ПРЕДИСЛОВИЕ РЕДАКТОРА ПЕРЕВОДА

Программирование богато и многообразно. Ведь кажется нет такой сферы человеческой деятельности, где нельзя было бы с пользой применить вычислительную машину для оценки, информационно-справочного обслуживания, планирования, моделирования и т. п. И это многообразие задач переходит в многообразие программ, которые должны разрабатывать программисты. Они пытаются справиться с этим многообразием, "заключив" его в проблемно-ориентированные языки программирования. Языки вбирают в себя специфические черты конкретных сфер программирования — характерные структуры данных, принципы организации типичных процессов, соответствующую терминологию — и таким образом делают сам процесс программирования более универсальным. Одновременно они освобождают программистов от необходимости детализировать программы до уровня слишком мелких машинных команд и даже от необходимости знать особенности конкретных вычислительных машин. Более того, операционные системы призваны превратить вычислительные машины из предмета постоянного беспокойства в "существа", которые сами заботятся о программисте и готовы оказывать всяческие услуги ему и его программе.

И тем не менее, после того, как любая более или менее сложная задача сформулирована (пусть даже в адекватных и удобных терминах) и машина выбрана (пусть даже в самом деле готовая к всевозможным услугам), каждый программист снова и снова остается один на один со своей собственной задачей: ему нужно составить программу! Выбрать, как именно следует расположить и связать данные в памяти, понять, какая именно последовательность операторов, — способных сделать все что угодно и оттого одновременно и податливых и опасных — выполнит поставленную задачу. И как организовать операторы в цикл, который будет с каждым шагом приближать машину к намеченной цели. Выбрать, понять, изобрести, проверить, усомниться и повторить все сначала.

Показательно, что хотя в таких "внешних" разделах программирования, как языки и трансляторы, операционные системы и базы данных, широко развивается и используется теория (от математической лингвистики и логики до статистики и теории массового обслуживания), во внутренних, собственных разделах программирования господствуют умение и интуиция или в лучшем случае "полезные советы". Есть, правда, некоторые исследования, касающиеся уже готовых программ, но нет никакой теории самого процесса программирования.

Предлагаемая книга представляет собой один из первых шагов в этом направлении. В ней с каждым оператором, с их комбинациями и, в том числе, с циклами связываются преобразования предикатов, являющиеся формальным выражением их свойств, и определение свойств программы в целом превращается в задачу логического вывода. Особенно важно, что автор объясняет и демонстрирует на примерах, как этим формализмом можно пользоваться для практического программирования. При этом он, разумеется, не дает ответа на вопрос о том, как написать любую программу, — ответа на этот вопрос вообще не существует. Он предлагает нам инструмент, позволяющий проверять наши гипотезы в процессе проектирования программ, а иногда и подсказывающий те или иные варианты решений.

А это совсем не мало. Представьте себе человека, живущего в эпоху зарождения математического анализа, которому нужно находить неопределенные интегралы от весьма громоздких подынтегральных выражений и который знает только определение первообразной функции и не знает никаких приемов интегрирования. Каким для него было бы подспорьем, если бы ему сообщили о правиле интегрирования по частям или о таком приеме, как замена переменных! Его работа не перестанет быть творческой, но насколько расширятся его возможности и как много перейдет из области находок в область техники. То, что было сложным, станет простым, то, что было непосильным, станет только сложным. Систематическое использование, обобщение и расширение этих приемов позволит постепенно перейти к формализации, алгоритмизации, а затем и к автоматизации отдельных этапов решения или полного решения специальных классов задач

Автор книги, Э. Дейкстра, не нуждается в представлении,— его работы хорошо известны советским программистам. Вместе с Ч. Хоаром они недавно совершили поездку по крупнейшим городам нашей страны, во время которой выступили с лекциями перед многочисленными аудиториями.

О структуре и стиле книги достаточно полно сказано в предисловии автора. Там же он предупреждает, что читать его книгу трудно. Причина этого заключается в сложности самих программ, послуживших для нее материалом. Я хотел бы добавить, что они сложны для нас только сейчас, когда теория программирования делает свои первые шаги. Придет время, и такие программы сможет составлять (или выводить) прямо на уроке каждый школьник. И чтобы это время приблизить, надо осваивать, внедрять и развивать теорию. А этот процесс легко не проходит.

Перевод предисловия и глав 1–7 выполнен В.В. Мартынюком, глав 8–21 — И.Х. Зусман, глав 22–27 — Л.В. Уховым.

Э. З. Любимский

ПРЕДИСЛОВИЕ

Историки таких древних интеллектуальных дисциплин, как поэзия, музыка, живопись и наука, высоко оценивают роль выдающихся практиков, чьи достижения обогатили опыт и расширили представления поклонников этих дисциплин, пробудили и укрепили таланты последователей. То новое, что ими внесено, основывается на сочетании виртуозного практического мастерства и проницательного осмысливания фундаментальных принципов. Во многих случаях влияние этих людей усиливалось благодаря их высокой культуре, богатству и выразительности их речи.

В этой книге в присущем ее автору утонченном стиле представлена принципиально новая точка зрения на существо программирования. Исходя из этой точки зрения, автор разработал совокупность новых методов программирования и средств обозначения, которые демонстрируются и проверяются на многочисленных изящных и содержательных примерах. Этот труд, несомненно, будет признан одним из выдающихся достижений в разработке новой интеллектуальной дисциплины — программирования для вычислительных машин.

Ч. А. Р. Хоар

ОТ АВТОРА

Уже давно мне хотелось написать книгу такого рода. Я знал, что программы могут очаровывать глубиной своего логического изящества, но мне постоянно приходилось убеждаться, что большинство из них появляются в виде, рассчитанном на механическое исполнение, но совершенно непригодном для человеческого восприятия — где уж там говорить об изяществе. Меня не удовлетворяло и то, что алгоритмы часто публикуются в форме готовых изделий, почти без упоминания тех рассмотрений, которые проводились в процессе разработки и служили обоснованием для окончательного вида завершенной программы. Сначала я задумал изложить некоторое количество изящных алгоритмов таким способом, чтобы читатель смог прочувствовать их красоту; для этого я собирался каждый раз описывать истинный или воображаемый процесс построения, который приводил бы к получению искомой программы. Я не изменил своему первоначальному намерению в том смысле, что основой этой монографии остается длинная последовательность глав, в каждой из которых ставится и решается новая задача. Тем не менее окончательный вариант книги существенно отличается от ее первоначального замысла поскольку возложенная мною на себя обязанность преставлять решения в естественной и убедительной манере повлекла за собой гораздо большее, чем я ожидал, и я навсегда сохраню чувство благодарности судьбе за то, что взялся за эту работу.

Когда начинаешь писать подобную книгу, сразу возникает проблема: каким языком программирования пользоваться? И это не только вопрос представления! Наиболее важным, но в то же время и наиболее незаметным свойством любого инструмента является его влияние на формирование привычек людей, которые имеют обыкновение им пользоваться. Когда этот инструмент — язык программирования, его влияние, независимо от нашего желания, сказывается на нашем способе мышления. Проанализировав в свете этого влияния все известные мне языки программирования, я пришел к выводу, что ни они сами, ни их подмножества не подходят для моих целей. С другой стороны, я считал себя настолько не подготовленным к созданию нового языка программирования, что дал зарок не заниматься этим в ближайшее пятилетие, и я твердо знаю, что этот срок еще не вышел. (Прежде, помимо всего прочего, мне нужно было написать эту монографию.) Я попытался выбраться из этого тупика, создав лишь подходящий для моих целей мини-язык и включив в него только те элементы, которые представляются совершенно необходимыми и достаточно обоснованными.

Эти мои колебания и самоограничение, если их неправильно понять, могут разочаровать многих потенциальньных читателей. Наверняка разочаруются все те, кто отождествляет трудность программирования с трудностью изощренного использования громоздких и причудливых сооружений, известных под названием "языки программирования высокого уровня" или — еще хуже! — "системы программирования". Если они сочтут себя обманутыми из-за того, что я вовсе не касаюсь всех этих погремушек и свистулек, могу ответить им только одно: "А вполне ли вы уверены, что все эти погремушки и свистульки, все эти потрясающие возможности ваших, так сказать, "мощных" языков программирования имеют отношение к процессу решения, а не к самим задачам?" Мне остается лишь надеяться, что, несмотря на употребление мною мини-языка, они все же прочтут предлагаемый текст. Тогда они, возможно признают, что помимо погремушек и свистулек имеется очень богатое содержание и возникнет вопрос, стоило ли большинство из них вообще придумывать. А всем читателям, интересующимся преимущественно разработкой языков программирования, я могу только принести извинения в связи с тем, что еще не могу высказаться на эту тему более определенно. Тем временем эта монография, возможно, наведет их на некоторые размышления и поможет им избежать ошибок, которые они могли бы совершить, если бы не прочли ее.

Процесс работы над книгой, который явился для меня непрерывным источником удивления и вдохновения, привел к появлению текста, основательно отличающегося от первоначального замысла. Сначала у меня было вполне понятное намерение представить построение программ с помощью аппарата, чуть более формального чем тот, который я имел обыкновение использовать в своих вводных лекциях, где семантика обычно вводилась интуитивно, а доказательства правильности представляли собой смесь строгих рассуждений, жестикуляции и красноречия. Разрабатывая необходимые основы для более формального подхода, я обнаружил два неожиданных обстоятельства.

Первая неожиданность состояла в том, что так называемые "преобразователи предикатов", выбранные мною в качестве средства изъяснения, позволили прямо определять связь между начальным и конечным состояниями без каких-либо ссылок на промежуточные состояния, которые могут возникать во время выполнения программы. Я обрадовался тому, что это дает возможность провести четкое разграничение двух основных проблематик программирования: проблематики математической корректности (речь идет о проверке, определяет ли программа правильное соотношение между начальным и конечным состояниями — и преобразователи предикатов обеспечивают нам формальное средство для такого исследования без рассмотрения вычислительного процесса) и инженерной

проблематики эффективности (благодаря разграничению становится очевидным, что последняя проблематика определена только в связи с реализацией). Пожалуй, самое полезное открытие состоит в том, что один и тот же текст программы допускает две (в известном смысле дополняющие друг друга) интерпретации. Интерпретация в виде кода преобразователей предикатов, которая представляется более подходящей для нас, противостоит интерпретации в виде кода для выполнения — ее я предпочитаю оставлять машинам!

Второй неожиданностью оказалось то, что самые естественные и систематизированные "коды преобразователей предикатов", какие я мог себе представить, потребовали бы недетерминированной реализации, если рассматривать их как "коды для выполнения". Вначале я содрогался от мысли, что придется ввести недетерминированность уже в однопрограмном режиме (слишком хорошо мне были известны сложности, возникающие из-за этого в мультипрограммировании); однако потом я понял, что интерпретация текста как кода преобразователя предикатов имеет право на независимое существование. (Оглядываясь назад, мы можем отметить, что многие проблемы мультипрограммирования, ставившие нас прежде в тупик, являются всего лишь следствием априорной тенденции придавать детерминированности слишком большое значение.) В конце концов я пришел к тому, что стал считать недетерминированность естественной ситуацией, при этом детерминированность свелась к довольно банальному частному случаю.

Установив эти основы, я приступил, как и намеревался, к решению длинной последовательности задач. Это занятие оказалось неожиданно увлекательным. Я убедился в том, что формальный аппарат позволяет мне ухватывать существо дела гораздо четче, чем раньше. Я получил удовольствие, обнаружив, что явная постановка вопроса о завершимости может иметь большое эвристическое значение; тут я даже начал сожалеть об излишне распространенной склонности к частичной корректности. Но самое приятное состояло в том, что большинство решенных мною ранее задач теперь увенчалось более изящными решениями. Я воспринял это как весьма ободряющее свидетельство того, что разработанная методика и в самом деле улучшила мои программистские возможности.

Как следует изучать эту монографию? Лучшее, что я могу посоветовать: прерывайте чтение, как только усвоите постановку задачи, и пытайтесь сначала решить ее самостоятельно, затем продолжайте чтение. Попытка самостоятельного решения задачи представляется единственным способом прочувствовать, насколько она трудна; кроме того, вы можете сравнивать мое решение с вашим и получить удовлетворение, если ваше окажется лучше. Предупреждаю заранее: не огорчайтесь, когда увидите, что этот текст читается отнюдь не легко. Те, кто изучали его в рукописи, часто испытывали затруднения (но вполне вознаграждались за это). Впрочем, каждый при анализе их затруднений мы совместно убеждались в том, что "виновным" оказывался вовсе не текст (т. е. способ изложения), а сам излагаемый материал. Мораль этого может быть только такова: нетривиальный алгоритм и вправду нетривиален, а его окончательная запись на языке программирования слишком лаконична по сравнению с рассуждениями, обосновывающими его разработку; эта краткость окончательного текста не должна нас дезориентировать. Один из моих сотрудников внес предложение (а я довожу его до вашего сведения, поскольку оно может оказаться полезным), чтобы небольшие группы студентов изучали книгу вместе. (Здесь я должен добавить в скобках замечание по поводу "трудности" этого текста. Посвятив немало лет своей научной жизни тому чтобы прояснить задачи программиста и сделать их более подвластными нашему интеллекту, я обнаружил с удивлением (и раздражением), что мое стремление внести ясность приводит к систематическим обвинениям в том, что я "внес в программирование трудности". Но эти трудности всегда в нем были; и только сделав их видимыми, мы сможем надеяться, что научимся разрабатывать программы с высокой степенью надежности, а не просто "лепить команды", т. е. выдавать тексты, основанные на неубедительных предположениях, состоятельность которых может выявиться после первого же противоречащего примера. Незачем и говорить, что ни одна программа из этой монографии не проверялась на машине.)

Я должен объяснить читателю, почему я пользуюсь мини-языком, столь ограниченным, что в нем нет даже процедур и рекурсии. Поскольку каждое следующее расширение языка добавляло бы к этой книге еще несколько глав, тем самым соответственно увеличивая ее стоимость, отсутствие большинства возможных расширений (таких, как, например, мультипрограммирование) не нуждается в дополнительных оправданиях. Однако процедуры всегда занимали такое важное место, а рекурсия для вычислительной науки в такой степени считалась признаком академической респектабельности, что некоторое разъяснение представляется необходимым.

Прежде всего эта книга предназначена не для начинающих, и я рассчитываю, что мои читатели уже знакомы с указанными понятиями. Во-вторых, книга не является вводным текстом по какому-то конкретному языку программирования, так что отсутствие в ней этих конструкций и примеров их употребления не следует объяснять моей неспособностью или нежеланием ими пользоваться или же воспринимать как намек на то, что вообще лучше воздерживаться от их применения. Просто они не

потребовались мне для разъяснения моей главной мысли о том, насколько существенно тщательное разграничение проблематик для разработки всесторонне высококачественных программ; скромные средства мини-языка предоставляют нам более чем достаточный простор для нетривиальных и в то же время вполне приемлемых разработок.

Можно обойтись этим объяснением, но оно все же не является исчерпывающим. Я все равно считал себя обязанным ввести повторение как самостоятельную конструкцию, поскольку мне представлялось, что это следовало сделать уже давно. Когда языки программирования зарождались, "динамическая" природа оператора присваивания казалась не очень приспособленной к "статической" природе традиционной математики. Из-за отсутствия соответствующей теории математики ощущали некоторые затруднения, связанные с этим оператором, а поскольку именно конструкция повторения создает необходимость в присваиваниях переменным, математики ощущали затруднения и в связи с повторением. Когда были разработаны языки без присваивания и без повторений — такие, как чистый ЛИСП,— многие почувствовали значительное облегчение. Они снова ощутили под ногами знакомую почву и увидели проблеск надежды превратить программирование в занятие с твердой и солидной математической основой. (До сего времени среди склонных к теоретизированию специалистов по машинной математике все еще широко распространено мнение, что рекурсивные программы "более естественны", чем программы с повторениями.)

Другого выхода из положения путем надежного и действенного математического обоснования пары понятий "повторение" и "присваивание переменной" нам предстояло ждать еще десять лет. А выход, как показано в этой монографии, заключался в том, что семантику конструкции повторения можно описать с помощью рекуррентных отношений между предикатами, тогда как для описания семантики общей рекурсии требуются рекуррентные отношения между преобразователями предикатов. Отсюда совершенно очевидно, почему я считаю общую рекурсию на порядок более сложной конструкцией, чем простое повторение; и поэтому мне больно смотреть, как семантику конструкции повторения

"**while** B **do** S"

определяют как семантику обращения

"$whiledo(B, S)$"

к рекурсивной процедуре (описанной в синтаксисе языка АЛГОЛ 60):

procedure *whiledo*(*условие*, *оператор*);
begin if *условие* **then begin** *оператор*;
whiledo(*условие*,*оператор*) **end**
end

Несмотря на формальную правильность, это мне неприятно, потому, что я не люблю, когда из пушки стреляют по воробьям, вне зависимости от того, насколько эффективно пушка справляется с такой работой. Для поколения теоретиков машинной математики, которые подключались к этой тематике в течение шестидесятых годов, приведенное выше рекурсивное определение часто является не только "естественным", но даже "самым правильным". Однако ввиду того, что без понятия повторения мы не можем даже описать поведение машины Тьюринга, представляется необходимым произвести некоторое восстановление равновесия.

По поводу отсутствия библиографии я не предлагаю ни объяснений, ни извинений.

Благодарности. Следующие лица оказали непосредственное влияние на разработку этой книги, либо приняв участие в обсуждении ее предполагаемого содержания, либо высказав замечання относительно готовой рукописи или ее частей: К. Брон, Р. Берсталл, У. Фейен, Ч. Хоар, Д. Кнут, М. Рем, Дж. Рейнольдс, Д. Росс, К. Шолтен, Г. Зигмюллер, Н. Вирт и М. Вуджер. Я считаю честью для себя возможность публично выразить им мою признательность за сотрудничество. Кроме того, я весьма обязан корпорации Burroughs, создавшей мне благоприятные условия и предоставившей необходимые средства, и благодарен моей жене за неизменную поддержку и одобрение.

Э. В. Дейкстра

Нейен
Нидерланды

0 АБСТРАКЦИЯ ИСПОЛНЕНИЯ

Абстракция исполнения лежит в основе всего понятия "алгоритма" настолько глубоко, что обычно ее считают само собой разумеющейся и оставляют без внимания. Ее назначение в том, чтобы сопоставлять между собой различные вычисления. Иначе говоря, она предоставляет нам способ осмысливания конкретного вычисления как элемента большого класса различных вычислений; мы можем отвлечься от взаимных отличий элементов такого класса и, руководствуясь определением класса в целом, высказывать утверждения, применимые к каждому его элементу, а следовательно, справедливые и для конкретного вычисления, которое мы хотим рассматривать.

Чтобы разъяснить, что подразумевается под "вычислением", я опишу сейчас невычислительную конструкцию "получения" (я преднамеренно не употребляю термина "вычисление"), например, наибольшего общего делителя чисел 111 и 259. Она состоит из двух картонных карточек, расположенных одна поверх другой. На верхней карточке написан текст "НОД(111, 259) =". Чтобы получить от конструкции ответ, мы поднимаем верхнюю карточку и кладем ее слева от нижней, на которой теперь можно прочесть текст "37".

Простота карточной конструкции является большим достоинством, но она омрачается двумя недостатками — мелким и крупным. Мелкий недостаток состоит в том, что хотя эту конструкцию можно в самом деле использовать для получения наибольшего общего делителя чисел 111 и 259, но помимо этого она мало на что пригодна. Однако крупный недостаток в том, что, как бы тщательно мы ни проверяли устройство конструкции, наша вера в то, что она вырабатывает правильный ответ, может основываться только на нашем доверии к ее создателю: он мог ошибиться либо при проектировании своей машины, либо при изготовлении нашего конкретного экземпляра.

Чтобы преодолеть меньшее затруднение, мы могли бы рассмотреть изображение на огромном листе картона большого прямоугольного массива из сетевых точек с целыми координатами x и y, удовлетворяющими отношениям $0 \le x \le 500$ и $0 \le y \le 500$. Для каждой такой точки (x, y) с положительными координатами (т. е. за исключением точек на осях) мы можем выписать в соответствующей позиции значение НОД(x, y); предлагается двумерная таблица из 250 000 элементов. С точки зрения полезности это значительное усовершенствование: вместо конструкции, способной выдавать наибольший общий делитель единичной пары чисел, мы имеем теперь "конструкцию", способную выдавать наибольший общий делитель любой пары из 250 000 различных пар чисел. Это много, но особенно радоваться нечему, так как указанный ранее второй недостаток (почему мы должны верить, что конструкция выдает правильный ответ?) помножился на те же самые 250 000, и теперь от нас требуется уже совсем безграничное доверие к ее изготовителю. Поэтому перейдем к рассмотрению другой конструкции. На таком же листе картона с сетевыми точками написаны только числа, пробегающие значения от 1 до 500 вдоль обеих осей. Кроме того, начерчены следующие прямые линии:

1) вертикальные линии (с уравнением $x =$ *константа*);

2) горизонтальные линии (с уравнением $y =$ *константа*);

3) диагонали (с уравнением $x + y =$ *константа*);

4) "линия ответа" с уравнением $x = y$.

Чтобы работать на этой машине, мы должны следовать следующим инструкциям ("играть по следующим правилам"). Когда хотим найти наибольший общий делитель двух чисел X и Y, мы помещаем фишку — также поставляемую изготовителем — в сетевую точку с координатами $x = X$ и $y = Y$. Коль скоро фишка не находится на "линии ответа", рассматриваем наименьший равнобедренный прямоугольный треугольник, у которого вершина прямого угла совпадает с фишкой, а один из концов гипотенузы (либо ниже фишки, либо слева от нее) находится на одной из осей. (Поскольку фишка не лежит на линии ответа, такой прямоугольный треугольник будет иметь на осях только одну вершину.) Затем фишка перемещается в сетевую точку, совпадающую с другим концом гипотенузы. Такое перемещение повторяется до тех пор, пока фишка не достигнет линии ответа. После этого x-координата (или y-координата) окончательного положения фишки является искомым ответом.

Как нам убедиться в том, что эта машина будет выдавать правильный результат? Если (x, y) — любая из 249 500 точек не на линии ответа и $(x'y')$ — точка, в которую передвинется фишка за один шаг игры, то либо $x' = x$ и $y' = y - x$, либо $x' = x - y$ и $y' = y$. Нетрудно *доказать*, что НОД(x, y) = НОД(x', y'). Важный момент здесь состоит в том, что *одно и то же* рассуждение применяется одинаково верно к любому из возможных шагов! Кроме того, — и опять-таки без труда — мы можем *доказать* для любой точки (x, y) где $x = y$ (т.е. (x, y) является одной из 500 точек на линии ответа), что НОД$(x, y) = x$. И снова важный момент в том, что *одинаковое* рассуждение применимо к *любой* из 500 точек на линии ответа. В-третьих,— и вновь это не составит труда — нам нужно показать, что при любом исходном положении (X, Y) конечное число шагов в самом деле перенесет фишку на линию ответа, и опять важно отметить, что одно и то же рассуждение одинаково применимо к любому из 250 000 исходных положений (X, Y). Три простых рассуждения, пространность которых не зависит от числа сетевых то-

чек: эта миниатюра показывает, насколько велики возможности математики. Если обозначить через (x, y) произвольное положение фишки на протяжении игры, начатой в положении (X, Y), то первая наша теорема позволяет утверждать, что во время этой игры отношение НОД(x, y) = НОД(X, Y) будет всегда справедливо, или — выражаясь на соответствующем жаргоне — "оно сохраняет инвариантность". Далее, вторая торема гласит, что мы можем интерпретировать x-координату окончательного положения фишки как требуемый ответ, а третья теорема гласит, что такое окончательное положение существует (т. е. будет достигнуто за конечное число шагов) И этим завершается анализ того, что мы могли бы назвать "нашей абстрактной машиной".

Теперь нам остается убедиться в том, что лист, поступивший от изготовителя, является на самом деле правильной моделью абстрактной машины. Для этого нужно проверить нумерацию вдоль обеих осей, а также проверить, правильно ли проведены все прямые линии. Это несколько затруднительно, так как предстоит исследовать объекты, число которых пропорционально N, где N (в нашем примере 500) — длина стороны квадрата, но все же предпочтительнее, чем N^2, число возможных вариантов вычисления.

Другая машина могла бы работать не с огромным листом картона, а с двумя девятибитовыми регистрами, в каждом из которых можно запомнить двоичное число от 0 до 500. При этом мы могли бы использовать один регистр для запоминания значения x-координаты, а другой — для запоминания значения y-координаты, соответствующей "текущему положению фишки". Перемещение тогда соответствует уменьшению содержимого одного регистра на содержимое другого. Мы могли бы реализовать арифметику самостоятельно, но, разумеется, лучше, если машина сможет делать это за нас. Если мы захотим полагаться на полученный ответ, то нам нужно уметь убеждаться в том, что машина правильно выполняет операции сравнения и вычитания. В уменьшенном масштабе повторяется та же история: мы выводим единожды и на все случаи, т.е для любой пары n-разрядных двоичных чисел, уравнения для устройства двоичного вычитания, а затем удостоверяемся в том, что наша физическая машина правильно моделирует это абстрактное устройство. Если это устройство параллельного вычитания, то число проверок — пропорциональное числу элементов и их взаимосвязей — пропорционально значению $n = \log_2 N$. В последовательной машине сделан еще один шаг на пути упрощения оборудования за счет расхода времени.

Теперь я попытаюсь, хоть бы для своего собственного просвещения, уловить основной смысл приведенного примера.

Вместо того чтобы рассматривать одиночную проблему вычисления НОД$(111, 259)$, мы обобщили ее и подошли как к частному случаю более широкого класса проблем вычисления НОД(X, Y), Стоит отметить, что мы могли бы обобщить проблему вычисления НОД$(111, 259)$ по-разному: можно было бы рассматривать эту задачу как частный случай иного, более широкого класса задач, например вычисления НОД$(111, 259)$, НОК$(111, 259)$, 111×259, $111 + 259$, $111/259$, $111 - 259$, 111^{259}, дня недели для 111-го дня 259-го года нашей эры и т. д. В результате мог бы появиться "процессор для 111 и 259", и для того, чтобы он выдал упомянутый выше ответ, нам следовало бы дать на его вход команду "НОД, пожалуйста". Вместо этого мы предложили "НОД-вычислитель", которому для получения этого ответа потребуется задать на вход пару чисел "111, 259" и это совсем другая машина!

Другими словами, когда требуется выработать один или несколько результатов, обычная практика состоит в том, чтобы обобщить проблему и рассматривать эти результаты как частные случаи некоего более широкого класса. Однако мало радости, если ограничиться утверждением, что любой предмет является частным случаем чего-то более общего. Если мы хотим следовать этому подходу, то на нас возлагаются две обязанности:

1. Мы должны иметь полную ясность относительно способа обобщения, т. е. должны тщательно выбрать и явно определить более широкий класс, поскольку наши рассуждения должны применяться ко всему этому классу.
2. Мы должны выбрать ("изобрести", если вам угодно) такое обобщение, которое окажется полезным для наших целей.

В нашем примере я, разумеется, отдаю предпочтение "НОД-вычислителю", а не "процессору для 111 и 259", и сравнение этих двух конструкций дает нам намек на то, какие характеристики делают обобщение "полезным для наших целей". Машина, которая по команде может вырабатывать в качестве ответа значения всех видов забавных функций от 111 и 259, становится все более неудобной для проверки по мере того, как растет набор функций. В этом явный контраст с нашим "НОД-вычислителем".

НОД-вычислитель был бы столь же плох, если бы он представлял собой таблицу из 250 000 записей, содержащих "заготовленные" ответы. Его характерное отличие в том, что он может быть задан в форме компактного набора "правил игры", который, если играть в соответствии с этими правилами, обеспечит выдачу нужного ответа.

Огромный выигрыш состоит в том, что единое рассуждение применительно к этим правилам позволяет нам доказывать существенные утверждения о результатах любого варианта игры. Это достигается ценой того, что при каждом из 250 000 конкретных применений этих правил мы получаем ответ "не сразу": каждый раз игра должна быть сыграна в соответствии с правилами!

Тот факт, что мы в состоянии дать столь компактную формулировку правил игры, когда единое рассуждение позволяет нам выводить заключения о любых вариантах игры, непосредственно связан с систематизированным расположением 250 000 узловых точек. Мы оказались бы беспомощными, если бы лист картона содержал беспорядочный случайный разброс точек, исключающий какую-либо систематизацию. В нашем случае мы могли бы разделить свою фишку на две половинки и двигать одну половинку вниз, пока она не ляжет на горизонтальную ось, а другую половинку влево, пока она не ляжет на вертикальную ось. Вместо того чтобы воспроизводить с одной фишкой 250 000 возможных положений, мы могли добиться того же с двумя фишками, для каждой из которых нужно только 500 возможных положений, т. е. всего 1000 положений в общей сложности. Мы бы достигли того же уровня в 250 000 позиций, используя то обстоятельство, что любое из 500 положений одной половинки фишки может комбинироваться с любым из 500 положений другой половинки: число положений неразделенной фишки равно произведению числа положений одной половинки на число положений другой. На принятом жаргоне мы говорим, что "общее пространство состояний рассматривается как декартово произведение пространств состояний переменных x и y".

Возможность замены одной фишки с двумерной свободой выбора положения на две половинки с одномерной свободой используется в предложенной выше двухрегистровой машине. С точки зрения технической реализации это представляется весьма заманчивым: требуется только построить регистры, способные различать 500 разных случаев ("значений"), а за счет простого объединения этих двух регистров общее число разных случаев возводится в квадрат! Это перемножительное правило позволяет нам различать громадное число возможных общих состояний с помощью ограниченного числа компонентов, у каждого из которых только ограниченное число возможных состояний. По мере добавления таких компонентов размер пространства состояний возрастает экспоненциально, но нам следует иметь в виду, что это допустимо только при условии, что обоснование нашего нововведения остается весьма компактным; если такое обоснование тоже возрастает экспоненциально, то вообще нет никакого смысла создавать такую машину.

Замечание. Убедительную иллюстрацию к сказанному выше можно найти в изобретении, возраст которого уже превысил десять веков: в десятичной системе счисления! Она обладает тем поистине пленительным свойством, что число необходимых цифр возрастает всего лишь пропорционально логарифму максимального из чисел, которые должны быть представлены. Двоичная система счисления — это то, что получается, когда вы забываете, что на каждой руке имеется по пяти пальцев. *(Конец замечания.)*

Выше мы занимались одним аспектом множественности, а именно большим числом позиций фишки (= возможных состояний). Имеется еще одна аналогичная множественность, а именно большое число различных игр (= вычислений), которые могут состояться в соответствии с нашими правилалами игры: по одной игре для каждого начального положения, если говорить точнее. Наши правила игры являются очень общими в том смысле, что они применимы к любому начальному положению. Но мы настаиваем на компактности обоснования правил игры, а это означает, что и сами правила должны быть компактными. В нашем примере это достигалось следующим приемом: вместо перечисления "сделай это, сделай то" мы задали правила игры в виде правил для выполнения "шага" в сочетании с критерием того, должен ли "шаг" быть выполнен в следующий раз. (На самом деле шаг должен повторяться, пока не будет достигнуто состояние, в котором он становится неопределенным.) Иначе говоря, даже всю игру от начала до конца можно произвести с помощью повторяемых применений одного и того же "подправила".

Это очень продуктивный прием. Один алгоритм заключает в себе проект определенного класса вычислений, которые могут выполняться под его управлением; благодаря условному повторению "шага", вычисления из такого класса могут иметь различные протяженности. Этим объясняется, как короткий алгоритм может занимать машину в течение длительного времени. С другой стороны, в этом можно усмотреть начальный намек на то, зачем нам могут понадобиться особенно быстрые машины.

Меня завораживает мысль, что эта глава могла писаться еще в времена, когда Евклид мог смотреть на нее из-за моего плеча.

1 Роль языков программирования

В главе "Абстракция исполнения" я дал неформальные описания различных "машин", предназначенных для вычисления наибольшего общего делителя двух положительных (и не слишком больших) чисел. Одно описание было выражено с помощью фишки, передвигаемой по листу картона, другое — с помощью двух половинок фишки, каждая из которых двигалась вдоль своей оси, а последнее — с помощью двух регистров, каждый из которых мог содержать целое число (не превышающее некоторой границы). Физически эти три "машины" весьма различны, однако математически они очень сходны: основная часть доказательства их способности вычислять наибольший общий делитель совпадает для всех трех машин. Это объясняется тем, что они представляют всего лишь различные воплощения одного и того же набора "правил игры", и это именно тот набор правил, который ставляет сущность реального изобретения, известного как "алгоритм Евклида".

В предыдущей главе алгоритм Евклида описывался словесно, не совсем формальным образом. Однако мы отметили, что число соответствующих ему возможных вычислений столь велико, что нужно доказать его корректность. Коль скоро алгоритм дан только не формально, он не является вполне подходящим объектом для формального рассмотрения. Для дальнейшего нам потребуется описание этого алгоритма в некоторой удобной формальной записи.

Такая формальная запись может обладать многочисленными преимуществами. Любой способ записи подразумевает, что всякий описываемый с его помощью предмет задается как конкретный представитель (часто бесконечного) класса объектов, которые могут быть описаны этим способом. Наш спсоб описания должен, разумеется, обеспечивать изящное и точное описание алгоритма Евклида, но когда мы этого достигнем, алгоритм Евклида окажется в действительности заданным, как представитель огромного класса всех видов алгоритмов. И в описаниях некоторых из этих других алгоритмов мы сможем найти более интересные применения нашего способа записи. В случае алгоритма Евклида есть основания утверждать, что он настолько прост, что можно обойтись его неформальным описанием. Сила формальной записи должна проявиться в таких достижениях, которых без нее мы никогда не смогли бы добиться!

Второе преимущество формального способа записи состоит в том, что он дает нам возможность изучать алгоритмы как математические объекты; при этом формальное описание алгоритма служит основой, позволяющей нам интеллектуально охватить этот алгоритм. Благодаря этому мы сумеем доказывать теоремы о классах алгоритмов, например пользуясь тем, что их описания обладают некоторым общим структурным свойством.

Наконец, такой способ записи позволит нам описывать алгоритмы настолько точно, что если будет задан описанный так алгоритм и заданы значения аргументов (вход), то не будет никаких сомнений относительно того, какими должны быть соответствующие ответы (выход). При этом можно считать, что вычисление выполняется автоматом, который, получив (формально описанный) алгоритм и аргументы, порождает ответы без дальнейшего вмешательства человека. Такие автоматы, способные обеспечивать взаимное воздействие алгоритма и аргумента, и в самом деле были построены. Они называются "автоматическими вычислителями". Алгоритмы, предназначенные для автоматического выполнения такими вычислителями, называются "программами", и с конца пятидесятых годов формальные способы, используемые для записи программ, называются "языками программирования". (Введение термина "язык" применительно к способам записи программ вызывает смешанные чувства. С одной стороны, оно оказалось весьма полезным, поскольку существующая теория лингвистики обеспечила естественную основу и устоявшуюся терминологию ("грамматика", "синтаксис", "семантика" и т. д.) для рассмотрений применительно к этой новой тематике. С другой стороны, необходимо отметить, что аналогия с (ныне именуемыми так) "естественными языками" часто оказывалась дезориентирующей, так как для естественных языков, неформальных по существу, источником как их слабости, так и их силы является присущая им неопределенность и неточность).

В историческом плане этот последний аспект, т. е. возможность использования языков программирования в качестве средства общения с существующими автоматическими вычислителями, в течение долгого времени рассматривался как их наиболее важное свойство. Эффективность, с которой существующие автоматические вычислители могли бы выполнять программы, записанные на конкретном языке, становилась главным критерием качества этого языка. Как огорчительное следствие мы нередко обнаруживаем, что аномалии существующих вычислительных машин старательно воспроизводятся в языках программирования, причем это происходит в ущерб интеллектуальной управляемости программ, выражаемых на таком языке (как будто программирование и без этих аномалий не было уже достаточно трудным!). В рамках нашего подхода мы постараемся восстановить равновесие и поэтому будем относиться к возможности фактического выполнения наших алгоритмов вычислительной машиной только как к счастливой случайности, которая не должна занимать центрального места в наших рассмотрениях. (В одном недавно опубликованном учебном тексте по

программированию на PL/1 можно найти настойчивый совет избегать обращений к процедурам, насколько это возможно, "потому что они делают программу столь неэффективной". В свете того, что процедуры в языке PL/1 являются одним из основных средств описания структуры, этот совет выглядит ужасно, настолько ужасно, что вряд ли можно называть упомянутый текст "учебным". Если вы убеждены в полезности понятия процедуры и соприкасаетесь с приложениями, где дополнительные затраты на аппарат процедур обходятся слишком дорого, то порицайте эти неподходящие приложения, а не возводите их на уровень стандартов. Равновесие и в самом деле надлежит восстановить!)

Я рассматриваю язык программирования преимущественно как средство для описания (потенциально весьма сложных) абстрактных конструкций. Как показано в главе "Абстракция исполнения", первейшим достоинством алгоритма является потенциальная компактность рассуждений, на которых может основываться наше проникновение в его сущность. Как только эта компактность потеряна, алгоритм в значительной мере теряет "право на существование", и поэтому мы будем стремиться к сохранению такой компактности. Соответственно и наш выбор языка программирования будет подчинен той же цели.

2. СОСТОЯНИЯ И ИХ ХАРАКТЕРИСТИКА

В течение многих столетий человек оперирует натуральными числами. Мне представляется, что в доисторические времена, когда перед нашими предками впервые забрезжило понятие числа, они изобретали индивидуальные имена для каждого числа, на которое у них обнаруживалась потребность сослаться. Им приходилось иметь имена для чисел точно так же, как мы имеем имена "один, два, три, четыре и так далее".

Это поистине "имена" в том смысле, что при исследовании последовательности "один, два, три" никакое правило не поможет нам умозаключить, что следующим именем будет "четыре". Вы, конечно же, должны это *знать*. (В том возрасте, когда я вполне удовлетворительно умел считать — по-голландски,— мне пришлось учиться счету по-английски и во время экзамена самое изощренное знание слов "семь" и "девять" не смогло помочь мне установить, как писать слово "восемь", не говоря уж о том, как произносить его.)

Очевидно, что такое несистематизированное разнообразие позволяет нам выделять индивидуально только весьма ограниченное количество различных чисел. Чтобы избежать подобного ограничения, каждый язык в цивилизованном мире вводит (более или менее) систематизированное именование натуральных чисел, и обучение счету в значительной степени сводится к обнаружению системы, лежащей в основе этого именования. Когда ребенок учится считать до тысячи, ему не приходится заучивать наизусть эту тысячу имен (в их точном порядке!); он руководствуется определенными правилами: наступает момент, когда ребенок узнает, как переходить от любого числа к следующему, например от "четыреста двадцать девять" к "четыреста тридцать".

Легкость обращения с числами сильно зависит от выбранной нами для них систематизации. Гораздо труднее установить, что

$$\text{дюжина дюжин} = \text{гросс}$$

$$\text{одиннадцать плюс двенадцать} = \text{двадцать три}$$

$$XLVII + IV = LI$$

чем установить, что

$$12 \times 12 = 144$$

$$11 + 12 = 23$$

$$47 + 4 = 51$$

так как последние три ответа можно получить, применив простой набор правил, которыми может руководствоваться любой восьмилетний ребенок.

При механических манипуляциях над числами преимущества десятичной системы счисления проявляются гораздо более отчетливо. Уже в течение столетий мы располагаем механическими сумматорами, показывающими ответ в окошке, позади которого имеется несколько колес с десятью различными положениями, причем каждое колесо в любом своем положении показывает одну десятичную цифру. (Теперь уже уже не является проблемой показать "00000019", прибавить 4 и затем показать "00000023"; было бы проблемой — по крайней мере если пользоваться чисто механическими средствами — показать вместо этого "девятнадцать" и "двадцать три".)

Существенное свойство такого колеса состоит в том, что оно обладает десятью различными устойчивыми положениями. Терминологически это выражается разными способами. Например, колесо называется "переменной с десятью значениями", и если мы хотим большей точности, то даже перечисляем эти значения: от 0 до 9. Здесь каждое "положение" колеса отождествляется со "значением" переменной. Колесо называется "переменной", потому что, хотя его положения и устойчивы, колесо может повернуться в другое положение: "значение" может измениться. (Я вынужден отметить, что этот термин неудачен по крайней мере в двух отношениях. Во-первых, такое колесо, которое (почти) всегда находится в одном из своих десяти положений и, следовательно, (почти) всегда представляет некое значение, является понятием, весьма отличным от того, что математики называют "переменной", поскольку, как правило, математическая переменная вообще не представляет какого-либо конкретного значения. Если мы говорим, что для любого целого числа n утверждение $n^2 \geq 0$ истинно, то здесь n является переменной совсем другого рода. Во-вторых, в нашем контексте мы используем термин "переменная" для того, что существует во времени и чье значение, если его не трогать, остается постоянным! Термин "изменяемая константа" подошел бы лучше, но мы не станем вводить его, а будем придерживаться прочно установившейся традиции.)

Другой способ терминологически отразить сущность такого колеса, которое (почти) всегда находится в одном из десяти различных положений или "состояний", состоит в том, чтобы поставить этому колесу в соответствие "пространство состояний из десяти точек". Здесь каждое состояние (положение) связывается с "точкой", а собрание этих точек называется — в соответствии с математической

традицией — пространством или, если мы хотим большей точности, "пространством состояний". Вместо того чтобы говорить, что переменная обладает (почти) всегда одним из своих возможных значений, мы можем теперь выразить то же, сказав, что система, состоящая из единственной переменной, находится (почти) всегда в одной из точек своего пространства состояний. Пространство состояний описывает степень свободы системы; больше ей некуда переходить.

Отвлечемся теперь от отдельно взятого колеса и сосредоточим наше внимание на регистре с восемью такими колесами в строке. Поскольку каждое из этих колес находится в одном из десяти различных состояний, этот регистр, рассмотренный как целое, находится в одном из 100 000 000 возможных различных состояний, каждое из которых естественно идентифицируется числом (а точнее, строкой из восьми цифр), показываемым в окошке.

Если заданы состояния всех колес, то состояние регистра в целом однозначно определено; и обратно, по любому состоянию регистра в целом однозначно определяется состояние каждого отдельного колеса. В этом случае мы говорим (в предыдущей главе нами уже использован этот термин), что получаем (или строим) пространство состояний регистра в целом, формируя "декартово произведение" пространств состояний восьми отдельных колес. Общее число точек в этом пространстве состояний является произведением чисел точек в тех пространствах состояний, из которых оно построено (именно поэтому оно называется декартовым *произведением*).

В зависимости от того, что нас интересует в этом предмете, мы либо рассматриваем такой регистр как единую переменную с 10^8 различными возможными значениями, либо как составную переменную, составленную из восьми различных переменных, называемых "колесами", с десятью значениями. Если мы интересуемся только показываемыми значениями, то будем, по-видимому, рассматривать регистр как неделимое понятие, тогда как инженер-эксплуатационник, которому нужно заменить колесо со сломанным зубцом, будет, конечно рассматривать регистр как составной объект.

Мы наблюдали другой пример построения пространства состояний как декартова произведения более мелких пространств состояний, когда обсуждали алгоритм Евклида и обнаружили, что положение фишки где-то на листе можно полностью идентифицировать с помощью двух полуфишек, каждая из которых находится где-то на своей оси, т. е. с помощью комбинации (а точнее, упорядоченной пары) двух переменых "x" и "y". (Идея идентификации положения точки на плоскости значениями ее координат x и y идет от Декарта, который разработал аналитическую геометрию и в честь которого названо декартово произведение.) Фишка на листе была введена для демонстрации того факта, что развивающийся вычислительный процесс — такой, как выполнение алгоритма Евклида,— можно рассматривать как систему, движущуюся по своему пространству состояний. В соответствии с этой метафорой начальное состояние именуется также "исходной точкой".

В этой книге мы будем преимущественно — а может быть, даже исключительно — заниматься системами, пространства состояний которых будут в конечном итоге рассматриваться как некие декартовы произведения. Разумеется, не следует усматривать в этом мое мнение, будто бы пространства состояний, построенные с помощью декартовых произведений, являются общим и окончательным решением всех наших проблем; я отлично знаю, что это не так. Из дальнейшего станет очевидным, почему они заслуживают столь большого нашего внимания и в то же время почему это понятие играет столь главенствующую роль во многих языках программирования.

Прежде чем продолжить рассуждения, отмечу одну проблему, с которой нам придется столкнуться. Когда мы образуем пространство состояний, строя декартово произведение, то вовсе не обязательно, что нам окажутся полезными все его точки. Типичным примером этому является характеристика дней данного года с помощью пары (месяц, день), где "месяц" — переменная с 12 значениями (от "янв" до "дек"), а "день" — переменная с 31 значениями (от 1 до 31). В этом случае мы образуем пространство состояний из 372 точек, тогда как никакой год не содержит более чем 366 дней. Что нам делать, скажем, с парой (июн, 31)? Либо мы запрещаем ее, вводя понятие "невозможных дат" и тем самым создавая в некотором смысле возможность внутренних противоречий в системе, либо мы разрешаем ее как другой вариант имени для одного из "истинных" дней, например, приравнивая ее к (июл,1). Феномен "неиспользуемых точек пространства состояний" возникает всякий раз, когда число различных возможных значений, которые мы желаем распознавать, оказывается простым числом.

Систематизация, которая автоматически вводится, когда мы строим пространство состояний как декартово произведение, позволяет нам идентифицировать отдельно взятую точку. Например, я могу утверждать, что мой день рождения — это (май, 11). Однако благодаря Декарту нам известен теперь и другой способ фиксации этого факта: мой день рождения приходится на ту дату (месяц, день),

которая соответствует решению уравнения[1]

$$(\textit{месяц} = \textit{май})\ \textbf{and}\ (\textit{день} = 11)$$

Приведенное выше уравнение имеет только одно решение и поэтому представляет собой несколько усложненный способ указания этого единственного дня в году. Однако преимущество использования уравнения состоит в том, что оно позволяет нам охарактеризовать множество всех его решений, и такое множество может содержать значительно больше чем одну точку. Тривиальным примером может служить определение рождества:

$$(\textit{месяц} = \textit{дек})\ \textbf{and}\ ((\textit{день} = 25)\ \textbf{or}\ (\textit{день} = 26))$$

Более примечательным примером является определение множества дат, в которые выплачивается мое ежемесячное жалованье:

$$(\textit{день} = 23)$$

и разумеется, это гораздо более компактное описание, чем перечисление вида "(янв, 23), (фев, 23), (мар, 23)" и т. д.

Из сказанного выше явствует, что степень легкости, с которой мы используем такие уравнения для характеристики множеств состояний, зависит от того, насколько множества, которые мы хотим охарактеризовать, "соответствуют" структуре пространства состояний, т. е. "соответствуют" введенной системе координат. Например, в рассмотренной выше системе координат было бы несколько неудобно охарактеризовать множество дней, которые приходятся на тот же день недели, что и (янв, 1). Во многих случаях программисту приходится при решении задач вводить пространства состояний с подходящими для его целей системами координат; причем дополнительные требования часто вынуждают его вводить также пространства состояний, в которых число точек во много раз больше, чем число различных возможных значений, которые он должен распознавать.

Другой пример использования уравнения для характеристики множества состояний встречался в нашем описании картонной машины для вычисления НОД(X, Y), где

$$x = y$$

характеризует все точки того, что мы назвали "линией ответа"; это множество конечных состояний, т. е. вычисление прекращается тогда и только тогда, когда достигнуто какое-нибудь состояние, удовлетворяющее уравнению $x = y$.

Помимо координат пространства состояний, т. е. переменных, в терминах значений которых выражается ход вычислительного процесса, мы встречали в наших уравнениях константы (такие, как "май" или "23"). Кроме того, можно пользоваться так называемыми "свободными переменными", которые можно представить себе как "неспецифицированные константы". Мы применяем их специально для обозначения связей различных состояний, которые могут встречаться на последовательных шагах *одного и того же* вычислительного процесса. Например, во время некоего конкретного выполнения алгоритма Евклида с начальной точкой (X, Y) все состояния (x, y) будут удовлетворять формуле

$$\text{НОД}(x, y) = \text{НОД}(X, Y)\ \textbf{and}\ 0 < x \leq X\ \textbf{and}\ 0 < y \leq Y$$

Здесь X и Y не являются такими переменными, как x и y. Они представляют собой "их начальные значения". Это константы с точки зрения конкретного вычисления, но они не специфицированы в том смысле, что мы могли бы запустить алгоритм Евклида с любой точкой сетки в качестве начального положения нашей фишки.

В заключение некоторые терминологические замечания. Я буду называть такие уравнения "условиями" или "предикатами". (Я мог бы, а возможно, и должен бы различать эти понятия, зарезервировав термин "предикат" для формального выражения, обозначающего "условие". Тогда мы имели бы возможность сказать, например, что два разных предиката "$x = y$" и "$y = x$" обозначают одно и то же условие. Однако, зная себя, я не надеюсь преуспеть в такой манере изложения.) Я буду считать синонимичными такие выражения, как "состояние, для которого предикат истинен", "состояние, которое удовлетворяет условию", "состояние, в котором условие удовлетворяется", "состояние, в котором условие справедливо" и т. д. Если система заведомо достигнет состояния, удовлетворяющего условию P, то мы будем говорить, что система заведомо "обеспечит истинность P".

---

[1] Здесь и далее в математических формулах используются для обозначения логических операций символы **and** (конъюнкция), **or** (дизъюнкция) и **non** (отрицание).— *Прим. перев.*

Предполагается, что всякий предикат определен в каждой точке рассматриваемого пространства состояний: в каждой точке значением предиката является либо "истина", либо "ложь", и предикат служит для характеристики множества всех точек, для которых (или в которых) этот предикат истинен.

Мы будем называть два предиката P и Q равными (формально "$P = Q$"), если они обозначают одно и то же условие, т.е характеризуют одно и то же множество состояний.

Два предиката будут играть особую роль, и мы резервируем для них имена "T" и "F".

T —это предикат, который истинен во всех точках рассматриваемого пространства состояний; ему соответствует полное множество состояний.

F — предикат, который ложен во всех точках пространства состояний; он соответствует пустому множеству.

3 ХАРАКТЕРИСТИКА СЕМАНТИКИ

Нас интересуют преимущественно такие системы, которые, приступив к работе в некоем "начальном состоянии", завершат ее в каком-то "конечном состоянии", которое, как правило, зависит от выбора начального состояния. Этот подход несколько отличается от концепции автомата с конечным числом состояний, который, с одной стороны, воспринимает поток входных символов, а с другой стороны, порождает поток выходных символов. Чтобы перевести это в нашу схему, мы должны предположить, что значение входа (т. е. аргумент) отражается в выборе начального состояния и что значение выхода (т. е. ответ) отражается в выборе конечного состояния. Наш подход избавляет нас от всевозможных непринципиальных сложностей.

Первая часть этой главы посвящена почти исключительно так называемым "детерминированным машинам", тогда как во второй части (которую можно пропустить при первом чтении) речь идет о так называемых "недетерминированных машинах". Различие между этими двумя понятиями состоит в том, что для детерминированной машины событие, которое произойдет после запуска конструкции, полностью определяется ее начальным состоянием. Если она запускается дважды в одинаковых начальных состояниях, то произойдут одинаковые события: поведение детерминированной машины полностью воспроизводимо. Для недетерминированной машины, напротив, запуск в заданном начальном состоянии приведет к какому-то одному событию из класса возможных событий; начальное состояние только фиксирует этот класс в целом.

Теперь я предполагаю, что проектирование такой системы является целенаправленной деятельностью, т. е. мы желаем достичь чего-то с помощью этой системы. Например, если мы хотим создать машину, способную вычислять наибольший общий делитель, то можем потребовать, чтобы конечное состояние удовлетворяло условию

$$x = \text{НОД}(X, Y) \tag{1}$$

В рассмотренной ранее машине мы будем иметь также $y = \text{НОД}(X, Y)$, потому что игра заканчивается при $x = y$, но это отнюдь не часть наших требований, когда мы решаем принять конечное значение x в качестве ответа.

Мы называем условие (1) (желаемым) "постусловием" — "пост", потому что оно налагается на то состояние, в котором система должна оказаться *после* своей работы. Заметим, что постусловие может удовлетворяться многими возможными состояниями. В таком случае мы естественно считаем каждое из них одинаково удовлетворительным, так что нет оснований требовать, чтобы конечное состояние было однозначной функцией от начального состояния. (Читатель увидит из дальнейшего, что именно здесь проявляется потенциальная полезность недетерминированного устройства.)

Для того чтобы использовать эту машину, когда мы хотим получить от нее ответ (например, хотим "чтобы она достигла конечного состояния, удовлетворяющего постусловию (1) для заданного набора значений X и Y"), нам желательно знать множество соответствующих начальных состояний, а более точно, множество таких начальных состояний, при которых запуск обязательно приведет к событию правильного завершения причем система останется в конечном состоянии, удовлетворяющем постусловию. Если мы можем привести систему без вычислительных усилий в одно из таких состояний, то мы уже знаем, как использовать эту систему для получения желаемого ответа. Приведем пример для евклидовой игры на картоне: мы можем гарантировать конечное состояние, удовлетворяющее постусловию (1) для любого начального состояния, удовлетворяющего условию

$$\text{НОД}(x, y) = \text{НОД}(X, Y) \textbf{ and } 0 < x \leq 500 \textbf{ and } 0 < y \leq 500 \tag{2}$$

(Верхние границы добавлены, чтобы учесть ограниченный размер листа картона. Если мы начинаем с парой (X, Y), такой, что $\text{НОД}(X, Y) = 713$, то не существует пары (x, y), удовлетворяющей условию (2), т. е. для таких значений X и Y условие (2) сводится к предикату F, а это означает, что рассматриваемая машина не может быть использована для вычисления $\text{НОД}(X, Y)$ применительно к этой паре значений X и Y).

При многих комбинациях (X, Y) многие состояния удовлетворяют условию (2). В случае $0 < X \leq 500$ и $0 < Y \leq 500$ *тривиальным* выбором является $x = X$ и $y = Y$. Этот выбор может быть произведен без какого-либо вычисления функции НОД и даже без учета того факта, что это симметричная функция от своих аргументов.

Условие, характеризующее множество всех начальных состояний, при которых запуск обязательно приведет к событию правильного завершения, причем система останется в конечном состоянии, удовлетворяющем заданному постусловию, называется "слабейшим предусловием, соответствующим этому постусловию". (Мы называем его "слабейшим", поскольку, чем слабее условие, тем больше состояний удовлетворяют ему, а мы стремимся здесь охарактеризовать все возможные начальные состояния, которые неизбежно приведут к желаемому конечному состоянию.)

Если система (машина, конструкция) обозначается через S, а желаемое постусловие — через R, то соответствующее слабейшее предусловие мы обозначим[1]

$$\mathrm{wp}(S, R)$$

Если начальное состояние удовлетворяет $\mathrm{wp}(S, R)$, то конструкция обязательно обеспечит в конце концов истинность R. Поскольку $\mathrm{wp}(S, R)$ — это слабейшее предусловие, мы знаем также, что если начальное состояние не удовлетворяет $\mathrm{wp}(S, R)$, то такой гарантии дать нельзя, т. е. ход событий может привести к завершению работы в конечном состоянии, не удовлетворяющем R, а может даже и вообще помешать достижению какого бы то ни было конечного состояния (как мы увидим далее, либо потому, что система оказывается вовлеченной в выполнение незавершимого задания, либо потому, что система попадает в тупик).

Мы считаем, что нам достаточно хорошо известно, как работает конструкция S, если можем вывести для любого постусловия R соответствующее слабейшее предусловие $\mathrm{wp}(S, R)$, поскольку тем самым мы уловили, что́ эта конструкция способна сделать для нас; это называется "ее семантикой".

Здесь уместны два замечания. Во-первых, множество возможных постусловий, вообще говоря, столь огромно, что эта информация в табличной форме (т. е. в виде таблицы, где каждому постусловию R соответствует запись, в которой содержится соответствующее предусловие $\mathrm{wp}(S, R)$), оказалась бы совершенно необозримой, а следовательно, бесполезной. Поэтому определение семантики той или иной конструкции всегда дается другим способом — в виде правила, описывающего, как для любого заданного постусловия R можно вывести соответствующее слабейшее предусловие $\mathrm{wp}(S, R)$). Для фиксированной конструкции S такое правило, которое по заданному предикату R, означающему постусловие, вырабатывает предикат $\mathrm{wp}(S, R)$, означающий соответствующее слабейшее предусловие, называется "преобразователем предикатов". Когда мы просим, чтобы нам сообщили описание семантики конструкции S, то в сущности речь идет о соответствующем этой конструкции преобразователе предикатов.

Во-вторых,— и я чувствую искушение добавить "благодаря судьбе" — часто нас не интересует полная семантика конструкции. Это объясняется тем, что мы стремимся использовать конструкцию S только для конкретной надобности, а точнее, для того, чтобы обеспечить истинность весьма конкретного постусловия R, ради которого производилась разработка конструкции. И даже применительно к этому конкретному постусловию R нас часто не интересует точный вид $\mathrm{wp}(S, R)$; зачастую мы удовлетворяемся более сильным условием P, т. е. таким условием, для которого можно показать, что утверждение

$$P \Rightarrow \mathrm{wp}(S, R) \qquad \text{для всех состояний} \tag{3}$$

справедливо. (Предикат "$P \Rightarrow Q$" (читается "из P следует Q") ложен только в тех точках пространства состояний, где условие P справедливо, а Q не справедливо; во всех остальных точках он истинен. Требуя, чтобы утверждение "$P \Rightarrow \mathrm{wp}(S, R)$" было справедливо для всех состояний, мы тем самым требуем, чтобы всякий раз когда P — истина, условие $\mathrm{wp}(S, R)$ тоже было истиной; тогда P — достаточное предусловие. В теоретико-множественной терминологии это означает, что множество состояний, характеризуемое условием P, является подмножеством того множества состояний, которое характеризуется условием $\mathrm{wp}(S, R)$.) Если для заданных P, S и R отношение (3) выполняется, это часто можно доказать, не прибегая к точной формулировке,— или, если вам так больше нравится, "вычислению" или "выводу" — предиката $\mathrm{wp}(S, R)$. И это отрадное обстоятельство, поскольку, за исключением тривиальных случаев, следует ожидать, что явная формулировка условия $\mathrm{wp}(S, R)$ превысит по крайней мере резерв нашей писчей бумаги, нашего терпения или нашей (аналитической) изобретательности (или какую-то комбинацию этих резервов).

Смысл $\mathrm{wp}(S, R)$, т. е. "слабейшего предусловия для начального состояния, при котором запуск обязательно приведет к событию правильного завершения, причем система S останется в конечном состоянии, удовлетворяющем постусловию R" позволяет нам установить, что преобразователь предикатов, рассматриваемый как функция от постусловия R, обладает рядом определенных свойств.

Свойство 1. Для любой конструкции S мы имеем

$$\mathrm{wp}(S, F) = F \tag{4}$$

Предположим, что оказалось не так; при этом нашлось хоть одно состояние, удовлетворяющее условию $\mathrm{wp}(S, F)$. Возьмем такое состояние в качестве начального состояния для конструкции S. Тогда, согласно нашему определению, запуск привел бы к событию правильного завершения, причем система осталась бы в конечном состоянии, удовлетворяющем предикату F. Но здесь возникает

[1] По начальным буквам английских слов weakest pre-condition.— Прим. перев.

противоречие, так как по определению предиката F не существует состояний, удовлетворяющих F; тем самым доказано отношение (4). Свойство 1 известно также под названием "закон исключенного чуда".

Свойство 2. Для любой конструкции S и любых постусловий Q и R, таких, что

$$Q \Rightarrow R \qquad \text{для всех состояний}$$

справедливо также отношение

$$\text{wp}(S, Q) \Rightarrow \text{wp}(S, R) \qquad \text{для всех состояний} \tag{6}$$

В самом деле, при любом начальном состоянии, удовлетворяющем wp(S, Q), согласно определению, по окончании работы системы будет обеспечена истинность Q. С учетом отношения (5) тем самым будет обеспечена и истинность R. Следовательно, данное начальное состояние будет одновременно удовлетворять и условию wp(S, R), как записано в (6). Свойство 2 — это свойство монотонности.

Свойство 3. Для любой конструкции S и для любых постусловий Q и R справедливо

$$(\text{wp}(S, Q) \textbf{ and } \text{wp}(S, R)) = \text{wp}(S, Q \textbf{ and } R) \tag{7}$$

В любой точке пространства состояний из левой части отношения (7) логически следует его правая часть потому, что для любого начального состояния, удовлетворяющего wp(S, Q), и wp(S, R), мы знаем в совокупности, что установится конечное состояние, удовлетворяющее и Q, и R. Далее, поскольку, в силу определения,

$$(Q \textbf{ and } R) \Rightarrow Q \qquad \text{для всех состояний}$$

свойство 2 позволяет нам заключить, что

$$\text{wp}(S, Q \textbf{ and } R) \Rightarrow \text{wp}(S, Q) \qquad \text{для всех состояний}$$

и аналогично

$$\text{wp}(S, Q \textbf{ and } R) \Rightarrow \text{wp}(S, R) \qquad \text{для всех состояний}$$

Однако из $A \Rightarrow B$ и $A \Rightarrow C$, согласно исчислению предикатов, можно вывести, что $A \Rightarrow (B \textbf{ and } C)$. Поэтому правая часть отношения (7) логически следует из его левой части в любой точке пространства состояний. Поскольку обе части всюду следуют друг из друга, они должны быть равны, а тем самым доказано свойство 3.

Свойство 4. Для любой конструкции S и любых постусловий Q и R справедливо

$$(\text{wp}(S, Q) \textbf{ or } \text{wp}(S, R)) \Rightarrow \text{wp}(S, Q \textbf{ or } R) \qquad \text{для всех состояний} \tag{8}$$

Поскольку, в силу определения,

$$Q \Rightarrow (Q \textbf{ or } R) \qquad \text{для всех состояний}$$

свойство 2 позволяет нам прийти к заключению, что

$$\text{wp}(S, Q) \Rightarrow \text{wp}(S, Q \textbf{ or } R) \qquad \text{для всех состояний}$$

и аналогично

$$\text{wp}(S, R) \Rightarrow \text{wp}(S, Q \textbf{ or } R) \qquad \text{для всех состояний}$$

Но, согласно исчислению высказываний, из $A \Rightarrow C$ и $B \Rightarrow C$ можно заключить, что $(A \textbf{ or } B) \Rightarrow C$, и этим доказана справедливость соотношения (8). Вообще говоря, следование в обратном направлении не имеет места. Так про женщину, которая ждет ребенка, неверно утверждение, что она родит обязательно сына и неверно утверждение, что она родит обязательно дочь; однако с абсолютной уверенностью можно утверждать, что она родит сына или дочь. Впрочем, для детерминированных конструкций справедливо следующее, более сильное свойство.

Свойство 4'. Для любой детерминированной конструкции S и любых постусловий Q и R справедливо

$$(\text{wp}(S, Q) \textbf{ or } \text{wp}(S, R)) \Rightarrow \text{wp}(S, Q \textbf{ or } R)$$

Нам нужно показать, что следование выполняется и в левую сторону. Рассмотрим какое-то начальное состояние, удовлетворяющее wp(S, Q **or** R), этому начальному состоянию соответствует *единственное* конечное состояние, удовлетворяющее либо Q, либо R, либо обоим условиям; следовательно, данное начальное состояние должно удовлетворять либо wp(S, Q), либо wp(S, R), либо обоим условиям соответственно, т. е. он должно удовлетворять (wp(S, Q) **or** wp(S, R)). И этим доказано свойство 4'.

В этой книге я буду — и это может оказаться одной из ее отличительных особенностей — рассматривать недетерминированность как правило, а детерминированность как исключение: детерминированная машина будет рассматриваться как частный случай недетерминированной, как конструкция, для которой справедливо свойство 4' а не только более слабое свойство 4. Такое решение отражает коренное изменение в моем мировоззрении. В 1958 г. я был одним из первых, кто разрабатывал базовое программное обеспечение для машины с прерываниями ввода-вывода, и невоспроизводимость поведения такой во всех отношениях недетерминированной машины явилась горестным обстоятельством. Когда впервые была предложена идея прерываний ввода-вывода, меня настолько пугала сама мысль о необходимости разрабатывать надежное программное обеспечение для такого неукротимого зверя, что я оттягивал принятие решения о допущении таких прерываний по крайней мере в течение трех месяцев. И даже после того, как я сдался (мое сопротивление сломили лестью), чувствовал я себя весьма неуютно: ведь ошибка в программе способна вызвать несуразное поведение системы, столь сходное с невоспроизводимым машинным сбоем. Кроме того,— и это в то время, когда для детерминированных машин мы все еще полагались на "отладку",— с самого начала было совершенно очевидно, что тестирование программ оказывалось теперь совсем непригодным средством для достижения должного уровня надежности.

В течение многих лет впоследствии я относился к невоспроизводимости поведения недетерминированной машины как к добавочному осложнению, которого следует избегать любыми способами. Прерывания были для меня не чем иным, как злыми кознями инженеров по аппаратуре против бедных разработчиков программного обеспечения. Из этого моего страха родилась дисциплина "гармонически взаимодействующих последовательных процессов". Несмотря на ее успех, мои опасения сохранялись, поскольку наши решения — хоть бы и с доказанной правильностью — представлялись частными решениями проблемы "укрощения" (именно так мы это воспринимали!) конкретных видов недетерминированности. Основанием для моего страха было отсутствие общей методологии.

С тех пор два обстоятельства изменили картину. Первое возникло с пониманием того, что даже в случае полностью детерминированных машин полезность тестирования программ оказывается сомнительной. Как я уже много раз говорил и во многих местах писал, тестирование программы может вполне эффективно служить для демонстрации наличия в ней ошибок, но, к сожалению, непригодно для доказательства их отсутствия. Другим прояснившимся тем временем обстоятельством явилось обнаружение необходимости того, чтобы всякая дисциплина проектирования должным образом учитывала тот факт, что само проектирование конструкции, предназначенной для какой-то цели, тоже должно быть целенаправленной деятельностью. В нашем конкретном случае это означает естественность предположения, что отправной точкой для проектных рассмотрений будет служить заданное постусловие. В каком-то смысле мы будем "работать вспять". Действуя так, мы обнаружим, что весьма существенным оказывается следование из свойства 4, тогда как от равенства из свойства 4' мы получим очень мало пользы.

Как только мы разработали математический аппарат, позволяющий проектировать недетерминированные конструкции, достигающие цели, недетерминированная машина перестает нас пугать. Напротив! Мы научимся даже ценить ее как важную степень на пути разработки проекта в конечном итоге полностью детерминированной конструкции.

(Последующая часть этой главы может быть пропущена при первом чтении.) Ранее мы договорились, что представляем функционирование конструкции S достаточно хорошо, если только знаем, как соответствующий ей преобразователь предикатов wp(S, R) действует над любым постусловием R. Если нам известно также, что данная конструкция детерминирована, то знание этого преобразователя предикатов полностью определяет возможное поведение конструкции. Для детерминированной конструкции S и некоторого постусловия R всякое начальное состояние попадает в одно из трех непересекающихся множеств в соответствии со следующими взаимно исключающими возможностями:

(а) Запуск конструкции S приведет к конечному состоянию, удовлетворяющему R.

(б) Запуск конструкции S приведет к конечному состоянию, удовлетворяющему **non** R.

(в) Запуск конструкции S не приведет к какому бы то ни было конечному состоянию, т. е. работа не сможет завершиться правильным образом.

Первое множество характеризуется выражением wp(S, R), второе множество — выражением wp(S, **non** R), их объединение — выражением (wp(S, R) **or** wp(S, **non** R)) = wp(S, R **or non** R) = wp(S, T) а следовательно, третье множество характеризуется выражением **non** wp(S, T).

Труднее дать полную семантическую характеристику недетерминированной системы. По отношению к любому заданному постусловню R мы снова имеем три возможных типа событий, как перечислено выше в пунктах (а), (б) и (в). Однако в случае недетерминированной системы одно и то же начальное состояние не обязательно приводит к единственному событию, которое по определению относилось бы к одному из трех взаимно исключающих типов; для любого начального состояния возможные события могут теперь относиться к одному, двум или даже ко всем трем типам.

Чтобы описывать такие ситуации, мы можем использовать понятие "свободного предусловия". Ранее мы рассмотривали такие предусловия, при которых гарантировалась достижимость "правильного результата", т. е. конечного состояния, удовлетворяющего R. Свободное предусловие является более слабым: оно гарантирует только, что система не выдаст неправильного результата, т. е. не достигнет такого конечного состояння, которое не удовлетворяло бы R, однако не исключается возможность незавершения работы системы. Аналогично и для свободных предусловий мы можем ввести понятие "слабейшего свободного предусловия"; обозначим его через wlp(S, R), При этом пространство начальных состояний подразделяется в принципе на семь взаимно непересекающихся областей, ни одна из которых не обязана быть пустой. (Их семь, потому что из трех объектов можно сделать семь непустых выборок.) Все они легко характеризуются тремя предикатами: wlp(S, R), wlp(S, **non** R) и wp(S, T).

(а) $$\text{wp}(S, R) = (\text{wlp}(S, R)\ \textbf{and}\ \text{wp}(S, T))$$

Запуск обеспечит истинность **R**.

(б) $$\text{wp}(S, \textbf{non}\ R) = (\text{wlp}(S, \textbf{non}\ R)\ \textbf{and}\ \text{wp}(S, T))$$

Запуск обеспечит истинность **non** R.

(в) $$\text{wlp}(S, F) = (\text{wlp}(S, R)\ \textbf{and}\ \text{wlp}(S, \textbf{non}\ R))$$

Запуск не сможет привести к правильно завершаемой работе

(аб) $$\text{wp}(S, T)\ \textbf{and non}\ \text{wlp}(S, R)\ \textbf{and non}\ \text{wlp}(S, \textbf{non}\ R)$$

Запуск приведет к завершаемой работе, но по начальному стоянию нельзя определить, будет ли конечное состояние удовлетворять условию R или нет.

(ав) $$\text{wlp}(S, R)\ \textbf{and non}\ \text{wp}(S, T)$$

Если запуск приводит к какому-то конечному состоянию, такое конечное состояние будет удовлетворять R, но по начальному состоянию нельзя определить, завершится ли работа системы.

(бв) $$\text{wlp}(S, \textbf{non}\ R)\ \textbf{and non}\ \text{wp}(S, T)$$

Если запуск приводит к какому-то конечному состоянию, такое конечное состояние не будет удовлетворять R, но по начальному состоянию нельзя определить, завершится ли работа системы.

(абв) $$\textbf{non}(\text{wlp}(S, R)\ \textbf{or}\ \text{wlp}(S, \textbf{non}\ R)\ \textbf{or}\ \text{wp}(S, T)$$

По начальному состоянию нельзя определить, приведет ли запуск к завершаемой работе, а в случае завершаемости нельзя определить, будет ли удовлетворяться условие R.

Последние четыре возможности существуют только для недетермированных машин.

Из определения $wlp(S, R)$ следует, что

$$\text{wlp}(S, T) = T$$

Ясно также, что

$$(\text{wlp}(S, F)\ \textbf{and}\ \text{wp}(S, T)) = F$$

Если бы было не так, то существовало бы начальное состояние, для которого можно было бы гарантировать и завершимость, и незавершимость работы. На рис. 3.1 дано графическое

//рис.31

представление пространства состояний,причем внутренние части прямоугольников удовлетворяют wlp(S, R), wlp($S,$ **non** R) и wp(S, T) соответственно.

Этот анализ приведен для полноты, а также потому, что на практике понятие свободного предусловия оказывается весьма полезным. Если, например, кто-то реализует язык программирования, то он не станет доказывать, что при этой реализации любая правильная программа выполняется правильно. Он был бы удовлетворен и счастлив, если бы твердо знал, что никакая правильная программа не будет выполнена правильно без соответствующего предупреждения,— разумеется, при условии, что класс программ, которые на самом деле будут выполняться правильно, достаточно велик для того, чтобы эта реализация представляла какой-то интерес.

Однако пока мы не будем рассматривать понятие свободного предусловия, а сосредоточим свое внимание на способе характеристики тех начальных состояний, которые гарантируют получение правильного результата. Когда такой способ будет разработан, мы посмотрим, как можно его видоизменить таким образом, чтобы это позволило нам рассуждать о свободных предусловиях в той мере, в которой они нас интересуют.

4 СЕМАНТИЧЕСКАЯ ХАРАКТЕРИСТИКА ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

В предыдущей главе мы выдвинули тезис, что знаем семантику конструкции S достаточно хорошо, если знаем ее "преобразователь предикатов", т. е. правило, указывающее нам, как вывести по любому постусловию R соответствующее слабейшее предусловие (которое мы обозначили через $\mathrm{wp}(S, R)$), характеризующее те начальные состояния, при которых запуск приведет к событию правильного завершения, причем система останется в конечном состоянии, удовлетворяющем постусловию R. Вопрос в том, как выводить $\mathrm{wp}(S, R)$ для заданных S и R.

Оставим пока вопрос об одиночной конкретной конструкции S. Программа, написанная на хорошо определенном языке программирования, может рассматриваться как некая конструкция, такая конструкция, которую мы знаем достаточно хорошо, если знаем соответствующий преобразователь предикатов однако язык программирования полезен только при том условии, что его можно применять для записи многих различных программ, и для всех этих программ нам желательно знать соответствующие им преобразователи предикатов.

Каждая такая программа задается своим текстом на хорошо определенном языке программирования, и поэтому ее текст должен служить для нас отправной точкой. Но теперь перед нами неожиданно открываются два совершенно различных назначения такого текста программы. С одной стороны, текст программы предназначен для *машинной* интерпретации, если мы хотим, чтобы она могла выполняться автоматически, если мы хотим, чтобы по ней для нас был произведен какой-либо конкретный расчет. С другой стороны, желательно, чтобы текст программы говорил *нам* о том, как строить соответствующий преобразователь предикатов, как производить преобразование предикатов, чтобы выводить предусловие $\mathrm{wp}(S, R)$ для любого данного постусловия R, которое нас почему-либо заинтересовало. Это замечание позволяет понять, что подразумевается под "хорошо определенным языком программирования" с *нашей* точки зрения. Когда семантика конкретной конструкции (или программы) задается ее преобразователем предикатов, мы рассматриваем семантическую характеристику языка программирования как набор правил, которые позволяют любой программе, написанной па этом языке, поставить в соответствие преобразователь предикатов. С такой точки зрения мы можем рассматривать программу как "код" для соответствующего преобразователя предикатов.

При желании можно подойти к проблеме проектирования языка программирования с такой позиции. Можно руководствоваться (довольно формально) тем, что правила построения преобразователей предикатов должны быть такими, чтобы, применяя их, нельзя было построить ничего другого, кроме как преобразователя предикатов, обладающего свойствами 1–4 из предыдущей главы. В самом деле, если правила не дают такой гарантии, то это означает, что вы деформируете предикаты таким образом, что они уже не могут интерпретироваться как постусловия и соответствующие слабейшне предусловия.

Сразу напрашиваются два примера весьма простых преобразователей предикатов, которые обладают требуемыми свойствами.

Начнем с тождественного преобразования, т. е, с конструкции S, такой, что для любого постусловия R мы имеем $\mathrm{wp}(S, R) = R$. Эту конструкцию знают и любят все программисты: она известна им под названием "пустой оператор", и в своих программах они часто используют ее, оставляя пропуск в том месте текста, где синтаксически требуется какой-то оператор. Это не слишком похвальный прием (компилятор должен знать, что он "видит" этот оператор, на том основании, что он ничего не видит, и поэтому мы дадим этой конструкции наименование, скажем, "*пропустить*". Итак, семантика оператора "*пропустить*" определяется следующим образом:

$$\mathrm{wp}(\textit{пропустить}, R) = R \qquad \text{для любого постусловия } R$$

(Как и все, я буду пользоваться термином "оператор", поскольку он прочно вошел в жаргон. Когда люди сообразили, что "команда" могла бы оказаться более подходящим термином, было уже слишком поздно[1].)

Замечание. Тем, кто считает расточительством символов введение явного имени "*пропустить*" для пустого оператора, когда "пусто" столь красноречиво выражает его семантику, следует осознать, что десятичная система счисления оказалась возможной только благодаря введению символа "0" для понятия "ничто". *(Конец замечания.)*

Прежде чем продолжить наши рассуждения, мне хотелось бы не упустить возможность отметить, что тем временем мы уже определили некий язык программирования. Остается добавить только одно:

---

[1] По традиции мы переводим английский термин "statement" (утверждение, предложение) термином "оператор", введенным в программирование А. А. Ляпуновым, и таким образом, русский читатель оказывается в более выгодном положении, чем английский.—*Прим. ред.*

это однооператорпый язык, в котором можно описать только одну конструкцию, причем единственное, что способна сделать для нас данная конструкция, это "оставить все, как есть" (или "ничего не делать", но такое негативное употребление языка представляет опасность, см. следующий абзац).

Другой простой преобразователь предикатов приводит к постоянному слабейшему предусловию, которое вовсе не зависит от постусловия R. Мы имеем два предиката-константы, T и F. Конструкция S, для которой $\mathrm{wp}(S, R) = T$ при всех R, не может существовать, потому что она нарушила бы закон исключенного чуда. Однако конструкция S, для которой $\mathrm{wp}(S, R) = F$ при всех R, обладает преобразователем предикатов, удовлетворяющим всем необходимым свойствам. Этому оператору мы тоже присвоим имя, назовем его "*отказать*". Итак, семантика оператора "*отказать*" задается следующим образом:

$$\mathrm{wp}(\textit{отказать}, R) = F \qquad \text{для любого постусловия R}$$

Этот оператор не может даже "ничего не делать" в смысле "оставить все, как есть"; он вообще ни на что не способен. Если мы полагаем $R = T$, т. е. не накладываем на конечное состояние никаких дополнительных требований, кроме самого факта его существования, даже тогда не найдется соответствующего начального состояния. Если запустить конструкцию по имени "*отказать*", она не сможет достичь конечного состояния: сама попытка ее запуска является гарантией неудачи. (Нас не должно занимать теперь (а также и впоследствии) то, что позднее мы введем структуру операторов, в которой содержатся как частные случаи семантические эквиваленты для "*пропустить*" и "*отказать*".)

Теперь мы располагаем неким (все еще весьма зачаточным) двухоператорным языком программирования, в котором можем описать две конструкции; одна из них ничего не делает, а вторая всегда терпит неудачу. Со времени опубликования знаменитого "Собщения об алгоритмическом языке АЛГОЛ 60" в 1960 г. никакой уважающий себя ученый, занимающийся программированием, не позволит себе обойтись на этом этапе без формального определения синтаксиса столь далеко продвинутого языка программирования в системе обозначений, называемой "НФБ" (сокращение от "Нормальная форма Бэкуса"), а именно:

$$\langle \text{оператор} \rangle ::= \textit{пропустить} | \textit{отказать}$$

(Читается так: "Элемент синтаксической категории, именуемой "оператор" (именно это обозначают забавные скобки "$<$" и "$>$"), определяется как (это обозначает знак "::=") "*пропустить*" или (это обозначает вертикальная черта "|") "*отказать*". Колоссально! Но не беспокойтесь; более впечатляющие применения НФБ в качестве способа записи последуют в надлежащее время.)

Один класс безусловно более интересных преобразователей предикатов основывается на подстановке, т. е. на замене всех вхождений некоей переменной в формальном выражении для постусловия R на (одно и то же) "что-нибудь другое". Если в предикате R все вхождения переменной x заменяются некоторым выражением (E), то мы обозначаем результат этого преобразования через $R_{E \to X}$. Теперь мы можем рассмотреть для заданных x и E такую конструкцию, чтобы для любых постусловий R получалось $\mathrm{wp}(S, R) = R_{E \to X}$; здесь x — "координатная переменная" нашего пространства состояний, а E — выражение соответствующего типа.

Замечание. Такое преобразование путем подстановки удовлетворяет свойствам 1–4 из предыдущей главы. Мы не станем пытаться демонстрировать это и предоставим самому читателю решать в зависимости от своего вкуса, будет ли он относиться к этому как к тривиальному или же как к глубокому математическому результату. *(Конец замечания.)*

Таким способом вводится целый класс преобразователей предикатов, целый класс конструкций. Они обозначаются оператором, который называется "оператор присваивания"; такой оператор должен определять три вещи:

1) наименование переменной, подлежащей замене;

2) тот факт, что правило, соответствующее преобразованию предикатов, есть подстановка;

3) выражение, которым должно заменяться всякое вхождение этой переменной в постусловии.

Если переменная x должна быть заменена выражением (E) то обычная запись такого оператора выглядит следующим образом:

$$x := E$$

(где так называемый знак присваивания ":=" следует читать как "становится").

Сказанное можно суммировать, определив

$$\mathrm{wp}("x := E", R) = R_{E \to X} \qquad \text{Для любого постусловия } R$$

При желании это можно рассматривать как семантическое определение оператора присваивания для любой координатной переменной x и любого выражения E соответствующего типа. Наслаждаясь,

по нашему обыкновению, употреблением НФБ, мы можем расширить наш формальный синтаксис, чтобы он читался так:

⟨оператор⟩ ::= *пропустить*|*отказать*|⟨оператор присваивания⟩
⟨оператор присваивания⟩ ::= ⟨переменная⟩ := ⟨выражение⟩

причем последнюю строчку следует читать так: "Элемент синтаксической категории, именуемой "оператор присваивания", определяется как элемент синтаксической категории, именуемой "переменная", за которым следуют сначала знак присваивания ":=", а затем элемент синтаксической категории, именуемой "выражение". Перед тем как продолжить рассуждение, представляется разумным убедиться в том, что этим формальным описанием оператора присваивания в самом деле охватывается наше интуитивное понимание оператора присваивания — если оно у нас есть! Рассмотрим пространство состояний с двумя целыми координатными переменными "a" и "b". Тогда

$$\mathrm{wp}(\text{"}a := 7\text{"}, a = 7) = \{7 = 7\}$$

и, поскольку логическое выражение в правой части является истиной для всех значений a и b, т. е. для всех точек пространства состояний, мы можем упрощенно записать это так:

$$\mathrm{wp}(\text{"}a := 7\text{"}, a = 7) = T$$

Иначе говоря, всякое начальное состояние гарантирует, что присваивание "$a := 7$" обеспечит истинность "$a = 7$". Аналогично

$$\mathrm{wp}(\text{"}a := 7\text{"}, a = 6) = \{7 = 6\}$$

и, поскольку это логическое выражение является ложью для всех значений a и b, мы получаем

$$\mathrm{wp}(\text{"}a := 7\text{"}, a = 6) = F$$

Это означает, что не существует никакого начального состояния, для которого мы могли бы гарантировать, что присваивание "$a := 7$" обеспечит истинность "$a = 6$". (Это находится в сооветствии с нашим предыдущим результатом, что все начальные состояния обеспечат конечную истинность "$a = 7$", а следовательно, конечную ложность "$a \neq 7$".) Далее,

$$\mathrm{wp}(\text{"}a := 7\text{"}, b = b0) = \{b = b0\}$$

т. е. если мы хотим гарантировать, что после присваивания "$a := 7$" переменная b имеет некоторое значение $b0$, то нужно, чтобы b имела это значение еще в начальном состоянии. Другими словами, все переменные, за исключением "a", не изменяются, они сохраняют те значения, которые имели раньше; присваиванне "$a := 7$" перемещает в пространстве состояний точку, соответствующую текущему состоянию системы, параллельно оси a так, что обеспечивается конечное выполнение равенства "$a = 7$".

Вместо того чтобы брать в качестве выражения E константу, мы могли бы взять какую-то функцию от начального состояния. Это иллюстрируется следующими примерами:

$$\mathrm{wp}(\text{"}a := 2 * b + 1\text{"}, a = 13) = \{2 * b + 1 = 13\} = \{b = 6\}$$
$$\mathrm{wp}(\text{"}a := a + 1\text{"}, a > 10) = \{a + 1 > 10\} = \{a > 9\}$$
$$\mathrm{wp}(\text{"}a := a - b\text{"}, a > b) = \{a - b > b\} = \{a > 2 * b\}$$

Возникает небольшое осложнение, если мы разрешаем выражению E быть частично определенной функцией начачального состояния, т. е. такой функцией, попытка вычисления которой для начального состояния, не входящего в область определения, не приведет к правильно завершаемой работе. Если мы хотим предусмотреть и эту ситуацию, то нам нужно уточнить наше определение семантики оператора присваиваивания и записать

$$\mathrm{wp}(\text{"}x := E\text{"}, R) = \{D(E)\ \mathbf{cand}\ R_{E \to X}\}$$

Здесь предикат $D(E)$ означает "в области определения E"; логическое выражение "$B1\ \mathbf{cand}\ B2$" (так называемая "условная конъюнкция") имеет то же значение, что и "$B1\ \mathbf{and}\ B2$", если оба операнда определены, но помимо этого по определению оно имеет значение "ложь", если $B1$ является "ложью"; последнее справедливо вне зависимости от того, определен ли операнд $B2$. Обычно условие $D(E)$ не

подчеркивается явно либо потому, что оно=T, либо потому, что мы исходим из предположения, что оператор присваивания никогда не будет запущен в начальных состояниях, не принадлежащих области определения выражения E.

Естественным обобщением оператора присваивания является излюбленное некоторыми программистами так называемое "одновременное присваивание". В этом случае возможна одновременная замена для нескольких *различных* переменных. Оператор одновременного присваивания обозначается списком различных переменных), подлежащих замене (эти переменные разделяются запятыми), слева от знака присваивания и столь же протяженным списком выражений (также разделяемых запятыми) справа от знака присваивания. Итак, разрешается писать

$$x1, x2 := E1, E2 \qquad x1, x2, x3 := E1, E2, E3$$

Заметим, что i-я переменная из списка левой части должна быть заменена на i-е выражение из списка правой части, так что, например, при заданных $x1$, $x2$, $E1$ и $E2$ оператор

$$x1, x2 := E1, E2$$

семантически эквивалентен оператору

$$x2, x1 := E2, E1$$

Одновременное присваивание позволяет нам предписать, чтобы две переменные x и y обменялись своими значениями с помощью оператора

$$x, y := y, x$$

В иной записи эта операция выглядела бы более громоздкой. Легкость реализации одновременного присваивания и возможность с его помощью избегать некоторой избыточности записи являются причинами популярности таких операторов. Заметим, что если списки становятся слишком длинными, то получаемая программа становится весьма неудобочитаемой.

Истинный любитель НФБ расширит свой синтаксис, обеспечив две различные формы для оператора присваивания, следующим образом:

⟨оператор присваивания⟩ ::=⟨переменная⟩ := ⟨выражение⟩|
⟨переменная⟩, ⟨оператор присваивания⟩, ⟨выражение⟩

Это так называемое "рекурсивное определение", поскольку одна из альтернативных форм для синтаксической единицы, именуемой "оператор присваивания" (а именно вторая форма), содержит в качестве одного из своих компонентов снова эту же синтаксическую единицу, именуемую "оператор присваивания", т. е. ту самую синтаксическую единицу, которую мы определяем! На первый взгляд такое циклическое определение выглядит ужасающе, но после более внимательного изучения мы можем убедиться в том, что, по крайней мере с синтаксической точки зрения, в этом нет ничего неправильного. Например, поскольку, согласно первой альтернативе,

$$x2 := E1$$

является примером оператора присваивания, то формула

$$x1, x2 := E1, E2$$

допускает разбор вида

$$x1, ⟨\text{оператор присваивания}⟩, E2$$

а следовательно, согласно второй альтернативе, также является оператором присваивания. Однако с семантической точки зрения это отвратительно, потому что получается, что $E2$ ассоциируется с $x1$ вместо того, чтобы ассоциироваться с $x2$.

В сравнении с двухоператорным языком, содержащим только "пропустить" и "отказать", наш язык с оператором присваивания выглядит значительно более богатым: уже нет какой бы то ни было верхней границы для числа различных примеров синтаксической единицы "оператор присваивания". Но он все же явно недостаточен для наших целей; нам нужна возможность строить более

изощренные программы, более сложные конструкции. Для построения потенциально сложных конструкций мы пользуемся схемой, которую можно рекурсивно описать так:

⟨конструкция⟩ ::= ⟨простая конструкция⟩|

⟨правильная композиция записей вида: ⟨конструкция⟩ ⟩

Для того чтобы эта схема могла принести хоть какую-нибудь пользу, необходимо выполнение двух условий: в нашем распоряжении должны иметься "простые конструкции", чтобы было с чего начать, и, кроме того, мы должны знать, как строить "правильные композиции". Введенные раньше операторы можно взять в качестве простых конструкций; оставшаяся часть этой главы посвящена именно процессу правильной композиции некоей новой конструкции из заданных конструкций. Новая конструкция в свою очередь может выступать в роли части еще более сложного составного объекта.

После того, как объект был образован композицией частей, мы можем рассматривать его двумя способами. Во-первых, мы можем считать его "нераздельным целым", воспринимая его свойства в большей или меньшей степени как магические (или принимая их на веру или как постулаты). При таком подходе существенны только свойства конструкции; не имеет никакого значения, каким образом она образована из своих частей. С такой точки зрения любые две конструкции, обладающие одинаковыми свойствами, эквивалентны. Или же мы рассматриваем конструкцию как "составной объект", образованный так, что мы можем понять, почему она обладает объявленными свойствами. При этом мы воспринимаем части как "малые" нераздельные целые, для которых принимаются в расчет только их общие свойства. Второй подход вносит ясность в то, что мы понимаем под "композицией". Композиция должна определять, как свойства целого следуют из свойств частей.

После этих общих замечаний вернемся к нашим конкретным конструкциям, свойства которых, как мы считаем, выражаются их преобразователями предикатов. Точнее говоря, если заданы две конструкции $S1$ и $S2$, чьи преобразователи предикатов известны, можем ли мы представить себе правило вывода нового преобразователя предикатов из двух заданных? Если да, то мы можем считать результирующий преобразователь предикатов описанием свойств составного объекта, построенного специальным образом из частей $S1$ и $S2$.

Одним из простейших способов получения новой функции из двух заданных является так называемая "функциональная композиция", т. е. предоставление значения одной функции в качестве аргумента для другой. Составной объект, соответствующий такому преобразователю предикатов, принято обозначать через "S1; S2", и мы определяем

$$\mathrm{wp}(\text{"}S1; S2\text{"}, R) = \mathrm{wp}(S1, \mathrm{wp}(S2, R))$$

что при желании можно рассматривать как семантическое определение точки с запятой.

Замечание. Из того факта, что преобразователи предикатов для $S1$ и $S2$ обладают свойствами 1–4 из предыдущей главы, можно заключить, что и определенный выше преобразователь предикатов для "$S1; S2$" также обладает этими четырьмя свойствами. Например, поскольку для $S1$ и $S2$ справедлив закон исключенного чуда:

$$\mathrm{wp}(S1, F) = F \quad \text{и} \quad \mathrm{wp}(S2, F) = F$$

мы заключаем, подставляя F вместо R в верхнее определение что

$$\begin{aligned}\mathrm{wp}(\text{"}S1; S2\text{"}, F) &= \mathrm{wp}(S1, \mathrm{wp}(S2, F)) \\ &= \mathrm{wp}(S1, F) \\ &= F\end{aligned}$$

Читателю предоставляется в качестве упражнения проверить, что остальные три свойства тоже сохраняются. *(Конец замечания.)*

Перед тем как продолжить наши рассуждения, убедимся в том, что этим формальным описанием семантики точки с запятой охватывается наше интуитивное представление о ней (если оно у нас есть!), т. е. что составная конструкция "$S1; S2$" может быть реализована но правилу "сначала запустить конструкцию $S1$ и по окончании ее работы запустить $S2$". В самом деле, в нашем определении $\mathrm{wp}(\text{"}S1; S2\text{"}, R)$ мы представляем R-постусловие для составной конструкции — как постусловие к преобразователю предикатов для $S2$, и этим отражается то, что общая работа конструкции "$S1; S2$" может закончиться с окончанием работы $S2$. Соответствующее слабейшее предусловие для $S2$, т. е. $\mathrm{wp}(S2, R)$, представляется как постусловие к преобразователю предикатов для $S1$; тем самым мы

явно отождествляем начальное состояние для $S2$ с конечным состоянием для $S1$. Однако именно так и бывает, если работа $S1$ непосредственно предшествует во времени запуску $S2$.

Чтобы удостовериться в этом, рассмотрим пример. Пусть "$S1; S2$" представляет собой

$$"a := a + b; b := a * b"$$

и пусть нашим постусловием является некоторый предикат $R(a, b)$; в таком случае

$$\begin{aligned} \text{wp}(S2, R(a,b)) &= \text{wp}("b := a * b", R(a,b)) \\ &= R(a.c * b) \end{aligned}$$

и

$$\begin{aligned} \text{wp}("S1; S2", R(a,b)) &= \text{wp}(S1, \text{wp}(S2, R(a,b))) \\ &= \text{wp}(S1, R(a, a*b)) \\ &= \text{wp}("a := a + b", R(a, a*b)) \\ &= R(a+b, (a+b)*b) \end{aligned}$$

т.е мы можем гарантировать выполнение отношения R между конечными значениями a и b при условии, что первоначально то же отношение выполняется между $a + b$ и $(a + b) * b$ соответственно.

И наконец, поскольку функциональная композиция обладает свойством ассоциативности, то не имеет значения, будем ли мы разбирать "S1; S2; S3" как "[S1; S2]; S3" или же как "S1;[S2;S3]". Иначе говоря, мы имеем полное право трактовать точку с запятой как символ сочленения; поэтому не возникает никакой неопределенности, когда мы выписываем операторный список вида "$S_1; S_2; S_3; \ldots; S_n$", и мы будем без стеснения поступать так, когда для этого представятся подходящие случаи.

Упражнение. Проверьте, что конструкции

$$"x1 := E1; x2 := E2" \text{ и } "x2 := E2; x1 := E1"$$

семантически эквивалентны, если переменная $x1$ не встречается в выражении $E2$, а переменная $x2$ не встречается в выражении $E1$. На самом деле, в таком случае обе эти конструкции семантически эквивалентны одновременному присваиванию "$x1, x2 := E1, E2$". (Эта эквивалентность является одним из аргументов в пользу одновременного присваивания; ее применение позволяет нам избежать избыточности последовательной записи, более того, при одновременном присваивании становится очевидным, что два выражения $E1$ и $E2$ могут вычисляться одновременно, и это последнее обстоятельство представляет интерес при некоторых методиках реализации. Помимо того, быть может, более интересная возможность, что конструкция "$x1, x2 := E1, E2$" не окажется семантически эквивалентной ни конструкции "$x1 := E1; x2 := E2$", ни конструкции "$x2 := E2; x1 := E1$".) (Конец упражнения.)

До введения точки с запятой мы могли писать только однооператорные программы; с помощью точки с запятой мы обрели способность писать программы в виде сочленения из n, ($n > 0$) операторов: "$S_1; S_2; S_3; \ldots; S_n$". Если исключить промежуточную незавершенность, выполнение каждой программы всегда означает временную последовательность выполнений n операторов, сначала S_1, потом S_2 и так далее до S_n включительно. Однако из нашего примера игры на листе картона, реализующей алгоритм Евклида, мы знаем, что должны уметь описывать более широкий класс "правил игры": всякая игра будет состоять из последовательности перемещений, где каждое перемещение имеет вид либо "$x := x - y$", либо "$y := y - x$", но способ чередования этих перемещений во времени и даже их общее число будут изменяться от игры к игре; он зависит от начального положения фишки, он зависит от начального состояния системы. Если точка с запятой является единственным нашим средством для составления нового целого из заданных частей, то мы не в состоянии этого выразить, и поэтому нам необходимо искать нечто новое.

Пока точка с запятой остается единственным имеющимся в нашем распоряжении средством соединения, единственным обстоятельством, при котором происходит запуск одной из составляющих конструкций S_i ($i > 1$), являются правильное завершение работы (лексикографически) предшествующей конструкции. Чтобы добиться нужной нам гибкости, необходимо обеспечить возможность запуска той или иной (под) конструкции в зависимости от текущего состояния системы. Для этого мы введем — в два этапа — понятие "охраняемой команды", синтаксис которой задается следующим образом:

⟨охраняющий заголовок⟩ ::= ⟨логическое выражение⟩ →
⟨оператор⟩

⟨охраняемая команда⟩ ::= ⟨охраняющий заголовок⟩{;
⟨оператор⟩}

где фигурные скобки "{" и "}" следует читать так: "сопровождается любым числом (быть может, нулем) экземпляров содержимого скобок".

(Другой вариант синтаксиса охраняемой команды может выглядеть так:

⟨список операторов⟩ ::= ⟨оператор⟩{; ⟨оператор⟩}

⟨охраняемая команда⟩ ::= ⟨логическое выражение⟩ → ⟨список операторов⟩

Однако по причинам, которые не должны нас теперь интересовать, я предпочитаю синтаксис, в котором вводится понятие охраняющего заголовка).

В этом сочетании логическое выражение, предшествующее стрелке, называется "предохранителем". Идея состоит в том, что следующий за стрелкой список операторов будет выполняться лишь при том условии, что обеспечена начальная истинность значения соответствующего предохранителя. Предохранитель позволяет нам избежать выполнения списка операторов при тех первоначальных обстоятельствах, когда это полнение оказалось бы нежелательным или (если употребляются частично определенные операции) невозможным.

Истинность значения предохранителя является необходимым предварительным условием для выполнения охраняемой команды как целого; это условие, разумеется, не является достаточным, поскольку тем или иным способом — с двумя такими способами мы встретимся — до него еще должна дойти потенциальная "очередь". Именно поэтому охраняемая команда не рассматривается как оператор: оператор безоговорочно выполняется, когда наступает его очередь, а охраняемая команда может использоваться в качестве строительного блока для оператора. Если говорить точнее, мы предложим два различных способа составления оператора из набора охраняемых команд.

После некоторого осмысливания рассмотрение набора охраняемых команд представляется вполне естественным. Предположим, что нам требуется построить конструкцию, такую, чтобы при условии, что начальное состояние удовлетворяет Q, конечное состояние удовлетворяло R. Предположим далее, что нам не удается найти единый список операторов, который выполнял бы эту работу в любых случаях. (Если бы такой список операторов существовал, то мы именно его бы и использовали и не возникло бы потребности в охраняемых командах.) Однако нам может удасться найти несколько списков операторов, каждый из которых будет выполнять нужную работу для некоего подмножества возможных начальных состояний. К каждому из этих списков операторов мы можем присоединить в качестве предохранителя логическое выражение, характеризующее то подмножество, для которого этот список подходит, и если у нас имеется вполне достаточный набор предохранителей, такой, что из истинности Q всегда логически следует истинность значения хотя бы одного предохранителя, то для каждого начального состояния, удовлетворяющего Q, мы располагаем конструкцией, которая переведет систему в состояние, удовлетворяющее условию R, причем этой конструкцией служит одна из охраняемых команд, чей предохранитель имел начальное значение "истина". Чтобы выразить это, определим сначала

⟨набор охраняемых команд⟩ ::= ⟨охраняемая команда⟩{[]⟨охраняемая команда⟩}

где символ "[]" выступает в роли разделителя вариантов, в остальном не упорядоченных. Один из способов формирования оператора из набора охраняемых команд состоит в том, чтобы включить такой набор в пару скобок "**if** ... **fi**", при этом наш синтаксис для синтаксической категории, именуемой "оператор", пополняется следующей формой:

⟨оператор⟩ ::= **if**⟨набор охраняемых команд⟩**fi**

Таким образом, мы получаем возможность объединить несколько охраняемых команд в новую конструкцию. Мы можем представить себе работу, которая произойдет после запуска этой конструкции, следующим образом. Выбирается одна из охраняемых команд, чей предохранитель является истинным, и запускается ее список операторов.

Прежде чем мы перейдем к изложению формального описания семантики нашей новой конструкции, уместно сделать три замечания.

1. Предполагается, что все предохранители определены; если это не так, т. е. вычисление какого-то предохранителя может приисти к работе без правильного завершения, то допускается, что и вся конструкция не сможет правильно завершить свою работу.
2. Вообще говоря, наша конструкция приведет к недетерминированности при тех начальных состояниях, для которых истинны значения более чем одного предохранителя, поскольку остается

неопределенным, какой из соответствующих списков операторов будет тогда выбираться для запуска. Никакой недетерминированности не возникает, если все предохранители попарно исключают друг друга.

3. Если начальное состояние таково, что ни один из предохранителей не является истиной, то мы встречаемся с начальным состоянием, для которого не подходит ни один из вариантов, а следовательно, и вся конструкция в целом. Запуск при таком начальном состоянии приведет к отказу.

Замечание. Если мы допускаем также и пустой набор охраняемых команд, то оператор "**if-fi**" семантически эквивалентен нашему прежнему оператору "отказать". *(Конец замечания.)*

(В следующем формальном определении слабейшего предусловия для конструкции **if-fi** мы ограничимся случаем, когда все предохранители являются полностью определенными функциями. Если это не так, то нужно переписать выражение с использованием символа **cand** при дополнительном требовании, чтобы начальное состояние принадлежало области определения всех предохранителей.) Обозначим через "**IF**" оператор 1

$$\textbf{if} B_1 \to SL_1 \textbf{[]} B_2 \to SL_2 \textbf{[]} \ldots \textbf{[]} B_n \to SL_n \textbf{fi}$$

Тогда для любого постусловия R

$$\text{wp}(IF, R) = (\exists j : 1 \le j \le n : B_j) \textbf{ and } (\forall j : 1 \le j \le n : B_j \Rightarrow wp(SL_j, R))$$

Эту формулу следует читать так: wp(IF, R) является истиной для каждой такой точки в пространстве состояний, где хотя бы одно значение j из отрезка $1 \le j \le n$, такое, что B_j является истиной, и где, кроме того, для всех j из отрезка $1 \le j \le n$, таких, что B_j — истина, значение $wp(SL_j, R)$ также является истиной. Используя символ "...", как мы уже делали при описании самого оператора IF, мы можем предложить иную форму определения:

$$\begin{aligned} \text{wp}(IF, R) = &(B_1 \textbf{ or } B_2 \textbf{ or } \ldots \textbf{ or } B_n)\textbf{and} \\ &(B_1 \Rightarrow \text{wp}(SL_1, R))\textbf{and} \\ &(B_2 \Rightarrow \text{wp}(SL_2, R))\textbf{ and } \ldots \textbf{ and} \\ &(B_n \Rightarrow \text{wp}(SL_n, R)) \end{aligned}$$

Понимание этих формул не представляет особого труда. Требованием, чтобы значение хотя бы одного предохранителя являлось истиной, отражается факт отказа в случае, когда значения для всех предохранителей ложны. Кроме того, мы требуем для каждого начального состояния, удовлетворяющего wp(IF, R), чтобы выполнялось $B_j \Rightarrow wp(SL_j, R)$ для всех j. Для тех значений j, при которых B_j — ложь, это следование истинно независимо от значения $wp(SL_j, R)$, т. е. для таких значений j, разумеется, безразлично, что будет делать SL_j. При нашей реализации это учитывается в том, что для запуска не берется SL_j с ложным начальным значением предохранителя B_j. При тех значениях j, для которых B_j — истина, данное следование может быть истиной только в том случае, когда $wp(SL_j, R)$ также является истиной. Поскольку наше формальное определение требует истинности следования при всех значениях j, то реализация на самом деле имеет свободу выбора, когда более чем один предохранитель является истиной. Конструкция **if-fi** представляет собой только один из двух возможных способов построения оператора из набора охраняемых команд. В конструкции **if-fi** состояние, при котором все предохранители имеют ложные значения, ведет к отказу. В нашей второй форме мы разрешаем, чтобы состояние, при котором нет ни одного предохранителя с истинным значением, приводило к правильному завершению; поскольку в этой ситуации не запускается никакой список операторов, вполне естественно, что при этом возникает семантическая эквивалентность с пустым оператором. Однако это разрешение на правильное завершение, когда нет ни одного предохранителя с истинным значением, дополняется тем, что работе не разрешается завершаться, пока хотя бы один предохранитель является истиной. Итак, после запуска проверяются все предохранители. Если нет ни одного истинного значения предохранителя, то работа завершается; если же имеются предохранители с истинными значениями, то один из соответствующих списков операторов запускается, а после его завершения реализация снова начинает общую проверку всех предохранителей. Эта вторая конструкция обозначается путем заключения списка охраняемых команд в пару скобок "**do .. od**".

Формальное определение слабейшего предусловия для конструкции **do-od** является более сложным, чем для конструкции **if-fi**; дело в том, что первое выражается через второе. Мы сначала приведем это формальное определение, а затем объясним его. Обозначим через "DO" оператор

$$\textbf{do} B_1 \to SL_1 \textbf{[]} B_2 \to SL_2 \textbf{[]} \ldots \textbf{[]} B_n \to SL_n \textbf{od}$$

1 SL- аббревиатура для Statement List — список операторов. —*Прим. перев.*

и через "IF" оператор, получаемый путем заключения того же набора охраняемых команд в пару скобок "**if … fi**". Если условия $H_k(R)$ задаются в виде

$$H_0(R) = R \textbf{ and non}(\exists j : 1 \leq j \leq n : B_j)$$

и при $k > 0$

$$H_k(R) = \text{wp}(IF, H_{k-1}(R)) \textbf{ or } H_0(R)$$

то

$$\text{wp}(DO, R) = (\exists k : k \geq 0 : H_k(R))$$

Здесь интуитивно мы понимаем $H_k(R)$ следующим образом: это слабейшее предусловие, такое, что конструкция **do-od** завершит свою работу после не более чем k выборок охраняемых команд, причем система останется в конечном состоянии, удовлетворяющем постусловию R.

При $k = 0$ требуется, чтобы конструкция **do-od** завершила работу, не производя выборки какой-либо охраняемой команды, т. е. не допускается существования какого-либо предохранителя с истинным значением, что и выражено вторым логическим сомножителем. При этом начальная истинность R очевидным образом является необходимым условием для конечной истинности R, что и выражено первым логическим сомножителем.

При $k > 0$ нам нужно различать два случая: либо ни один из предохранителей не имеет истинного значения, но тогда должно выполняться условие R, и это приводит нас ко второму логическому слагаемому; либо хотя бы один предохранитель является истиной, и тогда события развиваются так, как при однократном запуске оператора "IF" (в начальном состоянии, которое не может привести к немедленному отказу вследствие отсутствия истинных значений предохранителей). Однако после такого выполнения, при котором производилась выборка одной охраняемой команды, нам необходима гарантия попадания в состояние, такое, чтобы потребовалось не более чем $(k-1)$ дальнейших выборок для достижения завершения работы в конечном состоянии, удовлетворяющем R. В соответствии с нашим определением, таким постусловием для этого оператора "IF" служит $H_{k-1}(R)$.

Согласно последней строке, определяющей $\text{wp}(DO, R)$, должно существовать такое значение k, чтобы потребовалось не более чем k выборок для обеспечения завершения работы в конечном состоянии, удовлетворяющем постусловию R.

Замечание. Если мы допускаем также и пустой набор охраняемых команд, то в силу сказанного оператор "**do od**" семантически эквивалентен нашему прежнему оператору "пропустить". *(Конец замечания.)*

5 ДВЕ ТЕОРЕМЫ

В этой главе мы выводим две теоремы об операторах, которые строятся из наборов охраняемых команд. Первая теорема касается конструкции выбора if-fi, а вторая — конструкции повторения do-od. В этом главе мы будем рассматривать конструкции, выводимые из набора охраняемых команд

$$B_1 \to SL_1 [] B_2 \to SL_2 [] \ldots [] B_n \to SL_n$$

Будем обозначать через "IF" и "DO" операторы, получаемые заключением этого набора охраняемых команд в пары скобок "if ... fi" и "do ... od" соответственно. Мы будем также использовать сокращение

$$BB = (\exists j : 1 \le j \le n : B_j)$$

Теорема. (Основная теорема для конструкции выбора)

Используя введенные только что обозначения, мы можем сформулировать основную теорему для конструкции выбора:

Пусть конструкция выбора IF и пара предикатов Q и R таковы, что

$$Q \Rightarrow BB \tag{1}$$

и

$$(\forall j : 1 \le j \le n : (Q \text{ and } B_j) \Rightarrow \text{wp}(SL_j, R)) \tag{2}$$

одновременно справедливы для всех состояний. Тогда

$$Q \Rightarrow \text{wp}(IF, R) \tag{3}$$

справедливо также для всех состояний. Поскольку, в силу определения,

$$\text{wp}(IF, R) = BB \text{ and } (\forall j : 1 \le j \le n : B_j \Rightarrow \text{wp}(SL_j, R))$$

и, согласно (1), из Q логически следует первый член правой части, то отношение (3) доказывается, если на основании (2) мы можем сделать вывод, что

$$Q \Rightarrow (\forall j : 1 \le j \le n : B_j \Rightarrow \text{wp}(SL_j, R)) \tag{4}$$

справедливо для всех состояний. Для любого состояния, при котором Q является ложью, отношение (4) истинно в силу определения логического следования. Для любого состояния, при котором Q является истиной, и для любого значения j мы можем различать два случая: либо B_j является ложью, но тогда $B_j \Rightarrow \text{wp}(SL_j, R)$ является истиной в силу определения следования, либо B_j является истиной, но тогда, согласно (2), $\text{wp}(SL_j, R)$ является истиной, а следовательно, $B_j \Rightarrow \text{wp}(SL_j, R)$ тоже является истиной. В результате мы доказали отношение (4), а следовательно, и (3).

Замечание. В частном случае бинарного выбора ($n = 2$) и при $B_2 = \text{non } B_1$ мы имеем $BB = T$, и слабейшее предусловие преобразуется так:

$$\begin{aligned} &(B_1 \Rightarrow \text{wp}(SL_1, R)) \text{ and } (\text{non } B_1 \Rightarrow \text{wp}(SL_2, R)) = \\ &(\text{non } B_1 \text{ or } \text{wp}(SL_1, R)) \text{ and } (B_1 \text{ or } \text{wp}(SL_2, R)) = \\ &(B_1 \text{ and } \text{wp}(SL_1, R)) \text{ or } (\text{non } B_1 \text{ and } \text{wp}(SL_2, R)) \end{aligned} \tag{5}$$

Последнее преобразование возможно потому, что из четырех перекрестных произведений член B_1 **and non** $B_1 = F$ может быть отброшен, а член $\text{wp}(SL_1, R)$ **and** $\text{wp}(SL_2, R)$ тоже может быть отброшен по следующей причине: в любом состоянии, где он истинен, обязательно является истиной какой-то один из двух членов формулы (5), и поэтому его самого можно исключить из дизъюнкции. Формула (5) имеет прямое отношение к предложенному Хоаром способу описания семантики конструкции **if-then-else** из языка АЛГОЛ 60. Поскольку здесь $BB = T$ логически следует из чего угодно, мы можем вывести (3) при более слабом предположении:

$$((Q \text{ and } B_1) \Rightarrow \text{wp}(SL_1, R)) \text{ and } ((Q \text{ and non } B_1) \Rightarrow \text{wp}(SL_2, R))$$

(Конец замечания.)

Теорема для конструкции выбора представляет особую важность в случае, когда пара предикатов Q и R может быть записана в виде

$$R = P$$
$$Q = P \text{ and } BB$$

В этом случае предпосылка (1) выполняется автоматически, а поскольку $(BB \text{ and } B_j) = B_j$, предпосылка (2) сводится к виду

$$(\forall j : 1 \leq j \leq n : (P \text{ and } B_j) \Rightarrow \text{wp}(SL_j, P)) \tag{6}$$

из чего мы можем вывести, согласно (3),

$$(P \text{ and } BB) \Rightarrow \text{wp}(IF, P) \qquad \text{для всех состояний} \tag{7}$$

Это отношение послужит предпосылкой для нашей следующей теоремы.

Теорема. (Основная теорема для конструкции повторениения.)

Пусть набор охраняемых команд с построенной для него конструкцией выбора IF и предикат P таковы, что

$$(P \text{ and } BB) \Rightarrow \text{wp}(IF, P) \tag{7}$$

справедливо для всех состояний. Тогда для соответствующей конструкции повторения DO можно вывести, что

$$(P \text{ and } \text{wp}(DO, T)) \Rightarrow \text{wp}(DO, P \text{ and non } BB) \tag{8}$$

для всех состояний.

Эту теорему, которая известна также под названием "основная теорема инвариантности для циклов", на интуитивном уровне не трудно понять. Предпосылка (7) говорит нам, что если предикат P первоначально истинен и одна из охраняемых команд выбирается для выполнения, то после ее выполнения P сохранит свою истинность. Иначе говоря, предохранители гарантируют, что выполнение соответствующих списков операторов не нарушит истинности P, если начальное значение P было истинным. Следовательно, вне зависимости от того, как часто производится выборка охраняемой команды из имеющегося набора, предикат P будет справедлив при любой новой проверке предохранителей. После завершения всей конструкции повторения, когда ни один из предохранителей не является истиной, мы тем самым закончим работу в конечном состоянии, удовлетворяющем P **and non** BB. Вопрос в том, завершится ли работа правильно. Да, если условие $\text{wp}(DO, T)$ справедливо и вначале; поскольку любое состояние удовлетворяет T, то $\text{wp}(DO, T)$ по определению является слабейшим предусловием для начального состояния, такого, что запуск оператора DO приведет к правильно завершаемой работе.

Формальное доказательство основной теоремы для конструкции повторения основывается на формальном описании семантики этой конструкции (см. предыдущую главу), из которого мы выводим

$$H_0(T) = \text{non } BB \tag{9}$$
$$\text{при } k > 0 : H_k(T) = \text{wp}(IF, H_{k-1}(T)) \text{ or non } BB \tag{10}$$
$$H_0(P \text{ and non } BB) = P \text{ and non } BB \tag{11}$$
$$\text{при } k > 0 : H_k(P \text{ and non } BB) = \text{wp}(IF, H_{k-1}(P \text{ and non } BB)) \text{ or } P \text{ and non } BB \tag{12}$$

Начнем с того, что докажем посредством математической индукции, что предпосылка (7) гарантирует справедливость

$$(P \text{ and } H_k(T)) \Rightarrow H_k(P \text{ and non } BB) \tag{13}$$

для всех состояний.

В силу отношений (9) и (11) отношение (13) справедливо при $k = 0$. Мы покажем, что отношение (13) может быть доказано при $k = K$ $(K > 0)$ на основании предположения, что (13) справедливо при $k = K - 1$.

$$\begin{aligned} P \text{ and } H_k(T) &= P \text{ and } \text{wp}(IF, H_{K-1}T)) \text{ or } P \text{ and non } BB \\ &= P \text{ and } BB \text{ and } \text{wp}(IF, H_{K-1}(T)) \text{ or } P \text{ and non } BB \\ &\Rightarrow \text{wp}(IF, P) and wp(IF, H_{K-1}(T)) \text{ or } P \text{ and non } BB \\ &= \text{wp}(IF, P) and H_{K-1}(T)) or P and non BB \\ &\Rightarrow wp(IF, H_{K-1}(P and non BB)) or P and non BB \\ &= H_K(P and non BB) \end{aligned}$$

Равенство в первой строке следует из (10), равенство во второй строке следует из того, что всегда wp(IF, R) $\Rightarrow$ BB, логическое следование в третьей строке вытекает из (7), равенство в четвертой строке основывается на свойстве 3 преобразователей предикатов, следование в пятой строке вытекает из свойства 2 преобразователей предикатов и из индуктивного предположения (13) для $k = K - 1$, и последняя строка следует из (12). Итак, мы доказали (13) для $k = K$, а следовательно, для всех значений $k \geq 0$.

Наконец, для любой точки в пространстве состояний мы имеем, в силу (13),

$$\begin{aligned} P \textbf{ and } \text{wp}(DO, T) &= (\exists k : k \geq 0 : PandH_k(T)) \\ &\Rightarrow (\exists k : k \geq 0 : H_k(P \textbf{ and non } BB)) \\ &= \text{wp}(DO, P \textbf{ and non } BB) \end{aligned}$$

и тем самым доказана основная теорема (8) для конструкции повторения. Ценность основной теоремы для конструкции повторения основывается на том, что ни в предпосылке, ни в ее следствии не упоминается фактическое число выборок охраняемой команды. Поэтому она применима даже в тех случаях, когда это число не определяется начальным состоянием.

6 О ПРОЕКТИРОВАНИИ ПРАВИЛЬНО ЗАВЕРШАЕМЫХ КОНСТРУКЦИЙ

Основная теорема для конструкции повторения применительно к условию P, сохраняемому инвариантно истинным, утверждает, что

$$(P \textbf{ and } \mathrm{wp}(DO, T)) \Rightarrow (DO, P \textbf{ and non } BB)$$

Здесь член $\mathrm{wp}(DO, T)$ представляет собой слабейшее предусловие, такое, что конструкция повторения заввершится. Если задана произвольная конструкция DO, то в общем случае очень трудно (а может быть, невозможно) определить $\mathrm{wp}(DO, T)$. Поэтому я предлагаю проектировать наши конструкции повторения, постоянно помня о требовании завершимости, т. е. предлагаю искать подходящее доказательство завершимости и строить программу таким способом, чтобы она удовлетворяла предположениям, на которых основывается это доказательство.

Предположим опять, что P — отношение, которое сохраняется инвариантно истинным, т.е.

$$(P \textbf{ and } BB) \Rightarrow \mathrm{wp}(IF, P) \qquad \text{для всех состояний}$$

Пусть t — конечная целочисленная функция от текущего состояния, такая, что

$$(P \textbf{ and } BB) \Rightarrow (t > 0) \qquad \text{для всех состояний}$$

и, кроме того, для любого значения t_0 и для всех

$$(P \textbf{ and } BB \textbf{ and } t \leq t_0 + 1) \Rightarrow \mathrm{wp}(IF, t \leq t_0) \tag{3}$$

Тогда мы докажем, что

$$P \Rightarrow \mathrm{wp}(DO, T) \qquad \text{для всех состояний} \tag{4}$$

Сопоставив этот факт с основной теоремой для повторения, мы можем заключить, что имеем для всех состояний

$$P \Rightarrow \mathrm{wp}(DO, P \textbf{ and non } BB) \tag{5}$$

Мы покажем это, доказав сначала методом математической индукции, что

$$(P \textbf{ and } t \leq k) \Rightarrow H_k(T) \qquad \text{для всех состояний} \tag{6}$$

справедливо при всех $k \geq 0$. Начнем с обоснования истинности (6) при $k = 0$. Поскольку $H_0(T) =$ **non** BB, нам требуется показать, что

$$(P \textbf{ and } t \leq 0) \Rightarrow \textbf{non } BB \qquad \text{для всех состояний} \tag{7}$$

Однако 7 — это просто другая форма записи выражения (2): оба они равны выражению

$$\textbf{non } P \textbf{ or non } BB \textbf{ or } (t > 0)$$

и поэтому (6) справедливо при $k = 0$.

Предположим теперь, что (6) справедливо при $k = K$; тогда

$$\begin{aligned}(P \textbf{ and } BB \textbf{ and } t \leq K + l) &\Rightarrow \mathrm{wp}(IF, P \textbf{ and } t \leq K) \\ &\Rightarrow \mathrm{wp}(IF, H_K(T));\end{aligned} \qquad (P \textbf{ and non } BB \textbf{ and } t \leq K + 1) \Rightarrow \textbf{non } BB = H_0(T)$$

И эти два логических следования можно объединить (из $A \Rightarrow B$ и $B \Rightarrow D$ мы можем заключить, что справедливо (A **or** $B \Rightarrow C$ **or** D)):

$$(P \textbf{ and } t \leq K + 1) \Rightarrow \mathrm{wp}(IF, H_K(T)) \textbf{ or } H_0(T) = H_{K+1}(T)$$

и тем самым истинность (6) доказана для всех $k \geq 0$. Поскольку t — ограниченная функция, мы имеем

$$(\exists k : k \geq 0 : t \leq k)$$

и

$$\begin{aligned} P &\Rightarrow (\exists k : k \geq 0 : P \textbf{ and } t \leq k) \\ &\Rightarrow (\exists k : k \geq 0 : H_k(T)) \\ &= \mathrm{wp}(DO, T) \end{aligned}$$

и тем самым доказано (4).

Интуитивно теорема совершенно ясна. С одной стороны, P останется истиной, а следовательно, $t \geq 0$ тоже останется истиной; с другой стороны, из отношения (3) следует, что каждая выборка охраняемой команды приведет к эффективному уменьшению t по крайней мере на 1. Неограниченное количество выборок охраняемых команд уменьшило бы значение t ниже любого предела, что привело бы к противоречию.

Применимость этой теоремы основывается на выполнении условий (2) и (3). Отношение (2) является достаточно простым, отношение (3) выглядит более запутанным. Наша основная теорема для конструкции повторения при

$$Q = (P \textbf{ and } BB \textbf{ and } t \leq t_0 + 1)$$
$$R = (t \leq t_0)$$

(присутствие свободной переменной t_0 в обоих предикатах является причиной того, что мы говорили о "паре предикатов") позволяет нам заключить, что условие (3) справедливо, если

$$(\forall j : 1 \leq j \leq n : (P \textbf{ and } B_j \textbf{ and } t \leq t_0 + 1) \Rightarrow \text{wp}(SL_j, t \leq t_0))$$

Иначе говоря, нам нужно доказать для всякой охраняемой команды, что выборка приведет к эффективному уменьшению t. Помня о том, что t является функцией от текущего состояния, мы можем рассмотреть

$$\text{wp}(SL_j, t \leq t_0) \tag{8}$$

Это предикат, включающий, помимо координатных переменных пространства состояний, также и свободную переменную t_0. До сих пор мы рассматривали такой предикат как предикат, характеризующий некое подмножество состояний. Однако для любого заданного состояния мы можем также рассматривать предикат как условие, налагаемое на t_0. Пусть $t_0 = t_{min}$ представляет собой минимальное решение уравнения (8) относительно t_0, тогда мы можем интерпретировать значение t_{min} как наименьшую верхнюю границу для конечного значения t. Если вспомнить, что, подобно функции t, t_{min} также является функцией от текущего состояния, то можно интерпретировать предикат

$$t_{min} \leq t - 1$$

как слабейшее предусловие, при котором гарантируется, что выполнение SL_j, Уменьшит значение t по крайней мере на 1.

Обозначим это предусловие, где — мы повторяем —аргумент является целочисленной функцией от текущего состояния, через

$$\text{wdec}(SL_j, t)$$

При этом инвариантность P и эффективное уменьшение t гарантируются, если мы имеем при всех значениях j

$$(P \textbf{ and } B_j) \Rightarrow (\text{wp}(SL_j, P) \textbf{ and } \text{wdec}(SL_j, t))$$

Обычно практический способ отыскания подходящего предохранителя B_j состоит в следующем. Уравнение (9) относится к типу

$$(P \textbf{ and } Q) \Rightarrow R$$

где (практически вычислимое!) значение Q нужно найти для заданных значений P и R. Мы замечаем, что

1. $Q = R$ является решением.
2. $Q = (Q1 \textbf{ and } Q2)$ является решением и $P \Rightarrow Q2$, то $Q1$ тоже является решением.
3. Если $Q = (Q1 \textbf{ or } Q2)$ является решением и $P \Rightarrow \textbf{non}\, Q2$, (или, что сводится к тому же самому, $(P \textbf{ and } Q2) = F$), то $Q1$ тоже является решением.
4. Если Q является решением и $Q1 \Rightarrow Q$, то $Q1$ тоже является решением.

Замечание 1. Если, действуя таким образом, мы приходим к кандидатуре Q для B_j, такой, что $P \Rightarrow \textbf{non}\, Q$, то эта кандидатура может быть далее упрощена (в соответствии с предыдущим наблюдением 3, поскольку при любом Q мы имеем $Q = (\textit{ложь} \textbf{or} Q)$) к виду $Q = \textit{ложь}$; это означает, что рассматриваемая охраняемая команда введена неудачно: ее можно исключить из набора, потому что она никогда не будет выбираться. *(Конец замечания 1.)*

Замечание 2. Часто на практике расщепляют уравнение (9) на два уравнения:

$$(P \textbf{ and } B_j) \Rightarrow \text{wp}(SL_j, P) \tag{9а}$$

$$(P \textbf{ and } B_j) \Rightarrow \text{wdec}(SL_j, t) \tag{9б}$$

и рассматривают их по отдельности. Тем самым разделяются две задачи: (9а) относится к тому, что остается инвариантным, тогда как (9б) относится к тому, что обеспечивает продвижение вперед. Если, имея дело с уравнением (9а), мы приходим к решению B_j, такому, что $P \Rightarrow B_j$, то тогда очевидно, что это условие не будет удовлетворять уравнению (9б), поскольку при таком B_j инвариантность P привела бы к недетерминированности *(Конец замечания 2.)*

Таким образом, мы можем построить конструкцию DO, такую, что

$$P \Rightarrow \text{wp}(DO, P \textbf{ and non } BB)$$

Наши условия B_j должны быть достаточно сильными, чтобы удовлетворялись следования (9); в результате этого новое гарантируемое постусловие P **and non** BB может оказаться слишком слабым и не обеспечить нам желаемого постусловия R. В таком случае мы все-таки не решили нашу проблему и нам следует рассмотреть другие возможности.

7 ПЕРЕСМОТРЕННЫЙ АЛГОРИТМ ЕВКЛИДА

Рискуя наскучить моим читателям, я посвящу еще одну главу алгоритму Евклида. Полагаю, что к этому времени некоторые из читателей уже закодируют его в виде

$x,\ y := X,\ Y$;
do $x \neq y \ \rightarrow$ **if** $x > y \ \rightarrow \ x := x - y$
[] $y > x \ \rightarrow \ y := y - x$
od;
печатать(x)

где предохранитель конструкции повторения гарантирует, что конструкция выбора не приведет к отказу. Другие читатели обнаружат, что этот алгоритм можно закодировать более просто следующим образом:

$x, y := X,\ Y$;
do $x > y \ \rightarrow \ x := x - y$
[] $y > x \ \rightarrow \ y := y - x$
od;
печатать(x)

Попробуем теперь забыть игру на листе картона и попытаемся изобрести заново алгоритм Евклида для отыскания наибольшего общего делителя двух положительных чисел X и Y. Когда мы сталкиваемся с такого рода проблемой, в принципе всегда возможны два подхода.

Первый состоит в том, чтобы пытаться следовать определению требуемого ответа настолько близко, насколько это возможно. По-видимому, мы могли бы сформировать таблицу делителей числа X; эта таблица содержала бы только конечное число элементов, среди которых имелись бы 1 в качестве наименьшего и X в качестве наибольшего элемента. (Если $X = 1$, то наименьший и наибольший элементы совпадут. Затем мы могли бы сформировать также аналогичную таблицу делителей Y. Из этих двух таблиц мы могли бы сформировать таблицу чисел, присутствующих в них обеих. Она представляет собой таблицу *общих* делителей чисел X и Y и обязательно является непустой, так как содержит элемент 1. Следовательно, из этой третьей таблицы мы можем выбрать (поскольку она тоже конечная!) максимальный элемент, и он был бы *наибольшим* общим делителем.

Иногда близкое следование определению, подобное обрисованному выше, является лучшим из того, что мы можем сделать. Существует, однако, и другой подход, который стоит испробовать, если мы знаем (или можем узнать) свойства функции, подлежащей вычислению. Может оказаться, что мы знаем так много свойств, что они в совокупности определяют эту функцию, тогда мы можем попытаться построить ответ, используя эти свойства.

В случае наибольшего общего делителя мы замечаем, например, что, поскольку делители числа $-x$ те же самые, что и для самого числа x, НОД(x, y) определен также для отрицательных аргументов и не меняется, если мы изменяем знак аргументов. Он определен и тогда, когда один из аргументов равен нулю; такой аргумент обладает бесконечной таблицей делителей (и поэтому нам не следует пытаться строить эту таблицу!), но поскольку второй аргумент ($\neq 0$) обладает конечной таблицей делителей, таблица общих делителей является все же непустой и конечной. Итак, мы приходим к заключению, что НОД(x, y) определен для всякой пары (x, y), такой, что $(x, y) \neq (0, 0)$. Далее, в силу симметрии понятия "общий" наибольший общий делитель является симметричной функцией своих двух аргументов. Еще одно небольшое умственное усилие позволит нам убедиться в том, что наибольший общий делитель двух аргументов не изменяется, если мы заменяем один из этих аргументов их суммой или разностью. Объединив все эти результаты, мы можем записать: для $(x, y) \neq (0, 0)$

(а) НОД(x, y) = НОД(y, x).

(б) НОД(x, y) = НОД$(-x, y)$.

(в) НОД(x, y) = НОД$(x + y, y)$ = НОД$(x - y, y)$ и т. д.

(г) НОД$(x, y) = abs(x)$, если $x = y$.

Предположим для простоты рассуждений, что этими четырьмя свойствами исчерпываются наши познания о функции НОД. Достаточно ли их? Вы видите, что первые три отношения выражают наибольший общий делитель чисел x и y через НОД для другой пары, а последнее свойство выражает его непосредственно через x. И в этом уже просматриваются контуры алгоритма, который для начала обеспечивает истинность

$$P = (\text{НОД}(X, Y) = \text{НОД}(x, y))$$

(это легко достигается путем присваивания "$x, y := X, Y$"), после чего мы "утрамбовываем" пару значений (x, y) таким образом, чтобы в соответствии с (а), (б) или (в) отношение P оставалось инвариантным. Если мы можем организовать этот процесс утрамбовки так, чтобы достигнуть состояния, удовлетворяющего $x = y$, то, согласно (г), мы находим искомый ответ, взяв абсолютное значение x.

Поскольку наша конечная цель состоит в том, чтобы обеспечить при инвариантности P истинность $x = y$, мы могли бы испытать в качестве монотонно убывающей функции функцию

$$t = \text{abs}(x - y).$$

Чтобы упростить наш анализ (всегда похвальная цель!), мы отмечаем, что если начальные значения x и y неотрицательные, то ничего нельзя выиграть введением отрицательного значения: если присваивание $x := E$ обеспечило бы $x < 0$, то присваивание $x := -E$ никогда не привело бы к получению большего конечного значения t (потому, что $y \geq 0$). Поэтому мы усиливаем наше отношение P, которое должно сохраняться инвариантным:

$$P = (P1 \textbf{ and } P2)$$

при

$$P1 = (\text{НОД}(X, Y) = \text{НОД}(x, y))$$

и

$$P2 = (x \geq 0 \textbf{ and } y \geq 0)$$

Это означает, что мы отказываемся от всякого использования операций $x := -x$ и $y := -y$, т.е. преобразований, допустимых по свойству (б). Нам остаются операции

$$\begin{aligned} &\text{из (а):}\ x, y := y, x \\ &\text{из (в):}\quad x := x + y\ y := y + x \\ &\qquad\quad x := x - y\ y := y - x \\ &\qquad\quad x := y - x\ y := x - y \end{aligned}$$

Будем заниматься ими по очереди и начнем с рассмотрения $x, y := y, x$:

$$\text{wp}(\text{"}x, y := y, x\text{"}, \text{abs}(x - y) \leq t_0) = (\text{abs}(y - x) \leq t_0)$$

поэтому

$$t_{min}(x, y) = \text{abs}(y - x)$$

следовательно

$$\text{wdec}(\text{"}x, y := y, x\text{"}, \text{abs}(x - y)) = (\text{abs}(y - x) \leq \text{abs}(x - y) - 1) = F$$

И здесь — для тех, кто не поверил бы без формального вывода,—мы доказали (или, если угодно, обнаружили) с помощью нашего исчисления, что преобразующая операция $x, y := y, x$ не представляет интереса, так как она не может привести к желаемому эффективному уменьшению выбранной нами функции t.

Следующей испытывается операция $x := x + y$, и мы находим, снова применяя исчисление из предыдущих глав:

$$\begin{aligned} \text{wp}(\text{"}x := x + y\text{"}, \text{abs}(x - y) \leq t_0) &= (\text{abs}(x) \leq t_0) \\ t_{min}(x, y) &= \text{abs}(x) = x \end{aligned}$$

(мы ограничиваемся состояниями, удовлетворяющими P)

$$\begin{aligned} \text{wdec}(\text{"}x := x + y\text{"}, \text{abs}(x - y)) &= (t_{min}(x, y) \leq t(x, y) - 1) \\ &= (x \leq \text{abs}(x - y) - 1) \\ &= (x + 1 \leq \text{abs}(x - y)) \\ &= (x + 1 \leq x - y \textbf{ or } x + 1 \leq y - x) \end{aligned}$$

Поскольку из P следует отрицание первого члена и к тому же $P \Rightarrow \text{wp}(\text{"}x := x + y\text{"}, P)$, то уравнение нашего предохранителя

$$(P \textbf{ and } B_j) \Rightarrow (\text{wp}(SL_j, P) \textbf{ and } \text{wdec}(SL_j, t))$$

удовлетворяется последним членом, и мы нашли нашу первую, а также (из соображений симметрии) нашу вторую охраняемую команду:

$$x + 1 \leq y - x \rightarrow x := x + y$$

и

$$y + 1 \leq x - y \rightarrow y := y + x$$

Аналогично мы находим (формальные манипуляции предоставляются в качестве упражнения прилежному читателю)

$$1 \leq y \textbf{ and } 3 * y \leq 2 * x - 1 \rightarrow x := x - y$$

и

$$1 \leq x \textbf{ and } 3 * x \leq 2 * y - 1 \rightarrow y := y - x$$

и

$$x + 1 \leq y - x \rightarrow x := y - x$$

и

$$y + 1 \leq x - y \rightarrow y := x - y$$

Разобравшись в том, чего мы достигли, мы вынуждены прийти к досадному заключению, что способом, намеченным в конце предыдущей главы, нам не удалось решить свою задачу: из P **and non** BB не следует, что $x = y$. (Например, при $(x, y) = (5, 7)$ значения всех предохранителей оказываются ложными.) Мораль сказанного, разумеется, в том, что наши шесть шагов не всегда обеспечивают такой путь из начального состояния в конечное состояние, при котором abs$(x - y)$ монотонно уменьшается. Поэтому нам нужно испытать другие возможности.

Для начала заметим, что не повредит несколько усилить условие $P2$:

$$P2 = (x > 0 \textbf{ and } y > 0)$$

так как начальные значения x и y удовлетворяют такому условию, и, кроме того, нет никакого смысла в генерации равного нулю значения, поскольку это значение может возникнуть только при вычитании в состоянии, где $x = y$, т.е. когда уже достигнуто конечное состояние. Но это только малая модификация; основная модификация должна быть связана с введением новой функции t, и я предлагаю взять такую функцию t, которая только ограничена снизу в силу инвариантности P. Очевидным примером является

$$t = x + y$$

Мы выясняем, что для одновременного присваиваивания

$$\text{wdec}(\text{"}x, y := y, x\text{"}, x + y) = F$$

и поэтому одновременное присваивание отвергается.

Мы находим для присваивания $x := x + y$

$$\text{wdec}(\text{"}x := x + y\text{"}, x + y) = (y < 0)$$

Истинность этого выражения исключается истинностью инвариантного отношения P, а следовательно, такое присваивание (наряду с присваиванием $y := y + x$) также отвергается.

Однако для следующего присваивания $x := x - y$ мы находим

$$\text{wdec}(\text{"}x := x - y\text{"}, x + y) = (y > 0)$$

т. е. условие, которое, логически следует из условия P (усиленного мною ради этого).

Окрыленные надеждой, мы изучаем

$$\text{wp}(\text{"}x := x - y\text{"}, P) = (\text{НОД}(X, Y) = \text{НОД}(x - y, y) \textbf{ and } x - y > 0 \textbf{ and } y > 0)$$

Крайние члены можно отбросить, потому что они следуют из P, и у нас остается средний член: итак, мы находим

$$x > y \rightarrow \quad x := x - y$$

и

$$y > x \rightarrow \quad y := y - x$$

И на этом мы могли бы прекратить исследование, так как, когда значения обоих предохранителей становятся ложными, выполняется желаемое нами отношение $x = y$. Если мы продолжим, то найдем третий и четвертый варианты:

$$x > y - x \quad \textbf{and} \quad y > x \rightarrow \quad x := y - x$$

и

$$y > x - y \quad \textbf{and} \quad x > y \rightarrow \quad y := \quad x - y$$

но не ясно, что можно выиграть их включением.

Упражнения.

1. Исследуйте при том же P выбор $t = \max(x, y)$.

2. Исследуйте при том же P выбор $t = x + 2 * y$.

3. Докажите, что при $X > 0$ и $Y > 0$ следующая программа, оперирующая над четырьмя переменными,

$$\begin{array}{l} x,\ y,\ u,\ v := X,\ Y,\ Y,\ X; \\ \textbf{do}\ \ x > y \rightarrow \quad x,\ v := x - y,\ v + u \\ \quad [\,]\ \ y > x \quad \rightarrow \quad y,\ u := \quad y - x,\ u + v \\ \textbf{od}; \\ \textit{печатать}((x + y)/2);\ \ \textit{печатать}((u + v)/2) \end{array}$$

печатает наибольший общий делитель чисел X и У, а следом за ним их наименьшее общее кратное. (Конец упражнений.)

Наконец, если наш маленький алгоритм запускается при паре (X, Y), которая не удовлетворяет предположению $X > 0$ **and** $Y > 0$, то произойдут неприятности: если $(X, Y) = (0, 0)$, то получится неправильный нулевой результат, а если один из аргументов отрицательный, то запуск приведет к бесконечной работе. Этого можно избежать, написав

$$\begin{array}{l} \textbf{if}\ \ X > 0 \ \ \textbf{and} \ \ Y > 0 \ \ \rightarrow \ \ x, y := X, Y; \\ \quad \textbf{do}\ \ x > y \rightarrow \ \ x := x - y [\,]\ \ y > x \rightarrow \ \ y := y - x \ \ \textbf{od};\ \ \textit{печатать}(x) \\ \textbf{fi} \end{array}$$

Включив только один вариант в конструкцию выбора, мы явно выразили условия, при которых ожидается работа этой маленькой программы. В таком виде это хорошо защищенный и довольно самостоятельный фрагмент, обладающий тем приятным свойством, что попытка запуска вне его области определения приведет к немедленному отказу.

8 ФОРМАЛЬНОЕ РАССМОТРЕНИЕ НЕСКОЛЬКИХ НЕБОЛЬШИХ ПРИМЕРОВ

В этой главе я проведу формальное построение нескольких небольших программ для решения простых задач. Не следует понимать эту главу как предложение строить программы только так и не иначе: такое предложение было бы довольно смехотворным. Я полагаю, что многим из моих читателей знакомо большинство примеров, а если нет, они, вероятно, не задумываясь смогут написать любую из этих программ.

Следовательно, построение программ проводится здесь совсем по другим причинам. Во-первых, это ближе познакомит нас с формализмом, развитым в предыдущих главах. Во-вторых, убедит нас в том, что по крайней мере в принципе, формализм способен сделать ясным и совершенно строгим то, что часто объясняется при помощи энергичной жестикуляции. В-третьих, многие из нас настолько хорошо знают эти программы, что уже забыли, каким образом давным-давно мы убедились в их правильности: в этом отношении настоящая глава напоминает начальные уроки планиметрии, которые по традиции посвящаются доказательству очевидного. В-четвертых, мы можем случайно обнаружить, к своему удивлению, что маленькая знакомая задачка не такая уже знакомая. Наконец, процесс построения программ может пролить свет на осуществимость, трудности и возможности автоматического составления программ или использования машины в процессе программирования. Это может оказаться важным, даже если автоматическое составление программ не вызывает у нас ни малейшего интереса, потому что может помочь нам лучше оценить ту роль, которую могут или должны играть наши изобретательские способности.

В моих примерах я буду формулировать требования задачи в форме "для фиксированных x, y, ...", что является сокращенной записью формы "для любых значений x_0, y_0, ... постусловие вида $x = x_0$ **and** $y = y_0$ **and** ... должно вызывать предусловие, из которого следует, что $x = x_0$ **and** $y = y_0$ **and** ...". Эта связь между постусловием и предусловием будет гарантирована тем, что мы будем относиться к таким величинам как к "временным константам"; они не будут встречаться в левых частях операторов присваивания.

Первый пример

Для фиксированных x и y обеспечить истинность отношения $R(m)$.

$$(m = x \textbf{ or } m = y) \textbf{ and } m \geq x \textbf{ and } m \geq y$$

Для любых значений x и y отношение $m = x$ может стать истинным только в результате присваивания $m := x$; следовательно, истинность $(m = x$ **or** $m = y)$ обеспечивается только выполнением либо $m := x$, либо $m := y$. В виде блок-схемы:

Picture 8.1

Все дело в том, что на входе нужно сделать правильный выбор, который обеспечит истинность $R(m)$ после завершения вычислений. Для этого мы "проталкиваем постусловие через альтернативы":

Picture 8.2

и мы получили предохранители! Так как

$$R(x) = ((x = x \textbf{ or } x = y) \textbf{ and } x \geq x \textbf{ and } x \geq y) = (x \geq y)$$

и

$$R(y) = ((y = x \textbf{ or } y = y) \textbf{ and } y \geq x \textbf{ and } y \geq y) = (y \geq x)$$

мы приходим к нашему решению:

$$\textbf{if } x \geq y \rightarrow m := x \ [] \ y \geq x \rightarrow m := y \textbf{ fi}$$

Поскольку $(x \geq y$ **or** $y \geq x) = T$, отказа никогда не произойдет (попутно мы получили доказательство существования: для любых значений x и y существует m, удовлетворяющее $R(m)$). Поскольку $(x \geq y$ **and** $y \geq x) \neq F$, наша программа не обязательно детерминирована. Если первоначально $x = y$, то оказывается неопределенным, какое из двух присваиваний будет выбрано для исполнения; такая недетерминированность совершенно корректна, так как мы показали, что выбор не имеет значения.

Замечание. Если бы среди доступных примитивов имелась функция "max", мы могли бы написать программу $m := \max(x, y)$, поскольку $R(\max(x, y)) = T$. *(Конец замечания.)*

Полученная программа не производит очень сильного впечатления; с другой стороны, мы видим, что в процессе вывода программы из постусловия на долю нашей изобретательности почти ничего не осталось.

Второй пример

Для фиксированного значения n ($n > 0$) задана функция $f(i)$ для $0 \leq i < n$. Обеспечить истинность R:

$$0 \leq k < n \textbf{ and } (\forall i : 0 \leq i < n : f(k) \geq f(i))$$

Поскольку наша программа должна работать при любом положительном значении n, трудно себе представить, как можно обеспечить истинность R, не прибегая к помощи цикла; поэтому мы ищем отношение P, которое легко можно сделать истинным в начале программы и такое, что в конце $(P \textbf{ and non } BB) \Rightarrow R$. Следовательно, отыскивая P, мы ищем отношение, более слабое, чем R; иначе говоря, мы хотим получить обобщение нашего конечного состояния. Стандартный способ обобщения отношения — это замена константы переменной (возможно, в ограниченном интервале), и здесь мой опыт подсказывает мне, что нужно заменить константу n новой переменной, например j, и взять в качестве P

$$0 \leq k < j \leq n \textbf{ and } (\forall i : 0 \leq i < j : f(k) \geq f(i))$$

где условие $j \leq n$ добавлено для того, чтобы область определения функции f была конечной. Теперь, имея такое обобщение, мы легко получаем

$$(P \textbf{ and } j = n) \Rightarrow R$$

Чтобы подтвердить возможность использования таким образом выбранного P, мы должны располагать легким средством для обеспечения истинности P в начале программы. Поскольку

$$(k = 0 \textbf{ and } j = 1) \Rightarrow P$$

мы отваживаемся предложить следующую структуру для нашей программы (комментарии даются в скобках):

$k, j := 0, \ 1$ $\{P$ стало истинным$\}$;
do $j \neq n \ \rightarrow$ некий шаг к $j = n$ при инвариантности P **od**
$\{R$ стало истинным$\}$

Снова мой опыт подсказывает мне выбрать в качестве монотонно убывающей функции t текущего состояния функцию $t = (n - j)$, которая и в самом деле такова, что $P \Leftarrow (t \geq 0)$. Чтобы гарантировать монотонное убывание t, я предлагаю увеличивать j на 1, после чего мы можем написать

$$\begin{gathered}\text{wp}(\text{"}j := j + 1\text{"}, P) = \\ 0 \leq k < j + 1 \leq n \textbf{ and } (\forall i : 0 \leq i < j + 1 : f(k) \geq f(i)) = \\ 0 \leq k < j + 1 \leq n \textbf{ and } (\forall i : 0 \leq i < j : f(k) \geq f(i)) \textbf{ and } f(k) \geq f(j)\end{gathered}$$

Первые два члена следуют из P **and** $j \neq n$ (потому что $(j \leq n \textbf{ and } j \neq n) \Rightarrow (j + 1 \leq n)$, и именно поэтому мы решили увеличить j только на 1). Следовательно,

$$(P \textbf{ and } j \neq n \textbf{ and } f(k) \geq f(j)) \Rightarrow \text{wp}(\text{"}j := j + 1\text{"}, P)$$

и мы можем использовать последнее условие в качестве предохранителя. Программа

$k, \ j := 0, 1$;
do $j \ \neq \ n \rightarrow$ **if** $f(k) \geq \ f(j) \rightarrow \ j := j + 1$ **fi od**

действительно даст правильный ответ, если она завершится нормально. Однако нормальное завершение не гарантировано, поскольку конструкция выбора может привести к отказу — и это действительно так и будет, если $k = 0$ не удовлетворяет R. Если неверно, что $f(k) \geq f(i)$, мы можем сделать это отношение верным при помощи присваивания $k := j$ и продолжить наши исследования:

$$\begin{gathered}\text{wp}(\text{"}k, j := j, j + 1\text{"}, P) = \\ 0 \leq j < j + 1 \leq n \textbf{ and } (\forall i : 0 \leq i < j + 1 : f(j) \geq f(i)) = \\ 0 \leq j < j + 1 \leq n \textbf{ and } (\forall i : 0 \leq i < j : f(j) \geq f(i))\end{gathered}$$

С огромным облегчением мы замечаем, что

$$(P \textbf{ and } j \neq n \textbf{ and } f(k) \leq f(j)) \Rightarrow \text{wp}(\text{"}k, j := j, j + 1\text{"}, P)$$

и следующая программа будет работать, не подвергаясь опасности отказа.

$$k,\ j = 0, 1;$$
$$\textbf{do}\ j \neq n \rightarrow\ \textbf{if}\ f(k) \geq f(j) \rightarrow\ j := j + 1$$
$$[]\ f(k) \leq f(j) \rightarrow\ k, j := j, j + 1\ \textbf{fi}\ \textbf{od}$$

Следует сделать несколько примечаний. Первое: поскольку предохранители конструкции выбора не обязательно исключают друг друга, программа таит в себе внутреннюю недетерминированность того же типа, что и в первом примере. Эта недетермированность может проявиться и внешне. Функция f могла оказаться такой, что конечное значение k неоднозначно; в этом случае наша программа может выработать любое допустимое значение!

Второе примечание: то, что мы построили правильную программу, не означает, что мы справились с задачей. Программирование в равной степени является как математической, так и инженерной дисциплиной; мы должны заботиться о правильности в той же мере, как скажем, и об эффективности. Исходя из предположения, что на вычисление значения функции f для данного аргумента затрачивается относительно много времени, хороший инженер должен обратить внимание на то, что в программе, по всей вероятности, будет многократно вычисляться $f(k)$ для одного и того же значения k. В этом случае рекомендуется пойти на сделку и пожертвовать некоторым объемом памяти, чтобы сэкономить часть времени, затрачиваемого на вычисление. Но при этом наши усилия, направленные на повышение эффективности программы и уменьшение затрат времени, ни в коей мере не должны служить извинением за внесение беспорядка в программу. (Это очевидная вещь, но я хочу подчеркнуть эту мысль, потому что очень часто запутанность программы оправдывается ссылкой на эффективность. Однако при ближайшем рассмотрении такое оправдание всегда оказывается необоснованным, должно быть, потому, что запутанность программы всегда неоправданна.) Чтобы аккуратно реализовать обмен времени вычислений на память, можно воспользоваться методом введения одной или нескольких дополнительных переменных, чье значение можно использовать, так как сохраняется инвариантным некоторое отношение. В данном примере было замечено, что возможны частые повторные вычисления f(k) для одного и того же k, а это наводит на мысль о введении дополнительной переменной, скажем max, и расширении инвариантного отношения за счет дополнительного члена

$$max = f(k)$$

Это отношение должно стать истинным, когда k получит начальное значение, и должно сохраняться инвариантным (при помощи явного присваивания значения переменной max) после изменения k. Мы приходим к следующей программе:

$$k, j,\ max := 0, 1, f(0);$$
$$\textbf{do}\ j \neq n \rightarrow\ \textbf{if}\ max \geq f(j) \rightarrow\ j := j + 1$$
$$[]\ max \leq f(j) \rightarrow\ k,\ j,\ max := j,\ j + 1,\ f(j)\ \textbf{fi}\ \textbf{od}$$

Вероятно, эта программа гораздо более эффективна, чем наша предыдушая версия, В таком случае хороший инженер на этом не остановится, потому что теперь он обнаружит, что мог бы распорядиться повторными вычислениями $f(j)$ для одного и того же j. Для этого можно ввести дополнительную переменную, скажем h, и обеспечить инвариантность

$$h = f(j)$$

Однако мы не можем сделать это на том же глобальном уровне, как для нашего предыдущего члена: значение $j = n$ не исключено, а при этом значение $f(j)$ не обязательно определено. Следовательно, истинность отношения $h = f(j)$ должна заново обеспечиваться после каждой проверки, показывающей, что $j \neq n$, по завершении внешней охраняемой команды (так сказать, "как раз перед od") мы имеем $h = f(j - 1)$, но нас это не должно беспокоить, и пусть все остается так, как есть.

$$k,\ j,\ max := 0,\ 1,\ f(0);$$
$$\textbf{do}\ j \neq n \rightarrow\ h := f(j);$$
$$\textbf{if}\ max \geq h \rightarrow\ j := j + 1$$
$$[]\ max \leq h \rightarrow\ k,\ j,\ max := j,\ j + 1,\ h\ \textbf{fi}\ \textbf{od}$$

Последнее примечание касается не столько нашего решения задачи, сколько наших соображений о подходе к решению. Мы говорили о математическом подходе, мы говорили об инженерном подходе

и провели разделение между этими подходами, сосредоточивая внимание то на одном, то на другом аспекте. Хотя разделение подходов абсолютно необходимо, когда речь идет о более сложных задачах, я должен подчеркнуть, что, сосредоточивая внимание на одном из аспектов, не следует полностью игнорировать другие аспекты. Разрабатывая математическую часть проекта, мы должны помнить, что даже математически правильная программа может погибнуть, если она плоха с инженерной точки зрения. Аналогично, осуществляя "сделку" между памятью и временем, мы должны делать это аккуратно и систематически, чтобы по неряшливости не внести ошибок в программу; кроме того, хотя предварительно был проведен математический анализ, мы должны к тому же достаточно хорошо разбираться в задаче, чтобы судить о том, приведут ли предполагаемые изменения к значительному улучшению программы.

Замечание. До того как я начал пользоваться этими формальными построениями, я всегда использовал "$j < n$" в качестве предохранителя в этой конструкции повторения, теперь я должен отучитъся от этой привычки, так как в случае, подобном данному, конечно, предпочтительнее предохранитель "$j \neq n$". Для этого имеются две причины. Предохранитель "$j \neq n$" позволяет нам сделать вывод, что по завершении программы $j = n$, не ссылаясь при этом на инвариантное отношение P; тем самым упрощается дискуссия о том, что дает нам вся конструкция в целом по сравнению с предохранителем "$j < n$". Гораздо важнее, однако, что предохранитель "$j \neq n$" ставит завершение программы в зависимость от (части) инвариантного отношения, а именно $j \leq n$, и поэтому ему следует отдать предпочтение, так как его использование приведет к усилению конструкции. Если в результате выполнения $j := j + 1$ значение j так возрастет, что станет справедливым отношение $j > n$, предохранитель "$j < n$" не поднимет тревогу, тогда как предохранитель "$j \neq n$" по крайней мере не допустит нормального завершения. Такой аргумент представляется вполне весомым, даже если не принимать во внимание сбой машины. Пусть в последовательности $x_0, x_1, x_2, \ldots$ значение x_0 задается, а значение x_i для $i > 0$ определяется как $x_i = f(x_{i-1})$, где f — некоторая вычислимая функция. Будем внимательно и точно следить за тем. чтобы сохранялось инвариантным отношение $X = x_i$. Предположим, что в программе имеется монотонно возрастающая переменная n, такая, что для некоторых значений n нас интересуют x_n. При условии что $n \geq i$, мы всегда можем сделать истинным отношение $X = x_n$ при помощи

$$\textbf{do } i \neq n \rightarrow i,\ X := i + 1,\ f(X) \textbf{ od}$$

Если же отношение $n \geq i$ не обязательно имеет место (может быть, последующие изменения в программе привели к тому, что в процессе вычислений n будет не только возрастать), приведенная выше конструкция (к счастью!) не придет к завершению, в то время как конструкция

$$\textbf{do } i < n \rightarrow i,\ X := i + 1,\ f(X) \textbf{ od}$$

не обеспечит истинности отношения $X = x_n$. Мораль этой истории такова, что при прочих равных условиях мы должны выбирать наши предохранители как можно более слабыми. *(Конец замечания.)*

Третий пример

Для фиксированных $a(a \geq 0)$ и $d(d > 0)$ требуется обеспечить истинность R:

$$0 \leq r < d \textbf{ and } d|(a - r)$$

(Здесь вертикальная черта "|" заменяет собой слова "является делителем".) Иначе говоря, нам требуется вычислить наименьший неотрицательный остаток r, полученный в результате деления a на d. Чтобы эта задача действительно была задачей, мы должны ограничить использование арифметических операций только операциями сложения и вычитания. Поскольку условие $d|(a - r)$ выполняется при $r = a$ и при этом, так как $a \geq 0$, верно, что $0 \leq r$, предлагается выбрать в качестве инвариантного отношения P:

$$0 \leq r \textbf{ and } d|(a - r)$$

В качестве функции t, убывание которой должно обеспечить завершение работы программы, мы выбираем само r. Поскольку изменение r должно быть таким, чтобы неизменно выполнялось условие $d|(a - r)$, r можно изменять только на число, кратное d, например на само d. Таким образом, нам, как оказалось, предлагается вычислить

$$\text{wp}("r := r - d", P) \textbf{ and } \text{wdec}("r := r - d", r) = 0 \leq r - d \textbf{ and } d|(a - r + d) \textbf{ and } d > 0$$

Поскольку член $d > 0$ можно было бы добавить к инвариантному отношению P, обоснования требует только первый член; мы находим соответствующий предохранитель "$r \geq d$" и пробуем составить

такую программу:

if $a \geq 0$ **and** $d > 0 \rightarrow$
  $r := a$;
  do $r \geq d \rightarrow r := r - d$ **od**
fi

По завершении программы стало истинным отношение P **and non** $r \geq d$, из чего следует истинность R, и, таким образом, задача решена.

Предположим теперь, что нам, кроме того, потребовалось бы присвоить q такое значение, чтобы по окончании программы было бы также верно, что

$$a = d * q + r$$

иначе говоря, требуется найти не только остаток, но и частное. Попробуем добавить этот член к нашему инвариантному отношению. Поскольку

$$(a = d * q + r) \Rightarrow (a = d * (q + 1) + (r - d))$$

мы приходим к программе

if $a \geq 0$ **and** $d > 0 \rightarrow$
  $q,\ r := 0,\ a$;
  do $r \geq d \rightarrow q,\quad r := q + 1,\ r - d$ **od**
fi

На выполнение приведенных выше программ будет, конечно, затрачиваться очень много времени, если частное велико. Можем ли мы сократить это время? Очевидно, этого можно добиться, если уменьшать r на величину, кратную и большую d. Вводя для этой цели новую переменную, скажем dd, мы получаем условие, которое должно неизменно выполняться на протяжении всей работы программы:

$$d|dd \textbf{ and } dd \geq d$$

Мы можем ускорить нашу первую программу, заменив "$r := r - d$" уменьшением, возможно повторным, r на dd, предоставляя в то же время возможность dd, первоначально равному d, довольно быстро возрастать, например каждый раз удваивая dd. Тогда мы приходим к следующей программе:

if $a \geq 0$ **and** $d > 0 \rightarrow$
  $r := a$;
  do $r \geq d \rightarrow$
    $dd := d$;
    do $r \geq dd \rightarrow r := r - dd;\ dd := dd + dd$ **od**
  od
fi

Ясно, что отношение $0 \leq r$ **and** $d|(a - r)$ сохраняется инвариантным, и поэтому программа обеспечит истинность R, если она завершится нормально, но так ли это? Конечно, так, поскольку внутренний цикл, который завершается при $dd > 0$, возбуждается только при начальных состояниях, удовлетворяющих условию $r \geq dd$, и поэтому уменьшение $r := r - dd$ выполняется по крайней мере один раз для каждого повторения внешнего цикла.

Но такое рассуждение (хотя и достаточно убедительное!) является весьма неформальным, а поскольку в этой главе мы говорим о формальном построении программ, мы можем попробовать сформулировать и доказать теорему, которой интуитивно воспользовались.

Пусть IF, DO и BB понимаются в обычном смысле, и P — это отношение, сохраняющееся инвариантным, т.е.

$$(P \textbf{ and } BB) \Rightarrow \mathrm{wp}(IF, P) \qquad \text{для всех состояний} \tag{1}$$

и пусть t — целочисленная функция, такая, что для любого значения t_0 и для всех состояний

$$(P \textbf{ and } BB \textbf{ and } t \leq t_0 + 1) \rightarrow \mathrm{wp}(IF, t \leq t_0) \tag{2}$$

или, что то же,

$$(P \textbf{ and } BB) \Rightarrow \mathrm{wdec}(IF, t) \qquad \text{для всех состояний} \tag{3}$$

Тогда для любого значения t_0 и для всех состояний

$$(P \text{ and } BB \text{ and } \mathrm{wp}(DO, T) \text{ and } t \leq t_0 + 1) \Rightarrow \mathrm{wp}(DO, t \leq t_0) \tag{4}$$

или, что то же,

$$(P \text{ and } BB \text{ and } \mathrm{wp}(DO, T)) \Rightarrow \mathrm{wdec}(DO, t) \tag{5}$$

В словесной формулировке: если сохраняемое инвариантным отношение P гарантирует, что каждая выбранная для исполнения охраняемая команда вызывает действительное уменьшение t, то конструкция повторения вызовет действительное уменьшение t, если она нормально завершится после по крайней мере одного выполнения какой-либо охраняемой команды. Теорема настолько очевидна, что было бы досадно, если бы ее доказательство оказалось трудным, но, к счастью, это не так. Мы покажем, что из (1) и (2) следует, что для любого значения t_0 и для всех состояний

$$(P \text{ and } BBandH_k(T)) \text{ and } t \leq t_0 + 1) \Rightarrow H_k(t \leq t_0) \tag{6}$$

для всех $k \geq 0$. Это справедливо для $k = 0$, поскольку $(BB \text{and} H_0(T)) = F$, и, исходя из предположения, что (6) справедливо для $k = K$, мы должны показать, что (6) справедливо и для $k = K + 1$.

$$\begin{aligned}
&(P \text{ and } BB \text{ and } H_{K+1}(T) \text{ and } t \leq t_0 + 1) \\
&\Rightarrow \mathrm{wp}(IF, P) \text{ and } \mathrm{wp}(IF, H_K(T)) \text{ and } \mathrm{wp}(lF, t \leq t_0) \\
&= \mathrm{wp}(IF, P \text{ and } H_K(T) \text{ and } t \leq t_0) \\
&\Rightarrow \mathrm{wp}((IF, (P \text{ and } BB \text{ and } H_K(T) \text{ and } t \leq t0 + 1) \text{ or } (t \leq t_0 \text{ and non } BB)) \\
&\Rightarrow \mathrm{wp}(IF, H_K(t \leq t_0) \text{ or } H_0(t \leq t_0)) \\
&= \mathrm{wp}(IF, H_K(t \leq t_0)) \\
&\Rightarrow \mathrm{wp}(IF, H_K(t \leq t_0)) \text{ or } H_0(t \leq t_0) \\
&= H_{K+1}(t \leq t_0)
\end{aligned}$$

Первая импликация следует из (1), определения $H_{K+1}(T)$ и (2); равенство в третьей строке очевидно; импликация в четвертой строке выводится при помощи присоединения (BB **or non** BB) и последующего ослабления обоих членов; импликация в пятой строке следует из (6) при $k = K$ и определения $H_0(t \leq t_0)$; остальное понятно. Таким образом, мы доказали справедливость (6) для всех $k > 0$, а отсюда немедленно вытекают (4) и (5).

Упражнение

Измените нашу вторую программу таким образом, чтобы она вычисляла также и частное, и дайте формальное доказательство правильности вашей программы. *(Конец упражнения.)*

Предположим теперь, что имеется маленькое число, скажем 3, на которое нам позволено умножать и делить, и что эти операции выполняются достаточно быстро, так что имеет смысл воспользоваться ими. Будем обозначать произведение "$m * 3$" (или "$3 * m$") и частное "$m/3$"; последнее выражение будет использоваться только при условии, что первоначально справедливо $3|m$. (Мы ведь работаем с целыми числами, не так ли?)

И опять мы пытаемся обеспечить истинность желаемого отношения R при помощи конструкции повторения, для которой инвариантное отношение P выводится путем замены константы переменной. Заменяя константу d переменной dd, чьи значения будут представляться только в виде $d *$ (степень 3), мы приходим к инвариантному отношению P:

$$0 \leq r < dd \text{ and } dd \,(\mathrm{a} - r) \text{ and } (\exists i : i \geq 0 : dd = d * 3^i)$$

Мы обеспечим истинность этого отношения и затем, заменяя его инвариантным, постараемся достичь состояния, при котором $d = dd$.

Чтобы обеспечить истинность P, нам понадобится еще одна конструкция повторения: сначала мы обеспечим истинность отношения

$$0 \leq r \text{ and } dd \,(a - r) \text{ and } (\exists i : i \geq 0 : dd = d * 3^i)$$

а затем будем увеличивать dd, пока его значение не станет достаточно большим и таким, что $r < dd$. Получим следующую программу:

if $a \geq 0$ **and** $d > 0 \rightarrow$
$r, dd := a, d;$
do $r \geq dd \rightarrow dd := dd * 3$ **od;**
do $dd \neq d \rightarrow dd := dd/3;$
do $r \geq dd \rightarrow r := r - dd$ **od**
od
fi

Упражнение

Измените приведенную выше программу таким образом, чтобы она вычисляла также и частное, и дайте формальное доказательство правильности вашей программы, Это доказательство должно наглядно показывать, что всякий раз, когда вычисляется $dd/3$, имеет место $3|dd$. *(Конец упражнения.)*

Для приведенной выше программы характерна довольно распространенная особенность. На внешнем уровне две конструкции повторения расположень подряд; когда две или больше конструкций повторения на одном и том же уровне следуют друг за другом, охраняемые команды более поздних конструкций оказываются, как правило, более сложными, чем команды предыдущих конструкций. (Это явление известно как "закон Дейкстры", который, однако, не всегда выполняется.) Причина такой тенденции ясна: каждая конструкция повторения добавляет свое "**and non** BB" к отношению, которое она сохраняет инвариантным, и это дополнительное отношение следующая конструкция также должна сохранять инвариантным. Если бы не внутренний цикл, второй цикл был бы в точности противоположен первому; и функция дополнительного оператора

do $r \geq dd \rightarrow r := r - dd$ **od**

именно в том и состоит, чтобы восстанавливать возможно нарушенное отношение $r < dd$, достигаемое при выполнении первого цикла.

Четвертый пример

Для фиксированных $Q1$, $Q2$, $Q3$ и $Q4$ требуется обеспечить истинность R, причем R задается как $R1$ **and** $R2$, где

R1: последовательность значений (q1, q2, q3, q4) есть некоторая перестановка значений (Ql, Q2, Q3, Q4)

R2: q1≤q2 ≤q3 ≤q4

Если мы возьмем $R1$ в качестве сохраняемого инвариантным отношения P, получим такое возможное решение:

$q1, q2, q3, q4 := Q1, Q2, Q3, Q4;$
do $q1 > q2 \rightarrow q1, q2 := q2, q1$
[] $q2 > q3 \rightarrow q2, q3 := q3, q2$
[] $q3 > q4 \rightarrow q3, q4 := q4, q3$
od

Очевидно, что после первого присваивания P становится истинным и ни одна из охраняемых команд не нарушает его истинности. По завершении программы мы имеем **non** BB, а это есть отношение $R2$. В том, что программа действительно придет к завершению, каждый из нас убеждается по-разному в зависимости от своей профессии: математик мог бы заметить, что число перестановок убывает, исследователь операций будет интерпретировать это как максимизацию $q1 + 2 * q2 + 3 * q3 + 4 * q4$, а я — как физик — сразу "вижу", что центр тяжести смещается в одном направлении (вправо, если быть точным). Программа примечательна в том смысле, что, какие бы предохранители мы ни выбрали, никогда не возникнет опасности нарушения истинности P: предохранители, используемые в нашем примере, являются чистым следствием необходимости завершения программы.

Замечание. Заметьте, что мы могли бы добавить также и другие варианты, такие, как

$q1 > q3 \rightarrow q1, q3 := q3, q1$

причем их нельзя использовать для замены одного из трех, перечисленных в программе. *(Конец замечания.)*

Это хороший пример того, какого рода ясности можно достичь при нашей недетерминированности; излишне говорить однако, что я не рекомендую сортировать большое количество значений аналогичным способом.

Пятый пример

Нам требуется составить программу аппроксимации квадратного корня; более точно: для фиксированного n ($n > 0$) программа должна обеспечить истинность

$$R : a^2 \leq n \textbf{ and } (a+1)^2 > n$$

Чтобы ослабить это отношение, можно, например, отбросить один из логических сомножителей, скажем последний, и сосредоточиться на

$$P : a^2 \leq n$$

Очевидно, что это отношение верно при $a = 0$, поэтому выбор начального значения не должен нас беспокоить. Мы видим, что если второй член не является истинным, то это вызывается слишком маленьким значением a, поэтому мы могли бы обратиться к оператору "$a := a + 1$". Формально мы получаем

$$\text{wp}(\text{"}a := a+1\text{"}, P) = ((a+1)^2 \leq n)$$

Используя это условие в качестве (единственного!) предохранителя, мы получаем (P **and non** BB) = R и приходим к следующей программе:

if $n \geq 0 \rightarrow$
$a := 0$ $\{P$ стало истинным$\}$;
do $(a+1)^2 \leq n \rightarrow$ $a := a+1$ $\{P$ осталось истинным$\}$ **od**
$\{R$ стало истинным $\}$
fi $\{R$ стало истинным $\}$

При составлении программы мы исходили из предположения, что она завершится, и это действительно так, поскольку корень из неотрицательного числа есть монотонно возрастающая функция: в качестве t мы можем взять функцию $n - a^2$.

Эта программа довольно очевидна, к тому же она и не очень эффективна: при больших значениях n она будет работать довольно долго. Другой способ обобщения R — это введение другой переменной (скажем, b — и снова в ограниченном интервале изменения), которая заменит часть R, например,

$$P : \quad a^2 \leq n \textbf{ and } b^2 > n \textbf{ and } 0 \leq a < b$$

Благодаря такому выбору P обладает тем приятным свойством, что

$$(P \textbf{ and } (a+1 = b)) \Rightarrow R$$

Таким образом, мы приходим к программе, представленной в такой форме (здесь и далее конструкция **if** n≥0 →... **fi** опускается):

$a, b := 0, \ n+1$ $\{P$ стало истинным$\}$;
do $a+1 \neq b \rightarrow$ уменьшить $b-a$ при инвариантности P **od**
$\{R$ стало истинным$\}$

Пусть d будет величиной, на которую уменьшается разность $b - a$ при каждом выполнении охраняемой команды. Разность может уменьшаться либо за счет уменьшения b, либо за счет увеличения a, либо за счет и того и другого. Не теряя общности, мы можем ограничиться рассмотрением таких шагов, когда изменяется либо a, либо b, но не a и b одновременно: если a слишком мало и b слишком велико и на одном шаге только уменьшается b, тогда на следующем шаге можно увеличить a. Эти соображения приводят нас к программе в следующей форме:

$a, \ b := 0, \ n+1$ $\{P$ стало истинным$\}$;
do $a+1 \neq b \rightarrow$
$d := \ldots$ $\{d$ имеет соответствующее значение и P по − прежнему выполняется$\}$;
if $\ldots \rightarrow$ $a := a+d$ $\{P$ осталось истинным$\}$
[] $\ldots \rightarrow$ $b := b-d$ $\{P$ осталось истинным$\}$
fi $\{P$ стало истинным$\}$
od $\{R$ стало истинным$\}$

Теперь

$$\text{wp}(\text{"}a := a+d\text{"}, P) = ((a+d)^2 \leq n \textbf{ and } b^2 > n)$$

Поскольку истинность второго члена следует из истинности P, мы можем использовать первый член в качестве нашего первого предохранителя; аналогично выводится второй предохранитель, и мы получаем очередную форму нашей программы:

$$
\begin{array}{l}
a,\ b := 0,\ n+1; \\
\textbf{do}\ a+1 \neq b \rightarrow d := \ldots; \\
\qquad \textbf{if}(a+d)^2 \leq n \rightarrow a := a+d \\
\qquad [\!]\ (b-d)^2 > n \rightarrow b := b-d \\
\qquad \textbf{fi}\ \{P \text{ осталось истинным}\} \\
\textbf{od}\ \{R \text{ стало истинным}\}
\end{array}
$$

Нам остается еще соответствующим образом выбрать d. Поскольку мы взяли $b - a$ (на самом деле, $b - a - 1$) в качестве нашей функции t, эффективное убывание должно быть таким, чтобы d удовлетворяло условию $d > 0$. Кроме того, последующая конструкция выбора не должна приводить к отказу, т. е. по крайней мере один из предохранителей должен быть истинным. Это значит, что из отрицания одного, $(a + d)^2 > n$, должен следовать другой, $(b - d)^2 > n$; это гарантируется, если

$$a + d \leq b - d$$

или

$$2 * d \leq b - a$$

Наряду с нижней границей мы установили также и верхнюю границу для d. Мы могли бы взять $d = 1$, но чем больше d, тем быстрее работает программа, поэтому мы предлагаем

$$
\begin{array}{l}
a, b := 0, n+1; \\
\textbf{do}\ a+1 \neq b \rightarrow d := (b-a) \div 2; \\
\qquad \textbf{if}\ (a+d)^2 \leq n \rightarrow a := a+d \\
\qquad\quad [\!]\ (b-d)^2 > n \rightarrow b := b-d \\
\qquad \textbf{fi} \\
\textbf{od}
\end{array}
$$

где $n \div 2$ есть $n/2$, если $2|n$ и $(n - 1)/2$, если $2|(n - 1)$.

Использование действия $\div$ побуждает нас разобрать в том, что произойдет, если мы навяжем себе ограничение на $b - a$, полагая, что $b - a$ четно при каждом вычислении d. Вводя $c = (b - a)$ и исключая b, мы получаем инвариантное отношение

$$P: \qquad a^2 \leq n \textbf{ and } (a+c)^2 > n \textbf{ and } (\exists i : i \geq 0 : c = 2^i)$$

и программу (в которой c играет роль d)

$$
\begin{array}{l}
a, c := 0,\ 1;\ \textbf{do}\ c^2 \leq n \rightarrow c := 2 * c\ \textbf{od}; \\
\textbf{do}\ c \neq 1 \rightarrow c := c/2; \\
\qquad \textbf{if}\ (a+c)^2 \leq n \rightarrow a := a+c \\
\qquad\quad [\!]\ (a-c)^2 > n \rightarrow \textit{пропустить} \\
\qquad \textbf{fi} \\
\textbf{od}
\end{array}
$$

Замечание. Эта программа очень похожа на последнюю программу для третьего примера, где вычислялся остаток в предположении, что мы имеем право умножать и делить на 3. В приведенной выше программе можно было бы заменить конструкцию выбора на

$$\textbf{do}\ (a+c)^2 \leq n \rightarrow a := a+c\ \textbf{od}$$

Если условие, которому должен удовлетворять остаток, $0 \leq r < d$, записать как $r < d$ **and** $(r + d) \geq d$, сходство станет еще более отчетливым. *(Конец замечания.)*

Понимая, что так можно окончательно "замучить" этот маленький приимер, я бы хотел тем не менее предложить последний вариант преобразования программы. Мы написали программу, исходя из предположения, что операция возведения числа в квадрат входит в состав имеющихся операций; но предположим, что это не так и что единственные операции типа умножения, имеющиеся в нашем

распоряжении,— это умножение и деление на (малые) степени 2. Тогда наша последняя программа оказывается не такой уж хорошей, т. е. она нехороша, если мы предполагаем, что значения переменных, с которыми непосредственно оперирует машина, отождествлены со значениями переменных a и c, как это было бы при "абстрактных" вычислениях. Другими словами, мы можем рассматривать a и c как абстрактные переменные, чьи значения представляются (в соответствии с соглашением, предусматривающим более сложные условия, чем просто тождественность) значениями других переменных, с которыми в действительностм оперирует машина при выполнении программы. Мы можем позволить машине работать не непосредственно с a и c, а с p, q, r, такими, что

$$p = a * c$$
$$q = c^2$$
$$r = n - a^2$$

Мы видим, что это — преобразование координат, и каждой траектории в нашем (a, c)-пространстве соответствует траектория в нашем (p, q, r)-пространстве. Обратное не всегда верно, так как значения p, q и r не независимы: в терминах p, q и r мы получаем избыточность и, следовательно, потенциальную возможность, пожертвовав некоторым объемом памяти, не только выиграть время вычислений, но даже избавиться от необходимости возводить в квадрат! (Совершенно ясно, что преследовалась именно эта цель, когда строилось преобразование, переводящее точку в (a, c)-пространстве в точку в (p, q, r)-пространстве.) Теперь мы можем попытаться перевести все булевы выражения и перемещения в (a, c)-пространстве в соответствующие булевы выражения и перемещения в (p, q, r) -пространстве. Если бы нам удалось сделать это в терминах допустимых операций, наша задача была бы успешно решена. Предлагаемое преобразование и в самом деле отвечает нашим требованиям, и результатом его является следующая программа (переменная h была введена для очень локальной оптимизации):

$p, q, r := 0, 1, n$; **do** $q \leq n \rightarrow q := q * 4$ **od**;
do $q \neq 1 \rightarrow q := q/4$; $h := p + q$; $p := p/2$ $\{h = 2 * p + q\}$
  if $r \geq h \rightarrow p, r := p + q, r - h$
    [] $r < h \rightarrow$ *пропустить*
  fi
od { p имеет значение, требуемое для a }

Этот пятый пример был включен благодаря истории его создания (правда, несколько приукрашенной). Когда младшей из двух наших собак было всего несколько месяцев, я однажды вечером пошел погулять с ними обеими. Назавтра мне предстояло читать лекции студентам, которые всего лишь несколько недель назад начали знакомиться с программированием, и мне нужна была простая задача, чтобы на ее примере я мог "массировать" решения. В течение часовой прогулки я продумал первую, третью и четвертую программы, правда, правильно ввести h в последней программе я смог только при помощи карандаша и бумаги, когда вернулся домой. Ко второй программе, той, которая оперирует с a и b и которая была здесь представлена как переходная ступень к нашему третьему решению, я пришел лишь несколько недель спустя — и она была тогда в гораздо менее элегантной форме, чем представлена здесь. Вторая причина для ее включения в число представленных программ — это соотношение между третьей программами: по отношению к четвертой программе третья представляет собой наш первый пример так называемой "абстракции представления".

Шестой пример

Для фиксированных X ($X > 1$) и Y ($Y \geq 0$) программа должна обеспечить истинность отношения

$$R: \quad z = X^Y$$

при очевидном предположении, что операция возведения в степень не входит в набор доступных операций. Эта задача может быть решена при помощи "абстрактной переменной", скажем h; при решении задачи мы будем пользоваться циклом, для которого инвариантным является отношение

$$P: \qquad h * z = X^Y$$

и наша (в равной степени "абстрактная") программа могла бы выглядеть так:

$h, z := X^Y, 1$ {P стало истинным};
do $h \neq 1 \rightarrow$ сжимать h при инвариантности P **od**
{ R стало истинным }

Последнее заключение справедливо, поскольку (P **and** $h = 1$) $\Rightarrow R$. Приведенная выше программа придет к завершению, если после конечного числа применений операции "сжимания" h станет равным 1. Проблема, конечно, в том, что мы не можем представить значение h значением конкретной переменной, с которым непосредственно оперирует машина; если бы мы могли так сделать, мы могли бы сразу присвоить z значение X^Y, не затрудняя себя введением h. Фокус в том, что для представления текущего значения h мы можем ввести две — на этом уровне конкретные — переменные, скажем x и y, и наше первое присваивание предлагает в качестве соглашения об этом представлении

$$h = x^y$$

Тогда условие "$h \neq 1$" переходит в условие "$y \neq 0$", и наша следующая задача состоит в том, чтобы подыскать выполнимую операцию "сжимания". Поскольку при этой операции произведение $h * z$ должно оставаться инвариантным, мы должны делить h на ту же величину, на которую умножается z. Исходя из представления h, наиболее естественным кандидатом на эту величину можно считать текущее значение x. Без дальнейших затруднений мы приходим к такому виду нашей абстрактной программы:

$x, y, z := X, Y, 1$ $\{P$ стало истинным$\}$;
do $y \neq 0 \rightarrow y, z := y - 1,\ z * x$ $\{P$ осталось истинным$\}$ **od**
$\{R$ стало истинным$\}$

Глядя на эту программу, мы понимаем, что число выполнений цикла равно первоначальному значению Y, и можем задать себе вопрос, нельзя ли ускорить программу. Ясно, что задачей охраняемой команды является сведение y к нулю; не изменяя *значения* h, мы можем проверить, нельзя ли изменить *представление* этого значения в надежде уменьшить значение y. Попытаемся воспользоваться тем фактом, что конкретное представление значения h, заданное как x^y, вовсе не является однозначным. Если y четно, мы можем разделить y на 2 и возвести x в квадрат, при этом значение h совсем не изменится. Непосредственно перед операцией сжимания мы вставляем преобразование, приводящее к наиболее привлекательному преобразованию h и получаем следующую программу:

$x, y, z := X, Y, 1$;
do $y \neq 0 \rightarrow$ **do** $2 \mid y \rightarrow x,$ y := $x * x,\ y/2$ **od**;
$y, z := y - 1, z * x$
od $\{R$ стало истинным $\}$

Существует только одна величина, которую можно бесконечно делить пополам и она не станет нечетной, эта величина — нуль; иначе говоря, внешний предохранитель гарантирует нам, что внутренний цикл придет к завершению.

Я включил этот пример по разным причинам. Меня поразило открытие, что если просто вставить в программу что-то, что на абстрактном уровне действует как пустой оператор, можно так изменить алгоритм, что число операций, которое раньше было пропорционально Y, станет пропорционально только $\log(Y)$. Это открытие было прямым следствием того, что я заставил себя думать в терминах отдельной абстрактной переменной. Программа возведения в степень, которую я знал, была следующей:

$x, y, z := X, Y, 1$;
do $y <> 0 \rightarrow$ **if non** $2 \mid y \rightarrow y, z := y - 1, z * x$
[] $2 \mid y \rightarrow$ *пропустить* **fi**;
$x, y := x * x, y/2$
od

Эта последняя программа очень хорошо известна, многие из нас пришли к ней независимо друг от друга. Поскольку последнее возведение x в квадрат, когда y стал равным нулю, уже излишне, на эту программу часто ссылались как на пример, подтверждающий необходимость иметь то, что мы назвали бы "промежуточнными выходами". Принимая во внимание нашу программу, я прихожу к заключению, что этот довод слаб.

Седьмой пример

Для фиксированного значения n ($n \geq 0$) задана функция $f(i)$ для $0 \leq i < n$. Присвоить булевой переменной "*всешесть*" такое значение, чтобы в конечном результате стало справедливым

$$R: \qquad \textit{всешесть} = (\forall i : 0 \leq i < n : f(i) = 6)$$

(Этот пример обнаруживает некоторое сходство со вторым примером данной главы. Заметим, однако, что в этом примере допускается равенство n нулю. В таком случае интервал возможных значений для квантора "$\forall$" пуст и должно быть верным, что *всешесть* = *истина*.) По аналогии со вторым примером введем инвариантное отношение

$$P: \qquad \textit{всешесть} = (\forall i : 0 \leq i < j : f(i) = 6) \textbf{ and } 0 \leq j \leq n$$

поскольку это отношение легко сделать истинным при $j = 0$ и к тому же $(P \textbf{ and } j = n) \Rightarrow R$. Единственное, что мы должны сделать, это понять, как увеличивать j при инвариантности P. Поэтому берем

$$\text{wp}("j := j + 1", P) = (\textit{всешесть} = (\forall i : 0 \leq i < j + 1 : f(i) = 6)) \textbf{ and } 0 \leq j + 1 \leq n$$

Справедливость последнего члена следует из справедливости P **and** $j \neq n$; и в этом нет никакой проблемы, так как мы уже решили, что условие $j \neq n$, взятое в качестве предохранителя, является остаточно слабым, чтобы делать выводы о значении