textit{лев}!v \\ d?x \text{ для } d.\textit{прав}?x \end{array} \right\} \text{ внутри главного процесса}$$

$$\left. \begin{array}{l} !v \text{ для } \textit{прав}!v \\ ?x \text{ для } \textit{лев}?x \end{array} \right\} \text{ внутри подчиненного процесса}$$

Пусть $D = ?x \rightarrow !(x + x) \rightarrow D$

$$P = d!3 \rightarrow d?y \rightarrow P(y)$$

$$Q = d!4 \rightarrow d?z \rightarrow Q(z)$$

$$R = (d : D // (P \parallel Q)) \quad (\text{как в X1 выше})$$

Тогда $P \parallel Q = (d!3 \rightarrow ((d?y \rightarrow P(y)) \parallel Q)) \parallel d!4 \rightarrow (P \parallel (d?z \rightarrow Q(z)))$ по 3.6.1 L7

Каждый процесс-пользователь начинает с вывода в сторону разделяемого процесса, которому и предлагается сделать выбор между ними. Но так как разделяемый процесс готов к приему любого из этих сообщений, то после упрятывания выбор становится недетерминированным.

$$\begin{aligned} (d : D // (P \parallel Q)) &= ((d : (13 + 3 \rightarrow D)) // ((d?y \rightarrow P(y)) \parallel Q)) \quad \left\{ \begin{array}{l} 4.5.1 \text{ L1} \\ \cap ((d : (14 + 4 \rightarrow D)) // (P \parallel (d?z \rightarrow Q(z)))) \quad \left\{ \begin{array}{l} 3.5.1 \text{ L5} \\ = (d : D // (P(6) \parallel Q)) \cap (d : D // (P \parallel Q(8))) \quad \text{и т. д.} \end{array} \right. \end{array} \right. \end{aligned}$$

Разделяемый процесс предлагает свой результат любому из процессов-пользователей, готовых принять его; а так как один из этих процессов все еще находится в состоянии ожидания вывода, результат достается тому процессу, который передал аргументы. Вот почему при вызове совместно используемой подпрограммы так важно жесткое чередование ввода и вывода.

X2. Разделяемая структура данных

В системе бронирования авиабилетов одновременно работает много разных кассиров, поочередно регистрирующих заявки. Процесс бронирования состоит в занесении фамилии пассажира в список пассажиров данного рейса и передаче специального признака, был ли данный пассажир уже зарегистрирован или нет. В этом переупрощенном примере в качестве разделяемого подчиненного процесса можно восполь-

зоваться построенным в 4.5 X8 множеством с номером рейса в качестве имени:

$AG109 : \text{МНОЖ} // (\dots (\text{КАССИР} \parallel \text{КАССИР} \parallel \dots) \dots)$

Каждый *КАССИР* регистрирует пассажира, делая вызов

$AG109!nacc \text{ нет?}x,$

что соответствует более полной записи

$(AG109.лев!nacc \text{ нет} \rightarrow AG109.прав?x \rightarrow \text{ПРОПУСК})$

В этих двух примерах каждое использование разделяемого ресурса включает ровно два взаимодействия: одно — посылающее параметры, а другое — принимающее результаты; после каждой пары таких взаимодействий подчиненный процесс возвращается в состояние, в котором он готов обслуживать другой или тот же процесс. Однако часто мы хотим сделать так, чтобы вся цепь взаимодействий происходила только между двумя процессами без угрозы вмешательства третьего. Например, одно дорогостоящее устройство вывода может совместно использоваться несколькими параллельными процессами. При каждом его использовании требуется, чтобы строки, составляющие выходной файл, выводились последовательно без угрозы того, что текст будет перемежаться строками, посылаемыми другим процессом. С этой целью выводу каждого файла может предшествовать событие *занят*, указывающее, что ресурс переходит в пользование к данному процессу; по завершении же работы событие *свободен* вновь делает ресурс доступным.

X3. Совместно используемое алфавитно-цифровое печатающее устройство:

$АЦПУ = занят \rightarrow \mu X. (лев?s \rightarrow h!s \rightarrow X \mid свободен \rightarrow АЦПУ)$

Здесь *h* — канал, соединяющий *АЦПУ* с аппаратной частью устройства. После наступления события *занят АЦПУ* копирует в аппаратную часть последовательно поступающие по левому каналу строки, пока сигнал *свободен* не приведет его в исходное состояние, в котором он доступен для использования другими процессами. Этот процесс используется как разделяемый ресурс:

$ацпу : АЦПУ // \dots (P \parallel Q) \dots$

Внутри *P* или *Q* вывод последовательности строк, образующей файл, заключен между событиями *ацпу.занят* и *ацпу.свободен*:

$ацпу.занят \rightarrow \dots ацпу.лев! "Э. Джонс" \rightarrow \dots$

$ацпу.лев! следстр \rightarrow \dots ацпу.свободен \rightarrow \dots$

Х4. Усовершенствованный пример Х3

Когда печатающее устройство имеет много пользователей, часть бумаги, содержащую каждый напечатанный файл, приходится вручную отделять от распечаток предыдущих и последующих файлов. Для этого бумага, как правило, разбита на страницы, разделенные перфорацией, а аппаратная часть устройства имеет операцию *прогон*, быстро сдвигающую бумагу к концу текущей страницы или, еще лучше, к началу очередной лицевой стороны бумаги в стопке. Чтобы было удобнее отделять друг от друга различные выдачи, файлы должны начинаться и кончаться с самого края страницы, а в конце последней и в начале первой страницы файла должен печататься целый ряд звездочек. Чтобы не возникало путаницы, запрещается печатать полную строку звездочек в середине файла.

$$\begin{aligned} \text{АЦПУ} = & (h! \text{прогон} \rightarrow h! \text{звездочки} \rightarrow \text{занят} \rightarrow h! \text{звездочки} \rightarrow \\ & \mu X. (\text{лев?} s \rightarrow \text{if } s \neq \text{звездочки then } (h!s \rightarrow X) \text{ else } X \\ & \quad | \text{свободен} \rightarrow \text{АЦПУ}) \end{aligned}$$

Этот вариант АЦПУ используется точно так же, как и предыдущий.

В последних двух примерах использование сигналов *занят* и *свободен* препятствует произвольному чередованию строк из различных файлов и при этом не угрожает системе дедлоком. Но если совместно использовать таким образом более одного ресурса, то риск тупиковой ситуации уже нельзя не учитывать.

Х5. Дедлок

Энн и Мери хорошие подруги и хорошие хозяйки. Они пользуются общим котелком и общей сковородой, которые занимают, используют и освобождают по необходимости.

$$\begin{aligned} \text{ПОСУДА} = & (\text{занят} \rightarrow \text{использ} \rightarrow \\ & \quad \text{использ} \rightarrow \dots \rightarrow \text{свободен} \rightarrow \text{ПОСУДА}) \\ \text{котелок} : & \text{ПОСУДА} // \text{сковорода} : \text{ПОСУДА} // (\text{ЭНН} \parallel \text{МЕРИ}) \end{aligned}$$

Энн готовит по рецепту, согласно которому сначала требуется котелок, а затем — сковорода, тогда как Мери сначала нужна сковорода, а затем — котелок.

$$\begin{aligned} \text{ЭНН} = & \text{котелок.занят} \rightarrow \dots \text{сковорода.занят} \rightarrow \dots \\ \text{МЕРИ} = & \dots \text{сковорода.занят} \rightarrow \dots \text{котелок.занят} \rightarrow \dots \end{aligned}$$

На свою беду, они решили приготовить обед примерно в одно и то же время. Каждая взяла нужную ей посуду, но когда ей потребовалось взять другую, оказалось, что сделать это невозможно, так как посуда занята другой хозяйкой.

Историю об Энн и Мери можно наглядно проиллюстрировать в виде двумерной диаграммы, где жизнь Энн откладывается вдоль вертикальной оси, а жизнь Мери — вдоль горизонтальной. Система начинает работу с левого нижнего угла, где начинается жизнь Энн и Мери. При каждом действии Энн система сдвигается на один шаг вверх. При каждом действии Мери система сдвигается на один шаг вправо. Изображенная на графике траектория представляет типичное

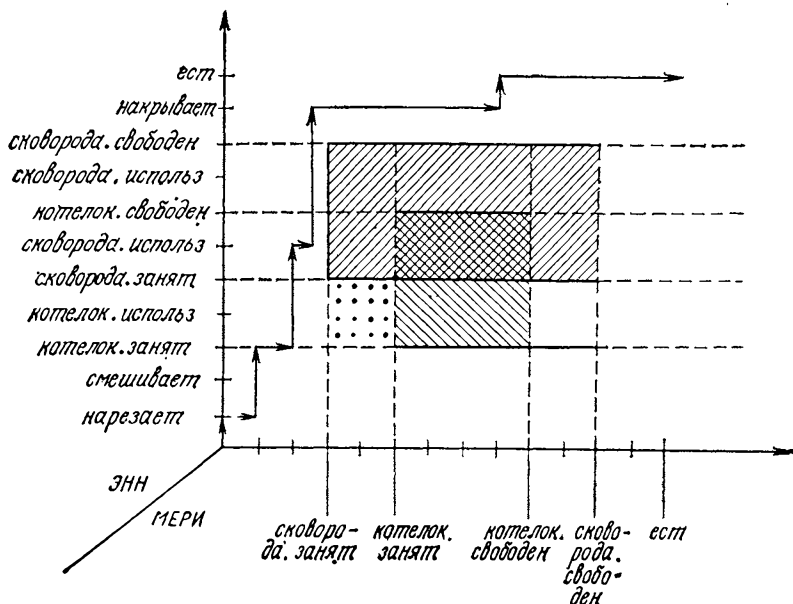


Рис. 6.1

чередование действий Энн и Мери. К счастью, данная траектория достигает правого верхнего угла диаграммы, где у обеих хозяек готов обед.

Но уверенности в таком благополучном исходе у нас нет. Поскольку хозяйки не могут одновременно пользоваться общей посудой, на нашей диаграмме существуют некоторые прямоугольные участки, пересекать которые траектория не может. Например, на участке, заштрихованном в направлении /, обеим хозяйкам требуется сковорода, что мы обеспечить не можем. Аналогично невозможность одновременного пользования котелком не позволяет траектории пересекать участок, заштрихованный в направлении \. Таким образом, если траектория достигает одной из запретных зон, она может проходить только вдоль нее (вверх — в случае верти-

кальной границы и вправо — в случае горизонтальной). В течение этого времени одна из хозяек ожидает, пока другая не освободит нужную ей посуду.

Теперь рассмотрим участок, отмеченный точками. Если траектория попадает внутрь его, в правом верхнем углу этого участка неизбежно возникает дедлок. Цель данной диаграммы — показать, что опасность дедлока возникает только вследствие наличия вогнутости в ближайшем к началу участка запретной зоны. Все остальные вогнутости вполне безопасны. Кроме того, из этого рисунка видно, что единственный надежный способ избежать дедлока — это расширить запретный участок, покрывающий опасную зону так, чтобы вогнутость исчезла. Технически это можно осуществить введением дополнительного искусственного ресурса, который необходимо занять прежде, чем котелок или сковороду, и не освобождать до тех пор, пока оба предмета не освободятся. Это решение аналогично введению слуги в истории об обедающих философах (разд. 2.5.3), где разрешение сесть за стол играет роль своеобразного ресурса, четыре экземпляра которого совместно используются пятью философами. Более простым решением будет требование, чтобы хозяйка, желающая использовать оба предмета, сначала брала сковороду. Этот пример принадлежит Э. Дейкстре.

Более простое решение, предложенное в предыдущем примере, обобщается на любое число пользователей и ресурсов. При условии что все пользователи занимают нужные им ресурсы в некотором фиксированном порядке, опасности дедлока не возникает. Пользователи должны лишь освобождать ресурсы, как только они закончили работу с ними; порядок освобождения значения не имеет. Пользователь даже может занять ресурс и не в установленном порядке, при условии, что к этому моменту он освободил все ресурсы, стоящие позже в стандартном упорядочении. Соблюдение этого порядка пользования ресурсами часто можно проверить обычным визуальным просмотром текста процесса-пользователя.

6.3. ОБЩАЯ ПАМЯТЬ

Цель этого раздела — привести ряд аргументов против использования общей памяти. Те, кого в этом убеждать не надо, могут его пропустить.

Поведение систем параллельных процессов может быть без труда реализовано на обыкновенной вычислительной машине с хранимой программой с помощью техники, известной как режим деления времени, при котором единственный

процессор поочередно выполняет каждый из процессов, причем смена выполняемого процесса происходит по прерыванию, исходящему от некоторого внешнего или от синхронизирующего устройства. При такой реализации очень легко позволить параллельным процессам совместно использовать общую память, выборка и загрузка которой осуществляются посредством обычных машинных инструкций, содержащихся в коде каждого процесса.

Ячейку общей памяти в нашей теории можно промоделировать разделяемой переменной (4.2 X7) с подходящим именем, например:

(счет : ПЕРЕМ // (счет.лев!0 \rightarrow (P \parallel Q)))

Общую память надо четко отличать от локальной памяти, описанной в разд. 5.5. Простота законов для последовательных процессов вытекает исключительно из того, что значение каждой переменной изменяется не более чем одним процессом и что в этих законах не приходится иметь дела ни с одной из многочисленных опасностей, возникающих в результате произвольного чередования присваиваний, выполняемых различными процессами.

Наиболее полно эти опасности иллюстрирует следующий пример.

X1. Взаимное влияние

Разделяемая переменная *счет* используется для подсчета числа исполнений некоторого важного события. При каждом наступлении этого события соответствующий процесс *P* или *Q* пытается изменить значение счетчика парой взаимодействий

счет.прав?x; счет.лев!(x + 1)

К сожалению, эти два взаимодействия могут перемежаться аналогичной парой взаимодействий от другого процесса, в результате чего мы получим последовательность

*счет.прав?x \rightarrow счет.прав?y \rightarrow счет.лев!(y + 1) \rightarrow
 \rightarrow счет.лев!(x + 1) \rightarrow ...*

В итоге значение счетчика увеличится лишь на единицу, а не на два. Такого рода ошибки известны как *взаимное влияние* и часто допускаются при проектировании процессов, совместно использующих общую память. Кроме того, реальное проявление такой ошибки в высшей степени недетерминировано; ее воспроизводимость очень ненадежна, и поэтому ее практически невозможно диагностировать обычными методами тестирования. Полагаю, что в результате всего этого немалое число широко используемых операционных систем

регулярно позволяет себе небольшие погрешности в выдаваемых сводках, статистиках и счетах.

Возможным решением этой проблемы может быть контроль за тем, чтобы смена процесса не происходила при совершении последовательности действий, нуждающихся в защите от чередования. Такая последовательность называется *критическим участком*. При реализации с одним процессором требуемое исключение обычно достигается запрещением всех прерываний на протяжении критического участка. Нежелательным эффектом такого решения является задержка ответа на прерывание; но хуже всего то, что оно оказывается полностью несостоятельным, как только к машине добавляется второй процессор.

Более приемлемое решение было предложено Э. Дейкстрой, которому принадлежит идея использования двоичных семафоров. Семафор можно описать как процесс, поочередно выполняющий действия с именами P и V :

$$SEM = (P \rightarrow V \rightarrow SEM)$$

Он описывается как совместно используемый ресурс

$$(vзаискл : SEM // \dots).$$

При условии что все процессы подчиняются этой дисциплине, каждый из двух процессов не сможет влиять на изменение счетчика — произвести действие *vзаискл.V*. Таким образом, критический участок, на котором происходит увеличение счетчика, должен иметь вид

$$\begin{aligned} &vзаискл.P \rightarrow \\ &счет.прав?x \rightarrow счет.лев!(x + 1) \rightarrow \\ &vзаискл.V \rightarrow \dots \end{aligned}$$

При условии что все процессы подчиняются этой дисциплине, каждый из двух процессов не сможет влиять на изменение счетчика своим партнером. Но если какой-нибудь процесс пропустит P или V или выполнит их в обратном порядке, результат будет непредсказуемым и может привести к катастрофической или (что, возможно, еще хуже) неуловимой ошибке.

Избежать взаимного влияния гораздо более надежным способом можно,строив необходимую защиту в саму конструкцию общей памяти, воспользовавшись знанием о предполагаемом способе ее использования. Если, например, переменная будет использоваться только как счетчик, то ее увеличение можно задать одной элементарной операцией

счет.вверх, а соответствующий разделяемый ресурс определить как CT_0 (1.1.4 X2):

$$\text{счет} : CT_0 // (\dots P \parallel Q \dots)$$

На самом деле, есть все основания рекомендовать, чтобы каждый совместно используемый ресурс заранее проектировался для своих целей и чтобы в разработке системы с элементами параллелизма универсальная память не использовалась совместно. Этот метод не только предупреждает серьезную опасность случайного взаимного влияния, но и позволяет получать конструкции, поддающиеся эффективной реализации на сетях распределенных процессорных элементов, а также на одно- и многопроцессорных ЭВМ с физически общей памятью.

6.4. КРАТНЫЕ РЕСУРСЫ

В разд. 6.2 мы описали, как некоторое число параллельных процессов с различным поведением могут совместно использовать один подчиненный процесс. Каждый процесс-пользователь соблюдает дисциплину чередования ввода и вывода или чередования сигналов *занят/свободен* с тем, чтобы в каждый момент времени разделяемый ресурсом пользовался не более чем один процесс. Такие ресурсы называют *последовательно переиспользуемыми*. В этом разделе мы вводим массивы процессов, представляющие кратные ресурсы с одинаковым поведением; индексы в массиве обеспечивают тот факт, что каждый элемент достоверно взаимодействует только с использующим его процессом.

Поэтому здесь мы будем широко использовать индексы и операторы с индексами, смысл которых очевиден. Например:

$$\bigparallel_{i < 12} P_i = (P_0 \parallel P_1 \parallel \dots \parallel P_{11})$$

$$\bigparallel_{i < 4} P = (P \parallel P \parallel P \parallel P)$$

$$\bigparallel_{i \geq 0} P_i = (P_0 \parallel P_1 \parallel \dots)$$

$$\bigparallel_{i \geq 0} (f(i) \rightarrow P_i) = (f(0) \rightarrow P_0 \mid f(1) \rightarrow P_1 \mid \dots)$$

В последнем примере мы требуем, чтобы f была взаимно однозначной для того, чтобы выбор между альтернативами осуществлялся обстановкой,

Примеры

Х1. Повторно входимая подпрограмма

Последовательно переиспользуемая общая подпрограмма может обслуживать вызывающие ее процессы только по одному. Если выполнение подпрограммы требует значительных вычислений, соответствующие задержки могут возникнуть и в вызывающем процессе. Если же для вычислений доступны несколько процессоров, нам ничто не мешает позволить нескольким экземплярам подпрограммы параллельно исполняться на различных процессорах. Подпрограмма, имеющая несколько параллельно работающих экземпляров, называется *повторно входимой* и определяется как массив параллельных процессов

$$удв : \left(\parallel_{i < 27} (i : УДВ) \right) // \dots$$

Типичным вызовом этой подпрограммы будет

$$(удв.3.лев!30 \rightarrow удв.3.прав?у \rightarrow ПРОПУСК)$$

Присутствие индекса 3 гарантирует, что результат вызова получен от того же самого экземпляра *удв*, которому были посланы аргументы, даже несмотря на то, что в это же время некоторые другие параллельные процессы могут вызывать другие элементы массива, что приводит к чередованию сообщений типа

$$\begin{aligned} удв.3.лев.30, \dots удв.2.лев.20, \dots удв.3.прав.60, \dots \\ удв.2.прав.40, \dots \end{aligned}$$

Когда процесс вызывает повторно входимую подпрограмму, на самом деле не имеет значения, какой из элементов массива ответит на этот вызов; годится любой в данный момент свободный процесс. Поэтому вместо того, чтобы указывать конкретный индекс 2 или 3, вызывающий процесс должен делать произвольный выбор, используя конструкцию

$$\Pi_{i \geq 0} (удв.i.лев!30 \rightarrow удв.i.прав?у \rightarrow ПРОПУСК)$$

При этом по-прежнему существенно требование, чтобы для передачи аргументов и (сразу после этого) получения результата использовался один и тот же индекс.

В приведенном нами примере введено произвольное ограничение на число одновременных активаций подпрограммы (в нашем случае — 27). Поскольку достаточно просто сделать так, чтобы процессор одновременно уделял внимание гораздо большему числу процессов, таких произвольных

ограничений можно избежать введением бесконечных массивов параллельных процессов: $удв: \left(\prod_{i \geq 0} i : D \right)$, где D теперь можно определить как процесс, обслуживающий единственный вызов, а затем останавливающийся:

$$D = лев?x \rightarrow прав!(x + x) \rightarrow СТОП$$

Подпрограмма, не имеющая ограничений на повторную входимость, называется *процедурой*¹⁾.

Процедура используется для того, чтобы эффект каждого вызова

$$\prod_{i \geq 0} (удв.i.лев!x \rightarrow удв.i.прав?y \rightarrow ПРОПУСК)$$

совпадал с вызовом подчиненного процесса D , описанного непосредственно перед этим:

$$(удв : D // (удв.лев!x \rightarrow удв.прав?y \rightarrow ПРОПУСК))$$

Последний называется *локальным* вызовом процедуры, поскольку предполагается, что выполнение процедуры происходит на том же процессоре, что и выполнение вызывающего процесса; в противном случае вызов общей процедуры называется *дистанционным*, поскольку предполагает исполнение на отдельном, возможно, удаленном процессоре. Так как эффект дистанционного и локального вызова должен быть одним и тем же, причины для использования именно дистанционного вызова могут быть только политическими или экономическими — например, для хранения кода процедуры в секрете или для исполнения его на машине, имеющей какие-то специальные средства, слишком дорогие для установки их на той машине, на которой исполняются процессы-пользователи.

Типичным примером таких дорогостоящих устройств могут служить внешние запоминающие устройства большой емкости — такие, как дисковая или доменная память.

Х2. Общая внешняя память

Запоминающая среда разбита на B секторов, запись и чтение с которых могут производиться независимо. В каждом секторе может храниться один блок информации, которая поступает слева и выводится направо. К несчастью, запоминающая среда реализована по технологии разрушающего

¹⁾ Заметим, что употребление термина «процедура» в указанном смысле не имеет прямой связи с понятием «процедура» в таких языках программирования, как Алгол 60, Алгол 68, Паскаль и т. д., так как в последних свойство повторной входимости семантикой языков не оговаривается. — *Прим. ред.*

считывания, так что каждый блок может быть считан только один раз. Таким образом, каждый сектор ведет себя скорее как *КОПИР* (4.2X1), нежели как *ПЕРЕМ* (4.2X7). Дополнительная память в целом представляет собой массив таких секторов с индексами, не превосходящими B :

$$\text{ДОППАМ} = \parallel_{i < B} i : \text{КОПИР}$$

Предполагается, что эта память используется как подчиненный процесс

$$(\text{доп} : \text{ДОППАМ} // \dots).$$

Внутри основного процесса доступ к этой памяти осуществляется взаимодействиями

$$\text{доп}.i.\text{лев!блок} \rightarrow \dots \text{доп}.i.\text{прав?у} \rightarrow \dots$$

Дополнительная память может также совместно использоваться параллельными процессами. В этом случае действие

$$\parallel_{i < B} (\text{доп}.i.\text{лев!блок} \rightarrow \dots)$$

одновременно займет произвольный свободный сектор с номером i и запишет в него значение *блок*. Аналогично, $\text{доп}.i.\text{прав?х}$ за одно действие считает содержимое сектора i в x и освободит этот сектор для дальнейшего использования, вполне вероятно, уже другим процессом. Именно это упрощение и послужило истинной причиной использования *КОПИР* для моделирования каждого сектора; рассказ же о разрушающем считывании приведен для пушей убедительности.

Конечно, успешное совместное использование этой дополнительной памяти требует от процессов-пользователей строжайшего соблюдения дисциплины. Процесс может совершить ввод информации с некоторого сектора, только если именно этот процесс последним выводил туда информацию, причем за каждым выводом рано или поздно должен последовать такой ввод. Несоблюдение этого порядка приводит к тупиковой ситуации или к еще худшим последствиям. Необременительный способ поддержания требуемого порядка описан после следующего примера и широко проиллюстрирован в последующем построении модулей операционной системы (разд. 6.5).

Х3. Два АЦПУ

Два одинаковых АЦПУ установлены для обслуживания группы процессов-пользователей. Оба они нуждаются в средствах защиты от чередования, имеющихся у процесса *АЦПУ*

(6.2X4). Поэтому мы описываем массив из двух экземпляров *АЦПУ*, каждый из которых имеет целый индекс, указывающий его позицию в массиве:

$$АЦПУ2 = (0 : АЦПУ \parallel 1 : АЦПУ)$$

Этот массив сам может получить имя и использоваться как разделяемый ресурс:

$$ацпу : АЦПУ2 // \dots)$$

Каждому вхождению *АЦПУ* теперь предшествуют две величины — имя и индекс, и поэтому его взаимодействия с процессом-пользователем состоят из трех-четырёх компонент, например:

$$ацпу.0.занят, ацпу.1.лев."Э.Джонс", \dots$$

Как и в случае повторно входимой процедуры, когда процесс хочет занять один ресурс из массива одинаковых ресурсов, какой именно элемент массива будет в этом случае выбран, значения не имеет. Приемлемым будет любой элемент, готовый к приему сигнала *занят*. Необходимый произвольный выбор осуществляется с помощью конструкции генерального выбора

$$\begin{aligned} \Pi_{i \geq 0} (ацпу.i.занят \rightarrow \dots ацпу.i.лев!x \rightarrow \dots ацпу.i.свободен \rightarrow \\ \rightarrow \text{ПРОПУСК}) \end{aligned}$$

Здесь в результате начального действия *ацпу.i.занят* будет занят тот из двух процессов *АЦПУ*, который готов к этому событию. Если не готов ни один, запрашивающий процесс будет ждать; если же готовы оба, выбор между ними недетерминирован. После того как произошло начальное событие *ацпу.i.занят*, связанная переменная *i* получает в качестве значения индекс выбранного ресурса, и все последующие взаимодействия будут происходить именно с этим ресурсом.

Когда разделяемый ресурс занимают для временного использования внутри другого процесса, он должен вести себя в точности как локально описанный подчиненный процесс, взаимодействующий только со своим главным процессом. Поэтому вместо громоздкой конструкции

$$\begin{aligned} \Pi_{i \geq 0} (ацпу.i.занят \rightarrow \dots ацпу.i.лев!x \dots; ацпу.i.свободен \rightarrow \\ \rightarrow \text{ПРОПУСК}) \end{aligned}$$

мы можем использовать слегка видоизмененное обозначение для подчинения и писать

$$(мойфайл :: ацпу // \dots мойфайл.лев!x \dots)$$

Введенное здесь локальное имя *мойфайл* соответствует имени с индексом *ацпу.i*, что позволяет скрыть все технические подробности, связанные с занятием и освобождением ресурса. Новое обозначение «::» называется *дистанционным подчинением* и отличается от знакомого нам «:» тем, что справа в нем стоит не процесс целиком, а имя отдельно расположенного массива процессов.

Х4. Два выходных файла

Процессу-пользователю одновременно требуются два АЦПУ для вывода двух файлов *f1* и *f2*:

$$(f1 :: ацпу // (f2 :: ацпу // \dots f1.лев!s1 \rightarrow f2.лев!s2 \rightarrow \dots))$$

Здесь процесс-пользователь по очереди выводит строки двух различных файлов, но печать каждой строки осуществляется соответствующим устройством. Конечно, любая попытка одновременно описать *три* печатающих устройства неминуемо приведет к дедлоку; то же, вероятнее всего, произойдет и при описании двух АЦПУ внутри каждого из двух параллельных процессов, в чем мы имели возможность убедиться на примере истории с Энн и Мери (6.2 Х5).

Х5. Вспомогательный файл

Вспомогательный файл используется для вывода последовательности блоков. Когда вывод завершается, происходит обратная перемотка файла, после чего вся последовательность блоков может считываться с самого начала. Когда считывание закончено, вспомогательный файл подает лишь сигнал *пуст*, а дальнейшее считывание или запись невозможны. Таким образом, вспомогательный файл ведет себя как магнитная лента, которая нуждается в перемотке перед считыванием. Сигнал *пуст* служит признаком конца файла.

$$\begin{aligned} \text{ВСПОМ} &= \text{ЗАПИСЬ}_{\langle \rangle} \\ \text{ЗАПИСЬ}_s &= (\text{лев?}x \rightarrow \text{ЗАПИСЬ}_{s \sim \omega}) \\ &\quad | \text{перемотка} \rightarrow \text{ЧТЕНИЕ}_{\langle \rangle} \\ \text{ЧТЕНИЕ}_{\omega \sim s} &= (\text{прав!}x \rightarrow \text{ЧТЕНИЕ}_s) \\ \text{ЧТЕНИЕ}_{\langle \rangle} &= (\text{пуст} \rightarrow \text{ЧТЕНИЕ}_{\langle \rangle}) \end{aligned}$$

Его удобно использовать как простой неразделяемый подчиненный процесс

$$\begin{aligned} (\text{мойфайл} : \text{ВСПОМ} // \dots \text{мойфайл.лев!}v \dots \\ \dots \text{мойфайл.перемотка} \dots \\ \dots (\text{мойфайл.прав?}x \rightarrow \dots \\ \quad | \text{мойфайл.пуст} \rightarrow \dots) \dots) \end{aligned}$$

В дальнейшем он послужит моделью совместно используемого процесса.

Х6. Вспомогательные файлы во внешней памяти

Вспомогательный файл, описанный в **Х5**, можно без труда реализовать так, чтобы последовательность блоков хранилась в основной памяти машины. Но если размер блоков велик, а последовательность длинная, это может привести к неэквивалентному расходованию основной памяти. Поскольку каждый блок во вспомогательном файле считывается и записывается только один раз, нам вполне достаточно дополнительной памяти с разрушающим считыванием (**Х2**). В оперативной же памяти хранится обычный вспомогательный файл, содержащий индексы секторов дополнительной памяти, на которых хранятся соответствующие реальные блоки информации; это обеспечивает требуемый порядок считывания нужных блоков:

$$\begin{aligned} \text{ДОПВСПОМ} = & (\text{таблстр} : \text{ВСПОМ} // \\ & \mu X. (\text{лев?}x \rightarrow (\prod_{i < V} \text{доп}.i.\text{лев!}x \rightarrow \text{таблстр}.\text{лев!}i \rightarrow X) \\ & \quad | \text{перемотка} \rightarrow \text{таблстр}.\text{перемотка} \rightarrow \\ & \quad \mu Y. (\text{таблстр}.\text{прав?}i \rightarrow \text{доп}.i.\text{прав?}x \rightarrow \text{прав!}x \rightarrow Y \\ & \quad | \text{таблстр}.\text{пуст} \rightarrow \text{пуст} \rightarrow Y))) \end{aligned}$$

ДОПВСПОМ использует имя *доп* для обращения к дополнительной памяти (**Х2**) как к подчиненному процессу. Чтобы это было возможно, надо описать

$$\text{ВСПОМДОП} = (\text{доп} : \text{ДОППАМ} // \text{ДОПВСПОМ})$$

ВСПОМДОП можно использовать как простой неразделяемый подчиненный процесс в точности так же, как вспомогательный файл из примера **Х5**:

$$(\text{мойфайл} : \text{ВСПОМДОП} // \dots \text{мойфайл}.\text{лев!}v \dots)$$

Результат его работы будет таким же, как в случае использования **ВСПОМ**, с той разницей, что максимальная длина вспомогательного файла не превышает *V* блоков.

Х7. Последовательно переиспользуемые вспомогательные файлы

Пусть мы хотим, чтобы вспомогательный файл совместно использовался несколькими чередующимися пользователями, которые бы поочередно занимали, использовали и освобождали его, как это было в примере с печатающим устройством (6.2 **Х3**). С этой целью мы должны изменить **ДОПВСПОМ**, чтобы он умел реагировать на сигналы *занят* — *свободен*. Если пользователь освобождает свой вспомогательный файл, не дочитав его до конца, существует опасность, что несчитанные блоки в дополнительной памяти останутся невостребованными. Ее можно избежать введением

цикла, считывающего, а затем сбрасывающего эту информацию:

$$\text{ПРОСМОТР} = \mu X. (\text{таблстр.} \text{прав?} i \rightarrow \text{доп.} i. \text{прав?} x \rightarrow X \\ | \text{таблстр.} \text{пусто} \rightarrow \text{ПРОПУСК})$$

Совместно используемый файл приобретает пользователя, после чего ведет себя как *ДОПВСПОМ*. Сигнал *свободен* вызывает прерывание (разд. 5.4) и запуск процесса *ПРОСМОТР*:

$$\text{СОВМДОПВСПОМ} = \text{занят} \rightarrow (\text{ДОПВСПОМ}^{\wedge} (\text{свободен} \rightarrow \rightarrow \text{ПРОСМОТР}))$$

Последовательно переиспользуемый вспомогательный файл можно получить, описав простой цикл **СОВМДОПВСПОМ*, использующий *ДОППАМ* в качестве подчиненного процесса:

$$\text{доп} : \text{ДОППАМ} // * \text{СОВМДОПВСПОМ}$$

Х8. Мультиплексное использование вспомогательных файлов

В двух предыдущих примерах вспомогательные файлы использовались только поочередно. Обычно же размер дополнительной памяти достаточно велик и допускает одновременное существование большого числа вспомогательных файлов, занимающих непересекающиеся подмножества доступных секторов. Дополнительная память, таким образом, может совместно использоваться неограниченным массивом вспомогательных файлов. Каждый вспомогательный файл занимает, когда ему требуется, сектор, выводя на него блок информации, и автоматически освобождает его после считывания этого блока. Использование дополнительной памяти основано на методе множественной пометки (разд. 2.6.4), где в качестве меток используются те же индексы (натуральные числа), что и при построении массива процессов-пользователей.

$$\text{ФАЙЛСИСТ} = \mathbb{N} : (\text{доп} : \text{ДОППАМ}) // \left(\bigparallel_{i \geq 0} i : \text{ФАЙЛПОЛЬЗ} \right),$$

где $\mathbb{N} = \{i \mid i \geq 0\}$.

Предполагается, что эта файловая система будет применяться в качестве подчиненного процесса, поочередно используемого произвольным числом пользователей:

$$\text{файлсист} : \text{ФАЙЛСИСТ} // \dots (\text{ПОЛЬЗ1} \parallel \text{ПОЛЬЗ2} \parallel \dots)$$

Новый вспомогательный файл может быть занят, использован и освобожден внутри каждого пользователя методом ди-

станционного подчинения:

*мойфайл :: файлсист// (... мойфайл .лев!v ...
... мойфайл .перемотка ... мойфайл .прав?х ...)*

что должно (помимо ограничений на ресурсы) иметь тот же самый эффект, что и простое подчинение личного вспомогательного файла (X5):

*(мойфайл : ВСПОМ//... мойфайл .лев!v ...
... мойфайл .перемотка ... мойфайл .прав?х ...)*

Структура этой файловой системы (X8) и способ ее применения представляют собой образец решения проблемы разделения ограниченного числа реальных ресурсов (секторов дополнительной памяти) между неизвестным числом пользователей. Пользователи не осуществляют прямого взаимодействия с ресурсами; для этого существует промежуточный *виртуальный ресурс (ФАЙЛПОЛЬЗ)*, который они описывают и используют как личный подчиненный процесс. Функция виртуального ресурса двояка:

(1) Он предоставляет пользователю предельно ясный интерфейс; в нашем примере *ФАЙЛПОЛЬЗ* склеивает в один непрерывный вспомогательный файл набор секторов, разбросанных в дополнительной памяти.

(2) Он гарантирует соблюдение надлежащей дисциплины доступа к реальному ресурсу; *ФАЙЛПОЛЬЗ*, например, гарантирует, что каждый пользователь производит считывание только из отведенных ему секторов и не может забыть освободить их при завершении работы с вспомогательным файлом.

Отметим, что благодаря п. (1) дисциплина п. (2) не является обременительной.

Парадигма реальных и виртуальных ресурсов очень важна при проектировании систем с разделением ресурсов. Ее математическое определение весьма сложно, поскольку для реализации необходимого динамического создания новых виртуальных процессов и новых каналов для взаимодействия с ними используется неограниченное множество натуральных чисел. При практической машинной реализации они будут представлены блоками управления, указателями на активационные записи и т. п. Конечно же, для эффективного использования парадигмы желательно забыть весь способ реализации. Но для тех, кто, прежде чем забыть, хочет глубже понять его, может быть полезным дальнейшее разъяснение примера X8.

Внутри процесса-пользователя с помощью механизма дистанционного подчинения создается вспомогательный файл *мойфайл :: файлисист //*(...*мойфайл.лев!v...*
...*мойфайл.перемотка...мойфайл.прав?x...*)

По определению дистанционного подчинения это эквивалентно

($\Pi_{i \geq 0}$ *файлисист.i.занят* →
файлисист.i.лев!v...файлисист.i.перемотка...
файлисист.i.прав?x...файлисист.i.свободен →
→ ПРОПУСК)

Таким образом, все взаимодействия *файлисист* с его пользователями начинаются с *файлисист.i ...*, где *i* — индекс конкретного экземпляра *ФАЙЛПОЛЬЗ*, занятого в конкретном случае конкретным пользователем. Более того, каждое его использование заключено между парой соответствующих друг другу сигналов

(*файлисист.i.занят* → *файлисист.i.свободен*)

Со стороны подчиненного процесса каждый виртуальный вспомогательный файл начинается с захвата пользователя и продолжается по схеме, описанной в **X6** и **X7**:

(*занят* → ...*лев?x...перемотка...прав!v...свободен...*).

Все прочие взаимодействия виртуального вспомогательного файла происходят с подчиненным процессом *ДОППАМ* и скрыты от пользователя. Каждый экземпляр виртуального вспомогательного файла сначала помечается отличным от других индексом *i*, а затем получает имя *файлисист*. Поэтому наблюдаемое извне поведение каждого экземпляра описывается как

(*файлисист.i.занят* →
файлисист.i.лев?x...файлисист.i.перемотка...
...*файлисист.i.прав!v...*
файлисист.i.свободен)

Это в точности соответствует схеме пользовательского взаимодействия, как она была описана в предыдущем абзаце. Пара сигналов *занят* — *свободен* гарантирует, что ни один пользователь не сможет попасть во вспомогательный файл, занятый другим пользователем.

Теперь обратимся к взаимодействиям, происходящим внутри *ФАЙЛСИСТ* между массивами виртуальных вспомогательных файлов и дополнительной памятью. Они скрыты от

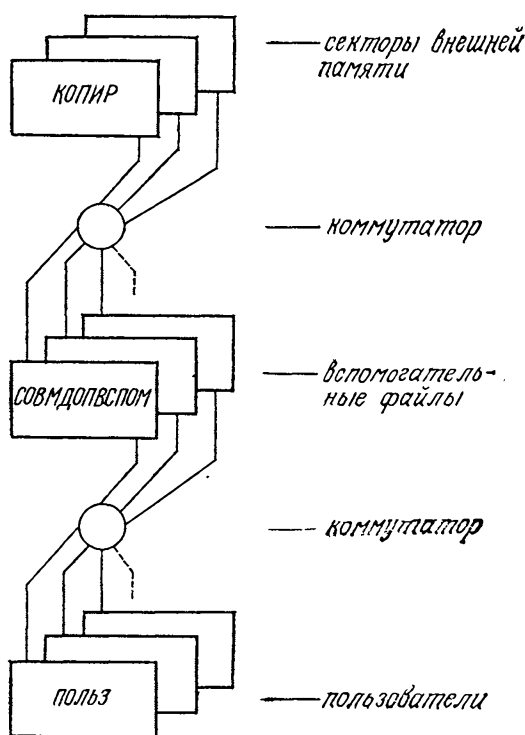


Рис. 6.2

пользователя и даже не имеют в своих именах составляющей файлист. Соответствующими событиями будут:

- $i.доп.j.лев.v$ — обозначает передачу блока v из i -го элемента массива вспомогательных файлов в j -й сектор дополнительной памяти,
 $i.доп.j.прав.v$ — обозначает взаимодействие в обратном направлении.

Каждый сектор дополнительной памяти ведет себя как КОПИР. Снабженный индексом j и именем $доп$, j -й сектор ведет себя как

$$\mu X. (доп.j.лев?x \rightarrow доп.j.прав!x \rightarrow X)$$

После множественной пометки натуральными числами он ведет себя как

$$\mu X. \left(\prod_{i \geq 0} i.доп.j.лев?x \rightarrow \left(\prod_{k \geq 0} k.доп.j.прав!x \rightarrow X \right) \right)$$

Теперь он в любой подходящий момент готов взаимодействовать с любым элементом массива виртуальных вспомогательных файлов. Каждый отдельный вспомогательный файл соблюдает порядок, при котором чтение производится только из тех секторов, на которые он сам последний раз записывал.

Натуральные числа i и j вводятся здесь просто для того, чтобы позволить вспомогательному файлу взаимодействовать с любым сектором на диске и обеспечить надежное взаимодействие файла с занимающим его пользователем. Таким образом, индексы служат своего рода математическим представлением координатного коммутатора, позволяющего в телефонной связи одному абоненту взаимодействовать с другим. Грубо это можно себе представить, как показано на рис. 6.2.

Если число секторов дополнительной памяти бесконечно, **ФАЙЛСИСТ** ведет себя в точности как аналогично построенный массив простых вспомогательных файлов:

$$\|_{i \geq 0} i : (\text{занят} \rightarrow (\text{ВСПОМ} \wedge (\text{свободен} \rightarrow \text{СТОП})))$$

Когда размер дополнительной памяти конечен, может случиться, что вся она окажется занятой в тот момент, когда все пользователи продолжают записывать в свои вспомогательные файлы, что может привести к тупиковой ситуации. На практике риск этого обычно сводится к минимуму путем задержки создания новых файлов, когда дополнительная память почти полна.

6.5. ОПЕРАЦИОННЫЕ СИСТЕМЫ

Пользователь большой ЭВМ вводит для исполнения программу в виде колоды перфокарт. Данные для каждой программы следуют непосредственно за ней. Задачей операционной системы с пакетным режимом является эффективное распределение машинных ресурсов между этими заданиями. Для этого мы требуем, чтобы каждая пользовательская программа выполнялась процессом **ЗАДАНИЕ**, который вводит по правому каналу *усп.прав* перфокарты с текстом программы, исполняет программу на данных, непосредственно следующих за ней в устройстве считывания перфокарт, и выводит результаты исполнения по каналу *аппу.лев*. О внутренней структуре процесса **ЗАДАНИЕ** нам знать ничего не надо — в старые времена это была мониторная система для Фортрана. Однако мы должны исходить из предположения, что **ЗАДАНИЕ** успешно завершится в разумных пределах

времени после начала. Поэтому определим алфавит процесса **ЗАДАНИЕ** как

$$\alpha\text{ЗАДАНИЕ} = \{usc.прав, ацпу.лев, \surd\}$$

Если **ААЦПУ** представляет аппаратную часть алфавитно-цифрового печатающего устройства, а **АУСП** — аппаратную часть устройства считывания перфокарт, то единственное задание единственного пользователя исполняется процессом

$$\text{ЗАДАНИЕ1} = (usc : \text{АУСП} // ацпу : \text{ААЦПУ} // \text{ЗАДАНИЕ})$$

От операционной системы, завершающейся после выполнения единственного задания, пользы, конечно, мало. Простейшим способом разделения одной ЭВМ между большим числом пользователей является выполнение заданий последовательно, одного за другим:

$$\text{ПАКЕТ0} = (usc : \text{АУСП} // ацпу : \text{ААЦПУ} // * \text{ЗАДАНИЕ})$$

Однако эта конструкция не учитывает некоторых административных деталей, таких, как отделение друг от друга файлов, выводимых различными заданиями, или отделение колоды с очередным заданием от предыдущего, чтобы текущее задание не могло считывать карты последующего. Для решения этой проблемы мы используем процесс **АЦПУ**, определенный в 6.2 **X4**, и процесс **УСП**, определенный ниже (**X1**):

$$\begin{aligned} \text{ЗАДАНИЯ} = &*((usc.занят \rightarrow ацпу.занят \rightarrow \text{ЗАДАНИЕ}); \\ &(usc.свободен \rightarrow ацпу.свободен \rightarrow \text{ПРОПУСК})) \\ \text{ПАКЕТ1} = &(usc : \text{УСП} // ацпу : \text{АЦПУ} // \text{ЗАДАНИЯ}) \end{aligned}$$

ПАКЕТ1 представляет собой абстрактное описание простейшей жизнеспособной операционной системы, позволяющей совместно использовать вычислительное устройство большому числу пользователей, чьи задания выполняются последовательно, одно за другим. Операционная система осуществляет смену последовательных заданий и защищает каждое задание от возможного влияния со стороны его предшественников.

Примеры

X1. Совместно используемое устройство считывания перфокарт

Перед началом каждой колоды задания в устройство считывания карт закладывается специальная карта-разделитель. Устройство считывает все карты одной колоды задания, после чего освобождается. Если пользователь попытается читать за пределами разделителя, ввод с устройства считы-

вания производиться не будет, а будут появляться лишь дальнейшие копии разделительной карты. Если пользователь не дочитал колоду до разделительной карты, то остаток колоды сбрасывается. Лишние разделители игнорируются. Ввод с аппаратуры производится с помощью команды $a?x$.

Совместно используемое устройство считывания перфокарт должно считывать на одну карту вперед, и поэтому значение буферизованной карты используется в качестве индекса:

$$УСП = a?x \rightarrow \text{if } x = \text{разделитель then } УСП \text{ else (занят} \rightarrow УСП_x)$$

где $УСП_x = (\text{прав!}x \rightarrow a?y \rightarrow$

$$\begin{aligned} &\text{if } y \neq \text{разделитель then } УСП_y \\ &\text{else } \mu X. (\text{прав!разделитель} \rightarrow X \mid \text{свободен} \rightarrow \\ &\hspace{15em} \rightarrow УСП) \end{aligned}$$

$$\mid \text{свободен} \rightarrow$$

$$\mu X. (a?y \rightarrow \text{if } y = \text{разделитель then } УСП \text{ else } X))$$

Пропустив начальную подпоследовательность разделителей, этот процесс захватывает пользователя и копирует по правому каналу последовательность неразделительных карт, которые он считывает с устройства ввода. Обнаружив карту-разделитель, он копирует ее до тех пор, пока пользователь не освободит ресурс. Но если пользователь освобождает ресурс прежде, чем достигнут разделитель, оставшиеся в колоде карты следует сбросить, читая и игнорируя их.

Операционная система *ПАКЕТ1* является логически полной. Но с ростом быстродействия аппаратной части центрального процессора возможности последнего начинают опережать возможности устройств печати и считывания в обеспечении ввода, передачи и выдачи заданий. В целях приведения в соответствие скорости ввода, вывода и обработки необходимо использовать два или более считывающих и печатающих устройств. Поскольку задания обрабатываются процессором по одному, дополнительные устройства считывания должны быть заняты считыванием колоды следующего задания (или заданий), а дополнительные устройства печати должны быть заняты печатью выходных файлов предыдущего задания (или заданий). Поэтому каждый входной файл в период между реальным вводом с устройства считывания перфокарт и потреблением его процессом *ЗАДАНИЕ* временно должен храниться во вспомогательном файле; в свою очередь каждый выходной файл в период между выработкой его строк процессом *ЗАДАНИЕ* и их реальной выдачей печатающим устройством должен быть аналогичным образом

буферизован. Этот метод называется *спулингом*, или *подкачкой/откачкой*.

Общая структура операционной системы со спулингом

$ОПСИСТ1 = ввсис : ПОДКАЧКА // вывсис : ОТКАЧКА //$
 $// ПАКЕТ$

Процесс *ПАКЕТ* похож на *ПАКЕТ* с той разницей, что для работы с ожидающими входными файлами и выходными файлами, предназначенными для последующей печати, он использует механизм дистанционного подчинения:

$ПАКЕТ = *(усп :: ввсис // ацпу :: вывсис // ЗАДАНИЕ)$

Процессы подкачки и откачки определены в следующих двух примерах.

Х2. Вывод с откачкой

Отдельное виртуальное устройство печати использует для хранения блоков, выведенных процессом-пользователем, временный вспомогательный файл (6.4 Х5). Когда процесс-пользователь сигнализирует об освобождении виртуального печатающего устройства, для печати временного файла захватывается реальное печатающее устройство (6.4 Х3):

$ВАЦПУ = (врем : ВСПОМ //$
 $\mu X.лев?x \rightarrow врем.лев!x \rightarrow X$
 $| свободен \rightarrow врем.перемотка \rightarrow$
 $(реальн :: ацпу //$
 $\mu Y(врем.прав?y \rightarrow реальн.лев!y \rightarrow Y$
 $| врем.пуст \rightarrow ПРОПУСК)))$

Требуемый неограниченный массив виртуальных печатающих устройств определяется как

$МВАЦПУ = \bigparallel_{i \geq 0} i : (занят \rightarrow ВАЦПУ)$

Поскольку мы хотим использовать реальные печатающие устройства (6.4 Х3) только в режиме откачки, мы можем описать их как локальные по отношению к системе подкачки-откачки, используя множественную пометку для их совместного использования всеми элементами массива *МВАЦПУ*, как в 6.4 Х8:

$ОТКАЧКА = (N : (ацпу : АЦПУ2 // МВАЦПУ)$

Х3. Ввод с подкачкой

Подкачка очень напоминает откачку, только для каждого отдельного задания вначале захватывается реальное устройство считывания перфокарт, а в конце ввода оно освобождает

ется; затем задание передается для выполнения процессу-пользователю, которому и выводится содержимое перфокарт:

$$\begin{aligned} \text{ВУСП} = & \text{врем} : \text{ВСПОМ} // \\ & (\text{реальн} :: \text{усп} // \\ & (\mu X. \text{реальн} . \text{прав?} x \rightarrow \text{if } x = \text{разделитель} \\ & \quad \text{then ПРОПУСК} \\ & \quad \text{else } \text{врем} . \text{лев!} x \rightarrow X)); \\ & (\text{врем} . \text{перемотка} \rightarrow \text{занят} \rightarrow \\ & \mu Y. (\text{врем} . \text{прав?} x \rightarrow \text{прав!} x \rightarrow Y \\ & \quad | \text{врем} . \text{пуст} \rightarrow \text{прав!} \text{разделитель} \rightarrow \\ & \quad \rightarrow Y)) \wedge (\text{свободен} \rightarrow \text{ПРОПУСК})) \\ \text{ПОДКАЧКА} = & (\mathbb{N} : \text{усп} : (0 : \text{УСП} \parallel 1 : \text{УСП})) // \left(\bigparallel_{i \geq 0} i : \text{ВУСП} \right) \end{aligned}$$

Итак, устройства откочки и подкачки предоставляют в пользование процессу ЗАДАНИЕ неограниченное количество виртуальных устройств считывания и виртуальных печатающих устройств. В результате становится возможным параллельное исполнение двух или более процессов ЗАДАНИЕ, совместно использующих эти виртуальные ресурсы. Поскольку никаких взаимодействий между заданиями не предполагается, подходящим способом совместного использования будет простое чередование. Эта техника известна под названием *мультипрограммирования*; если же используется более одного реального процессора, ее называют *мультипроцессорной обработкой*. Однако логический эффект мультипрограммирования и мультипроцессорной обработки один и тот же; и, разумеется, определенная ниже операционная система имеет ту же логическую спецификацию, что и ОПСИСТ1, описанная выше:

$$\text{ОПСИСТ} = \text{ввсист} : \text{ПОДКАЧКА} // \text{вывсист} : \text{ОТКАЧКА} // \text{ПАКЕТ4},$$

где $\text{ПАКЕТ4} = \left(\bigparallel_{i < 4} \text{ПАКЕТ} \right)$

Как видите, в математической записи переход к мультипрограммированию произошел на редкость просто: в реальной же действительности он сопровождался преодолением мучительных трудностей.

При описании процесса ВАЦПУ внутри процесса ОТКАЧКА (Х2), подчиненный процесс ВСПОМ использовался для хранения строк, выработанных процессом ЗАДАНИЕ, до тех пор, пока они не были выведены реальным устройством печати. Обычно выходные файлы слишком велики, чтобы храниться в основной памяти машины, и должны поэтому храниться в дополнительной памяти, как было показано в

6.4 X8. Все временные файлы должны совместно использовать общую дополнительную память, поэтому подчиненный процесс

врем : ВСПОМ // ...

внутри ВАЦПУ надо заменить на описание дистанционно подчиненного процесса

врем :: файлсист // ...

а затем объявить файловую систему (6.4 X8) подчиненным процессом процесса откатки:

файлсист : ФАЙЛСИСТ // ОТКАЧКА)

Если имеет значение размер входной колоды, то аналогичные изменения надо внести и в процесс ПОДКАЧКА. Если для этого в нашем распоряжении есть отдельное внешнее запоминающее устройство, эти изменения сделать просто. Если нет, нам придется организовать совместное использование одного внешнего запоминающего устройства временными файлами как процесса откатки, так и процесса подкачки. Это означает, что ФАЙЛСИСТ должен быть объявлен подчиненным процессом, разделяемым между процессами подкачки и откатки с помощью множественной пометки. Это же влечет за собой изменение структуры системы. Мы произведем эту перестройку методом сверху вниз, попытавшись использовать повторно как можно больше ранее определенных модулей.

Операционная система состоит из системы мультипрограммирования в пакетном режиме ПАКЕТ4 и системы ввода-вывода, которая служит подчиненным процессом:

ОС = СИСТВВ // ПАКЕТ4

В системе ввода-вывода СИСТВВ файловая система совместно используется процессами откатки и подкачки:

*СИСТВВ = СОВМ : (файлсист : ФАЙЛСИСТ)
// (ацпу : ОТКАЧКА' || усн : ПОДКАЧКА')*

где $СОВМ = \{ацпу.i | i \geq 0\} \cup \{усн.i | i \geq 0\}$, а ОТКАЧКА' и ПОДКАЧКА' — те же, что и в X2 и X3, с той разницей, что *врем : ВСПОМ* заменено в них на эквивалентное дистанционное подчинение *врем :: файлсист*.

В конструкции описанных в этой главе четырех операционных систем (ПАКЕТ1, ОПСИСТ1, ОПСИСТ и ОС) сле-

дует подчеркнуть их модульный характер. Это означает, что в более поздних вариантах системы мы имели возможность использовать крупные части более ранних ее вариантов. Что еще важнее, конструкция каждой детали заключена внутри одного-двух модулей системы. Следовательно, если некоторая деталь требует модификации, очень легко определить подлежащий изменению модуль, и все изменения будут ограничены этим модулем. Среди легко изменяемых параметров назовем число печатающих устройств, число устройств считывания перфокарт, число параллельных пакетов. Но не все изменения можно сделать так просто: изменение значения карты-разделителя затронет три модуля — *УСП(Х1)*, *ПОДКАЧКА(Х3)* и *ЗАДАНИЕ*.

Кроме того, существует целый ряд полезных усовершенствований системы, внесение которых потребовало бы значительного изменения ее структуры:

(1) Задания пользователей должны иметь доступ к файловой системе и кратным виртуальным устройствам ввода и вывода.

(2) Пользовательские файлы должны постоянно храниться между заданиями, которым они подчинены.

(3) Для быстрого восстановления после сбоя может потребоваться механизм контрольных точек.

(4) Если существует журнал введенных, но еще не завершенных заданий, необходим некоторый метод управления порядком запуска ожидающих заданий. Более полно этот пункт будет рассмотрен в следующем разделе.

Одной из проблем, возникающих при внесении этих усовершенствований, является невозможность совместного использования ресурсов подчиненным и главным процессом там, где неприменим метод множественной пометки. Возможно, требуется новое определение подчинения, в котором алфавит подчиненного процесса не был бы подмножеством основного процесса. Но это — тема для дальнейшего исследования.

6.6. ПЛАНИРОВАНИЕ РЕСУРСОВ

Когда ограниченное число ресурсов разделено между большим числом потенциальных пользователей, всегда существует возможность того, что некоторым пользователям, стремящимся занять ресурс, приходится ждать, пока его освободит другой процесс. Если к моменту освобождения ресурса его хотят занять два или более процесса, выбор того, который из ожидающих процессов получит ресурс, во всех

приводившихся примерах был недетерминированным. Само по себе это большого значения не имеет, но предположим, что к тому моменту, когда ресурс снова освободится, к множеству ожидающих присоединится еще один процесс. Поскольку выбор между ожидающими процессами по-прежнему недетерминирован, может случиться, что повезет именно вновь присоединившемуся процессу. Если ресурс сильно загружен, так может случиться снова и снова. В результате может оказаться, что некоторые процессы будут откладываться бесконечно или по крайней мере в течение полностью неопределенного времени. С этой проблемой, называемой бесконечным перехватом, мы уже знакомы (разд. 2.5.5).

Одним из решений проблемы является обеспечение того, чтобы все ресурсы были несильно загружены. Этого можно достигнуть либо введением дополнительных ресурсов, либо установлением высокой платы за предоставляемые услуги. Фактически это единственно приемлемые решения в случае постоянно загруженного ресурса. К сожалению, даже в среднем несильно загруженный ресурс достаточно часто оказывается сильно загруженным в течение длительных периодов (в часы пик). Иногда проблему удастся сгладить введением дифференцированного тарифа в попытке регулирования спроса, но это не всегда помогает и даже не всегда удается. В течение таких пиков задержки процесса-пользователя в среднем неизбежны. Важно лишь следить за сообразностью и предсказуемостью таких задержек — вы, несомненно, предпочтете знать, что вас обслужат в течение часа, чем гадать, сколько еще придется ждать — одну минуту или целые сутки.

Задача распределения ресурса между ожидающими пользователями известна как *планирование ресурсов*. Для успешного планирования необходимо знать, какие процессы в текущий момент ожидают получения ресурса. По этой причине получение ресурса отныне нельзя рассматривать как одно элементарное событие. Его необходимо разбить на два события:

пожалуйста, осуществляющее запрос ресурса,
спасибо, сопровождающее реальное получение ресурса.

Период между *пожалуйста* и *спасибо* для каждого процесса является временем, в течение которого он ждет ресурс. Чтобы различать ожидающие процессы, каждое вхождение события *пожалуйста*, *спасибо* и *свободен* помечено отличным от других натуральным индексом. При каждом запросе ресурса процесс получает номер с помощью той же конструк-

ции, что и та, которая лежит в основе дистанционного подчинения (6.4 X3):

$\Pi_{i \geq 0} (\text{рес. } i. \text{пожалуйста}; \text{рес. } i. \text{спасибо}; \dots; \text{рес. } i. \text{свободен} \rightarrow \rightarrow \text{ПРОПУСК})$

Простым и эффективным способом планирования ресурса является назначение его процессу, ожидавшему дольше всех. Такая политика называется «*первым пришел — первым обслужен*» (FCFS) или «*первым пришел — первым ушел*» (FIFO) и представляет собой принцип очереди, соблюдаемый, к примеру, пассажирами на автобусной остановке.

В заведении же типа поликлиники, где посетители не могут или не хотят выстраиваться в очередь, для достижения того же результата действует другой механизм. Регистратура выдает талоны со строго возрастающими последовательными номерами. При входе в поликлинику посетитель берет талон. Когда врач освободился, он вызывает посетителя, имеющего талон с наименьшим номером, но еще не принятого. Этот алгоритм, называемый алгоритмом поликлиники, более строго описан ниже. Мы будем предполагать, что одновременно могут обслуживаться до R посетителей.

Пример

X1. Алгоритм поликлиники

Нам потребуются три счетчика:

p — посетители, сказавшие *пожалуйста*,
 t — посетители, сказавшие *спасибо*,
 r — посетители, освободившие свои ресурсы.

Очевидно, что в любой момент времени $r \leq t \leq p$. Кроме того, p всегда будет номером, который получает очередной посетитель, приходящий в поликлинику, а t — номером очередного обслуживаемого посетителя; далее, $p - t$ будет числом ожидающих посетителей, а $R + r - t$ — числом ожидающих врачей. Вначале значения всех счетчиков равны нулю и могут быть вновь положены равными нулю в любой момент, когда их значения совпадают — например, вечером, после ухода последнего посетителя.

Одной из основных задач алгоритма является обеспечение того, чтобы никогда не было одновременно свободного ресурса и ждущего посетителя; как только возникает такая

ситуация, следующим событием должно стать *спасибо* посетителя, получающего ресурс.

$$\begin{aligned}
 & \text{ПОЛИКЛИНИКА} = B_{0,0,0} \\
 & B_{p,t,r} = \text{if } 0 < r = t = p \text{ then ПОЛИКЛИНИКА} \\
 & \quad \text{else if } R + r - t > 0 \ \& \ p - t > 0 \\
 & \quad \quad \text{then } t.\text{спасибо} \rightarrow B_{p,t+1,r} \\
 & \quad \text{else } (p.\text{пожалуйста} \rightarrow B_{p+1,t,r} \\
 & \quad \quad | \left(\bigwedge_{i < t} i.\text{свободен} \rightarrow B_{p,t,r+1} \right))
 \end{aligned}$$

Алгоритм поликлиники принадлежит Лесли Лэмпорту ¹⁾.

¹⁾ В оригинале — The bakery algorithm, т.е. «алгоритм булочной», и именно так он называется в книге Дейтела (Операционные системы. — М.: Мир, 1987). Здесь мы сочли целесообразным прибегнуть к более близкой нашему читателю аналогии с поликлиникой. — Прим. перев.

Глава 7. Обсуждение

7.1. ВВЕДЕНИЕ

Основной целью моих исследований взаимодействующих процессов был поиск как можно более простой математической теории, обладающей следующими желательными свойствами:

(1) Она должна описывать широкий круг интересных машинных применений по управлению процессами и дискретному моделированию событий, начиная с торговых автоматов и кончая операционными системами с разделяемыми ресурсами.

(2) Она должна обеспечивать эффективную реализацию разнообразной удобной и современной машинной архитектуры от систем разделения времени до мультипроцессорных устройств и сетей взаимодействующих микропроцессоров.

(3) Она должна прояснять программисту все вопросы в деле спецификации, разработки, реализации, верификации и анализа сложных вычислительных систем.

Конечно, невозможно потребовать, чтобы все эти цели достигались оптимальным образом. Всегда остается шанс, что радикально отличный подход или какие-то значительные изменения в определениях приведут к большему успеху в достижении одной или более из перечисленных целей. В этой главе мы приступаем к обсуждению некоторых исследованных мною и другими альтернативных подходов, а также соображений, по которым я от них отказался. Кроме того, здесь я использую возможность воздать должное влиянию оригинальных разработок других авторов в этой области. И наконец, я надеюсь вдохновить читателя на дальнейшие исследования как в области основ предмета, так и в области расширения его практических применений.

7.2. ОБЩАЯ ПАМЯТЬ

Впервые потребность в программировании параллельных вычислений на одном вычислительном устройстве возникла в 1960-х годах как естественное следствие современного развития вычислительной архитектуры и операционных систем.

В то время вычислительные мощности были недостаточными и дорогостоящими, и задержки процессора при взаимодействии с медленными периферийными устройствами или еще более медленным человеком-оператором считались большим расточительством. Как следствие появились более дешевые процессоры специального назначения (каналы), отдельно осуществляющие ввод-вывод, освобождая при этом центральный процессор для выполнения других заданий. Чтобы дорогостоящий центральный процессор не простаивал, мультипрограммная операционная система обеспечивала, чтобы в оперативной памяти машины находилось целиком несколько программ и чтобы в любой момент несколько программ использовали процессоры ввода-вывода, в то время как другая программа использовала центральный процессор. По окончании операции ввода-вывода возникает прерывание, позволяющее операционной системе решить, какой задаче предоставить услуги центрального процессора.

Вышеописанная схема предполагает, что центральный процессор и все каналы связаны со всей оперативной памятью машины и обращения различных процессоров к различным участкам памяти чередуются. При этом каждая исполняемая программа обычно является замкнутым заданием, заказанным отдельным пользователем и полностью независимым от остальных заданий.

По этой причине при разработке программных и технических средств большие усилия расходовались на разбиение памяти на непересекающиеся сегменты, чтобы гарантировать, что ни одна программа не будет влиять на память других программ. С появлением возможности снабжать одну машину несколькими независимыми центральными процессорами пропускная способность машины возрастает; и если исходная операционная система хорошо структурирована, это повышение достигается лишь незначительными изменениями команд операционной системы и еще меньшими изменениями в программах исполняемых заданий.

Недостатками разделения общего вычислительного устройства между несколькими различными заданиями являются:

- (1) Количество требуемой памяти возрастает в линейной зависимости от числа исполняемых заданий.
- (2) Возрастает время ожидания пользователем результатов своего задания, за исключением заданий с наивысшим приоритетом.

Поэтому идея позволить отдельному заданию воспользоваться аппаратно реализованными средствами параллелизма.

инициализируя несколько параллельных процессов внутри одной и той же области памяти, выделенной одной программе, представляется очень соблазнительной.

7.2.1. Многопоточная обработка

Первые предложения в этом направлении были основаны на идее оператора перехода (команда **go to**). Если *L* — метка некоторого места в программе, то команда

fork L

передает управление на метку *L*, а также и на следующую команду в тексте программы. В результате создается эффект, что с этого момента два процессора одновременно исполняют одну и ту же программу; каждый из них независимо обрабатывает свою последовательность команд. Поскольку каждая такая последовательность обработки может снова разветвиться, эта техника получила название *многопоточной обработки*.

Введя способ разбиения одного процесса на два, мы нуждаемся и в способе слияния двух процессов в один. Проще всего ввести команду **join**, которая может выполняться только при одновременном исполнении ее *двумя* процессами. Первый достигший этой команды процесс должен ждать, когда ее достигнет другой. После этого уже только один процесс продолжает исполнение последующих команд.

Во всей своей полноте техника многопоточной обработки чрезвычайно сложна и чревата ошибками и может быть рекомендована лишь в самых небольших программах. В свое оправдание мы можем сказать, что она была изобретена задолго до начала эпохи структурированного программирования, еще в те времена, когда даже Фортран считался языком высокого уровня!

Разновидность команды ветвления до сих пор используется в операционной системе UNIX™. При этом ветвление не подразумевает переход по метке. Его эффект заключается во взятии совершенно новой копии всей памяти программы и передачи этой копии новому процессу. Как исходный, так и новый процессы продолжают исполнение с команды, следующей за командой ветвления. У каждого процесса есть средство определить, является ли он *порождающим* (отец) или *порождаемым* (сын). Выделение процессам непересекающихся участков памяти снимает основные трудности и опасности многопоточной обработки, но может быть неэффективным как по времени, так и по объему памяти. Это означает, что параллелизм допустим только на самом внешнем (самом

глобальном) уровне задания, а использование его в мелком масштабе затруднительно.

7.2.2 **cobegin...coend**

Решение проблемы многопоточной обработки было предложено Э. Дейкстрой: надо убедиться, что после разветвления два процесса исполняют полностью различные блоки программы, и передачи управления между ними невозможны. Если P и Q — такие блоки, то действие составной команды

cobegin P ; Q coend

заключается в одновременном запуске P и Q и их параллельном исполнении до тех пор, пока оба они не завершатся. Последующие же команды исполняются уже только одним процессором. Эта структурная команда может быть реализована с помощью неструктурных команд **fork** и **join** и меток L и J :

fork L ; P ; go to J ; L : Q ; J :join

Обобщение на случай более чем двух составляющих процессов очевидно:

cobegin P ; Q ; ...; R coend

При такой структурированной форме записи намного понятнее, что должно произойти, особенно если в различных блоках используются различные переменные (ограничение, выполнение или контроль за которыми может взять на себя транслятор с языка высокого уровня). В этом случае процессы называются *непересекающимися* и (в случае отсутствия взаимодействия между ними) параллельное исполнение P и Q имеет в точности тот же результат, что и их последовательное исполнение в любом порядке:

begin P ; Q end = begin Q ; P end = cobegin P ; Q coend

Более того, методы доказательства корректности параллельной композиции могут оказаться даже проще, чем в последовательном случае. Вот почему в этой книге в основе конструкции параллелизма лежит предложение Дейкстры. Разница в основном только в обозначениях; я ввел для разделения процессов оператор **||**, чтобы избежать путаницы с последовательной композицией, а это позволяет вместо громоздких **cobegin ... coend** использовать простые скобки.

7.2.3. Условные критические участки

Ограничение, состоящее в том, что параллельные процессы не могут иметь общих переменных, приводит к тому, что у них нет возможности взаимодействовать друг с другом —

ограничение, серьезно сокращающее потенциальные возможности параллелизма.

По прочтении этой книги очевидным решением покажется введение (имитированных) каналов ввода-вывода; но в прежние времена обычным методом (подсказываемым архитектурой вычислительной машины) было взаимодействие через совместное использование оперативной памяти несколькими параллельными процессами. Безопасный способ, основанный на критических участках (разд. 6.3), защищенных семафорами взаимного исключения, был указан Дейкстрой. Позже я предложил, чтобы этот метод был формализован в терминах языка высокого уровня. Группа переменных, значения которых должны изменяться в критическом интервале несколькими процессами, должна быть описана как разделяемый ресурс, например:

shared *n*: integer;

shared позиция: record *x*, *y*; real end

Каждый критический участок, в котором эта переменная изменяется, начинается с члена **with**, за которым следует имя переменной:

with *n* **do** *n* := *n* + 1;

with позиция **do begin** *x* := *x* + дельта*x*; *y* := *y* + дельта*y*
end

Преимущество этой записи в том, что транслятор автоматически вводит необходимые семафоры и окружает каждый критический участок необходимыми *P*- и *V*-операциями. Более того, во время трансляции он может делать проверку, что доступ и изменение значений общих переменных осуществляются только изнутри критического интервала, защищенного соответствующим семафором.

Взаимодействие процессов, совместно использующих память, может потребовать синхронизации другого рода. Предположим, например, что один процесс изменяет некоторую переменную с целью, чтобы другой процесс считывал ее новое значение. Второй процесс не должен считывать значения переменной до тех пор, пока оно не будет изменено. Аналогично, первый процесс не должен изменять значение переменной до тех пор, пока все остальные процессы не считают ее предыдущие значения.

Для решения этой проблемы предложено удобное средство, называемое условным критическим участком. Он имеет вид

with *общперем* **when** *условие* **do** критический участок

При входе в критический участок проверяется значение условия. Если оно истинно, критический участок выполняется как обычно, но если условие ложно, данный вход в критический участок задерживается, чтобы позволить другим процессам войти в свои критические участки и изменить общую переменную. По завершении каждого такого изменения происходит перепроверка условия. Если оно стало истинным, отложенному процессу позволяют продолжать исполнение своего критического участка; в противном случае процесс вновь откладывается. Если можно запустить более чем один из приостановленных процессов, выбор между ними произвольный.

Для решения проблемы обновления и чтения сообщения несколькими процессами опишем как часть ресурса целочисленную переменную для подсчета числа процессов, которые должны прочесть сообщение, прежде чем оно вновь изменится:

```
shared сообщ: record счет: integer; содерж:...end;  
сообщ.счет := 0;
```

Процесс, изменяющий сообщение, содержит критический участок

```
with сообщ when счет = 0 do  
  begin содерж := ...;  
  ...;  
  счет := число читателей  
end
```

Каждый читающий процесс содержит критический участок

```
with сообщ when счет > 0 do  
  begin мой экз := содерж; счет := счет - 1 end
```

Условные критические участки можно реализовать средствами семафоров. По сравнению с непосредственным использованием программистом синхронизирующих семафоров накладные расходы условных критических участков могут быть достаточно высоки, поскольку условия всех процессов, ожидающих входа в критический участок, должны перепроверяться при каждом выходе из него. К счастью, более частой перепроверки этих условий не требуется, поскольку ограничения на доступ к общим переменным гарантируют, что значение проверенного ожидающим процессом условия может измениться, только когда изменится значение самой общей переменной. Все остальные переменные в условии должны быть локальными по отношению к ожидающему процессу, который, очевидно, не может изменять их, находясь в состоянии ожидания.

7.2.4. Мониторы

Своим возникновением и развитием мониторы обязаны понятию класса в языке Симула 67, представляющим в свою очередь обобщение концепции процедуры в Алголе 60. Основной идеей является то, что все осмысленные операции над данными (включая их инициализацию) должны быть собраны вместе с описанием структуры и типа самих данных; активизация этих операций должна происходить при вызове процедуры всякий раз, когда этого требуют процессы, совместно использующие данные. Важной характеристикой монитора является то, что одновременно может быть активным только одно из его процедурных тел; даже когда два процесса одновременно делают вызов процедуры (одной и той же или двух различных), один из вызовов откладывается до завершения другого. Таким образом, тела процедур ведут себя как критические участки, защищенные одним и тем же семафором.

Приведем пример очень простого монитора, ведущего себя как счетчикова переменная. В обозначениях языка PASCAL PLUS он имеет вид

```

1 monitor счет;
2 var n : integer;
3 procedure *вверх; begin n := n + 1 end;
4 procedure *вниз; when n > 0 do begin n := n - 1 end;
5 function *приземл : Boolean; begin приземл := (n = 0) end;
6 begin n := 0;
7   ...;
8   if n ≠ 0 then print(n)
9 end
```

- Строка 1 описывает монитор с именем *счет*.
 2 описывает локальную для монитора общую переменную *n*. Она доступна только внутри самого монитора.
 3 } описывают три процедуры и их тела. Звездочки
 4 } обеспечивают вызов этих процедур из программы,
 5 } использующей монитор.
 6 Здесь начинается исполнение монитора.
 7 Три жирные точки обозначают *внутреннее* предложение, соответствующее блоку, который будет использовать монитор.
 8 При выходе из использующего блока печатается конечное значение *n* (если оно ненулевое).

Новый экземпляр этого монитора может быть описан локальным для блока *P*:

```
instance ракета : счет; P
```


Внутри блока P можно вызывать помеченные звездочками процедуры, используя команды

ракета.вверх; ... ракетa.вниз; ...; if ракетa.приземл then ...

Непомеченная же процедура или такая переменная, как n , недостижимы из P , а соблюдение этого ограничения обеспечивается транслятором. Свойственное мониторам взаимное исключение позволяет вызывать процедуру монитора любому числу процессов внутри P без взаимного влияния при изменении n . Заметим, что попытка вызвать *ракета.вниз* при $n = 0$ будет отложена до тех пор, пока некоторый другой процесс внутри P не вызовет *ракета.вверх*. Это гарантирует, что значение n никогда не станет отрицательным.

Действие, производимое объявлением экземпляра монитора, поясняется с помощью правила, напоминающего копирование вызова процедуры в Алголе 60. Сначала берется копия текста монитора; на место многоточия в тексте подставляется копия использующего блока P , и все локальные имена в мониторе «расклеиваются», т. е. получают приставку в виде имени этого экземпляра, как показано ниже:

```

ракета.n : integer;
procedure ракета.вверх; begin ракета.n := ракета.n + 1 end;
procedure ракета.вниз;
  when ракета.n > 0 do begin ракета.n := ракета.n - 1 end;
function ракета.приземл : Boolean;
  begin ракета.приземл := (ракета.n = 0) end
begin ракета.n := 0;
  P;
  if ракета.n ≠ 0 then print(ракета.n)
end

```

Заметим, что правило копирования не позволяет процессу-пользователю забыть инициализировать значение n или, в случае необходимости, забыть напечатать его конечное значение.

Неэффективность повторяющейся проверки входных условий привела к появлению мониторов с более сложной схемой явного ожидания и явной подачей сигнала о возобновлении ожидающего процесса. Эти схемы даже позволяют приостанавливаться процедурному вызову в процессе его исполнения под воздействием автоматически возникающего исключения до того момента, когда некоторый последующий вызов процедуры другим процессом подаст сигнал о возобновлении приостановленного процесса. Таким путем можно эффективно реализовать множество оригинальных способов планирова-

ния; однако сейчас я думаю, что это едва ли окупает возникающие при этом сложности.

7.2.5. Вложенные мониторы

Экземпляр монитора может использоваться как семафор для защиты единственного ресурса, например печатающего устройства, которое не должно одновременно использоваться более чем одним процессом. Такой монитор можно описать как

```
monitor один ресурс;
var своб : Boolean;
procedure *занят; when своб do своб := false;
procedure *свободен; begin своб := true end;
begin своб := true; ... end
```

Однако этот монитор не может защитить от процесса, который использует ресурс, не занимая его; процесс, забывший освободить ресурс после пользования им, также сводит эту защиту на нет. Обе эти опасности можно предотвратить, введя конструкцию наподобие виртуального ресурса (6.4 X4). Она имеет вид монитора, локально описанного внутри монитора реального ресурса, приведенного выше. Чтобы имя виртуального ресурса было доступно для описания процессам-пользователям, оно помечается звездочкой. С имен же **занят* и **свободен* звездочки удаляются, чтобы их можно было использовать только внутри монитора виртуального ресурса и другие процессы не могли бы их неправильно употребить.

```
monitor один ресурс;
var своб : Boolean;
procedure занят;
  when своб do своб := false;
procedure свободен;
  begin своб := true; end
monitor *виртуальн;
  procedure *использ(l : line); begin ... end;
  begin занят; ...; свободен end
begin своб := true; ...; end
```

Экземпляр этого монитора описывается как

```
instance аппусист : один ресурс; P
```

Блок внутри *P*, требующий вывести файл на печатающее устройство, выглядит как

```
instance мой : аппусист.виртуальн;
  begin ... мой.использ(l1); ... мой.использ(l2); ... end
```

Необходимые действия по занятию и освобождению печатающего устройства вставляются виртуальным монитором автоматически до и после этого использующего блока так, чтобы воспрепятствовать антиобщественному использованию этого устройства. В принципе возможно, чтобы использующий блок состоял из параллельных процессов, каждый из которых использует *мой* экземпляр виртуального монитора, но здесь такая цель, вероятно, не ставилась. Монитор, предназначенный для использования единственным процессом, называется в PASCAL PLUS *конвертом* и может быть более эффективно реализован без использования исключения или синхронизации; транслятор же следит за тем, чтобы такой монитор случайно не использовали как общий.

Значение этих экземпляров описаний можно вычислить последовательным применением правила копирования, в результате чего мы получим:

```

var ацнусист.своб : Boolean;
procedure ацнусист.занят;
  when ацнусист.своб do ацнусист.своб := false;
procedure ацнусист.свободен;
  begin ацнусист.своб := true end;
begin ацнусист.своб := true
  .
  .
  .
  begin
  procedure мой.ацнусист.использ(l : line); begin...end;
    ацнусист.занят;
    ...мой.ацнусист.использ(l1);
    ...мой.ацнусист.использ(l2);
    ацнусист.свободен;
  end;
  .
  .
  .
end

```

Явное копирование приводится здесь только в качестве пояснения для начинающих; более опытный программист никогда не захочет ни увидеть, ни даже думать о такой неэкономной версии.

Эти обозначения использовались в 1975 г. для описания операционной системы, аналогичной нашей из разд. 6.5; позже они были реализованы в языке PASCAL PLUS. Сочетанием пометки звездочкой и вложенности можно достигнуть весьма

изошренных эффектов, однако обозначения, основанные на языках Паскаль и Симула, выглядят довольно неуклюже, а рассуждения в терминах подстановок и переименований воспринимаются не лучшим образом. Критика Э. Дейкстрой именно этих особенностей и навела меня впервые на мысль о создании теории взаимодействующих последовательных процессов.

Однако на примере конструкций из разд. 6.5 мы уже могли убедиться, что сложности при управлении совместным использованием возникают независимо от того, выражено ли оно в рамках концепции взаимодействующих процессов, или же в семантике правил копирования и процедурных вызовов языка PASCAL PLUS. Выбор между этими языками — это, отчасти, дело вкуса или, возможно, эффективности. При реализации операционной системы на машине с общей оперативной памятью предпочтение, вероятно, следует отдать языку PASCAL PLUS.

7.2.6. Ада™

Предлагаемые в Аде средства параллельного программирования представляют собой слияние понятия дистанционного вызова процедуры в языке PASCAL PLUS с менее структурированной формой взаимодействия посредством ввода и вывода. Процессы называются задачами и взаимодействуют с помощью операторов вызова *call* (напоминающих вызовы процедуры с входными и выходными параметрами) и операторов приема *accept* (синтаксически и по своему действию напоминающих описания процедур). Типичный пример оператора приема:

```
accept put(V : in integer; PREV : out integer) do
  PREV := K; K := V end
```

Соответствующий вызов может иметь вид

```
put(37, X)
```

Идентификатор *put* называется *именем входа*.

Одноименные операторы вызова и приема в различных задачах выполняются, когда оба процесса одновременно готовы их исполнить. Эффект такого исполнения будет следующим:

- (1) Входные параметры копируются из вызова в принимающий процесс.
- (2) Исполняется тело оператора приема.
- (3) Значения выходных параметров копируются обратно в оператор вызова.

(4) Обе задачи продолжают исполнение своих очередных операторов.

Исполнение тела оператора приема называется *рандеву*, потому что можно представить себе, что вызывающая и принимающая задачи выполняют его вместе. Рандеву является привлекательной чертой Ады, поскольку упрощает очень распространенное чередование ввода и вывода без больших осложнений в случае, когда требуется только ввод или только вывод.

Аналогом операции II служит в Аде оператор выбора **select**, который имеет вид

```
select
  accept get(v : out integer) do v := B[i] end; i := i + 1; ...
or accept put(v : in integer) do B[j] := v end; j := j + 1; ...
or ...
end select
```

Для исполнения будет выбрана ровно одна из альтернатив, разделенных знаком **or** в зависимости от выбора, сделанного вызывающей задачей (или задачами). Остальные операторы выбранной альтернативы, следующие за закрывающей скобкой **end** ее **accept**-части, исполняются по окончании рандеву параллельно с продолжением вызывающей задачи. На выбор альтернативы можно налагать ограничение, вводя перед ней условие, начинающееся с **when**, например:

when not *полон* \Rightarrow **accept**...

Этим достигается эффект условного критического участка.

Одна из альтернатив в операторе выбора может начинаться не с оператора приема, а с оператора задержки **delay**. Эта альтернатива может быть выбрана, если до исчерпания указанного в секундах интервала задержки не будет выбрана никакая другая альтернатива. Ее цель — предохранить от той опасности, что в результате программной или аппаратной ошибки оператор выбора будет ожидать вечно. Поскольку в нашей математической модели мы сознательно абстрагируемся от временных привязок, мы не можем должным образом представить и задержку, разве что позволив начинаться с нее полностью недетерминированному выбору альтернативы.

Одной из альтернатив в операторе выбора может быть оператор завершения **terminate**. Эта альтернатива выбирается, когда все задачи, которые могли бы вызывать данную, завершились; в этом случае завершается и она. Это не так удобно, как внутренний оператор в PASCAL PLUS, позво-

ляющий монитору, закончив работу, приводить себя в порядок.

Оператор выбора может иметь альтернативу, начинающуюся с **else**, которая выбирается, если все остальные альтернативы не могут быть выбраны немедленно, либо когда ложны все условия **when**, либо потому, что в других задачах нет соответствующего готового ожидающего вызова. Это, видимо, эквивалентно альтернативе с нулевой задержкой.

Оператор вызова также может быть защищен от произвольно долгой задержки оператором **delay** или **else**-частью. Это может привести к некоторой неэффективности при реализации на распределенной сети процессов.

Задачи в Аде описываются аналогично подчиненным процессам (разд. 4.5), но, как и мониторы в PASCAL PLUS, каждая может обслуживать любое число вызывающих процессов. Кроме того, программист должен заботиться о нормальном окончании задачи. Определение задачи разбивается на две части — спецификацию и тело. Спецификация дает имя задаче, а также задает имена и типы параметров всех входов, обращениями к которым эта задача может быть вызвана. Эта информация требуется тому, кто пишет программу, использующую данную задачу, и транслятору этой программы. Тело задачи задает ее поведение и может быть транслировано независимо от использующей программы.

Каждая задача в Аде может получить фиксированный *приоритет*. Если готовы выполняться несколько задач, но свободно меньше число процессоров, обрабатываются задачи с наибольшим приоритетом. Исполнение рандеву имеет наивысший приоритет среди вызывающих и принимающих задач. Признак приоритета называется *указанием*; оно вводится с целью улучшить критическое время реакции по сравнению с некритическим, не затрагивая при этом логического поведения программы. Это превосходная идея, поскольку она отделяет заботу об абстрактной логической правильности от проблем реакции в реальном масштабе времени, которые часто проще решить экспериментально или разумным выбором аппаратуры.

Ада располагает рядом дополнительных средств. Имеется возможность узнать, сколько вызовов ожидается на данном входе. Одна задача может вызвать принудительное прекращение другой задачи (оператор **abort**), при котором прекращаются и все ее подчиненные. Задачи могут считывать и изменять значения общих переменных. Результат этих действий будет еще более непредсказуемым, если учесть, что транслятор имеет право откладывать, переупорядочивать и объединять подобные действия над переменными так, как если бы

они не были общими. Есть некоторые дополнительные сложности и эффекты взаимодействий другого рода, которые я не упомянул.

Если забыть о сложностях, о которых говорилось в предыдущем абзаце, механизм задач в Аде представляется нам весьма удобным для реализации и использования на мультипроцессорных устройствах с общей памятью.

7.3. ВЗАИМОДЕЙСТВИЕ

Исследование возможности придания программе структуры сети взаимодействующих процессов мотивировалось, помимо всего прочего, впечатляющими успехами развития вычислительной техники. Появление микропроцессоров на несколько порядков снизило стоимость вычислительных мощностей. Однако индивидуальная мощность каждого отдельного микропроцессора оставалась все-таки несколько ниже, чем мощность типичного традиционного и по-прежнему дорогого вычислительного устройства. Поэтому наиболее экономным способом получения большей мощности представлялось использование для выполнения одного задания нескольких взаимодействующих микропроцессоров. Микропроцессоры соединялись обыкновенными проводами, по которым они могли друг с другом взаимодействовать. Каждый микропроцессор имел собственную локальную оперативную память с высокой скоростью доступа к ней, что позволяло избегать неэкономных узких мест, обычно возникающих, когда процессоры имеют доступ к общей памяти только по очереди.

7.3.1. Транспортёры

Простейшей схемой взаимодействия между элементарными процессорами является однонаправленная передача сообщений от каждого процесса его соседу по линейному транспортёру, как это описано в разд. 4.4. Впервые эта идея была предложена Конвеем, проиллюстрировавшим ее на примерах, аналогичных 4.4 X2 и X3, с той разницей, что в них предполагалось, что все компоненты транспортёра вместо того, чтобы работать бесконечно, успешно завершаются. Он предложил использовать структуру транспортёра при написании многопроходного транслятора. На машине с оперативной памятью достаточного объема одновременно активны все проходы, и управление передается между ними вместе с сообщениями, моделируя таким образом параллельное исполнение. На машине с меньшим объемом оперативной памяти одновременно активен **только** один проход, который **посылает**

свои выходные данные в файл во внешней памяти. По завершении каждого прохода начинается следующий, берущий входные данные из файла, выработанного его предшественником. Окончательный результат работы транслятора, однако, будет в точности тем же, несмотря на все отличия в способе его получения. Удачная абстракция в программировании как раз и характерна тем, что она допускает несколько различных способов реализации, эффективных при различных обстоятельствах. В данном случае предложение Конвея могло бы оказаться очень полезным при программной реализации для машинных конфигураций с широким диапазоном возможных размеров памяти.

Транспортер является также стандартным способом взаимодействия в операционной системе UNIXTM, где он называется программным каналом, а вместо обозначения \gg используется $|$.

7.3.2. Множественные буферизованные каналы

Конструкция транспортера позволяет линейной последовательности процессов передавать сообщения только в одном направлении независимо от того, буферизована ли последовательность сообщений или нет. Естественным обобщением транспортера будет позволить любому процессу взаимодействовать с любым другим процессом в обоих направлениях; и на первый взгляд таким же естественным кажется обеспечение буферизации на всех каналах, по которым происходит взаимодействие. При разработке операционной системы RC 4000 в ее ядре было реализовано средство буферизованного взаимодействия, которое использовалось для обмена сообщениями между модулями, обеспечивающими функционирование системы на более высоком уровне. В таких крупномасштабных сетях пакетной коммутации с промежуточным хранением, как сеть ARPA в Соединенных Штатах, вставка буферов между источником и местом назначения сообщений неизбежна.

Когда схема взаимодействия между процессами от линейной последовательности обобщается до, возможно, циклической сети, наличие или отсутствие буферизации может оказаться для логического поведения системы жизненно важным. Наличие буферизации не всегда желательно: можно, например, написать программу, которая придет в тупиковое состояние, если позволить размеру буфера превышать пять, точно так же, как и другую программу, где дедлок будет наступать, если не позволять размеру буфера превышать пяти. Для того чтобы справляться со всеми такими

нерегулярностями, длина всех буферов должна быть неограниченной. К сожалению, если оперативная память оказывается забитой буферизованными сообщениями, возникают серьезные проблемы в смысле эффективной реализации. Затруднения возникают и в математическом плане, так как каждая сеть в этом случае представляет собой бесконечный автомат, даже если число составляющих процессов конечно. И наконец, буферы становятся препятствием на пути быстрого и управляемого взаимодействия между человеком и машиной, поскольку могут порождать задержки между сигналом и ответом на него. Если при обработке буферизованного сигнала что-нибудь случится, будет гораздо труднее проследить и исправить ошибку. Буферизация — это техника пакетной обработки, и там, где скорость взаимодействия важнее, чем полнота загрузки процессора, ее следует избегать.

7.3.3. Функциональная мультипроцессорная обработка

Детерминированный процесс можно определить в терминах математического отображения из множества его входных каналов во множество выходных каналов. Каждый канал отождествляется с бесконечно продолжаемой последовательностью передаваемых по нему сообщений. Такие функции задаются обычным образом, рекурсией по структуре входных последовательностей, за исключением того, что не рассматривается случай пустой входной последовательности. Определим, например, процесс, выводящий результат умножения каждого входного числа на n :

$$\text{произв}_n(\text{лев}) = \langle n \times \text{лев}_0 \rangle \hat{\text{произв}}_n(\text{лев}')$$

Функция, получающая в качестве параметров два упорядоченных потока (без повторяющихся элементов) и выдающая их упорядоченное слияние (с удаленными повторяющимися элементами), определяется как

$$\begin{aligned} \text{слияние2}(\text{лев1}, \text{лев2}) = \\ \text{if } \text{лев1}_0 < \text{лев2}_0 \text{ then } \langle \text{лев1}_0 \rangle \hat{\text{слияние2}}(\text{лев1}', \text{лев2}) \\ \text{else if } \text{лев2}_0 < \text{лев1}_0 \text{ then } \langle \text{лев2}_0 \rangle \hat{\text{слияние2}}(\text{лев1}, \text{лев2}') \\ \text{else } \langle \text{лев2}_0 \rangle \hat{\text{слияние2}}(\text{лев1}', \text{лев2}') \end{aligned}$$

Ациклическую сеть можно представить в виде композиции таких функций. Например, функцию слияния трех упорядоченных входных потоков можно определить как

$$\begin{aligned} \text{слияние3}(\text{лев1}, \text{лев2}, \text{лев3}) = \\ = \text{слияние2}(\text{лев1}, \text{слияние2}(\text{лев2}, \text{лев3})) \end{aligned}$$

На рис. 7.1 показана коммутационная схема этой функции,

Циклическую сеть можно задать системой взаимно рекурсивных уравнений. Рассмотрим в качестве примера задачу, приписываемую Дейкстрой Хеммингу, а именно: определить функцию, выдающую возрастающую последовательность всех чисел, имеющих нетривиальными множителями только 2, 3 и

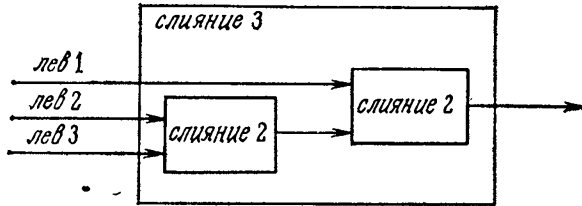


Рис. 7.1

5. Первое такое число — 1, и если x — такое число, то и $2 \times x$, $3 \times x$ и $5 \times x$ будут такими числами. Поэтому для порождения этих произведений мы будем использовать три процесса (*произв₂*, *произв₃* и *произв₅*), а затем подавать их результаты обратно процессу *слияние3*, обеспечивающему их окончательный вывод в нужном порядке (рис. 7.2).

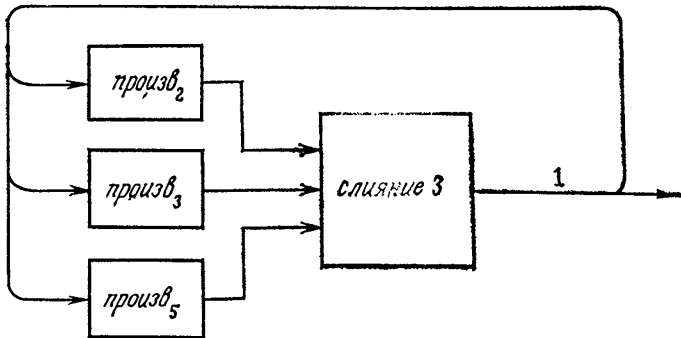


Рис. 7.2

Функция, выдающая требуемый результат, не имеет входов. Она определяется просто как

Хемминг =

$$\langle 1 \rangle \text{слияние3}(\text{произв}_2(\text{Хемминг}), \text{произв}_3(\text{Хемминг}), \text{произв}_5(\text{Хемминг}))$$

Функциональный подход к мультипроцессорным сетям имеет значительные отличия от рассмотренного в этой книге в следующих отношениях:

(1) Реализация во всей ее общности требует неограниченной буферизации каналов.

(2) Каждое попадающее в буфер значение должно оставаться там, откуда им не воспользуются все вводящие процессы, что они могут делать в разное время.

(3) Процесс не имеет возможности ожидать одного из двух вводов в зависимости от того, какой придет первым.

(4) Все процессы детерминированы.

Недавние исследования были направлены на снижение неэффективности (1) и (2) и на ослабление ограничений (3) и (4).

7.3.4. Небуферизованное взаимодействие

Уже много лет назад я принял решение взять за основу небуферизованное (синхронное) взаимодействие. Сделано это было по следующим соображениям:

(1) Оно весьма точно соответствует физической реализации, при которой процессоры соединяются проводами. Провода же не могут хранить сообщения.

(2) Оно весьма точно соответствует эффекту вызова и возврата из подпрограмм внутри единственного процессора с копированием значений параметров и результатов.

(3) В случае необходимости буферизация может быть реализована как обыкновенный процесс; при этом программист может точно задать степень буферизации.

(4) Другие недостатки буферов уже обсуждались в конце разд. 7.3.2.

Разумеется, ни один из этих аргументов не является абсолютно убедительным. Например, если взять за основу буферизованное взаимодействие, то все эти соображения не имеют значения для общего случая вызовов и возвратов подпрограмм; во всех же остальных случаях синхронизация может быть достигнута вводом подтверждения вслед за каждым выводом и выводом подтверждения вслед за каждым вводом.

7.3.5. Взаимодействующие последовательные процессы

Так называлось мое первое законченное изложение языка программирования, основанного на параллелизме и взаимодействии. Эту раннюю версию от настоящей книги отличают два важных момента.

(1) Параллельная композиция

Каналы не именованы. Вместо этого составные процессы в параллельной конструкции имеют различные имена, пред-

существующие им и отделенные от них парой двоеточий:

$$[a :: P \parallel b :: Q \parallel \dots \parallel c :: R]$$

Внутри процесса P команда $b!v$ выводит значение v процессу с именем b . Это значение вводится командой $a?x$ внутри процесса Q . Имена процессов локальны по отношению к параллельной команде, в которой они введены, а взаимодействия между составляющими процессами скрыты.

Преимущество этой схемы заключается в том, что в ней не требуется введения в язык понятия канала или его описания. Более того, в ней логически невозможно нарушить то ограничение, что канал соединяет только два процесса, один из которых использует его для ввода, а другой — для вывода. Но она имеет и некоторые недостатки как в практическом, так и в теоретическом плане.

(1) Серьезным практическим недостатком является то, что подчиненному процессу требуется знать имя использующего его процесса; это затрудняет построение библиотеки подчиненных процессов.

(2) Недостаток математической основы состоит в том, что параллельная композиция представляет собой операцию с переменным числом параметров и не может быть сведена к ассоциативному двуместному оператору типа \parallel .

(2) Автоматическое завершение

В ранней версии предполагалось, что все процессы в параллельном операторе завершаются. Основанием к этому служила надежда, что корректность процесса можно описать так же, как и корректность обычной программы, с помощью постулов, т. е. предиката, который должен быть истинным при успешном завершении. (Надежда эта не оправдалась, и теперь более приемлемыми кажутся другие методы доказательства (см. разд. 1.10).) Требование завершения подчиненного процесса налагает на использующий процесс обременительное обязательство сигнализировать о своем завершении всем своим подчиненным процессам. Поэтому было введено следующее соглашение. Цикл вида

$$*[a?x \rightarrow P \parallel b?x \rightarrow Q \parallel \dots]$$

завершается автоматически по завершении **всех** процессов a, b, \dots , от которых запрашивается ввод. Это позволяет подчиненному процессу завершить перед окончанием всю необходимую реинициализацию кода — средство, оправдавшее себя в языках Симула и PASCAL PLUS.

Проблема с этим соглашением состоит в том, что его трудно строго описать и реализовать, а методы доказательства правильности программ с ним оказываются не проще, чем без него. Теперь мне кажется (как и в разд. 4.5), что лучше снять ограничение об обязательном завершении простых подчиненных процессов, а в более сложных случаях (разд. 6.4) принимать другие меры.

7.3.6. Оккам

В противоположность Аде язык Оккам очень прост, и построен он на основе принципов, очень близких к изложенным в этой книге. Из различий прежде всего бросаются в глаза отличия в обозначениях. Синтаксис Оккама создавался в расчете на экраный синтаксически ориентированный редактор; в нем используются инфиксные, а не префиксные операции, а выделения производятся не с помощью скобок, а структурированным расположением текста.

SEQ	соответствует $(P; Q; R)$
P	
Q	
R	
PAR	соответствует $(P \parallel Q \parallel R)$
P	
Q	
R	
ALT	соответствует $(c?x \rightarrow P \parallel d?y \rightarrow Q)$
$c?x$	
P	
$d?y$	
Q	
IF	соответствует $(P \triangleleft B \triangleright Q)$
B	
P	
$NOT B$	
Q	
$WHILE B$	соответствует (B^*P)
P	

Конструкция ALT соответствует оператору выбора в Аде и содержит аналогичный набор альтернатив. Возможность выбора альтернативы может зависеть от истинности некоторого логического условия B :

$B \& c?x$
 P

Ввод можно заменить на *ПРОПУСК*; в этом случае альтернатива может быть выбрана всякий раз, когда истинно предвещающее ее логическое условие. Замена ввода ожиданием позволяет выбрать альтернативу по истечении указанного временного интервала.

В языке Оккам не предусмотрено специальных обозначений для транспортеров (разд. 4.4), подчиненных (разд. 4.5) или разделяемых (разд. 6.4) процессов. Все необходимые схемы взаимодействия достигаются явным отождествлением имен каналов. Для этих целей имеется возможность описывать процедуры с каналами в качестве параметров. Например, простой процесс копирования можно описать как

```
PROC копир(CHAN лев,прав) =
  WHILE TRUE
    VAR x:
    SEQ
      лев?x
      прав!x;
```

Теперь можно определить двойной буфер КОПИР»КОПИР:

```
CHAN средн:
PAR
  копир(лев,средн)
  копир(средн,прав)
```

Цепочка из n буферов строится при помощи вектора из n каналов и итеративной формы параллельной конструкции, объединяющей $n - 1$ процесс, соответствующий значениям i от 0 до $n - 2$ включительно:

```
CHAN средн[n - 1]:
PAR
  копир(лев,средн[0])
  PAR i = [0 FOR n - 2]
    копир(средн[i],средн[i + 1])
  копир(средн[n - 2],прав)
```

Поскольку Оккам предполагает реализацию со статическим размещением памяти при фиксированном числе процессоров, величина n в предыдущем примере должна быть постоянной. По этой же причине не допускаются рекурсивные процедуры,

Эффект подчиненного процесса может быть достигнут с помощью аналогичной конструкции, например:

```
PROC удвоить(лев,прав) ==
  WHILE TRUE
    VAR x:
    SEQ
      лев?x
      прав!(x + x):
```

Этот процесс можно описать как подчиненный по отношению к единственному использующему процессу *P*:

```
CHAN удв.лев,удв.прав:
PAR
  удвоить(удв.лев, удв.прав)
P
```

Внутри *P* удвоение числа производится с помощью команд

```
удв.лев!4; удв.прав?y;...
```

Процессы могут совместно использоваться с помощью векторов каналов (где каждый элемент соответствует одному использующему процессу), а также с помощью итеративной формы конструкции *ALT*. Для примера рассмотрим интегратор, выдающий после каждого нового ввода сумму всех уже введенных чисел:

```
CHAN слож,интеграл[n - 1]:
PAR
  VAR сумма,x:
  SEQ
    сумма := 0
    ALT i := [0 FOR n]
      слож[i]?x
    SEQ
      сумма := сумма + x
      интеграл[i]!сумма
  PAR i := [0 FOR n]
  ...процессы-пользователи...
```

Так же как и Ада, Оккам позволяет программисту приписывать процессам, объединенным параллельной композицией, относительные приоритеты. Для этого вместо *PAR* используется конструкция *PRI PAR*; при этом приоритет процесса определяется его местом в списке (чем ближе к началу, тем выше приоритет). Предоставляемые языком средства экран-

ного редактирования позволяют при необходимости переупорядочивать процессы. Аналогичный выбор предоставляется и для конструкции *ALT*, которая в этом случае имеет вид *PRI ALT*. Она обеспечивает, что если к моменту выбора готовы несколько альтернатив, то из них будет выбрана текстуально первая; иначе эффект этого оператора тот же, что и просто *ALT*. Конечно, от программиста требуется, чтобы его программа была логически правильной независимо от приписываемых приоритетов.

В Оккамe имеются возможности для распределения процессов между различными процессорами и задания того, какие физические выходы каждого процессора будут использоваться для соответствующих каналов Оккам-программы, а какие — для загрузки кода самой программы.

7.4. МАТЕМАТИЧЕСКИЕ МОДЕЛИ

Идея того, что язык программирования должен обладать точным математическим смыслом или семантикой, была осознана еще в начале 1960-х годов. Математика предоставляет надежную, недвусмысленную, точную и стабильную спецификацию языка, которая может служить согласованным интерфейсом между его пользователями и его реализаторами. Более того, лишь математика может позволить нам обоснованно утверждать, что различные реализации являются реализациями одного и того же языка. Поэтому можно сказать, что в деле стандартизации языков математическая семантика столь же существенна, как счет и измерение в стандартизации гаек и болтов.

В конце 1960-х годов сложилось понимание еще более важной роли математической семантики, а именно как средства, помогающего программисту реализовать свою ответственность в установлении правильности его программы. Флорид высказал идею, что семантику удобнее формулировать в виде набора корректных правил доказательства, нежели в виде явной математической модели. Эта идея была реализована в спецификации языков Паскаль, Евклид и Джипси.

Ранняя версия теории взаимодействующих последовательных процессов (разд. 7.3.5) не имела математической семантики и оставляла открытыми ряд вопросов, существенных для разработчика, например:

- (1) Допустима ли вложенность одной параллельной команды в другую?
- (2) Если да, то можно ли написать рекурсивную процедуру, которая вызывает себя параллельно?

(3) Имеется ли теоретическая возможность использовать команды вывода в предваряющем выражении?

Представленная в этой книге математическая теория дает положительные ответы на все эти вопросы.

7.4.1. Исчисление взаимодействующих систем

Решающий успех в математическом моделировании параллелизма был достигнут Робинот Милнером. Целью его исследования было предоставить основу для построения и сравнения различных моделей, обладающих разной степенью абстракции. Поэтому для начала он определил базисный синтаксис выражений для обозначения процессов и ввел ряд эквивалентностей между этими выражениями, наиболее важными из которых являются: сильная эквивалентность, наблюдаемая эквивалентность, наблюдаемая конгруэнтность. Каждая эквивалентность задает свою модель параллелизма. Под известным сокращением CCS (Calculus of Communicating Systems) обычно подразумевается модель, где за определение отношения равенства процессов берется наблюдаемая конгруэнтность.

Основные обозначения в CCS можно проиллюстрировать следующими соотношениями:

$a.P$	соответствует выражению $a \rightarrow P$
$(a.P) + (b.Q)$	соответствует выражению $(a \rightarrow P \mid b \rightarrow Q)$
NIL	соответствует выражению <i>СТОП</i>

По сравнению с этими различиями в обозначениях гораздо более важной является разница в трактовке упрятывания. В CCS существует специальный символ τ , соответствующий наступлению скрытого события или внутреннему переходу. Преимуществом такого протоколирования скрытого события является то, что его можно свободно использовать для предварения рекурсивных уравнений, гарантируя таким образом единственность их решений, как это было в разд. 2.8.3. Вторым (возможно, менее значительным) преимуществом является то, что процессы, которые могут участвовать в бесконечной последовательности τ -событий, не обязательно эквивалентны процессу *ХАОС*; таким образом, имеется возможность с пользой различать потенциально расходящиеся процессы. В CCS, однако, не удастся различить потенциально расходящиеся процессы и нерасходящиеся процессы, обладающие аналогичным поведением. Я думаю, что это делает невозможной эффективную реализацию полного исчисления взаимодействующих систем.

Среди элементарных операторов в CCS отсутствует аналог операции \sqcap . Однако недетерминизм можно промоделировать с помощью τ -событий, например:

$(\tau.P) + (\tau.Q)$ соответствует выражению $P \sqcap Q$
 $(\tau.P) + (a.Q)$ соответствует выражению $P \sqcap (P \parallel (a \rightarrow Q))$

Правда, эти соответствия неточны, поскольку в CCS недетерминизм, определенный в терминах τ , не будет ассоциативным, что иллюстрируется рис. 7.3, где деревья различны.

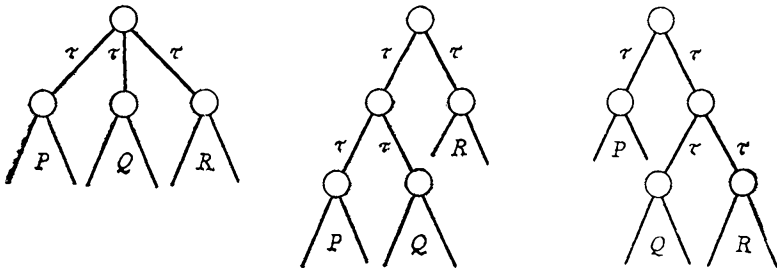


Рис. 7.3

Кроме того, префиксация не дистрибутивна относительно недетерминизма. В качестве примера приведены деревья на рис. 7.4, которые различны, если $P \neq Q$.

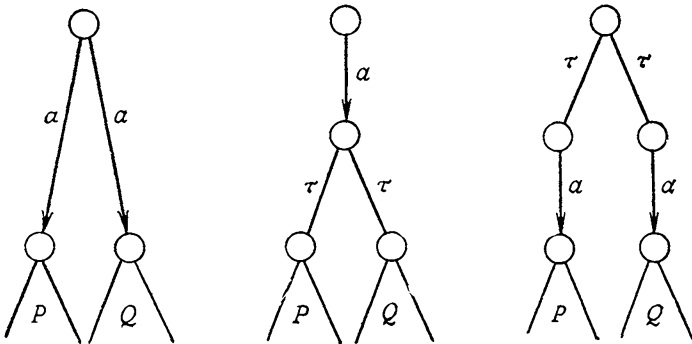


Рис. 7.4

Эти примеры показывают, что между процессами, которые в данной книге рассматривались как тождественные, в CCS может существовать много различий. Причина в том, что CCS призвана служить основой для целого семейства моделей, в каждой из которых могут вводиться дополнительные

по сравнению с CCS отождествления, но не могут нарушаться уже имеющиеся. Во избежание сужения диапазона получаемых моделей в CCS вводятся лишь те отождествления, которые представляются наиболее существенными. В этой книге при построении математической теории мы преследовали прямо противоположную цель — мы делали как можно больше отождествлений, оставляя лишь наиболее существенные различия. Поэтому наш набор алгебраических законов неизмеримо богаче. Мы надеемся, что эти законы принесут практическую пользу в рассуждениях, касающихся разработки и реализации; в частности, они допускают больше преобразований и оптимизаций, чем CCS.

Основной оператор параллельной комбинации в CCS обозначается одной чертой ($|$). Он намного сложнее оператора \parallel в силу того, что наряду с синхронизацией он включает в себя элементы упрятывания, недетерминизма и чередования. Каждое событие в CCS имеет две формы — простую (a), и с чертой (\bar{a}). Когда два процесса объединяются для параллельного исполнения, синхронизация возникает только тогда, когда один из процессов исполняет событие с чертой, а другой — соответствующее простое событие. Их совместное участие в этом общем событии тотчас же скрывается, преобразуясь в τ . Синхронизация, однако, не является обязательной; каждое из двух событий может происходить явно и независимо при взаимодействии с внешним окружением. Таким образом, в CCS

$$\begin{aligned}(a.P)|(\bar{b}.Q) &= a.(P|(\bar{b}.Q)) + b.((a.P)|Q) \\ (a.P)|(a.Q) &= a.(P|(a.Q)) + a.((a.P)|Q) \\ (a.P)|(\bar{a}.Q) &= \tau.(P|Q) + a.(P|(\bar{a}.Q)) + \bar{a}.((a.P)|Q)\end{aligned}$$

Следовательно, синхронизованное взаимодействие может произойти только между двумя процессами; если готовы больше двух процессов, выбор пары происходит недетерминированно:

$$\begin{aligned}(a.P)|(a.Q)|(\bar{a}.R) &= \tau.(P|(a.Q)|R) + \tau.((a.P)|Q|R) \\ &\quad + a.(P|(a.Q)|(\bar{a}.R)) \\ &\quad + a.((a.P)|Q|(\bar{a}.R)) \\ &\quad + \bar{a}.((a.P)|(a.Q)|R)\end{aligned}$$

Благодаря дополнительной сложности параллельного оператора не возникает необходимости в операторе упрятывания. Вместо него имеется оператор *сужения* \backslash , который просто не допускает наступления соответствующих событий и удаляет их из алфавита процесса вместе с их надчеркну-

тым вариантом. Его действие иллюстрируется следующими законами CCS:

$$\begin{aligned}
 (a.P) \setminus \{a\} &= (\bar{a}.P) \setminus \{a\} = NIL \\
 (P + Q) \setminus \{a\} &= (P \setminus \{a\}) + (Q \setminus \{a\}) \\
 ((a.P) | (\bar{a}.Q)) \setminus \{a\} &= \tau.((P | Q) \setminus \{a\}) \\
 ((a.P) | (a.Q) | (\bar{a}.R)) \setminus \{a\} &= \tau.((P | (a.Q) | R) \setminus \{a\}) \\
 &\quad + \tau.(((a.P) | Q | R) \setminus \{a\})
 \end{aligned}$$

Последний из этих законов иллюстрирует мощность параллельного комбинирующего оператора в CCS, позволяющую достигнуть эффекта разделения процесса $(\bar{a}.R)$ между двумя использующими процессами $(a.P)$ и $(a.Q)$. Целью CCS и являлось достижение максимальной выразительной мощности средствами как можно меньшего числа различных элементарных операций. Это составляет источник элегантности и силы CCS и сильно упрощает исследование семейств моделей, задаваемых различными отношениями эквивалентности.

В данной книге избран противоположный подход. Здесь простота достигается за счет создания единственной простой модели, в терминах которой легко определить столько операций, сколько может потребоваться для исследования спектра различных понятий. Например, операция недетерминированного выбора \square задает недетерминизм в его самом чистом виде и совершенно не зависит от совершаемого обстановкой выбора, представленного операцией $(x: B \rightarrow P(x))$. Аналогично, операция \parallel задает параллелизм и синхронизацию, совершенно независимые как от недетерминизма, так и от упрятывания, каждое из которых в свою очередь представлено отдельной операцией. На тот факт, что соответствующие понятия различны, указывает, пожалуй, простота алгебраических законов. Для успешной практической реализации математических теорий нам необходим достаточно широкий спектр операций. Минимизация набора операций, однако, тоже бывает полезной, особенно в теоретических исследованиях.

Для спецификации наблюдаемого поведения процесса Милнер ввел особую форму модальной логики. Модальность

$$\Diamond a \ S$$

описывает процесс, который может выполнить действие a , а затем вести себя в соответствии с S , а двойственная модаль-

$$\Box a \ S$$

ность описывает процесс, который, начав с a , должен затем вести себя в соответствии с S . Вводится исчисление коррект-

ности, позволяющее доказать, что процесс P соответствует спецификации S — факт, выраженный в традиционных обозначениях логики:

$$P \models S$$

Это исчисление существенно отличается от того, которому подчиняется отношение уд , потому что оно базируется, скорее, на структуре спецификации, нежели на структуре программы. Например, правило для отрицания имеет вид

$$\text{если неверно, что } P \models F, \text{ то } P \models \neg F.$$

Это означает, что процесс P должен быть полностью записан, прежде чем начнется доказательство его правильности. Использование же отношения уд , напротив, позволяет строить доказательство правильности составного процесса из доказательств для составляющих его частей. Кроме того, не возникает необходимости доказывать, что процесс *не* удовлетворяет своей спецификации. Модальная логика представляет большой теоретический интерес, но в условиях взаимодействующих процессов надежды на ее успешное практическое применение пока не оправдываются.

Вообще равенство в CCS является сильным отношением, поскольку равные процессы должны иметь сходство как в наблюдаемом поведении, так и в структуре скрытого поведения. Поэтому CCS представляет собой хорошую модель для формулирования и исследования разнообразных более слабых определений эквивалентности, в которых многие аспекты скрытого поведения не принимаются во внимание. Милнер сделал это, введя понятие наблюдаемой эквивалентности. Оно включает определение множества наблюдений или экспериментов, которые можно провести над процессом; тогда два процесса эквивалентны, если не существует эксперимента, который можно сделать над одним из них и невозможно сделать над другим — хорошее приложение философского принципа отождествления неразличимого. Этот принцип был взят за основу математической теории в этой книге, где процесс отождествляется с множеством возможных наблюдений за его поведением. Признаком успешного применения этого принципа служит то, что два процесса P и Q эквивалентны тогда и только тогда, когда они удовлетворяют одним и тем же спецификациям:

$$\forall S. P \models S \equiv Q \models S$$

К сожалению, это не всегда так просто. Если два процесса решено считать равными, равными должны быть и резуль-

таты преобразования их с помощью одной и той же функции, т. е.

$$(P \equiv Q) \Rightarrow (F(P) \equiv F(Q))$$

Поскольку предполагается, что событие τ скрыто, из естественного определения наблюдения должна следовать эквивалентность

$$(\tau.P) \equiv P$$

Однако процесс $(\tau.P + \tau.NIL)$ не должен быть эквивалентен процессу $(P + NIL)$, равному P , поскольку первый из них может недетерминированно выбрать дедлок вместо того, чтобы вести себя как P .

Для решения этой проблемы Милнер предложил вместо эквивалентности использовать понятие *конгруэнтности*. Один из экспериментов, которые можно провести над процессом P , состоит в помещении его в обстановку $F(P)$ (где F строится из других процессов средствами операторов языка), а затем — наблюдении за поведением полученной системы. Процессы P и Q называются (наблюдаемо) конгруэнтными, если для каждой выразимой в языке обстановки F процесс $F(P)$ наблюдаемо эквивалентен процессу $F(Q)$. Согласно этому определению, $\tau.P$ не конгруэнтен P . Открытие полного набора законов конгруэнтности является важным математическим достижением.

Потребность в дополнительных сложностях, связанных с наблюдаемой конгруэнтностью, происходит от невозможности достаточно глубокого наблюдения за поведением процесса P без помещения его в обстановку $F(P)$. Вот почему мы ввели понятие *множества* отказов процесса, а не просто отказа от единственного события. Множество отказов представляется нам наиболее слабым типом наблюдения, эффективно отражающего возможность недетерминированного дедлока; соответственно оно ведет к гораздо более слабой эквивалентности и более мощному набору алгебраических законов, чем CCS.

В приведенном здесь описании сделан слишком большой упор на отличия CCS и практическую применимость подхода, предложенного в этой книге. На самом деле оба подхода объединяет большинство важнейших характеристик, а именно прочная математическая основа в рассуждениях о спецификации, разработке и реализации, и любой из них может быть использован как в теоретических исследованиях, так и в практических приложениях.

ИЗБРАННАЯ ЛИТЕРАТУРА

- Conway W. E. Design of a Separable Transition-Diagram Compiler, *Comm. ACM* 6 (7), 8—15 (1963).
Классическая работа, посвященная сопрограммам.
- Hoare C. A. R. Monitors: An Operating System Structuring Concept, *Comm. ACM* 17 (10), 549—557 (1974).
Языковое средство построения операционных систем.
- Hoare C. A. R. Communicating Sequential Processes, *Comm. ACM* 21 (8), 666—677 (1978).
Разработка языка программирования, являющаяся ранней версией разработки, представленной в этой книге.
- Milner R. *A Calculus of Communicating Systems*, Lecture Notes in Computer Science 92, Springer-Verlag, New York (1980).
Ясная математическая трактовка общей теории параллелизма и синхронизации.
- Kahn G. The Semantics of a simple language for Parallel Programming. In *Information Processing*, 74, North Holland, Amsterdam, pp. 471—475 (1974).
Изысканная трактовка функционального мультипрограммирования.
- Welsh J.,
McKeag R. M. *Structured System Programming*, Prentice-Hall, London, pp. 324 (1980).
Сообщение о языке PASCAL PLUS и его использовании в структурировании операционных систем и трансляторов.
- Filman R. E.,
Friedman D. P. *Coordinated Computing, Tools and Techniques for Distributed Software*, McGraw-Hill, pp. 370 (1984).
Полезный обзор и дальнейшая литература.
- Dahl O.-J. Hierarchical Program Structures, in *Structured Programming*, Academic Press, London, pp. 175—220 (1982).
Знакомство с основными идеями языка Симула 67.
- (INMOS Ltd.) *occam™ Programming Manual*, Prentice-Hall International, pp. 100 (1984).
- (ANSI/MIL-STD 1815A) *Ada™ Reference Manual*.
В главе 9 описан механизм задач.
- Brookes S. D.,
Hoare C. A. R. A Theory of Communicating Sequential Processes, *Journal ACM* 31 (7), 560—599 (1984).
- Roscoe A. W. Рассматриваются математические аспекты недетерминированных процессов, за исключением расхождимости.
- Brookes S. D.,
Roscoe A. W. An Improved Failures Model for Communicating Sequential Processes, in *Proceedings NSF—SERC Seminar on Concurrency*, Springer-Verlag, New York, Lecture Notes in Computer Science (1985).
Здесь вводится понятие расхождимости.

УКАЗАТЕЛЬ СИМВОЛОВ

Логика

Обозначение	Значение	Пример
$=$	равно	$x = x$
\neq	отлично от	$x \neq x + 1$
$P \& Q$	P и Q (истинны оба)	$x \leq x + 1 \& x \neq x + 1$
$P \vee Q$	P или Q (истинно одно или оба)	$x \leq y \vee y \leq x$
$\neg P$	не P (P не истинно)	$\neg 3 > 5$
$P \Rightarrow Q$	если P , то Q	$x < y \Rightarrow x \leq y$
$P \equiv Q$	P тогда и только тогда, когда Q	$x < y \equiv y > x$
$\exists x. P$	существует x , такой, что P	$\exists x. x > y$
$\forall x. P$	P для всех x	$\forall x. x < x + 1$
$\exists x: A. P$	существует x из множества A , такой, что P	
$\forall x: A. P$	P для всех x из множества A	

Множества

Обозначение	Значение	Пример
\in	принадлежит	Наполеон \in человечество
\notin	не принадлежит	Наполеон \notin русские
$\{\}$	пустое множество (не содержащее элементов)	$\neg \text{Наполеон} \in \{\}$
$\{a\}$	одноэлементное множество, состоящее из a ; a — его единственный элемент	$x \in \{a\} \equiv x = a$
$\{a, b, c\}$	множество с элементами a, b, c	$c \in \{a, b, c\}$
$\{x \mid P(x)\}$	множество из всех x , таких, что $P(x)$	$\{a\} = \{x \mid x = a\}$
$A \cup B$	объединение A и B	$A \cup B = \{x \mid x \in A \vee x \in B\}$
$A \cap B$	пересечение A и B	$A \cap B = \{x \mid x \in A \& x \in B\}$
$A - B$	A минус B	$A - B = \{x \mid x \in A \& \neg x \in B\}$
$A \subseteq B$	A содержится в B	$A \subseteq B = \forall x: A. x \in B$
$A \supseteq B$	A содержит B	$A \supseteq B \equiv B \subseteq A$
$\{x: A \mid P(x)\}$	множество x из A , таких, что $P(x)$	
\mathbb{N}	множество натуральных чисел	$\{0, 1, 2, \dots\}$

Множества

Обозначение	Значение	Пример
$\mathbb{P}A$	множество-степень A	$\mathbb{P}A = \{X \mid X \subseteq A\}$
$\bigcup_{n \geq 0}$	объединение семейства множеств	$\bigcup_{n \geq 0} A_n = \{x \mid \exists n \geq 0. x \in A_n\}$
$\bigcap_{n \geq 0}$	пересечение семейства множеств	$\bigcap_{n \geq 0} A_n = \{x \mid \forall n \geq 0. x \in A_n\}$

Функции

Обозначение	Значение	Пример
$f: A \rightarrow B$	f — функция, отображающая каждый элемент A в некоторый элемент B	квадрат: $\mathbb{N} \rightarrow \mathbb{N}$
$f(x)$	тот элемент B , в который f отображает x (из A)	
инъективная функция	функция f , отображающая различные элементы A в различные элементы B	$x \neq y \Rightarrow f(x) \neq f(y)$
f^{-1}	функция обратная к инъективной функции f	$x = f(y) \equiv y = f^{-1}(x)$
$\{f(x) \mid P(x)\}$	множество, образованное применением f ко всем x , таким, что $P(x)$	$\{y \mid \exists x. y = f(x) \ \& \ P(x)\}$
$f(C)$	образ C при отображении f	квадрат($\{3, 5\}$) = $\{9, 25\}$
$f \circ g$	композиция f и g	$f \circ g(x) = f(g(x))$
$\lambda x. f(x)$	функция, отображающая каждое значение x в $f(x)$	$(\lambda x. f(x))(3) = f(3)$

Протоколы

Раздел	Обозначение	Значение
1.5	$\langle \rangle$	пустой протокол
1.5	$\langle a \rangle$	протокол, содержащий только a (одноэлементная последовательность)
1.5	$\langle a, b, c \rangle$	протокол из трех символов a , потом b , потом c
1.6.1	\wedge	затем (между протоколами)
1.6.1	s^n	s , повторенный n раз
1.6.2	$s \upharpoonright A$	сужение s на A
1.6.5	$s \leq t$	s является префиксом t
4.2.2	$s \leq_n t$	s — префикс t и короче его не более, чем на n символов
1.6.5	$s \text{ в } t$	s содержится в t
1.6.6	$\# s$	длина s
1.6.8	$s \downarrow b$	число вхождений символа b в протокол s
1.9.6	$s \downarrow c$	взаимодействия по каналу c , зафиксированные в протоколе s
1.9.2	\wedge / s	конкатенация s
1.9.7	$s; t$	s , за которым следует t

(одноэлементная последовательность)

 a , потом b , потом c $\langle a, b, c \rangle = \langle a, b \rangle \wedge \langle \rangle \wedge \langle c \rangle$ $\langle a, b \rangle^2 = \langle a, b, a, b \rangle$ $\langle b, c, d, a \rangle \upharpoonright \{a, c\} = \langle c, a \rangle$ $\langle a, b \rangle \leq \langle a, b, c \rangle$ $\langle a, b \rangle \leq_2 \langle a, b, d, c \rangle$ $\langle c, d \rangle \text{ в } \langle b, c, d, a, b \rangle$ $\# \langle b, c, b, a \rangle = 4$ $\langle b, c, b, a \rangle \downarrow b = 2$ $\langle c.1, a.4, c.3, d.1 \rangle \downarrow c = \langle 1, 3 \rangle$ $\wedge / \langle \langle a, b \rangle, \langle \rangle, \langle c \rangle \rangle = \langle a, b, c \rangle$
 $(s \wedge \langle \sqrt{\rangle}) : t = s \wedge \bar{t}$

Протоколы

Раздел Обозначение Значение

1.6.4	A^*	множество последовательностей с элементами из A	$A^* = \{s \mid s \uparrow A = s\}$
1.6.3	s_0	голова s	$\langle a, b, c \rangle_0 = a$
1.6.3	s'	хвост s	$\langle a, b, c \rangle' = \langle b, c \rangle$
1.9.4	$s[i]$	i -й элемент s	$\langle a, b, c \rangle[1] = b$
1.9.1	$f^*(s)$	f со звездочкой от s	$\text{квадрат}^*((1, 5, 3)) = \langle 1, 25, 9 \rangle$
1.9.4	\bar{s}	обратная последовательность к s	$\langle a, b, c \rangle = \langle c, b, a \rangle$

Специальные события

Раздел Обозначение Значение

1.9.7	\surd	успех (успешное завершение)	
2.6.2	$l.a$	участие процесса с именем l в событии a	
4.1	$c.v$	взаимодействие (передача или прием) по каналу a величины v	
4.5	$l.c$	канал c процесса с именем l	
4.5	$l.c.v$	передача (прием) сообщения v по каналу $l.c$	
5.4.1	\lceil	катастрофа (молния)	
5.4.3	\otimes	смсна	
5.4.4	\odot	контрольная точка для последующего восстановления	
6.2	занят	захват ресурса	
6.2	свободен	освобождение ресурса	

Процессы

Раздел Обозначение Значение

1.1	αP	алфавит процесса P	
4.1	ac	множество сообщений, которые можно передавать по каналу c	
1.1.1	$a \rightarrow P$	P за a	
1.1.3	$(a \rightarrow P \mid b \rightarrow Q)$	P за a или Q за b (при условии, что $a \neq b$)	
1.1.3	$(x: A \rightarrow P(x))$	P от x за x (выбранным) из A	
1.1.2	$\mu X: A.F(X)$	процесс X с алфавитом A , такой, что $X = F(X)$	
1.8.3	P/s	P после (участия в событиях из протокола) s	
2.3	$P \parallel Q$	P параллельно с Q	
2.6.2	$l: P$	P с именем l	
2.6.4	$L: P$	P с именами из множества L	
3.2	$P \sqcap Q$	P или Q (недетерминированно)	
3.3	$P \sqcup Q$	выбор между P и Q	
3.5	$P \setminus C$	P без C (упрятывание)	
3.6	$P \parallel\!\!\!\parallel Q$	чередование P и Q	
4.4	$P \gg Q$	сцепление P с Q	
4.5	$P // Q$	P — подчиненный к Q	
6.4	$l: p // Q$	дистанционное подчинение	
5.1	$P; Q$	Q следует за P	
5.4	$P \sim Q$	P прерывается Q	
5.4.1	$P \hat{\sim} Q$	P , а в случае катастрофы Q	
5.4.2	\hat{P}	перезапускаемый P	
5.4.3	$P \otimes Q$	сменяющие друг друга P и Q	

Процессы

Раздел	Обозначение	Значение
5.5	$P \nless b \nless Q$	если b , то P иначе — Q
5.1	$*P$	повторение P
5.2	b^*P	пока b повторять P
5.4	$x := e$	x получает значение e
4.2	$b!e$	вывод (значения) e по (каналу) b
4.2	$b?x$	x присвоить значение, поступившее по каналу b
6.2	$le?x$	вызов разделяемой подпрограммы с именем l , с входными параметрами e и записью результатов в x
1.10.1	$P \text{ уд } S$	(процесс) P удовлетворяет (спецификации) S
1.10.1	np	произвольный протокол специфицируемого процесса
3.7	$отк$	произвольный отказ специфицируемого процесса
5.5.2	x^\vee	конечное значение x , вырабатываемое специфицируемым процессом
5.5.1	$пер(P)$	множество переменных, которым P может присваивать значения
5.5.1	$дост(P)$	множество переменных, значения которых доступны для P
2.8.2	$P \sqsubseteq Q$	(детерминированный процесс) Q может делать по крайней мере столько же, сколько P
3.9	$P \sqsubseteq Q$	(недетерминированный процесс) Q не хуже P
5.5.1	$\mathcal{D}e$	и, возможно, лучше выражение e определено

Алгебра

Термин (свойство)	Указанным свойством обладает	
рефлексивность	отношение R , такое, что	xRx
антисимметричность	отношение R , такое, что	$xRy \ \& \ yRx \Rightarrow x = y$
транзитивность	отношение R , такое, что	$xRy \ \& \ yRz \Rightarrow xRz$
частичный порядок	отношение \leq , которое рефлексивно, антисимметрично и транзитивно	
наименьший элемент	элемент \perp , такой, что	$\perp \leq x$
монотонность	функция f , сохраняющая частичный порядок	$x \leq y \Rightarrow f(x) \leq f(y)$
строгость	функция f , такая, что	$f(\perp) = \perp$
идемпотентность	двуместный оператор f такой, что	$xfx = x$
симметричность	двуместный оператор f , такой, что	$xfy = yfx$
ассоциативность	двуместный оператор f , такой, что	$xf(yfz) = (xfy)fz$
дистрибутивность	f дистрибутивна относительно g , если	$xf(ygz) = (xfy)g(xfz)$ $\wedge (ygz)fx = (yfx)g(zfx)$
единица	элемент 1 такой, что	$xf1 = 1fx = x$
нуль	элемент 0 такой, что	$xf0 = 0fx = 0$

Предметный указатель

- Ада (Ada) 235
активационные записи (activation records) 212
алгебраические законы (algebraic laws) 101
Алгол 60, 231
алфавит (alphabet) 18
альтернатива (alternative) 26
ангельский недетерминизм (angelic non-determinism) 99

бесконечный перехват (infinite overtaking) 75
блок управления (control block) 212
бронирование авиабилетов (flight reservation) 197
буфер (buffer) 155
БУФЕР (BUFFER) 134

ввод в подкачку (spooled input) 218
вершина (node) 29
взаимная рекурсия (mutual recursion) 28
взаимное влияние (interference) 202
взаимодействие (communication; interaction) 60, 129
взвешенная сумма (weighted sum) 141
ВИЛКА (FORK) 70
виртуальный ресурс (virtual resource) 212
вложение мониторов (nested monitors) 232
внутреннее предложение (inner statement) 231
вспомогательный файл (scratch file) 209
вставка битов (bit stuffing) 157
входной канал (input channel) 130
выбор (choice) 24
выбор 2 (choice 2) 34
выборка (selection) 52
вывод с откачкой (spooled output) 218

выходной канал (output channel) 130

генеральный выбор (general choice) 101
главный процесс (main process, master) 159
голова (head) 40

двоичное дерево (binary tree) 162
двоичный семафор (binary exclusion semaphore) 203
двойной буфер (double buffer) 150
дедлок (deadlock) 61
детерминированный процесс (deterministic process) 87
Джипси (Gypsy) 247
дистанционный вызов (remote call) 206
дистрибутивная функция (distributive function) 39
длина (length) 42
дополнительная память (backing storage) 207
допускающий процесс (accepting process) 170
дост (acc) 186
дуга (arc) 29

единственное решение (unique solution) 92

законы (laws) 31
замыкание (livelock) 152
занят (acquire) 198
защита (protection) 232
звездочка (star) 40

или 1 (or 1) 99
или 2 (or 2) 99
или 3 (or 3) 99
имя входа (entry name) 235
индекс (subscript) 52
ИСП_A (RUN_A) 27

- канал (channel) 129
 катастрофа (catastrophe) 178
 класс (class) 231
 композиция (composition) 53
 конгруэнтность (congruence) 253
 конечное множество (finite set) 162
 конкатенация (concatenation) 51
 конструктивная функция (constructive function) 92
 контекстно-свободная грамматика (context-free grammar) 171
 контрольные точки (checkpoints) 181
 координатный коммутатор (crossbar switch) 215
КОПИБИТ (COPYBIT) 26
КОПИР (COPY) 131
 критический участок (critical region) 203, 228

ЛАКЕЙ (LACKEY) 84
 ЛИСП (LISP) 34
 ЛИСП-бит (LISPbit) 36
ЛОГ (BOOL) 81
 локальный вызов процедуры (local procedure call) 206

 математическая семантика (mathematical semantics) 247
МЕДЛЧАСТН (LONGQUOT) 185
 меню (menu) 27
 меню (menu) 35
 многократный выбор (multiple choice) 98
 многопоточная обработка (multithreading) 227
МНОЖ (SET) 162
 множества готовности (ready sets) 104
 множественная пометка (multiple labelling) 84
 множественные контрольные точки (multiple checkpoints) 182
 модульность (modularity) 149
 монитор (monitor) 231
 мониторинговая система для Фортрана (FORTRAN monitor system) 215
 монотонная функция (monotonic function) 42

 наблюдаемая эквивалентность (observational equivalence) 252
 наименьшая верхняя грань (least upper bound) 90
 неструктивный (nondestructive) 93
 непересекающиеся процессы (disjoint processes) 228
 неподвижная точка (fixed point) 89
 непредваренная рекурсия (unguarded recursion) 109
 непрерывный (continuous) 90
НЕСТОП (NONSTOP) 118
 неудачи (failures) 125
НОВПАНШОН (NEWCOLLEGE) 73

 обедающие философы (dining philosophers) 69
 область (domain) 125
 общая память (shared storage) 201
ОБЩИЙ ЛАКЕЙ (SHARED LACKEY) 84
 одноэлементная последовательность (singleton) 39
 Оккам (occam) 244
 оператор приема (accept) 235
 операционная система (operating system) 196, 215
 основной процесс *см.* главный процесс
 отказы (refusals) 103
 отложенные вычисления (lazy evaluation) 36

 пакетная коммутация (packet switching) 239
 пакетная обработка (batch processing) 215
ПАНШОН (COLLEGE) 72
 параллелизм (concurrency) 63
 передатчик (transmitter) 156
 перезапуск (restart) 179
 переименование (change of symbol) 76
пер (var) 186
ПЕРЕМ (VAR) 138
 переход (transition) 29
 Пиджингол (PIDGINGOL) 169
 планирование (scheduling) 221
 повторно входимая программа (reentrant subroutine) 205
 подкачка/откачка *см.* спулинг
 подпрограмма (subroutine) 159
 подтверждение (acknowledgement) 156
 подчинение (subordination) 159
 подчиненный процесс (slave, subordinate process) 159
 полный частичный порядок (complete partial order) 90

- пометка процесса (process labeling) 81
после (after) 48
последовательная композиция (sequential composition) 172
последовательное программирование (sequential programming) 184
последовательно переиспользуемый ресурс (serially reusable resource) 204
постусловие (post-condition) 243
поточковый (data flow) 140
np (*tr*) 54
правило копирования (copy rule) 234
предваренный процесс (guarded process) 23
предваренный слева процесс (left-guarded process) 153
предел (limit) 90
предложение (sentence) 169
предусловие (precondition) 190
прерывания (interrupts) 177
префикс (prefix) 21
префикс ли (*isprefix*) 44
приемник (receiver) 156
признак конца файла (end-of-file marker) 209
приоритет (priority) 237
присваивание (assignment) 184
ПРОВОД (*WIRE*) 156
программный канал (pipe) 239
продуктивность (fairness) 98
ПРОПУСК (*SKIP*) 168
просмотр с возвратом (backtracking) 171
протокол (protocol, trace) 36, 37, 138
протокол ли (*istrace*) 48
процедура (procedure) 206
процесс (process) 20
путь (path) 47

развертка (unfolding) 23
разделение времени (timesharing) 226
разделяемая структура данных (shared data structure) 197
рандеву (rendezvous) 236
РАСПАК (*UNPACK*) 132
распределенная обработка данных (distributed processing) 204
расходимость (divergence) 109
расходится (*diverges*) 111
расширение алфавита (alphabet extension) 109
реализация (implementation) 33
реальный ресурс (actual resource) 232
регулярные выражения (regular expressions) 171
рекурсия (recursion) 22
рисунки (pictures) 29, 130

свободен (*release*) 198
свободные переменные (free variables) 54
сдвиг влево (left shift) 137
сектор (sector) 206
семафор (semaphore) 203
сеть ARPA (ARPA net) 239
символические вычисления (symbolic execution) 150
Симула 67 (*SIMULA 67*) 231
синхронизация (synchronisation) 61
система управления базой данных (data base system) 181
систолический массив (systolic array) 142
скалярное произведение (scalar product) 142
СЛИЯНИЕ (*MERGE*) 138
слой (layer) 156
слуга (footman) 73
совместно используемое устройство считывания перфокарт (shared card reader) 216
сокрытие (concealment) 105
сообщ (*message*) 129
состояние (state) 28
спецификация (specification) 54
спрячь (*hide*) 110
спулинг (spooling) 218
статическое связывание (static binding) 36
стек (stack) 134
СТОПА (*STOP_A*) 20
строгий (strict) 38, 122
структура данных (data structure) 161
структурный конфликт (structure clash) 149
сужение (restriction) 39
схема коммутационная (connection diagram) 68, 143, 148
цепление (chaining) 148

ТАДОВЕР (*VMCRED*) 26
ТАП (*VMS*) 24
ТАП2 (*VMS2*) 26
ТАС (*VMC*) 25

- ТАШИ (VMCT)** 25
ТАШУМ (NOISYVM) 64
 транспортер (pipe) 147
 транспортный протокол связей
 (transport communication proto-
 col) 138
 тупиковая ситуация *см.* дедлок
- УДВ (DOUBLE)** 131
 указание (pragma) 237
 указатель (pointer) 212
УПАК (PACK) 132
 управление потоками (flow con-
 trol) 158
 условный критический участок (con-
 ditional critical region) 228
 условный оператор (conditional)
 184
 успех (success) 168
 успешное завершение (successful
 termination) 53
- Фазовое кодирование (phase enco-
 ding)** 154
 факториал (factorial) 161
ФИБ (FIB) 137
ФИЛ (PHIL) 71
 функциональная мультипроцессор-
 ная обработка (functional multip-
 rocessing) 240
- ХАОС (CHAOS)** 122
 хвост (tail) 40
- Хемминг (Hamming) 241
- цепь (chain) 90
ЦЕПЬ2 (CHAIN2) 77
 цикл (loop) 169
 циклический процесс (cyclic pro-
 cess) 49
- частичный порядок (partial order)
 41, 90
ЧАСТН (QUOT) 185
ЧАСЫ (CLOCK) 23
 чередование (alternation, interlea-
 ving) 51, 114, 180
- экземпляр (instance) 231
- язык (language) 169
- CCS 248
 cobegin ... coend 228
 CT 33
 FCFS 223
 FIFO 223
 fork 227
 join 227
 LABEL 34
 NIL 43
 PASCAL PLUS 234
 RC4000 239
 UNIX™ 227, 239

Оглавление

От редактора перевода	5
Предисловие	7
От автора	9
Краткое содержание	14
Глава 1. Процессы	18
1.1. Введение	18
1.2. Рисунки	29
1.3. Законы	31
1.4. Реализация процессов	33
1.5. Протоколы	36
1.6. Операции над протоколами	38
1.7. Реализация протоколов	42
1.8. Протоколы процесса	44
1.9. Дальнейшие операции над протоколами	50
1.10. Спецификации	54
Глава 2. Параллельные процессы	60
2.1. Введение	60
2.2. Взаимодействие	60
2.3. Параллелизм	63
2.4. Рисунки	68
2.5. Пример: обедающие философы	69
2.6. Переименование	76
2.7. Спецификации	86
2.8. Математическая теория детерминированных процессов	87
Глава 3. Недетерминизм	95
3.1. Введение	95
3.2. Недетерминированный выбор	96
3.3. Генеральный выбор	101
3.4. Отказы	103
3.5. Сокрытие	105
3.6. Чередование	114
3.7. Спецификации	117
3.8. Расходимость	121
3.9. Математическая теория недетерминированных процессов	125
Глава 4. Взаимодействие	129
4.1. Введение	129
4.2. Ввод и вывод	130

4.3. Взаимодействия	138
4.4. Транспортёры	147
4.5. Подчинение	158
Глава 5. Последовательные процессы	168
5.1. Введение	168
5.2. Законы	172
5.3. Математическая трактовка	174
5.4. Прерывания	177
5.5. Присваивание	184
Глава 6. Разделяемые ресурсы	195
6.1. Введение	195
6.2. Поочередное использование	196
6.3. Общая память	201
6.4. Кратные ресурсы	204
6.5. Операционные системы	215
6.6. Планирование ресурсов	221
Глава 7. Обсуждение	225
7.1. Введение	225
7.2. Общая память	225
7.3. Взаимодействие	238
7.4. Математические модели	247
Избранная литература	254
Указатель символов	255
Предметный указатель	259

Научное издание

Чарльз Энтони Ричард Хоар

ВЗАИМОДЕЙСТВУЮЩИЕ ПОСЛЕДОВАТЕЛЬНЫЕ ПРОЦЕССЫ

Заведующий редакцией чл.-корр. АН СССР В. И. Арнольд

Зам. зав. редакцией А. С. Попов

Научн. ред. М. В. Хатунцева

Мл. ред. Т. Ю. Дехтярева, Т. А. Денисова, Н. С. Полякова

Художник О. С. Василькова, Художественный редактор В. И. Шаповалов

Технический редактор Е. В. Алексина

Корректор Т. П. Пашковская

ИБ № 6611

Сдано в набор 08.02.88. Подписано к печати 12.10.88. Формат 60×90¹/₁₆. Бумага книжно-журнальная. Печать высокая. Гарнитура литературная. Объем 8.25 бум. л. Усл. печ. л. 16,50. Усл. кр.-отт. 16,85. Уч.-изд. л. 14,03. Изд. № 1/5694. Тираж 12 300 экз. Заказ № 952. Цена 1 руб. 20 коп.

Издательство «Мир» В/О «Союзэкспорткнига» Государственного комитета СССР по делам издательств, полиграфии и книжной торговли. 129820, ГСП, Москва, И-110, 1-й Рижский пер., 2.

Ленинградская типография № 2 головное предприятие ордена Трудового Красного Знамени Ленинградского объединения «Техническая книга» им. Евгения Соколовой Союзполиграфпрома при Государственном комитете СССР по делам издательств, полиграфии и книжной торговли. 198052, г. Ленинград, Л-52, Измайловский проспект, 29.

I p. 20 к.

ISBN 5-03-001043-2 (русск.)
ISBN 0-13-153271-5 (англ.)