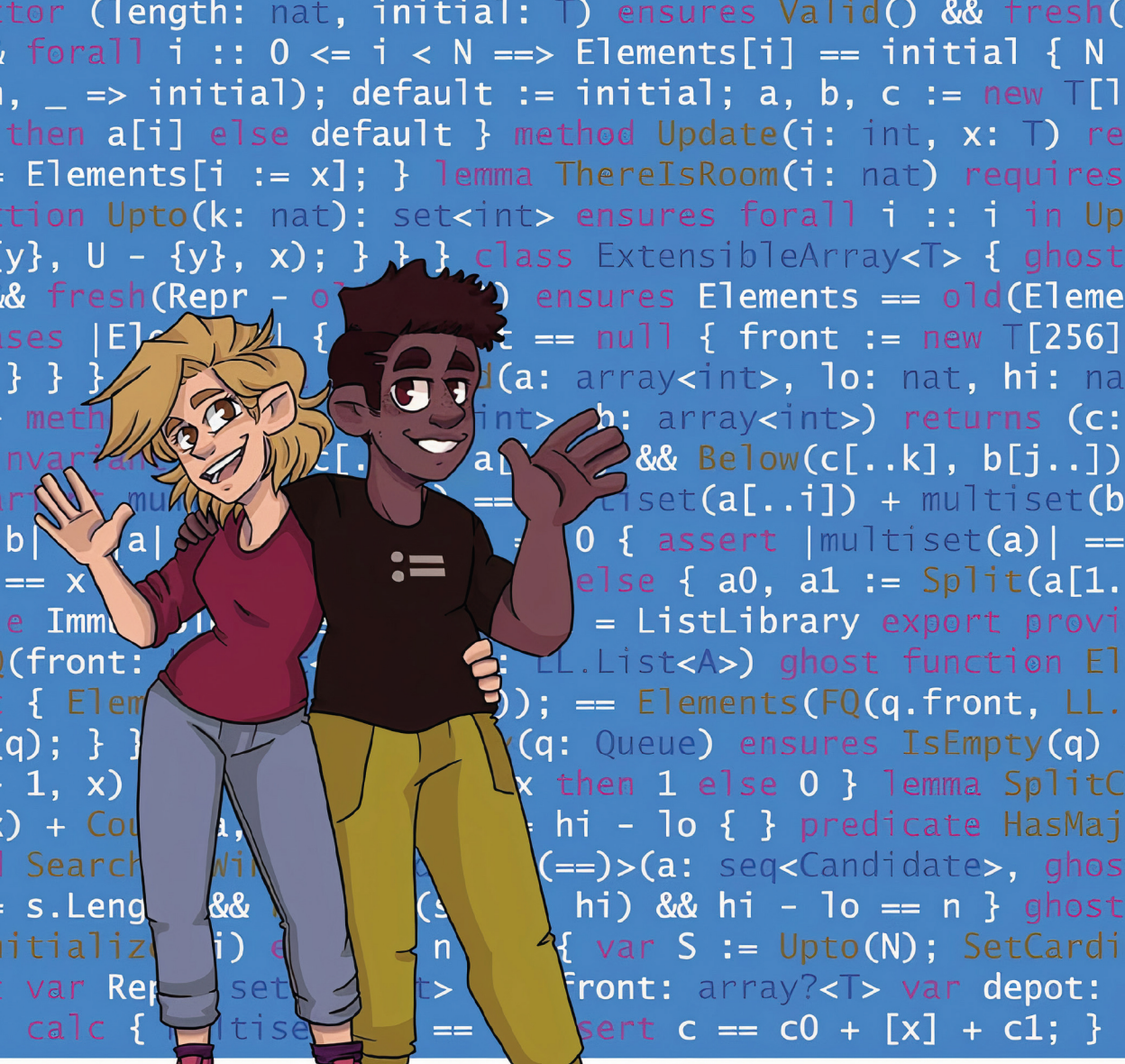


К. Рустан М. Лейно

Доказательство корректности программ

ДМК Пресс
издательство

К. Рустан М. Лейно

Доказательство корректности программ

Program Proofs

K. Rustan M. Leino

Illustrated by Kaleb Leino

The MIT Press
Cambridge, Massachusetts
London, England

Доказательство корректности программ

К. Рустан М. Лейно

Иллюстрации: Калеб Лейно

ДМК Пресс
издательство
Москва, 2024

УДК 004.052.42
ББК 32.973.2
Л42

Л42 К. Рустан М. Лейно

Доказательство корректности программ / Пер. с англ. А. Н. Киселева – М.: ДМК Пресс, 2024. – 530 с.: ил.

ISBN 978-5-93700-199-3

Данная книга учит формально рассуждать о компьютерных программах, используя последовательный подход и язык программирования Dafny, поддерживающий верификацию. Показано, как писать спецификации для программ, как удовлетворить требования этих спецификаций и как писать доказательства корректности программ относительно спецификаций. Автор сначала представляет теоретические предпосылки, лежащие в основе рассуждений о программном коде, а затем постепенно переходит к реальным примерам, использующих объекты, структуры данных и нетривиальную рекурсию. Книга написана простым и понятным языком, содержит множество забавных иллюстраций и практических упражнений.

Издание будет полезно студентам вузов, преподавателям, исследователям в области формальной верификации, а также сотрудникам компаний, применяющих дедуктивную верификацию на практике.

ISBN 978-0-262-54623-2 (англ.)
ISBN 978-5-93700-199-3 (рус.)

Оглавление

Предисловие редактора

Перед читателями книги открывается обширное введение в мир дедуктивной верификации программного обеспечения. Книга предназначена для студентов высших учебных заведений, изучающих курсы по формальным методам в программировании, преподавателей таких курсов, исследователей в области формальной верификации и сотрудников компаний, применяющих дедуктивную верификацию на практике.

Актуальность изучения дедуктивной верификации программ повышается с каждым годом, так как только формальная верификация может гарантировать корректность программного обеспечения. Необходимо отметить, что классическим методом проверки корректности программ является тестирование, но тестирование не может гарантировать корректность. Применение дедуктивной верификации позволяет гарантированно сделать программное обеспечение доверенным. В первую очередь дедуктивную верификацию важно применять для проверки корректности программного обеспечения таких систем, к корректности которых предъявляются повышенные требования, в частности встроенных систем, микроконтроллеров, интернета вещей, беспилотных авиационных систем, транспортных систем, роботизированных систем, космических аппаратов. Поэтому книга адресована в том числе и разработчикам программного обеспечения таких систем.

Автором книги является Лейно, известный специалист в области дедуктивной верификации программ. Лейно смог описать область дедуктивной верификации программ простым и понятным языком, последовательно увеличивая сложность и глубину изложения материала. Особенно стоит отметить, что стиль изложения книги позволяет программистам освоить дедуктивную верификацию без предварительных знаний в данной области.

В настоящее время основные исследования по дедуктивной верификации в России сконцентрированы в научных институтах, университетах и компаниях из Москвы, Санкт-Петербурга, Новосибирска, Барнаула, Ярославля и Иннополиса. Там же в основном осуществляется и применение дедуктивной верификации. Можно надеяться, что издание данной книги расширит географию применения дедуктивной верификации программ в России. Важно отметить две разработанные в России системы для дедуктивной верификации программ на языке C: систему AstraVer, разработанную в Институте системного программирования им. В. П. Иванникова РАН, и систему C-lightVer, разработанную в Институте систем информатики им. А. П. Ершова СО РАН. С 2023 г. стартовала серия всероссийских соревнований по формальной верификации программ VeHa, где один из треков

соревнования посвящен дедуктивной верификации. Можно полагать, что издание данной книги послужит дальнейшей популяризации и еще большему развитию дедуктивной верификации программ в России.

к. ф.-м. н. Кондратьев Дмитрий Александрович,
научный сотрудник Института систем информатики им. А. П. Ершова СО РАН,
ассистент Новосибирского государственного университета

Вступление

Добро пожаловать в «Доказательство корректности программ»!

Цель этой книги – на практике показать, что подразумевается под спецификациями кода и что подразумевается под проверкой соответствия кода этим спецификациям. В этом вступлении я хочу рассказать вам о самой книге и о том, как ею пользоваться.

Программы и доказательство их корректности

Когда я впервые познакомился с понятием верификации программ, разработка и доказательство корректности программ выполнялись вручную. Мне понравилось заниматься этим. Но думаю, что я был единственным в группе, кто тратил время на доказательство. Но, даже если вам нравится заниматься этим, не всегда понятно, как связать теорию, почерпнутую из учебников, с практикой, когда вы сидите перед компьютером и пытаетесь заставить программу работать. А если вы с самого начала не занимались доказательствами и не получили достаточно практики, то вам может быть неясно, есть ли вообще хоть какая-то связь между теорией и практикой.

Чтобы сблизить их, нужно получить практический опыт доказательства на языке программирования, распознаваемом компьютером. А написание спецификаций и доказательств соответствия этим спецификациям вместе с самим программным кодом дает дополнительное преимущество автоматической проверки доказательств компьютером. В результате вы получаете мгновенную обратную связь, которая поможет вам ощутить достоинства верификации. Вместо сдачи рукописного домашнего задания ассистенту и получения результата от него через неделю (когда вы уже позабыли суть упражнения, а комментарии ассистента кажутся менее важными, чем задание, которое нужно сдать на следующей неделе) вы можете *использовать* автоматический верификатор десятки раз за короткое время и наблюдать результаты его работы в контексте разрабатываемой программы!

Попытка преподавания концепций доказательства корректности программ с привязкой к реальному языку программирования может показаться безумием. Большинство языков не предназначено для верификации, и попытки прикрутить к такому языку возможности создания спецификаций и проверки соответствия им в лучшем случае будут выглядеть неуклюже. Более того, если придется изучать отдельный синтаксис описания доказательств или взаимодействий с автоматическим верификатором, то нагруз-

ка на учащегося вырастет еще больше. Чтобы по-настоящему связать программирование и верификацию, последнюю нужно изучать с точки зрения концепций программной инженерии (таких как предусловия, инварианты и утверждения), а не с точки зрения индуктивных схем, преобразований семантических отображений и директив доказательств.

К счастью, существует несколько языков программирования, поддерживающих создание спецификаций и доказательств (так называемые *языки с поддержкой верификации*), а также интегрированных сред разработки (IDE), способных запускать автоматические верификаторы (иногда их называют *автоактивными верификаторами*: автоматизированные инструменты, поддерживающие взаимодействие через текст программы [82]). К их числу относятся функциональные языки WhyML [20] и F* [53], язык SPARK, основанный на Ada [43, 117], объектно ориентированный язык Eiffel [89, 44, 121], императивные языки GRASShopper [126] и Whiley [109] и, наконец, язык Dafny [76, 78, 35], который я использую в этой книге. Также существуют языки аннотаций, которые были добавлены к существующим языкам программирования: ACL2 (для Applicative Common LISP) [71], VeriFast (для C и Java) [64], набор инструментов KeY (для Java) [2], OpenJML (для Java с аннотациями JML) [105, 26, 66], набор инструментов Frama-C (для C) [14], Stainless (для Scala) [118], Prusti (для Rust) [5], Nagini (для Python) [45], Gobra (для Go) [4] и LiquidHaskell (для Haskell) [86]. В примечаниях в конце некоторых глав я буду отмечать альтернативные обозначения или другие отличия этих инструментов, чтобы упростить применение идей и опыта, представленных в этой книге, и к этим языкам и инструментам.

Материал

Эта книга была написана для поддержки студентов университетов второго года обучения, изучающих информатику. Она также может служить всесторонним введением в верификацию для инженеров-программистов, разрабатывающих промышленное программное обеспечение, плохо знакомых со спецификациями и верификацией и желающих применить эти методы в своей работе.

Книга предполагает наличие у читателя базовых знаний программ и программирования. Стиль этого программирования (функциональный, императивный) и конкретный используемый язык не так важны, но будет полезно, если используемые читателями языки программирования поддерживают концепцию типов.

Книга также предполагает знание некоторых основ логики. Операторы «и», «или» и «не», широко используемые в программировании, будут иметь большое значение, но не менее важно уверенное владение импликацией (логическим следствием). Например, ожидается, что читатель сможет понять значение такой формулы, как

```
2 <= x ==> 10 <= 4 * (x + 1)
```

В приложении B рассматриваются некоторые полезные логические правила, но его нельзя считать введением в логику. Для этого я бы порекомендовал семестровый курс логики.

Помимо основ логики, важную роль в доказательстве корректности программ играют такие понятия, как математическая индукция и фундированные упорядочения. Эти понятия будут разъясняться в книге по мере необходимости.

Книга разделена на три части. Часть 0 охватывает некоторые основы написания доказательств. Часть 1 посвящена функциональным программам (спецификациям и их доказательству), а часть 2 – императивным программам. Кроме некоторых перекрестных ссылок, части 1 и 2 независимы друг от друга.

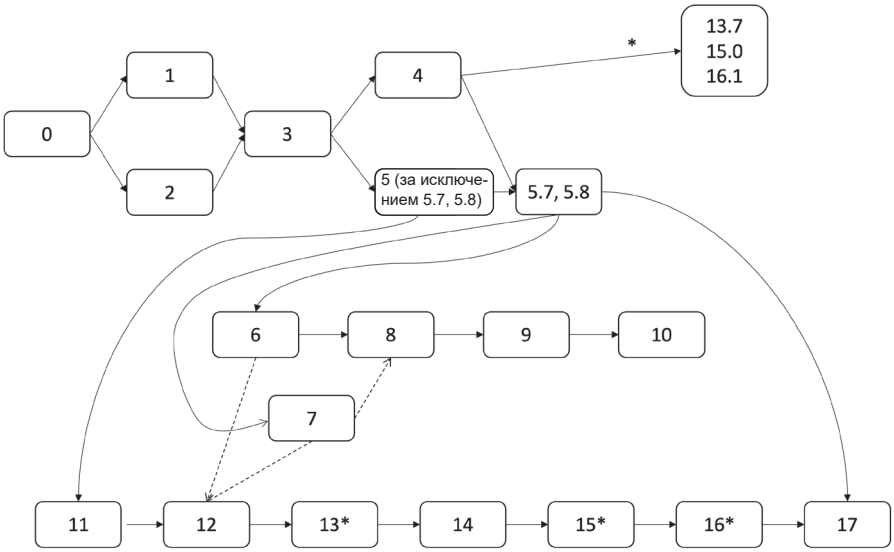
Чем не является эта книга

Ниже перечислено то, чего нет в этой книге.

- Это не руководство по программированию для начинающих. В книге предполагается, что читатель имеет опыт написания (а также компиляции и запуска) простых программ на функциональном или императивном языке. Это предположение кажется разумным для изучающих информатику на втором курсе университета.
- Это не руководство по логике для начинающих, тем не менее в приложении B вы найдете обзор некоторых полезных логических правил и упражнений.
- Это не справочник и не руководство по языку Dafny. Основное внимание уделяется обучению доказательству корректности программ. Однако для поддержки этого обучения в книге объясняются некоторые языковые конструкции Dafny и приемы их использования, а в приложении A представлена шпаргалка по языку.
- Это не исследовательская работа. Существует множество (зрелых и развивающихся) методов рассуждения о программах и множество полезных парадигм программирования, которые здесь не рассматриваются. Обсуждение математического аппарата и мотивов, лежащих в основе этих методов, выходит за рамки этой книги. Вместо этого книга фокусируется на обучении базовым понятиям и представляет эффективные практики.
- Книга не учит *создавать* верификаторы программ. На протяжении всей книги я рассматриваю верификатор как черный ящик, а главной темой является процесс построения доказательств вручную, что служит хорошей практикой взаимодействия с любым верификатором.

Как читать эту книгу

На рисунке ниже показан примерный граф зависимостей глав.



Разделы 13.7, 15.0 и 16.1 опираются на сведения в главе 4, но остальные разделы этих глав – нет. Пунктирные линии показывают рекомендуемые зависимости – например, будет полезно (но не обязательно) прочитать главу 7 перед главами 8 и 12 или прочитать главу 6 перед главой 12.

Dafny

Все спецификации, программы и доказательства корректности программ в книге написаны с использованием языка программирования Dafny и могут быть проверены с помощью системы верификации Dafny. Вообще говоря, конструкции языка Dafny поддерживают четыре вида конструкций.

- Конструкции императивного программирования, такие как операторы присваивания, циклы, массивы и динамически выделяемые объекты. Простейшие из них являются основой многих классических методов доказательства корректности программ.
- Конструкции функционального программирования, такие как рекурсивные функции и алгебраические типы данных. В Dafny они действуют, так же как в математике; например, функции детерминированы и не могут изменить состояние программы.
- Конструкции для написания спецификаций, такие как предусловия, инварианты цикла и меры завершимости. На способы их интеграции в язык повлияли новаторский язык Eiffel и Java Modeling Language

(JML). Спецификации могут использовать любые возможности функционального языка, что делает их достаточно выразительными.

- Конструкции для разработки доказательств, такие как леммы и вычисление доказательств.

Эти различные возможности прекрасно сочетаются друг с другом. Например, все конструкции используют один и тот же язык выражений; выражения, включая цепочки выражений (например, `0 <= x < y < 100`), импликацию (`==>`), кванторы (`forall`, `exists`) и множества (например, `{2, 3, 5}`), часто встречаются в спецификациях и математических выкладках, но также могут использоваться в программах; методы, функции и доказательства могут связываться с локальными переменными; в методах оператор `if` разделяет поток управления, а в леммах – цели доказательства; переменные могут быть отмечены как `ghost`, что делает их пригодными для абстракции, но в остальном они ведут себя подобно обычным скомпилированным переменным; а индукция достигается простым рекурсивным вызовом леммы, где условие завершимости указывается и проверяется так же, как в методах и функциях.

Язык Dafny не только имеет универсальный синтаксис, но и широкую область применения. Инструменты разработки Dafny доступны для Windows, MacOS и Linux. VS Code поддерживает автоматический запуск верификатора. Программы на Dafny компилируются в выполняемый код для нескольких языковых платформ, включая .NET, Java, JavaScript и Go. Сам набор инструментов свободно доступен по адресу:

```
github.com/dafny-lang/dafny
```

Еще до выхода этой книги Dafny использовался для обучения более десяти лет. Он также использовался в нескольких впечатляющих исследовательских проектах (например, в Microsoft Research, VMware Research, ConsenSys R&D, CMU, U. Michigan и MIT) и в настоящее время используется в промышленности (например, в Amazon Web Services).

Дополнительная информация

Дополнительную информацию для этой книги можно найти по адресу:

```
www.program-proofs.com
```

Благодарности

Я хочу поблагодарить всех, кто оказал помощь в создании этой книги.

Я выражаю глубокую благодарность Радживу Джоши (Rajeev Joshi), Розмари Монахан (Rosemary Monahan), Брайану Парно (Bryan Parno), Чезаре Тинелли (Cesare Tinelli) и особенно Грэму Смиту (Graeme Smith), использовавшим ранние версии этой книги в своей преподавательской деятельнос-

ти. Книга сильно выиграла от их отзывов и отзывов их студентов.

Подробные комментарии Раджива Джоши (Rajeev Joshi), Янника Мой (Yannick Moy), Жана-Кристофа Филлиетра (Jean-Christophe Filliâtre), Питера Мюллера (Peter Müller) и Рана Эттингера (Ran Ettinger) превысили все мои ожидания и были действительно полезны! Я также получил ценные отзывы от Нады Амин (Nada Amin), Натана Чонга (Nathan Chong), Дэвида Кока (David Cok), Джоша Каупера (Josh Cowper), Микаэля Майера (Mikaël Mayer), Гауравы Партасарати (Gaurav Parthasarathy) и Робина Салкельда (Robin Salkeld).

Я благодарен за поддержку Байрону Куку (Byron Cook) и Рето Крамеру (Reto Kramer) из Automated Reasoning Group в Amazon Web Services, где я работаю.

Термин «дедуктивная верификация программ» как название раздела науки и техники, о котором идет речь в книге, был предложен Ником Свами (Nik Swamy).

Для написания и оформления текста этой книги я использовал систему Madoko и благодарю Даана Лейена (Daan Leijen) за создание Madoko и за помощь в ее настройке.

Огромный привет Калебу (Kaleb), нарисовавшему забавные иллюстрации к главам.

И наконец, спасибо Гвен (Gwen) за ее поддержку и бесчисленные выходные, которые мы провели в кафе, пока я писал книгу.

Спасибо вам всем!

К. Р. М. Л.

Примечания для преподавателей

В этой книге большое внимание уделяется выбору и порядку подачи материала. Далее я более подробно опишу цель и мотивацию глав. Если вы студент и просто хотите начать читать книгу, то переходите сразу к главе 0. Если вы преподаватель и хотите составить план обучения, то этот раздел для вас.

Часть 0

Если вы хотите не только весело бултыхать ногами в воде, я настоятельно рекомендую научиться плавать и обратить особое внимание на главы 3, 4 и 5 в части 0.

Dafny предлагает высокий уровень автоматизации. Он позволяет писать программы, обрабатывающие массивы в циклах и описанные в главе 13, просто описывая необходимые пред- и постусловия и инварианты циклов, не утомляя себя деталями доказательств. На практике, начав писать более сложные программы, вы неизбежно столкнетесь с ситуацией, когда существующих инструментов автоматизации будет недостаточно. Специалисту, занимающемуся доказательством корректности программ на практике, необходимо знать, как действовать в таких ситуациях, и часть 0 поможет вам заложить необходимый фундамент.

Глава 0

Глава 0 закладывает основу для всей книги и содержит инструкции по установке интегрированной среды разработки (IDE) Dafny.

Главы 1 и 2

В главе 1 представлены некоторые концепции программирования, такие как методы и функции, а также фундаментальные концепции спецификаций, такие как пред- и постусловия.

Если вы хотите начать с формальных основ семантики программ, то в этом вам поможет глава 2 (после знакомства с приложением Б). Если вы хорошо владеете логическими формулами, то вы бы наверняка предпочли ужать главу 2 до одной страницы, содержащей только определения. Но большинство учащихся не способно освоить тему на одних только формулах, поэтому в главе 2 я развиваю повествование постепенно, используя

такие вспомогательные средства, как блок-схемы. Диаграммы отражают основополагающую работу Флойда, и я также включил метод Хоара объяснения поведения программы с помощью троек, а также метод Дейкстры вычисления первого или последнего компонента таких троек с учетом двух других.

Если вас мало интересуют логические формулы, то беглого знакомства с главой 2 будет достаточно для чтения остальной части книги, за одним исключением. В части 2, особенно в главах 13 и 14, я часто вычисляю необходимые условия корректности, применяя слабейшие предусловия к инварианту цикла и обновлению счетчика цикла. Я называю это «обратным проходом». Эти детали подробно описаны в главе 2, и я думаю, что они важны для понимания корректности императивных программ.

Я хочу, чтобы читатели как можно больше могли «думать как программист», поэтому я предварил главу 2 более простым и неформальным взглядом на рассуждения о том, что известно на различных этапах выполнения программы. Для учащихся, уверенно владеющих навыками разработки программ, но плохо разбирающихся в математических формулах, глава 1 может стать хорошей подготовкой к главе 2. Поэтому я рекомендую читать эти главы по порядку, но вообще можно пропустить главу 1 и сразу перейти к главе 2 или, начав с более формальной главы 2, можно затем использовать главу 1 как справочник по синтаксису Dafny.

Глава 3

Глава 3, обсуждающая завершимость, сосредоточена на концепции фундированного порядка и ее применении к рекурсивным вызовам (свойство завершимости циклов рассматривается в главе 11, где представлены рассуждения о циклах). Может показаться странным поместить главу, посвященную завершимости, почти в самое начало книги. Действительно, многие программы на Dafny можно писать, не отвлекаясь на проблему завершимости. Это объясняется тем, что Dafny решает большую часть проблем с завершимостью полностью автоматически. Однако я обнаружил, что в тот день, когда пользователь впервые сталкивается с программой, требующей ручного вмешательства в доказательство завершимости, возникающее удивление и потребность в обретении дополнительных знаний, необходимых для понимания того, что делать дальше, требуют отвлечения на ознакомление с дополнительными материалами. Поэтому я посчитал, что завершимость – это достаточно простая тема, чтобы обсудить ее на ранних этапах, прежде чем переходить к другим, более сложным проблемам. Кроме того, изучение некоторых доказательств завершимости дает учащемуся хорошую возможность попрактиковаться во взаимодействии человека с верификатором.

Есть еще одна важная причина, почему тема завершимости рассматривается так рано. Без ее понимания трудно, почти невозможно объяснить математическую индукцию. Глава 5, вся часть 1 и глава 12 в значительной

степени опираются на индуктивные доказательства, поэтому будет полезно, если читатель подойдет к этим главам с хорошим пониманием проблем завершимости.

Я хочу отметить, что я рассуждаю об индукции иначе, чем во многих книгах по математике. Многие методы индукции очень строги в отношении формата базовый случай / гипотеза индукции / шаг индукции. Их часто называют *индуктивными схемами*. Строгий формат упрощает знакомство с индукцией в старших классах, а индуктивные схемы играют важную роль в логике и теории типов, когда требуется формально *обосновать*, почему индукция работает. Но, с моей точки зрения, математическая индукция – это просто рекурсивный вызов лемм. Когда программистов обучают рекурсии, им не говорят о каком-то строгом синтаксическом формате определения рекурсивных вызовов или о какой-то заранее объявляемой «схеме рекурсии», предшествующей телу рекурсивного метода. Нет, программисты просто выполняют рекурсивный (или взаимно рекурсивный) вызов всякий раз, когда необходимо повторно получить поведение метода при меньшем размере задачи. Ключевым моментом *корректности* таких рекурсивных вызовов является их завершимость. Именно так я преподношу индукцию в этой книге: вы свободно можете вызвать любую лемму рекурсивно, но, делая это, убедитесь, что рекурсивный вызов рано или поздно обязательно завершится. Другими словами, завершимость не встроена в какую-то схему рекурсии, а представляет санитарную меру, действие которой вы подтверждаете при любом вызове.

Большинство индуктивных доказательств в этой книге следуют простым схемам индукции с использованием проверенных идиоматических синтаксических форматов. Поэтому не волнуйтесь, если раньше вы использовали индукцию только таким способом. (В разделе 5.6 приводится пример, подчеркивающий разницу.)

Во всяком случае, именно поэтому завершимость рассматривается уже в главе 3.

Глава 4

В главе 4 представлены алгебраические типы данных. Эта простая глава покажется знакомой всем, кто имеет опыт функционального программирования. Алгебраические типы обсуждаются тоже довольно рано по той простой причине, что они отлично подходят для обучения доказательству корректности программ, которому посвящены последующие главы. Типы данных широко используются в части 1, а также в некоторых разделах части 2.

Глава 5

Умение писать доказательства вручную необходимо для разработки любой нетривиальной программы. Глава 5 показывает, как это делается. Основное внимание в ней уделяется формулировке теорем и их доказательствам, поэтому я постарался выбрать наиболее знакомые темы тео-

рем – арифметику и алгебраические типы данных, представленные в предыдущей главе. Я рекомендую прочитать главу 4 перед главой 5, но если вы решите пропустить главу 4, то точно так же можете пропустить разделы 5.7 и 5.8.

Чтобы получить больше практики в выработке доказательств, я советую после главы 5 прочитать главы 6 и 7 в любом порядке.

В математике доказательство играет две роли. Одна из них – представить доказательство другим математикам. Другая – служить инструментом в процессе размышлений над другими теоремами. В отношении программ доказательства играют те же две роли. Когда доказательство проверяется машиной, упомянутая связь устанавливается между пользователем и автоматическим верификатором. Поэтому крайне важно, чтобы доказательства осуществлялись в интерактивном режиме с верификатором, точно так же как важно при изучении иностранного языка общение с его носителями, потому что невозможно выработать свободное владение иностранным языком, просто читая книги.

Поэтому не ограничивайтесь простым *чтением* главы 5. Пробуйте выполнить шаги доказательств самостоятельно, чтобы получить опыт взаимодействия с верификатором. Также старайтесь выполнять упражнения, не заглядывая в ответы. Я более чем уверен, что на этом этапе учащийся будет стремиться погрузиться в программы, и освоение приемов доказательства и взаимодействия с автоматическим верификатором обязательно принесут пользу в будущем.

Часть 1

Часть 1 посвящена спецификациям и доказательствам корректности функциональных программ. Функциональная природа программ дает два преимущества для разработки доказательств. Во-первых, структуры данных неизменяемы, поэтому нет необходимости следить за изменением состояния программы. Другое преимущество – данные и операции имеют тенденцию определяться рекурсивно, что обеспечивает последовательный и естественный способ структурирования доказательств. Даже если большинство ваших программ написано в императивном стиле, в своих спецификациях вы наверняка будете использовать функциональные фрагменты программ. И если язык программирования предлагает и императивные, и функциональные конструкции (подобно Dafny), вы найдете много хороших применений функциональных возможностей в тех частях ваших императивных программ, которые на самом деле являются неизменяемыми.

Главы 6, 7 и 8

В главе 6 представлены основные способы определения и обоснования индуктивно определенных структур данных, в частности списков. В главе 7

приводится индуктивное представление унарных чисел. Хотя сами по себе унарные числа не так интересны, они дают широкие возможности попрактиковаться в навыках доказательства. В главе 8 определены и проверены два алгоритма сортировки.

Главы 9 и 10

В главах 9 и 10 рассматривается структура более крупных программ и немалое внимание уделяется абстракции и сокрытию информации. Эти концепции лежат в основе информатики и играют решающую роль в доказательстве программ. Модуль, обеспечивающий высокий уровень абстракции, часто проще использовать и проверить, чем модуль, раскрывающий своим клиентам слишком много деталей реализации.

Модульность, абстракция и сокрытие информации в равной степени применимы и к императивным программам, но впервые эти темы рассматриваются в части 1. Тем не менее часть 2 можно прочитать, не читая часть 1.

Глава 9 знакомит с некоторыми механизмами организации кода в модули и затрагивает множество небольших дизайнерских решений в выборе, что показывать снаружи, а что скрывать внутри модуля.

Глава 10 знакомит еще с одним краеугольным камнем информатики: инвариантами. Здесь под инвариантами подразумеваются свойства неизменяемых структур данных, а во второй части – состояние перед итерациями цикла (глава 11) и устойчивость состояния изменяемых структур данных (глава 16).

В главах 10 и 16 инварианты структуры данных по соглашению фиксируются в предикате с именем `Valid()`, выбранном по соглашению. Такой подход обеспечивает более единообразную обработку функциональных и императивных окружений, уменьшает количество концепций, необходимых для понимания инвариантов, и всегда позволяет понять, *какие* свойства имеются и *где* они находятся. Недостаток явных предикатов `Valid()` – многословность спецификаций. Различные языки программирования (включая Dafny) предлагают *подтипы предикатов* (также известные как *подмножества типов*, *уточняющие типы* или *зависимые типы*), которые фактически включают предикат `Valid()` в тип.

Если бы в первую часть можно было добавить еще две главы, то я бы тоже осветил их, но, выбирая между ними, я решил отделить друг от друга типы и другие инварианты.

Часть 2

Часть 2 знакомит со способами рассуждения об императивных программах. Хотя перед чтением части 2 необязательно читать часть 1, тем не менее императивные программы тоже используют функции и модули для организации кода и написания модульных спецификаций. В части 2 об этом

упоминается лишь по мере необходимости, а для более полного рассмотрения вопроса рекомендуется обращаться к части 1. Как я уже упоминал, если вы хотите научиться хорошо доказывать корректность императивных программ, то, помимо краткого ознакомления с главой 11, я настоятельно рекомендую предварительно изучить концепции завершимости и доказательств из части 0.

Глава 11

Одной из наиболее заметных конструкций императивного программирования является цикл, и рассуждения о циклах с использованием инвариантов цикла являются темой главы 11. Большинство новичков испытывают затруднения с инвариантами цикла. С точки зрения преподавания я могу дать две рекомендации, которые отражены в книге.

Первая рекомендация – рассматривать инварианты цикла как спецификации циклов, а не как второстепенную мысль, объясняющую, что делает тело цикла. Один из способов сделать это – скрыть тело цикла из поля зрения. На практике для этого можно свернуть тело цикла с помощью функций свертывания блоков кода в IDE. В Dafny, помимо этого, имеется также возможность вообще опустить тело цикла! Например, верификатор Dafny примет и подтвердит корректность следующей программы:

```
method BodylessLoop() {
  var s, n := 0, 0;
  while n < 100
    invariant 0 <= n <= 100 && s == 4 * n
  assert s + n == 500;
}
```

даже притом, что тело цикла опущено. Необходимо отметить, что рассуждать об использовании цикла можно только на основе инварианта, *не* вглядываясь в его тело, а рассуждать о корректности тела можно, не принимая во внимание, где используется цикл. То есть инвариант цикла подобен контракту между кодом, использующим цикл, и реализацией цикла. Эта же идея лежит в основе рассуждений о методах с точки зрения их спецификаций, а не их реализации (как объясняется в главе 1). Тем не менее я обнаружил, что применить эту идею к циклам труднее, чем к методам, что, как я считаю, связано с тем, что тело цикла находится «прямо тут», и новичкам трудно устоять перед искушением заглянуть в тело цикла.

Итак, изучая инварианты цикла, я рекомендую стараться, насколько это возможно, отделять спецификацию цикла от его тела. (То же самое рекомендуют и другие источники, такие как Hehner [61] и Morgan [93].)

Другая моя рекомендация, касающаяся циклов, – избегать циклов **for**. Как только вы освоитесь с инвариантами циклов, циклы **for** станут удобными и краткими. Но до этого момента тот факт, что обновление счетчика

цикла (например, `i := i + 1;`) происходит неявно, значительно усложняет понимание, для какого значения счетчика должен сохраняться инвариант цикла. Кроме того, становится трудно объяснить полезность шага «обратного прохода» от инварианта цикла, не видя, как обновляется счетчик цикла в конце тела.

Поэтому я рекомендую при обучении придерживаться более многословных циклов **`while`** (по крайней мере, до окончания главы 13).

Глава 12

После знакомства с циклами и инвариантами циклов в главе 11 глава 12 представляет важную практическую тему: переход от спецификаций с рекурсивным определением к итеративным реализациям. Эта тема отсутствует во многих книгах по верификации программ, потому что в них предполагается, что доказательства будут выполняться вручную. Когда вы выполняете эту работу с бумагой и ручкой, вы можете изобретать удобные обозначения, которые «понимаете», без соблюдения всех формальностей. Например, вы можете записать количество целых чисел от a до b, удовлетворяющих предикату P, как

$$(\#i :: a \leqslant i < b \wedge P(i)).$$

Это обозначение никак не зависит от того, с какого «конца» вы удаляете элемент. Например, если мы напишем $||P(a)||$ для обозначения 1, когда выполняется $P(a)$, и 0 в противном случае, то для $a < b$ свойства

$$(\#i :: a \leqslant i < b \wedge P(i)) = ||P(a)|| + (\#i :: a + 1 \leqslant i < b \wedge P(i))$$

и

$$(\#i :: a \leqslant i < b \wedge P(i)) = (\#i :: a \leqslant i < b - 1 \wedge P(i)) + ||P(b - 1)||$$

одинаково очевидны. Но если понимание # определяется рекурсивно (индуктивно), что наиболее вероятно в системе автоматизированной верификации (когда # не является встроенным обозначением, см., например, [81]), то вам придется выбрать, на каком конце диапазона $a \leqslant i < b$ происходит рекурсия (индукция). Это влияет на действия, предпринимаемые при проверке цикла, для вычисления, поскольку цикл тоже должен выбрать, в каком направлении (вверх или вниз) изменять свой счетчик. Я видел, как многие (не только новички) застревали на этом этапе, поэтому посвятил этой теме главу 12, которую желательно прочитать, прежде чем переходить к более интересным алгоритмам в главе 13 и далее.

Глава 13

1980-е гг. принесли не только много хорошей музыки, но и несколько отличных книг по доказательству корректности программ. Классические труды Backhouse [12], Cohen [29], Dijkstra и Feijen [41], Gries [55], Hehner [61], Kaldewaij [68], Morgan [93], Reynolds [113] и Van de Snepscheut [122] часто

рассматривали некоторую булеву алгебру, а затем некоторую формальную семантику программы (обычно тройки Хоара или слабейшие предусловия) и, наконец, примеры применения того, что можно назвать «программами с циклами и массивами». Это замечательные учебники. И очень жаль, что в 1980-е не было быстрых процессоров и развитых инструментов, необходимых для автоматического выполнения этих доказательств на компьютере.

Если вы знакомы с этими классическими трудами, то почувствуете себя как дома, читая главу 13, где рассказывается о нескольких забавных и поучительных алгоритмах, многие из которых были созданы под впечатлением от примеров в классических трудах. Если вы решите прочитать лишь несколько разделов из главы 13, то можете пропустить главу 12. Однако желающим хорошо овладеть умением доказывать корректность программ я думаю, что соображения, представленные в главе 12, покажутся не менее поучительными, чем доказательства алгоритмов в главе 13.

Хорошим вводным курсом по доказательству корректности программ мог бы стать обзор булевой алгебры в приложении B, некоторые разделы главы 2, касающиеся семантики программ, и примеры программ с циклами и массивами в главах 11, 12 и 13. Такой курс не даст такой глубины и практических навыков, как начало всей части 0, но позволит учащемуся быстро пройти путь к доказательству свойств небольших интересных программ.

Главы 14 и 15

В главе 15 рассматриваются дополнительные алгоритмы работы с массивами, но, поскольку эти алгоритмы (в отличие от алгоритмов, рассматриваемых в главе 13) *изменяют* массивы, в главе 14 я впервые затрагиваю тему модификации состояния. Эта тема является частью того, что в более общем смысле называется *фреймингом*. Глава 14 дает достаточно полное введение в фрейминг, что делает ее чтение желательным перед чтением главы 15.

Я полагаю, что многие преподаватели в университетах, желающие охватить все основы доказательства корректности императивных программ, сочтут главу 15 подходящей для завершения курса.

Главы 16 и 17

Последние две главы знакомят с более сложными вариантами изменения динамически выделяемых структур данных. Некоторые программисты, работающие на C- и Java-подобных языках, могут почувствовать острую необходимость изучить эти главы. При этом я чувствую себя обязанным отметить, что многие задачи программирования вполне можно решать, используя неизменяемые структуры данных, подобные тем, что описаны в первой части книги. В предыдущем предложении я мог бы сказать «вынужден напомнить, что» вместо «чувствую себя обязанным отметить», но, по моему опыту, этот момент не всегда очевиден для тех, чей основной язык программирования не поддерживает такие типы. Если вашу задачу можно решить с помощью типов данных, а не классов, тогда вам придется

прикладывать меньше усилий по доказательству корректности (обратите внимание, например, что класс в упражнении 16.4 использует тип `List`, а не императивный связанный список, элементы которого содержат указатели друг на друга).

В главах 16 и 17 объясняется, как определять и проверять структуры данных, размещаемые в куче, которые могут меняться со временем. Основное отличие, которое делает эту задачу более сложной, чем определение неизменяемых структур данных в главах 9 и 10, – это фрейминг.

По сути, фрейминг сводится к отслеживанию (или, в некоторых случаях, разделению) всех объектов, используемых как части изменяемой структуры данных. Все это в отношении простых случаев рассмотрено в главе 14, поэтому основным дополнительным ингредиентом в главах 16 и 17 является абстракция, т. е. возможность выбрать фрейм как определенный набор объектов без разглашения точной идентичности этих объектов клиентам.

Интересно отметить, что единственные средства фрейминга и абстракции, которые должны иметься в языке, – это операторы **`modifies`**/**`reads`**, множества и призрачные переменные. Соответственно, эти две последние главы прекрасно дополняют предыдущий материал. Некоторые языки и верификаторы предоставляют возможности ограничения использования ссылок на объекты. Это может упростить некоторые шаблоны спецификаций, но требует объяснения накладываемых ограничений (например, систем владения объектами по образу и подобию Spec# [13] или Rust [115]) или более сложной базовой логики (например, логики разделения [64] или неявных динамических фреймов [98]). Проблемы и задачи одни и те же, поэтому все, что мы узнаем из спецификаций в этой книге, можно перенести и на другие формализмы.

Опущенные сведения

Одни виды программ сложнее поддаются механический верификации, другие легче. Я старался избегать любых осложнений по мере возможности. Например, в книге используется лишь ограниченное количество операций умножения, поскольку умножение нескольких переменных – это область, где автоматическая верификация обычно испытывает некоторые трудности. Если вы решите разработать свои упражнения, то я рекомендую не злоупотреблять умножением (и обязательно попробуйте сначала выполнить эти упражнения самостоятельно).

Другими выражениями, требующими некоторой утонченности, являются кванторы и включения. В книге используется множество кванторов, но я старался избегать вложенных и других сложных кванторов. (Единственная программа, представленная до главы 13, в которой используются кванторы, – это предикат `IsMin` в главе 10.) Квантификаторы играют важную роль, но я считаю неправильным и ненужным погружать студентов в способы работы с квантификаторами, пока они не освоят достаточно хорошо до-

казательство корректности программ и не научатся взаимодействовать с верификатором.

Единственная сложная область, которую мне не удалось обойти, – это *расширяемость* типов коллекций (в частности, мультимножеств и последовательностей). В разделах 10.2.4 и 13.4 я несколько отступаю от своего правила и даю подробные объяснения.

Глава 0

Введение

Многие из нас начинали изучать программирование с простых скриптов, которые выводят сообщения в терминал, отображают на экране прямоугольники разных цветов или, может быть, даже отправляют сообщения в чат на мобильных устройствах. Процесс написания или модификации такой программы заключается в добавлении пары строк кода и последующем запуске программы, чтобы увидеть, какой эффект имели ваши изменения. Когда мы видим, что программа выводит «Привет, Рустан!», рисует прямоугольники синего и пурпурного цвета или автоматически вставляет смайлики, когда в сообщении присутствуют определенные ключевые слова, тогда мы испытываем чувство выполненного долга от того, что программа действует так, как мы хотели. Программа *работает*!

Но не все программы такие. Обычно недостаточно запустить программу один раз, чтобы убедиться, что она всегда будет работать *корректно*. Фактически можно запустить программу много раз и каждый раз наблюдать корректное поведение, но затем кто-то (возможно, клиент нашего электронного бизнеса) найдет способ использовать программу так, что она начинает вести себя не так, как мы предполагали. Программа завершается с ошибкой. Программа портит данные клиентов. Хуже того, программа открывает несанкционированный доступ к личной информации. Хотелось бы быть уверенными, что наши программы не страдают подобными проблемами. Но как определить, корректно ли работает программа? И что вообще подразумевается под корректностью?

Рассуждения о поведении программ являются темой этой книги. К этому предмету лучше подходить после изучения программирования хотя бы в течение одного семестра, а может быть и двух. То есть вы должны

овладеть базовыми навыками программирования, написать несколько программ, с которыми у вас возникли проблемы, попробовать самостоятельно определить, корректно ли работают ваши программы, узнать, как их тестировать и отлаживать, и предпринять несколько попыток исправить их.

Эта книга научит вас, как рассуждать о программах и *доказывать* их корректность, т. е. создавать точные доказательства, показывающие, что программы ведут себя так, как задумано. Такие строгие доказательства также помогут обострить ваше мышление, и вам будет легче понять, как писать *корректные* программы; в конце концов, единственные программы, корректность которых вы можете *доказать*, – это программы, которые *являются корректными*. В процессе доказательства корректности программы часто обнаруживаются ошибки (логические ошибки, дефекты, упущения, опечатки – называйте их как хотите), и исправление ошибок является частью пути к корректности.

Приемы рассуждения о программах попадают в категорию *формальных методов*. Это название подразумевает использование точных математических методов, логики и формальных доказательств. И действительно, на протяжении всей книги я буду использовать строгие доказательства и подробные логические обоснования – настолько подробные, что обоснования могут быть механически проверены компьютером. Основные понятия, входящие в этот формальный процесс (например, понятия *предусловия* или *инварианта*), можно использовать также при неформальном описании поведения программы. Формальный машинный процесс помогает вам не забыть и не упустить какие-либо детали, и его можно будет автоматически применить повторно после внесения изменений в программу.

0.0. Предварительные сведения

Описывая приемы рассуждений о программах в этой книге, я расскажу вам о спецификациях, абстракциях, леммах и (множестве) доказательств. Я не предполагаю, что у вас есть какие-либо предварительные знания по этим темам. Если вы познакомились с основами доказательства методом индукции в школе и еще кое-что помните, то это определенно станет плюсом для вас, но это не является обязательным требованием. (На самом деле если вы хорошо знакомы с такими темами, как индукция, то вы обнаружите, что я рассказываю и думаю о ней несколько иначе, чем общепринято. Я связываю это с тем, что я программист, а не логик.)

Я предполагаю, что у вас есть некоторые навыки программирования. В частности, я предполагаю, что вам знакомы такие понятия, как переменные, связывание переменных со значениями и разрушающее присваивание, условные операторы, циклы и рекурсия. Я предполагаю, что вы понимаете, что выполнение программы является образуемой последовательностью исполнения операторов программы трассой (поток управле-

ния) и что состояние программы определяется значениями переменных программы (данные программы).

При создании программ программисты неизбежно сталкиваются с логикой – логическими значениями (`false` и `true`) и общими операциями с этими значениями. Я предполагаю, что вы встречали такие выражения, как «А и Б», «А или Б» и «не А». В логике их часто обозначают соответственно как

$$A \wedge B \qquad A \vee B \qquad \neg A,$$

тогда как в исходном коде программ (и в этой книге) они чаще записываются как

```
A && B    A || B   !A
```

Логическое «И» также называется *конъюнкцией*, поэтому операнды `&&` мы называем *конъюнктами*. Аналогично логическое «ИЛИ» называется *дизъюнкцией*, поэтому операнды `||` мы называем *дизъюнктами*. Оператор отрицания имеет более высокий приоритет, чем «И» и «ИЛИ», поэтому выражение `!A && B` равносильно выражению `(!A) && B`, и вам придется оставить круглые скобки в выражении `!(A && B)`, если вы захотите выразить условие «не истинно хотя бы одно значение из *A* и *B*».

В спецификациях часто используется логическое выражение «*A* подразумевает *B*». Оно записывается как *A* `==>` *B* или `A ==> B`, это выражение говорит, что «если *A* истинно, то *B* тоже истинно». Это условие можно записать в терминах других операторов как `!A || B`, но стрелка значительно упрощает понимание импликации в спецификациях. Например, представьте, что мы хотим записать условие: «если я в кафе, то я пью эспрессо» и «если я в спортзале, то я пью воду». Его можно сформулировать так:

```
(AtCoffeeShop ==> DrinkEspresso) && (AtGym ==> DrinkWater)
```

Оператор `==>` имеет более низкий приоритет, чем `&&` и `||`, о чем свидетельствует тот факт, что `==>` состоит из трех символов, тогда как остальные операторы состоят всего из двух символов.

Упражнение 0.0

> Запишите импликацию с употреблением кофе и воды, используя форму `!A || B` вместо `A ==> B`. Не забывайте вставлять круглые скобки в нужных местах. Какая форма записи кажется вам более понятной?

Обозначение импликации с помощью стрелки также предполагает упорядоченность операндов `A` и `B`. Если `A ==> B` – это условие, которое всегда выполняется, тогда мы говорим, что *A сильнее*, чем *B*, и `B` *слабее*, чем `A`. Например, `AtCoffeeShop` сильнее, чем `DrinkEspresso`, а `DrinkWater` слабее, чем `AtGym`. Если мы говорим о каком-то условии `A` и добавляем к нему конъюнкт `B`, то

мы говорим, что *усиливаем* A с помощью B, потому что A && B сильнее, чем A (т. е. A && B ==> A выполняется всегда). Аналогично если к условию A добавить дизъюнкт B, то мы говорим, что *ослабляем* A, потому что A || B слабее, чем A (т. е. A ==> A || B выполняется всегда). Самое сильное из всех условий является ложным (**false**), а самое слабое – истинным (**true**).

Здесь и далее, используя сравнения «сильнее», «слабее», «ниже» или «выше», я также допускаю, что сравниваемые операнды могут быть равны. Если потребуется исключить равенство, я буду говорить *строго* сильнее, слабее, ниже или выше.

Теперь перейдем к обозначениям в исходном коде программ. Будет очень хорошо, если вы умеете хотя бы читать синтаксис C- или Java-подобного языка, предполагающий определение типов переменных и сигнатур функций, использование фигурных скобок для заключения блоков кода и применение таких операторов, как == (равенство) и && (конъюнкция, «И»). Проще говоря, если следующий фрагмент, вычисляющий сумму элементов целочисленного массива a, кажется вам понятным:

```
int sum(int[] a, int n) {
  int s = 0;
  for (int i = 0; i < n; i++) {
    s += a[i];
  }
  return s;
}
```

то можете считать себя готовыми к чтению этой книги. Нелишним будет и опыт программирования на функциональном языке, таком как OCaml или Haskell.

0.1. Краткое содержание книги

Я начну с нескольких простых программ, на примере которых мы сможем поговорить о том, что означает *сохранение* свойства в определенной точке программы (глава 1). В этой главе я использую параметризованные процедуры, представлю пред- и постусловия, а также расскажу, как рассуждать о переменных и потоке управления. В главе 2 я представлю формальную трактовку того же материала. Это основа всей книги, и ее важно понимать. В оставшейся части книги я буду полагаться на понимание способов рассуждения о программах, которое вы получите в главах 1 и 2, и буду проводить рассуждения непосредственно на программах, а не продолжать показывать этапы базовой семантики.

Следующая моя цель – подготовить вас к написанию доказательств. Проблема, возникающая повсюду, – своевременная *завершимость*. Проблема завершимости в индукции, рекурсии или циклах становится запутанной и часто переплетается с другими проблемами, вызывая путаницу. Поэтому в главе 3 я рассмотрю саму тему завершимости.

Чтобы научиться индукции и написанию доказательств, нам нужно что-то, для чего мы могли бы писать доказательства. Некоторые структуры данных из функциональных языков идеально подходят на эту роль, поскольку они определяются рекурсивно и носят математический характер. Поэтому в главе 4 я расскажу об индуктивных типах данных. Если у вас уже есть опыт функционального программирования, то вы легко освоите эту главу.

В главе 5 я представлю *леммы* – утверждения, требующие доказательств. Я покажу, как писать вычислительные доказательства и как использовать индукцию.

Эти начальные главы собраны в части 0 и охватывают основы программирования, которые потребуются в остальной части книги. Затем в части 1 мы рассмотрим функциональные программы, а в части 2 – императивные. Части 1 и 2 в основном независимы друг от друга, поэтому после чтения части 0 вы можете сразу перейти к части 2, если вас больше всего интересуют императивные программы.

В главах 6, 7 и 8 части 1 рассматриваются стандартные идеи функционального программирования: списки, унарные числа и некоторые процедуры сортировки. В главах 9 и 10 на более крупных примерах обсуждаются модули, абстрактные функции и инварианты структур данных. Абстракции и инварианты структур данных представлены в части 1, но одинаково важны и для императивных программ.

Часть 2 посвящена императивным программам с изменяемым состоянием, т. е. с переменными, значения которых можно изменять. Для императивных программ характерной программной конструкцией является цикл – мы обсудим ее в главе 11. Здесь я познакомлю вас с инвариантами цикла, еще одним примером общей концепции инвариантов, которые являются краеугольным камнем в рассуждениях о программах. Глава 12 объединяет определения рекурсивных функций и итеративные циклы. Циклы, массивы и кванторы подобны трем горошинам в стручке, и в главе 13 мы рассмотрим их на многочисленных примерах. Чтобы разобраться в программах, изменяющих массивы и объекты в куче (т. е. в области динамической памяти программы), в главе 14 я познакомлю вас с понятием фрейма, позволяющим спецификации описать части кучи. За ней следует глава 15, где представлены дополнительные примеры изменения содержимого массивов. В главе 16 я продолжу рассмотрение кадров и уделю основное внимание классам и объектам, инвариантам классов, абстракции (так же как в главах 9 и 10) и динамическим кадрам, которые используются для задания изменений объектов. Затем в главе 17 мы применим эти методы еще к четырем программам, использующим динамические изменяемые структуры данных.

0.2. Dafny

На протяжении всей книги для иллюстрации основных моментов я использую язык программирования Dafny [78, 76]. Dafny отлично подходит для этой цели, поскольку он был разработан для логических рассуждений и

поддерживает как императивный, так и функциональный стили программирования, а также написание спецификаций, определение математических задач, формулирование лемм и написание доказательств.

Важно отметить, что в Dafny есть верификатор программ, проверяющий корректность программы и все этапы доказательств. Верификатор обеспечивает высокую степень автоматизации, поэтому вам не придется выполнять множество мелких проверок вручную. Фактически, когда я перейду к теме индуктивных доказательств в главе 5, нам придется отключить часть автоматизации Dafny, иначе она выполнит большую часть доказательств за нас, и вы ничему не научитесь.

Язык Dafny уже более десяти лет используется во многих университетах по всему миру. Он также использовался в некоторых проектах по созданию и верификации систем, а также в промышленных приложениях.

Синтаксис Dafny с фигурными скобками для оформления блоков кода похож на Java, как и его императивные циклы и классы [54]. Функциональные конструкции Dafny включают функции и индуктивные типы данных, подобные функциям и типам в языке ML [90] (а также коиндуктивные типы данных, подобные их аналогам в Haskell, но я не буду рассматривать их в этой книге [110]). Eiffel-подобные конструкции спецификации типа («контракты» [89]) проверяются верификатором Dafny. Краткое описание синтаксиса Dafny и его конструкций вы найдете в приложении A.

Dafny – это проект с открытым исходным кодом (***github.com/dafny-lang/dafny***) и поддерживает все основные платформы: Windows, Mac и Linux. Он имеет несколько интегрированных сред разработки (IDE), включая VS Code (***code.visualstudio.com***).

Для начала установите VS Code на свой компьютер. Затем щелкните по кнопке **Extensions** (Расширения), найдите расширение поддержки Dafny от `dafny-lang` и установите его. Эта книга была написана для Dafny версии 4 (обратите внимание, что расширение VS Code использует другую схему обозначения версий, но в нем упоминается базовая версия Dafny). Файлы с исходным кодом программ на Dafny имеют расширение `.dfy`, поэтому после выбора пункта меню **File -> New Text File** (Файл -> Создать текстовый файл) сохраните файл с расширением `.dfy` (например, `MyProgram.dfy`).

Верификатор будет запускаться автоматически по мере ввода программы, поэтому вам не понадобится знание дополнительных команд.

Программы *верифицируются* только в процессе ввода. Чтобы запустить программу, вам нужно сначала щелкнуть правой кнопкой мыши на буфере и выбрать в контекстном меню **Compile** (Компилировать) или **Compile and Run** (Компилировать и запустить) (точкой входа в программу является метод `Main()`). Я ничуть не удивлюсь, если в процессе чтения вы забудете, что программы можно запускать, – большую часть времени мы будем тратить на определение, написание и верификацию программ, и когда, наконец, верификатор перестанет жаловаться, вы будете точно знать, что программа удовлетворяет спецификации (так зачем вообще запускать ее, верно? ☺).

Если вы решите написать свой сценарий сборки для проекта или по какой-то другой причине захотите запустить инструмент `dafny` из командной строки, то обращайтесь к инструкциям по установке на ***github.com/dafny-lang/dafny***.

0.3. Другие языки

Для обучения концепциям доказательства корректности программ я использую Dafny, но эти же концепции применимы и к другим языкам. Например, Жан-Кристоф Филлиетр (Jean-Christophe Filliâtre) написал версии многих программ, представленных в этой книге, на WhyML. Они доступны по адресу ***github.com/backtracking/program-proofs-with-why3*** и могут быть верифицированы с помощью инструмента Why3. Точно так же Питер Мюллер (Peter Müller) собрал версии многих программ из этой книги на Viper, Prusti и Gobra; они доступны на сайте ***www.pm.inf.ethz.ch/program-proofs***.

Аналогичные возможности верификации предоставляют также такие инструменты, как SPARK и F*.

Примечания

Формальность помогает добиться точности. Но, понимая, как рассуждать о программах, можно сохранять точность, не будучи полностью формальными. Кэрролл Морган (Carroll Morgan) ведет курс «Неформальные методы», в котором подчеркивается, что строгость имеет решающее значение для рассуждений о программах, независимо от того, записываете ли вы какие-либо математические формулы или нет [94].

Часть 0

Знакомство с основами

Глава 1

Основы

В этой главе мы пройдемся по некоторым основам программирования, поговорим об условиях, которые всегда выполняются в определенной точке программы, и посмотрим, как писать простые спецификации.

1.0. Методы

Метод – это определение программы, реализующее определенное поведение. Например, вот определение метода `Triple`:

```
method Triple(x: int) returns (r: int) {
  var y := 2 * x;
  r := x + y;
}
```

Этот метод принимает *входной параметр* `x` целочисленного типа и возвращает *выходной параметр* `r`, также целочисленного типа. *Тело* метода, заключенное в фигурные скобки и следующее за сигнатурой метода, – это список *инструкций*, определяющих реализацию метода. В этом примере тело состоит из двух инструкций. Первая объявляет локальную переменную `y`, которой присваивается значение `2*x`. Вторая присваивает сумму входного параметра `x` и локальной переменной `y` выходному параметру `r`.

Упражнение 1.0

Если `x` равно `10`, какое значение метод присвоит параметру `r`?

В Dafny методы могут иметь любое количество входных и любое количество выходных параметров. В теле метода выходные параметры действуют как локальные переменные: их значения можно читать и изменять. В конце тела метода любые значения выходных параметров будут возвращены

вызывающей стороне. Входные параметры тоже можно читать, но их нельзя изменить.

Вот инструкция, демонстрирующая, как вызывается метод:

```
var t := Triple(18); // присвоит t значение 54
```

Рядом с инструкцией я написал комментарий, сообщающий, что в результате этого вызова метода переменная `t` получит значение `54`, т. е. результат умножения числа `18` на `3`. К концу следующей главы мы научимся превращать подобные комментарии в условия, истинность которых можно будет доказать.

1.1. Инструкции assert

Исследовав метод `Triple`, приведенный выше, можно заметить, что он возвращает значение `3*x`. Это наблюдение можно явно зафиксировать в программе, используя инструкцию **`assert`** для задания утверждений (иногда ее называют *встроенным утверждением*, потому что она находится в определенной точке в коде):

```
method Triple(x: int) returns (r: int) {
  var y := 2 * x;
  r := x + y;
  assert r == 3 * x;
}
```

Использование утверждений (или комментариев в некоторых других языках) для записи существенных и неочевидных условий считается хорошей практикой в разработке программного обеспечения. Запись условия помогает нам рассуждать и служит документацией.

В этой книге подобные утверждения играют более важную роль, чем простое документирование. Они являются *целями доказательства*. Чтобы программа считалась корректной, мы должны доказать, что заявленное условие выполняется каждый раз, когда поток управления достигает утверждения. Доказательство должно быть применимо ко всем возможным значениям параметров объемлющего метода. Это означает, что простого *запуска* программы никогда не будет достаточно для доказательства – слишком много входных данных, которые нужно проверить. Вместо этого мы построим математическое доказательство, в котором для обозначения входных данных используем символические *переменные* вместо конкретных значений.

Встретив утверждение в программе, верификатор Dafny немедленно пытается выполнить доказательство. Для утверждения в `Triple` верификатор легко построит доказательство, подтверждающее, что заявленное условие всегда выполняется. В остальных случаях если верификатор не сможет доказать утверждение (например, если мы изменим приведенное выше условие на `3*x + 1`), то он выдаст сообщение об ошибке.

Упражнение 1.1

Напишите метод `Triple` в Dafny IDE. Если вы не допустили никаких опечаток, то внизу должно появиться сообщение «verified» («верифицировано»). Измените заявленное условие на `3*x+1`. Что произойдет?

1.2. Работа с верификатором

Получив от верификатора сообщение об ошибке, необходимо разобраться в проблеме (другими словами, нужно отладить верификацию) или обратиться за помощью к кому-нибудь. И то и другое может быть затруднительно.

Если проблема не очевидна сразу, можно попытаться выяснить ее причины, добавив дополнительные утверждения в различных точках программы. Например, можно добавить еще одно утверждение между двумя приведенными выше инструкциями присваивания. Инструкция утверждения равнозначна вопросу, задаваемому верификатору: «Знаете ли вы, что это условие (всегда) выполняется в этой точке программы?»

В некоторых случаях анализ может помочь обнаружить ошибку в коде. В других случаях мы можем прийти к выводу, что записанное условие неверно, и улучшить свое понимание кода. В третьих случаях условие действительно может всегда выполняться, но недостаток информации не позволяет убедиться в этом. Типичным выходом из таких ситуаций является запись некоторых предусловий или усиление некоторого инварианта; в следующих нескольких главах вы увидите множество примеров такого подхода. Наконец, в четвертых случаях проблема может заключаться в недостаточной «интеллектуальности» верификатора, чтобы выполнять доказательство автоматически, и в этом случае мы должны ему помочь. Как правило, верификатор хорошо справляется с программами, обсуждаемыми в следующих нескольких главах, но для более сложных программ часто необходимо предоставить леммы или шаги доказательства.

Любое сообщение об ошибке, полученное от верификатора, следует воспринимать как *потенциальную* ошибку (в коде, в спецификации или в каком-то вспомогательном объявлении, необходимом для доказательства), а не как свидетельство, что программа не будет работать должным образом. Проще говоря, думайте о верификаторе как о коллеге, преданном своему делу и внимательном к деталям, который постоянно заглядывает через ваше плечо. Иногда он быстро обнаруживает ошибки. Иногда, как и многие наши коллеги, верификатор не сможет вас понять, и тогда вы должны предоставить дополнительную информацию в виде подсказок, которые помогут объяснить, почему программу можно считать корректной.

Программа, поток управления которой достигает ошибочного утверждения (т. е. утверждения, условие которого оценивается как **`false`**), является ошибочной и не продолжит выполнение после утверждения. Другими словами, после утверждения продолжат выполняться только потоки управле-

ния, в которых условие выполняется. Для иллюстрации рассмотрим программу:

```
method Triple(x: int) returns (r: int) {
  var y := 2 * x;
  r := x + y;
  assert r == 10 * x; // ошибка
  assert r < 5;
  assert false; // ошибка
}
```

Первое утверждение выполняется не всегда, поэтому верификатор выдает сообщение об ошибке. Условие в первом утверждении выполняется, только когда `x == 0` (т. е. `3*x == 10*x`). Второе утверждение выполняется всегда, поэтому у верификатора нет претензий к нему. Но даже когда поток управления благополучно минует два первых утверждения, он никогда не пройдет третье, поэтому верификатор сообщает об ошибке.

Упражнение 1.2

Попробуйте ввести эту программу с тремя инструкциями утверждения в Dafny IDE. Измените второе утверждение так, чтобы верификатор жаловался на первые два утверждения, но не на третье.

1.3. Пути потока управления

Поток управления в простых примерах кода, приводившихся до сих пор, имеет только один путь через последовательность инструкций. Однако с помощью операторов `if` и других инструкций условного выполнения фрагмент кода может содержать множество путей. Программа корректна, когда поток управления на *всех* путях выполняется верно.

Например, вернемся снова к методу `Triple`, но на этот раз добавим в него условный оператор[1]:

```
method Triple(x: int) returns (r: int) {
  if x == 0 {
    r := 0;
  } else {
    var y := 2 * x;
    r := x + y;
  }
  assert r == 3 * x;
}
```

Оператор **`if`** создает два пути в потоке управления программы: проходящий через ветвь «then» (когда `x == 0`) и проходящий через ветвь «else»

[1] Обратите внимание, что, в отличие от некоторых других популярных языков, Dafny не требует включать условие в круглые скобки. Без них текст программы выглядит чище.

(когда `x != 0`). Чтобы признать программу корректной, заявленное условие должно выполняться независимо от выбранной ветви.

Рассуждая о пути, пролегающем через ветвь «then», мы можем ограничить свое внимание случаем, когда значение `x` равно `0`, потому что, только когда `x` равно `0`, выполнение программы может пойти по этому пути. Аналогично, рассуждая о ветви «else», мы можем ограничить свое внимание случаем, когда значение `x` не равно `0`, потому что только тогда, когда `x` не равно `0`, программа может пойти по этому пути.

Выбор пути потока управления в обычном операторе **`if`** *детерминирован*. Это означает, что входные данные программы однозначно определяют выбираемый путь. Выбор пути также может быть *недетерминированным*, когда многократное выполнение программы с одними и теми же входными данными может приводить к выбору разных путей.

Пример недетерминированного выбора пути можно найти в операторе **`if-case`** языка Dafny, где условные ветви могут перекрываться. Это иллюстрируется следующей, чрезмерно запутанной версией `Triple`:

```
method Triple(x: int) returns (r: int) {
  if {
    case x < 18 =>
      var a, b := 2 * x, 4 * x;
      r := (a + b) / 2;
    case 0 <= x =>
      var y := 2 * x;
      r := x + y;
  }
  assert r == 3 * x;
}
```

Рассуждая о первой ветви, можно предположить, что `x < 18`, потому что эта ветвь выбирается, только если `x` меньше `18`. Аналогично, рассуждая о второй ветви, можно предположить, что `0 <= x`, потому что эта ветвь выбирается, только если `x` имеет неотрицательное значение. Если значение `x` *одновременно* меньше `18` и неотрицательно, то может быть выбрана любая ветвь; поэтому, чтобы утверждение было верным, оба пути потока управления должны быть проверены на корректность для таких значений `x`. В этом примере обе ветви вычисляют одно и то же значение `r`, но вообще разные ветви могут иметь совершенно разное поведение.

В отношении последнего примера отметим еще два момента. Во-первых, в первой ветви используется *множественное присваивание*, когда сначала вычисляются все выражения справа, а затем полученные значения присваиваются переменным слева. Во-вторых, область видимости локальных переменных, объявленных в ветках **`case`**, ограничена соответствующими ветками, даже если ветки не заключены в пары фигурных скобок.

1.4. Контракты методов

Клиент метода (функции, типа или модуля) – это фрагмент кода, использующий этот метод (функцию, тип или модуль). Рассмотрим клиента метода `Triple`:

```
method Caller() {
  var t := Triple(18);
  assert t < 100;
}
```

Взглянув на тело метода `Triple`, представленного в предыдущих разделах, легко понять, что `t` получит значение `54`, поэтому утверждение в вызывающем коде будет выполнено. Однако в соответствии с принципом *сокрытия информации*, широко используемом в программной инженерии, тело метода считается частным делом этого метода и не должно приниматься во внимание вызывающим кодом. Это позволяет изменять детали реализации метода, не затрагивая вызывающую сторону.

Упражнение 1.3

Как реагирует верификатор на определение `Triple` из раздела 1.0 и только что данное определение `Caller`?

Итак, если вызывающему коду не разрешено заглядывать в тело метода, то как он узнает, что делает вызываемый метод? Ответ на этот вопрос прост: для вызываемого метода дополнительно объявляется *спецификация*. Спецификация описывает поведение метода, но не раскрывает деталей его реализации. Точнее, метод использует «соглашение» или *контракт* между вызывающей стороной и реализацией, в котором указано, на какой результат может рассчитывать вызывающая сторона. То есть контракт дает автору метода возможность описать его интерфейс, не раскрывая деталей текущей реализации. Поскольку в точках вызова используется контракт метода, а не тело, мы говорим, что методы *непрозрачны*.

Контракт метода состоит из двух фундаментальных частей: *предусловия* и *постусловия*. Предусловие определяет, в каких случаях вызывающей стороне разрешено вызвать метод. Это необходимо доказать в каждой точке вызова, и можно предположить, что оно выполняется в начале тела метода. Постусловие необходимо доказать в каждой точке возврата из тела метода, и можно предположить, что оно выполняется после возврата из вызова в точке вызова.

В нашем примере мы можем написать постусловие для `Triple`, которое позволит `Caller` доказать верность утверждения в его теле. Постусловие вводится с помощью ключевого слова **`ensures`**:

```
method Triple(x: int) returns (r: int)
  ensures r == 3 * x
```

```
{
    var y := 2 * x;
    r := x + y;
}
```

Метод `Caller` теперь проверяет то же условие, что и `Triple`.

Наш пример не требует наличия предусловия, но давайте все же объявим его для полноты картины. Предусловие вводится с помощью ключевого слова **`require`**:

```
method Triple(x: int) returns (r: int)
  requires x % 2 == 0
  ensures r == 3 * x
{
  var y := x / 2;
  r := 6 * y;
}
```

Эта версия `Triple` требует, чтобы параметр `x` имел четное значение. И действительно, новое тело реализации полагается на четность `x` – без выполнения предусловия значение, присвоенное `r`, получится меньше `3*x` (поскольку целочисленное деление нечетных чисел выполняет некоторое округление), и верификатор выдаст сообщение об ошибке, что постусловие `Triple` может не выполняться.

Упражнение 1.4

Удалите (или закомментируйте) предусловие в Triple. Какую ошибку выдаст верификатор?

Упражнение 1.5

Напишите две более строгие альтернативы предусловию `x % 2 == 0`, которые тоже выполняют проверку метода `Triple`.

Рассмотрим еще несколько примеров.

1.4.0. Неполная спецификация

Рассмотрим следующую спецификацию метода:

```
method Index(n: int) returns (i: int)
  requires 1 <= n
  ensures 0 <= i < n
```

Предусловие гласит, что `Index` можно вызывать только для положительных целых чисел. Постусловие гласит, что `Index(n)` вернет число `i` в диапазоне от `0` до `n`.

Врезка 1.0

В этой книге, говоря «от `0` *до* `n`», я подразумеваю полуоткрытый интервал возможных значений `i`: `0 <= i < n`. А говоря «от `0` *по* `n`», я подразумеваю закрытый интервал `0 <= i <= n`.

Итак, какое число от `0` до `n` будет возвращено? В спецификации об этом ничего не говорится. Поэтому, рассуждая о вызовах `Index`, мы должны учитывать все такие значения. С точки зрения вызывающей стороны этот метод выглядит как недетерминированный.

Вот один из вариантов реализации метода:

```
method Index(n: int) returns (i: int)
  requires 1 <= n
  ensures 0 <= i < n
{
  i := n / 2;
}
```

Эта реализация корректна: для каждого возможного входного значения `n`, удовлетворяющего предусловию, тело метода выполняет постусловие.

Вот еще один вариант реализации `Index`:

```
method Index(n: int) returns (i: int)
  requires 1 <= n
  ensures 0 <= i < n
{
  i := 0;
}
```

Эта реализация тоже корректна. Поскольку методы непрозрачны (т. е. вызывающая сторона видит только спецификацию метода, а не его тело), то можно изменить тело `Index` с `i := n / 2;` на `i := 0;`, не влияя на корректность поведения вызывающей стороны.

Обе только что рассмотренные реализации являются детерминированными. То есть для заданных входных данных тело метода вычисляет результат совершенно одинаково. Но поскольку методы непрозрачны, вызывающая сторона не может полагаться на детерминированность. Например, в

```
var x := Index(50);
var y := Index(50);
assert x == y; // ошибка
```

верность утверждения невозможно доказать, потому что из спецификации `Index` нам известно только, что `Index` вернет результат

```
0 <= x < 50 && 0 <= y < 50
```

но мы не знаем, как связаны `x` и `y` между собой.

Хотя спецификация допускает любое значение, реализация может быть (и чаще всего является) детерминированной. Эта важная идея называется *неполной спецификацией*. Она точно определяет свободу, обеспечиваемую контрактом реализации вызывающего объекта.

1.4.1. Множественные постусловия

Рассмотрим метод, возвращающий меньшее из двух заданных значений:

```
method Min(x: int, y: int) returns (m: int)
  ensures m <= x && m <= y
```

В этой спецификации говорится, что возвращаемое значение не превышает `x` или `y`. Но она допускает, что возвращаемое значение может быть значительно меньше обоих параметров.

Упражнение 1.6

Напишите реализацию `Min`, которая удовлетворяет постусловию выше, но не всегда возвращает меньшее из двух чисел, `x` и `y`.

Чтобы спецификация соответствовала ожидаемому результату, – меньшему значению из двух чисел, – нужно дополнительно провозгласить, что возвращаемый результат является одним из двух входных значений:

```
method Min(x: int, y: int) returns (m: int)
  ensures m <= x && m <= y
  ensures m == x || m == y
```

Упражнение 1.7

Вот сигнатура метода, вычисляющего `s` как сумму `x` и `y` и `m` – как наибольшее значение из `x` и `y`:

```
method MaxSum(x: int, y: int) returns (s: int, m: int)
```

а. Запишите предполагаемое постусловие этого метода.

б. Напишите метод, вызывающий `MaxSum` с входными аргументами `1928` и `1`. После вызова укажите, какие значения должны получить два выходных параметра. Если верификатор сообщит, что утверждение может быть нарушено, вернитесь к пункту а) и доработайте спецификацию. Это полезный способ «проверки спецификации». Обратите внимание: вы тестируете спецификацию, используя верификатор, а не запуская программу.

в. Напишите реализацию `MaxSum`.

Упражнение 1.8

Пусть есть метод, пытающийся восстановить аргументы x и y по значениям, возвращаемым методом MaxSum из упражнения 1.7. Другими словами, пусть есть метод со следующей сигнатурой и постусловием:

```
method ReconstructFromMaxSum(s: int, m: int) returns (x: int, y:int)
 ensures s == x + y
  ensures (m == x || m == y) && x <= m && y <= m
```

а. Попробуйте написать тело этого метода. Вы обнаружите, что это невозможно. Напишите подходящее предусловие для метода, позволяющее реализовать этот метод.

б. Напишите следующую тестовую программу для проверки спецификации метода:

```
method TestMaxSum(x: int, y: int) {
 var s, m := MaxSum(x, y);
  var xx, yy := ReconstructFromMaxSum(s, m);
  assert (xx == x && yy == y) || (xx == y && yy == x);
}
```

Как можно изменить спецификацию ReconstructFromMaxSum, чтобы обеспечить успешное выполнение утверждения в тестовой программе?

1.5. Функции

Есть еще одна важная конструкция объявления, которую мы часто будем видеть: функции. Как вы знаете из математики, *функция* обозначает значение, вычисленное на основе заданных аргументов. Ключевым свойством функции является ее *детерминированность*, т. е. любые два вызова функции с одинаковыми аргументами возвращают одно и то же значение.

Вот пример объявления функции в Dafny:

```
function Average(a: int, b: int): int {
  (a + b) / 2
}
```

Обратите внимание, что, в отличие от методов, объявляющих некоторое количество выходных параметров, функция объявляет тип результата, и в отличие от тела метода, которое является инструкцией, тело функции является выражением.

Функции можно использовать в выражениях, поэтому можно написать такую спецификацию:

```
method Triple'(x: int) returns (r: int)
  ensures Average(r, 3 * x) == 3 * x
```

> **Врезка 1.1**
>
> Идентификаторы в языке Dafny могут содержать символ `'`. Здесь я объявил метод с именем `Triple'`, которое можно произнести как «трипл штрих».

Упражнение 1.9

Спецификация `Triple'` отличается от спецификации `Triple`.

а. Напишите корректное тело для `Triple'`, которое не соответствует спецификации `Triple`.

б. Как можно усилить спецификацию `Triple'` с минимальными изменениями, чтобы сделать ее эквивалентной спецификации `Triple`?

в. Как можно использовать Dafny, чтобы доказать, что спецификации в пункте «б» действительно эквивалентны?

Пример выше подчеркивает еще одно важное различие между методами и функциями в Dafny: методы непрозрачны, а функции *прозрачны*. Это совершенно необходимо, потому что если бы вызывающие программы должны были понимать функцию только по ее спецификации, то спецификации функций никогда не смогли бы использовать функции, что серьезно ограничило бы то, что можно сказать о функции.

Тем не менее функция может иметь спецификации. Особое значение имеет предусловие функции, сообщающее, при каких обстоятельствах можно вызвать функцию. Например, мы можем ограничить использование `Average` неотрицательными аргументами:

```
function Average(a: int, b: int): int
  requires 0 <= a && 0 <= b
{
  (a + b) / 2
}

method Triple(x: int) returns (r: int)
  ensures r == 3 * x
{
  if 0 <= x {
    r := Average(2 * x, 4 * x);
  } else {
    r := -Average(-2 * x, -4 * x);
  }
}
```

Предусловия функций применяются в местах их вызова, как и предусловия методов.

Поскольку утверждения, предусловия и постусловия используют логические условия, часто случается так, что мы объявляем логические функции (т. е. функции с типом результата `bool`) для использования в спецификациях. Логические функции также часто называют *предикатами*, и Dafny резервирует для этой цели ключевое слово **`predicat`**. Например, следующее объявление

```
predicate IsEven(x: int) {
  x % 2 == 0
}
```

идентично объявлению

```
function IsEven(x: int): bool {
  x % 2 == 0
}
```

1.6. Компилируемые и призрачные конструкции

Наше знакомство с основами я закончу представлением еще одной важной концепции. Для рассуждений о программе часто требуется больше информации, чем доступно компилятору или во время выполнения. Для этой цели в языке Dafny предусмотрено несколько возможностей. Объявления, переменные, инструкции и т. д., которые используются только для целей спецификации, называются *призрачными конструкциями*. Верификатор учитывает все призрачные конструкции, тогда как компилятор элиминирует все призрачные конструкции в процессе создания выполняемого кода. Программные конструкции, которые превращаются в выполняемый код, называются *компилируемыми* или *непризрачными*.

Мы уже видели несколько призрачных конструкций. Пред- и постусловия (объявленные в инструкциях **`require`** и **`ensures`**) – это призрачные конструкции. С их помощью определяется поведение программы и задается контракт между вызывающим кодом и реализациями. Мы также видели призрачную инструкцию: **`assert`**.

Предусловия, постусловия и утверждения проверяются верификатором в процессе ввода программы. Поскольку они являются призрачными конструкциями, компилятор элиминирует их и, соответственно, они не привносят никаких накладных расходов во время выполнения. Даже если бы мы не заботились о накладных расходах во время выполнения, все равно не было бы смысла проверять утверждения во время выполнения, потому что, как только программа прошла проверку верификатором, мы можем быть уверены, что все утверждения будут выполняться в каждом прогоне.

В Dafny есть также несколько видов объявлений, существующих как в призрачной, так и в компилируемой форме. Они выглядят практически одинаково и одинаково обрабатываются верификатором, но цели их ис-

пользования часто различаются. Чтобы объявить призрачными переменную, входной или выходной параметр, метод, функцию или предикат, достаточно предварить объявление ключевым словом **ghost**. Чтобы убедиться в возможности удаления призрачных конструкций, Dafny проверяет, не использует ли компилируемый код призрачных конструкций. Например, нельзя использовать призрачную переменную справа от оператора присваивания значения компилируемой переменной:

```
method IllegalAssignment() returns (y: int) {
  ghost var x := 10;
  y := 2 * x; // ошибка: призрачную переменную нельзя
              // присвоить компилируемой переменной
}
```

Упражнение 1.10

Функция и метод в следующем фрагменте объявлены как компилируемые:

```
function Average(a: int, b: int): int {
  (a + b) / 2
}

method Triple(x: int) returns (r: int)
  ensures r == 3 * x
{
  r := Average(2 * x, 4 * x);
}
```

а. Измените этот пример так, чтобы функция `Average` объявлялась как призрачная. Какое сообщение об ошибке вы получите?

б. Объявите функцию `Average` и метод `Triple` как призрачные. Это допустимо?

в. Объявите функцию `Average` как компилируемую, а `Triple` как призрачную. Это допустимо?

1.6.0. Пример призрачного метода

Позвольте мне проиллюстрировать (бессмысленно) использование призрачных конструкций на примере еще одной версии нашего рабочего примера:

```
method Triple(x: int) returns (r: int)
  ensures r == 3 * x
{
  var y := 2 * x;
  r := x + y;
```

```
  ghost var a, b := DoubleQuadruple(x);
  assert a <= r <= b || b <= r <= a;
}

ghost method DoubleQuadruple(x: int) returns (a: int, b: int)
  ensures a == 2 * x && b == 4 * x
{
  a := 2 * x;
  b := 2 * a;
}
```

Этот призрачный метод вычисляет значения `2*x` и `4*x` и возвращает их в двух выходных параметрах. Как видите, если не считать ключевого слова **`ghost`** перед объявлением метода, в остальном `DoubleQuadruple` выглядит как обычный компилируемый метод.

Эта версия `Triple` вызывает призрачный метод, сохраняя его выходные параметры в двух локальных призрачных переменных: `a` и `b`. В последующем утверждении в ожидаемом условии используются как призрачные, так и компилируемые переменные. При компиляции этого кода постусловие `Triple`, локальные переменные `a` и `b` в `Triple`, вызов `DoubleQuadruple`, утверждение и весь метод `DoubleQuadruple` будут элиминированы.

Врезка 1.2

Пример выше (а также пример метода `Index` в разделе 1.4.0) иллюстрирует примечательную особенность, которую мы часто будем использовать: возможность объединения в цепочку операторов сравнения. Например, операторы `<`, `<=` и `==` можно объединять в одно выражение, чтобы не повторять общие операнды. Цель состоит в том, чтобы объединить парные сравнения. Например, выражение

```
0 <= i < j < a.Length == N
```

имеет тот же смысл, что и

```
0 <= i && i < j && j < a.Length && a.Length == N
```

но короче и проще читается. Инструкцию **`assert`** в последней версии нашего метода `Triple` можно было бы записать так:

```
assert (a <= r && r <= b) || (b <= r && r <= a);
```

1.7. Итоги

В этой главе я представил методы, функции, пред- и постусловия, простые операторы и поток управления, а также объяснил разницу между призрачными и компилируемыми программами.

Метод может иметь любое количество входных и выходных параметров. Для него определяются пред- и постусловия, а его тело представляет собой список инструкций. Тело метода непрозрачно, т. е. вызывающие стороны могут рассуждать о вызовах методов только с точки зрения спецификации. Поведение метода может быть недетерминированным.

Функция может иметь любое количество входных параметров и ровно один результат. Она может иметь спецификацию, а ее тело является выражением. Тело прозрачно, т. е. вызывающие стороны могут заглянуть в тело функции, рассуждая о ее вызове. Поведение функции детерминировано. Функция, возвращающая результат логического типа, называется предикатом.

Инструкция **assert** определяет условие, которое, как ожидается, должно выполняться. Это порождает необходимость доказывания, которое должно быть выполнено прежде, чем программа будет скомпилирована и выполнена. Таким образом, в верифицированной программе можно смело рассчитывать на выполнение всех заявленных условий.

Пред- и постусловия описывают поведение методов. Предусловие ограничивает использование метода, объявляя, при каких обстоятельствах может вызываться метод в каждой точке вызова. Постусловие ограничивает поведение реализации метода, объявляя обстоятельства, которые должны быть проверены для всех путей в потоке управления через тело метода.

Призрачной называют конструкцию, которая служит для объяснения предполагаемого поведения программы. Призрачные конструкции используются верификатором, но элиминируются компилятором. К призрачным конструкциям относятся предусловия, постусловия и утверждения.

Упражнение 1.11

а. Что верификатор должен сказать о двух утверждениях в этой программе?

```
function F(): int {
  29
}

method M() returns (r: int) {
  r := 29;
}

method Caller() {
  var a := F();
  var b := M();
  assert a == 29;
  assert b == 29;
}
```

б. Объясните почему.

в. Как можно изменить программу, чтобы проверить оба утверждения?

Примечания

В языках программирования нет единого мнения о том, какие подпрограммы следует называть процедурами. Например, в C и Python они называются «функциями», а в Rust и Go функции, принимающие параметр объекта-приемника (такой как **`this`**), называют «методами». В этой книге «методами» называются подпрограммы, имеющие тело, представленное списком инструкций, а «функциями» – имеющие тело в форме выражения. Важно отметить, что конструкции, называемые «функциями» в этой книге (а также функции в Dafny), ведут себя как математические функции. Наличие у метода или функции параметра-приемника зависит от того, объявлен ли он/она как член типа (вам придется подождать до главы 16, прежде чем вы увидите параметр-приемник).

Метод с побочными эффектами не подходит для использования в спецификациях. Поскольку в Java есть только методы, язык моделирования Java Modeling Language (JML) [31] позволяет отметить метод как «чистый», что делает его пригодным для использования в спецификациях примерно как математическую функцию. Инструмент OpenJML тоже поддерживает возможность отметить метод как «функцию», которая объявляет его похожим на функцию [32].

В WhyML [20] и F* [53] инструкции и выражения не различаются. Эти языки полагаются на другие механизмы (регионы и типы эффектов соответственно) для отслеживания возможности изменения состояния в их функциях.

Идея записи предусловий, постусловий и утверждений как частей исходного кода программы устарела. Например, они используются в качестве комментариев в ранних трудах, посвященных теме доказательства правильности программ (например, [51, 88]). Стандартный стиль документации языка CLU [87] включает пред- и постусловия (а также форму предложений **modifies**, которые мы увидим в части 2). Язык Euclid для целей верификации программ поддерживает пред- и постусловия как часть синтаксиса [74]. Роль спецификаций в разработке программ подчеркивают VDM [67] и Larch [57]. За дополнительными историческими справками я отсылаю вас к обзорной статье по языкам поведенческих спецификаций [59].

Именно язык Eiffel Бертрана Мейера (Bertrand Meyer) и сопровождающая его методология *проектирования на основе контрактов* (Design by Contract) популяризировали идею написания пред- и постусловий в тексте программы [89]. Эти контракты между вызывающим кодом и реализациями записываются с использованием обычного синтаксиса языка выражений – идея, которая повлияла на разработку JML [66] и Spec# [13].

Понятия призрачных переменных и призрачного кода, используемых в спецификациях, существуют уже не одно десятилетие. Их могут называть по-разному. Например, Owicki и Gries [106] называли их «вспомогательными» переменными [106]. В SPARK [43] и WhyML [20] призрачные функции известны как «логические функции». Более широкий взгляд на призрачные конструкции вы найдете в [48, 59].

И, наконец, важное замечание о целых числах. Тип `int` в Dafny, который я использую на протяжении всей книги, *не ограничен*. Это означает, что целые числа в Dafny ведут себя как в математике, они не имеют максимального или минимального значения. То есть, если ваша программа на Dafny вычисляет большое число (возможно, используя функцию Аккермана, описанную в разделе 3.3.1), складывает числа (как показано в разделе 12.3) или делает что-то столь же невинное, как утроение значения заданного параметра (как это делает метод `Triple` в данной главе), вам не нужно беспокоиться, что величина результата окажется слишком большой. (Разве что числа станут настолько большими, что вашему компьютеру не хватит памяти для их представления ☹ и программа остановится. Dafny не проверяет, достаточно ли ресурсов для запуска ваших программ.) Dafny поддерживает также *ограниченные целые числа*, например которые можно представить с помощью 32 битов, но я не буду использовать их в этой книге. Некоторые языки программирования поддерживают только неограниченные целые числа (например, Python), другие – только ограниченные (например, C и Java), а третьи поддерживают и те и другие (например, SPARK и JML).

Глава 2

Добавляем формальности

В этой главе определяется формальная основа того, что в предыдущей главе было описано неформальным языком. Смысл программ задается именно математическими формулами. Говорят, что они определяют *семантику* программных инструкций.

Все рассуждения в этой книге основаны на так называемой *логике Флойда*. Она использует логические формулы для описания того, что известно до и после каждой инструкции, позволяя нам разбивать рассуждения о программах на рассуждения об отдельных инструкциях. Попутно я определю *тройки Хоара*, способствующие хорошему пониманию семантики, и *слабейшие предусловия*, дающие удобную возможность автоматизации рассуждений о программе.

Формальные определения семантики программ могут показаться излишне перегруженными деталями. Представляйте их как длинные последовательности операций, составляющих умножение, подробно описывающие шаги вычисления результата. Применив такой подход, вы приобретете более глубокое понимание умножения. Затем обнаружите, что некоторые шаги можно сократить. И что еще более важно, сможете использовать умножение в повседневных задачах. Не маленькие шаги, составляющие умножение, позволяют решать задачи, а ваше знание принципов умножения. Умея выполнять эти шаги, вы знаете, как проверить те или иные детали, но на практике выполняете умножение с помощью калькулятора или электронной таблицы.

Дедуктивная верификация программ происходит так же. Выполнение небольших шагов позволяет выстроить цепочку рассуждений о программе. Если вы в чем-то не уверены можете опуститься до уровня базовых шагов

и убедиться в корректности программы (или найти ошибки). Однако на практике вы обычно будете использовать автоматизированный верификатор, выполняющий семантические вычисления за вас.

А теперь начнем углублять в детали. Встречайте: семантика программы!

2.0. Состояние программы

В программе имеется несколько видов *переменных*. К ним относятся входные и выходные параметры методов, а также локальные переменные, объявленные в теле метода. О переменных, доступных в некоторой точке программы, говорят, что они *находятся в области видимости* этой точки. *Состояние* в точке программы – это значения переменных в области видимости этой точки.

Для иллюстрации рассмотрим следующий метод:

```
method MyMethod(x: int) returns (y: int)
  requires 10 <= x
  ensures 25 <= y
⓪
{ ①
  var a, b;
  ②
  a := x + 3;
  ③
  if x < 20 {
    ④
    b := 32 - x;
    ⑤
  } else {
    ⑥
    b := 16;
    ⑦
  }
  ⑧
  y := a + b;
  ⑨
} ⑩
```

На входе в этот метод, т. е. в точке, отмеченной знаком ⓪, в области видимости находится только входной параметр метода (`x`). Внутри тела метода, в точке ①, в области видимости находятся оба параметра, входной и выходной (`x` и `y`). В точке ② после объявления локальных переменных `a` и `b` область видимости включает `x`, `y`, `a` и `b`. То же относится к точкам с ③ по ⑨. В точке выхода из метода ⑩ локальные переменные выходят из области видимости, поэтому она снова включает только параметры (`x` и `y`).

Состояние в точке ⓪, которое называется *начальным состоянием* метода, содержит только значение `x`. Например, `x` может иметь значение `18` или `37`.

Если это число 37, то состояние в точке ⑨ содержит значения 37 (для x), 56 (для y), 40 (для a) и 16 (для b). В точке ⑩, которая называется *конечным состоянием* метода, состояние содержит значения 37 (для x) и 56 (для y). Определить состояние можно, если проследить выполнение тела метода.

Упражнение 2.0

Для начального состояния, содержащего 18 для x, определите содержимое состояния в точке ⑨ и конечное состояние программы.

Как мы видели в предыдущей главе, код, вызывающий методы, несет ответственность за выполнение предусловий вызываемых методов. Следовательно, мы знаем, что начальное состояние удовлетворяет предварительному условию. Например, начальное состояние не может содержать 2 для x, потому что в этом случае не будет выполняться условие `10 <= x`.

Также мы видели в предыдущей главе, что реализация метода отвечает за выполнение соответствующего постусловия. Следовательно, чтобы объявить `MyMethod` корректной реализацией, мы должны проверить выполнение условия `25 <= y` в точке ⑩ для всех возможных путей выполнения в теле метода и для каждого возможного начального состояния.

Проверить это на практике будет весьма затруднительно, если пробовать разные варианты (например, начиная с x, равного 18 или 37) по одному. Поэтому нам нужен способ, позволяющий рассуждать о многих состояниях одновременно. Это делается с помощью логических выражений. Точнее, мы используем логическое выражение для обозначения тех состояний, при которых результат логического выражения принимает значение **`true`**. Например, логическое выражение `15 <= x <= 40` представляет все начальные состояния, в которых значение x находится в диапазоне от 15 до 40.

Другое название логического выражения – *логическая формула* (или просто *формула*) или *предикат*, что просто означает выражение, которое для данного состояния оценивается как **`false`** или **`true`**. О предикате также можно думать как о *характеристике набора состояний*.

Вместо отслеживания изменения состояния в процессе выполнения программы мы будем отслеживать значение предиката. Основная цель этой главы – показать вам, как это делается. Вот несколько примеров того, что нам требуется выяснить:

- если в точке ③ выполняется условие `12 <= x && a % 2 == 0`, то что можно сказать о возможных состояниях в точке ⑨?
- если мы хотим убедиться, что состояние удовлетворяет условию `a > b` всякий раз, когда поток управления достигает точки ⑧, то какое условие должно выполняться в точке ②?
- предположим, что в каждой точке с ① по ⑨ мы записываем предикат программы, в точке ⓪ – предусловие метода, в точке ⑩ – постусло-

вие, и затем проверяем, характеризует ли каждый предикат состояния, достижимые из предыдущих предикатов.

Достаточно ли этого, чтобы убедиться, что метод реализован правильно?

Ответ на первый вопрос можно получить с помощью так называемых сильнейших постусловий, на второй – с помощью слабейших предусловий, а третий пункт представляет идею, лежащую в основе логики Флойда. Слабейшие предусловия также являются основой того, что во второй части этой книги я буду называть «обратным проходом».

2.1. Логика Флойда

Рассмотрим простую программу:

```
method MyMethod(x: int) returns (y: int)
  requires 10 <= x
  ensures 25 <= y
⓪
{ ①
  var a := x + 3;
  ②
  var b := 12;
  ③
  y := a + b;
  ④
} ⑤
```

где я отметил различные точки в программе. Напишем предикат для каждой точки. В начальном состоянии единственное, что нам известно, – это предусловие метода, поэтому запишем его в точке ⓪. В конечном состоянии единственное, что нас интересует, – это постусловие метода, поэтому запишем его в точке ⑤:

```
⓪ 10 <= x
⑤ 25 <= y
```

Итак, в точке ⓪ мы видим, что нам дано, а в точке ⑤ – нашу цель доказательства.

Проследим выполнение программы, начав с исходного состояния. Нам известно, что условие `10 <= x` в точках ⓪ и ① выполняется. Соответственно, в каждом последующем состоянии нам *известно* следующее:

```
① 10 <= x
② 10 <= x && a == x + 3
③ 10 <= x && a == x + 3 && b == 12
④ 10 <= x && a == x + 3 && b == 12 && y == a + b
```

Я пока не рассказал вам, как придумал эти предикаты, но надеюсь, что они покажутся вам правдоподобными. Теперь, чтобы показать, что тело метода соответствует его спецификации, мы должны показать, что формула в позиции ④ логически подразумевает формулу в позиции ⑤. Другими словами, мы должны показать, что

```
10 <= x && a == x + 3 && b == 12 && y == a + b ==> 25 <= y
```

является *истинной* формулой, т. е. формулой, которая дает **`true`** для любых значений ее переменных.

Упражнение 2.1

Докажите – настолько строго, насколько сможете, – что формула действительно истинна.

Мы заполнили формулы для точек с ① по ④, проследив выполнение в прямом направлении. Каждая формула описывает состояние, образующееся по достижении этой точки, и каждая формула говорит «это то, что у нас есть» или «это то, что нам известно».

Однако формулы можно выводить и в обратном порядке 😮. В этом случае мы начинаем с *желаемого* конечного состояния и следуем в обратном направлении к начальному состоянию. Каждая формула при этом говорит «это то, что нам нужно» или «это условие, которое должно выполняться».

Давайте попробуем. Нам нужно, чтобы в точке ⑤, как и в точке ④, выполнялось одно и то же условие, а именно `25 <= y`. Соответственно, в каждом предыдущем состоянии нам нужно, чтобы выполнялись условия:

```
④ 25 <= y
③ 25 <= a + b
② 25 <= a + 12
① 25 <= x + 3 + 12
```

Теперь, чтобы показать, что тело метода соответствует спецификации, мы должны показать, что формула в точке ⓪ логически подразумевает формулу в точке ①. То есть мы должны показать, что

```
10 <= x ==> 25 <= x + 3 + 12
```

это истинная формула.

Упражнение 2.2

Приведите строгое доказательство, что формула действительно истинна.

Выбор направления, в котором заполняются формулы, не имеет значения. Важно лишь определить формулу для каждой точки программы. Имея формулы, мы можем проверить для каждой инструкции `S`, приводит ли со-

стояние, удовлетворяющее формуле до S, к состоянию, удовлетворяющему формуле после S. Также можно проверить, подразумевает ли предусловие метода формулу, соответствующую начальному состоянию (⓪ ==> ①), и подразумевает ли формула конечного состояния постусловие метода (④ ==> ⑤).

Этот способ рассуждений о программе был описан Робертом Флойдом (Robert W. Floyd) в его статье «Assigning Meanings to Programs» [50] и поэтому известен как *логика Флойда*.

2.2. Тройки Хоара

Нам нужно как-то обозначить центральный строительный блок логики Флойда, который можно обобщить как «инструкция, преобразующая некоторое предшествующее состояние в последующее». Воспользуемся тройками, введенными для этой цели Ч. А. Хоаром (C. A. R. Hoare) [63]. Обозначим через P предикат предшествующего состояния программы S, а через Q – предикат последующего состояния. Тогда *тройка Хоара*

```
{{ P }} S {{ Q }}
```

говорит, что если S выполнить, начиная с любого состояния, удовлетворяющего P, то S не вызовет аварию (и не приведет к другим отрицательным последствиям) и завершится в каком-то состоянии, удовлетворяющем Q[1].

Например, тройка Хоара

```
{{ x == 12 }} x := x + 8 {{ x == 20 }}
```

говорит, что если инструкцию `x := x + 8` выполнить, начиная с состояния, где значением x является 12, то она гарантированно завершится и приведет к состоянию, где значением x является 20. Это верно в отношении инструкции `x := x + 8`. То есть мы говорим, что эта тройка Хоара *истинна* или *корректна*.

> **Врезка 2.0**
>
> В обсуждениях семантики программ и в некоторых других местах в книге я могу опускать знаки препинания, обязательные в языке программирования Dafny. Например, в предыдущем абзаце я записал инструкцию присваивания как `x := x + 8`. Язык Dafny требует завершать инструкции точкой с запятой, например так: `x := x + 8;`.

Другой пример, тройка

```
{{ x < 18 }} S {{ 0 <= y }}
```

[1] В основополагающей статье Хоара, где он представил свои тройки, отсутствие сбоев и завершение не рассматривались [63]. Здесь я использую распространенную вариацию троек, которая учитывает эти проблемы «тотальной корректности» (см. [3]).

обозначает, что всякий раз, когда выполняется инструкция S с состоянием, где x меньше 18, то S приведет к состоянию, в котором y имеет неотрицательное значение. Эта тройка корректна для инструкции `y := 5` или `y := 18 - x`, но некорректна для инструкции `y := x` или `y := 2 * (x + 3)`, потому что две последние инструкции не гарантируют получение неотрицательного значения в y, если x меньше 18. Показательный пример: состояние, в котором x равно -5, удовлетворяет условию `x < 18`, но выполнение инструкции `y := x` или `y := 2 * (x + 3)` с этим состоянием приведет к состоянию с отрицательным значением y.

Для краткости и без риска путаницы мы обычно говорим о программе, «начинающейся с P» или «заканчивающейся с Q», когда имеем в виду «программа запускается с состоянием, удовлетворяющим предикату P» или «программа гарантированно завершится без сбоев и с состоянием, удовлетворяющим предикату Q».

Упражнение 2.3

Приведите строгое объяснение, почему корректна каждая из следующих троек.

а. `{{ x == y }} z := x - y {{ z == 0 }}`

б. `{{ true }} x := 100 {{ x == 100 }}`

в. `{{ true }} x := 2 * y {{ x - четное }}`

г. `{{ x == 89 }} y := x - 34 {{ x == 89 }}`

д. `{{ x == 3 }} x := x + 1 {{ x == 4 }}`

е. `{{ 0 <= x < 100 }} x := x + 1 {{ 0 < x <= 100 }}`

Упражнение 2.4

Для каждой из следующих троек найдите начальные значения x и y, при которых тройка некорректна.

а. `{{ true }} x := 2 * y {{ y <= x }}`

б. `{{ x == 3 }} x := x + 1 {{ y == 4 }}`

в. `{{ true }} x := 100 {{ false }}`

г. `{{ 0 <= x }} x := x - 1 {{ 0 <= x }}`

Упражнение 2.5

Для каждой из следующих троек придумайте предикат для подстановки на место вопросительного знака, который сделает ее корректной тройкой Хоара. Постарайтесь определить ваши условия максимально точно.

а. `{{ 0 <= x < 100 }} x := 2 * x {{ ? }}`

б. `{{ 0 <= x <= y < 100 }} z := y - x {{ ? }}`

в. `{{ 0 <= x < N }} x := x + 1 {{ ? }}`

Упражнение 2.6

Для каждой из следующих троек придумайте предикат для подстановки на место вопросительного знака, который сделает ее корректной тройкой Хоара. Постарайтесь определить ваши условия максимально точно.

а. `{{ -128 <= x < 0 }} x := 1 - x {{ ? }}`

б. `{{ 0 <= x <= y < 100 }} y := y - x {{ ? }}`

в. `{{ x - четное && y < 100 }} x, y := y, x {{ ? }}`

Упражнение 2.7

Для каждой из следующих троек придумайте предикат для подстановки на место вопросительного знака, который сделает ее корректной тройкой Хоара. Постарайтесь определить ваши условия максимально обобщенно.

а. `{{ ? }} x := 400 {{ x == 400 }}`

б. `{{ ? }} x := x + 3 {{ x - четное }}`

в. `{{ ? }} x := 65 {{ y <= x }}`

Упражнение 2.8

Для каждой из следующих троек придумайте предикат для подстановки на место вопросительного знака, который сделает ее корректной тройкой Хоара. Постарайтесь определить ваши условия максимально обобщенно.

а. `{{ ? }} b := y < 10 {{ b ==> x < y }}`

б. `{{ ? }} x, y := 2*x, x+y {{ 0 <= x <= 100 && y <= x }}`

в. `{{ ? }} x := 2*y {{ 10 <= x <= y }}`

2.3. Сильнейшие постусловия и слабейшие предусловия

В разделе 2.1 я обрисовал два подхода к генерации, или *выводу*, формул в логике Флойда для доказательства корректности программы. Первый предполагает вывод формул в прямом направлении, которое вы можете назвать подходом «что мы имеем», а второй – в обратном направлении, и его можно назвать подходом «что нам нужно». Пришло время определить эти два подхода более точно и систематически.

Выполняя вывод формул в прямом направлении, мы конструируем постусловие инструкции на основе заданного предусловия. Существует множество предикатов, условие которых выполняется после завершения инструкции. Например, все следующие тройки Хоара корректны:

```
{{ x == 0 }} y := x + 3 {{ y < 100 }}
{{ x == 0 }} y := x + 3 {{ x == 0 }}
{{ x == 0 }} y := x + 3 {{ 0 <= x && y == 3 }}
{{ x == 0 }} y := x + 3 {{ 3 <= y }}
{{ x == 0 }} y := x + 3 {{ true }}
```

Оказывается, что если обе тройки, `{{ P }} S {{ Q0 }}` и `{{ P }} S {{ Q1 }}`, корректны, то и тройка `{{ P }} S {{ Q0 && Q1 }}` корректна. Тогда при прямом выводе естественно желание вычислить сильнейший, т. е. самый точный, предикат, характеризующий последующее состояние. Прямой вывод вычисляет то, что называется *сильнейшим постусловием* инструкции (относительно заданного предиката, характеризующего предшествующее состояние).

Точно так же существует множество предикатов, характеризующих предшествующее состояние инструкции, которые гарантируют, что инструкция приведет к состоянию, удовлетворяющему данному предикату, характеризующему последующее состояние. Например, все следующие тройки Хоара корректны:

```
          {{ x <= 70 }} y := x + 3 {{ y <= 80 }}
{{ x == 65 && y < 21 }} y := x + 3 {{ y <= 80 }}
          {{ x <= 77 }} y := x + 3 {{ y <= 80 }}
{{ x*x + y*y <= 2500 }} y := x + 3 {{ y <= 80 }}
          {{ false }} y := x + 3 {{ y <= 80 }}
```

Если обе тройки, `{{ P0 }} S {{ Q }}` и `{{ P1 }} S {{ Q }}`, корректны, то и тройка `{{ P0 || P1 }} S {{ Q }` тоже корректна, а это предполагает, что мы должны вычислить слабейший, т. е. наиболее общий, предикат, характеризующий предшествующее состояние. Обратный вывод вычисляет то, что называется *слабейшим предусловием* инструкции (относительно заданного предиката последующего состояния).

Этот обратный вывод «что нам нужно» может показаться более трудным для понимания, чем прямой вывод «что мы имеем». Но, как оказывается, обратный вывод формул на самом деле проще. Вот как он производится. Если у вас есть инструкция присваивания `x := E`, для которой нужно определить формулу `Q`, то перед присваиванием должна соблюдаться формула `Q`, в которой все вхождения `x` заменяются на `E`. Другими словами, наиболее общее условие, которое можно вписать вместо знака вопроса в `{{ ? }} x := E {{ Q }}`, – это формула `Q`, в которой все вхождения `x` заменены на `E`. Эту формулу можно записать с использованием нотации `Q[x := E]`.

Например, если присваивание `y := a + b` в программе должно давать состояние, удовлетворяющее условию `25 <= y` (см. программу в разделе 2.1), то перед присваиванием должна выполняться формула `25 <= y`, в которой вместо `y` используется `a + b`, т. е. `25 <= a + b`. В тройках Хоара самое общее условие, которое можно подставить на место вопросительного знака в

```
{{ ? }} y := a + b {{ 25 <= y }}
```

имеет вид `25 <= a + b`. Совсем нетрудно, верно? Аналогично, чтобы определить наиболее общее условие, которое можно подставить на место вопросительного знака в

```
{{ ? }} a := x + 3 {{ 25 <= a + 12 }}
```

берем желаемое условие `25 <= a + 12` и заменяем `a` на `x + 3`, что дает `25 <= x + 3 + 12`.

В функциональной программе (т. е. в программе, написанной на функциональном языке программирования) присваивание значения переменной происходит один раз при ее объявлении. (В функциональном программировании это называется *привязкой let*.) В императивной программе оператор присваивания позволяет изменять значение существующей переменной. Более того, правая часть присваивания может ссылаться на саму переменную, что дает возможность обновить переменную с учетом ее предыдущего значения. Например, инструкция

```
x := x + 1
```

увеличит значение `x` на `1`.

Рецепт вычисления самого слабого предусловия также работает, если в правой части присваивания упоминается обновляемая переменная. Например, следуя тому же рецепту, что и раньше, для присваиваний вида `x := x + 1` и `x := 2*x + y` (следующие тройки лучше всего читать справа налево):

```
{{ x+1 <= y }} x := x + 1 {{ x <= y }}
{{ 3 * (2*x + y) + 5*y < 100 }} x := 2*x + y {{ 3*x + 5*y < 100 }}
```

2.3.0. Обмен местами значений переменных

Давайте вычислим слабейшие предусловия, чтобы проверить корректность программы, которая меняет местами значения, хранящиеся в `x` и `y`.

Чтобы указать, что ожидается от этой программы, нужен какой-то способ, который позволит в постусловии сослаться на начальные значения `x` и `y`. В тройках Хоара это достигается за счет введения *логических переменных*, т. е. переменных, которые обозначают некоторые значения в доказательстве, но не могут использоваться в программе. Используя `X` и `Y` в качестве логических переменных, мы пишем:

```
{{ x == X && y == Y }}
var tmp := x;
x := y;
y := tmp;
{{ x == Y && y == X }}
```

Данная программа состоит из трех последовательно выполняемых инструкций. Чтобы вычислить слабейшие предусловия для последовательно-

сти инструкций, мы просто выполняем их по одной, как показывает следующая последовательность цепочек троек Хоара:

```
{{ x == X && y == Y }}
{{ ? }}
var tmp := x;
{{ ? }}
x := y;
{{ ? }}
y := tmp;
{{ x == Y && y == X }}
```

В этой последовательности мы намерены использовать слабейшие предусловия для вычисления предикатов для подстановки взамен вопросительных знаков. По окончании первый вопросительный знак будет заменен слабейшим предусловием всей последовательности инструкций. Затем нам нужно доказать, что это предусловие является слабейшим предусловием. Итак, пойдем по порядку, от конца к началу (см. рис. 2.0).

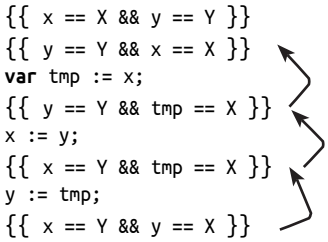


Рис. 2.0. Рассуждения, доказывающие, что этот фрагмент программы меняет местами значения *x* и *y*, показаны в порядке от желаемого постусловия. Здесь видно, что из первой аннотации вытекает вторая

Чтобы получить эти промежуточные формулы, мы просто применяем рецепт вычисления слабейших предусловий. Это похоже на длинную последовательность операций, составляющих умножение: мы систематически применяем шаги, с которыми уже познакомились. Далее нам нужно доказать логическое следствие:

```
x == X && y == Y  ==>  y == Y && x == X
```

В этом случае импликация тривиальна. Итак, мы доказали, что программа, приведенная выше, меняет местами x и y.

2.3.1. Упрощение нотации доказательства корректности программы

Давайте вернемся к упражнению по верификации программы, меняющей местами значения переменных, но на примере другой программы. На этот раз предположим, что x и y содержат целочисленные значения:

```
{{ x == X && y == Y }}
x := y - x;
y := y - x;
x := y + x;
{{ x == Y && y == X }}
```

Построив слабейшие предусловия, получим то, что показано на рис. 2.1.

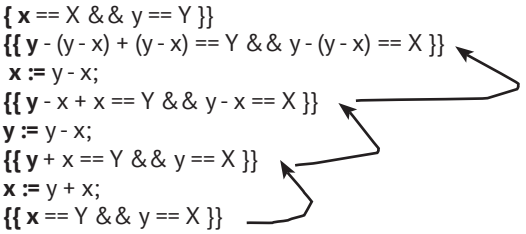


Рис. 2.1. Рассуждения в обратном порядке для доказательства корректности этой программы, меняющей местами значения целочисленных переменных

Заполнив недостающие части в порядке, указанном стрелками, мы просто произвели замены по рецепту конструирования слабейших предусловий. Теперь у нас есть условие корректности:

```
x == X && y == Y
==>
y - (y - x) + (y - x) == Y && y - (y - x) == X
```

После некоторых арифметических упрощений мы видим, что эта импликация справедлива, и программа действительно меняет местами значения переменных.

Создание слабейших предусловий помогает упростить формулы по мере продвижения, а не в самом конце. Удобным приемом в этом случае является запись упрощенной формулы над только что записанной строкой. Проведя несколько упрощений, мы сможем написать несколько формул друг над другом. При желании также можно усилить формулу, продолжая работать в обратном направлении. (Внимание! Распространенной ошибкой является *ослабление* формулы при движении назад. Это недопустимо, потому что не дает доказательства.) В конце концов мы получаем последовательность формул о свойствах программных инструкций. Для каждой пары формул, следующих друг за другом, мы должны доказать, что из первой следует вторая, и каждая инструкция программы между двумя формулами должна быть допустимой тройкой Хоара.

На рис. 2.2 показана предыдущая программа с только что описанными промежуточными упрощениями. Стрелки справа показывают порядок конструирования слабейших предусловий, образующих корректные тройки Хоара. Стрелки слева соединяют последовательные аннотации. Их можно конструировать в обратном направлении как усиления, но многие люди

более склонны читать их как следствия в прямом направлении. Поэтому стрелками слева я показываю направление вперед.

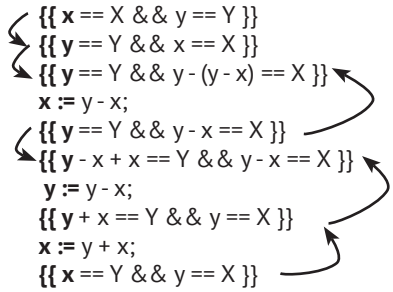


Рис. 2.2. Последовательные аннотации в этом фрагменте программы показывают, как происходит упрощение формул. Формула в одной такой аннотации подразумевает формулу в следующей. Я считаю, что такие упрощения легче читать в прямом направлении (в соответствии с импликацией). Аннотации вокруг инструкций программы, напротив, строятся задом наперед (с использованием слабейших предусловий), как показано стрелками

Упражнение 2.9

Докажите, что следующая программа корректно меняет местами значения переменных. Оператор ^ здесь обозначает операцию XOR (ИСКЛЮЧАЮЩЕЕ-ИЛИ):

```
x := x ^ y;
y := x ^ y;
x := x ^ y;
```

Вы можете использовать правила коммутативности и ассоциативности операции XOR, а также свойства `x: ^ x == 0` и `x ^ 0 == x`.

Упражнение 2.10

Найдите ошибку в следующем доказательстве:

```
{{ x == 0 }}
{{ x == 0 && y == 6 }}
x := x + 2
{{ x == 2 && y == 6 }}
{{ x + y == 8 }}
y := x + y
{{ y == 8 }}
```

2.3.2. Одновременное присваивание

Некоторые языки позволяют присваивать значения одновременно нескольким переменным в одном операторе. Это называется *одновременным присваиванием*. Например,

```
x, y := 10, 3;
```

присвоит переменной `x` значение `10` и одновременно присвоит переменной `y` значение `3`. Другой пример:

```
x, y := x + y, x - y;
```

Эта инструкция вычисляет сумму и разность `x` и `y`, а затем присвоит им эти значения соответственно. Обратите внимание, что все действия в правой части выполняются до присваивания значений переменным. Именно эта одновременность отличает одновременное присваивание от *последовательности* двух присваиваний:

```
x := x + y; y := x - y;
```

Семантика одновременного присваивания ничуть не сложнее присваивания одной переменной. Нам нужно лишь, чтобы одновременно произошла и замена самого слабого предусловия. Точнее, чтобы инструкция `x, y := E, F` соответствовала условию `Q`, перед присваиванием должно быть соблюдено условие `Q[x, y := E, F]`, т. е. условие `Q`, в котором мы одновременно подставляем `E` вместо `x` и `F` вместо `y`.

Одновременное присваивание дает смехотворно простой способ записи программы, меняющей значения двух переменных местами: `x, y := y, x`. Вот ее доказательство:

```
{{ x == X && y == Y }}
{{ y == Y && x == X }}
x, y := y, x
{{ x == Y && y == X }}
```

Упражнение 2.11

Подставьте соответствующие предикаты на место вопросительных знаков в следующих тройках Хоара. Упростите полученные формулы, вычислив слабейшие предусловия.

а. `{{ ? }} x, y := 6, 7 {{ x < 10 && y <= z }}`

б. `{{ ? }} x, y := x + 1, 2 * x {{ y - x == 3 }}`

в. `{{ ? }} x := x + 1; y := 2 * x {{ y - x == 3 }}`

2.3.3. Определение переменных

Локальные переменные определяются с помощью инструкции **`var`**. Чаще всего она используется вместе с оператором присваивания, например `var tmp := x`, как было показано в разделе 2.3.0. До сих пор, рассматривая семантику, я игнорировал фактическое определение переменной (т. е. часть «**`var`**») и все внимание сосредотачивал на присваивании. Но определение переменных тоже имеет семантику, поэтому мы должны думать о **`var`** как о

чем-то отдельном от любого первоначального присваивания и имеющем общий синтаксис. То есть инструкция `var tmp := x` на самом состоит из двух инструкций:

```
var tmp;
tmp := x;
```

Каждая переменная имеет тип, но тип локальной переменной обычно определяется автоматически, поэтому его можно не указывать явно.

Инструкция `var x` определяет переменную `x`, но ничего не обещает относительно ее начального значения (кроме того, что `x` получит некоторое значение своего типа). Таким образом, чтобы `var x` соответствовала предикату `Q`, перед определением переменной нужно, чтобы условие `Q` выполнялось для всех значений `x`. Вот как выглядит соответствующая тройка Хоара:

```
{{ forall x :: Q }} var x {{ Q }}
```

Например, для целочисленной переменной `x` предположим, что `Q` имеет вид `0 <= x < 100`. Тогда допустимой тройкой Хоара будет следующая:

```
{{ forall x :: 0 <= x < 100 }} var x {{ 0 <= x < 100 }}
```

Условие `forall x :: 0 <= x < 100` не будет выполняться, потому что условие `0 <= x < 100` выполняется не для всех целых чисел (пример, оно не выполняется для числа `102`). Это говорит о том, что `var x` не дает никаких гарантий, что `x` окажется в диапазоне от `0` до `100`.

Как правило, если `Q` упоминает `x` каким-то нетривиальным образом, то мы не сможем доказать, что `var x` будет соответствовать `Q`. Но не думайте, что определение переменных никогда не будет полезным – семантика лишь верно отражает тот факт, что неинициализированные переменные могут иметь любые значения. Если имеет место зависимость от значения переменной, то ей сразу же следует присвоить допустимое значение.

В качестве примера докажем, что можно определить локальную переменную для временного хранения значения, которое впоследствии будет присвоено другой переменной. Из самого слабого предусловия присваивания мы знаем, что `x := E` соответствует `Q` при условии, что инструкция начинается в `Q[x := E]`. Когда `var tmp := E; x := tmp` будет соответствовать `Q`, где `tmp` – переменная, не используемая ни в `E`, ни в `Q`? Следующая цепочка троек Хоара показывает, что ответом будет `Q[x := E]`, как и в случае прямого присваивания `x := E`:

```
{{ Q⟦x := E⟧ }}
{{ forall tmp :: Q⟦x := E⟧ }}
var tmp
{{ Q⟦x := E⟧ }}
{{ Q⟦x := tmp⟧⟦tmp := E⟧ }}
tmp := E
```

```
{{ Q⟦x := tmp⟧ }}
x := tmp
{{ Q }}
```

Как обычно, строки выше лучше читать снизу вверх. Упрощение `Q[x := tmp][tmp := E]` до `Q[x := E]` оправдано тем фактом, что `tmp` отсутствует в `Q`. Упрощение

```
forall tmp :: Q⟦x := E]
```

до `Q[x := E]` оправдано тем фактом, что `tmp` не встречается в выражении `Q[x := E]`.

Упражнение 2.12

Вот пример, который не зависит от начального значения `x`. Докажите:

```
{{ true }} var x; x := x * x {{ 0 <= x }}
```

2.3.4. Осложнения

Было бы несправедливо не показать, как вычислить самое сильное постусловие для присваивания, даже притом что мы редко будем использовать его. Это не простая замена, как при определении слабейших предусловий. Все намного сложнее, так что приготовьтесь.

Чтобы рассматривать рецепт поиска самого сильного постусловия не на пустом месте, возьмем за основу следующую, невинную на вид тройку Хоара:

```
{{ w < x < y }} x := 100 {{ ? }}
```

Очевидно, что постусловие имеет вид `x == 100`, но самое ли оно сильное? Условие `w < x < y` больше не выполняется, потому что присваивание перезаписывает предыдущее значение `x`. Однако в последующем состоянии существует некоторое значение x_0 (а именно значение `x` в предшествующем состоянии), для которого выполняется условие $w < x_0 < y$. Соответственно, наиболее точное условие, которое можно записать вместо вопросительного знака, таково:

```
exists x₀ :: w < x₀ < y && x == 100
```

Поскольку задействованные переменные являются целыми числами, эта формула логически эквивалентна формуле:

```
w + 1 < y && x == 100
```

В общем случае правая часть присваивания также может упоминать `x`, поэтому нам нужно использовать x_0 и там. Итак:

```
{{ P }} x := E {{ exists x₀ :: P⟦x := x₀⟧ && x == E⟦x := x₀⟧ }}
```

Разве вы не рады, что спросили??? 😁

Упражнение 2.13

Замените **?** в следующих тройках Хоара сильнейшими постусловиями. В каждом случае попробуйте упростить формулу, если это возможно.

а. `{{ y == 10 }} x := 12 {{ ? }}`

б. `{{ 98 <= y }} x := x + 1 {{ ? }}`

в. `{{ 98 <= x }} x := x + 1 {{ ? }}`

г. `{{ 98 <= y < x }} x := 3 * y + x {{ ? }}`

Упражнение 2.14

Проверьте корректность различных программ перемены значений переменных местами из раздела 2.3.0, используя сильнейшие постусловия.

2.4. WP и SP

Мы можем сэкономить на ненужных объяснениях, введя некоторые обозначения. Предположим, что `S` – это инструкция (statement). Тогда для любого предиката (predicate) `P`, характеризующего предшествующее состояние `S`, обозначим сильнейшее постусловие `S` относительно `P` как *SP*`[S, P]`. Аналогично для любого предиката `Q` в последующем состоянии `S` обозначим слабейшее предусловие `S` относительно `Q` как *WP*`[S, Q]`.

2.4.0. Обратный проход

С использованием этой нотации семантику присваивания можно определить следующим образом:

```
WP⟦x := E, Q⟧ = Q[x := E]
SP⟦x := E, P⟧ = exists x₀ :: P[x := x₀] && x == E[x := x₀]
```

Эти формулы для *WP* и *SP* применимы и к инструкциям присваивания одной переменной, и к инструкциям одновременного присваивания нескольким переменным, если списки переменных и выражений `x`, `x₀` и `E` имеют одинаковую длину.

С инструкцией присваивания очень полезно *работать в обратном направлении*, поскольку в этом случае появляется возможность преобразовать условие `Q`, которое должно выполняться после присваивания, в условие, которое должно выполняться до присваивания. Этот шаг я часто буду использовать во второй части.

Упражнение 2.15

Потренируйтесь в работе в обратном направлении: вычислите самое слабое предусловие следующих программ относительно постусловия `x + y <= 100`.

а. `x := 20`
б. `x := x + 1`
в. `x := 2 * x`
г. `x := -x`
д. `x := y`
е. `x := x + y`
ж. `x := y - x`
з. `x := x - y`
и. `z := x + y`

Упростите каждый свой ответ.

Упражнение 2.16

Вычислите самое сильное постусловие следующих программ относительно предусловия `x + y <= 100`.

а. `x := 5`
б. `x := x + 1`
в. `x := 2 * y`
г. `z := x + y`

Упростите каждый свой ответ.

2.4.1. Локальные переменные

В разделе 2.3.3 я представил семантику **`var`** в терминах слабейших предусловий. Вот его семантика с точки зрения *WP* и *SP*:

```
WP⟦var x, Q⟧  =  forall x :: Q
SP⟦var x, P⟧  =  exists x :: P
```

Упражнение 2.17

Вычислите.

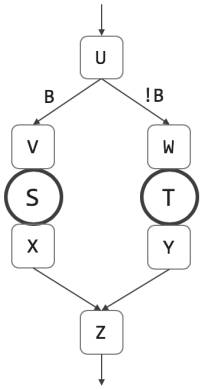
а. *WP*⟦**var** `x, x <= 100`⟧
б. *SP*⟦**var** `x, x <= 100`⟧

2.5. Условное выполнение потока управления

Чтобы понять, что логика Флойда говорит об условных инструкциях, таких как

```
if B { S } else { T }
```

полезно нарисовать граф потока управления. Здесь я использую `U`, `V`, `W`, `X`, `Y` и `Z` для обозначения предикатов в начале и в конце условной инструкции и двух вложенных в нее инструкций `S` и `T`:



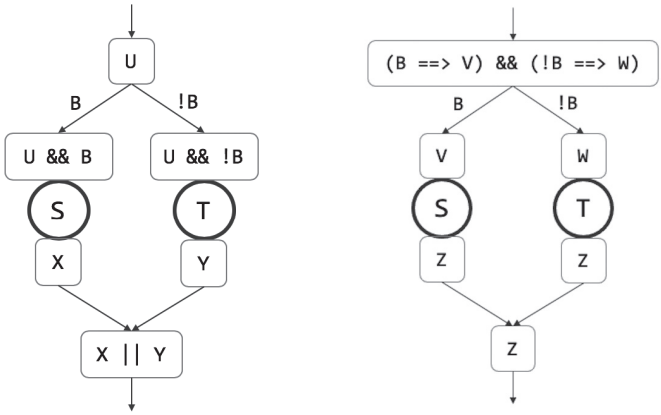
Логика Флойда подсказывает нам, что эта программа корректна, если верны следующие формулы и тройки Хоара:

- `U && B ==> V`;
- `U && !B ==> W`;
- `{{ V }} S {{ X }}`;
- `{{ W }} T {{ Y }}`;
- `X ==> Z`;
- `Y ==> Z`.

Для вычисления сильнейших постусловий мы:

- определим `V` как `U && B` и `W` как `U && !B`;
- вычислим `X` как самое сильное постусловие для `V,S` и `Y` как самое сильное постусловие для `W,T`;
- определим `Z` как `X || Y`.

Графически это выглядит как левая диаграмма на рисунке ниже:



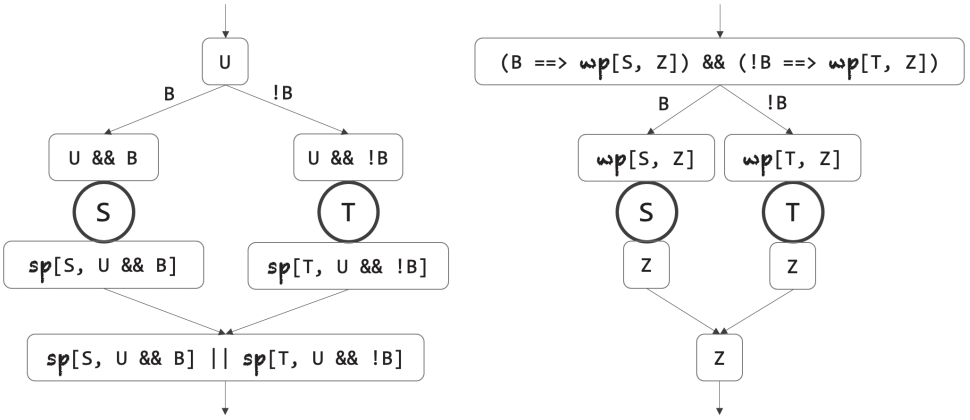
Чтобы вычислить слабейшие предусловия, мы:

- определим `X` и `Y` как `Z`;
- вычислим `V` как самое слабое предусловие для `S,X` и `W` как самое слабое предусловие для `T,Y`;
- определим `U` как `(B ==> V) && (!B ==> W)`.

Графически это выглядит как правая диаграмма на рисунке выше.

Независимо от того, как создаются промежуточные предикаты на этих диаграммах, мы видим, что они преобразуются по мере следования вперед или назад по ребрам графа потока управления.

Вот те же блок-схемы, но с обозначениями *SP* и *WP* на диаграммах:



Упражнение 2.18

а. Нарисуйте блок-схему, подобную приведенной выше, для условной инструкции:

```
if x < 3 { x, y := x + 1, 10; } else { y := x; }
```

б. Действуя в прямом направлении и используя `x + y == 100` в качестве предусловия (т. е. как `U`), заполните формулы в оставшихся полях.

в. Действуя в обратном направлении и используя `x + y == 100` в качестве постусловия (т. е. как `Z`), заполните формулы в оставшихся полях.

2.5.0. Просто формулы, мэм

Смысл инструкций **`if`**, записанный в терминах *SP* и *WP* только с помощью формул, без диаграмм, можно выразить так:

```
SP⟦if B { S } else { T }, P⟧ =
     SP⟦S, P && B⟧ || SP⟦T, P && !B⟧
WP⟦if B { S } else { T }, Q⟧ =
     (B ==> WP⟦S, Q⟧) && (!B ==> WP⟦T, Q⟧)
```

Упражнение 2.19

Предположим, что до выполнения инструкции

```
if x < 20 { y := 3; } else { y := 2; }
```

выполняется условие `x < 100`. Попробуйте определить самое сильное условие после этой инструкции. Другими словами, вычислите самое сильное постусловие инструкции относительно `x < 100`. Упростите условие после его вычисления.

Упражнение 2.20

Предположим, что вы хотите, чтобы после инструкции

```
if x < 20 { y := 3; } else { y := 2; }
```

выполнялось условие `x + y == 22`. Определите, с какими состояниями может начинать выполнение инструкция? Другими словами, вычислите самое слабое предусловие инструкции относительно `x + y == 22`. Упростите условие после его вычисления.

Упражнение 2.21

Вычислите самое слабое предусловие для следующей инструкции относительно `y < 10`. Упростите условие:

```
if x < 8 {
  if x == 5 {
    y := 10;
  } else {
    y := 2;
  }
} else {
  y := 0;
}
```

Упражнение 2.22

Вычислите самое слабое предусловие для следующей инструкции относительно `y % 2 == 0` (т. е. «y – четное»). Упростите условие:

```
if x < 10 {
  if x < 20 { y := 1; } else { y := 2; }
```

```
} else {
  y := 4;
}
```

Упражнение 2.23

Вычислите самое слабое предусловие для следующей инструкции относительно `y % 2 == 0` (т. е. «y – четное»). Упростите условие:

```
if x < 8 {
  if x < 4 { x := x + 1; } else { y := 2; }
} else {
  if x < 32 { y := 1; } else { }
}
```

Упражнение 2.24

Определите, при каких обстоятельствах следующая программа соответствует условию `0 <= y < 100`. Если вы начинаете понимать, как вычислять слабейшие предусловия, то попробуйте выполнить вычисления в уме. Запишите полученный ответ, а затем запишите все вычисления *WP* на бумаге, чтобы проверить себя.

```
if x < 34 {
  if x == 2 { y := x + 1; } else { y := 233; }
} else {
  if x < 55 { y := 21; } else { y := 144; }
}
```

2.6. Последовательная композиция

Другой способ объединить две инструкции – поставить их одну за другой. Эта простая идея получила название *последовательной композиции*. Для инструкций `S` и `T` мы можем передать семантику логики Флойда `S;T`, написав две пересекающиеся тройки Хоара:

```
{{ P }} S {{ Q }} T {{ R }}
```

Эта последовательность верна, если корректны следующие две отдельные тройки Хоара:

- `{{ P }} S {{ Q }}`;
- `{{ Q }} T {{ R }}`.

У нас нет причин вовлекать предикат `Q`, поскольку при выполнении условия `Q` в *SP*⟦`S, P`⟧ или *WP*⟦`T, R`⟧ потери в общности не будет. Другими словами, мы можем напрямую определить корректность тройки Хоара

```
{{ P }} S;T {{ R }}
```

в терминах сильнейших постусловий или слабейших предусловий. В формулах мы имеем:

$$SP[\![S;T,\ P]\!] = SP[\![T,\ SP[\![S,\ P]\!]\ ]\!]$$
$$WP[\![S;T,\ R]\!] = WP[\![S,\ WP[\![T,\ R]\!]\ ]\!]$$

Обратите внимание, что композиции в этих двух правых частях идут в противоположных направлениях: *SP* вычисляется в прямом направлении от `P`, а *WP* вычисляется в обратном направлении от `R`.

Упражнение 2.25

Какие из следующих комбинаций троек Хоара корректны?

а. `{{ 0 <= x }} x := x + 1 {{ -2 <= x }} y := 0 {{ -10 <= x }}`

б. `{{ 0 <= x }} x := x + 1 {{ true }} x := x + 1 {{ 2 <= x }}`

в. `{{ 0 <= x }} x := x + 1; x := x + 1 {{ 2 <= x }}`

г. `{{ 0 <= x }} x := 3 * x; x := x + 1 {{ 3 <= x }}`

д. `{{ x < 2 }} y := x + 5; x := 2 * x {{ x < y }}`

Упражнение 2.26

Потренируйтесь в работе в обратном направлении на нескольких операторах присваивания. Вычислите *WP* относительно `x + y <= 100` для следующих программ:

а) `x := x + 1; y := x + y`

б) `y := x + y; x := x + 1`

в) `x, y := x + 1, x + y`

Упражнение 2.27

Вычислите *SP* относительно `x + y <= 100` для следующих программ:

а) `x := x + 1; y := x + y`

б) `y := x + y; x := x + 1`

в) `x, y := x + 1, x + y`

Упражнение 2.28

Вычислите самое сильное постусловие и самое слабое предусловие для следующих инструкций относительно предиката `x + y < 100`. Упростите предикаты.

а. `x := 32; y := 40`

б. `x := x + 2; y := y - 3 * x`

Упражнение 2.29

Вычислите.

а. *WP*⟦`var x; x := 10, x <= 100`⟧

б. *SP*⟦`var x; x := 10, x <= 100`⟧

Упражнение 2.30

Вычислите самое сильное постусловие и самое слабое предусловие для следующих инструкций относительно предиката `x < 10`:

```
if x % 2 == 0 { y := y + 3; } else { y := 4; }
if y < 10 { y := x + y; } else { x := 8; }
```

2.7. Вызовы методов и постусловия

Методы непрозрачны. Это означает, что мы должны рассуждать о них с точки зрения их спецификации, а не реализации. Снова рассмотрим наш метод `Triple` из раздела 1.4:

```
method Triple(x: int) returns (y: int)
  ensures y == 3 * x
```

Ожидается, что мы сможем доказать тройки Хоара, такие как

```
{{ true }} t := Triple(u + 3) {{ t == 3 * (u + 3) }}
```

(Это *то*, чего вы и ожидали, не так ли?) Для уточнения нам нужно рассмотреть несколько деталей. Начнем с параметров.

2.7.0. Параметры

Для иллюстрации рассмотрим вызов:

```
t := Triple(u + 3)
```

В спецификации используются формальные параметры `x` и `y`, а в точке вызова – фактические параметры `u + 3` и `t`. Каким-то образом мы должны связать их. Чтобы избежать возможных конфликтов имен, включая любые совпадения имен формальных и фактических параметров, присвоим формальным параметрам *новые* имена, т. е. переименуем формальные параметры и дадим им имена, которые еще не используются в нашей проверке. Имена считаются новыми, если они уникальны для этой точки вызова.

На этом этапе я переименую `x` в `x'` и `y` в `y'`. С этими новыми именами метод и его спецификация выглядят так:

```
method Triple(x': int) returns (y': int)
  ensures y' == 3 * x'
```

Смысл вызова теперь можно объяснить как последовательную композицию: сначала формальным входным параметрам с только что выбранными новыми именами присваиваются фактические входные параметры, а затем выходные параметры с только что выбранными новыми именами присваиваются фактическим выходным параметрам, где – и это немаловажно – мы можем предположить связь между переменными с только что выбранными именами, вытекающую из постусловий. В нашем примере этими двумя присваиваниями являются `x' := u + 3` и `t := y'`, а отношение, которое мы предполагаем, – это `y' == 3 * x'`.

2.7.1. Предположения

Для большей точности было бы неплохо иметь программную инструкцию, вводящую предположение. С этой целью я определю инструкцию **`assume E`** для логического выражения **`E`**. Для любого предиката `P` предшествующего состояния и предиката `Q` последующего состояния семантика **`assume`** задается следующими двумя формулами:

```
SP⟦assume E, P⟧  =  P && E
WP⟦assume E, Q⟧  =  E ==> Q
```

Первая формула гласит: если предполагается (assume), что условие `E` выполняется для состояния, удовлетворяющего предикату `P`, то впоследствии можно предположить, что и `P`, и `E` выполняются. Вторая формула гласит, что для доказательства справедливости `Q` после **`assume E`** перед инструкцией достаточно доказать, что `Q` выполняется при предположении `E`.

Инструкция **`assume`** – фиктивная. То есть она не обрабатывается компилятором. Она просто вводит предположение, очень похожее на исполнение желания. Но если воспринимать фиктивную инструкцию как «о, в этой точке я, по-видимому, могу предположить, что условие `E` выполняется», то такая инструкция `assume` оказывается весьма удобной в роли примитива для определения семантики не фиктивных инструкций, таких как инструкция вызова.

2.7.2. Семантика вызова метода с постусловием

Используя переименование и предположение, можно определить семантику нашего вызова. Выше мы уже переименовали формальные параметры метода `Triple`. А семантику вызова задает следующий фрагмент программы:

```
var x', y';
x' := u + 3;
assume y' == 3 * x';
t := y'
```

Говоря другими словами, мы рассуждаем о вызове

```
t := Triple(u + 3)
```

точно так же, как об этих четырех инструкциях.

Соответственно, в терминах слабейших предусловий мы имеем для любого предиката `Q`:

```
    WP⟦t := Triple(u + 3), Q⟧
=       { определение вызова в терминах четырех инструкций }
    WP⟦var x',y'; x' := u + 3; assume y' == 3 * x'; t := y', Q⟧
= { определение WP для ; }
    WP⟦var x',y', WP⟦x' := u + 3,
      WP⟦assume y' == 3 * x', WP⟦t := y', Q⟧ ⟧ ⟧ ⟧
=       { определение WP для := и assume и var }
    forall x', y' :: (y' == 3 * x' ==> Q[t := y'])[x' := u + 3]
=       { подстановка для x' }
    forall x', y' :: y' == 3 * (u + 3) ==> Q[t := y']
=       { x' не используется }
    forall y' :: y' == 3 * (u + 3) ==> Q[t := y']
=       { логика }
    Q[t := y'][y' := 3 * (u + 3)]
=       { подстановка для y' }
    Q[t := 3 * (u + 3)]
```

Давайте проверим. Для примера возьмем в качестве `Q` условие `t == 54`. То есть нам нужно, чтобы вызов `t := Triple(u + 3)` соответствовал условию `t == 54`. Какое условие должно выполняться перед вызовом?

```
    WP⟦t := Triple(u + 3), t == 54⟧
=       { WP вызова, см. выше }
    (t == 54)[t := 3 * (u + 3)]
=       { подстановки }
3 * (u + 3) == 54
=       { арифметика }
u == 15
```

Итак, если нам нужно, чтобы инструкция `t := Triple(u + 3)` соответствовала условию `t == 54`, то перед вызовом должно выполняться условие `u == 15`. Это соответствует нашему пониманию вызова и его спецификации, потому что если `u == 15`, тогда `u + 3` будет равно `18` и `Triple` вернет `54`.

В более общем смысле для метода

```
method M(x: X) returns (y: Y)
  ensures R
```

семантика его вызова определяется так:

```
WP⟦t := M(E), Q⟧ =
```

```
  forall x', y' :: (R[x,y := x',y'] ==> Q[t := y'])[x' := E]
```

где `x'` и `y'` – новые имена, а `Q` – любой предикат, описывающий состояние после вызова. Обратите внимание, что я включил переименование прямо в формулу, не выполняя его как отдельный шаг. Поскольку `x'` – новое имя, оно не встречается в `Q`, поэтому мы можем упростить формулу, распределив операции подстановки:

```
WP⟦t := M(E), Q⟧ =
  forall y' :: R[x,y := E,y'⟧ ==> Q[t := y']
```

Написанное мною применимо к любому количеству входных и выходных параметров. То есть `x`, `y` и `t` можно интерпретировать как списки переменных, а `E` – список выражений (при условии, что `x` и `E` имеют одинаковую длину, `y` и `t` имеют одинаковую длину и в списке `t` нет дубликатов).

Упражнение 2.31

Вычислите самое слабое предусловие для вызова `t := Abs(7 * u)` относительно постусловия `u < t`, где `Abs` определяется следующим образом:

```
method Abs(x: int) returns (y: int)
  ensures 0 <= y && (x == y || x == -y)
```

Упростите ответ.

Упражнение 2.32

Вычислите самое слабое предусловие для вызова `t := Max(2 * u, u + 7)` относительно постусловия `t % 2 == 0`, где `Max` определяется следующим образом:

```
method Max(x: int, y: int) returns (m: int)
  ensures m == x || m == y
  ensures x <= m && y <= m
```

Упростите ответ.

Поскольку у нас есть способ задать семантику вызова в терминах четырех более простых инструкций, далее следует определение семантики вызова с сильнейшим постусловием. Для любого предиката `P` в состоянии, предшествующем нашему примеру вызова `Triple`, мы имеем:

```
  SP⟦t := Triple(u + 3), P⟧
=     { определение вызова в терминах четырех инструкций }
  SP⟦var x',y'; x' := u + 3; assume y' == 3 * x'; t := y', P⟧
=     { определение SP для ; }
  SP⟦var x',y', SP⟦t := y',
    SP⟦assume y' == 3 * x', SP⟦x' := u + 3, P⟧ ⟧ ⟧ ⟧
```

```
=     { определение SP для :=, упрощено, потому что x' – новое имя }
  SP⟦var x',y', SP⟦t := y',
    SP⟦assume y' == 3 * x', P && x' == u + 3⟧ ⟧ ⟧
=     { определение SP для assume }
  SP⟦var x',y', SP⟦t := y',
    P && x' == u + 3 && y' == 3 * x'⟧ ⟧
=     { определение SP для :=, где t₀ – новое имя }
  SP⟦var x',y',
    exists t₀ :: (P && x' == u + 3 && y' == 3 * x')[t := t₀⟧ && t == y'⟧
=     { упрощение }
  SP⟦var x',y',
    exists t₀ :: P[t := t₀] && x' == u + 3 && y' == 3 * x' && t == y'⟧
=     { определение SP для var }
  exists x', y' :: exists t₀ ::
    P[t := t₀⟧ && x' == u + 3 && y' == 3 * x' && t == y'
=     { логическое упрощение для x' }
  exists y' :: exists t₀ ::
    (P[t := t₀] && y' == 3 * x' && t == y')[x' := u + 3]
=     { применение подстановки для x' }
  exists y' :: exists t₀ ::
    P[t := t₀⟧ && y' == 3 * (u + 3) && t == y'
=     { логическое упрощение для y' }
  (exists t₀ :: P[t := t₀] && y' == 3 * (u + 3))[y' := t]
=     { применение подстановки для y' }
  exists t₀ :: P[t := t₀] && t == 3 * (u + 3)
```

В общем случае, когда `M` – описанный выше метод с постусловием `R`, мы имеем:

```
SP⟦t := M(E), P⟧ =
  exists t₀ :: P[t := t₀] && R[x := E[t := t₀⟧[y := t]
```

где t_0 – новое имя. (Не знаю, как вам, но мне формулировка самого слабого предусловия кажется проще формулировки самого сильного постусловия.)

2.8. Инструкции assert

В описании вызовов методов выше я показал только постусловия. Но, прежде чем перейти к предусловиям, я хочу определить семантику утверждений – инструкций **`assert`**.

Инструкция **`assert`** `E` является пустой операцией, если условие `E` выполняется; иначе инструкция вызывает сбой программы. Таким образом, чтобы доказать, что условие `Q` выполняется после некоторой инструкции, нужно доказать, что перед ней выполняются оба условия, `E` и `Q`: `E`, чтобы предотвратить сбой **`assert`**, и `Q`, чтобы подтвердить, что **`assert`** не меняет состояния. Это отражено в следующей формуле для *WP*:

WP⟦**assert** E, Q⟧ = E && Q

Мы всегда заинтересованы в том, чтобы наши программы работали без сбоев, поэтому эффект инструкции **assert** заключается в добавлении проверяемого условия.

Упражнение 2.33

Вычислите самое слабое предусловие следующих инструкций относительно постусловия `x < 100`. Упростите каждый ответ.

а. **assert** `y == 25`

б. **assert** `0 <= x`

в. **assert** `x < 200`

г. **assert** `x <= 100`

д. **assert** `0 <= x < 100`

2.8.0. Реальная разница между SP и WP

Вот формула для *SP* в случае **assert**:

SP⟦**assert** E, P⟧ = P && E

В нем говорится, что максимум, в чем вы можете быть уверены после безаварийного выполнения инструкции, – что оба условия, P и E, выполняются.

До сих пор выбор между *SP* и *WP* казался довольно произвольным или, может быть, лишь делом вкуса. Если говорить об инструкциях присваивания, то *SP* сложнее, чем *WP*, но есть ли более глубокая разница между *SP* и *WP*? Да, есть, и инструкция **assert** выявляет эту разницу.

Если вы решите определить формальный смысл инструкций, используя только *SP*, то у вас возникнут два вопроса в формуле для *SP* в случае инструкции **assert**, представленном выше.

1. При каких обстоятельствах **assert** E дает сбой? Например, всегда ли инструкция приводит к сбою, когда условие E не выполняется? И еще, можно ли быть уверенными, что инструкция *не* потерпит сбой, если условие E выполняется?
2. Предполагается, что **assert** дает возможность ввести проверяемое условие, но как *SP* отражает тот факт, что E является проверяемым условием?

Ответ на вопрос 1: *SP ничего не* говорит о том, когда инструкция может потерпеть сбой. Оно говорит лишь о том, что если инструкция выполнилась успешно, то мы знаем, какому условию будет соответствовать получившееся состояние.

Ответ на вопрос 2: никак. То есть *SP* ничего не знает о проверяемых условиях. Вернувшись к разделу 2.7.1, вы можете встревожиться, обнаружив, что для **assert** E и **assume** E формула *SP* выглядит одинаково. В том разделе я определил **assume** E как фиктивную инструкцию, которая вводит предположение. Это предположение не несет накладных расходов, как желание или мечта, и не требует выполнения никаких проверяемых условий. Идея **assert** E, напротив, состоит в том, чтобы ввести проверяемое условие: мы не просто предполагаем выполнение условия E, но должны доказать, что оно действительно выполняется. Поэтому *SP*⟦**assert** E, P⟧ совпадает с *SP*⟦**assume** E, P⟧, и оба говорят, что в случае успешного завершения инструкции последующее состояние удовлетворяет условию P && E.

Упражнение 2.34

Предположим, мы не использовали *WP* и определили семантику инструкции только в терминах *SP*. Тогда, основываясь только на формуле *SP*, можете ли вы сказать, что инструкция **assert** способна снизить ваш шанс на успех?

Итак, *SP* сообщает, что мы получим в случае успешного выполнения инструкции, но не говорит, при каких обстоятельствах инструкция выполнится успешно. *WP*, напротив, сообщает предварительное состояние, при котором инструкция гарантированно выполнится без сбоев и завершится в желаемом состоянии. Инструкции **assert** и **assume** подчеркивают это различие:

```
WP⟦assert E, Q⟧ = E && Q
WP⟦assume E, Q⟧ = E ==> Q
SP⟦assert E, P⟧ = P && E
SP⟦assume E, P⟧ = P && E
```

Эти формулы *WP* говорят, что если желательно, чтобы после инструкции выполнялось условие Q, то для **assert** E мы должны доказать, что выполняется E, *и* доказать, что выполняется Q, тогда как для **assume** E ничего доказывать не нужно, но при доказательстве Q мы можем предполагать, что E выполняется. Формулы *SP* просто говорят, что если инструкции завершаются успешно, то выполняется условие P && E.

2.8.1. Неофициальное чтение

Имена инструкций **assert** (утверждение) и **assume** (предположение) не случайны. Они были выбраны так, чтобы напоминать их назначение. Вы можете забыть математические определения этих инструкций, если (пока нет причин увлекаться формальными деталями) вы понимаете следующие высказывания.

Инструкция **assert** E говорит: «В этой точке программы мы утверждаем, что E выполняется; т. е. мы ожидаем, что E выполняется, и собираемся это доказать». Или другой вариант: «Мы ожидаем, что всякий раз, когда поток управления достигает этой точки программы, выполняется условие E, и мы собираемся доказать, что это действительно так для всех возможных путей выполнения».

Инструкция **assume** E гласит: «В этой точке программы мы не знаем, выполняется ли условие E, но, чтобы доказать корректность нашей программы, будем предполагать, что E действительно выполняется». Или другой вариант: «Когда поток управления программы достигает этой точки, E может выполняться или не выполняться, но в нашем доказательстве мы будем игнорировать любые случаи выполнения, когда E не выполняется».

2.8.2. Использование assume для рассуждения о вызовах

Мог ли я использовать **assume** при определении значения вызовов в разделе 2.7.2? Действительно ли мы доказываем корректность нашей программы, допуская возможность игнорирования результатов некоторых путей выполнения, используя **assume** в доказательстве? Да, в этом случае мы по-прежнему доказываем корректность нашей программы, потому что собираемся сделать так, чтобы ни один путь выполнения не был проигнорирован.

Для вызова метода M я использовал инструкцию **assume** в точке вызова, чтобы допустить корректность постусловия M. Рано или поздно в рассуждениях о программе мы докажем корректность реализации M. В этом доказательстве мы убедимся, что реализация метода соответствует постусловию. Таким образом, при возврате из метода всегда будет выполняться постусловие вызванного метода и ни один путь выполнения не остается без внимания.

Это оправдывает использование **assume** в определениях вызовов.

2.9. Слабейшие свободные предусловия

Моя цель в этой книге – показать вам, как рассуждать о программах, чтобы вы могли четко объяснить (себе, коллеге-программисту или даже машине), почему программу можно считать корректной. В базовой семантике есть прекрасный аспект, который не является строго необходимым для доказательства корректности ваших программ. В этом разделе я познакомлю вас с ним. Он также объясняет некоторое таинство разницы между *WP* и *SP*, о которой я говорил в разделе 2.8.0. Если вас не интересуют подобные обсуждения базовой семантики, то можете смело пропустить этот раздел и сразу перейти к разделу 2.10.

2.9.0. Превращение обработки проверяемых условий в отдельную задачу

В разделе 2.8.0 я указал, что *SP* не предполагает доказательство отсутствия сбоев, в отличие от *WP*. Также с самого начала я уточнил, что *SP* действует в прямом направлении, тогда как WP – в обратном. Означает ли, что в рассуждениях в прямом направлении мы не можем говорить о доказательстве выполнения без сбоев?

Не совсем. Мы могли бы определить вторую семантическую функцию в прямом направлении, которая будет отслеживать все проверяемые условия (см. упражнение 2.35). Но *WP*, двигаясь в обратном направлении, уже делает это. То есть по мере продвижения *WP* в обратном направлении оно несет в себе любые проверяемые условия, с которыми сталкивается. Например,

```
  WP⟦x := x + 1; assert x <= 10, true⟧
=     { определение WP для ; }
  WP⟦x := x + 1, WP⟦assert x <= 10, true⟧ ⟧
=     { определение WP для assert }
  WP⟦x := x + 1, x <= 10⟧
=     { определение WP для := }
  x + 1 <= 10
```

Здесь мы начали с вычисления самого слабого предусловия и в итоге пришли к утверждению, что условие `x + 1 <= 10` должно выполняться в предшествующем состоянии. Таким образом, предикат `x + 1 <= 10` является условием корректности.

Итак, если *WP* каким-то образом выполняет двойную функцию, есть ли способ разделить *WP* на более простые семантические функции? Да, есть. Функция *WP*, которую мы видели, иногда называют *слабейшим консервативным предусловием*. Мы можем определить его в терминах *слабейшего свободного предусловия*, которое игнорирует неудачное выполнение, и семантической функции, накапливающей проверяемые условия для предотвращения неудач. Давайте определим их.

Для любой инструкции `S` и любого предиката `Q` в последующем состоянии, получающемся после выполнения `S`, самое слабое свободное предусловие `S` по отношению к `Q`, записываемое как *WLP*⟦`S, Q`⟧, обозначает предшествующие состояния, когда каждое безаварийное выполнение `S` завершается в состоянии, удовлетворяющем `Q`. То есть если выполнение начинается в состоянии, удовлетворяющем *WLP*⟦`S, Q`⟧, то выполнение завершается либо сбоем, либо успехом в состоянии, удовлетворяющем `Q`. Обратите внимание: если `S` – это инструкция, которая всегда завершается без сбоя, то *WLP*⟦`S, Q`⟧ – это то же самое, что *WP*⟦`S, Q`⟧.

Для любой инструкции `S` нам нужна семантическая функция, сообщающая об условиях выполнения `S`, т. е. из каких состояний инструкция `S` гарантированно выполнится успешно. У нас уже есть такая функция, а

именно *WP*⟦S, **true**⟧. Вспоминая определение *WP*, мы видим, что *WP*⟦S, **true**⟧ описывает те предшествующие состояния, из которых выполнение S завершается успехом в состоянии, удовлетворяющем предикату **true**. Поскольку каждое состояние удовлетворяет условию, *WP*⟦S, **true**⟧ – это то, что мы ищем.

Объединив эти две части, получаем следующую важную функцию для любых S и Q:

```
WP⟦S, Q⟧  =  WLP⟦S, Q⟧ && WP⟦S, true⟧
```

2.9.1. Связь между WLP и SP

В разделе 2.8.0 я утверждал, что *WP* и *SP* не являются прямой и обратной формулировками одной и той же функции. Но это утверждение неверно для *WLP* и *SP*. Оказывается, что для любых S, P и Q справедливо следующее:

```
SP⟦S, P⟧ ==> Q  тогда и только тогда, когда  P ==> WLP⟦S, Q⟧
```

Я записал *SP* и *WLP* как принимающие два аргумента: инструкцию программы и предикат. Для фиксированной S мы можем думать о *SP*⟦S, _⟧ и *WLP*⟦S, _⟧ как о функциях предиката. Всякий раз, когда такие функции удовлетворяют приведенному выше отношению «тогда и только тогда, когда», говорят, что они образуют *соединение Галуа* [19]. Функция *SP*⟦S, _⟧ называется *нижним сопряжением*, а функция *WLP*⟦S, _⟧ – *верхним сопряжением*.

Не все функции могут образовывать соединение Галуа. Если функция имеет соответствующее верхнее сопряжение, то это верхнее сопряжение уникально, но функция может вообще не иметь верхнего сопряжения. Аналогично если функция имеет соответствующее нижнее сопряжение, то это нижнее сопряжение уникально, но функция может вообще не иметь нижнего сопряжения.

Функции соединения Галуа обладают множеством интересных свойств. Одно из них касается дистрибутивности функций по дизъюнкции и конъюнкции. Каждое нижнее сопряжение *универсально дизъюнктивно*, а каждое верхнее сопряжение *универсально конъюнктивно*. Для наших функций *SP* и *WLP* это означает:

```
SP⟦S, P0 || P1 || P2 || ...⟧ =
  SP⟦S, P0⟧ || SP⟦S, P1⟧ || SP⟦S, P2⟧ || ...
```

и

```
WLP⟦S, Q0 && Q1 && Q2 && ...⟧ =
  WLP⟦S, Q0⟧ && WLP⟦S, Q1⟧ && WLP⟦S, Q2⟧ && ...
```

где P0, P1, P2, ... и Q0, Q1, Q2, ... обозначают любые наборы предикатов. Я действительно имею в виду *любые*. Коллекция может быть бесконечной или конечной. Она может быть даже пустой коллекцией, дизъюнкция которой

ложна, а конъюнкция истинна. Итак, для любой инструкции S справедливо следующее:

```
SP⟦S, false⟧ = false
WLP⟦S, true⟧ = true
```

2.9.2. Самое сильное консервативное постусловие? Такого не существует!

Существует ли семантическая функция, которую можно объединить с *WP*, чтобы сформировать соединение Галуа? Если да, то *WP*⟦S, _⟧ будет верхним сопряжением и удовлетворять условию:

```
WP⟦S, true⟧ = true
```

Но это не относится к таким инструкциям, как **assert false**. Таким образом, мы приходим к выводу, что не существует версии *SP*, которая могла бы работать в паре с *WP*.

Упражнение 2.35

Для любой инструкции присваивания, инструкции **assert**, последовательной композиции или условной инструкции S определите семантическую функцию:

```
NOCRASH⟦S, P⟧
```

дающую условие, которое необходимо проверить, чтобы гарантировать отсутствие выполнения S из состояния, удовлетворяющего условию P, при котором происходит сбой (т. е. когда выполнение S из P не вызывает сбоев). Другими словами, *NOCRASH*⟦S, P⟧ должна давать предикат, эквивалентный `P ==> WP⟦S, true⟧`. Но вместо определения семантической функции задом наперед, как это делается для *WP*, определите *NOCRASH*⟦S, P⟧ в прямом направлении.

2.10. Вызовы методов с предусловиями

Вернемся к вызовам методов. В разделе 2.7 мы обсудили семантику вызова метода с параметрами и постусловием. Используя инструкцию assert из раздела 2.8, мы теперь можем расширить обработку вызова метода, чтобы добавить обработку предусловия, включающего проверяемые условия для вызывающей стороны.

В качестве основы для дальнейших рассуждений возьмем следующую спецификацию метода:

```
method M(x: X) returns (y: Y)
  requires P
  ensures R
```

Чтобы определить семантику вызова

```
t := M(E)
```

сначала найдем новые имена `x'` и `y'` для параметров, после чего семантика вызова приобретает вид, как показано ниже:

```
var x', y';
x' := E;
assert P[x := x'];
assume R[x,y := x',y'];
t := y'
```

Другими словами, мы присваиваем фактические входные параметры `E` только что созданным формальным входным параметрам `x'`, затем утверждаем предварительное условие вызова (т. е. проверяем выполнение предусловия в точке вызова), а затем, предполагая, что только что созданные формальные выходные параметры удовлетворяют постусловию метода, присваиваем их фактическим выходным параметрам.

Отсюда можно вычислить самое слабое предусловие вызова:

```
  WP[t := M(E), Q]
=     { семантика вызова }
  WP[var x',y'; x' := E; assert P[x := x'];
      assume R[x,y := x',y']; t := y', Q]
=     { определение WP для ; и := }
  WP[var x',y'; x' := E; assert P[x := x'];
      assume R[x,y := x',y']; Q[t := y'] ]
=     { определение WP для ; и assume }
  WP[var x',y'; x' := E; assert P[x := x'],
   R[x,y := x',y'] ==> Q[t := y'] ]
=     { определение WP для ; и assert }
  WP[var x',y'; x' := E,
   P[x := x'] && (R[x,y := x',y'] ==> Q[t := y']) ]
=     { определение WP для ; и := }
  WP[var x',y', (P[x := x'] && (R[x,y := x',y'] ==> Q[t := y']))[x' := E] ]
=     { определение WP для var }
  forall x', y' :: (P[x := x'] && (R[x,y := x',y'] ==> Q[t := y']))[x' := E]
=     { применение подстановки для x' }
  forall x', y' :: P[x := E] && (R[x,y := E,y'] ==> Q[t := y'])
=     { x' не используется }
  forall y' :: P[x := E] && (R[x,y := E,y'] ==> Q[t := y'])
```

Как мы уже говорили в разделе 2.7.2, переменная `y'` определяется количественно в последней строке, потому что `y'` обозначает любые значения формальных выходных параметров, удовлетворяющие постусловию. Итак, наша окончательная формула для *WP* в случае вызова метода, учитывающая как предусловия метода, так и его постусловие:

```
WP⟦t := M(E), Q⟧ =
  P[x := E] && forall y' :: R[x,y := E,y'] ==> Q[t := y']
```

Упражнение 2.36

Пусть `x'` – новое имя, докажите, что

```
x' := E; assert P[x := x']
```

это то же самое, что и

```
assert P[x := E]
```

показав, что слабейшие предусловия этих двух утверждений одинаковы.

2.11. Вызовы функций

Мы рассуждаем о вызовах методов с точки зрения их спецификаций, поскольку методы непрозрачны. Функции, напротив, прозрачны, поэтому мы рассуждаем о них, просто раскрывая определение функции.

Например, рассмотрим функцию вычисления абсолютного значения числа:

```
function Abs(x: int): int {
  if x < 0 then -x else x
}
```

Поскольку `Abs` – это функция (а не метод), ее тело является выражением (см. раздел 1.5). Здесь я использую *выражение* **`if-then-else`**, которое является формой выражения *инструкции* **`if`**. Ветви **`then`** и **`else`** этого выражения тоже являются выражениями (тогда как ветви инструкции **`if`** являются инструкциями). Используя функцию `Abs`, докажем, что в любой точке программы и для любого выражения `E` всегда выполняется условие `0 <= Abs(E)`. Это можно сделать, написав

```
assert 0 <= Abs(E)
```

в выбранной точке программы и доказать, что эта инструкция не дает сбоя. Чтобы доказать, что инструкция не вызывает сбоев и соответствует постусловию `Q`, мы действуем следующим образом:

```
  WP⟦assert 0 <= Abs(E), Q⟧
=     { WP для assert }
  0 <= Abs(E) && Q
=     { определение Abs }
  0 <= (if E < 0 then -E else E) && Q
=     { распределение <= через if-then-else }
```

```
  (if E < 0 then 0 <= -E else 0 <= E) && Q
=     { арифметика }
  true && Q
=
  Q
```

Упражнение 2.37

Вычислите `WP⟦m := Max(x, y), m <= 100⟧`, взяв функцию `Max` из упражнения 2.32. Упростите свой ответ.

Упражнение 2.38

Вычислите *WP*`⟦y := Plus3(x); x := Plus3(y), x < 10⟧`, учитывая

```
function Plus3(x: int): int {
  x + 3
}
```

Упростите свой ответ.

2.12. Частичные выражения

Выражение не всегда может быть определено. Например, деление `c / d` имеет смысл, только если `d` не равно `0`. Мы говорим, что такое выражение является *частичным*. Когда выражение определено в контексте, где оно используется, мы описываем его с помощью таких слов, как *правильно построенное* или *корректно определенное*, а если оно всегда определено, мы говорим, что оно является *тотальным*.

Когда мы рассуждаем об инструкциях, возникает проверяемое условие, что все выражения в инструкциях корректно определены. Формально такие проверяемые условия выражаются с помощью *WP*. Это выглядит примерно так:

```
WP⟦x := E, Q⟧  =  DEFINED⟦E⟧ && Q[x := E]
```

где *DEFINED*`[E]` обозначает выражение, которое (само по себе является тотальным и) имеет значение **true** именно тогда, когда `E` корректно определено. Например,

```
WP⟦x := c / d, x == 100⟧
```

есть

```
d != 0 && c / d == 100
```

Я не собираюсь давать определение *DEFINED* для всех выражений, используемых в этой книге, и не собираюсь переделывать формулы для *WP*,

которые мы видели до сих пор, включая *DEFINED* в каждое подвыражение. Вместо этого я буду просто говорить о существовании некоторого проверяемого условия. Например, я буду говорить, что инструкция `x := c / d` порождает проверяемые условия `d != 0`.

Вместо включения *DEFINED* в определение *WP* мы можем думать о каждой инструкции как начинающейся с неявного утверждения **`assert`**, содержащего условия корректности инструкции. Если всегда применять *WP* к таким неявным утверждениям, то вы будете получать желаемые предикаты, используя *WP*, как я определил в этой главе.

Позвольте мне показать вам, как это будет происходить. Когда я говорю «выражение `c / d` порождает проверяемое условие, что `d` не равно `0`», я имею в виду, что неявное утверждение, предшествующее `x := c / d`, имеет вид:

```
assert d != 0
```

А когда я указываю, что фрагмент программы, такой как

```
if c / d < u / v {
  x := a[i];
}
```

порождает проверяемые условия, что `d` и `v` не равны `0` и индекс `i` является корректным индексом в массиве `a`, то на самом деле я подразумеваю, что программа имеет следующие неявные утверждения:

```
assert d != 0 && v != 0;
if c / d < u / v {
  assert 0 <= i < a.Length; // утверждает, что i находится в диапазоне индексов
  x := a[i];
}
```

По сути, это преобразование единообразно превращает все проверяемые условия корректности определений в утверждения. Таким образом, преобразованная программа потерпит сбой, если исходная программа выполнит деление на ноль, использует индекс, выходящий за границы массива, или выполнит любое другое действие, дающее неопределенный результат.

Не все частичные выражения являются встроенными операциями. Пользовательские функции могут иметь предусловия. Например, рассмотрим функцию `MinusOne`:

```
function MinusOne(x: int): int
  requires 0 < x
```

Для вызова `MinusOne` необходимо доказать, что аргумент является положительным. Таким образом, инструкция `z := MinusOne(y)` порождает следующее неявное утверждение:

```
assert 0 < y
```

Упражнение 2.39

Как выглядит формула для *DEFINED* в случае следующих выражений?

а. `x / (y + z)`

б. `a / b < c / d`

в. `a[2 * i]`

г. `MinusOne(MinusOne(y))`

Операторы `&&`, `||` и `==>` называются в языках программирования *операторами, вычисляемыми по короткой схеме*, потому что их четкая определенность зависит от значения левого аргумента. У нас есть

```
DEFINED⟦E && F⟧ = DEFINED⟦E⟧ && (E ==> DEFINED⟦F⟧)
DEFINED⟦E || FK = DEFINED⟦E⟧ && (E || DEFINED⟦F⟧)
DEFINED⟦E ==> F⟧ = DEFINED⟦E⟧ && (E ==> DEFINED⟦F⟧)
```

Как показывают эти определения, корректность каждого из трех выражений зависит от корректности `F`, только если `E` (корректно определено и) оценивается как **`true`**, **`false`** и **`true`** соответственно. Подобным же образом корректность выражения **`if-then-else`** зависит от защитного условия:

```
DEFINED⟦if B then E else F⟧ =
  DEFINED⟦B⟧ && if B then DEFINED⟦E⟧ else DEFINED⟦F⟧
```

Упражнение 2.40

Как выглядит формула для *DEFINED* в следующих выражениях?

а. `p && c / d == 100`

б. `a / b < 10 || c / d < 100`

в. **`if`** `a == b` **`then`** `c / d` **`else`** `10`

г. `MinusOne(y) == 8 ==> a[y] == 20`

Упражнение 2.41

Используйте выражение **`if-then-else`**, которое эквивалентно (как по значению, так и по корректности) следующему: а) `E && F`, (b) `E || F` и (c) `E ==> F`.

Упражнение 2.42

Вернитесь к упражнению 2.35, но на этот раз определите *NOCRASH*, включив частичные выражения для выражений, встречающихся в тексте программы.

2.13. Корректность метода

Чтобы доказать, что реализация метода соответствует его спецификации, нам нужно показать, что данное предусловие подразумевает самое слабое предусловие тела метода по отношению к постусловию.

Например, чтобы доказать корректность следующего метода:

```
method M(x: X) returns (y: Y)
  requires P
  ensures Q
{
  Body
}
```

мы должны доказать:

```
P ==> WP[[Body, Q]]
```

2.14. Итоги

В этой главе я определил семантику наиболее распространенных инструкций, с которыми нам придется столкнуться. Кратко напомню, что логика Флойда сводится к написанию формул, характеризующих состояние программы, и проверке того, что программа «преобразует» одни предикаты в другие. Чтобы обозначить, что инструкция программы принимает состояние, соответствующее заданному предикату предусловия, и преобразует его в состояние, соответствующее заданному предикату постусловия, мы часто используем тройки Хоара. Самое слабое предусловие инструкции дает систематический способ вычисления состояния, соответствующего предикату предусловия на основе желаемого предиката постусловия. Слабейшие предусловия позволяют записать канонические тройки Хоара, вычисляя первый компонент тройки Хоара как *WP* двух других компонентов. При использовании вычисления *WP* результирующий предикат предшествующего состояния включает условия корректности программы.

Многие из наиболее сложных дискуссий в этой главе посвящены использованию сильнейших постусловий. Я включил эти дискуссии, потому что самые сильные постусловия вычисляются в направлении выполнения программы, что на первый взгляд делает их более очевидными. Кроме того, с моей стороны было бы безответственно опустить сильнейшие постусловия в главе, посвященной семантике программ. Однако, как мы видели, самые слабые предусловия не только проще вычислить, но и по мере продвижения по ним также накапливаются проверяемые условия. Вы не сможете доказать, что ваша программа действительно обеспечивает все необходимое, если эти проверяемые условия не будут выполнены. Тем не

менее всегда желательно явно проверять условия корректности, поскольку это позволяет возложить ответственность за использование различных инструкций программы, например убедиться, что наши программы не выполняют деление на ноль или не вызывают метод в состоянии, не соответствующем объявленному предусловию.

Разобравшись с семантикой программ в этой главе, я советую вам еще раз прочитать неформальные аргументы в главе 1. Вы почувствуете, что освоили материал, когда сможете подкрепить неформальные аргументы из главы 1 формальными аргументами, обсуждавшимися здесь, в уме и на бумаге. Тогда вы также сможете оценить работу автоматизированных верификаторов программ, вычисляющих *WP* или *SP + NOCRASH* за вас.

Примечания

Практика применения логики Флойда ориентирована на язык блок-схем, что позволяет применять ее к программам с любым типом потока управления [50]. Представление Хоара логики Флойда не только использует элегантные тройки Хоара, но также ограничивает внимание структурированными инструкциями программ [63]. Такое ограничение помогает эффективнее применять правила композиционного проектирования к использованию конструкций в языке программирования. Например, формулировка Хоара подчеркивает роль *инварианта* при рассуждениях о цикле. (Инварианты циклов мы увидим в части 2.)

Подробное сравнение ранних подходов к доказательству корректности программ можно найти в обзорной статье по программной логике [58].

В своей основополагающей книге «A Discipline of Programming»[1] (1976) Эдсгер В. Дейкстра (Edsger W. Dijkstra) выделил пять *условий работоспособности*, которым должны удовлетворять инструкции каждого разумного языка программирования [40]. Одно из них, получившее название «Закон исключенного чуда», гласит, что каждая инструкция `S` должна удовлетворять условию:

```
WP⟦S, false⟧ = false
```

То есть ни при каких обстоятельствах инструкция `S` не должна гарантировать ложность постусловия.

В 1980-х гг. Бэк [7], Могран [92], Моррис [96] и Нельсон [99] утверждали, что полезно разрешать использовать такие конструкции, как **`assume`**, в спецификациях и при изучении семантики программы. Обратите внимание, что *WP*⟦**`assume false, false`**⟧ истинно, что, согласно терминологии Дейкстры, делает эту инструкцию чудесной. Нельсон назвал такие инструкции *частичными командами* и использовал частичность в определении инструкций **`if`** и циклов. Морган определил общую *спецификацию инструкции*, которая может быть частичной. Морган назвал частичную форму инструкций как

[1] Эдсгер Вибе Дейкстра, «Дисциплина программирования», RUGRAM, 2013, ISBN: 978-5-458-33419-8. – *Прим. перев.*

coersions и показал, что частичные формы инструкций можно комбинировать для создания компилируемых программ [95].

Бэк и фон Райт построили прекрасную теоретико-структурную платформу программ и спецификаций, в которой инструкции **`assert`** и **`assume`** являются дуальными по отношению друг к другу [8, 9]. Они рассматривают эти инструкции как «ходы» в игре между двумя «агентами», часто называемыми «демоном» и «ангелом».

Вопреки Дейкстре Берроуз и Нельсон включили инструкции **`assume`** в полезный язык программирования для обработки строк [101]. В шутку они назвали его LIM – Language of the *Included* Miracle (язык *включенного* чуда).

Глава 3

Рекурсия и завершимость

Ваша подруга Амелия попросила одолжить ей ваш телефон. Вы соглашаетесь, при условии что Амелия пообещает вернуть его. Затем приходит Бенни и просит Амелию одолжить ваш телефон. Амелия дает телефон Бенни, но заставляет того пообещать вернуть его вам. Пока телефон находился у Бенни, Шанталь попросил у него одолжить телефон, затем Доминикас одолжил телефон у Шанталя, Эдсгер у Доминикаса, Фелистас у Эдсгера и т. д. Если так будет продолжаться вечно, то вы никогда не получите свой телефон обратно. Однако каждый из ваших друзей старается следовать обещанию вернуть телефон, заключая соглашение о его возврате с кем-то еще. Если позволить бесконечное повторение даже вполне оправданных действий, мы не достигнем ожидаемых результатов.

В этой главе я объясню, как предотвратить бесконечные повторения.

3.0. Бесконечная задача

Вот пять небольших программ, демонстрирующих важность рассмотрения вопроса о завершимости.

Первая – это метод, утверждающий, что возвращает значение, в два раза превышающее его аргумент:

```
method BadDouble(x: int) returns (d: int)
  ensures d == 2 * x
{
  var y := BadDouble(x - 1);
  d := y + 2;
}
```

Конечно, если рекурсивный вызов присвоит переменной `y` удвоенное значение выражения `x - 1`, то выражение `y + 2` вернет удвоенное значение `x`. Но каждый рекурсивный вызов повлечет еще один рекурсивный вы-

зов, поэтому, как и в случае с действиями ваших друзей, одалживающих телефон, этот метод не завершится. Это называется *бесконечной рекурсией*. Бесконечная рекурсия – нежелательное явление, потому что бессмысленно говорить, что такой метод правильно вычисляет `2 * x`.

В качестве второго примера рассмотрим метод, утверждающий, что возвращает значение своего аргумента. Что может быть проще?

```
method PartialId(x: int) returns (y: int)
  ensures y == x
{
  if x % 2 == 0 {
    y := x;
  } else {
    y := PartialId(x);
  }
}
```

Если этому методу передать четное значение в параметре `x`, то он действительно вернет `x`. Но если передать ему нечетное значение, то метод продолжит работу, вызывая метод `PartialId` (т. е. самого себя), который утверждает, что возвращает свой аргумент. Здесь мы наблюдаем другую картину: если `PartialId` завершается, то он дает правильный результат. Это называется *частичной корректностью*, когда каждый завершающийся вызов корректен, но это не означает, что все вызовы действительно завершаются.

Метод `BadDouble` выше тоже частично корректен. Фактически, поскольку `BadDouble` никогда не завершается, он будет удовлетворять условию частичной корректности, даже если вы измените выражение присваивания `d := y + 2` на `d := y + 5` или, если уж на то пошло, на `d := 1000`.

Программу называют *тотально корректной*, если она частично корректна и всегда завершается. На протяжении всей этой книги мы будем доказывать тотальную корректность. Мы уже видели эти концепции раньше. Если рассматривать незавершимость как форму сбоя, то предикат *WLP*`[S, Q]` из раздела 2.9.0 говорит, что инструкция `S` частично корректна относительно `Q`. Более того, *WP*`[S, true]` говорит, что `S` завершается без сбоя. Комбинация этих двух предикатов показывает тотальную корректность, отраженную в формуле, которую мы видели в разделе 2.9.0:

WP`[S, Q] = `*WLP*`[S, Q] && `*WP*`[S, true]`

В третьем примере мы имеем недетерминированный метод, утверждающий, что вычисляет фактический квадрат числа, используя предположение `guess`:

```
method Squarish(x: int, guess: int) returns (y: int)
  ensures x * x == y
{
```

```
    if
    case guess == x * x => // верное предположение!
      y := guess;
    case true =>
      y := Squarish(x, guess - 1);
    case true =>
      y := Squarish(x, guess + 1);
  }
```

Реализация частично корректна, и может случиться так, что выполняемый ею бесцельный поиск найдет ответ. Однако нет никаких *гарантий*, что действие прекратится. Чтобы получить корректную программу, мы должны предоставить эту гарантию, представив *доказательство* завершимости работы.

Четвертый пример пытается получить невозможное значение. И все бы ничего, если бы не необходимость доказать, что он рано или поздно завершится.

```
method Impossible(x: int) returns (y: int)
  ensures y % 2 == 0 && y == 10 * x - 3
{
  y := Impossible(x);
}
```

В качестве последнего примера рассмотрим это сомнительное определение функции:

```
function Dubious(): int {
  1 + Dubious()
}
```

Здесь мы наблюдаем не только случай бесконечной рекурсии, но и скрытую *логическую противоречивость*. Для такого определения легко доказать его ложность, как показывает рассуждение в форме утверждений в этом фрагменте программы:

```
var a := Dubious();
assert a == Dubious(); // это то, что получено
                       // предыдущим присваиванием
// следующее утверждение выполняет ту же проверку, что
// и предыдущее, но развертывает тело Dubious()
assert a == 1 + Dubious();
// объединение двух предыдущих утверждений
assert a == 1 + a;
// вычтем a из обеих сторон
assert 0 == 1;
// арифметика
assert false;
```

Невозможно доказать ложность – это было бы признаком математической *непоследовательности*. Из этого примера мы делаем вывод, что не следует допускать логически противоречивые функции, такие как `Dubious()`. Самый распространенный способ избежать логически противоречивого определения функций – доказать, что функции завершаются.

Итак, мы должны избегать бесконечной рекурсии и других форм бесконечного повторения, избегать негарантированной завершимости и логической противоречивости. Мы должны выйти за рамки частичной корректности, доказав отсутствие сбоев и завершимость, чтобы достичь тотальной корректности. Далее давайте посмотрим, как это сделать.

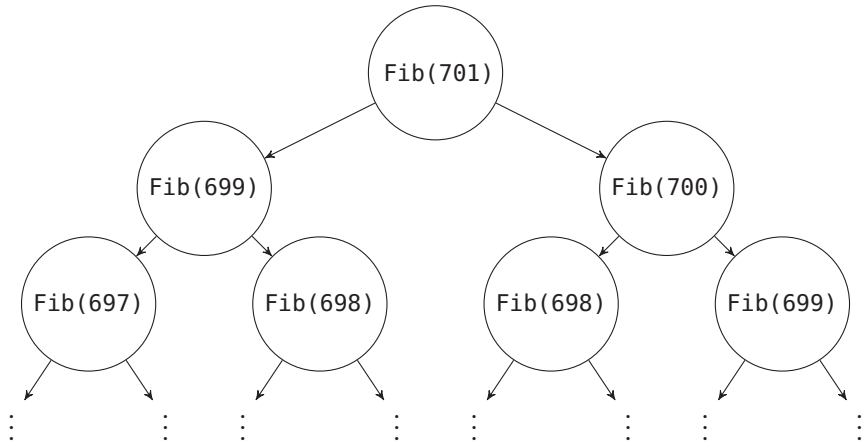
3.1. Как избежать бесконечной рекурсии

В качестве примера рассмотрим известную функцию вычисления чисел Фибоначчи:

```
function Fib(n: nat): nat {
  if n < 2 then n else Fib(n - 2) + Fib(n - 1)
}
```

В этом примере я использовал новый тип **`nat`**. Он обозначает *натуральные числа*, т. е. 0, 1, 2, ... – подмножество неотрицательных целых чисел. (Альтернативный способ определить `Fib` для натуральных чисел – объявить `n` с типом **`int`** и добавить предусловие **`requires`** `0 <= n`. По сути, это то же самое, но использование типа `nat` позволяет записать код короче.)

Функция `Fib` имеет два рекурсивных вызова: `Fib(n-2)` и `Fib(n-1)`. Если `Fib` вызывает саму себя, то как нам убедиться, что рекурсия в конце концов завершится? Представьте себе граф вызовов функций, полученных в результате вычисления `Fib`. Например, вот небольшая часть этого графа, начинающаяся с вызова `Fib(701)`:



Теперь представьте, что вы снабжаете каждый узел (т. е. каждый вызов функции) каким-то натуральным числом. Удобный способ выбрать такое натуральное число для каждого из узлов – выбрать то же число, что и аргумент функции `Fib`. Если число в каждом узле графа строго меньше числа в родительском узле, то каждый путь в графе будет конечен. Почему? Потому что не существует бесконечной убывающей последовательности натуральных чисел. (С этого момента я буду использовать общую фразу: бесконечная убывающая *последовательность*.)

Это свойство дает возможность проверить, завершается ли функция. Все, что нужно для этого, – это механизм, определяющий число для каждого вызова функции, после чего нам останется только убедиться, что число, передаваемое вызываемой функции, строго меньше числа, полученного вызывающей функцией. Это называется *оценочной функцией*, или *оценкой завершимости*.

В языке Dafny оценка завершимости функции или метода определяется с помощью ключевого слова **`decreases`**. Определение `Fib` с упоминанием этой метрики можно записать так:

```
function Fib(n: nat): nat
  decreases n
{
  if n < 2 then n else Fib(n - 2) + Fib(n - 1)
}
```

Выражение, следующее за ключевым словом **`decreases`** (в данном случае выражение `n`), будет оценено и использовано в качестве значения вызова `Fib(n)`. Таким образом, значения для применений функции, возникающих в результате двух рекурсивных вызовов, равны `n-2` и `n-1` соответственно. Dafny генерирует и проверяет выполнение условия `n-2 < n` для первого вызова и `n-1 < n` для второго вызова. Эти проверяемые условия подразумевают, что рекурсия `Fib` прекращается.

Во многих случаях ключевое слово **`decreases`** можно читать так: «С каждым рекурсивным вызовом `Fib` аргумент `n` уменьшается». Однако обратите внимание, что ключевое слово **`decreases`** просто говорит о том, какое значение придать вызову функции. Как только вызов функции получит значение в соответствии с инструкцией **`decreases`**, оцениваемой на входе в функцию или метод, это значение больше не меняется. Именно *соотношение* между значениями аргументов вызывающей и вызываемой функций должно показывать уменьшение (decrease).

Есть много других способов придать значение вызову функции `Fib`, чтобы доказать завершимость рекурсии. Например, мы могли бы использовать

```
decreases 3*n + 2
```

и в этом случае условие завершимости для первого рекурсивного вызова будет иметь вид: `3*(n-1) + 2 < 3*n + 2`.

Давайте рассмотрим другой пример. Следующая функция вычисляет сумму элементов целочисленной последовательности от индекса `lo` до индекса `hi`:

```
function SeqSum(s: seq<int>, lo: int, hi: int): int
  requires 0 <= lo <= hi <= |s|
  decreases hi - lo
{
  if lo == hi then 0 else s[lo] + SeqSum(s, lo + 1, hi)
}
```

Этот пример вводит новый для нас тип **`seq`** в Dafny – тип неизменяемых последовательностей элементов. Длина последовательности `s` определяется выражением `|s|`, а элемент с индексом `i`, где `0 <= i < |s|` извлекается выражением `s[i]`. В последующих главах мы часто будем использовать последовательности.

Оценка завершимости для функции `SeqSum` вычисляется сложнее. Данная инструкция **`decreases`** действительно дает выражение, которое уменьшается с каждым рекурсивным вызовом. В частности, рекурсивный вызов имеет проверяемое условие `hi - (lo+1) < hi - lo`.

А что, если вычислять оценочную функцию просто как `-lo`? Ведь, в конце концов, `-(lo+1) < -lo`. Проблема такой оценочной функции в том, что если не ограничиваться подмножеством неотрицательных целых чисел, то мы получим бесконечные убывающие последовательности, поэтому мы не можем утверждать, что рекурсия завершается. При рекурсивном вызове функции, подобной `SeqSum`, Dafny проверяет – неотрицательна ли оценка завершимости вызывающей стороны. Попытавшись использовать **`decreases`** `-lo`, вы получите сообщение о невозможности проверить завершимость.

Упражнение 3.0

Напишите инструкцию **`decreases`**, доказывающую завершимость следующей функции:

```
function F(x: int): int {
  if x < 10 then x else F(x - 1)
}
```

Упражнение 3.1

Напишите инструкцию **`decreases`**, доказывающую завершимость следующей функции:

```
function G(x: int): int {
  if 0 <= x then G(x - 2) else x
}
```

Упражнение 3.2

Напишите инструкцию **decreases**, доказывающую завершимость следующей функции:

```
function H(x: int): int {
  if x < -60 then x else H(x - 1)
}
```

Упражнение 3.3

Напишите инструкцию **decreases**, доказывающую завершимость следующей функции:

```
function I(x: nat, y: nat): int {
  if x == 0 || y == 0 then
    12
  else if x % 2 == y % 2 then
    I(x - 1, y)
  else
    I(x, y - 1)
}
```

Упражнение 3.4

Напишите инструкцию **decreases**, доказывающую завершимость следующей функции:

```
function J(x: nat, y: nat): int {
  if x == 0 then
    y
  else if y == 0 then
    J(x - 1, 3)
  else
    J(x, y - 1)
}
```

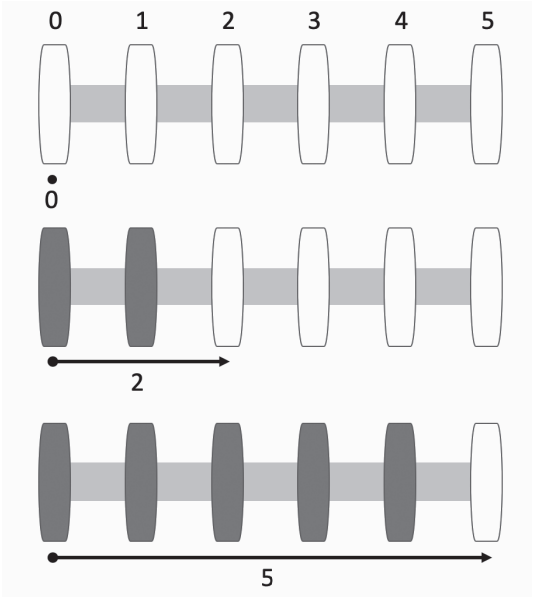
Упражнение 3.5

Напишите инструкцию **decreases**, доказывающую завершимость следующей функции:

```
function K(x: nat, y: nat, z: nat): int {
  if x < 10 || y < 5 then
    x + y
  else if z == 0 then
    K(x - 1, y, 5)
  else
    K(x, y - 1, z - 1)
}
```

Врезка 3.0

Выполняя 5, вы последовательно окажетесь в 6 точках: в любой момент вы можете сделать 0, 1, 2, 3, 4 или все 5 шагов. На следующих диаграммах показано, где вы будете находиться после 0 шагов, 2 шагов и 5 шагов соответственно:



Если через n обозначить количество выполненных шагов, то мы получаем

```
0 <= n <= 5
```

То есть n может быть числом от 0 до 5 включительно. Обратите внимание что условие <= (в отличие от <) включает верхнюю и нижнюю границы. Именно такой интервал получается, когда вы выражаете количество уже сделанных шагов, т. е. ваше местоположение в данный момент.

В более общем смысле если для преодоления всего пути требуется N шагов, то существует N + 1 различных точек.

Путь, начинающийся в lo и заканчивающийся в hi (как в интервале элементов последовательности, суммируемых функцией SeqSum), имеет hi - lo шагов. На этом пути есть hi + 1 - lo различных точек. Например, вызов SeqSum(s, 0, 5) просуммирует 5 - 0 элементов, и тело SeqSum в общей сложности будет выполнено 6 раз.

Упражнение 3.6

Напишите инструкцию **decreases**, доказывающую завершимость следующей функции:

```
function L(x: int): int {
  if x < 100 then L(x + 1) + 10 else x
}
```

Упражнение 3.7

Следующая функция вычисляет *последовательность Хофштадтера*:

```
function G(n: nat): nat {
  if n == 0 then 0 else n - G(G(n - 1))
}
```

Напишите инструкцию **decreases**, доказывающую завершимость функции G.

3.2. Отношения полной фундированности

Оценочные функции не ограничиваются натуральными числами. Пока используемое отношение «меньше чем» *полностью фундировано* (это понятие я определю чуть позже), подход, обсуждавшийся выше, позволяет доказать завершимость. Фундированное упорядочение я буду обозначать символом $\succ$, в котором меньший элемент находится справа. То есть вместо «меньше чем» я буду использовать отношение «больше чем». Это позволяет нам рассуждать о переходе от оценки завершимости вызывающей стороны (слева) к оценке завершимости вызываемой стороны (справа). Выражение `a ≻ b` можно читать как «a превышает b», «от a к b идет уменьшение значения» или «от a к b идет сокращение значения».

Бинарное отношение $\succ$ является *полностью фундированным*, если выполняются следующие три условия:

- $\succ$ является *иррефлексивным*: отношение никогда не связывает элемент сам с собой, т. е. условие $a \succ a$ не выполняется ни для какого a;
- $\succ$ является *транзитивным*: всякий раз, когда выполняются условия $a \succ b$ и $b \succ c$, то выполняется и условие $a \succ c$ для любых a, b и c;
- $\succ$ удовлетворяет *условию обрыва убывающих цепей*: отношение не имеет *бесконечно убывающих последовательностей элементов*, т. е. не существует бесконечной последовательности значений a_0, a_1, a_2, ... такой, что

$$a_0 \succ a_1 \succ a_2 \succ \cdots$$

Отношение, удовлетворяющее первым двум условиям, называется *строгим частичным порядком* (или, как его еще называют, *иррефлексивным частичным порядком*). *Частичный* порядок допускает упорядочивание не всех элементов. Например, могут существовать элементы a и b, такие что не выполняются ни $a \succ b$, ни $b \succ a$. (Если для каждой пары различных значений a и b выполняется условие $a \succ b$ или $b \succ a$, то строгий порядок называется *пол-*

ным.) Итак, фундированное упорядочение – это строгий частичный порядок, дополнительно удовлетворяющий условию обрыва убывающих цепей[1].

Обобщая упорядоченность натуральных до фундированного упорядочения $\succ$ на значениях всех типов в нашей программе, мы получаем следующий рецепт доказательства невозможности бесконечной рекурсии: присвойте каждому вызову функции такое значение (любого типа), что для каждого вызова значение a вызывающей стороны и значение b вызываемой стороны будут удовлетворять условию $a \succ b$.

Мы будем использовать одно фиксированное отношение полной фундированности $\succ$ на протяжении всей книги. Dafny предопределяет отношение $\succ$ для каждого типа, а в некоторых случаях и для значений разных типов (но в большинстве случаев значения разных типов не связаны этим частичным порядком).

Вот отношения полной фундированности для некоторых типов, встроенные в Dafny:

тип X и y	**X $\succ$ y («X сводится к y»)**
bool	`X && !y`
int	`y < X && 0 <= X`
real	`y <= X - 1.0 && 0.0 <= X`
set<T>	y – собственное подмножество X
seq<T>	y – последовательная собственная подпоследовательность X
индуктивные типы данных	y структурно включено в X

Здесь следует сделать несколько замечаний. Для логических значений упорядоченность подразумевает, что **`true`** больше **`false`**. В соответствии с отношением полной фундированности для целых чисел в Dafny неотрицательные числа больше отрицательных, но сами отрицательные числа не упорядочены. Отношения между целыми и действительными числами на самом деле одинаковы, если думать о `y < X` для целых чисел как о `y <= X - 1`. Обратите внимание, что `y < X && 0.0 <= X` не считается фундированным порядком для действительных чисел, потому что имеются бесконечные убывающие последовательности элементов. Об индуктивных типах данных мы поговорим в главе 4 (где в разделе 4.3 рассказывается о структурном включении).

Упражнение 3.8

Напишите инструкцию **`decreases`**, доказывающую завершимость следующей функции:

[1] Технически фундированное упорядочение представляет собой строгий частичный порядок, в котором каждое непустое подмножество имеет минимальный элемент. Это то же самое, что удовлетворять условию обрыва убывающих цепей, при условии принятия так называемой *аксиомы зависимого выбора*. Лично я считаю, что условие убывающей последовательности легче понять и использовать.

```
function M(x: int, b: bool): int {
  if b then x else M(x + 25, true)
}
```

Упражнение 3.9

Напишите инструкцию **decreases**, доказывающую завершимость следующей функции:

```
function N(x: int, y: int, b: bool): int {
  if x <= 0 || y <= 0 then
    x + y
  else if b then
    N(x, y + 3, !b)
  else
    N(x - 1, y, true)
}
```

Упражнение 3.10

Ответ на упражнение 3.1: **decreases** `x`.

а. Напишите условие завершимости, используя $\succ$.

б. Разверните $\succ$ для типа **int**.

3.3. Лексикографические кортежи

Иногда при доказательстве завершимости необходимо или удобно использовать *лексикографический кортеж*. Кортеж – это список значений, например пара или тройка. *Лексикографический порядок* в кортеже определяется покомпонентным сравнением, при котором более ранние компоненты считаются более значимыми. Например, для пар натуральных чисел: пара `4,12` превышает пары `4,11`, `4,2`, `3,525600` и `2,0`. Пара `4,12` превышает `4,2`, потому что их первые компоненты равны, а второй компонент `12` первой пары превышает компонент `2` второй пары. Пара `4,12` превышает `3,525600`, потому что уже первый компонент `4` первой пары превышает первый компонент `3` второй пары, и вторые их компоненты просто не имеют значения.

Мы даже можем сравнивать кортежи разной длины. Например, можно сравнить 2-элементный кортеж с 4-элементным или 16-элементный с 3-элементным. В этих случаях если один кортеж совпадает с началом другого, то считается, что более короткий кортеж превышает другой; в противном случае сравнивается столько компонентов, сколько требуется для получения ответа. Например, кортеж `4,6,0` превышает `4,6,0,25,3`, потому что `4,6,0` совпадает с началом `4,6,0,25,3`, но короче его, а кортеж `2,5` превышает `1`, потому что первый элемент `2` первого кортежа превышает первый элемент `1` второго кортежа.

Если установить некоторую верхнюю границу на длину кортежей, то обоснованность определенного выше лексикографического порядка вытекает из фундированного упорядочения компонентов.

Для использования лексикографических кортежей в доказательствах завершимости Dafny позволяет передавать каждой инструкции **`decreases`** *список* выражений. Ограничений на количество выражений в **`decreases`** нет, но в любой конкретной программе списки выражений в **`decreases`** не превышают определенной максимальной длины. Поэтому использование лексикографического порядка действительно обосновано. Для обозначения лексикографического порядка кортежей, а также порядка каждого компонента я буду использовать все тот же символ $\succ$.

Упражнение 3.11

Определите, превышает ли первый кортеж второй.

а. `2,5` и `1,7`

б. `1,7` и `7,1`

в. `5,0,8` и `4,93`

г. `4,9,3` и `4,93`

д. `4,93` и `4,9,3`

е. `3` и `2,9`

ж. **`true`**`,80` и **`false`**`,66`

з. **`true`**`,2` и `19,1`

и. `4,`**`true`**`,50` и `4,`**`false`**`,800`

к. `7.0,{3,4,9},`**`false`**`,10` и `7.0,{3,9},`**`true`**`,10`

В пункте «к» выражения `{3,4,9}` и `{3,9}` обозначают множества.

Далее я покажу еще несколько примеров использования лексикографических кортежей.

3.3.0. Оставшиеся предметы для изучения

Предположим, вам осталось изучить `n` предметов до окончания обучения в университете. В начале изучения нового предмета `c` вызывается метод

```
method RequiredStudyTime(c: nat) returns (hours: nat)
```

сообщающий, сколько часов необходимо для изучения. Объем материала, рассматриваемого в ходе изучения, зависит от преподавателя, поэтому мы не можем точно определить количество часов, возвращаемое в выходном параметре `hours`.

Следующий метод имитирует вычисление времени до окончания учебы. Он сообщает, сколько часов `h` осталось до завершения изучения предмета `n`, после которого вам также необходимо изучить предметы от `0` до `n`.

```
method Study(n: nat, h: nat)
  decreases n, h
{
  if h != 0 {
    // учимся один час и затем:
    Study(n, h - 1);
  } else if n == 0 {
    // вы только что закончили изучение предмета 0 - пришло время выпуска!
  } else {
    // узнать, сколько времени нужно на изучение следующего предмета
    var hours := RequiredStudyTime(n - 1);
    // начать изучать предмет n-1:
    Study(n - 1, hours);
  }
}
```

Чтобы доказать, что обучение рано или поздно завершится, нам нужен лексикографический кортеж. Давайте посмотрим, как **decreases** `n,h` доказывает завершимость. У нас есть два рекурсивных вызова. Для первого нужно доказать

```
n, h ≻ n, h-1
```

а для второго нужно доказать

```
n, h ≻ n-1, hours
```

Оба этих условия выполняются, поэтому метод `Study` завершается.

Упражнение 3.12

Предположим, что университет требует от профессоров ограничить время изучения предмета `200` часами. В этом случае можно написать постусловие для метода `RequiredStudyTime`:

```
method RequiredStudyTime(c: nat) returns (hours: nat)
  ensures hours <= 200
```

Зная границу, вы все еще можете использовать оценку завершимости `n,h` для `Study`, но теперь появляется возможность написать оценочную функцию без использования лексикографического кортежа. Напишите ее.

3.3.1. Функция Аккермана

Следующий пример – знаменитая функция Аккермана. Она работает долго, но в конце концов завершается. Доказать завершимость легко, если воспользоваться лексикографической парой.

```
function Ack(m: nat, n: nat): nat
  decreases m, n
```

```
{
  if m == 0 then
    n + 1
  else if n == 0 then
    Ack(m - 1, 1)
  else
    Ack(m - 1, Ack(m, n - 1))
}
```

Чтобы убедиться, что функция рано или поздно завершится, мы проверяем уменьшение лексикографической пары `m,n` натуральных чисел с каждым рекурсивным вызовом. Рекурсивный вызов во второй ветви и внешний рекурсивный вызов в третьей ветви используют `m - 1` в качестве первого аргумента, тем самым уменьшая первый компонент пары `m, n`. Во внутреннем рекурсивном вызове в третьей ветви мы видим, что `m, n` превышает `m, n - 1`. Таким образом, гарантии завершимости можно считать доказанными.

Врезка 3.1

Чтобы использовать лексикографический кортеж в Dafny для доказательства завершимости, мы просто перечисляем компоненты кортежа после ключевого слова **decreases**. Язык Dafny поддерживает типы кортежей, но они являются встроенными индуктивными типами, поэтому их порядок определяется структурным включением. Таким образом, **decreases** E,F использует лексикографическую пару E,F как оценочную функцию, а **decreases** (E,F) – единственное значение (E,F).

В лексикографическом упорядочении **true**,5 ≻ **false**, потому что первый компонент **true** кортежа слева превышает **false**. Аналогично в лексикографическом упорядочении ({2,4,6},**true**) ≻ {2,4,6}, потому что множество {2,4,6} структурно включено в пару ({2,4,6},**true**). Однако следующие условия (**true**,5) ≻ **false** и {2,4,6},**true** ≻ {2,4,6} не выполняются.

3.3.2. Взаимно рекурсивные методы

Одна из общих закономерностей взаимно рекурсивных методов – один из методов можно рассматривать как «внешнее» вычисление, вызывающее «внутреннее» вычисление. Эту закономерность демонстрирует вариант примера `Study` из раздела 3.3.0. Здесь я использую `n` как количество *пройденных* предметов из `40`, а `h` как количество часов, оставшихся до конца изучения текущего предмета.

```
method StudyPlan(n: nat)
  requires n <= 40
  decreases 40 - n
```

```
{
  if n == 40 {
    // конец
  } else {
    var hours := RequiredStudyTime(n);
    Learn(n, hours);
  }
}

method Learn(n: nat, h: nat)
  requires n < 40
  decreases 40 - n, h
{
  if h == 0 {
    // изучение предмета n завершено; продолжить изучение
    // других предметов по плану
    StudyPlan(n + 1);
  } else {
    // очередной час изучения предмета...
    Learn(n, h - 1);
  }
}
```

Метод `StudyPlan` – «внешний», а `Learn` – «внутренний».

С учетом указанных инструкций **`decreases`** условия завершимости выражаются так:

Вызов	Условия завершимости
`StudyPlan` вызывает `Learn`	`40 - n ≻ 40 - n, h`
`Learn` вызывает `StudyPlan`	`40 - n, h ≻ 40 - (n + 1)`
`Learn` вызывает `Learn`	`40 - n, h ≻ 40 - n, h - 1`

Для первого вызова условие выполняется, потому что `40-n` совпадает с началом кортежа `40-n, h`. Для второго вызова условие выполняется, потому что первый компонент уменьшается. Для третьего вызова первый компонент остается прежним, а второй компонент уменьшается. Отсюда следует, что оба метода завершаются.

Упражнение 3.13

Добавьте инструкции **`decreases`**, чтобы доказать завершимость следующего варианта программы `StudyPlan`, описанной выше:

```
method Outer(a: nat) {
  if a != 0 {
    var b := RequiredStudyTime(a - 1);
```

```
    Inner(a, b);
  }
}

method Inner(a: nat, b: nat)
  requires 1 <= a
{
  if b == 0 {
    Outer(a - 1);
  } else {
    Inner(a, b - 1);
  }
}
```

Упражнение 3.14

Добавьте инструкции **decreases**, чтобы доказать завершимость следующего варианта программы `StudyPlan`, описанной выше:

```
method Outer(a: nat) {
  if a != 0 {
    var b := RequiredStudyTime(a - 1);
    Inner(a - 1, b);
  }
}

method Inner(a: nat, b: nat) {
  if b == 0 {
    Outer(a);
  } else {
    Inner(a, b - 1);
  }
}
```

3.3.3. Рефакторинг вложенных вычислений

Вот пример, иллюстрирующий полезный трюк с лексикографическими кортежами. Рассмотрим следующую функцию, которая вычисляет $2^n - 1$:

```
function ExpLess1(n: nat): nat {
  if n == 0 then 0 else 2 * ExpLess1(n - 1) + 1
}
```

Легко видеть, что `ExpLess1` завершается, поскольку уменьшает `n`. Теперь предположим, что мы решили вынести вычисление в ветви `else` в функцию `ExpLess2(n)`, которая вычисляет $2^n - 2$:

```
function ExpLess1(n: nat): nat {
  if n == 0 then 0 else ExpLess2(n) + 1 // что можно сказать
                              // о завершимости вызова ExpLess2?
}

function ExpLess2(n: nat): nat
  requires 1 <= n
{
  2 * ExpLess1(n - 1)
}
```

Очевидно, что эта реорганизация не меняет ничего из того, что делает `ExpLess1`, поэтому мы ожидаем, что `ExpLess1` в конечном итоге завершится. Но как доказать завершимость взаимно рекурсивной пары `ExpLess1` и `ExpLess2`?

Если объявить обе функции с инструкцией **`decreases`** `n`, мы сможем доказать завершимость вызова `ExpLess1` из `ExpLess2`, но не докажем завершимость вызова `ExpLess2` из `ExpLess1`, потому что этот вызов передает одно и то же значение для `n`. Если, с другой стороны, добавить **`decreases`** `n` в `ExpLess1` и **`decreases`** `n-1` в `ExpLess2`, то мы докажем завершимость вызова `ExpLess2` из `ExpLess1`, но не докажем завершимость вызова `ExpLess1` из `ExpLess2`.

В такой ситуации, когда одна функция вызывает другую с теми же значениями параметров, мы должны расположить одну из функций «ниже» другой. Это можно сделать, добавив второй компонент в лексикографический кортеж. Значения этого второго компонента неважны, если значение для `ExpLess1` больше, чем для `ExpLess2`. Вот самый простой способ:

```
function ExpLess1(n: nat): nat
  decreases n, 1
{
  if n == 0 then 0 else ExpLess2(n) + 1
}

function ExpLess2(n: nat): nat
  requires 1 <= n
  decreases n, 0
{
  2 * ExpLess1(n - 1)
}
```

При чтении этой спецификации вслух фраза «уменьшает 1» может показаться нелогичной, потому что `1` – это константа. Но не забывайте, что инструкция **`decreases`** – это всего лишь способ указать, как вычислять значения оценочных функций. То есть инструкция **`decreases`** на самом деле просто говорит: «Так должна вычисляться оценка завершимости для этой функции». Следуя нашим правилам проверки завершимости, легко увидеть, что эти инструкции **`decreases`** действительно доказывают завершимость:

Вызов	Условия завершимости
`ExpLess1 вызывает ExpLess2`	`n, 1 ≻ n, 0`
`ExpLess2 вызывает ExpLess1`	`n, 0 ≻ n - 1, 1`

Фактически благодаря правилу, согласно которому если кортеж совпадает с началом другого более длинного кортежа, то более короткий кортеж превышает другой, мы вполне могли бы выбрать **`decreases`** `n` для `ExpLess1` и **`decreases`** `n, 257` для `ExpLess2`.

Упражнение 3.15

Следующие функции `F` и `M` вычисляют *последовательности Хофштадтера женщин и мужчин*:

```
function F(n: nat): nat {
  if n == 0 then 1 else n - M(F(n - 1))
}

function M(n: nat): nat {
  if n == 0 then 0 else n - F(M(n - 1))
}
```

Добавьте инструкции **`decreases`**, чтобы доказать завершимость `F` и `M`.

Упражнение 3.16

В примере `StudyPlan` в разделе 3.3.2 использовалось правило, согласно которому если кортеж совпадает с началом другого более длинного кортежа, то более короткий кортеж превышает более длинный лексикографический кортеж. Напишите инструкции **`decreases`** для `StudyPlan` и `Learn` без использования этого правила, доказывающие завершимость. То есть ваши инструкции **`decreases`** для `StudyPlan` и `Learn` должны быть одинаково длинными.

Упражнение 3.17

Напишите инструкцию **`decreases`**, доказывающую завершимость следующей функции. (Это непросто.)

```
function F(x: nat, y: nat): int {
  if 1000 <= x then
    x + y
  else if x % 2 == 0 then
    F(x + 2, y + 1)
  else if x < 6 then
    F(2 * y, y)
```

```
    else
      F(x - 4, y + 3)
  }
```

3.4. Инструкции decreases по умолчанию

В языке Dafny если рекурсивная функция или метод не имеет явной инструкции **decreases**, то верификатор автоматически создает ее. Оценочная функция по умолчанию определяется как лексикографический кортеж параметров функции или метода, следующих в порядке их объявления.

Например, вот сигнатуры некоторых примеров функций в этой главе и соответствующие инструкции **decreases** по умолчанию, которые они получат, если инструкция **decreases** не будет указана явно:

```
function Fib(n: nat): nat
  decreases n
function SeqSum(s: seq<int>, lo: int, hi: int): int
  decreases s, lo, hi
method Study(n: nat, h: nat)
  decreases n, h
function Ack(m: nat, n: nat): nat
  decreases m, n
method StudyPlan(n: nat)
  decreases n
method Outer(a: nat) // из упражнения 3.13
  decreases a
method Inner(a: nat, b: nat) // из упражнения 3.13
  decreases a, b
```

Оценка завершимости по умолчанию для функции `SeqSum` не подходит для доказательства (для этого требуется `hi - lo`), поэтому инструкцию **decreases** необходимо указать явно. То же касается метода `StudyPlan`, для которого нужно указать оценочную функцию `40 - n`. Но в остальных случаях инструкции **decreases** верные, поэтому если в них опустить инструкции **decreases**, то верификатор автоматически докажет завершимость, не требуя указывать что-либо явно.

Я рассказал вам *не все детали* определения инструкции **decreases** по умолчанию. Фактические правила немного сложнее. К счастью, эти детали не важны, потому что среда разработки на Dafny сообщит их, если навести указатель мыши на выбранную функцию или метод. Благодаря этому можно не утруждать себя запоминанием точных правил выбора инструкций **decreases** по умолчанию в Dafny, а просто посмотреть, какая инструкция используется, и добавить ее явно, если потребуется.

Также желательно помнить, что инструкция **decreases** по умолчанию – это просто предположение, которое, впрочем, очень часто оказывается верным. В оставшейся части книги я буду опускать инструкции **decreases**, если Dafny генерирует верные инструкции **decreases** по умолчанию.

Упражнение 3.18

Попробуйте удалить явные инструкции **decreases** в примерах, показанных в этой главе. Какие из них Dafny угадает правильно? Для этого наводите указатель мыши на определения функций или методов, чтобы увидеть всплывающую подсказку.

3.5. Итоги

Ключом к доказательству завершимости является добавление к рекурсивным (и взаимно рекурсивным) функциям и методам оценок завершимости и демонстрация их уменьшения по мере выполнения программы. Все сравнения оценок завершимости выполняются с использованием фиксированного отношения полной фундированности, потому что отсутствие бесконечно убывающих последовательностей элементов в фундированном порядке подразумевает завершимость.

Функция или метод получает оценку завершимости путем объявления инструкции `decreases`. Эта инструкция принимает список выражений, который интерпретируется как лексикографический кортеж. На практике лексикографические кортежи в оценочных функциях часто совпадают со списком параметров функции или метода. Поэтому принято эффективное соглашение, способствующее уменьшению беспорядка в тексте программы, согласно которому следует явно объявлять инструкцию **decreases** только тогда, когда его нельзя угадать по сигнатуре функции или метода. Dafny поддерживает это соглашение, генерируя инструкции **decreases** по умолчанию, и с этого момента я буду пользоваться этим соглашением и показывать инструкции **decreases**, только когда они не совпадают с инструкциями **decreases** по умолчанию.

Примечания

Функция Аккермана была разработана Вильгельмом Аккерманом (Wilhelm Ackermann) как пример полностью вычислимой функции, выходящей за рамки так называемой *примитивной* рекурсии.

Именно использование фундированного порядка позволяет доказать завершимость. Именно такой подход к доказательству завершимости использовал Флойд [50]. В классической статье [70] вы найдете сравнение нескольких идей доказательства завершимости.

В отсутствие инструкции **decreases** Dafny угадывает ее по *сигнатуре* метода или функции. Иногда он может пытаться *вывести* инструкцию **decreases** по умолчанию из *реализации* метода или функции. В настоящее время существует большое количество работ, посвященных такому выводу (см., например, [21, 33]).

Некоторые программы предназначены для работы в течение неограниченного периода времени. Простым примером является программа чата, которая неустанно принимает и пересылает новые сообщения и заверша-

ет работу, только если пользователь решит выйти из приложения. Другим примером может служить операционная система, которая непрерывно обслуживает запущенные процессы. В ситуациях, когда не требуется доказывать завершимость метода, Dafny позволяет объявить метод с инструкцией **`decreases *`**, которая говорит: «Этому методу разрешено не завершаться». Инструкцию **`decreases *`** можно применять к методам (и циклам), но нельзя к функциям.

Не все инструменты верификации проверяют тотальную корректность. Некоторые проверяют только частичную корректность, а некоторые проверяют комбинацию из частичной и тотальной корректности. Например, верификатор OpenJML [105] проверяет завершимость циклов, но не проверяет завершимость рекурсивных вызовов.

Если бы имелась возможность объявлять функции (такие как `Dubious()` в начале главы) как незавершающиеся, то это могло бы приводить к логической противоречивости. Доказательство завершимости функций обеспечивает математическую непротиворечивость, но, помимо доказательства завершимости, существуют и другие способы обеспечения математической непротиворечивости. Например, ACL2 [71] допускает определять функции с хвостовой рекурсией независимо от того, завершаются они или нет. В этой книге мы будем доказывать завершимость всех функций.

Глава 4

Индуктивные типы данных

Программам часто приходится объявлять свои собственные структуры данных. Один из элегантных и удобных способов – использовать *типы данных* (**`datatype`**). Они определяют, какие формы или *варианты* могут иметь данные, а также значения или *полезную нагрузку*, которую несет каждый вариант. Например, тип данных, представляющий напитки, подаваемые в кофейне, может иметь варианты `MatchaTea` и `Latte`, а полезная нагрузка `Latte` может быть целым числом, описывающим количество порций эспрессо.

Тип данных описывает неизменяемые *значения* (а не изменяемое *состояние*, подобное классам, которые мы будем изучать в главах 16 и 17). Это делает их подходящими для использования не только в коде, но также в спецификациях и доказательствах. Из-за этой математической природы типы данных иногда называют *алгебраическими типами данных*.

Типы данных часто определяются рекурсивно. То есть значения могут конструироваться из простых вариантов или из вариантов, содержащих

другие типы данных. По этой причине их еще называют *индуктивными типами данных*.

4.0. Сине-желтые деревья

Тип данных определяет возможную структуру своих значений, разделенную на различные *варианты*. В качестве примера рассмотрим структуру двоичного дерева, где каждый листовой узел имеет либо синий, либо желтый цвет. Существует три варианта таких сине-желтых деревьев: узел синего листа представляет сине-желтое дерево, узел желтого листа представляет сине-желтое дерево и внутренний узел с двумя поддеревьями (т. е. с двумя компонентами, которые сами являются сине-желтыми деревьями) – тоже сине-желтое дерево. Вот иллюстрированное изображение трех форм сине-желтых деревьев:

В тексте программы назовем этот тип данных `BYTree` и объявим следующим образом:

```
datatype BYTree = BlueLeaf | YellowLeaf | Node(BYTree, BYTree)
```

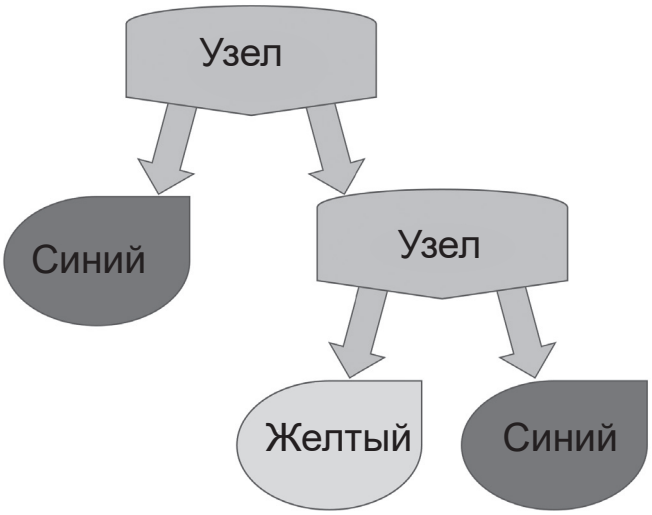
Часть определения справа от знака `=` описывает *все возможные варианты* `BYTree`, которые можно построить. Это важное свойство типов данных означает, что, получив неизвестное значение данного типа, вы будете точно знать, что это один из вариантов, объявленных в этом типе. Каждый вариант имеет именованный *конструктор*. Конструкторы разделяются символами `|`, а порядок их перечисления не имеет значения.

В этом примере `BlueLeaf` и `YellowLeaf` – это конструкторы без параметров, а `Node` – конструктор с двумя параметрами. Dafny позволяет опускать круглые скобки `()` в определениях конструкторов без параметров. В этом примере параметры конструктора `Node` сами имеют тип `BYTree`, но вообще параметры конструктора могут иметь любые другие типы.

Конструкторы используются как функции (или, в случае конструкторов без параметров, как константы), поэтому выражение

```
Node(BlueLeaf, Node(YellowLeaf, BlueLeaf))
```

фактически описывает дерево:



4.1. Сопоставление типов данных

В теле функции, обрабатывающей тип данных, обычно приходится различать его варианты. Это проще всего сделать с помощью выражения **match**. Для иллюстрации рассмотрим функцию, подсчитывающую количество синих узлов в заданном сине-желтом дереве:

```
function BlueCount(t: BYTree): nat {
  match t
  case BlueLeaf => 1
  case YellowLeaf => 0
  case Node(left, right) => BlueCount(left) + BlueCount(right)
}
```

Выражение **match** принимает исходное выражение (в данном случае `t`) и сопровождается несколькими ветвями **case**. Каждая ветвь **case** определяет *шаблон*, и **match** возвращает результат выражения, следующего за `=>` в ветви **case**, шаблон которой соответствует исходному выражению. Шаблон – это конструктор типа данных, параметры которого (если таковые имеются) задаются связанными переменными или вложенными шаблонами. Связанные переменные (`left` и `right` в случае с `Node` в этом примере) связываются со значениями, которые использовались как параметры `Node` при его создании. Область видимости связанных переменных ограничивается телом ветви **case**, в шаблоне которой они объявлены.

Выражение **match** должно включать ветвь **case** для каждого возможного значения, которое может давать исходное выражение. Сопоставление выполняется в порядке следования ветвей **case** в коде.

Упражнение 4.0

Что произойдет, если удалить строку `case Yellow => 0` в функции `BlueCount`?

Если связанная переменная в шаблоне не используется в теле соответствующей ветви **case**, то вместо нее можно указать _. Например, вот функция, которая вычисляет длину самой левой ветви дерева:

```
function LeftDepth(t: BYTree): nat {
  match t
  case BlueLeaf => 0
  case YellowLeaf => 0
  case Node(left, _) => 1 + LeftDepth(left)
}
```

Поскольку этой функции не требуется второй параметр конструктора `Node`, в шаблоне `Node` вместо имени этого параметра используется символ подчеркивание.

Упражнение 4.1

Напишите функцию `ReverseColors`, которая принимает сине-желтое дерево и возвращает точно такое же дерево, за исключением того, что в нем все синие узлы преобразованы в желтые, а желтые – в синие.

Упражнение 4.2

Верификатор обычно используется для доказательства корректного поведения программы при получении любых входных данных. Но его также можно использовать для *проверки* корректности функции при получении некоторых простых входных данных. Например, следующий метод вызывает `ReverseColors` из упражнения 4.1 и использует инструкции **assert**, чтобы проверить поведение функции при получении двух входных данных:

```
method TestReverseColors() {
  var a := Node(BlueLeaf, Node(BlueLeaf, YellowLeaf));
  var b := Node(YellowLeaf, Node(YellowLeaf, BlueLeaf));
  assert ReverseColors(a) == b;
  assert ReverseColors(b) == a;
}
```

Далее в книге мы увидим еще больше подобных примеров тестирования, а пока напишите еще три простых теста для `ReverseColors`. Также напишите «отрицательный тест», который наверняка потерпит неудачу.

Упражнение 4.3

Напишите функцию `Oceanize`, принимающую сине-желтое дерево и возвращающую точно такое же дерево, в котором все листовые узлы преобразованы в синие листовые узлы.

4.2. Дискриминаторы и деструкторы

Иногда бывает нужно узнать вариант значения типа данных, т. е. какой конструктор использовался для создания значения. Для этой цели можно определить функцию. Вот функция `IsNode`, которая возвращает **`true`**, если заданное значение было создано с использованием конструктора `Node`:

```
predicate IsNode(t: BYTree) {
  match t
  case BlueLeaf => false
  case YellowLeaf => false
  case Node(_, _) => true
}
```

(Напомню, что предикат (**`predicate`**) – это функция (`function`), возвращающая результат типа **`bool`**.)

Поскольку `IsNode` – довольно типичная операция, Dafny заранее определяет такой *дискриминатор* для каждого конструктора. Имя дискриминатора совпадает с именем конструктора, за которым следует вопросительный знак, поэтому для типа `BYTree` автоматически будут определены дискриминаторы `BlueLeaf?`, `YellowLeaf?` и `Node?`. Дискриминаторы являются *членами* типа, т. е. доступ к ним осуществляется путем добавления точки и имени члена после выражения. Таким образом, если для выражения `t` типа `BYTree` предикат `IsNode`, который мы определили выше, вызывается как `IsNode(t)`, то аналогичный ему дискриминатор вызывается как `t.Node?`. Обратите внимание, что для конструктора без параметров, такого как `BlueLeaf`, выражение `t.BlueLeaf?` эквивалентно выражению `t == BlueLeaf`.

Еще одна полезная операция, поддерживаемая типами данных **`datatype`**, – извлечение одного из параметров, переданных конструктору. Например, следующая функция извлекает первый параметр `Node`:

```
function GetLeft(t: BYTree): BYTree
  requires t.Node?
{
  match t
  case Node(left, _) => left
}
```

Обратите внимание на предусловие функции `GetLeft`: оно гласит, что функцию можно применить только к варианту `Node` сине-желтого дерева, т. е. выражение **`match`** должно учитывать только один случай.

Поскольку GetLeft – типичная операция, Dafny предлагает удобный способ объявления такого *деструктора* для каждого параметра конструктора простой ссылкой на имя параметра. Например, следующее объявление типа данных определяет деструктор для каждого аргумента Node, давая им имена left и right соответственно:

```
datatype BYTree = BlueLeaf
                | YellowLeaf
                | Node(left: BYTree, right: BYTree)
```

Давать имена деструкторам всех аргументов конструктора совершенно необязательно, но вообще такая возможность имеется.

Как и дискриминаторы, деструкторы являются членами типа данных. Таким образом, если для выражения t типа BYTree функция GetLeft, которую мы определили выше, вызывается как GetLeft(t), то аналогичный ей деструктор вызывается как t.left. Точно так же деструктор автоматически получает предусловие, которое мы объявили в функции GetLeft. В частности, предусловием t.left является t.Node?, поэтому деструктор left можно применять только к вариантам Node значений BYTree.

Обратите внимание, что мы можем написать альтернативную версию BlueCount и использовать в ней дискриминаторы и деструкторы:

```
function BlueCount(t: BYTree): nat {
  if t.BlueLeaf? then 1
  else if t.YellowLeaf? then 0
  else BlueCount(t.left) + BlueCount(t.right)
}
```

Упражнение 4.4

Напишите альтернативную версию функции LeftDepth, использовав дискриминаторы и деструкторы и единственное выражение **if-then-else**.

Упражнение 4.5

Подумайте, что должен делать предикат HasLeftTree, получающий два сине-желтых дерева t и u и определяющий, является ли t внутренним узлом в левом поддереве дерева u. Реализуйте эту функцию двумя способами: с помощью выражения **match** и с помощью дискриминатора и деструктора. Как гарантировать соответствие деструктора своему предусловию?

4.3. Структурное включение

Типы данных часто определяются рекурсивно. Мы видели это выше, где конструктор Node типа BYTree принимает аргументы типа BYTree. Поэтому

функции, обрабатывающие типы данных, тоже часто являются рекурсивными. Мы видели это выше на примере рекурсивных функций `BlueCount` и `LeftDepth`. Как можно убедиться, что такие рекурсивные функции завершаются?

Важным свойством индуктивных типов данных является возможность получения их значений с помощью конечного числа вызовов конструктора. Например, последнее дерево, изображенное в разделе 4.0, можно записать как выражение

```
Node(BlueLeaf, Node(YellowLeaf, BlueLeaf))
```

использующее пять вызовов конструктора. Мы говорим, что параметры типа данных, передаваемые конструктору этого типа данных, *структурно включены* в результат. Ввиду конечности индуктивных типов данных структурное включение является фундированным отношением. Например, для любых значений `t` и `u` типа `BYTree` имеем:

```
Node(t, u) > t
```

и

```
Node(t, u) > u
```

Структурное включение – это встроенный фундированный порядок индуктивных типов данных в Dafny. Например, инструкция **`decreases`** для `BlueCount` и `LeftDepth`, доказывающая их завершимость, имеет вид: **`decreases`** `t`. Поскольку `t` является единственным параметром этих функций, Dafny по умолчанию добавляет эту инструкцию **`decreases`**, если она не указана явно (см. раздел 3.4), что объясняет, почему приведенные выше примеры не вызвали никаких жалоб со стороны верификатора.

4.4. Перечисления

Предположим, что нам нужно определить деревья, листовые узлы которых могут иметь больше цветов, не только синий и желтый. Для этого мы могли бы добавить дополнительные конструкторы, такие как `GreenLeaf` и `RedLeaf`. В качестве альтернативы можно объявить тип данных дерева всего с двумя конструкторами, `Leaf` и `Node`, и параметризовать `Leaf` цветом:

```
datatype ColoredTree = Leaf(Color)
                     | Node(ColoredTree, ColoredTree)
```

Если набор цветов фиксирован, то можно определить тип данных со всеми возможными вариантами цвета. Например:

```
datatype Color = Blue | Yellow | Green | Red
```

В этом особом случае, когда ни один конструктор не имеет параметров, мы говорим, что тип данных является *перечислением*.

Как нетрудно догадаться, для перечислений тоже можно определять функции, например:

```
predicate IsSwedishFlagColor(c: Color) {
  c.Blue? || c.Yellow?
}

predicate IsLithuanianFlagColor(c: Color) {
  c != Blue
}
```

Упражнение 4.6

Определите предикат `IsSwedishColoredTree` для многоцветных деревьев, который определяет, имеют ли все листья цвета, присутствующие на шведском флаге.

4.5. Параметры типа

К настоящему времени мы уже видели два вида деревьев: сине-желтые и многоцветные деревья. Нетрудно представить, что нам также могут понадобиться деревья многих других видов, имеющие общую базовую структуру лист/узел. Чтобы получить такие деревья, можно параметризовать определение дерева типом его полезной нагрузки, т. е. типом данных, хранящихся в дереве.

Вот обобщенное определение бинарных (двоичных) деревьев. В качестве параметра принимается тип `T`:

```
datatype Tree<T> = Leaf(data: T)
                 | Node(left: Tree<T>, right: Tree<T>)
```

Это объявление определяет целое семейство деревьев, по одному для каждого возможного типа `T`. Например, параметризируя `Tree` перечислением `Color`, мы можем записать `Tree<Color>` и получить тип, аналогичный типу `ColoredTree`, определенному выше. Однако мы обычно думаем и говорим о `Tree` (без создания экземпляра параметра) как об одном типе. Тип, принимающий параметры типов, часто называют *обобщенным* типом.

Конечно же, мы можем определить функцию для определенного типа дерева, например:

```
predicate AllBlue(t: Tree<Color>) {
  match t
  case Leaf(c) => c == Blue
  case Node(left, right) => AllBlue(left) && AllBlue(right)
}
```

Такую функцию можно применить только к значениям типа `Tree<Color>`. Однако функцию можно параметризовать типом и получить возможность применять ее к данным любого типа из семейства `Tree`. Вот функция `Size`, параметризованная типом `T` и принимающая значение `t` типа `Tree<T>`, которая возвращает количество листовых узлов в `t`:

```
function Size<T>(t: Tree<T>): nat {
  match t
  case Leaf(_) => 1
  case Node(left, right) => Size<T>(left) + Size<T>(right)
}
```

В точке вызова функции компилятор обычно сам может определить типы аргументов. В таких случаях их можно опустить (вместе с угловыми скобками). Например, функцию `Size` можно определить более привычным способом:

```
function Size<T>(t: Tree<T>): nat {
  match t
  case Leaf(_) => 1
  case Node(left, right) => Size(left) + Size(right)
}
```

Упражнение 4.7

Определите функцию `Mirror`, возвращающую зеркальное отражение заданного дерева.

4.6. Абстрактные синтаксические деревья для выражений

Последний пример в этой главе определяет *абстрактное синтаксическое дерево* (Abstract Syntax Tree, AST) для простых арифметических выражений, таких как `(5 + 3) * 2` и `10 * (x + 7 * y)`. Такие структуры данных используются в каждом компиляторе. Этот пример иллюстрирует взаимную рекурсию, а также определяет общий тип `List`, который мы будем использовать во многих последующих примерах.

Например, выражения имеют три варианта: литеральные константы, переменные и операции. Соответствующий тип данных `Expr`:

```
datatype Expr = Const(nat)
              | Var(string)
              | Node(op: Op, args: List<Expr>)
```

Конструктор `Const` принимает параметр, представляющий числовой литерал. Параметр `Var` – имя переменной. Конструктор `Node` выполняет операцию, которую в этом простом примере я определяю как одну из следующих:

```
datatype Op = Add | Mul
```

и список аргументов `Expr` (любых чисел).

Обобщенный тип `List` определяется так:

```
datatype List<T> = Nil | Cons(head: T, tail: List<T>)
```

При этом используются общие имена `Nil` для обозначения пустого списка и `Cons` для обозначения непустого списка. Непустой список состоит из двух компонентов: головы `head`, несущей полезную нагрузку, и хвоста `tail`, обозначающего остальную часть списка.

Например, арифметическое выражение `10 * (x + 7 * y)` может быть представлено так:

```
Node(Mul,
  Cons(Const(10),
  Cons(Node(Add,
    Cons(Var(“x”),
    Cons(Node(Mul,
      Cons(Const(7),
      Cons(Var(“y”),
      Nil))),
    Nil))),
  Nil)))
```

Давайте определим простой интерпретатор, вычисляющий значение заданного выражения. Поскольку значение переменной зависит от некоторого *окружения*, мы предусматриваем передачу функциям *ассоциативных массивов*, отображающих имена переменных в их значения:

```
function Eval(e: Expr, env: map<string, nat>): nat {
  match e
  case Const(c) => c
  case Var(s) => if s in env.Keys then env[s] else 0
  case Node(op, args) => EvalList(args, op, env)
}

function EvalList(args: List<Expr>, op: Op,
                  env: map<string, nat>): nat
{
  match args
  case Nil =>
    (match op case Add => 0 case Mul => 1)
  case Cons(e, tail) =>
    var v0, v1 := Eval(e, env), EvalList(tail, op, env);
    match op
    case Add => v0 + v1
    case Mul => v0 * v1
}
```

Отмечу четыре важных момента.

Во-первых, позже мы увидим множество примеров использования подобных ассоциативных массивов. А пока вам достаточно знать, что ассоциативный массив типа `map<A, B>` отображает каждое значение `A` из конечного их числа в значение `B`. Первые называются *ключами* ассоциативного массива, а вторые – *значениями*. Набор ключей, имеющихся в ассоциативном массиве, можно извлечь с помощью члена `.Keys`. Таким образом, выражение `a in m.Keys` сообщает, является ли `a` одним из ключей в `m`. Если `a` – это один из ключей в `m`, то выражение `m[a]` вернет соответствующее значение `B`, которое `m` связывает с ключом `a`.

Я хотел, чтобы метод `Eval` мог работать в любом окружении, даже не имеющем переменных, упоминаемых в заданном выражении. Если имя переменной, используемой в выражении, отсутствует в окружении, то `Eval` возвращает значение по умолчанию `0`. Альтернативой может быть требование, выраженное в виде предусловия `Eval`, чтобы заданное окружение имело ключи для всех переменных, упоминаемых в заданном выражении, см. упражнения 4.9 и 4.10.

Во-вторых, в случае `Cons` используются локальные переменные для хранения результатов рекурсивных вызовов `Eval` и `EvalList`. Их можно было бы записать как

```
var v0 := Eval(e, env);
var v1 := EvalList(tail, op, env);
```

но я решил использовать операцию одновременного присваивания. Выбор между двумя способами присваивания – это во многом вопрос личных предпочтений, хотя вариант с одновременным присваиванием возможен, только если правые части можно оценить до того, как произойдет связывание (другими словами, если в выражении, определяющем значение для `v0`, не используется `v1`, и наоборот). В конструкции **`match`** любая переменная, введенная в ветви **`case`**, недоступна за ее пределами.

Привязка **`var`** в выражении обычно называется *let-выражением*. Это название соответствует общему синтаксису функциональных языков:

```
let x = E in F // этот синтаксис не поддерживается в Dafny
```

который оценивает выражение `E`, связывает результат с переменной `x`, а затем приводит к вычислению `F`, в котором может упоминаться `x`. Как мы только что видели, синтаксис такого let-выражения в Dafny имеет следующий вид:

```
var x := E; F // Синтаксис в Dafny, который читается как
              // «пусть x получит значение E из F»
```

который выглядит как аналог инструкции, объявляющей локальные переменные.

В-третьих, обратите внимание, что первое вложенное выражение **match** должно быть заключено в круглые скобки, иначе **case** Cons(e, Tail) будет интерпретироваться как третий вариант в инструкции **match** op. С другой стороны, отметьте, что второе вложенное выражение **match** не требует заключения в круглые скобки, потому что все оставшиеся инструкции **case** принадлежат ему. Синтаксически Dafny позволяет использовать фигурные скобки для группировки всех вариантов **case** в выражении **match**, поэтому вместо (match op...) то же самое выражение можно записать так:

```
match op {
  case Add => 0
  case Mul => 1
}
```

Обратите внимание, что фигурные скобки размещаются иначе, чем круглые.

В-четвертых, функции Eval и EvalList взаимно рекурсивны. Они завершаются, потому что аргумент типа данных, e в Eval и args в EvalList, структурно меньше для каждого (взаимно) рекурсивного вызова. То есть основными составляющими доказательства завершимости являются

```
Node(op, args) ≻ args
Cons(e, tail) ≻ e
Cons(e, tail) ≻ tail
```

Поскольку эти параметры перечислены первыми в сигнатурах функций, добавляемых языком Dafny инструкций **decreases** по умолчанию достаточно, чтобы верификатор мог проверить завершимость.

Упражнение 4.8

Измените сигнатуру функции EvalList, чтобы параметр op: Op предшествовал параметру args: List<Expr>. Это приведет к тому, что Dafny сообщит о невозможности доказать завершимость. Почему? Решите эту проблему, написав явную инструкцию **decreases** для EvalList.

Упражнение 4.9

Напишите предикат GoodEnv(e: Expr, env: **map<string, nat>**), который возвращает **true**, если env содержит имена всех переменных, упоминаемых в e. Для этого потребуется написать вспомогательный предикат не для одного выражения e, а для списка выражений.

Упражнение 4.10

Измените случай **case** Var в функции Eval, чтобы он содержал одно лишь выражение env[s]. Используя предикаты, разработанные в

упражнении 4.9, добавьте необходимые предусловия для проверки `Eval` и `EvalList`.

4.7. Итоги

Типы данных **`datatype`** дают возможность определить набор значений. Значения типа данных разделены на фиксированный набор вариантов. Тип данных имеет один конструктор для каждого варианта. Конструктор похож на функцию в том смысле, что он может принимать параметры. Это особая функция, поскольку она *инъективна*, что означает возможность восстановить параметры, переданные конструктору, из значения, созданного конструктором. Восстановить параметры можно с помощью инструкции или выражения **`match`**, а также с использованием комбинации дискриминаторов и деструкторов.

Я назвал типы данных «индуктивными». По сути, это означает, что каждое из их значений строится снизу вверх за конечное число шагов. Принято говорить, что значение, созданное конструктором, структурно включает любые значения типов данных, передаваемые конструктору в виде параметров. Из-за конечной структуры значений типов данных это структурное включение полностью фундировано.

В следующей главе мы исследуем математическую индукцию – способ записи доказательств в виде более мелких доказательств. Это мало чем отличается от индуктивных типов данных. Действительно, чтобы доказать свойства индуктивных типов данных, почти всегда используется математическая индукция, но существует также множество других приложений математической индукции.

Примечания

Индуктивные типы данных – это основа всех языков и верификаторов, поддерживающих парадигму функционального программирования. Например, определение типа `Tree` на F* [53] записывается так:

```
type tree (t:Type) : Type =
  | Leaf : data:t -> tree t
  | Node : left:tree t -> right:tree t -> tree t
```

Перечисления (т. е. типы данных, конструкторы которых не имеют параметров) поддерживаются большинством императивных языков. Использование полезных нагрузок также возможно в таких языках, как Scala, Rust, Java и C#, но, поскольку эти языки раскрывают детали размещения таких значений в памяти, их сложнее использовать в спецификациях и верификации.

Типы данных, представленные в этой главе, являются индуктивными, т. е. они имеют конечную структуру. Например, продолжая спускаться вниз по левой ветви `BYTree`, вы в конечном итоге доберетесь до листа. Существу-

ют также *коиндуктивные типы данных*, значения которых могут иметь бесконечную структуру, но они гораздо менее распространены. На первый взгляд может показаться, что такие значения невозможны в программах, выполняющихся в ограниченном пространстве памяти компьютера. Но, *лениво* оценивая эти структуры, вполне можно представлять конечные части бесконечных структур, что является отличительной чертой языка программирования Haskell [110]. Двойственность индуктивных и коиндуктивных типов данных подчеркивается в прототипе языка программирования Charity [28]. Коиндуктивные типы данных также могут использоваться в спецификациях для описания поведения некоторых программ. Например, коиндуктивный тип данных можно использовать для представления бесконечного потока запросов, обрабатываемых веб-сервером. Dafny, LiquidHaskell [86] и Agda [22] поддерживают коиндуктивные типы данных, но доказательство свойств программ, использующих их, – сложная тема, выходящая за рамки этой книги.

Глава 5

Леммы и доказательства

Рассуждая о программах, нам часто приходится писать некоторые части доказательств вручную. Если доказательство длинное или требуется использовать одно и то же доказательство в нескольких местах, мы даем доказываемому свойству имя. Это делается с помощью *леммы*, которую можно рассматривать как подпрограмму в более широком доказательстве.

В этой главе основное внимание уделяется написанию и доказательству лемм, которые занимают центральное место в рассуждениях о программах. Как мы увидим далее, между доказательствами и логикой Флойда существует тесная связь. Мы также изучим индукцию и будем писать доказательные вычисления. Наконец, в этой главе вы увидите, как помочь верификатору, если он вдруг сообщит, что не может подтвердить корректность вашей программы.

Размышления и отладка при доказательстве леммы ничем не отличаются от размышлений и отладки при доказательстве корректности программы. Поэтому навыки написания доказательств дадут вам возможность рассуждать о более сложных программах. Научившись писать доказательства, вы сможете начать рассуждать о более интересных аспектах ваших программ, а не тратить время на размышления о том, как диагностировать неудачные попытки доказательства.

5.0. Объявление леммы

Чтобы показать необходимость лемм, начнем с простого примера. Рассмотрим следующую функцию, которая всегда возвращает значение, большее, чем ее аргумент:

```
function More(x: int): int {
  if x <= 0 then 1 else More(x - 2) + 3
}
```

Я сказал, что `More` «всегда возвращает значение, большее, чем ее аргумент». Но так ли это на самом деле? Как убедиться в этом? В процессе размышлений вы бы рассмотрели два случая, соответствующие двум случаям в теле функции. В первом случае, когда `x` меньше или равно `0`, `More(x)` возвращает значение `1`, которое действительно больше такого аргумента `x`. В другом случае, когда `x` имеет строго положительное значение, вы попробовали бы «развернуть» цепочку рекурсивных вызовов до выполнения условия, соответствующего первому случаю. Функция добавляет `3` к результату каждого рекурсивного вызова, поэтому, корректно посчитав рекурсивные вызовы, вы, возможно, сможете убедить себя или, по крайней мере, найти это правдоподобным, что `More(x)` всегда возвращает некоторое значение, которое больше `x`.

Мы докажем это свойство `More` позже (в разделе 5.2). А пока просто подумаем, как записать само это свойство. Я назову это свойство `Increasing` и запишу его так:

```
lemma Increasing(x: int)
  ensures x < More(x)
```

Лемма, или *теорема*, как некоторые назвали бы ее, – это математическое утверждение, имеющее доказательство. В Dafny лемма во многом похожа на метод: у нее есть имя, ее можно параметризовать, у нее есть пред- и постусловие, и ее можно вызвать. Логическое утверждение леммы определяется постусловием. Точно так же, как сторона, вызывающая метод, узнает постусловие после завершения вызова, так и сторона, вызывающая лемму, узнает логическое утверждение, сделанное леммой. Любые ограничения на применение леммы определяются ее предусловием, как и в случае с методами.

Однако между методами и леммами есть разница: лемма никогда не компилируется в выполняемый код и используется только верификатором как призрачная конструкция. На самом деле лемма в Dafny – это призрачный метод. А это означает, что вы уже знаете, как объявлять, использовать и – да – *доказывать* леммы. Но, если рассматривать леммы как математические утверждения, то все перечисленное заслуживает дополнительного объяснения.

5.1. Использование леммы

Рассмотрим пример, иллюстрирующий использование леммы `Increasing`. Следующий метод выполняет два вызова `More`:

```
method ExampleLemmaUse(a: int) {
  var b := More(a);
  var c := More(b);
  assert 2 <= c - a;
}
```

Этот метод вычисляет `More(More(a))` и сохраняет результат в локальную переменную `c`, а затем с помощью инструкции **`assert`** утверждает, что разница между `c` и `a` никак не меньше `2`. Если `More` всегда возвращает значение, больше ее аргумента, то `More(a)` вернет значение больше, чем `a`, т. е. не меньше, чем `a+1`. Второй вызов `More` получает результат первого вызова, поэтому мы ожидаем, что результат этого вызова будет не меньше, чем `a+2`. Но верификатор сообщает, что это утверждение может быть неверным. Причина в том, что верификатор не имеет достаточных сведений о `More`. В частности, мы должны сообщить ему о свойстве `Increasing`, сформулированном в виде леммы в разделе 5.0.

Чтобы доказать утверждение в `ExampleLemmaUse`, воспользуемся леммой `Increasing`. Нам потребуется дважды вызвать лемму, чтобы подтвердить свойство `More` как для `a`, так и для `More(a)`.

```
method ExampleLemmaUse(a: int)
{ ⓪
  var b := More(a); ①
  Increasing(a); ②
  Increasing(b); ③
  var c := More(b); ④
  assert 2 <= c - a; ⑤
}
```

Теперь утверждение **`assert`** подтверждается. ☺

Достаточно всего один раз рассмотреть этот пример во всех подробностях, чтобы увидеть, как связаны леммы и корректность программы. Рассмотрим, что известно на каждом шаге, выполняемом методом.

- В точке ⓪ мы знаем только, что `a` – целое число. Это может быть любое целое число.
- В точке ① мы дополнительно узнаем, что `b` получает значение `More(a)`. Проследив определение `More`, мы могли бы узнать больше, но, поскольку `More` – рекурсивная функция, мы не продвинемся далеко без индуктивного доказательства, которого еще не сделали.
- Далее следует вызов леммы `Increasing`. Это означает, что мы должны доказать соответствие этого вызова предусловию. К счастью,

`Increasing` не имеет объявленного предусловия, поэтому мы имеем тривиальный случай. По возвращении из вызова в точке ② мы получаем информацию о постусловии вызова, а именно `a < More(a)`.

- Аналогично в точке ③ мы дополнительно узнаем постусловие `b < More(b)`. Зная, что `b` равно `More(a)`, мы можем заключить, что

  ```
  More(a) < More(More(a))
  ```

- В следующей строке локальной переменной `c` присваивается значение `More(b)`. Поскольку значение `b` равно `More(a)`, значение `c` равно `More(More(a))`. Таким образом, в точке ④ мы получаем следующие равенство и неравенства:

  ```
  a < More(a) < More(More(a)) == c
  ```

  Поскольку мы имеем дело с целыми числами, эти условия подразумевают `a + 2 <= c`.

- Теперь мы должны доказать, что утверждение **`assert`** верно, учитывая, что в точке ④, как нам известно, оно истинно. Само по себе это утверждение не дает никакой дополнительной информации, поэтому в ⑤ мы знаем то же самое, что и в ④.

Обратите внимание, что вызовы лемм аналогичны вызовам методов в том смысле, что нам нужно (конечно же!) вызывать их с корректными значениями и в нужное время. Например, следующий код не верифицирует утверждение **`assert`**, поскольку он вызывает лемму для `a` и `More(More(a))`, но не вызывает ее для `More(a)`:

```
var b := More(a);
Increasing(a); // это дает нам: a < More(a)
var c := More(b);
Increasing(c); // это дает нам:
               // More(More(a)) < More(More(More(a)))
assert 2 <= c - a; // это утверждение не может быть верифицировано
```

С другой стороны, поскольку лемма не изменяет состояния программы, а лишь дает нам больше знаний об этом состоянии, мы можем записать два вызова леммы в другом порядке:

```
var b := More(a);
Increasing(b); // дает: More(a) < More(More(a))
Increasing(a); // а это дает: a < More(a)
var c := More(b);
assert 2 <= c - a; // верификация выполняется успешно
```

Наконец, как и при вызовах других методов, информация, полученная в результате вызова леммы, применяется только к тем трассам в потоке управления, которые фактически вызывают лемму:

```
Increasing(a); // этот вызов распространяется на все пути в потоке управления
var b := More(a);
var c := More(b);
if a < 1000 {
  Increasing(More(a)); // мы узнаем, что More(a) < More(More(a))
                       // только здесь, т. е., когда a < 1000
  assert 2 <= c - a;   // верификация выполняется успешно
}
assert 2 <= c - a; // это утверждение не может быть верифицировано
```

Последнее утверждение **assert** в примере выше не верифицируется, потому что здесь не хватает информации `More(a) < More(More(a))`, когда выбирается (неявная) ветвь **else** в инструкции **if**.

Упражнение 5.0

Замените последнее утверждение в предыдущем примере на

```
assert 2 <= c - a || 200 <= a;
```

Что получилось? Почему?

Упражнение 5.1

В методе `ExampleLemmaUse` создаются две локальные переменные: `b` и `c`. Вместо создания переменной `c` мы могли бы обновить значение `b`, например так:

```
b := More(b);
```

Попробуйте повторно использовать `b` и исключить переменную `c` во всех пяти вариантах тела `ExampleLemmaUse` в этом разделе. Потребовало ли это скорректировать утверждения и вызовы лемм? Объясните.

5.2. Доказательство леммы

К настоящему моменту мы лишь объявили и использовали лемму, но не доказывали ее. Чтобы доказать лемму, нужно определить ее тело, подобно тому, как мы определяем тело метода. Лемма без тела является недоказанной, ее еще называют *предположением* или *аксиомой*.

Теперь добавим тело нашей леммы, подставив фигурные скобки:

```
lemma Increasing(x: int)

  ensures x < More(x)
{
}
```

Врезка 5.0

Верификатор без лишних слов принимает методы (и леммы) без тела, но компилятор обязательно выразит свое недовольство, встретив такой метод (или лемму). Даже притом что компилятор элиминирует все призрачные объявления, он сначала проверяет их допустимость и в процессе проверки сообщит об отсутствии тела у леммы. В интегрированной среде разработки (IDE) верификатор выполняется в фоновом режиме, но компилятор – нет. Поэтому вы не увидите сообщений об ошибке отсутствия тела у метода, леммы или функции, пока (вручную) не запустите компилятор.

Попробовав сделать это, вы можете удивиться, почему *до сих пор* не получили никаких сообщений от верификатора. А причина в том, что верификатор автоматически доказывает лемму, не требуя никаких дополнительных действий с вашей стороны (но вы должны объявить пустое тело, иначе верификатор даже не будет пытаться что-либо доказать). За кулисами верификатор применит автоматическую индукцию, и ее будет достаточно, чтобы доказать эту простую лемму. Это действительно слишком просто! Это ценное свойство, но оно не поможет нам получить знания о доказательствах и индукции. Поэтому до конца этой главы мы отключим автоматическую индукцию для доказательства лемм. Для этого нужно добавить в объявление леммы атрибут {`:induction` **`false`**}. (Также можно зайти в настройки Dafny IDE и повсюду отключить автоматическую индукцию.)

Итак, вот наша отправная точка:

```
lemma {:induction false} Increasing(x: int)
  ensures x < More(x)
{
} // теперь лемма не доказывается верификатором
```

Такое определение леммы заставит верификатор сообщить об ошибке, поскольку он не сможет убедиться, что все пути в потоке управления через тело метода гарантируют выполнение постусловия. Вспомним наши неформальные рассуждения в разделе 5.0. Мы говорили, что случай `x <= 0` легко доказуем. Давайте докажем его явно в тексте программы, для чего добавим инструкцию **`if`**:

```
lemma {:induction false} Increasing(x: int)
  ensures x < More(x)
{
  if x <= 0 { // этот случай верифицируется
  } else { // этот случай пока не верифицируется
  }
}
```

В ответ верификатор все так же выводит сообщение об ошибке, но теперь выделяет только ветвь **else** – он доказал постусловие для всех путей в потоке управления, проходящих через ветвь **then**. (Строго говоря, рано радоваться доказательству ветви **then**. Верификатор обычно рассматривает управляющие конструкции в порядке их следования в тексте программы, но не всегда.) Ветвь **then** выбирается, когда выполняется условие `x <= 0`; в этом случае `More(x)` возвращает 1, поэтому мы можем заключить, что `x <= 0 < 1 == More(x)`.

Совпадение условия **if** в лемме о функции `More` с условием **if** в определении функции `More` не случайное. Обычно доказательство зависит от того, какая часть определения функции используется. Хорошее практическое правило: выстраивать доказательство в соответствии с синтаксической структурой доказываемого.

Чтобы верификатор смог проверить ветвь **else** леммы, мы должны немного помочь ему. По определению `More(x)` равно `More(x-2) + 3`, когда условие `x <= 0` не выполняется. Поэтому мы должны доказать `x < More(x-2) + 3` или, что то же самое, `x-3 < More(x-2)`. Эта формула имеет ту же форму, что и в нашей лемме. Чтобы представить эту информацию, нам нужно лишь вызвать лемму о `x-2`:

```
lemma {:induction false} Increasing(x: int)
  ensures x < More(x)
{
  if x <= 0 {
  } else {
    Increasing(x-2); // этот вызов дает нам: x-2 < More(x-2)
  }
}
```

На этом доказательство завершено, но не будем забывать одну деталь, о которой верификатор позаботился автоматически. Поскольку лемма вызывается рекурсивно, нам нужно доказать, что она завершается. По умолчанию Dafny использует оценочную функцию **decreases** `x` (см. раздел 3.4), и действительно наш рекурсивный вызов уменьшает эту оценочную функцию.

Проверки завершимости важны, так как без них доказательства, подобные следующему, не принимаются:

```
lemma {:induction false} Increasing(x: int)
  ensures x < More(x)
{
  Increasing(x); // этот вызов мог бы дать нам x < More(x), но
                 // проверка завершимости терпит неудачу
}
```

Эта попытка доказательства терпит неудачу по той же причине, что и попытка реализации метода `Impossible` в разделе 3.0 – в обоих случаях делается попытка достичь постусловия с помощью бесконечной рекурсии.

Упражнение 5.2

Используя тот же уровень детализации, что и в маркированном списке в разделе 5.1, опишите, что известно в каждой точке в последней хорошей версии леммы `Increasing`.

Рекурсивный вызов леммы позволяет получить свойство, которое мы пытаемся доказать для меньшего экземпляра задачи. Это называется *индукцией*. *Принцип индукции* гласит, что при доказательстве свойства `P(x)` мы можем предположить, что `P(x')` выполняется для любого `x'`, меньшего `x`. «Меньший» означает меньший в фундированном порядке, как обсуждалось в разделе 3.2. Если просто думать о лемме как о методе, то все сводится к доказательству завершимости рекурсивных вызовов. Тем не менее, следуя стандартной математической терминологии, мы говорим, что рекурсивные вызовы лемм позволяют получить *индуктивное предположение*.

Как гласит наше эмпирическое правило, доказательство имеет тенденцию следовать той же структуре, что и доказываемая формула. Однако ему не обязательно иметь ту же синтаксическую форму. Например, мы можем изменить порядок ветвей **`then`** и **`else`**, можем опустить пустые ветви **`else`** и можем использовать локальные переменные. Вот еще одно доказательство леммы `Increasing`:

```
lemma {:induction false} Increasing(x: int)
  ensures x < More(x)
{
  if 0 < x {
    var y := x - 2;
    Increasing(y);
  }
}
```

5.3. Назад к основам

Доказательство леммы `Increasing` получилось довольно коротким. Оно убеждает верификатор, но что, если оно не убеждает нас самих? Или если доказательство выглядит убедительным, но было бы интересно понять, как мы пришли к нему. Давайте сосредоточимся на более подробной информации, начав с простой леммы `Increasing`.

5.3.0. Доказательство с помощью логики Флойда

В главе 2 я представил логику Флойда как основу для рассуждений о программах. Поскольку доказательства лемм – это всего лишь реализации методов, логика Флойда применима и к ним. Воспользуемся этим приемом, чтобы построить доказательство `Increasing`.

Общее доказательство `Increasing` начинается с предусловия леммы (в данном случае неявно устанавливается предусловие **`requires true`**) и переходит

к ее постусловию (цель доказательства: **ensures** `x < More(x)`). Таким образом, используя тройки Хоара, мы можем записать тело `Increasing` так:

```
lemma {:induction false} Increasing(x: int)
  ensures x < More(x)
{
  {{ true }}
  ?
  {{ x < More(x) }}
}
```

Сосредоточившись на том месте, где я поставил вопросительный знак, имеет смысл разбить доказательство на два случая, соответствующих двум случаям в определении `More`. Используя условные инструкции (см. раздел 2.5), запишем:

```
{{ true }}
if x <= 0 {
  {{ x <= 0 }}
  ?
  {{ x < More(x) }}
} else {
  {{ 0 < x }}
  ?
  {{ x < More(x) }}
}
{{ x < More(x) }}
```

Добавление инструкции **if** мотивировано определением `More`, поэтому добавим в каждую ветвь факт в форме `More(x) == ...`. Мы будем использовать нотацию последовательных формул, как в разделе 2.3.1:

```
{{ true }}
if x <= 0 {
  {{ x <= 0 }}
  {{ x <= 0 && More(x) == 1 }} // определение More для x <= 0
  ?
  {{ x < More(x) }}
} else {
  {{ 0 < x }}
  {{ 0 < x && More(x) == More(x - 2) + 3 }} // определение More для 0 < x
  ?
  {{ x < More(x) }}
}
{{ x < More(x) }}
```

Формула после первого знака вопроса вытекает из формулы перед ним. Это означает, что мы закончили доказательство первой ветви условного выражения.

Перейдя ко второй ветви, мы должны узнать больше о `More(x - 2)`. Вызывая лемму `Increasing` для `x - 2`, получаем индуктивное предположение `x - 2 < More(x - 2)`. Продолжим расширять ветвь **`else`** нашего доказательства:

```
{{ 0 < x }}
{{ 0 < x && More(x) == More(x - 2) + 3 }} // определение More для 0 < x
Increasing(x - 2); // индуктивное предположение
{{ 0 < x && More(x) == More(x - 2) + 3 && x - 2 < More(x - 2) }}
?
{{ x < More(x) }}
```

Похоже, нам больше не понадобится условие `0 < x`, поэтому отбросим его и перепишем `x - 2 < More(x - 2)`, прибавив `3` с каждой стороны:

```
{{ 0 < x }}
{{ 0 < x && More(x) == More(x - 2) + 3 }} // определение More для 0 < x
Increasing(x - 2); // индуктивное предположение
{{ 0 < x && More(x) == More(x - 2) + 3 && x - 2 < More(x - 2) }}
{{ More(x) == More(x - 2) + 3 && x + 1 < More(x - 2) + 3 }}
?
{{ x < More(x) }}
```

Два факта о `More(x - 2) + 3` в строке перед вопросительным знаком подразумевают `x + 1 < More(x)`, поэтому пишем:

```
{{ 0 < x }}
{{ 0 < x && More(x) == More(x - 2) + 3 }} // определение More для 0 < x
Increasing(x - 2); // индуктивное предположение
{{ 0 < x && More(x) == More(x - 2) + 3 && x - 2 < More(x - 2) }}
{{ More(x) == More(x - 2) + 3 && x + 1 < More(x - 2) + 3 }}
{{ x + 1 < More(x) }}
?
{{ x < More(x) }}
```

Теперь нетрудно заметить, что строка перед вопросительным знаком подразумевает строку после вопросительного знака (поскольку если значение `More(x)` превышает `x + 1`, то оно также превышает `x`). Итак, мы завершили доказательство.

В итоге полное доказательство `Increasing` выглядит следующим образом:

```
lemma {:induction false} Increasing(x: int)
  ensures x < More(x)
{
  {{ true }}
  if x <= 0 {
    {{ x <= 0 }}
    {{ x <= 0 && More(x) == 1 }} // определение More для x <= 0
    {{ x < More(x) }}
```

```
  } else {
    {{ 0 < x }}
    {{ 0 < x && More(x) == More(x - 2) + 3 }} // определение More для 0 < x
    Increasing(x - 2); // индуктивное предположение
    {{ 0 < x && More(x) == More(x - 2) + 3 &&
        x - 2 < More(x - 2) }}
    {{ More(x) == More(x - 2) + 3 &&
        x + 1 < More(x - 2) + 3 }}
    {{ x + 1 < More(x) }}
    {{ x < More(x) }}
  }
  {{ x < More(x) }}
}
```

В этом доказательстве с помощью логики Флойда вы можете проверить каждый шаг (т. е. каждую тройку Хоара и каждую пару последовательных формул). При проверке рекурсивного вызова `Increasing` вам также нужно проверить завершимость, о чем я умалчивал при построении этого доказательства.

5.3.1. Доказательства утверждений

Доказательство с помощью логики Флойда, приведенное выше, использует множество обозначений {{ }}, которые хорошо выглядят на бумаге, но не являются допустимыми синтаксическими конструкциями языка Dafny. А что, если нам понадобится помочь верификатору проверить эти условия? В этом случае мы можем заменить каждую из формул инструкцией утверждения **`assert`**.

Захваченные этой идеей, мы можем записать доказательство нашей леммы следующим образом (которое *примет* Dafny):

```
lemma {:induction false} Increasing(x: int)
  ensures x < More(x)
{
  assert true;
  if x <= 0 {
    assert x <= 0;
    assert x <= 0 && More(x) == 1; // определение More для x <= 0
    assert x < More(x);
  } else {
    assert 0 < x;
    assert 0 < x && More(x) == More(x - 2) + 3; // определение More для 0 < x
    Increasing(x - 2); // индуктивное предположение
    assert 0 < x && More(x) == More(x - 2) + 3 &&
        x - 2 < More(x - 2);
    assert More(x) == More(x - 2) + 3 &&
        x + 1 < More(x - 2) + 3;
```

```
    assert x + 1 < More(x);
    assert x < More(x);
  }
  assert x < More(x);
}
```

Если вы допустите ошибку в каком-либо из утверждений **assert**, то верификатор немедленно сообщит вам об этом. Отлично! Но такой код, насыщенный деталями, трудно читать. Хотелось бы, чтобы доказательства и программы были как можно более удобочитаемыми. Итак, давайте подумаем, как этого добиться.

Опустим первое и последнее утверждения **assert**, поскольку они описывают пред- и постусловие самой леммы. Также опустим утверждения в инструкции **if**, которые просто копируют условие из **if** (или его отрицание) в начало каждой ветви и постусловие в конец каждой ветви. В результате у нас остается:

```
lemma {:induction false} Increasing(x: int)
  ensures x < More(x)
{
  assert true;
  if x <= 0 {
    assert x <= 0 && More(x) == 1; // определение More для x <= 0
  } else {
    assert 0 < x && More(x) == More(x - 2) + 3; // определение More для 0 < x
    Increasing(x - 2); // индуктивное предположение
    assert 0 < x && More(x) == More(x - 2) + 3 &&
        x - 2 < More(x - 2);
    assert More(x) == More(x - 2) + 3 &&
        x + 1 < More(x - 2) + 3;
    assert x + 1 < More(x);
  }
}
```

Если из текста программы понятно, что переменные в формуле не изменяются, то формулу повторять не нужно. Поэтому опустим все повторяющиеся формулы. Это дает нам:

```
lemma {:induction false} Increasing(x: int)
  ensures x < More(x)
{
  assert true;
  if x <= 0 {
    assert x <= 0 && More(x) == 1; // определение More для x <= 0
  } else {
    assert 0 < x && More(x) == More(x - 2) + 3; // определение More для 0 < x
    Increasing(x - 2); // индуктивное предположение
```

```
    assert x - 2 < Моге(x - 2); // что мы получаем от рекурсивного вызова
    assert x + 1 < Моге(x - 2) + 3; // прибавляем 3 с каждой стороны
    assert x + 1 < Моге(x); // предыдущая строка и определение Моге выше
  }
}
```

И снова верификатор немедленно предупредит вас о любых ошибках, которые вы допустите при вводе этих утверждений.

Упражнение 5.3

Измените некоторые утверждения **assert** выше, намеренно внеся ошибки, и посмотрите, что из этого получится. Проверьте также, что произойдет, если опустить рекурсивный вызов `Increasing`?

5.4. Доказательные вычисления

Есть более структурированные способы записи доказательств. Я уже использовал формат записи последовательностей выражений и объяснял, почему каждое последующее выражение равно предыдущему. Такую форму записи, например, я использовал в вычислениях слабейших предусловий в разделах 2.7.2, 2.10 и 2.11. Аналогичный стиль утверждений, чередующихся с пояснениями, я использовал в примере `Dubious` в разделе 3.0. И последовательность формул, являющаяся формой пошаговых преобразований, тоже уже использовалась в разделе 2.3.1 для доказательства, что одна формула подразумевает другую. Эта структурированная форма записи доказательств называется *доказательными вычислениями*. Она напрямую поддерживается в Dafny, где широко известна как «инструкция **calc**».

Доказательные вычисления дают список выражений, связанных цепочкой операторов. Например:

```
calc {
  5 * (x + 3);
==  // закон дистрибутивности умножения относительно сложения
  5 * x + 5 * 3;
==  // использовать тот факт, что 5 * 3 == 15
  5 * x + 15;
}
```

Здесь утверждается, что значение выражения `5 * (x + 3)` равно значению выражения `5*x + 5*3` и, в свою очередь, что `5*x + 5*3` равно `5 * x + 15`. Обратите внимание, что каждое выражение завершается точкой с запятой, поэтому Dafny не интерпретирует оператор как часть выражения. (Если бы вы писали выражения на бумаге, то опустили бы эти точки с запятой, потому что разрывы строк и отступы наглядно показывают границы выражений. Dafny же требует добавлять точки с запятыми, поскольку про-

белы не имеют значения для него.) Верификатор проверяет каждый шаг вычисления, а затем на основании транзитивности равенства приходит к выводу, что `5 * (x + 3)` равно `5 * x + 15`. Когда, как в этом примере, в роли оператора используется оператор равенства, доказательное вычисление можно рассматривать как серию преобразований, сохраняющих эквивалентность: выражение `5 * (x + 3)` преобразуется в `5*x + 5*3`, а `5*x + 5*3` преобразуется в `5 * x + 15`.

Вот еще один пример:

```
calc {
  (x + y) * (x - y);
==  // дистрибутивность * относительно + и - (т. е. перекрестное умножение)
  x*x - x*y + y*x - y*y;
==  // * -- коммутативная операция: y*x == x*y
  x*x - x*y + x*y - y*y;
==  // члены - x*y и + x*y взаимно уничтожаются
  x*x - y*y;
}
```

В вычислениях можно использовать и другие операторы, если цепочка идет в том же направлении. Например, если `n` имеет тип **`nat`**, то мы имеем:

```
calc {
  3*x + n + n;
==  // n + n == 2*n
  3*x + 2*n;
<=  // 2*n <= 3*n, поскольку 0 <= n
  3*x + 3*n;
==  // дистрибутивность * относительно +
  3 * (x + n);
}
```

Проверяя каждый шаг в этом вычислении, приходим к выводу:

```
3*x + n + n <= 3 * (x + n)
```

согласно транзитивности цепочки операторов. Обратите внимание, что условие в середине выполняется благодаря нашему предположению `0 <= n`. Если бы `n` могло быть произвольным целым числом, то этот шаг доказательства оказался бы ошибочным (и верификатор отметил бы его как таковой).

Упражнение 5.4

Для каждой из следующих формул напишите доказательное вычисление, доказывающее его. Затем используйте Dafny, чтобы объявить лемму о свойстве, и запишите свое доказательство с помощью инструкции **`calc`**.

а. `5*x - 3 * (y + x) == 2*x - 3*y`

б. `2 * (x + 4*y + 7) - 10 == 2*x + 8*y + 4`

в. `7*x + 5 < (x + 3) * (x + 4)`

Упражнение 5.5

Для логических выражений `A`, `B` и `C` объясните разницу между:

```
calc {
  A;
==>
  B;
==>
  C;
}
```

и

```
assert A ==> B ==> C;
```

5.4.0. Подсказки с использованием проверенных доказательств

Некоторые этапы вычислений могут потребовать дальнейшего обоснования. Другими словами, верификатору могут потребоваться дополнительные подсказки. Например, для проверки корректности шага может потребоваться обращение к различным леммам. Такие подсказки можно давать в фигурных скобках после оператора, связывающего две последовательные строки в вычислении. Даже для шагов, которые верификатор может выполнить автоматически, нелишним будет предоставить удобочитаемый комментарий, что я и делаю в этой книге.

Вот версия **`calc`** с более подробным доказательством леммы `Increasing`, приведенной в разделе 5.2:

```
lemma {:induction false} Increasing(x: int)
  ensures x < Moge(x)
{
  if x <= 0 {
    // углубимся в детали, чтобы проиллюстрировать, что можно сделать
    calc {
      x;
    <= { assert x <= 0; }
      0;
    < // арифметика
      1;
    == // определение Moge
```

```
      Moge(x);
    }
  } else {
    // вычисление
    calc {
      Moge(x);
    == // определение Moge, поскольку 0 < x
      Moge(x - 2) + 3;
    > { Increasing(x - 2); }
      x - 2 + 3;
    > // арифметика
      x;
    }
  }
}
```

В обоих вычислениях выражения имеют целочисленный тип. Вот способ записать второе вычисление с использованием логических выражений:

```
calc {
  true;
==> { Increasing(x - 2); }
  x - 2 < Moge(x - 2);
== // прибавляем 3 с каждой стороны
  x + 1 < Moge(x - 2) + 3;
== // определение Moge, поскольку 0 < x
  x + 1 < Moge(x);
==> // арифметика
  x < Moge(x);
}
```

Обычно вычисления начинают с «более сложной» стороны, поскольку это дает больше подсказок относительно того, что следует применять на последующих шагах. Следуя этой практике, предыдущую инструкцию **calc** можно сформулировать в обратном порядке:

```
calc {
  x < Moge(x);
<== // арифметика
  x + 1 < Moge(x);
== // определение Moge, поскольку 0 < x
  x + 1 < Moge(x - 2) + 3;
== // вычитаем 3 с каждой стороны
  x - 2 < Moge(x - 2);
<== { Increasing(x - 2); }
  true;
}
```

При чтении этого доказательства сверху вниз внезапное введение +1 на первом шаге может стать неожиданностью. Почему мы решили прибавить 1 к x на первом шаге доказательства? Если это действительно стало неожиданностью для вас, то попробуйте изменить порядок шагов, например так:

```
calc {
  x < More(x);
== // определение More, поскольку 0 < x
  x < More(x - 2) + 3;
== // вычитаем 3 с каждой стороны, чтобы получить только More(x-2) справа
  x - 3 < More(x - 2);
<== // арифметика
  x - 2 < More(x - 2);
<== { Increasing(x - 2); }
  true;
}
```

Порядок формирования инструкции **calc** и вообще любого доказательства в основном зависит от личного вкуса.

Упражнение 5.6

Сформулируйте и докажите лемму о том, что для любого натурального числа n выполняется условие Ack(1, n) == n + 2, где Ack – функция Аккермана, определенная в разделе 3.3.1.

5.5. Пример: Reduce

Приведу еще один пример, включающий арифметику, леммы, доказательства и индукцию. В примере используется следующая функция, которая, в отличие от приводившейся выше функции More, менее очевидна:

```
function Reduce(m: nat, x: int): int {
  if m == 0 then x else Reduce(m / 2, x + 1) - m
}
```

Судя по телу этой функции, иногда она возвращает аргумент x, а иногда вычитает положительное значение из значения, возвращаемого рекурсивным вызовом. С каждым рекурсивным вызовом аргумент x увеличивается на 1, а вычитаемое значение уменьшается вдвое. Получается, что Reduce(m, x) никогда не сможет вернуть значение больше, чем x, верно? Попробуем это доказать.

5.5.0. Верхняя граница

Объявим лемму (опять же отключив автоматическую индукцию в Dafny, чтобы получить возможность сделать что-то самим) и, следуя определению Reduce, рассмотрим в теле леммы два случая:

```
lemma {:induction false} ReduceUpperBound(m: nat, x: int)
  ensures Reduce(m, x) <= x
{
  if m == 0 {
    // простой случай
  } else {
    // ...
  }
}
```

Случай, когда значение `m` равно `0`, непосредственно следует из определения `Reduce`, поскольку `Reduce(0, x) == x`. Верификатор автоматически выполнит этот тривиальный шаг, но, чтобы сделать доказательство более удобочитаемым для человека, мы могли бы добавить инструкцию **`assert`** с этим условием.

Другой случай более интересный. Начнем доказательное вычисление с выражения `Reduce(m, x)` и применим к нему определение `Reduce`:

```
calc {
  Reduce(m, x);
== // определение Reduce
  Reduce(m / 2, x + 1) - m;
```

Чтобы получить больше информации об этом вызове `Reduce`, применим индуктивное предположение. Для нашей собственной выгоды (поскольку в данный момент мы только учимся писать доказательства и потому делаем небольшие шаги) мы можем сопроводить рекурсивный вызов леммы утверждением **`assert`**, которое описывает постусловие леммы с применением заданных нами параметров:

```
== { ReduceUpperBound(m / 2, x + 1);
  assert Reduce(m / 2, x + 1) <= x + 1; }
```

Верификатор подтверждает, что мы можем ссылаться на индуктивное предположение по заданным аргументам. Другими словами, верификатор проверяет допустимость рекурсивного вызова. Напомню, эта проверка сводится к верификации предусловия и завершимости вызова (последнее следует из условия `m/2 < m`). Верификатор также подтверждает, что написанное нами утверждение верно.

Хорошо, и что теперь? У нас нет возможности выполнить преобразование, сохраняющее равенство, на основе информации, которую мы получили в этой подсказке. Видимо, мы поспешили написать `==` перед подсказкой. Информация, полученная с помощью индуктивного предположения, действительно дает возможность «вырастить» выражение `Reduce(m/2, x+1)` до чего-то большего, а именно `x+1`. Поэтому на этом этапе доказательства используем `<=` в качестве оператора отношения, что дает нам следующий шаг:

```
    Reduce(m / 2, x + 1) - m;
  <= { ReduceUpperBound(m / 2, x + 1);
       assert Reduce(m / 2, x + 1) <= x + 1; }
    x + 1 - m;
```

Выглядит неплохо, поскольку теперь мы исключили из выражения любое упоминание `Reduce`. Используя тот факт, что в этой ветви доказательства `m` строго положительно, мы можем преобразовать `x + 1 - m` в `x`. Вот полная лемма и ее доказательство:

```
lemma {:induction false} ReduceUpperBound(m: nat, x: int)
  ensures Reduce(m, x) <= x
{
  if m == 0 {
    // простой случай
  } else {
    calc {
      Reduce(m, x);
    == // определение Reduce
      Reduce(m / 2, x + 1) - m;
    <=  { ReduceUpperBound(m / 2, x + 1);
          assert Reduce(m / 2, x + 1) <= x + 1; }
      x + 1 - m;
    <= { assert 0 < m; }
      x;
    }
  }
}
```

Это доказательство имеет уровень детализации, подходящий для обучения на данном этапе. Оно также дает представление о том, как разрабатывать доказательства и как «отлаживать» неудачные попытки верификации. Это хорошие навыки для изучения, и они требуют практики. Поэтому в оставшейся части этой главы я продолжу писать доказательства, включающие больше деталей, чем нужно верификатору.

Упражнение 5.7

Напишите пять вариантов доказательства `ReduceUpperBound`. Вот кое-что, что вы можете попробовать: выполните вычислительные шаги в обратном порядке (начиная с `x` и заканчивая `Reduce(m, x)`), удалите шаги доказательства, которые применяют определение функции, удалите инструкции **`assert`**, замените вычисление простым содержимым подсказок, измените условие в **`if`** на `m != 0` и опустите ветвь **`else`**. (Здесь и в других местах книги вы можете попытаться самостоятельно изменить или исключить некоторые этапы верификации, чтобы узнать, что верификатор может делать автоматически, и выработать свой стиль и уровень детализации.)

Врезка 5.1

Dafny IDE может дать обратную связь о каждой строке доказательства прямо в процессе ее ввода. Это помогает быстро выявлять любые проблемы в вашем мышлении. Верификатор запускается автоматически в фоновом режиме, но только если программа не содержит синтаксических ошибок и прошла проверку типов данных. Поэтому я советую сразу же добавлять закрывающую фигурную скобку в инструкцию `calc`. Это обеспечит синтаксическую корректность программы и позволит верификатору работать.

5.5.1. Нижняя граница

Мы доказали верхнюю границу функции `Reduce(m, x)`. Докажем теперь нижнюю границу, а именно `x - 2 * m`. Начнем с объявления леммы и отдельно реализуем случай, когда `m` равно `0`:

```
lemma {:induction false} ReduceLowerBound(m: nat, x: int)
  ensures x - 2 * m <= Reduce(m, x)
{
  if m == 0 {
  } else {
    // ...
  }
}
```

Верификатор сразу заверяет нас, что с ветвью **`then`** все в порядке. Теперь перейдем к доказательному вычислению для ветви **`else`**. На этот раз начнем с правой части проверяемого условия, потому что она дает нам больше подсказок о том, как продолжить доказательство. Поскольку мы пытаемся доказать, что правая часть, по крайней мере, не меньше левой части, мы можем включить в вычисления шаги, способные уменьшить значение выражения, которым мы манипулируем. Вот наш первый шаг:

```
calc {
  Reduce(m, x);
==  // определение Reduce
  Reduce(m / 2, x + 1) - m;
```

Далее используем индуктивное предположение, как мы делали это при доказательстве `ReduceUpperBound`, но на этот раз неравенство идет в другую сторону:

```
  Reduce(m / 2, x + 1) - m;
>=  { ReduceLowerBound(m / 2, x + 1);
```

```
      assert x + 1 - 2 * (m / 2) <= Reduce(m / 2, x + 1); }
  x + 1 - 2 * (m / 2) - m;
```

Вы уже видите остальные доказательства? Верификатор доволен, если судить по отсутствию сообщений об ошибках. Остальная часть доказательства лично для меня неочевидна ☺, поэтому добавим еще несколько шагов. Первое, что приходит на ум, – удвоение того, что сначала мы уменьшили вдвое. Этот шаг вернет нас туда, откуда мы начали. Выразим это следующим образом:

```
  x + 1 - 2 * (m / 2) - m;
== // арифметика
  x + 1 - m - m; // ошибка: этот шаг доказательства может нарушить условие
```

Но верификатор сообщает об ошибке. Возможно, ему не хватает информации. Давайте отладим этот шаг и дадим подсказку верификатору. Это позволит нам выяснить, что не так или чего не хватает.

```
  x + 1 - 2 * (m / 2) - m;
==  { assert 2 * (m / 2) == m; } // ошибка
  x + 1 - m - m;
```

Это утверждение **assert** выражает свойство, пришедшее нам на ум. Но верификатор снова недоволен. Почему? Разве у него нет таких же обширных познаний в арифметике, как у нас? Иногда это действительно так. Но здесь верификатор имеет полное право выразить свое недовольство утверждением, поскольку записанное нами условие относится к целым числам (не к действительным) и выполняется только для четных целых чисел. Если `m` нечетно, то результат `2*(m/2)` получится на единицу меньше, чем `m`. (Если вам интересно, знает ли об этом верификатор, то можете написать следующие утверждения:

```
assert m % 2 == 0 ==> 2 * (m / 2) == m;
assert m % 2 == 1 ==> 2 * (m / 2) == m - 1;
```

Вы увидите, что он действительно осведомлен о правилах целочисленной арифметики.) Что еще нам осталось доказать? Давайте изменим утверждение так, чтобы отразить тот факт, что `m` может быть больше `2*(m/2)`. Кроме того, поскольку член `2*(m/2)` находится в отрицательном контексте (т. е. перед ним стоит знак минус), замена `2*(m/2)` на `m` может сократить выражение, поэтому используем оператор отношения `>=` на этапе вычислений:

```
  x + 1 - 2 * (m / 2) - m;
>=  { assert 2 * (m / 2) <= m; }
  x + 1 - m - m;
```

Вычитая 1 в этой строке, мы делаем результат еще меньше и приходим к формуле в левой части проверяемого условия. Вот полная лемма и доказательство:

```
lemma {:induction false} ReduceLowerBound(m: nat, x: int)
  ensures x - 2 * m <= Reduce(m, x)
{
  if m == 0 {
  } else {
    calc {
      Reduce(m, x);
    ==  // определение Reduce
      Reduce(m / 2, x + 1) - m;
    >=  { ReduceLowerBound(m / 2, x + 1);
          assert x + 1 - 2 * (m / 2) <= Reduce(m / 2, x + 1); }
      x + 1 - 2 * (m / 2) - m;
    >= { assert 2 * (m / 2) <= m; }
      x + 1 - m - m;
    >  // уменьшаем еще на 1
      x - 2 * m;
    }
  }
}
```

Обратите внимание, что на последнем шаге вместо >= мы можем использовать >. Таким образом, вычисление доказательства фактически доказывает, что `Reduce(m, x) > x - 2 * m`.

Упражнение 5.8

Возможно, вам удастся доказать более точную нижнюю границу. Попробуйте изменить постусловие леммы на `x - 2 * m < Reduce(m, x)`. Что из этого выйдет?

5.6. Пример: коммутативность умножения

В качестве упражнения давайте определим функцию, которая выполняет умножение натуральных чисел путем многократного сложения:

```
function Mult(x: nat, y: nat): nat {
  if y == 0 then 0 else x + Mult(x, y - 1)
}
```

Если эта функция на самом деле выполняет умножение в том виде, в каком мы его знаем, то она должна поддерживать свойство коммутативности как одно из ожидаемых, и перемена аргументов местами не должна влиять на результат. Убедимся, что это действительно так:

```
lemma {:induction false} MultCommutative(x: nat, y: nat)
  ensures Mult(x, y) == Mult(y, x)
```

Продолжить доказательство можно несколькими способами. Например, рассмотрим некоторые простые и интересные случаи, оставив напоследок более длинный общий случай. Простейший случай, который следует проверить, – когда `x` и `y` равны, и не требуется никаких доказательств:

```
{
  if x == y {
```

Другой простой случай, который приходит на ум, – когда один из аргументов, скажем `x`, равен `0`. Тогда мы имеем `Mult(x, y) == x + Mult(x, y-1)` и можем завершить доказательство, выполнив индукцию по другому аргументу:

```
  } else if x == 0 {
    MultCommutative(x, y - 1);
```

Далее рассмотрим столь же простой случай `y == 0`. Его доказательство полностью симметрично тому, что мы только что привели для случая `x == 0`. Но на этот раз давайте рассмотрим не только этот частный случай, а также все случаи, когда `y` меньше `x`:

```
  } else if y < x {
    MultCommutative(y, x);
```

Обратите внимание, что в этом случае мы действительно имеем право применить индуктивное предположение, т. е. рекурсивно вызвать `MultCommutative(y, x)`, потому что Dafny использует объявление меры завершимости по умолчанию.

```
decreases x, y
```

(см. раздел 3.4), а лексикографическая пара `y,x` считается строго меньше, чем `x,y`, при условии, что `y < x`.

К настоящему моменту мы исчерпали все простые случаи, которые только смогли придумать, поэтому засучим рукава и рассмотрим более общий случай. Итак, в ветви **`else`** нашей каскадной инструкции **`if`** мы начинаем вычисление с левой стороны проверяемого условия, `Mult(x, y)`. Единственный разумный шаг в этой точке – применить определение `Mult`:

```
  } else {
    calc {
      Mult(x, y);
    ==  // определение Mult
      x + Mult(x, y - 1);
```

Поскольку член `Mult` имеет аргументы, меньшие (в порядке **`decreases`**), чем те, с которых мы начали, мы можем применить индуктивное предположение:

```
    ==  { MultCommutative(x, y - 1); }
      x + Mult(y - 1, x);
```

Здесь мы можем сделать не так много, но можем применить определение `Mult` еще раз:

```
    ==  // определение Mult
      x + y - 1 + Mult(y - 1, x - 1);
```

Похоже, это был хороший шаг, потому что мы получили ситуацию, симметричную по `x` и `y`. Исключением является сам член `Mult`, но даже его можно считать симметричным по `x` и `y`, поскольку можно применить индуктивное предположение, чтобы получить `Mult(x-1, y-1)`. Или как? Если бы здесь мы просто вызвали `MultCommutative(y-1, x-1)`, то верификатор сообщил бы, что мы неправильно показали завершимость рекурсивного вызова – лексикографическая пара `y-1,x-1` не обязательно будет меньше, чем `x,y`!

Одно из возможных решений – изменить оценочную функцию доказываемой леммы. Например, использовав

```
decreases x + y
```

можно оправдать вызов `MultCommutative(y-1, x-1)` здесь (см. упражнение 5.9). Немного подумав, нетрудно понять, что на самом деле можно перейти к следующему шагу доказательного вычисления другим способом, без изменения оценочной функции. Постусловие вызова `MultCommutative(y-1, x-1)`, который мы пытаемся выполнить, сообщает нам, что члены `Mult(y-1, x-1)` и `Mult(x-1, y-1)` равны. Поскольку все настолько симметрично, мы можем получить то же равенство, вызвав `MultCommutative(x-1, y-1)`! Давайте сделаем это:

```
    ==  { MultCommutative(x - 1, y - 1); }
      x + y - 1 + Mult(x - 1, y - 1);
```

С этого момента оставшиеся шаги более или менее совпадают с шагами, которые привели нас сюда, но в обратном порядке и с переменой местами `x` и `y`:

```
    ==  // определение Mult
      y + Mult(x - 1, y);
    ==  { MultCommutative(x - 1, y); }
      y + Mult(y, x - 1);
    == // определение Mult
      Mult(y, x);
    }
  }
}
```

Еще раз окиньте взором это доказательство. В нем есть различные интересные шаги, которые демонстрируют, что делает их доказательством.

В частности, обратите внимание, что мы можем дробить задачу доказательства по своему усмотрению, при условии, что мы можем установить цель доказательства в каждом рассматриваемом случае. В нашем доказательстве мы рассматривали случаи:

- `x == y`;
- `0 == x != y`;
- `y < x`;
- `0 < x < y`.

Кроме того, нам разрешено рекурсивно вызвать лемму с любыми параметрами при условии, что каждый вызов соответствует предусловию и уменьшает любую оценочную функцию, которую мы решили использовать с леммой. В нашем доказательстве мы использовали рекурсивные вызовы:

- `MultCommutative(x, y - 1)` в случае `0 < y`;
- `MultCommutative(y, x)` в случае `y < x`;
- `MultCommutative(x, y - 1)` снова в случае `0 < y`;
- `MultCommutative(x - 1, y - 1)` в случае `0 < x` и `0 < y`;
- `MultCommutative(x - 1, y)` в случае `0 < x`.

Обратите внимание, что условие каждого вызова гарантирует неотрицательность обоих аргументов, как того требует тип параметров **`nat`**, и что каждый вызов уменьшает лексикографическую пару `x,y`.

Упражнение 5.9

Объявите лемму `MultCommutative(x, y)` с инструкцией **`decreases`** `x + y` и напишите ее доказательство.

5.7. Пример: зеркальное отражение дерева

Свойства индуктивных типов данных хорошо подходят для индуктивных доказательств. Давайте попробуем свои силы на некоторых из них.

5.7.0. Функция Mirror – это инволюция

Вот определение типа данных, представляющего деревья, которое мы видели в разделе 4.5:

```
datatype Tree<T> = Leaf(data: T)
                 | Node(left: Tree<T>, right: Tree<T>)
```

а вот функция, создающая зеркальное отражение заданного дерева (см. упражнение 4.7):

```
function Mirror<T>(t: Tree<T>): Tree<T> {
  match t
  case Leaf(_) => t
  case Node(left, right) => Node(Mirror(right), Mirror(left))
}
```

Обратите внимание: чтобы получить зеркальное отражение дерева, недостаточно просто поменять местами левое и правое поддеревья узла. Кроме этого, необходимо также вычислить зеркальное отражение каждого поддерева.

Двойное применение `Mirror` должно вернуть первоначальное дерево. Функция, обладающая таким свойством, называется *инволюцией*. Сформулируем и докажем лемму о том, что `Mirror` действительно является инволюцией. Лемма параметризована типом полезной нагрузки дерева `T`:

```
lemma {:induction false} MirrorMirror<T>(t: Tree<T>)
  ensures Mirror(Mirror(t)) == t
{
  match t
  case Leaf(_) =>
    // простой случай
  case Node(left, right) =>
```

Чтобы доказать лемму, рассмотрим два вида возможных деревьев отдельно и напишем инструкции **match** для этих двух случаев. Поскольку тело леммы состоит из списка инструкций, а не выражений, мы фактически используем здесь *инструкцию* **match**. Она выглядит и действует так же, как выражение **match**, за исключением того, что тело каждой ветви **case** представляет собой список инструкций, а не выражение. Поэтому вполне законно (и в этом случае желательно) оставить ветвь **case** `Leaf` пустой. Это похоже на то, как мы использовали инструкции **if** в леммах, рассматривавшихся выше, когда не было смысла что-либо писать, если доказательство тривиально простое.

В ветви **case** `Node` мы получаем сообщение о том, что постусловие леммы может не выполняться. Видимо, нам нужно помочь верификатору построить доказательство. Для этого добавим доказательное вычисление:

```
  calc {
    Mirror(Mirror(Node(left, right)));
  ==  // определение Mirror (внутренний вызов)
    Mirror(Node(Mirror(right), Mirror(left)));
  ==  // определение Mirror (внешний вызов)
    Node(Mirror(Mirror(left)), Mirror(Mirror(right)));
```

Начнем вычисление с левой части проверяемого условия, поскольку она сложнее правой и, соответственно, дает больше контекста для размышлений о том, что делать на следующих шагах доказательства. Итак, левая

часть проверяемого условия – это `Mirror(Mirror(t))`, но ветвь инструкции **`match`**, над которой мы работаем сейчас, гарантирует выполнение условия `t == Node(left, right)`.

На первых двух шагах мы просто применяем определение `Mirror`: сначала к внутреннему вызову `Mirror`, который, как мы видим, применяется к узлу `Node`, благодаря чему знаем, какой случай определения `Mirror` использовать; а затем к внешнему вызову `Mirror`, который к тому времени также имеет видимый узел `Node` как аргумент. Что дальше?

Два аргумента `Node` выглядят как левая часть леммы, которую мы пытаемся доказать, поэтому будем вызывать лемму рекурсивно. Как мы видели выше, это делается не в комментарии, а внутри фигурных скобок, поскольку нам нужно сообщить об этом верификатору. С таким же успехом мы могли бы выполнить оба рекурсивных вызова в одной и той же подсказке, поэтому получаем:

```
    ==  { MirrorMirror(left); MirrorMirror(right); } // И.П.
      Node(left, right);
    }
}
```

что завершает доказательство (поскольку мы снова работаем в ветви инструкции **`match`**, где `t` – это `Node(left, right)`). Если говорить более подробно, последняя подсказка вызывает `Mirror-Mirror` в поддеревьях слева и справа (а «И.П.» в комментарии означает «индуктивное предположение»). Поскольку они структурно включены в `t`, доказательства завершимости рекурсивных принимаются верификатором. Постусловия двух вызовов тогда дают нам знание:

```
Mirror(Mirror(left)) == left && Mirror(Mirror(right)) == right
```

Эти два равенства как раз и оправдывают переписывание последней строки.

5.7.1. Mirror сохраняет количество листьев

Докажем еще одно свойство `Mirror`, использовав функцию, возвращающую количество листьев в заданном дереве:

```
function Size<T>(t: Tree<T>): nat {
  match t
  case Leaf(_) => 1
  case Node(left, right) => Size(left) + Size(right)
}
```

Мы доказываем, что `Mirror` сохраняет количество листьев в дереве. Поскольку мы уже неплохо освоились с доказательствами, я просто покажу все сразу:

```
lemma {:induction false} MirrorSize<T>(t: Tree<T>)
  ensures Size(Mirror(t)) == Size(t)
{
  match t
  case Leaf(_) =>
  case Node(left, right) =>
    calc {
      Size(Mirror(Node(left, right)));
    ==  // определение Mirror
      Size(Node(Mirror(right), Mirror(left)));
    ==  // определение Size
      Size(Mirror(right)) + Size(Mirror(left));
    ==  { MirrorSize(right); MirrorSize(left); } // И.П.
      Size(right) + Size(left);
    ==  // определение Size
      Size(Node(left, right));
    }
}
```

Доказательство имеет привычную уже форму: сначала к исследуемому выражению применяются определения различных функций, а затем, когда появляются подвыражения, похожие на доказываемую лемму, применяется индуктивное предположение (в этом примере дважды). В этом доказательстве определение функции также применяется после использования индуктивного предположения. Этот прием часто можно встретить в индуктивных доказательствах. Общая схема такова: разбить формулу на части, позволяющие применить индуктивное предположение, затем применить индуктивное предположение и, наконец, воссоздать формулу, чтобы завершить доказательство. Конечно, не все доказательства имеют такую структуру и не все настолько же просты, как представленные выше, но эта общая схема может служить хорошей отправной точкой.

Упражнение 5.10

Замените доказательное вычисление в `MirrorSize` простым вызовом индуктивного предположения. Какая версия доказательства вам нравится больше – подробная (но многословная), объясняющая каждый шаг доказательства, или краткая (но малопонятная), которая просто показывает, где применяется индуктивное предположение?

5.7.2. Варианты в доказательных вычислениях

Позвольте мне сделать еще два замечания по поводу доказательства леммы `MirrorSize`.

Во-первых, на последнем шаге мы применяем определение `Size`, беззаботно меняя порядок `left` и `right`. Мы могли бы сделать это явно, предварив последний шаг, как показано ниже:

```
    Size(right) + Size(left);
==  // операция + коммутативная
    Size(left) + Size(right);
```

Но нам этого не потребовалось, потому что Dafny обладает знаниями об арифметическом сложении и с радостью подставит это доказательство автоматически. Однако для других операторов и операций может понадобиться указать такие шаги явно.

Во-вторых, последний шаг доказательства легко по невнимательности записать так:

```
    Size(right) + Size(left);
==  // определение Size
    Size(Node(right, left));
```

Этот вариант тоже с успехом можно использовать в качестве доказательства. Но в этом случае завершающая строка в доказательном вычислении (т. е. отношение между первой и последней строками) будет иметь вид:

```
Size(Mirror(Node(left, right))) == Size(Node(right, left))
```

что не является нашей целью доказательства.

Если вы попробуете этот вариант (обязательно сделайте это!), то обнаружите, что Dafny вполне удовлетворит такое доказательство леммы. Почему? Как всегда, верификатор сам подставляет различные необходимые фрагменты доказательства. Вам не обязательно точно понимать, что и как он делает, но если вам интересно, то в рассматриваемом примере он действует, как описано ниже.

Прежде всего верификатор автоматически применяет определения функций. Точнее, он делает это один раз при каждом упоминании функции в тексте программы. В цели леммы встречается вызов `Size(t)`, поэтому верификатор будет использовать равенство:

```
Size(Node(left, right)) == Size(left) + Size(right)
```

В конце инструкции **`calc`** встречается вызов `Size(Node(right, left))`, поэтому верификатор будет использовать равенство:

```
Size(Node(right, left)) == Size(right) + Size(left)
```

Верификатор знает правила арифметики достаточно хорошо, чтобы увидеть, что правые части этих двух равенств одинаковы, и поэтому он также выведет равенство:

```
Size(Node(left, right)) == Size(Node(right, left))
```

Именно поэтому верификатор благополучно завершит доказательство леммы, даже если инструкция **`calc`** будет доказывать только возможность перестановки `left` и `right` в последней строке.

Иногда мы можем случайно указать в доказательстве действительную, но нерелевантную подсказку и при этом не получить сообщения об ошибке от верификатора. В таких случаях мы можем удивиться (если заметим оплошность), но это не помешает Dafny рассуждать правильно и попытаться выйти за рамки доказательства, предоставленного явно.

5.7.3. Итоги

В этом разделе я определил две функции для деревьев. Сформулировав и доказав леммы об этих функциях, мы можем показать, что функции обладают некоторыми ожидаемыми свойствами. Этот процесс даст нам большую уверенность в корректности определений функций.

Упражнение 5.11

Функция `ReverseColors` из упражнения 4.1 меняет цвета в сине-желтом дереве на противоположные. Сформулируйте и докажите (отключив автоматическую индукцию) лемму, которая показывает, что `ReverseColors` является инволюцией.

Упражнение 5.12

Определите функцию `LeafCount`, которая возвращает количество синих или желтых листьев в любом сине-желтом дереве из раздела 4.0. Затем докажите (отключив автоматическую индукцию), что функция `ReverseColors` из упражнения 4.1 сохраняет `LeafCount` любого заданного дерева.

Упражнение 5.13

Функция `Oceanize` из упражнения 4.3 окрашивает все узлы дерева в синий цвет. Сформулируйте и докажите (отключив автоматическую индукцию) лемму, которая показывает, что `Oceanize` *идемпотентна*, т. е. многократное применение функции равносильно ее однократному применению.

Упражнение 5.14

Докажите лемму, которая показывает, что

```
BlueCount(t) <= BlueCount(Oceanize(t))
```

для любого сине-желтого дерева `t` из раздела 4.0, где функция `BlueCount` определена в разделе 4.1, а функция `Oceanize` определена в упражнении 4.3. Перед доказательством отключите автоматическую индукцию и используйте инструкцию `calc` в варианте для `Node`.

5.8. Пример: работа с абстрактными синтаксическими деревьями

В разделе 4.6 я определил абстрактное синтаксическое дерево (AST) для простых выражений и две взаимно рекурсивные функции для вычисления выражений. Давайте определим две функции на таких AST и докажем их свойства.

```
datatype List<T> = Nil | Cons(head: T, tail: List<T>)
datatype Op = Add | Mul
datatype Expr = Const(nat)
              | Var(string)
              | Node(op: Op, args: List<Expr>)
```

5.8.0. Корректность определения подстановки

Обычной операцией в анализе программ является замена переменной в AST другим выражением. Рассмотрим одну из таких подстановок, заменяющую переменную константой. Точнее, для данного имени переменной `n` и натурального числа `c` заменим в данном выражении каждое подвыражение `Var(n)` выражением `Const(c)`. Для этого определим две взаимно рекурсивные функции:

```
function Substitute(e: Expr, n: string, c: nat): Expr
{
  match e
  case Const(_) => e
  case Var(s) => if s == n then Const(c) else e
  case Node(op, args) => Node(op, SubstituteList(args, n, c))
}

function SubstituteList(es: List<Expr>, n: string, c: nat): List<Expr>
{
  match es
  case Nil => Nil
  case Cons(e, tail) =>
    Cons(Substitute(e, n, c), SubstituteList(tail, n, c))
}
```

Функция `Substitute` никак не воздействует на константы и заменяет выражение `Var`, если используемая в нем переменная имеет имя `n`. Для обработки оператора используется функция `SubstituteList`, которая рекурсивно производит подстановку во всех аргументах выражения.

Докажем, что подстановка оказывает ожидаемый эффект на вычисление выражения. Ожидается, что подстановка константы `c` вместо переменной `n` имеет тот же эффект, что и изменение окружения на окружение, в котором `n` отображается в `c`:

```
lemma EvalSubstitute(e: Expr, n: string, c: nat, env: map<string, nat>)
  ensures Eval(Substitute(e, n, c), env) == Eval(e, env[n := c])
{
  match e
  case Const(_) =>
  case Var(_) =>
  case Node(op, args) =>
    EvalSubstituteList(args, op, n, c, env);
}
```

В формулировке этой леммы используется выражение `env[n := c]`, которое обозначает окружение, подобное `env`, за исключением того, что оно отображает `n` в `c` (тогда как `env` может отображать `n` во что-то другое или вообще может не иметь отображения для `n`).

Первые два случая имеют тривиально простые доказательства, и Dafny верифицирует их автоматически. В ветви **`case`** `Node` нам понадобится вспомогательная взаимно рекурсивная лемма, следующая базовой структуре взаимно рекурсивных функций `Eval` и `EvalList`:

```
lemma {:induction false} EvalSubstituteList(
    args: List<Expr>, op: Op, n: string, c: nat, env: map<string, nat>)
  ensures EvalList(SubstituteList(args, n, c), op, env)
      == EvalList(args, op, env[n := c])
{
  match args
  case Nil =>
  case Cons(e, tail) =>
    EvalSubstitute(e, n, c, env);
    EvalSubstituteList(tail, op, n, c, env);
}
```

Доказательство несложное. Для заданного списка `Cons(e,tail)` она (взаимно рекурсивно) вызывает лемму `EvalSubstitute` для `e` и (рекурсивно) вызывает лемму `EvalSubstituteList` для `tail`.

Упражнение 5.15

Напишите более явное доказательство `EvalSubstituteList`. В частности, замените ветвь **`case`** `Cons` инструкцией **`calc`**, начинающейся с левой части проверяемого условия леммы и заканчивающейся ее правой частью. Подсказки для вычислений должны включать вызовы двух лемм, показанных выше.

Упражнение 5.16

Покажите, что для любого ключа `n` в заданном окружении `env` подстановка константы `env[n]` вместо переменной `n` не меняет результата выражения. То есть докажите следующую лемму:

```
lemma EvalEnv(e: Expr, n: string, env: map<string, nat>)
  requires n in env.Keys
  ensures Eval(e, env) == Eval(Substitute(e, n, env[n]), env)
```

Подсказка: проведите доказательство по индукции (и с использованием вспомогательной взаимно рекурсивной леммы). Кроме того, по причинам, которые я объясню в разделе 10.2.4, вам нужно включить следующее утверждение, которое говорит, что отображение не изменяется при изменении значения, уже имеющегося в нем:

```
assert env == env[n := env[n]];
```

Упражнение 5.17

Покажите, что для любого ключа `n`, отсутствующего в данном окружении `env`, подстановка константы `0` вместо переменной `n` не меняет результата выражения. То есть докажите следующую лемму:

```
lemma EvalEnvDefault(e: Expr, n: string, env: map<string, nat>)
  requires n !in env.Keys
  ensures Eval(e, env) == Eval(Substitute(e, n, 0), env)
```

Упражнение 5.18

Докажите, что подстановка *идемпотентна*, т. е. многократное применение операции равносильно однократному ее применению. Проще говоря, докажите следующую лемму:

```
lemma SubstituteIdempotent(e: Expr, n: string, c: nat)
  ensures Substitute(Substitute(e, n, c), n, c)
      == Substitute(e, n, c)
```

5.8.1. Корректность определения оптимизации

В качестве последнего примера в этом разделе рассмотрим лемму, которая потребует от нас дополнительных усилий. В ней достаточно мест, где можно допустить мелкие ошибки, из-за которых верификатор отвергнет доказательство леммы. Как бы нам ни хотелось думать, что наши действия по программированию и верификации просты и понятны, мы всегда будем сталкиваться с задачами, требующими приложить больше усилий, чем ожидалось. Итак, перекусите и наберитесь терпения, а потом мы начнем действовать.

Определение оптимизирующего преобразования

Функция, на которой мы сосредоточимся, выполняет некоторую оптимизацию выражений. В частности, она удаляет нейтральные операции, т. е. операции, которые прибавляют `0` или умножают на `1`. Эти нейтральные операции возвращаются следующей функцией:

```
function Unit(op: Op): nat {
  match op case Add => 0 case Mul => 1
}
```

(Объявив эту функцию раньше, мы могли бы использовать ее в определении `EvalList` в разделе 4.6.) Функция оптимизации определяется следующим образом:

```
function Optimize(e: Expr): Expr {
  if e.Node? then
    var args := OptimizeAndFilter(e.args, Unit(e.op));
    Shorten(e.op, args)
  else
    e // вернуть без изменений
}
```

Мы оптимизируем только выражения `Node`, поэтому я решил использовать инструкцию **`if`** вместо **`match`**. План оптимизации узла заключается в оптимизации каждого из его аргументов и удалении любых аргументов, соответствующих нейтральной операции. Это достигается вызовом `OptimizeAndFilter`, результат которого помещается в локальную переменную `args`. Мы могли бы просто вернуть `Node(e.op, args)`, но давайте рассмотрим еще одну оптимизацию: если у нас не останется аргументов, мы просто вернем нейтральную операцию, а если останется ровно один аргумент, то просто вернем его. Дальнейшая оптимизация выполняется в функции `Shorten`.

Функцию `Shorten` проще всего реализовать с помощью **`match`** с тремя случаями:

```
function Shorten(op: Op, args: List<Expr>): Expr {
  match args
  case Nil => Const(Unit(op))
  case Cons(e, Nil) => e
  case _ => Node(op, args)
}
```

Первая ветвь **`case`** соответствует списку `Nil`, т. е. списку длиной `0`. Вторая ветвь **`case`** соответствует операции `Cons`, второй аргумент которой равен `Nil` – списку длиной `1`. В третьей ветви **`case`** используется подчеркивание, она «перехватывает все». Поскольку ветви рассматриваются по порядку, последняя из них будет соответствовать спискам длиной `2` или больше.

Последняя функция в нашей реализации оптимизации – `OptimizeAndFilter`:

```
function OptimizeAndFilter(es: List<Expr>, unit: nat): List<Expr>
{
  match es
  case Nil => Nil
```

```
    case Cons(e, tail) =>
      var e', tail' := Optimize(e), OptimizeAndFilter(tail, unit);
      if e' == Const(unit) then tail' else Cons(e', tail')
  }
```

Получив непустой список аргументов, она оптимизирует голову списка и рекурсивно – хвост списка, помещая соответствующие результаты в локальные переменные `e'` и `tail'`. Если `e'` обозначает нейтральную операцию, то возвращается только оптимизированный хвост; в противном случае `e'` возвращается вместе с оптимизированным хвостом.

Упражнение 5.19

Найдите рекурсивные и взаимно рекурсивные вызовы в определениях `Optimize` и `OptimizeAndFilter` и покажите, почему эти вызовы завершимые.

Корректность Optimize

Далее следуют типичные действия: мы сформулируем и докажем по одной лемме для каждой из трех функций, начав с леммы, утверждающей корректность функции `Optimize`:

```
lemma OptimizeCorrect(e: Expr, env: map<string, nat>)
  ensures Eval(Optimize(e), env) == Eval(e, env)
```

Здесь нет никаких сюрпризов. Мы говорим, что `e` и `Optimize(e)` дают одно и то же значение для любого окружения. Для доказательства используем оператор **`if`**, имитирующий структуру функции `Optimize`. Затем приступим к вычислениям, начиная с более сложной стороны проверяемого условия. На первом шаге применяется определение `Optimize`, поэтому будет удобно дать имя длинному подвыражению, обозначающему вызов `OptimizeAndFilter`. Для этого введем локальную переменную, фактически следуя определению функции `Optimize`. В результате мы имеем:

```
  if e.Node? {
    var args := OptimizeAndFilter(e.args, Unit(e.op));
    calc {
      Eval(Optimize(e), env);
    ==  // определение Optimize
      Eval(Shorten(e.op, args), env);
```

Здесь мы применим лемму о корректности, которую сформулируем для `Shorten`. Как я уже упоминал выше, `Shorten(e.op, args)` должна быть семантически эквивалентна `Node(e.op, args)`, поэтому объявим такую лемму:

```
lemma ShortenCorrect(op: Op, args: List<Expr>, env: map<string, nat>)
  ensures Eval(Shorten(op, args), env) == Eval(Node(op, args), env)
```

Врезка 5.2

Разрабатывая это доказательство в IDE, сразу же добавьте закрывающие фигурные скобки в OptimizeCorrect, чтобы Dafny сообщал не о синтаксических ошибках, а об ошибках проверки типов и верификации. Лемму ShortenCorrect можно объявить в любом месте в файле, но естественнее поместить ее после OptimizeCorrect. Объявив эту лемму, вы сможете использовать ее, а верификатор проверит корректность ее использования

Следующий шаг в доказательном вычислении в `OptimizeCorrect`:

```
    ==  { ShortenCorrect(e.op, args, env); }
      Eval(Node(e.op, args), env);
```

Разумеется, следующий шаг будет связан с преобразованием, выполняемым функцией `OptimizeAndFilter` (напомню, что мы определили `args` как результат вызова `OptimizeAndFilter`). Мы знаем, где хотели бы оказаться, а именно в правой части проверяемого условия. Если нам повезет, мы окажемся всего в одном шаге от нее и сможем разработать лемму о корректности `OptimizeAndFilter`, исходя из того, что нам нужно. Итак, завершаем доказательство `OptimizeCorrect` следующим образом:

```
    ==  { OptimizeAndFilterCorrect(e.args, e.op, env); }
      Eval(Node(e.op, e.args), env);
    }
  }
}
```

что предполагает следующее определение леммы `OptimizeAndFilterCorrect`:

```
lemma OptimizeAndFilterCorrect(args: List<Expr>, op: Op,
                               env: map<string, nat>)
  ensures Eval(Node(op, OptimizeAndFilter(args, Unit(op))), env)
      == Eval(Node(op, args), env)
```

Прежде чем начать доказательство `ShortenCorrect`, давайте убедимся, что все, что мы написали (анализ, проверка типов и т. д.), успешно верифицируется. Если мы допустили ошибки в рассуждениях, при вводе функций и лемм или неоправданно уверовали в способность верификатора вывести недостающие доказательства, то именно сейчас самое время обратить на это внимание и внести необходимые исправления.

Упражнение 5.20

С учетом инструкций **`decreases`** по умолчанию в трех леммах, которые мы определили, запишите условия завершимости в теле `OptimizeCorrect`. Выполняются ли эти условия?

Корректность Shorten

Функция `Shorten` кажется очень простой – настолько простой, что, по нашему мнению, верификатор справится с ней автоматически. Но, как ни странно, этого не произошло. Иногда бывает трудно предсказать, когда доказательство может быть найдено автоматически, поэтому давайте попрактикуемся в отладке верификации.

Наша первая полная оптимизма попытка доказать `ShortenCorrect` – предоставить пустое тело:

```
{
}
```

Увы, верификатор сообщает, что постусловие леммы может не выполняться. Возможно, мы пытаемся доказать некорректную лемму, а может быть, верификатор просто не в состоянии построить доказательство автоматически. Чтобы пролить больше света на ситуацию, начнем писать доказательство вручную. Как обычно, структура доказательства будет следовать структуре определения `Shorten`:

```
match args
case Nil =>
case Cons(a, Nil) => // ошибка: постусловие может не выполняться
                     // в этом пути
case _ =>
```

Мы можем сказать, что *кое-что* сделали правильно, потому что жалоба на постусловие, которую мы получили, указывает на вторую ветвь **case**. Отсюда следует вывод, что верификатор нашел доказательство для первой ветви **case**. Если верификатор не может что-то доказать (в нашей ситуации это – постусловие), он выделяет только одну трассу, ведущую к ошибке. Итак, мы знаем, что первый случай доказан, а второй нет, и пока мы ничего не знаем о третьем случае.

Давайте закончим доказательство `ShortenCorrect`. В ветви **case** `Cons(a, Nil)` мы начинаем доказательное вычисление и развертываем определения функций как обычно:

```
calc {
  Eval(Node(op, Cons(a, Nil)), env);
==  // определение Eval
  EvalList(Cons(a, Nil), op, env);
==  // определение EvalList
  var v0, v1 := Eval(a, env), EvalList(Nil, op, env);
  match op
  case Add => v0 + v1
  case Mul => v0 * v1;
==  // определение EvalList
```

```
  var v0, v1 := Eval(a, env), Unit(op);
  match op
  case Add => v0 + v1
  case Mul => v0 * v1;
```

Выглядит немного не так, как раньше, но только потому, что формулы стали длиннее и включают в let-выражение и выражение **match**. Помните, что синтаксически каждое выражение в вычислении заканчивается точкой с запятой, которую не следует путать с точкой с запятой, являющейся частью синтаксиса let-выражений.

Мы почти закончили:

```
==  // подстановка для v0, v1
  match op
  case Add => Eval(a, env) + Unit(op)
  case Mul => Eval(a, env) * Unit(op);
==  // определение Unit в каждом случае
  Eval(a, env);
}
```

На этом мы завершаем доказательство леммы `ShortenCorrect`. Это была первая лемма в книге, которую верификатор не смог доказать автоматически (в данном случае потому, что он не разворачивает `EvalList` автоматически дважды, как того требует доказательство), плюс к тому же индукция отключена.

Корректность OptimizeAndFilter

Теперь обратимся к нашей последней лемме, `OptimizeAndFilterCorrect`, и посмотрим, позволит ли нам наше понимание индукции, которое мы уже неплохо развили, выяснить, как будет выглядеть доказательство, не погружаясь в механические вычисления. Целью доказательства нашей леммы является функция `OptimizeAndFilter`, поэтому рассмотрим ее определение внимательнее. Мы видим совпадение по первому аргументу, представляющему список. Затем следуют взаимно рекурсивный вызов `Optimize`, рекурсивный вызов `OptimizeAndFilter` и выражение **if-then-else**, которое кажется довольно простым. Исходя из этого можно догадаться, что доказательство должно включать вызов лемм корректности для `Optimize` и `OptimizeAndFilter`, параметризованных подобно вызовам этих функций в `OptimizeAndFilter`. Это приводит нас к следующей попытке доказательства:

```
{
  match args
  case Nil =>
  case Cons(e, tail) =>
    OptimizeCorrect(e, env);
    OptimizeAndFilterCorrect(tail, op, env);
}
```

Вот и все! Верификатор принимает наше доказательство. Если бы мы ошиблись в своих догадках, то могли бы прибегнуть к более подробным доказательным вычислениям, подобным тем, что мы видели раньше.

Краткие итоги примера оптимизации

В этом примере использовался тип данных `Expr`, рекурсивный относительно типа данных `List`, взаимно рекурсивные функции `Eval` и `EvalList`, взаимно рекурсивные функции `Optimize` и `OptimizeAndFilter`, а также вспомогательные функции `Unit` и `Shorten`. С ростом числа задействованных функций увеличивается и объем доказательств. Организация лемм и их доказательство имеет тенденцию следовать структуре задействованных функций. Например, мы видели, что лемма `OptimizeCorrect` вызывает `ShortenCorrect` и `OptimizeAndFilterCorrect` точно так же, как функция `Optimize` вызывает `Shorten` и `OptimizeAndFilter`.

Упражнение 5.21

Напишите еще одну функцию оптимизации `PartiallyEvaluate`, объединяющую все константные аргументы `Node` в один. Докажите, что это преобразование не меняет сути выражения.

5.9. Итоги

В этой главе мы рассмотрели несколько лемм и их доказательства.

Лемма в Dafny – это просто призрачный метод, имеющий пред- и постусловие. Предусловие является посылкой леммы, и Dafny позволяет использовать лемму, только когда предусловие выполняется. Постусловие является заключением леммы. Это цель доказательства в теле леммы, и код, вызывающий лемму, предполагает выполнение постусловия после возврата из вызова леммы.

Леммы могут вызывать себя рекурсивно. Это добавляет необходимость доказать завершимость рекурсии. Рекурсивный вызов леммы соответствует тому, что логики называют обращением к индуктивному предположению. Нам не нужно объяснять Dafny, что такое индуктивное предположение, потому что это всего лишь то, что утверждается в пред- и постусловиях лемм, которые мы вызываем рекурсивно.

Мы видели, что Dafny способен автоматически добавлять некоторые фрагменты доказательств. Мы не позволили себе положиться на эту его способность, тем не менее Dafny также старается найти способы автоматического рекурсивного вызова лемм, что, по сути, избавляет нас от явного использования индуктивного предположения. Однако Dafny не вызывает автоматически *другие* леммы, поэтому мы должны явно вызывать все вспомогательные и взаимно рекурсивные леммы.

Если лемма касается функции и эта функция определяет несколько случаев, то есть вероятность, что доказательство леммы разделит поток управления и на эти случаи.

Всякий раз, когда нам требовалось ввести в доказательство дополнительные детали с целью отладки верификации, нашим основным инструментом была инструкция **calc**. Она позволяет привести доказательные расчеты и обосновать каждый шаг подсказкой. В большинстве случаев, которые мы видели, эти доказательные вычисления имеют форму преобразования, сохраняющего равенство, но мы также видели несколько случаев использования операторов, отличных от равенства.

К настоящему моменту мы рассмотрели основы, формальные определения, свойство завершимости, типы данных и леммы и исследовали основные элементы программ и доказательств их корректности в Dafny. Мы также освоили взаимодействие с верификатором. Основная идея заключается в том, что если верификатор сообщает об ошибке, то мы должны предоставить дополнительные доказательства. Этот процесс приводит к тому, что либо верификатор примет доказательства, либо мы поймем, где допустили оплошность.

Примечания

Основной формат доказательства, который я использую в этой книге, – вычисления с подсказками между шагами, – был придуман Вимом Фейеном (Wim H. J. Feijen). Он используется во многих книгах (например, [41, 18, 11, 56, 9, 52], и это лишь некоторые из них). Другой распространенный формат вычислений предусматривает размещение подсказок в правом поле. Однако это не единственная форма структурирования доказательств (см., например, [72, 125]), а также не единственная форма доказательств, поддерживаемая в Dafny.

Доказательные вычисления с использованием инструкций **calc** [85] также поддерживаются в F* [53], LiquidHaskell [123] и Lean [75]. Язык доказательств Isar для Isabelle [102] имеет богатую систему обозначений, позволяя создавать удобочитаемые доказательства, включая доказательные вычисления [15]. Стандартная библиотека Agda [22] тоже предлагает естественный способ написания доказательных вычислений.

Освоив инструкцию **calc** в Dafny, вы можете заметить, что часто приходится повторять один и тот же оператор между строками. На самом деле Dafny позволяет опустить оператор между двумя строками и по умолчанию будет интерпретировать отсутствующий оператор как ==. С учетом этого обстоятельства первое доказательное вычисление в разделе 5.4 можно записать так:

```
calc {
  5 * (x + 3);
  5 * x + 5 * 3;
  5 * x + 15;
}
```

Если строки потребуется объединить с помощью других операторов, то вы сможете добавлять их по мере необходимости. Например, вычисление с использованием `3*(x+n)` в разделе 5.4 можно записать так:

```
calc {
  3*x + n + n;
  3*x + 2*n;
<=
  3*x + 3*n;
  3*(x + n);
}
```

Более того, при желании можно даже назначить оператором по умолчанию другой оператор, отличный от `==`. Его нужно указать сразу после ключевого слова **`calc`**. Например, те же вычисления, включающие `3*(x+n)`, можно записать как

```
calc <= {  // назначить <= оператором по умолчанию
  3*x + n + n;
==
  3*x + 2*n;
  3*x + 3*n;
==
  3*(x + n);
}
```

Возможность опускать оператор – желанная особенность для опытных пользователей, но из-за ее применения вычисления выглядят загадочными для новичков. В этой книге я продолжу использовать операторы в доказательствах.

При доказательстве леммы верификатор Dafny предпринимает попытку прибегнуть к индуктивному предположению, если необходимо. Эта скромная поддержка автоматической индукции удивительно эффективна [77]. Другие верификаторы уже давно поддерживают расширенные эвристики для автоматической индукции, в частности ACL2 [71] и его предшественники – средства доказательства теорем Бойера–Мура [23]. За более подробной информацией я отсылаю вас к историческому описанию автоматизированной индукции [91].

Часть I

Функциональные программы

Программы, написанные в функциональном стиле, структурируются как математические функции, возвращающие значения. Данные не изменяются и в большинстве своем организуются в структуры, определяемые индуктивными типами данных.

В этой части книги рассматриваются функциональные программы и рассказывается, как рассуждать о таких программах. Мы уже видели основные элементы функциональных программ в предыдущих главах. Теперь посмотрим, как создаются и верифицируются такие программы.

Я представлю понятия внутренних и внешних спецификаций, инвариантов структур данных, абстрактных функций и модулей.

Глава 6

Списки

Списки – наиболее распространенная структура данных в функциональных программах. Поэтому вполне оправданно посвятить первую главу этой части программам, обрабатывающим списки.

В главе 5 мы учились писать доказательства, отключив автоматическую индукцию в Dafny. Но с этого момента мы вновь позволим Dafny применять автоматическую индукцию. Это не избавит нас от необходимости писать доказательства вручную, но сократит объем работы.

6.0. Определение списка

В этой главе я буду использовать следующее определение типа данных списка:

```
datatype List<T> = Nil | Cons(head: T, tail: List<T>)
```

Значение `Nil` представляет пустой список, а `Cons(x, xs)` – список, начинающийся с элемента `x` и продолжающийся элементами из списка `xs`. Тип `List` параметризуется типом `T` – типом каждого элемента в списке.

Вот четыре примера списков. Слева показано, как можно записать каждый список на бумаге, в середине – соответствующее выражение типа `List`, а справа – тип значения списка.

Неформальное определение	Выражение List	
2, 5, 3	Cons(2, Cons(5, Cons(3, Nil)))	List<nat>
true, true	Cons(true, Cons(true, Nil))	List<bool
	Nil	List<int>
"gimmie", "lists"	Cons(«gimmie», Cons("lists", Nil))	

Элемент `Nil` может иметь тип `List<T>` для любого типа `T`. Обратите внимание, что все элементы в списке имеют один и тот же тип, т. е. нельзя создать такой список:

```
Cons(true, Cons(3, Nil)) // ошибка: все элементы должны быть одного типа
```

6.1. Length

Вот функция, возвращающая длину списка:

```
function Length<T>(xs: List<T>): nat {
  match xs
  case Nil => 0
  case Cons(_, tail) => 1 + Length(tail)
}
```

Она говорит, что пустой список имеет длину `0`, а длина любого другого списка на `1` длиннее его хвоста.

Давайте вспомним кое-что, о чем мы узнали в предыдущих главах.

Во-первых, `Length` возвращает результат типа **`nat`** – натуральное число, т. е. неотрицательное целое число.

Во-вторых, в определении `Length` используется выражение **`match`**, различающее два варианта списков. Вместо **`match`** также можно было бы использовать выражение **`if-then-else`**:

```
if xs == Nil then 0 else 1 + Length(xs.tail)
```

Выбор между управляющими структурами, такими как **`match`** и **`if`**, – это лишь вопрос вкуса, но при наличии у типа большого количества конструкторов выражение **`match`** выглядит предпочтительнее. Обратите внимание, что `tail` в первом определении – это связанная переменная, введенная в ветви **`case`** в **`match`**, тогда как во втором определении `tail` – это имя деструктора, возвращающего второй компонент `Cons`, как определено в объявлении типа данных `List`.

Упражнение 6.0

Определите `Length`, как указано выше, с помощью **`match`**, и функцию `Length'` с помощью **`if`** (но с рекурсивным вызовом `Length'`, а не `Length`).

Объявите и докажите лемму, утверждающую, что `Length'` и `Length` всегда возвращают одно и то же значение.

Конструктор `Cons` создает список, помещая элемент в начало другого списка. Иногда бывает нужно прямо противоположное – создать список, поместив элемент в конец другого списка. Эту операцию обычно называют `Snoc`. Поскольку доступ к индуктивным спискам осуществляется от начала к концу, `Snoc` требует больше вычислительных затрат, чем `Cons`. Вот ее определение:

```
function Snoc<T>(xs: List<T>, y: T): List<T> {
  match xs
  case Nil => Cons(y, Nil)
  case Cons(x, tail) => Cons(x, Snoc(tail, y))
}
```

Чтобы убедиться, что функция определена корректно, всегда полезно проверить, есть ли у нее некоторые ожидаемые свойства. Это можно сделать, сформулировав и доказав леммы. Например, мы ожидаем, что `Snoc(xs, x)` создаст список, длина которого будет на `1` больше длины `xs`:

```
lemma LengthSnoc<T>(xs: List<T>, x: T)
  ensures Length(Snoc(xs, x)) == Length(xs) + 1
{
}
```

Упражнение 6.1

Отключите автоматическую индукцию и напишите доказательство для `LengthSnoc`.

6.2. Внутренние и внешние характеристики

Мы подошли к важным различиям между двумя стилями написания спецификаций.

Давайте определим функцию `Append`, возвращающую конкатенацию двух списков:

```
function Append<T>(xs: List<T>, ys: List<T>): List<T> {
  match xs
  case Nil => ys
  case Cons(x, tail) => Cons(x, Append(tail, ys))
}
```

Между `Append` и `Length` существует связь. Зафиксируем ее в лемме, которая доказывается автоматически:

```
lemma LengthAppend<T>(xs: List<T>, ys: List<T>)
```

```
  ensures Length(Append(xs, ys)) == Length(xs) + Length(ys)
{
}
```

Как мы видели ранее, вводя такие леммы, мы обычно формулируем и доказываем свойства функций. Свойство, указанное в лемме, считается *внешним* по отношению к определению функции `Append`, потому что лемма не является частью самого объявления функции. Это не единственный способ сформулировать и доказать свойства функций. Другой способ – определить функцию с постусловием, как мы это делаем для методов. Свойство, заданное в постусловии функции, называется *внутренним*, потому что оно является частью объявления функции и проверяется в рамках проверки корректности функции.

Вот альтернативное определение `Append`, которое, по сути, определяет свойство, которое лемма `LengthAppend` считает внешним:

```
function Append<T>(xs: List<T>, ys: List<T>): List<T>
  ensures Length(Append(xs, ys)) == Length(xs) + Length(ys)
{
  match xs
  case Nil => ys
  case Cons(x, tail) => Cons(x, Append(tail, ys))
}
```

Для ссылки на значение, возвращаемое функцией, постусловие просто упоминает вызываемую функцию с ее параметрами, а именно `Append(xs, ys)`.

Методы в Dafny всегда непрозрачны, поэтому известны только те их свойства, которые указаны в спецификации метода. Другими словами, свойства методов всегда являются внутренними и никогда внешними. Функции, напротив, прозрачны, поэтому у нас есть возможность задавать их внутренние или внешние свойства.

Одно из преимуществ встроенных спецификаций – они используются при каждом применении функции. Внешнюю лемму, напротив, необходимо применять явно, чтобы использовать ее свойство во время верификации. Автоматическое использование спецификаций может показаться преимуществом, но это не так. Когда проверяемые условия становятся большими, наличие слишком большого количества информации может вызвать перегрузку верификатора, увеличить продолжительность верификации или даже ошибку (из-за исчерпания некоторых внутренних ресурсов). Поэтому для функций обычно принято объявлять и доказывать внешние свойства.

Иногда записать свойство как внутреннее просто невозможно, например когда свойство ссылается на функцию несколько раз. Так, к примеру, невозможно объявить свойство `Mirror(Mirror(t)) == t` в постусловии функции `Mirror` (раздел 5.7). Причина в том, что внешний вызов `Mirror` является рекурсивным, поэтому нужно доказать, что ее аргумент, а именно `Mirror(t)`, меньше `t`. Это невозможно для всех `t`: рассмотрим некоторые конкрет-

ные t и t', где t' == Mirror(t), означающее, что t == Mirror(t'). Мы, конечно же, не можем утверждать, что одновременно t меньше t' и t' меньше t. Другими примерами свойств, не подходящих для внутренних спецификаций, являются коммутативность и транзитивность.

Единственная ситуация, когда внутренние спецификации имеют смысл и могут быть рекомендованы, – если свойство будет представлять интерес для всех клиентов функции. В этой главе будет удобно рассматривать длину Append как подпадающую под эту категорию, поэтому я буду считать, что Append определена с постусловием:

```
Length(Append(xs, ys)) == Length(xs) + Length(ys)
```

Упражнение 6.2

Докажите, что Snoc – это частный случай Append, т. е. что

```
Snoc(xs, y) == Append(xs, Cons(y, Nil))
```

6.2.0. Другие свойства Append

Часто бывает полезно знать различные алгебраические свойства определяемых нами функций. Например, рассмотрим бинарную функцию $\oplus$, которую здесь я использую как инфиксный оператор. Значение L называется *левым нейтральным элементом* $\oplus$, если для всех x, $L \oplus x = x$, а значение R называется *правым нейтральным элементом* $\oplus$, если для всех x, $x \oplus R = x$.

По определению Append мы сразу видим, что Nil – левый нейтральный элемент, т. е. Append(Nil, x) == x. Мы также можем доказать, что Nil является правым нейтральным элементом Append:

```
lemma AppendNil<T>(xs: List<T>)
  ensures Append(xs, Nil) == xs
{
}
```

Еще одно полезное алгебраическое свойство функции $\oplus$ – *ассоциативность*. Согласно этому свойству, не имеет значения, как вы расставите круглые скобки. Для любых x, y и z:

$(x \oplus y) \oplus z = x \oplus (y \oplus z)$.

Функция Append ассоциативна:

```
lemma AppendAssociative<T>(xs: List<T>, ys: List<T>, zs: List<T>)

  ensures Append(Append(xs, ys), zs) == Append(xs, Append(ys, zs))
{
}
```

Упражнение 6.3

Добавьте в определение `AppendNil` атрибут `{:induction false}` и приведите ручное доказательство леммы.

Упражнение 6.4

Добавьте в определение `AppendAssociative` атрибут `{:induction false}` и приведите ручное доказательство леммы.

Упражнение 6.5

Выше мы увидели одну связь между `Append` и `Length` и решили определить ее как внутреннюю. Вот еще одно свойство этих двух функций, которое лучше всего определить как внешнее:

```
lemma AppendDecomposition<T>(a: List<T>, b: List<T>,
                             c: List<T>, d: List<T>)
  requires Length(a) == Length(c)
  requires Append(a, b) == Append(c, d)
  ensures a == c && b == d
```

Докажите эту лемму с помощью автоматической индукции и без нее.

Упражнение 6.6

Для целочисленного оператора с левым нейтральным элементом `L` и правым нейтральным элементом `R` докажите, что `L` и `R` равны. В Dafny это можно сделать так:

```
function F(x: int, y: int): int

const L: int
const R: int

lemma LeftUnit(x: int)
  ensures F(L, x) == x

lemma RightUnit(x: int)
  ensures F(x, R) == x
```

Здесь функция без тела `F(x, y)` обозначает произвольную операцию (которую выше я записал как $x \oplus y$). Я объявил имена `L` и `R` как произвольные константы, но их также можно было бы объявить как нульарные функции без тела. Две леммы, определяющие свойства `F`, `L` и `R`, даны как аксиомы, поэтому используйте их, но не пытайтесь их доказать. Сформулируйте и докажите лемму о том, что `L == R`.

Упражнение 6.7

Значение L называется *левым нулевым элементом* оператора $\oplus$, если для всех x, $L \oplus x = L$, а значение R называется *правым нулевым элементом*, если для всех x, $x \oplus R = R$. Если оператор имеет левый нулевой элемент L и правый нулевой элемент R, то эти два элемента равны. Сформулируйте и докажите это свойство для произвольного целочисленного оператора. Подсказка: см. упражнение 6.6, как это сделать.

6.3. Take и Drop

Функция `Append` объединяет два списка в один. Функции `Take` и `Drop` делают обратное: они разбивают список на две части.

```
function Take<T>(xs: List<T>, n: nat): List<T>
  requires n <= Length(xs)
{
  if n == 0 then Nil else Cons(xs.head, Take(xs.tail, n - 1))
}

function Drop<T>(xs: List<T>, n: nat): List<T>
  requires n <= Length(xs)
{
  if n == 0 then xs else Drop(xs.tail, n - 1)
}
```

Определенные с предусловием `n <= Length(xs)`, эти функции довольно требовательны к количеству элементов, которые можно извлечь (`Take`) или отбросить (`Drop`). Предусловие позволяет получить доступ к `xs.head` и `xs.tail` внутри функций (для чего требуется `xs.Cons?`) в ветви, где `n != 0`.

Упражнение 6.8

Напишите более свободные версии `Take` и `Drop` без предварительных условий. При попытке извлечь больше элементов, чем длина списка, свободная версия `Take` должна вернуть весь список. При попытке удалить больше элементов, чем длина списка, свободная версия `Drop` должна вернуть пустой список. Докажите, что если `n <= Length(xs)`, то свободные версии возвращают то же значение, что и соответствующие строгие версии, приведенные выше.

Упражнение 6.9

При выключенной автоматической индукции докажите лемму из упражнения 6.8. Легко доказать соответствие между `Append`, `Take` и `Drop`:

```
lemma AppendTakeDrop<T>(xs: List<T>, n: nat)
  requires n <= Length(xs)
  ensures Append(Take(xs, n), Drop(xs, n)) == xs
{
}

lemma TakeDropAppend<T>(xs: List<T>, ys: List<T>)
  ensures Take(Append(xs, ys), Length(xs)) == xs
  ensures Drop(Append(xs, ys), Length(xs)) == ys
{
}
```

Доказательство леммы `TakeDropAppend` использует внутреннее свойство, заданное постусловием функции `Append`.

Упражнение 6.10

Отключите автоматическую индукцию и докажите лемму `AppendTakeDrop`.

Упражнение 6.11

Отключите автоматическую индукцию и докажите лемму `TakeDropAppend`.

6.4. At

Элементы списка упорядочены, что позволяет обращаться к ним по их индексам.

```
function At<T>(xs: List<T>, i: nat): T
  requires i < Length(xs)
{
  if i == 0 then xs.head else At(xs.tail, i - 1)
}
```

Предусловие `At` требует, чтобы аргумент `i` был правильным индексом в списке `xs`. Обратите внимание, что это подразумевает `xs.Cons?` и тем самым оправдывает использование деструкторов `xs.head` и `xs.tail` в теле.

Элемент в позиции `i` в списке `xs` – это элемент, которому предшествует `i` элементов. Итак, этот элемент является головой списка, который получается после удаления `i` первых элементов из `xs`. Попробуем сформулировать это свойство в виде леммы:

```
lemma AtDropHead<T>(xs: List<T>, i: nat)
  requires i < Length(xs)
  ensures At(xs, i) == Drop(xs, i).head // ошибка: .head
                                        // требует Cons?
```

Предусловие для леммы совершенно необходимо, потому что для рассуждений о `At(xs, i)` необходимо выполнение условия `i < Length(xs)`. Но попытка сформулировать лемму в таком виде, как я показал, приводит к сообщению об ошибке из-за попытки получить доступ к элементу `head` списка, не имеющему его. Чтобы значение `Drop(xs, i).head` было корректно определено, нам нужно гарантировать, что `Drop(xs, i)` возвращает вариант списка `Cons` (т. е. список, отличный от `Nil`). Разрешить эту ситуацию можно несколькими способами. Самый простой из уместных здесь – включить еще и `Drop(xs, i).Cons?` как проверяемое условие леммы. Это дает нам такое определение:

```
lemma AtDropHead<T>(xs: List<T>, i: nat)
  requires i < Length(xs)
  ensures Drop(xs, i).Cons? && At(xs, i) == Drop(xs, i).head
{
}
```

которое обрабатывается автоматической индукцией Dafny.

Также давайте свяжем `At` и `Append` в лемме:

```
lemma AtAppend<T>(xs: List<T>, ys: List<T>, i: nat)
  requires i < Length(Append(xs, ys))
  ensures At(Append(xs, ys), i)
       == if i < Length(xs) then
            At(xs, i)
          else
            At(ys, i - Length(xs))
{
}
```

Упражнение 6.12

Отключите автоматическую индукцию и напишите доказательство для `AtAppend`. Подсказка: используйте доказательство в форме:

```
match xs
case Nil =>
case Cons(x, tail) =>
  if i == 0 {
    // вычисления...
  } else {
    // вычисления...
  }
```

(Вы довольны автоматической индукцией?)

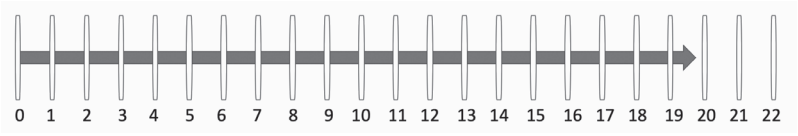
Врезка 6.0

Когда мы говорим о том, сколько шагов мы сделали в «путешествии», то понимаем, что означают цифры. Например, измеряя пульс, вы отмеряете 6-секундный интервал и ведете счет «ноль, один, два, ...»

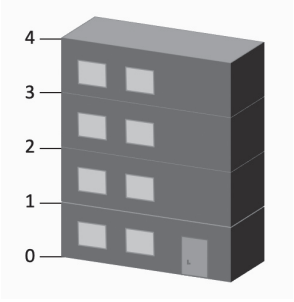
а затем умножаете на 10 последнее сказанное вслух число, чтобы получить количество ударов сердца в минуту.

Но что, если мы решим пронумеровать сами шаги (или, в данном случае, сердцебиения)? Для этого существует несколько соглашений.

Когда вам 19 лет и кто-то спрашивает ваш возраст, то вы говорите «19». В течение года сразу после рождения вы проживаете «нулевой год».



Если вы живете в квартире на 2 этажа выше первого (уровня улицы), то в некоторых странах (например, в большинстве стран Европы) вам придется нажать на кнопку «2» в лифте, чтобы подняться на свой этаж, тогда как в других странах (например, США) вы должны нажать «3».



Здесь и далее нумерацию элементов списка, последовательности или массива мы будем вести с 0, как, например, пульс, возраст и этаж в некоторых странах. Таким образом, функция `At(xs, i)` имеет предусловие `0 <= i < Length(xs)`.

6.5. Find

Определим функцию `Find`, возвращающую позицию элемента в списке. Если элемент встречается несколько раз, то возвращается индекс первого встретившегося элемента. Если искомый элемент отсутствует в списке, то возвращается длина списка. Иначе говоря, `Find(xs, x)` возвращает длину самого длинного префикса `xs`, не содержащего `x`.

```
function Find<T(==)>(xs: List<T>, y: T): nat
  ensures Find(xs, y) <= Length(xs)
{
  match xs
  case Nil => 0
  case Cons(x, tail) =>
    if x == y then 0 else 1 + Find(tail, y)
}
```

Все наши предыдущие операции были параметризованы типом элемента, но функция `Find` имеет смысл, только если тип элемента – это тип значений, которые можно сравнивать, т. е. тип, для которого определено понятие равенства. В Dafny все типы поддерживают операцию сравнения в призрачном контексте, поэтому мы можем написать `Find` как призрачную функцию. Но не все типы поддерживают сравнение в скомпилированном контексте. Например, нельзя сравнить две функции в скомпилированном коде, поскольку для вычисления может потребоваться бесконечное время. Если нужно, чтобы `Find` была компилируемой функцией, мы должны ограничить ее параметрами типа, поддерживающими скомпилированную версию сравнения. Это делается путем добавления суффикса `(==)` к параметру типа, как я сделал выше (подробнее я расскажу об этом в разделе 9.4).

Упражнение 6.13

Удалите суффикс `(==)` из параметра типа в `Find`. Какое сообщение об ошибке вы получите?

`Find(xs, x)` всегда возвращает значение от `0` до `Length(xs)` включительно. Это ее свойство понадобится большинству клиентов `Find`. Поэтому мы объявляем это свойство внутренним: нижнюю границу устанавливаем с помощью типа результата `nat`, а верхнюю границу – объявлением **`ensures`**.

Для различных функций списков, которые мы видели до сих пор, можно доказать многие свойства. Вот лишь четыре примера, где `xs` и `ys` имеют тип `List<T>` для некоторого типа `T`, `x` имеет тип `T`, а `i` имеет тип `nat`:

```
lemma AtFind<T>(xs: List<T>, y: T)
  ensures Find(xs, y) == Length(xs) || At(xs, Find(xs, y)) == y

lemma BeforeFind<T>(xs: List<T>, y: T, i: nat)
```

```
  ensures i < Find(xs, y) ==> At(xs, i) != y

lemma FindAppend<T>(xs: List<T>, ys: List<T>, y: T)
  ensures Find(xs, y) == Length(xs)
       || Find(Append(xs, ys), y) == Find(xs, y)

lemma FindDrop<T>(xs: List<T>, y: T, i: nat)
  ensures i <= Find(xs, y) ==> Find(xs, y) == Find(Drop(xs, i), y) + i
```

Упражнение 6.14

Проверяемым условием в леммах `AtFind` и `BeforeFind` является вызов функции `At`, которая имеет предусловие. Объясните, как выполняется предусловие для каждого из этих вызовов.

Упражнение 6.15

Докажите лемму `AtFind` с автоматической индукцией и без нее.

Упражнение 6.16

Докажите лемму `BeforeFind` с автоматической индукцией и без нее.

Упражнение 6.17

Докажите лемму `FindAppend` с автоматической индукцией и без нее.

Упражнение 6.18

Докажите лемму `FindDrop` с автоматической индукцией и без нее.

6.6. Перестановка элементов списка в обратном порядке

В качестве последней операции над списком рассмотрим перестановку элементов списка в обратном порядке. Вот одна из возможных функций, выполняющих эту операцию:

```
function SlowReverse<T>(xs: List<T>): List<T> {
  match xs
  case Nil => Nil
  case Cons(x, tail) => Snoc(SlowReverse(tail), x)
}
```

Эта версия проста и понятна, но она работает медленно. Точнее, количество шагов, выполняемых функцией `SlowReverse(xs)`, пропорционально квадрату длины `xs`. Давайте напишем более эффективную версию и докажем, что она возвращает то же значение, что и эта медленная версия. Но сначала

докажем, что SlowReverse(xs) возвращает список, длина которого равна длине списка xs.

```
lemma LengthSlowReverse<T>(xs: List<T>)
  ensures Length(SlowReverse(xs)) == Length(xs)
```

Посмотрим, сможем ли мы предсказать, что понадобится для доказательства. Глядя на определение SlowReverse, мы ожидаем, что вариант Nil, безусловно, удовлетворяет лемме. В случае Cons индуктивное предположение сообщает нам, что результат SlowReverse(tail) имеет ту же длину, что и tail, поэтому, подтвердив тот факт, что Snoc увеличивает длину списка на 1, мы можем посчитать доказательство законченным. Давайте попробуем:

```
{
  match xs
  case Nil =>
  case Cons(x, tail) =>
    LengthSnoc(SlowReverse(tail), x);
}
```

И действительно, верификатор смог восполнить остальные детали.

Вся хитрость более эффективной версии функции, переставляющей элементы списка в обратном порядке, заключается в дополнительном параметре-аккумуляторе, хранящем ту часть, которая уже обращена к данному моменту. Точнее, идея нашей вспомогательной функции ReverseAux(xs, acc) заключается в том, что она возвращает Append(xs', acc), где xs' – это перевернутый список xs. Вот ее определение:

```
function ReverseAux<T>(xs: List<T>, acc: List<T>): List<T>
{
  match xs
  case Nil => acc
  case Cons(x, tail) => ReverseAux(tail, Cons(x, acc))
}
```

Чтобы перевернуть Cons(x, tail) и добавить acc, мы переворачиваем tail и добавляем Cons(x, acc).

Если бы мы писали метод, то наверняка описали бы предполагаемый эффект ReverseAux в инструкции **ensures** как внутреннюю спецификацию. Мы могли бы сделать то же самое для ReverseAux. Во-первых, мы ожидаем, что вспомогательная функция будет вызываться только из одного места, поэтому применим наш руководящий принцип использования внутренней спецификации: любой вызывающий код заинтересован в этом свойстве. Однако для верификации свойства необходимо предоставить некоторые доказательства вручную. Я покажу, как это сделать, в разделе 6.7. А пока сформулируем и докажем это свойство как внешнее.

```
lemma ReverseAuxSlowCorrect<T>(xs: List<T>, acc: List<T>)
  ensures ReverseAux(xs, acc) == Append(SlowReverse(xs), acc)
{
  match xs
  case Nil =>
  case Cons(x, tail) =>
    calc {
      Append(SlowReverse(xs), acc);
    ==  // определение SlowReverse
      Append(Snoc(SlowReverse(tail), x), acc);
```

Начнем доказательство с правой части, поскольку она более сложная и, следовательно, может дать нам больше информации. После применения определения `SlowReverse` можно надеяться на применимость определения `Snoc`. Поскольку мы не знаем, какой из двух случаев `Snoc` применим, нам нужно будет ввести в вычисление все определение `Snoc`. Иногда такое действительно необходимо. Однако в текущей ситуации у нас есть альтернатива – переписать `Snoc` в терминах `Append`.

```
    ==  { SnocAppend(SlowReverse(tail), x); }
      Append(Append(SlowReverse(tail), Cons(x, Nil)), acc);
```

Здесь мы можем применить свойство ассоциативности `Append`.

```
    ==  { AppendAssociative(SlowReverse(tail), Cons(x, Nil), acc); }
      Append(SlowReverse(tail), Append(Cons(x, Nil), acc));
```

Добавление `acc` в конец одноэлементного списка `Cons(x, Nil)` даст список, начинающийся с `x` и заканчивающийся `acc`. У нас нет отдельной леммы для этого свойства, но она следует из двух развертываний определения `Append`. Чтобы задокументировать этот этап доказательства, используем утверждение **`assert`** в подсказке:

```
    ==  { assert Append(Cons(x, Nil), acc) == Cons(x, acc); }
      Append(SlowReverse(tail), Cons(x, acc));
```

Формула получилась подобной той, что указана в лемме, которую мы пытаемся доказать, поэтому прибегнем к индуктивному предположению:

```
    ==  { ReverseAuxSlowCorrect(tail, Cons(x, acc)); }
      ReverseAux(tail, Cons(x, acc));
```

Теперь верификатор удовлетворен доказательством. На самом деле верификатор был удовлетворен еще до явного вызова индуктивного предположения, потому что об этом позаботился механизм автоматической индукции. Но чтобы закончить удобочитаемое доказательство, сделаем еще один шаг:

```
    ==  // определение ReverseAux
      ReverseAux(xs, acc);
```

```
    }
}
```

Теперь мы доказали, что ReverseAux действует в точности с нашими намерениями. Остается только использовать ее в функции Reverse и доказать, что она дает тот же результат, что и SlowReverse.

```
function Reverse<T>(xs: List<T>): List<T> {
  ReverseAux(xs, Nil)
}
```

Доказательство корректности использует тот факт, что Nil является правым нейтральным элементом для Append, но в остальном оно простое:

```
lemma ReverseCorrect<T>(xs: List<T>)
  ensures Reverse(xs) == SlowReverse(xs)
{
  calc {
    Reverse(xs);
  ==  // определение Reverse
    ReverseAux(xs, Nil);
  ==  { ReverseAuxSlowCorrect(xs, Nil); }
    Append(SlowReverse(xs), Nil);
  ==  { AppendNil(SlowReverse(xs)); }
    SlowReverse(xs);
  }
}
```

Поскольку функция Reverse действует эффективнее, чем SlowReverse, то для нас предпочтительнее, чтобы в скомпилированном коде использовалась она. Чтобы отметить это обстоятельство в исходном коде, можно объявить SlowReverse призрачной функцией.

Тот факт, что функции Reverse и SlowReverse действуют одинаково, за исключением времени выполнения, означает, что они имеют одинаковые свойства. Как результат, мы можем теперь сформулировать лемму о корректности ReverseAux в терминах Reverse вместо SlowReverse.

```
lemma ReverseAuxCorrect<T>(xs: List<T>, acc: List<T>)
  ensures ReverseAux(xs, acc) == Append(Reverse(xs), acc)
{
  ReverseCorrect(xs);
  ReverseAuxSlowCorrect(xs, acc);
}
```

Аналогично из связи между Reverse и SlowReverse и уже доказанной нами леммы о сохранении длины списка функцией SlowReverse следует, что Reverse тоже не меняет эту длину.

```
lemma LengthReverse<T>(xs: List<T>)
  ensures Length(Reverse(xs)) == Length(xs)
{
  ReverseCorrect(xs);
  LengthSlowReverse(xs);
}
```

Упражнение 6.19

Докажите, что условие `Length(ReverseAux(xs, acc)) == Length(xs) + Length(acc)` выполняется для любых списков `xs` и `acc`.

Упражнение 6.20

Напишите альтернативное доказательство для `LengthReverse`, использовав лемму из упражнения 6.19.

Еще одно интересное свойство, которое следует учитывать, – это перестановка в обратном порядке элементов конкатенации двух списков. Начнем со свойства, связывающего `ReverseAux` и `Append`.

```
lemma ReverseAuxAppend<T>(xs: List<T>, ys: List<T>, acc: List<T>)
  ensures ReverseAux(Append(xs, ys), acc)
       == Append(Reverse(ys), ReverseAux(xs, acc))
{
  match xs
  case Nil =>
    ReverseAuxCorrect(ys, acc);
  case Cons(x, tail) =>
}
```

Необычность этой леммы заключается в том, что вариант `Nil` требует явного доказательства, а вариант `Cons` доказывается автоматически.

Упражнение 6.21

Добавьте в лемму `ReverseAuxAppend` атрибут `{:induction false}` и напишите доказательные вычисления для варианта `Cons`.

Упражнение 6.22

Докажите следующую лемму:

```
lemma ReverseAppend<T>(xs: List<T>, ys: List<T>)
  ensures Reverse(Append(xs, ys))
       == Append(Reverse(ys), Reverse(xs))
```

Упражнение 6.23.

Докажите, что `Reverse` является инволюцией, т. е. что

```
Reverse(Reverse(xs)) == xs
```

Подсказка: используйте леммы из предыдущего раздела и из упражнения 6.22.

6.7. Леммы в выражениях

Мы использовали три конструкции доказательства: встроенные утверждения, доказательные вычисления и вызовы лемм. Все это – инструкции, и мы использовали их в телах методов и лемм, чтобы помочь справиться с проверяемыми условиями. Существуют также условия корректности функций. До сих пор удавалось автоматически справляться со всеми подобными проверяемыми условиями, но существует множество функций, где это не так.

В Dafny общие инструкции не могут использоваться в выражениях, но это не относится к инструкциям доказательств. Если `S` – встроенное утверждение, доказательные вычисления или вызов леммы, а `E` – выражение, то `S E` тоже является выражением. Например, **`assert`** `0 < x; 100 / x` – это выражение.

Значением выражения `S E` является просто `E`. Инструкция `S` не влияет на значение `E`. Фактически часть `S` всегда интерпретируется как призрачная, поэтому она элиминируется во время компиляции и отсутствует во время выполнения. Инструкция доказательства `S` используется, только чтобы помочь доказать любые проверяемые условия, возникающие в `E` (и далее в любом объемлющем выражении). Например, значение **`assert`** `0 < x; 100 / x` равно `100 / x`, но проверяемое условие, что делитель `x` не равен нулю, следует из утверждения **`assert`** `0 < x;`. Конечно, часть `S` сама по себе может повлечь за собой проверяемые условия (например, `0 < x`, когда используется инструкция **`assert`** `0 < x`), и они тоже должны быть доказаны.

Следующие два примера демонстрируют использование конструкций доказательств в выражениях.

6.7.0. Внутренняя спецификация ReverseAux

В разделе 6.6 мы доказали лемму о том, что `ReverseAux` корректно реализует наше намерение. Я утверждал, что это свойство является хорошим кандидатом, чтобы объявить его внутренним, т. е. как условие **`ensures`** функции. Для этого объявим `ReverseAux` следующим образом:

```
function ReverseAux<T>(xs: List<T>, acc: List<T>): List<T>
  ensures ReverseAux(xs, acc) == Append(SlowReverse(xs), acc)
{
  match xs
  case Nil => acc
```

```
  case Cons(x, tail) =>
   ReverseAux(tail, Cons(x, acc)) // ошибка: невозможно доказать постусловие
}
```

Увы, верификатор не смог автоматически доказать это постусловие, исходя из данного тела функции. Точнее, верификатор не смог показать, что результаты вызовов `ReverseAux(xs, acc)` и `Append(SlowReverse(xs), acc)` в инструкции **`ensures`** равны.

Чтобы помочь верификатору, добавим перед рекурсивным вызовом доказательное вычисление, показывающее, что результаты вызовов `ReverseAux(xs, acc)` и `Append(SlowReverse(xs), acc)` равны:

```
function ReverseAux<T>(xs: List<T>, acc: List<T>): List<T>
  ensures ReverseAux(xs, acc) == Append(SlowReverse(xs), acc)
{
  match xs
  case Nil => acc
  case Cons(x, tail) =>
    calc {
      Append(SlowReverse(xs), acc);
    ==  // определение SlowReverse
      Append(Snoc(SlowReverse(tail), x), acc);
    ==  { SnocAppend(SlowReverse(tail), x); }
      Append(Append(SlowReverse(tail), Cons(x, Nil)), acc);
    ==  { AppendAssociative(SlowReverse(tail), Cons(x, Nil), acc); }
      Append(SlowReverse(tail), Append(Cons(x, Nil), acc));
    ==  { assert Append(Cons(x, Nil), acc) == Cons(x, acc); }
      Append(SlowReverse(tail), Cons(x, acc));
    ==  // постусловие ReverseAux
      ReverseAux(tail, Cons(x, acc));
    }
    ReverseAux(tail, Cons(x, acc))
}
```

Само доказательное вычисление идентично тому, что мы написали во внешней лемме `ReverseAuxSlowCorrect` в разделе 6.6, за исключением последней строки, апеллирующей к определению `ReverseAux`, которая не была включена в предыдущее вычисление.

Обратите внимание, что доказательное вычисление *предшествует* окончательному выражению `ReverseAux`, даже притом, что мы используем его для доказательства постусловия охватывающей функции. Это связано с тем, что синтаксис всегда имеет вид `S E`, где `S` – доказывающая инструкция, а `E` – выражение. Если вы предпочитаете иной порядок, то используйте let-выражение:

```
var r := ReverseAux(tail, Cons(x, acc));
calc {
  // здесь находятся доказательные вычисления
```

```
}
r
```

Независимо от выбранной формулировки наличие доказательства в теле функции ReverseAux позволяет определять свойство корректности ReverseAux как внутреннее. Однако такой подход приводит к загромождению тела функции. (Это никак не влияет на поведение функции во время выполнения, но читать ее становится труднее.) Альтернативное решение – выделить доказательное вычисление в отдельную лемму и вызвать эту лемму в теле функции ReverseAux:

```
function ReverseAux<T>(xs: List<T>, acc: List<T>): List<T>
  ensures ReverseAux(xs, acc) == Append(SlowReverse(xs), acc)
{
  match xs
  case Nil => acc
  case Cons(x, tail) =>
    ReverseAuxHelper(xs, acc);
    ReverseAux(tail, Cons(x, acc))
}

lemma ReverseAuxHelper<T>(xs: List<T>, acc: List<T>)
  requires xs.Cons?
  ensures ReverseAux(xs.tail, Cons(xs.head, acc))
       == Append(SlowReverse(xs), acc)
  decreases xs, acc, 0
{
  var x, tail := xs.head, xs.tail;
  calc {
    Append(SlowReverse(xs), acc);
  ==  // определение SlowReverse
    Append(Snoc(SlowReverse(tail), x), acc);
  ==  { SnocAppend(SlowReverse(tail), x); }
    Append(Append(SlowReverse(tail), Cons(x, Nil)), acc);
  ==  { AppendAssociative(SlowReverse(tail), Cons(x, Nil), acc); }
    Append(SlowReverse(tail), Append(Cons(x, Nil), acc));
  ==  { assert Append(Cons(x, Nil), acc) == Cons(x, acc); }
    Append(SlowReverse(tail), Cons(x, acc));
  ==  // постусловие ReverseAux
    ReverseAux(tail, Cons(x, acc));
  }
}
```

Обратите внимание, что это преобразование сделало функцию ReverseAux взаимно рекурсивной с ReverseAuxHelper – функция вызывает лемму, а лемма использует функцию (как в постусловии, так и в теле). По этой причине нам нужно подумать о свойстве завершимости. Поскольку в ReverseAux нет

явной инструкции **decreases**, Dafny по умолчанию использует лексикографический кортеж xs,acc (т. е. параметры функции). То же самое можно было бы сделать и с леммой, но тогда окажется, что при вызове леммы из функции ничего не уменьшается. Вместо этого мы вручную напишем инструкцию **decreases** для леммы. Для успеха достаточно взять любую лексикографическую тройку, начинающуюся с xs, поскольку, определяя порядок, Dafny всегда помещает более длинный кортеж за более коротким (см. раздел 3.3.3).

Упражнение 6.24

Компонент acc в инструкции **decreases** функции ReverseAux и леммы ReverseAuxHelper никогда не используется. Напишите явные инструкции **decreases** для ReverseAux и ReverseAuxHelper, доказывающие завершимость и не упоминающие acc.

6.7.1. Лемма об At и Reverse

Вот еще один пример, когда корректность выражений требует доказательства. Рассмотрим лемму, утверждающую, что i-й от начала элемент исходного списка xs совпадает с i-м от конца элементом списка, возвращаемого функцией Reverse(xs):

```
lemma AtReverse<T>(xs: List<T>, i: nat)
  requires i < Length(xs)
  ensures At(xs, i) == At(Reverse(xs), Length(xs) - 1 - i)
    // в предыдущей строке возможно нарушение предусловия
```

Инструкция **ensures** этой леммы, записанная в таком виде, не является корректно определенной. Проблема в том, что At требует корректного индекса. Предусловие леммы говорит нам, что i является корректным индексом в xs, но, ничего не зная о длине Reverse(xs), нельзя утверждать, что length(xs) - 1 - i является корректным индексом в Reverse(xs). Мы могли бы добавить еще одно постусловие:

```
lemma AtReverse<T>(xs: List<T>, i: nat)
  requires i < Length(xs)
  ensures Length(xs) - 1 - i < Length(Reverse(xs))
  ensures At(xs, i) == At(Reverse(xs), Length(xs) - 1 - i)
```

аналогично дополнительному конъюнкту, который мы добавили в постусловие леммы AtDropHead в разделе 6.4. Однако здесь этот дополнительный конъюнкт лишь загромождает формулировку леммы. Поэтому, чтобы убедить Dafny в том, что вызов At действительно удовлетворяет своему предусловию, мы вызовем лемму LengthReverse в самом выражении постусловия!

```
lemma AtReverse<T>(xs: List<T>, i: nat)
  requires i < Length(xs)
  ensures (LengthReverse(xs);
    At(xs, i) == At(Reverse(xs), Length(xs) - 1 - i))
```

(Для обратной совместимости со спецификациями, которые заканчиваются точкой с запятой, Dafny допускает необязательную точку с запятой в конце спецификаций. Неприятным следствием этой поддержки является необходимость заключать в круглые скобки спецификации, вызывающие лемму, как сделано в этом постусловии.)

В теле этой леммы мы используем доказательное вычисление, начинающееся с `At(Reverse(xs), Length(xs) - 1 - i)`. Но оно вызывает сообщение об ошибке, что в `At` может быть передан некорректный индекс. Наш новый вызов `LengthReverse` в постусловии дает информацию о `Length(Reverse(xs))`, которая используется в остальной части постусловия, но не делает эту информацию доступной в теле. Поэтому нам нужно также вызвать лемму в теле.

Очень скоро нам понадобятся голова и хвост `xs`, поэтому давайте попутно введем локальные переменные для их хранения. Как отмечалось, когда мы обсуждали функцию `At` в разделе 6.4, предусловие `i < Length(xs)` подразумевает `xs.Cons?`. Поэтому начнем тело `AtReverse` так:

```
{
  var x, tail := xs.head, xs.tail;
  LengthReverse(xs);
  calc {
    At(Reverse(xs), Length(xs) - 1 - i);
  ==  // определение Reverse
    At(ReverseAux(xs, Nil), Length(xs) - 1 - i);
  ==  // определение ReverseAux
    At(ReverseAux(tail, Cons(x, Nil)), Length(xs) - 1 - i);
  ==  { ReverseAuxSlowCorrect(tail, Cons(x, Nil)); }
    At(Append(SlowReverse(tail), Cons(x, Nil)), Length(xs) - 1 - i);
  ==  { ReverseCorrect(tail); }
    At(Append(Reverse(tail), Cons(x, Nil)), Length(xs) - 1 - i);
```

Эти шаги вычислений довольно просты. Несмотря на все разговоры о корректности выражений, у вас может возникнуть вопрос, почему Dafny не жалуется на предусловие `At` во второй и последующих строках? Дело в том, что, когда Dafny проверяет корректность строки в вычислениях, он предполагает корректность предыдущей строки. Обычно этот подход работает настолько безупречно, что вы, вероятно, даже не подумали бы о необходимости проверить это.

Текущая строка в нашем доказательстве применяет операцию `At` к результату `Append`. Лемма `AtAppend` говорит, что такое выражение можно переписать как применение `At` либо к первому, либо ко второму аргументу-

списку в вызове Append. Однако на этом этапе доказательства мы не можем сказать, какая это будет часть, поэтому мы должны добавить выражение **if-then-else**:

```
==  { AtAppend(Reverse(tail), Cons(x, Nil), Length(xs) - 1 - i); }
  if Length(xs) - 1 - i < Length(Reverse(tail)) then
    At(Reverse(tail), Length(xs) - 1 - i)
  else
    At(Cons(x, Nil), Length(xs) - 1 - i-Length(Reverse(tail)));
```

Не будем расстраиваться из-за большой длины этого выражения и попробуем упростить некоторые его части:

```
==  { LengthReverse(tail); }
  if Length(xs) - 1 - i < Length(tail) then
    At(Reverse(tail), Length(xs) - 1 - i)
  else
    At(Cons(x, Nil), Length(xs) - 1 - i - Length(tail));
==  // арифметика с учетом Length(xs) == Length(tail) + 1 и 0 <= i
  if 0 < i then
    At(Reverse(tail), Length(tail) - 1 - (i - 1))
  else
    At(Cons(x, Nil), 0);
```

Просто, продолжая переносить **if** в вычислениях, мы быстро запутаемся. Поэтому закроем вычисления и разобьем их на два случая, один из которых автоматически подтверждается верификатором:

```
  }
  if 0 < i {
```

Опираясь на это условие, продолжим вычисления со строки `At(Reverse(tail), Length(tail) - 1 - (i - 1))`. Новое вычисление не имеет информации о корректности любых других вычислений, поэтому нужно еще раз вызвать лемму LongReverse, на этот раз для tail.

```
    LengthReverse(tail);
    calc {
      At(Reverse(tail), Length(tail) - 1 - (i - 1));
```

У меня для вас хорошая новость: мы почти закончили. Это выражение имеет форму леммы, которую мы пытаемся доказать, поэтому прибегнем к индуктивному предположению и на этом закончим.

```
    ==  { AtReverse(tail, i - 1); }
      At(tail, i - 1);
    ==  // определение At
      At(xs, i);
    }
  }
}
```

При формулировке и доказательстве этой леммы нам пришлось несколько раз апеллировать к лемме `LengthReverse`. Мы могли бы избежать этих вызовов леммы, если бы описали свойство `LengthReverse` как постусловие функции `Reverse`. Написание таких внутренних спецификаций – заманчивая и иногда хорошая идея. Они упрощают доказательства, автоматически предоставляя дополнительные факты верификатору. Однако, как я уже упоминал, в более сложных ситуациях эти дополнительные факты могут сбивать верификатор с толку. Поэтому не злоупотребляйте внутренними спецификациями.

Упражнение 6.25

В разделе 6.4 мы столкнулись с проблемой при формулировке леммы `AtDropHead` из-за того, что выражение в проверяемом условии не было корректно определено. Сформулируйте лемму:

```
lemma DropLessThanEverything<T>(xs: List<T>, i: nat)
  requires i < Length(xs)
  ensures Drop(xs, i).Cons?
```

и используйте ее в проверяемом условии в первоначальной попытке написать лемму `AtDropHead`. Затем докажите `DropLessThanEverything` и новую `AtDropHead`.

6.8. Исключение аргументов типа

В этой главе мы рассмотрели множество примеров функций и лемм о списках. Во всех этих примерах тип элемента списка был абстрактным, т. е. наши функции и леммы были параметризованы типом, которому в этой главе я дал имя `T`. Поскольку конкретный тип полезной нагрузки не имеет значения, у вас может возникнуть ощущение, что сигнатуры типов, такие как

```
function ReverseAux<T>(xs: List<T>, acc: List<T>): List<T>
```

чересчур подробны и многословны. Если нет необходимости все время перечислять тип `T`, то можно воспользоваться правилом исключения в Dafny, полностью избавляющим от необходимости упоминать тип `T`. В других главах книги я буду использовать две формы этого правила. Правило исключения – уникальная особенность Dafny, поэтому я объясню его здесь.

Напомню, что в Dafny параметризуемый тип, такой как `List`, всегда должен создаваться с некоторым аргументом типа; например, `List<int>` или `List<T>`, где `T` – параметр типа. В сигнатуре функции, метода или леммы любой параметризуемый тип, упомянутый без аргументов типа (т. е. без угловых скобок, в которых задаются такие аргументы), автоматически заполняется списком параметров типа вмещающей функции, метода или

леммы. Эта форма правила исключения означает, что сигнатуру функции ReverseAux, показанную выше, можно записать как

```
function ReverseAux<T>(xs: List, acc: List): List
Сигнатуру функции Length (раздел 6.1) можно записать как
function Length<T>(xs: List): nat
а сигнатуру функции At (раздел 6.4) – как
function At<T>(xs: List, i: nat): T
```

Другая форма правила исключения: если параметр типа функции, метода или леммы больше нигде не упоминается, то его тоже можно исключить. Соответственно, сигнатуры ReverseAux и Length можно сократить до

```
function ReverseAux(xs: List, acc: List): List
```

и

```
function Length(xs: List): nat
```

Однако это второе правило неприменимо к функции At, потому что в At необходимо указать параметр типа в качестве типа результата.

Упражнение 6.26

Используя правила исключения, сократите сигнатуры Append, Take, AtAppend, Find, AtDropHead и Reverse.

6.9. Итоги

В этой главе я представил тип данных List, широко используемый в функциональном программировании. Операции с List определяются как рекурсивные функции, поэтому леммы, определяющие свойства операций, хорошо подходят для индуктивных доказательств.

Свойство функции, объявленное как ее постусловие, называется *внутренней* спецификацией. Это неотъемлемая часть функции и должна доказываться как часть доказательства корректности функции.

Свойство функции, которое объявляется отдельно от нее в виде леммы, называется *внешней* спецификацией. Такие свойства, как, например, ассоциативность, доказываются рассмотрением определения функции, и их легко идентифицировать по необходимости многократно вызывать функции.

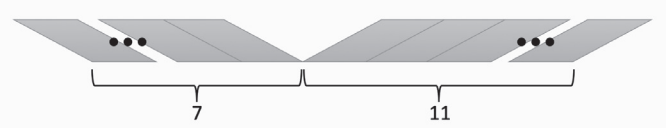
В этой главе я также показал использование аккумулятора для получения более эффективной реализации функции; использование характеристики типа (==), которая говорит, что тип поддерживает скомпилированные операции равенства (подробнее об этом рассказывается в разделе 9.4); различные методы, обеспечивающие корректность выражений, включая использование доказательных инструкций в выражениях; и правило Dafny для исключения параметров типа в сигнатурах функций, методов и лемм.

Примечания

В Dafny корректность постусловия функции или метода должна вытекать из предусловия безотносительно к телу функции или метода. В SPARK [43] и OpenJML [105] это делается по-другому, где корректность постусловия для функции или метода с телом проверяется при выходе из тела.

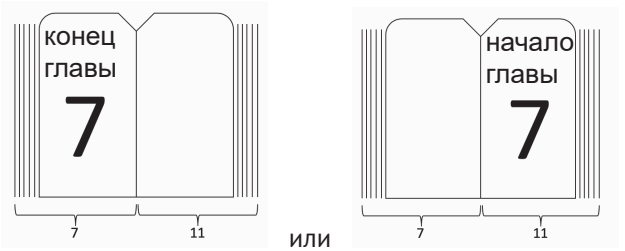
Врезка 6.1

Предположим, у вас есть книга с 18 главами, и вы открываете ее так, что 7 глав оказываются слева, а остальные 11 глав – справа, как показано здесь:



Если вы дочитали до того места, где открыли книгу, значит, вы прочитали 7 глав.

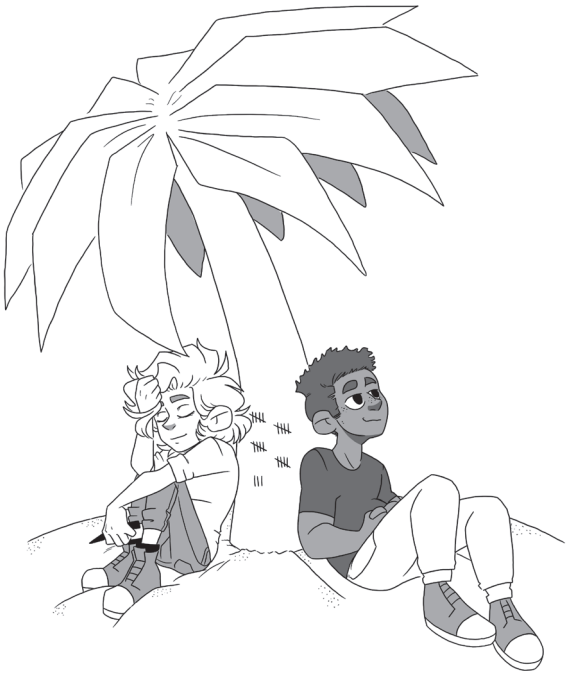
Давайте обозначим их номерами. Как мы пронумеруем главы, видимые на развороте? Конечно, будет удобно обозначить номером «7» главу слева или справа. У нас есть два варианта напечатать «7» на этом развороте



Лично я предпочитаю видеть номер главы в тот момент, когда начинаю ее читать, а не когда заканчиваю. Поэтому в этой книге, собираясь начать 7-ю главу, вы будете знать, что до нее уже прочитано 7 глав.

Глава 7

Унарные числа



В каменном веке людям не требовалось производить сложные вычисления. Их потребности в подсчетах вполне удовлетворялись несколькими черточками, нарисованными на стене пещеры, или несколькими сосновыми шишками, собранными в кучу. Но уже в чуть более сложных ситуациях мы обнаруживаем проблемы с такими унарными системами нумерации. В нашем более развитом мире мы неизбежно столкнулись бы с проблемами, обусловленными не только невозможностью отличить запись числа ноль от отсутствия записи, но также и неэффективностью способов выполнения арифметических действий с такими числами. И все же в этой главе я вернусь к примитивным унарным числам, чтобы определить некоторые функции и доказать некоторые леммы.

7.1. Основные определения

Унарные числа определяются индуктивно:

```
datatype Unary = Zero | Suc(pred: Unary)
```

Как видите, унарное число – это либо ноль, либо число, следующее за каким-то другим унарным числом. Определяя новый тип данных, мы не всегда можем позволить себе связать его с другим эквивалентным представлением. Вместо этого часто приходится доказывать, что наше определение имеет нужные нам свойства. Но в случае с унарными числами мы можем напрямую задать функции преобразования в натуральные числа и обратно:

```
function UnaryToNat(x: Unary): nat {
  match x
  case Zero => 0
  case Suc(x') => 1 + UnaryToNat(x')
}

function NatToUnary(n: nat): Unary {
  if n == 0 then Zero else Suc(NatToUnary(n-1))
}
```

Опираясь на эти определения, легко доказать, что унарные и натуральные числа находятся во взаимно однозначном соответствии друг с другом, т. е. что функция преобразования в каждом направлении инъективна.

```
lemma NatUnaryCorrespondence(n: nat, x: Unary)
  ensures UnaryToNat(NatToUnary(n)) == n
  ensures NatToUnary(UnaryToNat(x)) == x
{
}
```

Упражнение 7.0

Не обращая внимания на осмысленность, измените тела `UnaryToNat` и `NatToUnary` так, чтобы нарушить инъективность, т. е. сделайте лемму `NatUnaryCorrespondence` ложной.

В этой главе я буду использовать эти функции, чтобы доказать, что определяемые нами операции над унарными числами действительно соответствуют аналогичным операциям над натуральными числами, например что `Add` соответствует `+`. Эти свойства будут заданы в леммах с именами вида `...Correct`. Вся суть этой главы в том, чтобы заменить тип **`nat`**, встроенный в Dafny, типом `Unary`, поэтому не будем использовать эти функции преобразования или лемму `NatUnaryCorrespondence` для чего-либо еще.

7.2. Сравнения

Наша первая операция над унарными числами – сравнение:

```
predicate Less(x: Unary, y: Unary) {
  y != Zero && (x.Suc? ==> Less(x.pred, y.pred))
}
```

В этом определении говорится, что `x` (строго) меньше `y`, когда `y` не равно нулю, и, более того, если `x` также не равно нулю, то число, предшествующее `x`, меньше числа, предшествующего `y`. Как упоминалось в разделе 4.2, предикат `Less` можно также определить, используя выражение **`match`** с двумя ветвям **`case`**. Однако, как мне кажется, приведенное мной определение более простое.

Упражнение 7.1

Используя выражение **match**, определите предикат Less'. Докажите, что он эквивалентен предикату Less, представленному выше.

Упражнение 7.2

Определите Less без выражения **match**, без дискриминатора Suc? и без оператора импликации ==>.

Чтобы доказать себе, что мы определили корректный порядок, запишем следующую лемму, которая доказывается автоматически:

```
lemma LessCorrect(x: Unary, y: Unary)
  ensures Less(x, y) <==> UnaryToNat(x) < UnaryToNat(y)
{
}
```

Обратите внимание, что в постусловии леммы используется логический оператор эквивалентности <==> («тогда и только тогда», имеющий приоритет, более низкий, чем приоритет любого другого двухместного оператора). В качестве альтернативы мы могли бы использовать оператор равенства логических значений:

```
ensures Less(x, y) == (UnaryToNat(x) < UnaryToNat(y))
```

Однако, в отличие от ==, оператор <==> имеет более низкий приоритет, чем <, поэтому в первой формулировке использована пара круглых скобок.

Далее покажем, что Less обладает свойством *транзитивности*. Вот один из способов объявить и доказать это свойство:

```
lemma LessTransitive(x: Unary, y: Unary, z: Unary)
  requires Less(x, y) && Less(y, z)
  ensures Less(x, z)
{
}
```

Часто, когда лемма имеет посылку, такую как Less(x, y) && Less(y, z) в этом случае, его лучше записать в инструкции **require**. Благодаря этому если клиенту не удастся выполнить предусловие при вызове леммы, то этот вызов будет отмечен как ошибка. Однако для доказательства свойства транзитивности мы можем разрешить клиенту использовать это свойство в другом направлении, например !Less(y, z) вместо Less(x, y) и !Less(x, z). Поэтому лучше сформулировать эту лемму без предусловия и с импликацией в постусловии:

```
lemma LessTransitive(x: Unary, y: Unary, z: Unary)
  ensures Less(x, y) && Less(y, z) ==> Less(x, z)
```

Врезка 7.0

Приоритет многих операций в Dafny можно определить по длине соответствующего оператора: чем длиннее оператор, тем ниже его приоритет. Четырехсимвольный оператор <==> имеет более низкий приоритет, чем трехсимвольные операторы ==> и <==, а двухсимвольные операторы && и || имеют более высокий приоритет, чем четырех- и трехсимвольные. Еще более высокий приоритет имеют операторы сравнения, как одно-, так и двухсимвольные: ==, !=, <=, <, > и >=. Приоритет односимвольных операторов еще выше, в порядке возрастания приоритета первыми следуют аддитивные операторы + и -, затем мультипликативные операторы *, / и % и, наконец, самые высокоприоритетные – побитовые операторы &, | и ^.

Удивительно, но эта версия леммы не доказывается автоматически. Однако мы не будем разбираться в причинах неудачи, а просто напишем доказательство, которое проиллюстрирует полезные для обучения моменты. Мы доказываем `Less(x, z)` в предположении, что

```
Less(x, y) && Less(y, z)
```

Чтобы ввести это предположение в наше доказательство, используем условную инструкцию:

```
{
  if Less(x, y) && Less(y, z) {
```

и запишем доказательство для ветви **then**. Как это принято в программировании, предположим, что указанное условие выполняется при входе в ветвь **then**. В ветви **else** нам нечего доказывать, поскольку лемма доказывается тривиально, если условие `Less(x, y) && Less(y, z)` не выполняется.

Чтобы доказать корректность `Less(x, z)` внутри ветви **then**, заметим, что определение `Less(x, z)` состоит из двух конъюнктов. То есть, чтобы доказать `Less(x, z)`, нам нужно доказать оба этих конъюнкта. Первый конъюнкт `z != Zero` вытекает непосредственно из того же конъюнкта в определении `Less(y, z)`. Второй конъюнкт является импликацией, поэтому мы продолжим доказательство, добавив еще одну инструкцию **if**:

```
    if x.Suc? {
```

По определению `Less` внешний **if** подразумевает, что `y` и `z` не равны нулю, а внутренний **if** подразумевает, что `x` не равно нулю. Это означает, что мы можем рекурсивно вызвать лемму для значений, предшествующих значениям `x`, `y` и `z`. Другими словами, мы применяем индуктивное предположение к этим предшествующим значениям:

```
        LessTransitive(x.pred, y.pred, z.pred);
      }
    }
  }
```

Очевидно, что Dafny сможет автоматически добавить остальные детали доказательства, и на этом доказательство будет завершено.

Упражнение 7.3

Сформулируйте и докажите лемму о том, что `Less` трихотомична, т. е., что выполняется *ровно одно* из следующих условий: `Less(x, y)`, `x == y`, `Less(y, x)`. (Помните, что здесь и в других местах этой главы мы не должны позволять вашему доказательству зависеть от преобразования `Unary` в **`nat`**.)

Упражнение 7.4

Определите предикат `Below(x, y)` как `Less(x, y) || x == y`. Сформулируйте и докажите, что `Below` представляет *тотальный порядок*, т. е. рефлексивный, транзитивный, антисимметричный (`Below(x, y) && Below(y, x)` подразумевает `x == y`), и полноту (`Below(x, y) || Below(y, x)`).

7.2. Сложение и вычитание

Сложение двух унарных чисел определяется рекурсивной функцией `Add`:

```
function Add(x: Unary, y: Unary): Unary {
  match y
  case Zero => x
  case Suc(y') => Suc(Add(x, y'))
}
```

Эта функция соответствует сложению натуральных чисел, как показывает следующая лемма:

```
lemma AddCorrect(x: Unary, y: Unary)
  ensures UnaryToNat(Add(x, y)) == UnaryToNat(x) + UnaryToNat(y)
{
}
```

Определение `Add(x, y)` рассматривает разные варианты для `y`. Из этого немедленно вытекает свойство `Suc(Add(x, y)) == Add(x, Suc(y))`. Другими словами, `Suc` характеризуется свойством дистрибутивности по второму аргументу `Add`. Однако `Suc` также обладает свойством дистрибутивности по первому аргументу `Add`, но оно требует индуктивного доказательства:

```
lemma SucAdd(x: Unary, y: Unary)
```

```
  ensures Suc(Add(x, y)) == Add(Suc(x), y)
{
}
```

Мы могли бы определить `Add(x, y)`, анализируя различные варианты для `x`, и тогда автоматически получили бы свойство дистрибутивности `Suc` по первому аргументу `Add`, но тогда нам пришлось бы привести индуктивное доказательство свойства дистрибутивности `Suc` по второму аргументу `Add`.

Определение `Add` сразу же дает нам понять, что `Zero` является правым нейтральным элементом `Add` (т. е. `Add(x, Zero) == x` для любого `x`). Вот доказательство того, что `Zero` также является левым нейтральным элементом:

```
lemma AddZero(x: Unary)
  ensures Add(Zero, x) == x
{
}
```

Упражнение 7.5

Отключите автоматическую индукцию и напишите доказательства для:

а) `AddCorrect`,

б) `SucAdd` и

в) `AddZero`.

Упражнение 7.6

Докажите свойство ассоциативности `Add`:

```
Add(Add(x, y), z) == Add(x, Add(y, z))
```

Упражнение 7.7

Докажите свойство коммутативности `Add`:

```
Add(x, y) == Add(y, x).
```

Упражнение 7.8

Докажите следующую лемму, утверждающую, что `Add` обладает обоими свойствами, ассоциативности и коммутативности:

```
lemma AddCommAssoc(x: Unary, y: Unary, z: Unary)
  ensures Add(Add(x, y), z) == Add(Add(x, z), y)
```

Определим также функцию вычитания унарных чисел. Поскольку у нас нет отрицательных унарных чисел, то определим `Sub(x, y)` только для случая, когда `x` не меньше `y`.

```
function Sub(x: Unary, y: Unary): Unary
  requires !Less(x, y)
{
  match y
  case Zero => x
  case Suc(y') => Sub(x.pred, y')
}

lemma SubCorrect(x: Unary, y: Unary)
  requires !Less(x, y)
  ensures UnaryToNat(Sub(x, y)) == UnaryToNat(x) - UnaryToNat(y)
{
}
```

Упражнение 7.9

Напишите подходящее предусловие для следующей леммы. Затем докажите лемму:

```
lemma AddSub(x: Unary, y: Unary)

  // requires ?
  ensures Add(Sub(x, y), y) == x
```

7.3. Умножение

Для унарных чисел умножение определяется многократным сложением:

```
function Mul(x: Unary, y: Unary): Unary {
  match x
  case Zero => Zero
  case Suc(x') => Add(Mul(x', y), y)
}
```

Аналогично сложению докажем, что `Mul` соответствует операции умножения (`*`) натуральных чисел. На этот раз доказательство требует от нас небольшой помощи:

```
lemma MulCorrect(x: Unary, y: Unary)
  ensures UnaryToNat(Mul(x, y)) == UnaryToNat(x) * UnaryToNat(y)
{
  match x
  case Zero =>
  case Suc(x') =>
```

```
    calc {
      UnaryToNat(Mul(x, y));
    ==  // определение Mul
      UnaryToNat(Add(Mul(x', y), y));
    ==  { AddCorrect(Mul(x', y), y); }
      UnaryToNat(Mul(x', y)) + UnaryToNat(y);
      // Дальше Dafny может справиться сам
    }
}
```

7.4. Деление и остаток от деления

Далее, определяя операции деления и вычисления остатка от деления, мы сталкиваемся с несколькими любопытными моментами. Для унарных чисел деление и вычисление остатка от деления выполняются путем многократного вычитания. Поскольку обе операции производят схожие вычисления, упакуем их в одну функцию. Соответственно, определяемая нами функция будет возвращать *пару*. Для любых типов `A` и `B` в Dafny определяется тип, представляющий пару с одним значением `A` и с одним значением `B` (`A, B`):

```
function DivMod(x: Unary, y: Unary): (Unary, Unary)
```

Примем соглашение, в соответствии с которым первый компонент возвращаемой пары – это частное (результат деления `x` на `y`), а второй – остаток от деления. Конструирование пары из значений `d` и `m` записывается просто как `(d, m)`. Деструкторы для извлечения соответствующих компонентов будут называться `0` и `1`, т. е. если `r` – это пара, возвращаемая `DivMod(x, y)`, то `r.0` обозначает частное от деления `x` на `y`, а `r.1` – остаток.

Определим нашу функцию только для ненулевых делителей:

```
  requires y != Zero
```

Простое определение `DivMod` выглядит следующим образом:

```
{
  if Less(x, y) then
    (Zero, x)
  else
   var r := DivMod(Sub(x, y), y); // завершимость не доказывается автоматически
    (Suc(r.0), r.1)
}
```

Однако Dafny не может автоматически доказать завершимость рекурсивного вызова `DivMod`. Давайте рассмотрим два способа, как можно помочь верификатору доказать завершимость.

7.4.0. Доказательство завершимости через натуральные числа

Чтобы доказать завершимость, используем оценочную функцию, которая, как можно показать, уменьшается с каждым рекурсивным вызовом. Мы видим, что `DivMod(x, y)` вызывает `DivMod(Sub(x, y), y)`, поэтому наш единственный выбор – показать, что что-то в первом аргументе уменьшается. Интерпретируя аргументы как натуральные числа, получаем, что `UnaryToNat(x) - UnaryToNat(y)` меньше, чем `UnaryToNat(x)`, поскольку известно, что `y` не равно нулю. Чтобы показать, что в качестве оценочной функции мы используем натуральное число, соответствующее `x`, объявим свою инструкцию **`decreases`**:

```
decreases UnaryToNat(x)
```

Теперь, чтобы доказать завершимость рекурсивного вызова, нужно доказать:

```
UnaryToNat(x) > UnaryToNat(Sub(x, y))
```

Поскольку `UnaryToNat` создает значение типа **`nat`**, это выражение соответствует условию:

```
UnaryToNat(Sub(x, y)) < UnaryToNat(x)
```

Но, к сожалению, оно тоже не доказывается автоматически.

Чтобы еще больше помочь верификатору, вызовем лемму `SubCorrect`, которая гласит, что левая часть этого неравенства равна:

```
UnaryToNat(x) - UnaryToNat(y)
```

Вставим этот вызов леммы непосредственно перед let-выражением:

```
SubCorrect(x, y);
var r := DivMod(Sub(x, y), y);
```

Вызов леммы прекрасно вписывается в этот исходный код. При желании мы могли бы переместить его еще ближе к тому месту, где он нужен:

```
var r := (SubCorrect(x, y); DivMod(Sub(x, y), y));
```

Обратите внимание, что в этом случае необходимы дополнительные круглые скобки, чтобы анализатор не брал `SubCorrect(x, y)` как правую часть let-привязки.

Фактически мы можем поместить вызов леммы еще ближе. Например:

```
var r := DivMod(SubCorrect(x, y); Sub(x, y), y);
```

Важно предоставить верификатору информацию о лемме до того, как он проверит предусловие и завершимость вызова `DivMod`, что происходит после

оценки параметров вызова. Обычно желательно размещать вызов леммы рядом с тем местом, где она необходима, но нет смысла проявлять излишнее усердие, поэтому я предпочитаю вариант с размещением вызова леммы в отдельной строке перед let-привязкой.

7.4.1. Доказательство завершимости посредством структурного включения

В этом примере у нас получилось доказать завершимость, применив соответствие натуральным числам посредством использования `UnaryToNat` и `SubCorrect` в доказательстве завершимости. Но это не единственный способ. Другой способ – прямо доказать, что значение типа данных, возвращаемое функцией `Sub(x, y)`, структурно меньше, чем `x`. Другими словами, если представить унарные числа как риски, нацарапанные на стене пещеры, то `Sub(x, y)` дает более короткую последовательность рисок, чем `x`. Dafny знает, что параметры типов данных конструктора структурно меньше, чем то, что возвращает конструктор. Чтобы понять, что это свойство также выполняется транзитивно, нужно доказать лемму:

```
lemma SubStructurallySmaller(x: Unary, y: Unary)
  requires !Less(x, y) && y != Zero
  ensures Sub(x, y) < x
{
}
```

В Dafny оператор `<` перегружен для работы с различными типами аргументов. В постусловии этой леммы он применяется к значениям типов данных и означает структурное включение. Эта лемма доказывается автоматически по индукции.

Все, что нам нужно сделать, – вызвать лемму перед let-выражением в `DivMod`. Вот полное определение `DivMod`:

```
function DivMod(x: Unary, y: Unary): (Unary, Unary)
  requires y != Zero
{
  if Less(x, y) then
    (Zero, x)
  else
    SubStructurallySmaller(x, y);
    var r := DivMod(Sub(x, y), y);
    (Suc(r.0), r.1)
}
```

В разделе 6.2 я обсуждал о внутренних и внешних определениях и доказательствах свойств функций. В Dafny свойство завершимости функций (и методов тоже) всегда должно определяться и доказываться как внут-

реннее. То есть нельзя объявить функцию, игнорировать завершимость и пытаться доказать завершимость позже в отдельной лемме – завершимость должна быть доказана в самом объявлении функции (возможно, с помощью дополнительных лемм, как показано в примерах `SubCorrect` и `SubStructurallySmaller` выше).

7.4.2. let-выражения с шаблонами

Пока мы не ушли далеко от определения `DivMod`, воспользуемся возможностью познакомиться с еще одной удобной особенностью языка. Левая часть let-выражения может быть шаблоном, как в ветвях `case` в инструкции **`match`**. Например, если `F(e)` – это функция, о которой известно, что она возвращает ненулевое унарное число, то мы можем использовать шаблон `Suc(w)` в левой части let-выражения. Это свяжет `w` со значением параметра `Suc`, возвращаемого `F(e)`. Другими словами, это выражение вычислит значение `F(e)`, затем «срежет» внешний вызов `Suc` для этого значения и привяжет то, что осталось, к `w`. Но для этого необходимо знать, что `F(e)` действительно возвращает `Suc`, а не `Zero`. Итак, после следующих двух let-привязок:

```
var r := F(e);
var Suc(w) := F(e);
```

мы получаем `r == Suc(w)` и `r.pred == w`.

Пара – это встроенный тип данных, поддерживающий специальный синтаксис. В частности, его единственный конструктор записывается только в круглых скобках. Используя шаблон в let-выражении, мы можем записать последние две строки `DivMod` следующим образом:

```
var (d, m) := DivMod(Sub(x, y), y);
(Suc(d), m)
```

Наконец, обратите внимание на тонкую разницу между этим let-выражением, связывающим `d` и `m` с различными компонентами пары, возвращаемой `DivMod`, и некорректным выражением:

```
var d, m := DivMod(Sub(x, y), y); // ошибка
```

где есть две левые части (`d` и `m`) и только одна правая (вызов `DivMod`).

7.4.3. Корректность DivMod

В завершение этой главы я приведу доказательство корректности `DivMod`. Но вместо доказательства корректности с применением натуральных чисел докажем, что частное и остаток удовлетворяют ожидаемым свойствам. Эти свойства выполняются для любых натуральных чисел `a` и `b`:

```
(a / b) * b + (a % b) == a
(a % b) < b
```

(Все круглые скобки здесь излишни. Я добавил их, только чтобы выделить операции деления и вычисления остатка.) В виде леммы об унарных числах мы имеем:

```
lemma DivModCorrect(x: Unary, y: Unary)
  requires y != Zero
  ensures var (d, m) := DivMod(x, y);
    Add(Mul(d, y), m) == x &&
    Less(m, y)
```

Обратите внимание, насколько удобно использовать let-выражение в постусловии этой леммы для присваивания имен тому, что возвращает `DivMod`.

Начнем доказательство с введения локальных переменных (с теми же именами, что и связанные переменные внутри постусловия), обозначающих унарные числа, свойства которых мы собираемся доказать:

```
{
  var (d, m) := DivMod(x, y);
```

Далее рассмотрим два случая в определении `DivMod`. Для доказательства это не обязательно, но мы можем использовать инструкцию **`assert`**, чтобы напомнить себе о том, что возвращает `DivMod` в этом случае:

```
  if Less(x, y) {
    assert d == Zero && m == x; // поскольку (d, m) == DivMod(x, y)
```

Здесь мы сразу имеем `Less(m, y)`. Остальная часть этого случая представляет собой простое вычисление:

```
    calc {
      Add(Mul(d, y), m) == x;
    ==  // d, m
      Add(Mul(Zero, y), x) == x;
    ==  // определение Mul
      Add(Zero, x) == x;
    ==  { AddZero(x); }
      true;
    }
```

Доказательство другого случая начнем с введения имен для результатов, возвращаемых рекурсивным вызовом `DivMod`, и используем **`assert`**, чтобы напомнить себе о том, что возвращает `DivMod(x, y)` в этом случае:

```
  } else {
    var (d', m') := DivMod(Sub(x, y), y);
    assert d == Suc(d') && m == m'; // поскольку (d, m) == DivMod(x, y)
```

Продолжить можно несколькими способами. Обычно доказательные вычисления мы начинаем с более сложной стороны проверяемого условия, потому что это дает нам больше информации, которая может помочь в определении этапов доказательства. В данном случае поступим по-другому и начнем с формулы, находящейся на противоположном конце спектра структурного богатства формул, – с литерала **`true`**:

```
calc {
  true;
```

Это выглядит довольно странно, но, сделав еще один шаг, вы увидите, к чему это приведет: следующим шагом введем свойства, полученные с помощью индуктивного предположения:

```
==>  { SubStructurallySmaller(x, y);
       DivModCorrect(Sub(x, y), y); }
  Add(Mul(d', y), m) == Sub(x, y) && Less(m, y);
```

Обратите внимание: чтобы доказать завершимость рекурсивного вызова `DivModCorrect`, мы сначала вызываем лемму `SubStructurallySmaller` в подсказке. По связке в левом поле также можно сказать, что мы собираемся доказать форму `true ==>` цель доказательства. Остальная часть доказательства проста благодаря использованию лемм, доказанных в упражнениях 7.9 и 7.8:

```
==>  // прибавить y к обеим сторонам
  Add(Add(Mul(d', y), m), y) == Add(Sub(x, y), y) &&
  Less(m, y);
==  { AddSub(x, y); }
  Add(Add(Mul(d', y), m), y) == x && Less(m, y);
==  { AddCommAssoc(Mul(d', y), m, y); }
  Add(Add(Mul(d', y), y), m) == x && Less(m, y);
==  // определение Mul, d
  Add(Mul(d, y), m) == x && Less(m, y);
}
}
}
```

7.5. Итоги

Эта глава в основном служила площадкой для оттачивания практических навыков доказательства свойств функций на простом типе данных. Но мы также столкнулись с несколькими важными моментами.

Во многих рекурсивных функциях можно выбирать, как разложить параметры для рекурсивного вызова и как построить результат рекурсивно-

го вызова. Этот выбор делает доказательство некоторых свойств функции тривиально простым, в то время как аналогичные свойства могут потребовать доказательства по индукции. Например, если определить `Add`, разложив ее первый параметр, то определение сразу же показывает, что `Zero` – это левый нейтральный операнд `Add`, и тогда вам понадобится добавить индуктивное доказательство (с использованием леммы), чтобы показать, что `Zero` также является правым нейтральным операндом. Если определить `Add`, разложив ее второй параметр, то вам понадобится добавить индуктивное доказательство для свойства нейтральности левого операнда, тогда как свойство нейтральности правого операнда следует из определения. Полезно знать о компромиссах, связанных с этой асимметрией. Мы также видели эту асимметрию при использовании `Append` в разделе 6.2.0, а связанные с этим проблемы увидим в разделах 12.2 и 12.3.

В этой главе мы также узнали больше о свойстве завершимости. Во многих случаях завершимость доказывается автоматически, но, даже указывая оценочные функции, доказываемые автоматически, иногда приходится писать доказательства завершимости вручную. В Dafny роль оценочной функции завершимости обычно является (лексикографический кортеж, содержащий) список параметров, но, добавляя вручную инструкцию **`decreases`**, можно превратить в оценочную функцию любой список значений, вычисленных на основе параметров. Инструменты верификации (включая верификатор Dafny) обычно требуют внутреннего доказательства завершимости функции, т. е. условие завершимости должно быть определено как часть самой функции.

Завершимость функций, определяемых для типов данных, часто доказывается с использованием структурного включения. Прямое структурное включение автоматически учитывается верификатором, но, чтобы получить транзитивное структурное включение, может потребоваться доказать лемму.

Чтобы доказать такое свойство, как `P ==> Q`, часто используется доказательство в форме

```
if P {
  // доказательство Q
}
```

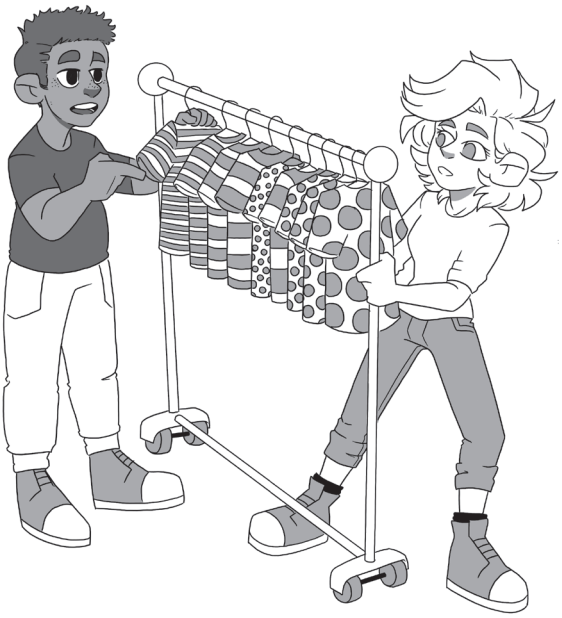
Примечания

Наши функции `UnaryToNat` и `NatToUnary` демонстрируют взаимно однозначное соответствие между значениями типа `Unary` и натуральными числами. Идея представления натуральных чисел с помощью индуктивного типа данных, такого как `Unary`, с конструкторами `Zero` и `Suc`, вытекает непосредственно из *аксиоматизации* натуральных чисел, разработанной Джузеппе Пеано (Giuseppe Peano). Поэтому множество унарных чисел – точнее, представление натуральных чисел `Zero`/`Suc` – часто называют числами Пеано.

Поскольку числа Пеано описываются таким простым индуктивным типом данных, они пользуются большой популярностью как предмет формальных доказательств.

Глава 8

Сортировка



Сортировка – одна из часто встречающихся тем в информатике. В этой главе рассматривается корректность двух алгоритмов сортировки, работающих со списками. Тип данных `List` и различные функции для работы со списками, такие как `Length`, взяты из главы 6.

В этой главе я начну отделять основные алгоритмические функции, которые в конечном итоге должны компилироваться, от вспомогательных функций, используемых только в спецификациях. Эти вспомогательные функции, предназначенные исключительно для спецификаций, будут объявляться как призрачные.

8.0. Спецификация

Начнем со спецификации сортировки. Очевидно, нам нужно, чтобы процедуры сортировки возвращали упорядоченные данные.

8.0.0. Свойство упорядоченности

После сортировки списка целочисленных значений его элементы должны располагаться в порядке возрастания. При работе со списками это свойство проще всего выразить, сказав, что любые два элемента списка, следующие друг за другом, упорядочены.

```
ghost predicate Ordered(xs: List<int>) {
  match xs
  case Nil => true
```

```
    case Cons(x, Nil) => true
    case Cons(x, Cons(y, _)) => x <= y && Ordered(xs.tail)
  }
```

Есть три случая. Первые два применяются к пустому и одноэлементному списку соответственно, которые всегда считаются отсортированными. Использование конструктора `Nil` внутри шаблона `Cons` во втором случае – это вложенный шаблон списка, состоящего из головы `xs` и хвоста `Nil`. Если этот случай применим, то `x` привязывается к голове `xs` для использования в теле конструкции `case`. В третьем случае также используется вложенный шаблон. Ему соответствуют случаи, когда `xs` и его хвост являются значениями `Cons`, и тогда `x` и `y` связываются с `xs.head` и `xs.tail.head` соответственно. Как обычно в шаблонах, подчеркивание в позиции `xs.tail.tail` говорит о том, что нам не нужна связанная переменная, обозначающая это подвыражение. Тело третьего случая оценивается как **`true`**, когда первые два элемента `xs` упорядочены, а элементы `xs.tail` упорядочены попарно.

Если вы находите определения, такие как `Ordered`, сложными или неочевидными, то рекомендуется проверить, обладает ли оно некоторыми ожидаемыми свойствами. От списка, удовлетворяющего предикату `Ordered`, ожидается, что любой элемент, стоящий раньше в списке, будет меньше любого элемента, стоящего позже в списке. Мы можем точно выразить это, используя функцию `At` (из раздела 6.4), которая извлекает определенный элемент списка. Это приводит нас к следующей лемме:

```
lemma AllOrdered(xs: List<int>, i: nat, j: nat)
  requires Ordered(xs) && i <= j < Length(xs)
  ensures At(xs, i) <= At(xs, j)
```

Эта лемма не доказывается автоматически, поэтому давайте поможем верификатору и напишем доказательство вручную. По определению `At` если `i` не равно нулю (что по предварительному условию `AllOrdered` подразумевает, что `j` тоже не равно нулю), то наша цель – доказать истинность условия:

```
At(xs.tail, i - 1) <= At(xs.tail, j - 1)
```

поэтому начнем доказательство `AllOrdered` следующим образом:

```
{
  if i != 0 {
    AllOrdered(xs.tail, i - 1, j - 1);
```

Если `i` равно `0`, то нам нужно доказать, что переход в сторону конца списка до позиции `j` даст нам элемент, величина которого не меньше головы `xs`. Это тривиально, если `j` тоже равно `0`:

```
  } else if i == j {
```

Если `j` не равно нулю, то мы получаем следующее свойство:

```
At(xs, 0) <= At(xs, 1) <= At(xs, j)
```

Условие `At(xs, 0) <= At(xs, 1)` вытекает из факта упорядоченности списка. Условие `At(xs, 1) <= At(xs, j)` эквивалентно условию `At(xs.tail, 0) <= At(xs.tail, j - 1)`, а значит его можно получить из индуктивного предположения, рекурсивно вызывая лемму. Соответственно наше доказательство `AllOrdered` заканчивается так:

```
  } else {
    AllOrdered(xs.tail, 0, j - 1);
  }
}
```

Теперь мы можем быть уверены, что корректно определили свойство `Ordered`.

8.0.1. Те же элементы

Не каждая функция, что выводит упорядоченный список элементов, является процедурой сортировки. Нам также нужно описать некоторую связь между входными и выходными элементами. Во-первых, нам нужно, чтобы выходной список состоял из *тех же элементов*, что и входной, пусть и расположенных в другом порядке. Словами это свойство можно выразить так: для любого элемента `p` количество его вхождений во входном списке равно количеству вхождений в выходном списке. Другими словами, *мультимножество* элементов на входе равно мультимножеству элементов на выходе. Мультимножество (иногда называемое *комплектом*) похоже на *множество*, но дополнительно хранит кратность каждого элемента.

В скобках отмечу, что это свойство подпрограмм сортировки часто описывается так: выходные данные являются *перестановкой* входных данных. Если бы мы попытались формализовать это описание, то почти наверняка запутались бы. Итак, хотя понятие «перестановки» предполагает возможное изменение порядка, на самом деле оно означает, что элементы те же самые.

Чтобы задать свойство «те же самые элементы» для процедуры сортировки, можно определить функцию, подсчитывающую количество вхождений данного значения `p` в список:

```
ghost function Count(xs: List<int>, p: int): nat {
  match xs
  case Nil => 0
  case Cons(x, tail) =>
    (if x == p then 1 else 0) + Count(tail, p)
}
```

В таком случае спецификация сортировки будет включать такое свойство, как

```
Count(xs, p) == Count(MySortingFunction(xs), p)
```

для каждого `p`.

Однако вместо `Count` я предложу нечто более общее, о чем расскажу далее.

Упражнение 8.0

Обобщите определение `Count` за пределы целочисленных списков. Какая характеристика типа необходима для типа элемента списка?

8.0.2. Стабильность

Стабильность – это свойство, которым может обладать или не обладать алгоритм сортировки. Это свойство представляет интерес, когда элементы, подлежащие сортировке, имеют ключ, не являющийся целым числом. Например, если вы сортируете список пар `(имя_студента, оценка_за_экзамен)` по оценкам, то при стабильной сортировке любые два студента с одинаковыми оценками будут возвращены в том же порядке, в каком они были перечислены во входном списке. Другими словами, стабильная сортировка сохраняет исходный порядок записей с одним и тем же ключом.

Представьте, что вы решили вывести список пар `(имя_студента, оценка_за_экзамен)` в порядке возрастания оценок, а внутри каждой группы с одинаковыми оценками – в алфавитном порядке по именам студентов. Этого можно добиться, сначала отсортировав список по именам, а затем – применив стабильную сортировку, по оценкам.

В этой главе я описываю сортировку только для списков целых чисел, поэтому ключом сортировки элемента является сам элемент. В этом случае стабильность не имеет значения; например, вы не сможете отличить список `Cons(4, Cons(4, Nil))` от списка `Cons(4, Cons(4, Nil))`. 😁 Тем не менее я определю функции, которые легко адаптировать для списков, где ключи вычисляются из элементов и важна стабильность. Следующая функция выберет элементы, равные (или имеющие ключ) `p`:

```
ghost function Project(xs: List<int>, p: int): List<int> {
  match xs
  case Nil => Nil
  case Cons(x, tail) =>
  if x == p then Cons(x, Project(tail, p)) else Project(tail, p)
}
```

Используя эту функцию `Project`, можно определить стабильность с помощью следующей *формулы стабильности*: для каждого `p`

```
Project(xs, p) == Project(MySortingFunction(xs), p)
```

Обратите внимание, что равенство в этой формуле относится к спискам, а не к числам, как в аналогичной формуле с `Count` выше. Важно отметить также, что стабильность (определенная таким способом) подразумевает также свойство «те же элементы» (см. [79]). Поэтому, хотя в этой главе стабильность не имеет значения, я буду формулировать свойство «те же самые элементы» так, как мы бы сформулировали стабильность, а именно как формулу стабильности, приведенную выше.

Упражнение 8.1

Для любого `p` и любых двух списков, которые после проекции на `p` равны, сформулируйте и докажите лемму, утверждающую, что эти два списка имеют одинаковое количество вхождений `p`.

В оставшейся части этой главы я определю два классических алгоритма сортировки: сортировку вставками и сортировку слиянием. Для каждого алгоритма мы докажем, что его выходные данные отсортированы и содержат те же самые элементы, что и входные. Оба алгоритма, сортировка вставкой и сортировка слиянием, являются стабильными, но мы не можем утверждать этого, поскольку я использую только списки целых чисел. Тем не менее, как я уже упоминал выше, я определю свойство «те же самые элементы» с помощью функции `Project`.

8.1. Сортировка вставками

Идея сортировки вставками проста. Чтобы отсортировать непустой список, нужно отсортировать его хвост, а затем вставить голову в соответствующую позицию. Вот функция, которая это делает:

```
function InsertionSort(xs: List<int>): List<int> {
  match xs
  case Nil => Nil
  case Cons(x, tail) => Insert(x, InsertionSort(tail))
}
```

При вставке вставляемый элемент сравнивается с головой списка, и если голова меньше, то элемент вставляется в хвост, в противном случае – перед головой списка:

```
function Insert(y: int, xs: List<int>): List<int> {
  match xs
  case Nil => Cons(y, Nil)
  case Cons(x, tail) =>
    if y < x then Cons(y, xs) else Cons(x, Insert(y, tail))
}
```

Вот и все. Теперь проверим корректность реализации.

Для начала докажем упорядоченность выходных данных. Как обычно, определим для каждой функции свою лемму. Они будут выглядеть примерно так:

```
lemma InsertionSortOrdered(xs: List<int>)
  ensures Ordered(InsertionSort(xs))

lemma InsertOrdered(y: int, xs: List<int>)
  ensures Ordered(Insert(y, xs))
```

Доказательство первой вызывает вторую, подобно тому, как функция `InsertionSort` вызывает `Insert`:

```
{
  match xs
  case Nil =>
  case Cons(x, tail) =>
    InsertOrdered(x, InsertionSort(tail));
}
```

Верификатору достаточно этого доказательства. Поскольку `Insert` – рекурсивная функция, ожидается, что лемма `InsertOrdered` тоже будет рекурсивной. А поскольку рекурсия довольно проста, то можно надеяться, что Dafny автоматически верифицирует `InsertOrdered`. Однако, если подставить пустое тело доказательства {}, то выяснится, что это не так. Чтобы устранить проблему, начнем заполнять доказательство вручную. В соответствии со стандартной методикой следования за структурой тела функции, корректность которой мы пытаемся доказать, запишем в лемме перечисленные ниже случаи:

```
{
  match xs
  case Nil =>
  case Cons(x, tail) =>
    if y < x { // невозможно доказать постусловие в пути возврата
    } else {
    }
}
```

Как видите, даже когда `Insert` возвращает `Cons(y, xs)` в ветви **then**, верификатор не в состоянии доказать постусловие. И это неудивительно! Нет никаких оснований думать, что `Cons(y, xs)` вернет отсортированный список только потому, что `y < xs.head`, – остальная часть `xs` может содержать элементы в любом порядке. Таким образом, мы должны сообщить лемме больше информации о данном списке `xs`. Давайте сформулируем это свойство, а именно `Sorted(xs)`, как предусловие леммы. Теперь лемма справедлива, и, более того, Dafny доказывает ее автоматически:

```
lemma InsertOrdered(y: int, xs: List<int>)
  requires Ordered(xs)
  ensures Ordered(Insert(y, xs))
{
}
```

Обратите внимание, что сама функция `Insert` не имеет никаких предусловий. В действительности `Insert` просто вставляет `y` после первых элементов в `xs`, которые меньше `y`. Необходимость в предусловии `Sorted(xs)` возникает только при попытке доказать, что `Insert` возвращает отсортированный список.

Теперь осталось только доказать свойство «те же самые элементы». Чтобы сформулировать его, мы должны говорить о произвольном значении того же типа, который имеют элементы списка. Это можно сделать, параметризовав леммы другим параметром, назовем его `p`, как я сделал в формуле стабильности в разделе 8.0.2. Мы формулируем и доказываем леммы для любого такого `p`, поэтому получаем доказательство искомого свойства «те же самые элементы».

```
lemma InsertionSortSameElements(xs: List<int>, p: int)
  ensures Project(xs, p) == Project(InsertionSort(xs), p)

lemma InsertSameElements(y: int, xs: List<int>, p: int)
  ensures Project(Cons(y, xs), p) == Project(Insert(y, xs), p)
```

Первая лемма гласит, что элемент `p` встречается в `InsertionSort(xs)` столько же раз, сколько и во входных данных `xs`, а вторая лемма гласит, что элемент `p` встречается в `Insert(y, xs)` столько же раз, сколько и в `Cons(y , xs)`.

Для доказательства первой леммы просто вызовем вторую, подобно тому, как `InsertionSort` вызывает `Insert`:

```
{
  match xs
  case Nil =>
  case Cons(x, tail) =>
    InsertSameElements(x, InsertionSort(tail), p);
}
```

Лемма `InsertSameElements` доказывается верификатором автоматически:

```
{
}
```

8.2. Сортировка слиянием

На больших объемах данных алгоритм сортировки слиянием работает быстрее алгоритма сортировки вставками. Асимптотически для входных

данных с длиной n сортировка слиянием выполняется за время $O(n \log n)$, а сортировка вставками – за время $O(n^2)$. Разница в скорости обусловлена тем, что в каждом рекурсивном вызове сортировка вставками уменьшает размер списка всего на 1, тогда как сортировка слиянием делит входные данные пополам. В частности, сортировка слиянием сначала разбивает заданный список на две половины, затем сортирует их и, наконец, объединяет отсортированные половины.

8.2.0. Реализация

Чтобы разбить входные данные на две равные (или почти равные) части, нужно знать длину входного списка. Однако, чтобы не вычислять длину списка в каждом рекурсивном вызове, мы вычислим ее вначале, а затем будем отслеживать длины половин. При таком походе основная часть работы в процессе сортировки слиянием будет выполняться вспомогательной функцией, которая принимает длину в качестве параметра. Вот функция верхнего уровня:

```
function MergeSort(xs: List<int>): List<int> {
  MergeSortAux(xs, Length(xs))
}
```

А это вспомогательная функция:

```
function MergeSortAux(xs: List<int>, len: nat): List<int>
  requires len == Length(xs)
```

Если длина списка равна 0 или 1, то он уже отсортирован, поэтому функция просто возвращает входные данные:

```
{
  if len < 2 then
    xs
  else
```

В противном случае нам понадобятся еще две функции: Split и Merge. Давайте сначала напишем эти функции, а затем вернемся к MergeSortAux.

Функция Split разбивает список на две половины и принимает параметр с желаемой длиной префикса:

```
function Split(xs: List, n: nat): (List, List)
  requires n <= Length(xs)
```

Тип результата, возвращаемого функцией Split, – это пара списков, на что указывает тип (List, List). (Если вам интересно, куда пропал параметр типа при List, который не имеет отношения к Split, то см. раздел 6.8.) Чтобы сообщить всем функциям, вызывающим Split, информацию о двух возвращаемых списках, напишем внутреннюю спецификацию в инструкции **ensures**:

```
  ensures var (left, right) := Split(xs, n);
    Length(left) == n &&
    Length(right) == Length(xs) - n &&
    Append(left, right) == xs
```

Это постусловие использует let-выражение, чтобы присвоить имена двум компонентам возвращаемой пары. Тело `Split` определяется следующим образом:

```
{
  if n == 0 then
    (Nil, xs)
  else
    var (l, r) := Split(xs.tail, n - 1);
    (Cons(xs.head, l), r)
}
```

Функция `Merge(xs, ys)` подобна уже знакомой нам функции `Append(xs, ys)` (раздел 6.2), за исключением того, что `Append` всегда берет следующие элементы из `xs` при условии, что `xs` – непустой список. `Merge`, напротив, берет следующий элемент из `ys`, если он меньше. Чтобы реализовать это, мы должны сопоставлять не список `xs`, а пару списков `(xs, ys)`:

```
function Merge(xs: List<int>, ys: List<int>): List<int>
{
  match (xs, ys)
  case (Nil, Nil) => Nil
  case (Cons(_, _), Nil) => xs
  case (Nil, Cons(_, _)) => ys
  case (Cons(x, xtail), Cons(y, ytail)) =>
    if x <= y then
      Cons(x, Merge(xtail, ys))
    else
      Cons(y, Merge(xs, ytail))
}
```

Определив `Split` и `Merge`, вернемся к `MergeSortAux` и заполним ветвь **`else`**, на которой мы остановились:

```
    var (left, right) := Split(xs, len / 2);
    Merge(MergeSortAux(left, len / 2),
          MergeSortAux(right, len - len / 2))
}
```

Это выражение разбивает `xs` на две половины, `left` и `right`, длина которых равна `len/2` и `len - len/2` соответственно. Два рекурсивных вызова `MergeSortAux` сортируют эти меньшие списки, а затем отсортированные списки объединяются.

Здесь верификатор сообщает, что не может доказать завершимость ни одного из рекурсивных вызовов. Мы организовали выполнение этих вызовов со списками с длиной `len/2` и `len - len/2`, каждый из которых действительно имеет длину меньше `len`, потому что рекурсивные вызовы выполняются в ветви, где `2 <= len`. Но для `MergeSortAux` используется оценочная функция по умолчанию – лексикографическая пара аргументов функции: `xs,n`. Было бы ошибкой считать, что `left` структурно включен в `xs`, поэтому у верификатора есть веские причины обратить наше внимание на первый рекурсивный вызов. (Во втором вызове `right` структурно включен в `xs`, но, чтобы сообщить об этом верификатору, нам придется доказать лемму.) Выход прост: мы вручную определим оценочную функцию на `len`, добавив следующую инструкцию после сигнатуры `MergeSortAux`:

```
decreases len
```

Упражнение 8.2

Целочисленный параметр функции `Split` позволяет увидеть происходящее. Чтобы использовать текущую реализацию функции `Split`, мы должны предварительно вычислить длину исходного списка (как мы это делали при вызове `MergeSortAux` из `MergeSort`). Однако нет необходимости добавлять это число в оценочную функцию, потому что длину списка можно определить, подсчитав его элементы. Реализуйте функцию со следующей спецификацией:

```
function Split'(xs: List, nn: List): (List, List)
  requires Length(nn) <= Length(xs)
  ensures var (left, right) := Split'(xs, nn);
    var n := Length(nn) / 2;
    Length(left) == n &&
    Length(right) == Length(xs) - n &&
    Append(left, right) == xs
```

(Последние три строки совпадают с последними строками в постусловии `Split`.)

Упражнение 8.3

Напишите новую функцию `MergeSort'(xs: List<int>)`, реализованную непосредственно в терминах `Split'` (из упражнения 8.2) и `Merge`. Докажите, что `MergeSort(xs) == MergeSort'(xs)`. Подсказка: для доказательства вам пригодится лемма `AppendDecomposition` из упражнения 6.5.

8.2.1. Корректность упорядоченности

Корректность сортировки слиянием доказывается в два этапа, как и сортировки вставками. Для начала покажем, что выходные данные `MergeSort`

упорядочены. В нашей реализации есть четыре функции (`MergeSort`, `MergeSortAux`, `Split` и `Merge`), поэтому разумно ожидать, что потребуется написать четыре леммы. Однако, как оказывается, достаточно трех, потому что основные свойства `Split` мы определили во внутренней спецификации.

Первые две леммы повторяют структуру соответствующих функций:

```
lemma MergeSortOrdered(xs: List<int>)
  ensures Ordered(MergeSort(xs))
{
  MergeSortAuxOrdered(xs, Length(xs));
}

lemma MergeSortAuxOrdered(xs: List<int>, len: nat)
  requires len == Length(xs)
  ensures Ordered(MergeSortAux(xs, len))
  decreases len
{
  if 2 <= len {
    var (left, right) := Split(xs, len / 2);
    MergeOrdered(MergeSortAux(left, len / 2),
                 MergeSortAux(right, len - len / 2));
  }
}
```

Лемма, которая показывает, что выходные данные `Merge` должны быть упорядочены, справедлива только в том случае, если входные данные `Merge` уже отсортированы, поэтому для леммы также нужно указать предусловие. Доказательство выполняется автоматически:

```
lemma MergeOrdered(xs: List<int>, ys: List<int>)
  requires Ordered(xs) && Ordered(ys)
  ensures Ordered(Merge(xs, ys))
{
}
```

Упражнение 8.4

Нам пришлось вручную написать часть доказательства `MergeSortAuxOrdered`, но мы не вызывали индуктивное предположение явно. Добавьте в лемму атрибут {`:induction` **`false`**} и вручную напишите необходимые вызовы.

Упражнение 8.5

Добавьте в лемму `MergeOrdered` атрибут {`:induction` **`false`**} и напишите ее доказательство, не полагаясь на автоматическую индукцию Dafny.

8.2.2. Те же самые элементы

Последнее проверяемое условие, которое, напомню, имеет вид формулы стабильности, определенной в разделе 8.0.2, требует дополнительной работы. Начало выглядит просто и следует структуре `MergeSort`, которая вызывает вспомогательную функцию:

```
lemma MergeSortSameElements(xs: List<int>, p: int)
  ensures Project(xs, p) == Project(MergeSort(xs), p)
{
  MergeSortAuxSameElements(xs, Length(xs), p);
}

lemma MergeSortAuxSameElements(xs: List<int>, len: nat, p: int)
  requires len == Length(xs)
  ensures Project(xs, p) == Project(MergeSortAux(xs, len), p)
  decreases len
```

Мы должны помочь верификатору доказать эту лемму, поэтому начнем как обычно и, следуя структуре соответствующей функции `MergeSortAux`, с более сложной стороны равенства, которое требуется доказать:

```
{
  if 2 <= len {
    var (left, right) := Split(xs, len / 2);
    calc {
      Project(MergeSortAux(xs, len), p);
    ==  // определение MergeSortAux
      Project(Merge(MergeSortAux(left, len / 2),
              MergeSortAux(right, len - len / 2)), p);
```

(Следующий шаг – самый трудный, поэтому приготовьтесь к более долгим рассуждениям. После этого шаги снова станут попроще.)

Чтобы упростить выражение `Project(..., p)`, нужно некоторое свойство `Project(Merge(...), p)`. Переключим внимание на такую лемму. Что *можно* сказать об элементах `p` слияния? Эти элементы извлекаются из двух объединяемых списков. Давайте дадим двум объединяемым и отсортированным спискам имена `xs` и `ys` и подумаем, что делает `Merge(xs, ys)`. Слияние происходит путем многократного выбора очередного элемента из `xs` или `ys` и добавления его в результат. Эта операция сначала выполняется для всех элементов, которые меньше `p`, а уже потом для любых элементов, равных или больше `p`. Слияние выполняется сначала для всех элементов, равных `p`, и только потом для любых элементов больше `p`. Следовательно, эффект проецирования результата слияния на элементы, равные `p`, сродни игнорированию элементов, отличных от `p`, в двух списках. Другими словами, мы ожидаем, что `Project` будет обладать свойством дистрибутивности по `Merge`:

```
Project(Merge(xs, ys), p) ==
Merge(Project(xs, p), Project(ys, p))
```

Кроме того, слияние двух списков элементов `p` равносильно добавлению одного из них в конец другого, что приводит нас к следующей лемме:

```
lemma MergeSameElements(xs: List<int>, ys: List<int>, p: int)
  requires Ordered(xs) && Ordered(ys)
  ensures Project(Merge(xs, ys), p)
      == Append(Project(xs, p), Project(ys, p))
{
}
```

Эта лемма доказывается автоматически.

Далее нам нужно применить эту лемму к вычислениям в лемме `MergeSortAuxSameElements`, на которой мы остановились. То есть нам нужно сделать следующий шаг:

```
      Project(Merge(MergeSortAux(left, len / 2),
                    MergeSortAux(right, len - len / 2)),
              p);
    ==  { MergeSameElements(
            MergeSortAux(left, len / 2),
            MergeSortAux(right, len - len / 2),
            p);
        }
    Append(
      Project(MergeSortAux(left, len / 2), p),
      Project(MergeSortAux(right, len - len / 2), p));
```

Но верификатор сообщает, что мы можем вызвать `MergeSameElements` только для отсортированных списков. Получить недостающую информацию можно вызовом леммы `MergeSortAuxOrdered`, которую доказали в разделе 8.2.1. Добавим два вызова этой леммы перед вызовом `MergeSameElements` внутри подсказки. Итак, наш следующий долгожданный шаг в `MergeSortAuxSameElements`:

```
      Project(Merge(MergeSortAux(left, len / 2),
                    MergeSortAux(right, len - len / 2)),
              p);
    ==  { MergeSortAuxOrdered(left, len / 2);
          MergeSortAuxOrdered(right, len - len / 2);
          MergeSameElements(
            MergeSortAux(left, len / 2),
            MergeSortAux(right, len - len / 2),
            p);
        }
      Append(
```

```
        Project(MergeSortAux(left, len / 2), p),
        Project(MergeSortAux(right, len - len / 2), p));
```

Здесь у нас есть два подвыражения, проецирующие результат `MergeSortAux`, поэтому можно применить индуктивное предположение:

```
    ==  { MergeSortAuxSameElements(left, len / 2, p);
          MergeSortAuxSameElements(right, len - len / 2, p); }
      Append(Project(left, p), Project(right, p));
```

Предварительное проецирование `left` и `right` на `p` и последующее их объединение дает тот же результат, что и предварительное объединение `left` и `right` и последующее проецирование на `p`, что мы доказываем отдельно с помощью другой леммы:

```
lemma AppendProject(xs: List<int>, ys: List<int>, p: int)
  ensures Append(Project(xs, p), Project(ys, p))
       == Project(Append(xs, ys), p)
{
}
```

Добавлением и использованием этой леммы мы завершаем доказательство `MergeSortAuxSameElements`:

```
    ==  { AppendProject(left, right, p); }
      Project(Append(left, right), p);
    ==
      Project(xs, p);
    }
  }
}
```

Уф-ф! Теперь мы доказали корректность нашей функции `MergeSort`.

Упражнение 8.6

Обычно в реализациях алгоритмов сортировки $O(n \log n)$ для сортировки коротких списков используют более простой алгоритм сортировки $O(n^2)$. Это может дать прирост производительности, потому что алгоритмы $O(n \log n)$ имеют значительные накладные расходы на рекурсивные вызовы. Измените функцию сортировки слиянием, представленную выше, так, чтобы она вызывала сортировку вставками для коротких входных списков. В частности, замените `if len < 2 then xs` в `MergeSortAux` на

```
if len < 8 then InsertionSort(xs)
```

Убедитесь, что новая программа сортировки слиянием выводит те же элементы, что получает на входе, но в упорядоченном виде.

8.3. Итоги

В этой главе мы рассмотрели задачу сортировки списков. Сначала мы написали общие спецификации алгоритмов сортировки, а затем применили их к сортировке вставками и сортировке слиянием.

Спецификация сортировки состоит из двух частей: выходные данные упорядочены и содержат те же элементы, что и входные данные. Свойство *стабильности* дополнительно говорит о том, что элементы с равными ключами на выходе располагаются в том же порядке, что и на входе.

В инженерном деле очень важно правильно составить спецификации. С помощью программных доказательств и автоматической верификации вы можете убедиться, что реализация соответствует спецификации. Формулировка спецификации влияет на ее удобочитаемость и простоту понимания человеком, а также на сложность доказательств и на способность верификатора автоматически доказать ее. В этой главе я сформулировал спецификацию «те же самые элементы» с точки зрения сохранения мультимножества элементов списка. Эта спецификация не только более простая для понимания, но также позволяет избежать сложностей при попытке определить, что такое перестановка.

Примечания

Галерея верифицированных программ, доступная на домашней странице Why3 [20], содержит множество алгоритмов, сортирующих заданный список `List`. (Здесь также имеются программы, сортирующие изменяемые массивы, о которых мы поговорим в главе 15.) В литературе можно найти множество «жемчужин доказательств» (коротких статей, демонстрирующих что-то интересное в доказательствах) алгоритмов сортировки. Например, алгоритм естественной сортировки слиянием, который используется в стандартной библиотеке Haskell, был верифицирован в Isabelle/HOL [119] и Dafny [79].

В главе 15, где обсуждаются императивные алгоритмы сортировки, вы найдете дополнительные ссылки и исторические заметки.

Глава 9

Абстракция

Абстракция и сокрытие информации имеют решающее значение для корректности дизайна программы. Они позволяют управлять сложностью взаимодействующих частей программы. До сих пор я приводил лишь небольшие примеры, показывающие, как рассуждать об алгоритмах и как писать доказательства. В них не было никакой необходимости в абстракции. В этой главе я покажу, как объединить тип и различные операции над этим типом в *модуль* и как провести границу между тем, что клиентам модуля разрешено знать, и тем, что должно оставаться скрытым от их глаз (детали реализации). Глава иллюстрирует эти организационные принципы с помощью модуля, реализующего очередь.

9.0. Группировка объявлений в модули

Одно из основных предназначений модуля – группировать связанные объявления. Например, в главе 6 мы определили тип `List` вместе с операциями над списками и леммами об этих операциях. Мы можем поместить эти объявления в модуль с названием `ListLibrary`.

```
module ListLibrary {
  datatype List<T> = Nil | Cons(head: T, tail: List<T>)

  function Append(xs: List, ys: List): List
  // ...

  lemma AppendAssociative(xs: List, ys: List, zs: List)
    ensures Append(Append(xs, ys), zs) == Append(xs, Append(ys, zs))
  // ...
```

```
  // множество других объявлений...
}
```

Однако модули не только группируют связанные объявления, но также разграничивают *пространство имен* программы. Это означает, что в разных модулях можно использовать одни и те же идентификаторы. Например, модуль `StackLibrary` тоже может объявить функцию (или тип, или лемму, или что-то еще) с именем `Append`. Чтобы уточнить, о каком именно идентификаторе `Append` идет речь, перед ним добавляется имя модуля и точка (`.`). Например, чтобы обратиться к функции `Append` из модуля `ListLibrary` из-за пределов модуля `ListLibrary`, мы пишем `ListLibrary.Append`.

Два объявления, следующие друг за другом, т. е. в одной *области видимости*, называются *одноуровневыми*. Предположим, что существует метод `Main`, одноуровневый с модулем `ListLibrary`. В таком случае этот метод сможет использовать имя `ListLibrary`. Более того (и что особенно интересно), метод может ссылаться на объявления в `ListLibrary`, используя полные имена элементов в `ListLibrary`, как иллюстрирует следующий пример:

```
method Main() {
  var xs, i := ListLibrary.Nil, 0;
  while i < 5 {
    xs, i := ListLibrary.Cons(i, xs), i + 1;
  }
  print xs, «\n»;
}
```

9.1. Импорт модулей

Модуль не может произвольно ссылаться на другие модули. Точнее, имена одноуровневых модулей недоступны автоматически. Например, следующая программа не является допустимой:

```
module ModuleA {
  function Plus(x: int, y: int): int {
    x + y
  }
}

module ModuleB {
  function Double(x: int): int {
    ModuleA.Plus(x, x) // ошибка: неизвестный идентификатор ‘ModuleA'
  }
}
```

Чтобы сделать имя модуля `ModuleA` доступным в соседнем с ним модуле `ModuleB`, последний должен *импортировать* первый, использовав объявление **`import`**:

```
module ModuleB {
  import ModuleA
  function Double(x: int): int {
    ModuleA.Plus(x, x)
  }
}
```

В более общем смысле объявление **import** дает импортирующему модулю возможность дать другое локальное имя для ссылки на импортируемый модуль. Например, в следующем определении модуля:

```
module ModuleB {
  import A = ModuleA
  function Double(x: int): int {
    A.Plus(x, x)
  }
}
```

объявление **import** вводит локальное имя A внутри ModuleB и связывает его с модулем ModuleA. Фактически более простое объявление **import** ModuleA – это лишь сокращенная форма объявления **import** ModuleA = ModuleA. (Обратите внимание, что в объявлении **import** справа от знака «=» допускается указывать идентификаторы, не входящие в область видимости внутри модуля.)

Импортирование модулей в программе не должно порождать циклических зависимостей. Другими словами, объявления **import** в допустимой программе должны образовывать линейную иерархию модулей. Например, если ModuleB импортирует ModuleA, то ModuleA не может импортировать ModuleB.

9.2. Экспортируемые наборы

Модули не просто разграничивают пространство имен программы. Они облегчают *сокрытие информации*, делая определенные объявления или части объявлений недоступными за пределами модуля. Скрывая некоторые части своих объявлений, модуль получает возможность принимать независимые решения по реализации и свободно менять их в будущих версиях модуля, не влияя на работу клиентов модуля.

Части модуля, доступные за его пределами, в Dafny определяется *экспортируемыми наборами*. Они определяются объявлением **export**. Если модуль не содержит объявления **export**, то все его содержимое становится доступным импортерам. Например, модуль ListLibrary в разделе 9.0 и ModuleA в разделе 9.1 по умолчанию экспортируют все свое содержимое.

Почти каждое объявление состоит из *сигнатуры* и *тела*. Сигнатура включает имя, ее тип и список параметров, а также любые определения спецификаций. Тело – это остальная часть объявления и обычно состоит из выражения или оператора внутри фигурных скобок или некоторого определения, следующего за знаком =.

Например, рассмотрим следующие два объявления:

```
datatype Color = Blue | Yellow | Green | Red
function Double(x: int): nat
  requires 0 <= x
  ensures Double(x) % 2 == 0
{
  if x == 0 then 0 else 2 + Double(x - 1)
}
```

Сигнатура типа `Color` – это просто имя `Color`, а сигнатура функции `Double` – это ее имя, тип, список параметров, предусловие и постусловие. Тело `Color` – это то, что следует за знаком «=», а тело `Double` – выражение в фигурных скобках.

Механизм экспорта в Dafny позволяет явно управлять экспортированием сигнатуры и тела объявления. В частности, в разделе **`provides`** объявления **`export`** перечисляются имена объявлений, которые экспортируют только сигнатуры, а в разделе **`reveals`** – имена объявлений, которые экспортируют сигнатуры и тела.

Для иллюстрации рассмотрим следующие модули:

```
module ModuleC {
  export
    provides Color
    reveals Double
  datatype Color = Blue | Yellow | Green | Red
  function Double(x: int): nat
    requires 0 <= x
    ensures Double(x) % 2 == 0
  {
    if x == 0 then 0 else 2 + Double(x - 1)
  }
}

module ModuleD {
  import ModuleC
  method Test() {
    var c: ModuleC.Color;
    c := ModuleC.Yellow; // ошибка: неизвестный идентификатор 'Yellow'
    assert ModuleC.Double(3) == 6;
  }
}
```

Экспортируемый набор `ModuleC` включает сигнатуру типа `Color` и сигнатуру с телом функции `Double`. Модуль `ModuleD` импортирует `ModuleC`, поэтому он может использовать тип `ModuleC.Color`. В примере этот тип используется в объявлении локальной переменной `c`. Однако, поскольку `ModuleC` экс-

портирует только сигнатуру `Color`, конструкторы `Color` недоступны в `ModuleD`, поэтому конкретный оператор присваивания в `ModuleD` вызывает ошибку. Фактически `ModuleD` даже не знает, что `Color` – это индуктивный тип данных. Ему известно лишь, что `Color` – это тип. С другой стороны, экспортируемый набор в `ModuleC` включает и сигнатуру, и тело `Double`, поэтому `ModuleD` может не только ссылаться на эту функцию, но и рассуждать о ее значении. В частности, в `ModuleD` можно верифицировать утверждение **`assert`**.

Экспортируемый набор должен обеспечивать автономное представление модуля: если удалить то, что не экспортируется, то останется программа, которая по-прежнему выполняет проверки типов. Во-первых, все упомянутые идентификаторы должны иметь доступные объявления. Для иллюстрации рассмотрим следующий модуль:

```
module ModuleE {
  datatype Parity = Even | Odd
  function F(x: int): Parity
  {
    if x % 2 == 0 then Even else Odd
  }
}
```

Предположим, что экспортируемый набор этого модуля, объявленный ниже, включает лишь сигнатуру функции `F`:

```
export provides F // несогласованный экспортируемый
                  // набор для ModuleE
```

Это некорректное объявление, потому что после удаления всего, что не экспортируется, остается только

```
function F(x: int): Parity
```

В результате тип `Parity` окажется неизвестным для импортирующих модулей. Другими словами, для модуля, импортирующего `ModuleE`, объявление `F` покажется ошибочным, потому что в нем упоминается тип `Parity`, объявление которого не известно импортеру. Аналогично объявление **`export`**:

```
export provides Parity reveals F // несогласованный экспортируемый
                                 // набор для ModuleE
```

тоже будет некорректным. Вот что будет доступно из этого экспортируемого набора:

```
type Parity
function F(x: int): Parity
{
  if x % 2 == 0 then Even else Odd
}
```

Здесь `Even` и `Odd` окажутся неизвестными идентификаторами. Теперь приведу пример согласованного экспортируемого набора для `ModuleE`:

```
export
  provides Parity, F
```

Он делает доступными следующие объявления:

```
type Parity
function F(x: int): Parity
```

Еще один согласованный экспортируемый набор:

```
export
  reveals Parity, F
```

Он делает доступным импортерам все, что касается типа `Parity` и функции `F`. Поскольку это единственные два объявления в модуле, объявление **`export`** также можно записать так:

```
export
  reveals *
```

Оно определяет экспортируемый набор, включающий все содержимое модуля, как и при полном отсутствии объявления **`export`**.

9.3. Модульная спецификация очереди

Давайте попробуем применить прием сокрытия информации к реализации неизменяемой очереди. *Очередь* – это список элементов, поддерживающий три операции: *инициализация* создает пустую очередь, *постановка в очередь* добавляет элемент в конец очереди, а *удаление из очереди* удаляет и возвращает элемент из начала очереди. Рассматриваемые нами очереди являются *неизменяемыми*, поэтому каждая операция возвращает значение очереди, а не изменяет какую-то существующую очередь.

9.3.0. Базовый интерфейс очереди

Вот самый простой модуль, описывающий очередь:

```
module ImmutableQueue {
  type Queue<A>
  function Empty(): Queue
  function Enqueue<A>(q: Queue, a: A): Queue
  function Dequeue<A>(q: Queue): (A, Queue)
    requires q != Empty()
}
```

Это еще не полный модуль – мы продолжим конструировать его шаг за шагом. Здесь я показал, как этот модуль будут видеть его клиенты (т. е.

пользователи модуля). Такое представление модуля называется *интерфейсом*, поскольку оно описывает, как клиенты модуля и его реализация взаимодействуют друг с другом.

Этот интерфейс модуля объявляет тип `Queue`, параметризованный некоторым типом `A` элемента. Модуль не раскрывает никакой информации об этом типе, кроме имени и количества параметров типа. Это хорошая практика сокрытия информации, поскольку она дает свободу реализации представления значений типа. Мы называем это *абстрагированием* от реализации. Это просто означает возможность говорить об очередях в более абстрактных терминах высокого уровня.

Три операции объявлены как функции. Поскольку операцию удаления из очереди нельзя применить к пустой очереди, у нее есть предусловие.

9.3.1. Функции абстракции

Базовый интерфейс очереди, приведенный выше, слишком прост, потому что не предоставляет клиенту информации для верификации своих операций с очередью. Например, исходя из базового интерфейса, клиент не в состоянии определить – может ли операция `Enqueue` вернуть пустую очередь, что затрудняет вызов `Dequeue`. Чтобы описать операции более подробно, будет полезно поговорить о том, что на самом деле представляет собой очередь.

Очередь состоит из списка элементов. Рассуждения как о списке позволят нам составить полезные и понятные спецификации. Это так просто! Давайте введем функцию, превращающую очередь в список ее элементов:

```
ghost function Elements(q: Queue): List
```

Функция объявлена призрачной, потому что она будет использоваться только в спецификациях. Сама очередь тоже может быть реализована как список, но факт наличия этой функции устраняет ограничение, требующее реализовать очередь как список. Мы говорим, что эта функция *абстрагирует* реализацию и что эта функция называется *функцией абстракции*, потому что она позволяет нам думать об очереди на более высоком уровне, чем ее конечная реализация.

Какой бы модуль вы ни проектировали, всегда полезно подумать о том, какую абстракцию он предоставляет. Это добавит вам ясности, когда вы углубитесь в детали реализации.

Используя функцию абстракции `Elements`, операции с очередью можно определить в терминах списков элементов. Спецификации могут быть заданы внутренне (в постусловиях функций) или внешне (в отдельных леммах). Давайте определим их внешне. Вот обновленный интерфейс модуля:

```
module ImmutableQueue {
  import LL = ListLibrary

  type Queue<A>
  function Empty(): Queue
  function Enqueue<A>(q: Queue, a: A): Queue
  function Dequeue<A>(q: Queue): (A, Queue)
    requires q != Empty()

  ghost function Elements(q: Queue): LL.List
  lemma EmptyCorrect<A>()
    ensures Elements(Empty<A>()) == LL.Nil
  lemma EnqueueCorrect<A>(q: Queue, x: A)
    ensures Elements(Enqueue(q, x)) == LL.Snoc(Elements(q), x)
  lemma DequeueCorrect(q: Queue)
    requires q != Empty()
    ensures var (a, q') := Dequeue(q);
      LL.Cons(a, Elements(q')) == Elements(q)
}
```

Чтобы модуль мог использовать списки, он импортирует модуль `ListLibrary`. Мы часто будем ссылаться на объявления в `ListLibrary`, поэтому разумно дать импортируемому модулю более короткое имя. По этой причине в объявлении **`import`** задано локальное имя `LL`, обозначающее импортируемую библиотеку.

В ссылках на объявления из модуля `ListLibrary` модуль `ImmutableQueue` использует префикс «`LL.`». Например, внутри модуля `ListLibrary` конструктор типа данных `Cons` может вызвать просто по имени `Cons` (поскольку это имя не конфликтует с другими объявлениями в `ListLibrary`) или как `List.Cons` или даже `List<B>.Cons`. Ссылки на этот конструктор в `ImmutableQueue` должны оформляться так: `LL.Cons`, `LL.List.Cons` или `LL.List<B>.Cons`.

Я решил объявить функцию `Elements` призрачной, потому что собираюсь использовать ее только в спецификациях. Это типично для функций абстракции. Но если тело функции должно компилироваться и быть доступным во время выполнения, то просто удалите ключевое слово **`ghost`** из объявления функции.

Упражнение 9.0

Функция `Elements` предоставляют абстракцию реализации очереди. Еще одной функцией абстракцией могла бы служить функция

```
ghost function Length(q: Queue): nat
```

возвращающая количество элементов в очереди. Напишите интерфейс модуля, который использует `Length` вместо `Elements`.

9.3.2. Объявление экспортируемого набора

Мы пока не объявили экспортируемый набор в нашем модуле. Поэтому по умолчанию Dafny экспортирует все. Это неплохо, когда в модуле присутствуют только объявления без тела. Но к моменту завершения модуля мы определим все типы, функции и леммы, и для нас было бы нежелательно раскрывать эти определения. Явно определив экспортируемый набор, мы можем контролировать представление модуля с точки зрения клиентов.

Имена объявлений, которые должны быть доступны за пределами модуля, мы определяем в инструкции **`provides`** в объявлении **`export`**. Чтобы обеспечить автономность экспортируемого набора, мы должны также экспортировать клиентам и библиотеку `ListLibrary`. Для этого укажем ее локальное имя `LL` в объявлении **`export`**.

Вот объявление **`export`**, которое мы добавим в модуль:

```
export
  provides Queue, Empty, Enqueue, Dequeue
  provides LL, Elements
  provides EmptyCorrect, EnqueueCorrect, DequeueCorrect
```

Поскольку на данный момент это все имена, объявленные в модуле, мы могли бы написать

```
export provides *
```

Однако такое объявление экспортирует также другие вспомогательные функции и леммы, которые мы можем добавить позже.

Еще одно замечание, касающееся экспорта импортированного модуля `ListLibrary`. Через экспортированное имя `LL` клиентский модуль, импортирующий `ImmutableQueue` под именем `IQ`, сможет ссылаться на тип `List` как `IQ.LL.List<B>` для любого типа `B`. Клиент также может сам импортировать модуль `ListLibrary`, например так:

```
import MyOwnLLImport = ListLibrary
```

а затем использовать `IQ.LL.List<B>` и `MyOwnLLImport.List<B>` как синонимы.

9.3.3. Пример клиента

Теперь, имея интерфейс модуля, проверим, сможет ли его использовать простой клиент. Наш клиент создает пустую очередь, добавляет в нее один элемент, а затем удаляет этот элемент из очереди:

```
module QueueClient {
  import IQ = ImmutableQueue
  method Client() {
    var q := IQ.Empty();
    q := IQ.Enqueue(q, 20);
```

```
    var (a, q') := IQ.Dequeue(q); // нарушает предусловие
  }
}
```

Верификатор сообщает, что вызов `Dequeue` может нарушить его предусловие. Мы предвидели это раньше, поэтому и ввели леммы. Давайте воспользуемся леммами – по одной для каждого вызова функции (уф-ф!) – и добавим в конце пару утверждений **`assert`**, чтобы убедиться, что в итоге получаем ожидаемые результаты:

```
module QueueClient {
  import IQ = ImmutableQueue
  method Client() {
    IQ.EmptyCorrect<int>();   var q := IQ.Empty();
    IQ.EnqueueCorrect(q, 20); q := IQ.Enqueue(q, 20);
    IQ.DequeueCorrect(q);     var (a, q') := IQ.Dequeue(q);
    assert a == 20;
    assert q' == IQ.Empty(); // утверждение не может быть верифицировано
  }
}
```

Теперь можно верифицировать предусловие `Dequeue` и убедиться в полезности наших лемм. Но они делают не все, что нам хотелось бы, как следует из сообщения верификатора для последнего утверждения **`assert`**.

Чтобы исправить ситуацию, можно попытаться предварить утверждение, вызывающее ошибку, следующим утверждением:

```
assert IQ.Elements(q') == IQ.LL.Nil;
```

Это утверждение принимается верификатором! Итак, если мы знаем, что `q'` – это пустой список элементов, то почему не проходит верификация сравнения списка с `IQ.Empty()`, который (как сообщает нам `EmptyCorrect()`) тоже представляет пустой список элементов?

Проблема в том, что может существовать несколько значений типа `Queue<int>`, представляющих пустой список элементов. Действительно, в зависимости от структуры данных, используемой в реализации `ImmutableQueue`, может существовать множество значений `Queue<int>`, представляющих один и тот же список элементов. В общем случае проверка каноничности всех значений типа, т. е. что каждое из них представляет отдельное абстрактное значение, может оказаться очень дорогостоящей в вычислительном отношении.

Мы должны принять решение. С одной стороны, мы можем решить, что клиент никогда не должен сравнивать значение `Queue` напрямую и всегда прибегать к сравнениям через функцию абстракции `Elements`. С другой стороны, мы можем решить, что реализация должна позволять сравнивать значения `Queue` напрямую, что, по сути, сводится к тому, чтобы убедиться, что функция `Elements` является *инъективной* (т. е. что `Elements` является од-

нозначной функцией). Мы остановимся на промежуточном выборе между этими двумя крайностями, а именно на том, чтобы сделать элементы инъективными для пустой очереди. Другими словами, мы сделаем `Empty()` единственным значением `Queue`, которое представляет пустой список элементов. Для этого добавим еще одну лемму в `ImmutableQueue`:

```
lemma EmptyUnique(q: Queue)
  ensures Elements(q) == LL.Nil ==> q == Empty()
```

и (не забудьте!) `EmptyUnique` в инструкцию **`provides`** объявления **`export`** модуля. Как обычно, свойство этой леммы можно выразить несколькими способами (см. обсуждение `LessTransitive` в разделе 7.1). Здесь я решил использовать импликацию в постусловии вместо посылки в предусловии. Это дает возможность использовать противопоставление леммы. То есть, вызывая лемму для очереди `q`, которая, как известно, не является `Empty()`, вызывающая сторона может прийти к выводу, что `Elements(q) != LL.Nil`.

Теперь наш простой клиент верифицируется:

```
module QueueClient {
  import IQ = ImmutableQueue
  method Client() {
    IQ.EmptyCorrect<int>();   var q := IQ.Empty();
    IQ.EnqueueCorrect(q, 20); q := IQ.Enqueue(q, 20);
    IQ.DequeueCorrect(q);     var (a, q') := IQ.Dequeue(q);
    assert a == 20;
    IQ.EmptyUnique(q');
    assert q' == IQ.Empty();
  }
}
```

Этот код выполняет свою работу, но выглядит беспорядочным. Мы поработаем над уменьшением беспорядка в разделе 10.3, а пока приступим к реализации `ImmutableQueue`.

Упражнение 9.1

Напишите клиентский модуль (например, `QueueClient`), чтобы протестировать интерфейс модуля в упражнении 9.0. Настройте интерфейс очереди так, чтобы ваш клиентский модуль благополучно верифицировался.

9.3.4. Реализация очереди

Вернемся к модулю `ImmutableQueue`. Чтобы завершить его, нам нужно определить тип `Queue`, четыре функции и четыре леммы.

Простейшим решением было бы использовать в реализации нашей очереди список, который возвращает `Elements`. Однако стоимость добавления элемента в конец списка, как это делает операция `Enqueue`, пропорциональ-

на длине очереди. Намного эффективнее было бы использовать два списка, назовем их `front` и `rear`. Часть элементов в начале очереди будет храниться в списке `front`, откуда их можно получать по очереди за постоянное время. Остальные элементы будут храниться в списке `rear` *в обратном порядке*, что позволит добавлять их в очередь тоже за постоянное время. Обнаружив, что список `front` пустой, операция `Dequeue` скопирует элементы в обратном порядке из `rear` в `front`. Следовательно, для извлечения каждого элемента из очереди будут выполняться две операции с постоянным временем и для всех элементов будет выполнена одна операции копирования с перестановкой в обратном порядке. Как мы видели в разделе 6.6, перестановка элементов списка в обратном порядке требует линейного времени, соответственно, `Dequeue` будет иметь *амортизированное* постоянное время выполнения. Это означает, что по мере развития очереди сумма времени выполнения операций `Enqueue`, создающих очередь, будет кратна произведению константы на число вызовов `Dequeue`.

Начнем реализацию с определения типа структуры данных и заменим объявление **`type`** `Queue<A>` следующим:

```
datatype Queue<A> = FQ(front: LL.List<A>, rear: LL.List<A>)
```

Обратите внимание, что, поскольку наш модуль экспортирует *только сигнатуру* типа `Queue`, тот факт, что `Queue` определен как тип данных (**`datatype`**), остается частным решением реализации модуля.

Основная идея этой реализации очереди отражена в определении `Elements`:

```
ghost function Elements(q: Queue): LL.List {
  LL.Append(q.front, LL.Reverse(q.rear))
}
```

Остальные функции тоже имеют простые определения:

```
function Empty(): Queue {
  FQ(LL.Nil, LL.Nil)
}

function Enqueue<A>(q: Queue, x: A): Queue {
  FQ(q.front, LL.Cons(x, q.rear))
}

function Dequeue<A>(q: Queue): (A, Queue)
  requires q != Empty()
{
  match q.front
  case Cons(x, front') =>
    (x, FQ(front', q.rear))
  case Nil =>
```

```
    var front := LL.Reverse(q.rear);
    (front.head, FQ(front.tail, LL.Nil))
}
```

Леммы `EmptyCorrect` и `EmptyUnique` доказываются автоматически – нам нужно лишь указать пустое тело {}, поэтому я не буду их здесь рассматривать. Доказательства двух других лемм послужит вам хорошим самостоятельным упражнением, поэтому оставлю их доказательство вам. 😁

Упражнение 9.2

Напишите доказательство для `EnqueueCorrect`.

Упражнение 9.3

Напишите доказательство для `DequeueCorrect`.

Упражнение 9.4

а. Напишите модуль `BlockChain`, объявляющий тип `Ledger` с тремя операциями, имеющими следующие сигнатуры:

```
function Init(): Ledger
function Deposit(ledger: Ledger, n: nat): Ledger
function Withdraw(ledger: Ledger, n: nat): Ledger
```

Напи1шите функцию абстракции

```
ghost function Balance(ledger: Ledger): int
```

возвращающую баланс бухгалтерского реестра, и добавьте в `Withdraw` предусловие `n <= Balance(ledger)`. Объявите экспортируемый набор для модуля.

б. Напишите внутренние спецификации функций в терминах `Balance`. Напишите простой клиент и протестируйте корректность поведения всех этих функций.

в. Реализуйте модуль со списком целых чисел. В частности, определите `Ledger` как синоним типа:

```
type Ledger = List<int>
```

где `List` – обычный тип списка. Реализации `Deposit` и `Withdraw` должны добавлять в начало списка значение (неотрицательное или неположительное соответственно) второго аргумента.
(Обратите внимание, что параметр `n` в `Deposit` и `Withdraw` имеет тип **`nat`**, тогда как `Balance` возвращает **`int`**. Чтобы объявить функцию `Balance` как возвращающую значение типа **`nat`**, понадобится инвариант структуры данных, о котором я расскажу в главе 10, см. упражнение 10.5.)

9.4. Типы, поддерживающие оператор равенства

Еще одна операция, которая может понадобиться клиентам нашей неизменяемой очереди и которую, соответственно, мы должны экспортировать: проверка на равенство. Если вас это тема не интересует, то можете пропустить данный раздел, поскольку в нем рассматривается ряд мелких особенностей языка Dafny.

9.4.0. Равенство

В Dafny каждый тип имеет встроенный оператор равенства (==). Например, равенство целых чисел (**int**) – это ожидаемое арифметическое равенство, а равенство множеств целых чисел (**set<int>**) – это математическое равенство множеств, когда элементы сравниваются с использованием равенства целых чисел. В Dafny равенство всегда можно использовать в призрачных контекстах, но не всегда – в компилируемых. Это связано с тем, что значения некоторых типов могут быть не вычислимыми за конечное время или вообще невычислимыми. Например, для сравнения потенциально бесконечных множеств целых чисел (**iset<int>**) требуется бесконечное количество проверок принадлежности множеству, а для сравнения функций, принимающих и возвращающих целые числа (**int -> int**), потребуется вызвать функции для бесконечного числа значений.

По этой причине Dafny следит за тем, какие типы имеют *компилируемую* операцию равенства. Их называют «типами, поддерживающими оператор равенства», хотя это несколько неверное название, поскольку все типы поддерживают оператор равенства в призрачных контекстах. Когда доступно определение типа, Dafny использует его, чтобы выяснить, поддерживает ли этот тип оператор равенства. Это невозможно для параметров типа, непрозрачных типов или типов, для которых экспортируются только сигнатуры. В таких случаях можно явно объявить, что тип относится к типу, поддерживающему оператор равенства. Для этого достаточно указать *характеристику типа* (==) после имени в его объявлении. (Мы уже видели, как это делается на примере функции `Find` в разделе 6.5.)

Рассмотрим клиента модуля `ImmutableQueue`, который определяет свой вариант операции `Dequeue`, применимый к любой очереди и возвращающий значение по умолчанию, если очередь окажется пустой. Вот клиентский модуль:

```
module QueueExtender {
  import IQ = ImmutableQueue
  function TryDequeue<A>(q: IQ.Queue, default: A): (A, IQ.Queue)
  {
    if q == IQ.Empty() then (default, q) else IQ.Dequeue(q)
  }
}
```

При использовании ImmutableQueue в том виде, как мы его определили, Dafny жалуется, что функция TryDequeue использует == для сравнения значений типа IQ.Queue<A>. В модуле QueueExtender неизвестно, поддерживает ли IQ.Queue<A> оператор равенства. Рассмотрим два возможных решения этой проблемы.

9.4.1. Разъяснение поддержки равенства

Один из способов решить проблему – экспортировать из модуля ImmutableQueue тип Queue<A>, который действительно поддерживает оператор равенства. Вы могли бы подумать, что для этого достаточно добавить (==) в объявление **datatype**:

```
datatype Queue(==)<A> = // синтаксическая ошибка
  FQ(front: LL.List<A>, rear: LL.List<A>)
```

Однако характеристику (==) можно использовать только в объявлениях **type** (и параметрах типов, как мы видели на примере функции Find в разделе 6.5). Но, вообще говоря, это не проблема – можно просто переименовать тип данных и с помощью объявления **type** определить *синоним типа* и экспортировать уже этот синоним:

```
type Queue(==)<A> = // заявленная поддержка понятия равенства ошибочна
  Queue'<A>
datatype Queue'<A> = FQ(front: LL.List<A>, rear: LL.List<A>)
```

У нас почти получилось! Но проблема в том, что тип данных Queue'<A> не всегда поддерживает понятие равенства! Он поддерживает его, только если этот оператор поддерживает параметр типа A. Итак, чтобы утверждать, что Queue<A> поддерживает оператор равенства, мы должны ограничить определение теми типами A, которые поддерживают равенство. Для этого нужно добавить характеристику (==) к параметру типа:

```
type Queue(==)<A(==)> = Queue'<A>
datatype Queue'<A> = FQ(front: LL.List<A>, rear: LL.List<A>)
```

Упражнение 9.5

Почему нам не понадобилась характеристика (==), когда мы сравнивали q' == IQ.Empty() в методе Client() в разделе 9.3.3?

9.4.2. Экспорт предиката IsEmpty

Другое решение проблемы – экспортировать из ImmutableQueue не саму операцию == с очередью, а предикат, сравнивающий очередь с Empty(). В конце концов, как я отметил в разделе 9.3.3, обосновывая введение леммы EmptyUnique, сравнение самих значений Queue не особенно полезно, потому что может существовать несколько значений Queue, представляющих

один и тот же список элементов. Но сравнение с пустой очередью – другое дело, потому что ImmutableQueue использует уникальное представление пустой очереди. На самом деле вы можете возразить, что использование предусловия q != Empty() – еще в разделе 9.3.0 – было ошибкой, потому что оно уже предполагает уникальное представление пустой очереди.

В соответствии с этой стратегией решения проблемы мы сохраним объявление **datatype** Queue в первоначальном виде и добавим в модуль ImmutableQueue следующий компилируемый предикат:

```
predicate IsEmpty(q: Queue)
  ensures IsEmpty(q) <==> q == Empty()
{
  q == FQ(LL.Nil, LL.Nil)
}
```

Мы также добавим IsEmpty в инструкцию provides в объявлении **export** в ImmutableQueue. Глядя на тело предиката IsEmpty, вы можете спросить, как он может сравнивать q, используя оператор ==, если тип q поддерживает оператор равенства только для типов элементов, поддерживающих равенство? Объясняется это просто: как особый случай Dafny допускает сравнение с выражениями, состоящими только из литералов, например FQ(LL.Nil, LL.Nil). В конце концов, такое сравнение можно также выполнить с помощью выражений match или дискриминаторов типов данных FQ? и Nil? (Напомню, что сравнение q == Empty() в постусловии тоже разрешено, поскольку Dafny всегда поддерживает равенство в призрачных контекстах.)

Упражнение 9.6

Не используя (==), перепишите модуль QueueExtender, использовав IsEmpty.

Упражнение 9.7

Экспортируйте два новых члена модуля ImmutableQueue: компилируемую функцию Length, которая возвращает количество элементов в заданной очереди, и лемму о корректности Length(q), которая гласит, что она возвращает длину списка Elements(q). Из этого следует, что проверка Length(q) == 0 дает тот же результат, что и q == Empty(). Перепишите функцию TryDequeue в модуле QueueExtender, использовав Length.

Упражнение 9.8

Реализация TryDequeue, предложенная в упражнении 9.7, не очень эффективна. Почему?

9.5. Итоги

Модульный дизайн имеет решающее значение в разработке программного обеспечения. Он помогает выполнить декомпозицию программы и скрыть решения по реализации внутри модулей. Дэвид Парнас (David Parnas) указал, что эту возможность можно описать лозунгом: «Каждый модуль имеет свои секреты» [108]. Чтобы иметь возможность писать спецификации и осуществлять верификацию при использовании приема сокрытия такой информации, мы должны абстрагироваться от деталей. В этой главе я показал, как это можно сделать с помощью функции абстракции (на примере `Elements`), которая отображает реализуемые нами значения в значения других известных и более простых типов.

Объявления делятся на модули, и модули могут импортировать другие необходимые им модули. Объявляя экспортируемый набор, модуль решает, какие объявления будут доступны клиентам. Для каждого объявления в экспортируемом наборе можно экспортировать только его сигнатуру (с использованием инструкции **`provides`**) или сигнатуру с телом (с использованием инструкции **`reveals`**).

Примечания

Множество интересных и полезных реализаций функциональных структур данных, таких как очереди в разделе 9.3.4, можно найти в книге Криса Окасаки (Chris Okasaki) «Purely Functional Data Structures»[1] [104].

Для поддержки сокрытия информации языки программирования предоставляют механизмы контроля доступа, ограничивающие видимость объявлений. Как мы видели, контроль доступа к объявлениям в модуле в Dafny определяется через экспортируемые наборы. Многие объектно ориентированные языки на основе классов, включая Java и C#, позволяют объявлять члены класса с такими модификаторами доступа, как **`private`** (доступные только в реализации класса) и **`public`** (доступные в любом контексте, где доступен сам класс). C++ и Eiffel дополнительно позволяют членам одного класса объявлять «дружественные» классы, получающие привилегированный доступ к этим членам. Другие языки, включая SPARK [117], Oberon-2 и Go, используют механизм приватного/публичного доступа на уровне модуля.

Во многих языках программирования механизмы контроля доступа определяют только доступность данного объявления клиентам. Этого достаточно для компиляции. Но когда речь идет о спецификациях и доказательствах, также важно иметь возможность ограничить, насколько это *осмысленно*, доступность объявлений клиентам. По этой причине Dafny предлагает два уровня экспорта объявлений (**`provides`** и **`reveals`**) [80], а F* [53] поддерживает включение в интерфейсы своих модулей семантической информации.

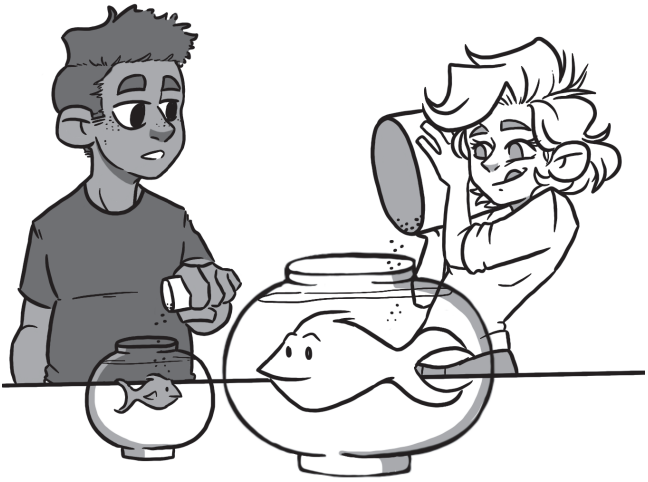
[1] Крис Окасаки, «Чисто функциональные структуры данных», ДМК-Пресс, 2016, ISBN: 978-5-97060-233-1. – *Прим. перев.*

Каждый модуль в этой книге имеет только один экспортируемый набор – типичный случай. В других случаях модулю может потребоваться определить разные интерфейсы для разных клиентов. Например, модуль может раскрывать некоторые детали какому-то «другому» модулю. Для этой цели Dafny позволяет модулю объявлять несколько экспортируемых наборов [80]. Подобную гибкость также обеспечивают модули и интерфейсы в Modula-3 [100].

Абстрагирование можно организовать разными способами. В этой главе мы использовали функции абстракции – призрачные функции, сообщающие о некоторых деталях значений, не ограничивая их реализации какими-либо конкретными представлениями. Функции абстракции часто встречаются в инструментах верификации. В главах 16 и 17 мы применим абстракцию, используя призрачные поля в сочетании с предикатом валидации. Язык SPARK [117] имеет специальную поддержку объявления абстрактного (`Abstract_State`) и конкретного (`Refined_State`) состояния. Существует множество замечательных книг, обучающих методам абстрагирования данных. Со своей стороны могу порекомендовать труды истинных мастеров абстракций: Morgan [93], где описывается абстрагирование с помощью операторов спецификации, Abrial [0] – абстрагирование с помощью Event-B и Lamport [73] – абстрагирование с помощью TLA+.

Глава 10

Инварианты структур данных



Это здорово, что существуют типы! Они позволяют описать скелет структуры данных и проанализировать программу, чтобы убедиться, что она поддерживает этот скелет. Это означает, что мы можем положиться, например, на тот факт, что значение типа `Tree`, определенного в разделе 4.5, является либо значением `Leaf`, либо `Node`.

Для большинства структур данных их архитектура – это нечто большее, чем просто скелет. Каждое значение структуры данных соответствует скелету, но не каждое значение, соответствующее скелету, является значением, которое можно создать с использованием операций, поддерживаемых структурой данных. Например, *двоичное дерево поиска* – это структура данных с древовидным скелетом и значениями полезной нагрузки, расположенными в отсортированном порядке. Подобные ограничения, выходящие за рамки скелета, называются *инвариантами структуры данных*. Инварианты имеют решающее значение для организации программы и ее корректности.

Инварианты структур данных я проиллюстрирую в этой главе на примере *очереди с приоритетами*. Как мы знаем, обычная очередь представляет собой набор элементов, доступ к которым осуществляется в порядке очередности (как мы видели в разделе 9.3), в отличие от нее очередь с приоритетами предоставляет доступ только к наименьшему элементу коллекции. Начнем со спецификации, а затем перейдем к реализации.

10.0. Спецификация очереди с приоритетами

Помимо создания и проверки наличия элементов, в число операций с очередью с приоритетами входят также вставка элемента и удаление элемента с наименьшим значением. Мы будем рассматривать очереди с приоритетами, хранящие целые числа, поэтому «наименьший» элемент будет определяться в соответствии с арифметическими правилами сравнения. Начнем спецификацию с объявления следующего простого (и неполного) модуля:

```
module PriorityQueue {
  type PQueue
  function Empty(): PQueue
  predicate IsEmpty(pq: PQueue)
  function Insert(pq: PQueue, y: int): PQueue
  function RemoveMin(pq: PQueue): (int, PQueue)
    requires !IsEmpty(pq)
}
```

Пока что модуль очень похож на аналогичный модуль с реализацией обычной очереди в разделе 9.3. Здесь я сразу решил, что мы не будем использовать уникальное представление значений, даже пустой очереди, поэтому в модуль включен предикат `IsEmpty` (как описано в разделе 9.4.2).

10.0.0. Абстракция

Чтобы сказать больше о том, что возвращают эти функции, нужна некоторая абстракция того, что представляет собой каждое значение `PQueue`. Так же как в разделе 9.3, для этой цели введем функцию абстракции, отображающую `PQueue` в ее элементы. Здесь у нас есть выбор. Одна из возможностей – абстрагировать элементы в отсортированную последовательность (типа **`seq<int>`**). Это облегчит написание спецификации `RemoveMin`, но усложнит спецификацию `Insert`. Другая возможность (именно ее мы и используем) – абстрагировать элементы в мультимножество (типа **`multiset<int>`**). Это упростит спецификацию `Insert`, но усложнит (незначительно) спецификацию `RemoveMin`.

Введем функцию абстракции `Elements` (только для спецификации) в нашем модуле и, используя ее, напишем внешние спецификации для наших функций в виде отдельных лемм:

```
ghost function Elements(pq: PQueue): multiset<int>
lemma EmptyCorrect()
  ensures Elements(Empty()) == multiset{}
lemma IsEmptyCorrect(pq: PQueue)
  ensures IsEmpty(pq) <==> Elements(pq) == multiset{}
lemma InsertCorrect(pq: PQueue, y: int)
  ensures Elements(Insert(pq, y))
       == Elements(pq) + multiset{y}
```

```
lemma RemoveMinCorrect(pq: PQueue)
  requires !IsEmpty(pq)
  ensures var (y, pq') := RemoveMin(pq);
    IsMin(y, Elements(pq)) &&
    Elements(pq') + multiset{y} == Elements(pq)
```

В лемме о `RemoveMin` используется предикат `IsMin`, проверяющий, является ли `y` наименьшим элементом в мультимножестве `Elements(pq)`. `IsMin` можно определить рекурсивно, но есть более простой способ – использовать *квантор всеобщности* (**forall**, в математике записывается как ∀), который может что-то сообщить о каждом элементе мультимножества. Более подробно я расскажу о кванторах в главе 13, а пока нам достаточно знать, что `IsMin` определен корректно:

```
ghost predicate IsMin(y: int, s: multiset<int>) {
  y in s && forall x :: x in s ==> y <= x
}
```

Проще говоря, предикат `IsMin(y, s)` возвращает **true**, если **y** является элементом мультимножества `s` и любой элемент `x` в `s` имеет значение выше, чем `y`.

Пока, с точки зрения клиентов, наш модуль выглядит довольно неплохо.

Упражнение 10.0

Напишите небольшой тест в клиентском модуле (см. `QueueClient` в разделе 9.3.3). Тестовая программа должна вставить два элемента во вновь созданную очередь с приоритетами, а затем дважды вызвать `RemoveMin` для их извлечения. Убедитесь, что элементы возвращаются в правильном порядке.

Подсказка: чтобы убедить верификатор в корректности, вам понадобятся некоторые леммы. Конечно, вам нужно будет вызвать леммы, объявленные в `PriorityQueue`, а также некоторые дополнительные леммы, которые помогут верификатору рассуждать о мультимножествах. В частности, вам понадобится написать несколько встроенных утверждений **assert** о том, что известно об элементах приоритета в промежуточных точках в тестовой программе.

10.0.1. Экспортируемый набор

Продолжим разработку модуля `PriorityQueue` и определим, что должны видеть клиенты, а что должно остаться скрытым от них. Для этого добавим явное объявление **export**. Модуль экспортирует только сигнатуры всех своих компонентов, за исключением предиката `IsMin`, который экспортируется полностью:

```
export
  provides PQueue, Empty, IsEmpty, Insert, RemoveMin
  provides Elements
```

```
provides EmptyCorrect, IsEmptyCorrect
provides InsertCorrect, RemoveMinCorrect
reveals IsMin
```

В этом экспортируемом наборе используется сочетание инструкций **provides** и **reveals**, которые я представил в разделе 9.2.

10.1. Проектирование структуры данных

Для реализации очереди с приоритетами мы используем *двоичную кучу*. Двоичная куча – это двоичное дерево с двумя дополнительными свойствами. Первое свойство – базовое: в отличие от дерева в разделе 9.3, где полезная нагрузка размещается в листьях, двоичная куча размещает свою полезную нагрузку во внутренних узлах дерева, а листья остаются пустыми. Второе свойство – инвариант структуры данных: каждый узел в дереве удовлетворяет так называемому *свойству кучи*, согласно которому значение, хранящееся в узле, является наименьшим среди всех значений в поддереве, корнем которого является этот узел.

Свойство кучи позволяет получить хорошую производительность: методы `Insert` и `RemoveMin` могут выполняться за время $O(\log n)$ (т. е. за время, пропорциональное логарифму размера приоритетной очереди) *при условии*, что дерево сбалансировано. Дерево считается *сбалансированным*, если любое левое поддерево имеет примерно ту же высоту, что и соседнее ему правое поддерево. Элегантная структура данных, гарантирующая сбалансированность дерева двоичной кучи, – это *дерево Брауна*. Именно его мы и возьмем за основу реализации. Основная идея дерева Брауна заключается в том, что либо левое и правое поддеревья узла имеют одинаковую высоту, либо левое поддерево имеет высоту на один элемент больше.

Итак, всего у нас есть три объекта, есть базовое свойство, согласно которому внутренние узлы хранят значения, есть свойство кучи и есть свойство сбалансированности дерева Брауна. Последние два свойства называются *инвариантами структуры данных*, потому что они не входят в набор базовых свойств.

Чтобы обеспечить базовые свойства двоичной кучи, определим тип `PQueue` как тип данных:

```
type PQueue = BraunTree
datatype BraunTree =
  | Leaf
  | Node(x: int, left: BraunTree, right: BraunTree)
```

(Dafny позволяет использовать вертикальную черту | перед именем первого конструктора, что выглядит естественно, когда конструкторы объявляются в отдельных строках, как я сделал это здесь.) Чтобы обеспечить инвариант деревьев Брауна, нужно использовать некоторые логические выражения.

10.1.0. Допустимые деревья

Корректность нашей очереди с приоритетами зависит от свойства кучи. Поэтому для верификации нам необходима возможность отличать деревья, обладающие свойством кучи, от других деревьев, которую (в принципе) можно получить из конструкторов `Leaf` и `Node`. Реализуем ее, введя предикат `Valid`. Идея состоит в том, что дерево удовлетворяет предикату `Valid` тогда и только тогда, когда это дерево может быть возвращено нашим модулем `PriorityQueue`.

Мы экспортируем клиентам сигнатуру предиката `Valid`, чтобы дать возможность использовать его в спецификациях, но сохраним детали реализации в секрете:

```
export // ...
  provides Valid

ghost predicate Valid(pq: PQueue)
```

Далее нужно сделать выбор: `Valid` может использоваться внутренне или внешне, т. е. в спецификациях самих операций с очередью с приоритетами (внутренне) или только в леммах корректности (внешне). Если бесперебойное функционирование древовидных операций будет зависеть от условия достоверности, то мы *должны* использовать его внутренне, но в остальном выбор аналогичен другим случаям выбора между внешним и внутренним использованием, которые мы видели выше (см. раздел 6.2). Пока я буду использовать `Valid` внешне, но вернусь к вопросу выбора в разделе 10.3.

Используя `Valid`, мы можем изменить определения лемм следующим образом:

```
lemma EmptyCorrect()
  ensures var pq := Empty();
    Valid(pq) &&
    Elements(pq) == multiset{}
lemma IsEmptyCorrect(pq: PQueue)
  requires Valid(pq)
  ensures IsEmpty(pq) <==> Elements(pq) == multiset{}
lemma InsertCorrect(pq: PQueue, y: int)
  requires Valid(pq)
  ensures var pq' := Insert(pq, y);
    Valid(pq') &&
    Elements(pq') == Elements(pq) + multiset{y}
lemma RemoveMinCorrect(pq: PQueue)
  requires Valid(pq) && !IsEmpty(pq)
  ensures var (y, pq') := RemoveMin(pq);
    Valid(pq') &&
    IsMin(y, Elements(pq)) &&
    Elements(pq') + multiset{y} == Elements(pq)
```

Упражнение 10.1

Вернитесь к упражнению 10.0 и убедитесь, что тестовая программа по-прежнему работает, потому что леммы о корректности учитывают инвариант структуры данных.

10.2. Реализация

10.2.0. Определение инварианта

Мы должны определить в `Valid` все нужные нам свойства, которые являются инвариантами нашей структуры данных. В дополнение к свойству кучи наша структура данных должна также удовлетворять свойству сбалансированности дерева Брауна. Вот все необходимые определения:

```
ghost predicate Valid(pq: PQueue) {
  IsBinaryHeap(pq) && IsBalanced(pq)
}

ghost predicate IsBinaryHeap(pq: PQueue) {
  match pq
  case Leaf => true
  case Node(x, left, right) =>
    IsBinaryHeap(left) && IsBinaryHeap(right) &&
    (left == Leaf || x <= left.x) &&
    (right == Leaf || x <= right.x)
}

ghost predicate IsBalanced(pq: PQueue) {
  match pq
  case Leaf => true
  case Node(_, left, right) =>
    IsBalanced(left) && IsBalanced(right) &&
    var L, R := |Elements(left)|, |Elements(right)|;
    L == R || L == R + 1
}
```

Предикат `IsBinaryHeap` рекурсивно проверяет условие для поддеревьев `left` и `right`, поэтому достаточно проверить свойство кучи для каждого узла, сравнив значение этого узла со значениями непосредственных дочерних элементов (см. упражнение 10.2 после определения `Elements` ниже).

Так же как в примере с очередью в разделе 9.3, центральное место во всех наших описаниях корректности занимает функция абстракции `Elements`. Она абстрагирует структуру данных от реализации и предоставляет клиентам простую мысленную модель работы приоритетной очереди. Эта функция определяется как объединение мультимножеств всех значений, хранящихся в дереве:

```
ghost function Elements(pq: PQueue): multiset<int> {
  match pq
  case Leaf => multiset{}
  case Node(x, left, right) =>
    multiset{x} + Elements(left) + Elements(right)
}
```

Упражнение 10.2

Предикат `IsBinaryHeap` рекурсивно проверяет условие для поддеревьев `left` и `right`; свойство «узел содержит наименьшее из всех значений в поддереве узла» проверяется простым сравнением значения текущего узла со значениями в его непосредственных дочерних элементах. Докажите правильность определения предиката. Другими словами, докажите, что значение любого узла, удовлетворяющего предикату `IsBinaryHeap`, является наименьшим в дереве, корнем которого является этот узел. То есть докажите следующую лемму:

```
lemma {:induction false} BinaryHeapStoresMinimum(pq: PQueue, y: int)
  requires IsBinaryHeap(pq) && y in Elements(pq)
  ensures pq.x <= y
```

Формулировка леммы довольно любопытна. Постусловие выбирает `.x`, поэтому вполне можно было бы ожидать, что предусловие будет иметь конъюнкт `pq.Node?`. Почему лемма в таком ее виде все еще верна?

Любопытно также доказательство этой леммы. (*Небольшая подсказка!*) Доказательство проводится индукцией по структуре `pq`. Большинство таких доказательств распадаются на структурные случаи, но не в этот раз. Здесь `pq` должен быть экземпляром `Node`. Поэтому случаи доказательства распадаются, как объединение трех частей в определении `Elements`.

Далее мы напишем реализацию и доказательства корректности операций создания и проверки пустой очереди, а также вставки и удаления.

10.2.1. Пустая очередь

Пустая приоритетная очередь представляется как лист, поэтому тело `Empty` определяется следующим образом:

```
function Empty(): PQueue {
  Leaf
}
```

Фактически в нашей реализации пустая приоритетная очередь имеет уникальное представление, поэтому мы определим предикат `IsEmpty` как проверку на равенство с `Leaf`:

```
predicate IsEmpty(pq: PQueue) {
  pq == Leaf
}
```

Далее докажем корректность `Empty` и `IsEmpty`, заполнив тела лемм `EmptyCorrect` и `IsEmptyCorrect`:

```
lemma EmptyCorrect()
  ensures var pq := Empty();
    Valid(pq) &&
    Elements(pq) == multiset{}
{
}

lemma IsEmptyCorrect(pq: PQueue)
  requires Valid(pq)
  ensures IsEmpty(pq) <==> Elements(pq) == multiset{}
{
}
```

Обе леммы доказываются автоматически.

10.2.2. Вставка

Чтобы вставить элемент в пустую двоичную кучу, достаточно вернуть дерево, хранящее элемент в узле с пустыми поддеревьями:

```
function Insert(pq: PQueue, y: int): PQueue {
  match pq
  case Leaf => Node(y, Leaf, Leaf)
```

Чтобы вставить элемент в узел `Node`, требуется выполнить дополнительные действия. Напомню, что двоичная куча – это дерево, в котором элементы, хранящиеся в каждом узле, меньше элементов, хранящихся в поддеревьях этого узла. Таким образом, если элемент, хранящийся в узле (назовем его `x`) меньше элемента `y`, который нужно вставить, то операция вставки создает узел, хранящий `x`, и вставляет `y` в его поддерево. В противном случае если `y` меньше `x`, то операция вставки делает все наоборот: создает узел, хранящий `y`, и вставляет `x` в его поддерево.

Когда `x` или `y` вставляется в поддерево, то какое поддерево следует выбрать для вставки? Для сохранности свойства кучи это не имеет значения. Мы можем выполнить рекурсивную вставку как в левое, так и в правое поддерево. Однако для сохранности свойства сбалансированности выбор поддерева имеет значение. Вставка должна производиться в меньшее поддерево.

Как определить, какое из двух поддеревьев меньше? Вот тут-то и вступает в игру ключевая особенность деревьев Брауна. В дереве Брауна правое поддерево никогда не бывает больше левого, поэтому вставку нужно выпол-

нить в него. Но если два поддерева изначально имеют одинаковый размер, то вставка в правое поддерево сделает его больше. Прекрасное наблюдение в отношении деревьев Брауна заключается в том, что мы можем сохранить инвариант дерева Брауна, (выполнив вставку в правое поддерево, а затем) поменяв местами левое и правое поддеревья. В результате получится дерево, в котором либо левое и правое поддеревья будут иметь одинаковый размер, либо левое поддерево будет на единицу больше.

Вот остальная часть определения метода `Insert`:

```
    case Node(x, left, right) =>
      if y < x then
        Node(y, Insert(right, x), left)
      else
        Node(x, Insert(right, y), left)
  }
```

Доказательство корректности `Insert` осуществляется по индукции, учитывая случаи в функции. На самом деле это настолько просто, что верификатор выполняет доказательство автоматически:

```
  lemma InsertCorrect(pq: PQueue, y: int)
    requires Valid(pq)
    ensures var pq' := Insert(pq, y);
      Valid(pq') &&
      Elements(pq') == Elements(pq) + multiset{y}
  {
  }
```

10.2.3. Удаление наименьшего элемента

Из-за предусловия `!IsEmpty(pq)` функция `RemoveMin(pq)` всегда применяется к узлу, а не к листу. Элемент этого узла помещается в первый компонент кортежа, возвращаемого функцией `RemoveMin`, поскольку в двоичной куче наименьший элемент хранится в корневом узле (см. также упражнение 10.2). В другом компоненте кортежа возвращается `pq`, но с удаленным минимальным элементом. Давайте добавим вспомогательную рекурсивную функцию `DeleteMin`, которую напишем позже, и реализуем `RemoveMin` следующим образом:

```
  function RemoveMin(pq: PQueue): (int, PQueue)
    requires !IsEmpty(pq)
  {
    (pq.x, DeleteMin(pq))
  }

  function DeleteMin(pq: PQueue): PQueue
    requires !IsEmpty(pq)
```

Определим лемму о корректности DeleteMin. Она позволит нам понять, что должна делать DeleteMin, а также доказать корректность RemoveMin:

```
lemma RemoveMinCorrect(pq: PQueue)
  requires Valid(pq) && !IsEmpty(pq)
  ensures var (y, pq') := RemoveMin(pq);
    Valid(pq') &&
    IsMin(y, Elements(pq)) &&
    Elements(pq') + multiset{y} == Elements(pq)
{
  DeleteMinCorrect(pq);
}

lemma DeleteMinCorrect(pq: PQueue)
  requires Valid(pq) && pq != Leaf
  ensures var pq' := DeleteMin(pq);
    Valid(pq') &&
    Elements(pq') + multiset{pq.x} == Elements(pq)
```

Кажется чудом, что доказательство RemoveMinCorrect вызывает только лемму DeleteMinCorrect, поскольку последняя ничего не говорит о IsMin. Чудо происходит благодаря механизму автоматической индукции в Dafny, который позволяет верификатору доказать проверяемое условие IsMin без дальнейших подсказок с нашей стороны. (Чтобы убедиться в использовании индукции, можно временно отключить автоматическую индукцию для RemoveMinCorrect, добавив атрибут {:induction **false**}. Это вынудит верификатор сообщить, что он не может доказать лемму.) Я не жду, что вы сможете написать доказательство для IsMin самостоятельно, потому что я еще ничего не сказал о том, как вручную доказать корректность квантора всеобщности в IsMin. Итак, если вы хотите более подробно разобраться, почему y является наименьшим среди всех значений в Elements(pq), то вместо этого я рекомендую выполнить упражнение 10.2.

10.2.4. Вспомогательная функция DeleteMin

Для небольшого дерева, в котором одно или оба поддерева пусты, удаление корня приводит к созданию только левого поддерева:

```
function DeleteMin(pq: PQueue): PQueue
  requires !IsEmpty(pq)
{
  if pq.left == Leaf || pq.right == Leaf then
    pq.left
```

Рассмотрим узел с двумя непустыми поддеревьями. Чтобы удалить наименьший элемент из него, ставим на его место корень одного из поддеревьев. При этом мы должны убедиться, что элемент, «продвигаемый» та-

ким образом, является меньшим из корней двух поддеревьев, поскольку именно это необходимо для сохранения свойства кучи. Итак, мы поступаем следующим образом:

```
  else if pq.left.x <= pq.right.x then
    Node(pq.left.x, ...)
  else
    Node(pq.right.x, ...)
```

В первом из этих двух случаев мы должны удалить элемент, который мы продвинули, из левого поддерева, поэтому вызываем `DeleteMin(pq.left)` рекурсивно. Это обеспечит сохранность свойства кучи дерева Брауна. Для поддержки сбалансированности дерева Брауна мы также меняем местами два поддерева, по сути, выполняя действие, противоположное операции `Insert`. Это превращает первый из двух случаев в следущее:

```
  else if pq.left.x <= pq.right.x then
    Node(pq.left.x, pq.right, DeleteMin(pq.left))
```

Второй случай не симметричен первому, потому что, вызвав функцию `DeleteMin(pq.right)`, мы создадим поддерево, которое может содержать на два элемента меньше, чем `pq.left`, и тем самым нарушим свойство сбалансированности дерева Брауна. Поэтому мы сначала поменяем местами корни левого и правого поддеревьев, а затем поступим, как в предыдущем случае. Реализуем перестановку как еще один вспомогательный метод `ReplaceRoot`:

```
  else
    Node(pq.right.x, ReplaceRoot(pq.right, pq.left.x),
         DeleteMin(pq.left))
}

function ReplaceRoot(pq: PQueue, y: int): PQueue
  requires !IsEmpty(pq)

lemma ReplaceRootCorrect(pq: PQueue, y: int)
  requires Valid(pq) && !IsEmpty(pq)
  ensures var pq' := ReplaceRoot(pq, y);
    Valid(pq') &&
    Elements(pq) + multiset{y} ==
    Elements(pq') + multiset{pq.x}
```

Упражнение 10.3

Самый первый случай в `DeleteMin` обрабатывает ситуацию, когда одно или оба поддерева пусты. В условии `if` я написал проверку в виде дизъюнкции («или»):

```
pq.left == Leaf || pq.right == Leaf
```

Если дерево сбалансировано, то условие `pq.left == Leaf` подразумевает `pq.right == Leaf`, поэтому дизъюнкция эквивалентна более простой проверке `pq.right == Leaf`. Почему я не использовал более короткий вариант? Что произойдет, если изменить текущую проверку на более простую?

Начнем доказательство `DeleteMinCorrect` как обычно, следуя за структурой функции `DeleteMin`:

```
lemma DeleteMinCorrect(pq: PQueue)
  requires Valid(pq) && pq != Leaf
  ensures var pq' := DeleteMin(pq);
    Valid(pq') &&
    Elements(pq') + multiset{pq.x} == Elements(pq)
{
  if pq.left == Leaf || pq.right == Leaf {
  } else if pq.left.x <= pq.right.x {
    DeleteMinCorrect(pq.left);
```

Первый случай обрабатывается автоматически. Во втором случае определение функции `DeleteMin` выполняет рекурсивный вызов для `pq.left`, поэтому мы ожидаем, что в лемме потребуется аналогичный рекурсивный вызов. Это действительно помогает доказать ее.

Третий случай сложнее. Начнем с присвоения имен подвыражениям, которые мы будем использовать:

```
  } else {
    var left, right :=
      ReplaceRoot(pq.right, pq.left.x), DeleteMin(pq.left);
    var pq' := Node(pq.right.x, left, right);
```

Получилось довольно много кода. Чтобы убедиться, что мы не допустили никаких ошибок, позволим верификатору проверить, что `pq'` действительно является результатом `DeleteMin`:

```
assert pq' == DeleteMin(pq);
```

Мы ожидаем, что нам понадобятся свойства корректности `left` и `right`, поэтому добавим вызовы соответствующих лемм:

```
ReplaceRootCorrect(pq.right, pq.left.x);
DeleteMinCorrect(pq.left);
```

Если бы нам повезло, то после этого верификатор завершил бы доказательство. Но, увы, это не так, поэтому продолжим работу. Продолжить можно разными способами. Если бы вы были одни, то, вероятно (как и я), попытались бы доказать каждую из частей заключения леммы. Когда я попробовал это сделать, у меня получилось длинное и уродливое доказа-

тельство. Рассмотрев различные части доказательства, проведя его рефакторинг и попробовав множество разных способов доказательства каждой части, я в итоге получил более простое доказательство. Здесь я покажу только полученный мною результат.

Он начинается с доказательного вычисления, показывающего свойство корректности `Elements` для `DeleteMin`. Это свойство показывает равенство двух мультимножеств. В таких случаях для поддержки верификатора обычно требуется написать большой объем кода. В отношении таких доказательств я хочу дать вам два совета.

Первый совет для доказательств, использующих мультимножества

Верификатор может доказать равенство двух мультимножеств, но он не очень изобретателен, когда дело доходит до выяснения, какие равенства следует попытаться доказать. Поэтому, если у вас есть выражение типа `...A...`, где `A` обозначает мультимножество, и вы хотите показать, что оно равно выражению `...B...` с некоторым другим мультимножеством `B`, то попробуйте структурировать доказательные вычисления следующим образом:

```
  ...A...;
==  { assert A == B; }
  ...B...;
```

В подсказке утверждение **`assert`** требует от верификатора доказать равенство мультимножеств `A` и `B`. Для этого верификатор попробует доказать, что мультимножества имеют одинаковые элементы (точнее, что мультимножества имеют одинаковую кратность каждого элемента). Как только он убедится, что `A` и `B` действительно равны, он использует это равенство для проверки шага доказательного вычисления.

Приведенное выше утверждение **`assert`** о равенстве имеет собственное название: *экстенсиональность*. Получив требование проверить `A == B`, верификатор проверяет, содержат ли `A` и `B` одни и те же элементы, и ему часто удается это, даже если он сам не помышлял о необходимости проверки равенства `A == B`.

Второй совет для доказательств, использующих мультимножества

Как и сложение в арифметике, объединение мультимножеств является ассоциативным и коммутативным. Например, `A + B + C` – это то же мультимножество, что и `A + C + B`. Верификатор может это проверить, но вам, возможно, придется помочь ему, обратив его внимание на равенство, как я показал выше. Заменяя равное равным, верификатор рассматривает объединение мультимножеств как левоассоциативное и не обязательно ком-

мутативное. Следовательно, предположим, что в формуле присутствует подвыражение `A + B + C`, и вы хотите показать, что формула не изменится, если заменить это подвыражение на `A + D + E`, при наличии некоторого способа доказать `B + C == D + E`. Если бы вы попробовали такое доказательство, как

```
  ...A + B + C...;
==  { assert B + C == D + E; } // Я полагаю, что у вас есть причина,
                               // почему это условие должно выполняться
  ...A + D + E...; // вы получите сообщение, что
                   // доказать этот шаг не получается
```

тогда верификатор все равно будет сообщать об ошибке, потому что он видит:

```
  ...(A + B) + C...;
==  { assert B + C == D + E; }
  ...(A + D) + E...; // вы получите сообщение, что
                     // доказать этот шаг не получается
```

Вместо этого вам придется вручную переставлять части формулы. Удобнее всего это сделать отдельными этапами:

```
  ...A + B + C...;
==  { assert A + B + C == A + (B + C); }
  ...A + (B + C)...;
==  { assert B + C == D + E; } // И снова я полагаю, что у вас есть причина,
                               // почему это условие должно выполняться
  ...A + (D + E)...;
==  { assert A + (D + E) == A + D + E; }
  ...A + D + E...;
```

Хватит советов, вернемся к доказательству

Теперь, опираясь на эти два совета, мы можем написать доказательное вычисление для `DeleteMin`. В шагах без подсказок я просто переставил операнды объединения мультимножеств.

```
calc {
  Elements(pq') + multiset{pq.x};
==   // определение Elements, поскольку pq' -- это Node
  multiset{pq.right.x} + Elements(left) +
  Elements(right) + multiset{pq.x};
==
  Elements(left) + multiset{pq.right.x} +
  Elements(right) + multiset{pq.x};
==  { assert Elements(left) + multiset{pq.right.x}
           == Elements(pq.right) + multiset{pq.left.x}; }
  Elements(pq.right) + multiset{pq.left.x} +
  Elements(right) + multiset{pq.x};
```

```
  ==
    Elements(right) + multiset{pq.left.x} +
    Elements(pq.right) + multiset{pq.x};
  ==  { assert Elements(right) + multiset{pq.left.x}
            == Elements(pq.left); }
    Elements(pq.left) + Elements(pq.right) +
    multiset{pq.x};
  ==
    multiset{pq.x} + Elements(pq.left) +
    Elements(pq.right);
  ==  // определение Elements, поскольку pq -- это Node
    Elements(pq);
  }
```

Чтобы завершить доказательство `DeleteMin`, нужна дополнительная информация о размерах `pq'.left` и `pq'.right`. Чтобы дерево `pq'` было сбалансированным, размер ветви `pq'.left` должен быть равен или на единицу больше размера ветви `pq'.right`. Поскольку дерево `pq` – сбалансированное, необходимое нам свойство вытекает из того, что рекурсивный вызов `DeleteMin` уменьшает размер на единицу, а вызов `replaceRoot` не меняет размер. Эти свойства можно доказать с помощью постусловий, написанных для лемм `DeleteMinCorrect` и `replaceRootCorrect`. Однако было бы удобнее, если бы в этих леммах были постусловия, явно упоминающие эти свойства размеров. Итак, давайте изменим постусловие в `DeleteMinCorrect`, добавив конъюнкт `|Elements(pq')| == |Elements(pq)| - 1`, а также изменим постусловие в `replaceRootCorrect`, добавив конъюнкт `|Elements(pq')|`.

Доказательство `DeleteMinCorrect` теперь проходит без дальнейшего нашего участия:

```
    }
  }
```

Итак, после усиления лемм и явного упоминания влияния `DeleteMin` и `replaceRoot` на размер дерева единственным нестандартным доказательством, которое нам пришлось предоставить, было доказательное вычисление для элементов в третьей ветви в `DeleteMin`.

10.2.5. Вспомогательная функция replaceRoot

Развязка уже близка – у нас осталась всего одна функция. Функция `replaceRoot(pq, y)` удаляет минимальный элемент из `pq` и вставляет `y`. Если ни в одном из поддеревьев `pq` нет элемента меньше `y`, то мы возвращаем узел, подобный `pq`, но с корнем `y`. Это происходит в нескольких случаях, в зависимости от количества непустых поддеревьев в `pq`:

```
function ReplaceRoot(pq: PQueue, y: int): PQueue
  requires !IsEmpty(pq)
```

```
{
  if pq.left == Leaf ||
     (y <= pq.left.x && (pq.right == Leaf || y <= pq.right.x))
  then
    Node(y, pq.left, pq.right)
```

Из случаев малого дерева остается только один, а именно когда `pq` имеет одно непустое поддерево с элементом, меньшим `y`. В этом случае мы продвигаем этот элемент и помещаем `y` в отдельное поддерево:

```
  else if pq.right == Leaf then
    Node(pq.left.x, Node(y, Leaf, Leaf), Leaf)
```

В остальных случаях мы продвигаем корень поддерева, имеющего минимальный элемент, а затем заменяем этот элемент на `y`:

```
  else if pq.left.x < pq.right.x then
    Node(pq.left.x, ReplaceRoot(pq.left, y), pq.right)
  else
    Node(pq.right.x, pq.left, ReplaceRoot(pq.right, y))
}
```

Доказательство корректности `replaceRoot` начинается как обычно, следуя за случаями в функции. Первые два случая обрабатываются автоматически. (Напомню, что доказательство `DeleteMinCorrect` заставило нас добавить в `replaceRootCorrect` конъюнкт, утверждающий, что `replaceRoot` не меняет размер дерева.)

```
lemma ReplaceRootCorrect(pq: PQueue, y: int)
  requires Valid(pq) && !IsEmpty(pq)
  ensures var pq' := ReplaceRoot(pq, y);
    Valid(pq') &&
    Elements(pq) + multiset{y} == Elements(pq') + multiset{pq.x} &&
    |Elements(pq')| == |Elements(pq)|
{
  if pq.left == Leaf ||
     (y <= pq.left.x && (pq.right == Leaf || y <= pq.right.x))
  {
  } else if pq.right == Leaf {
```

Третий случай начинается с введения имен для подвыражений, составляющих результат `replaceRoot`, как мы это сделали для третьего варианта в `DeleteMin` в разделе 10.2.4:

```
  } else if pq.left.x < pq.right.x {
    var left := ReplaceRoot(pq.left, y);
    var pq' := Node(pq.left.x, left, pq.right);
```

Далее проверяем, правильно ли написаны выражения:

```
assert pq' == ReplaceRoot(pq, y);
```

В доказательстве наверняка будет использоваться индуктивное предположение, параметризованное соответствующим образом, поэтому добавим и этот вызов:

```
ReplaceRootCorrect(pq.left, y);
```

Затем докажем заключение леммы об `Elements`. Это еще один пример способности верификатора проверять свойства мультимножеств, но мы должны подсказать ему, какие свойства проверять.

```
calc {
  Elements(pq) + multiset{y};
==  // определение Elements, поскольку pq -- это Node
  multiset{pq.x} + Elements(pq.left) +
  Elements(pq.right) + multiset{y};
==
  Elements(pq.left) + multiset{y} +
  Elements(pq.right) + multiset{pq.x};
==  { assert Elements(pq.left) + multiset{y}
         == Elements(left) + multiset{pq.left.x}; } // Г. И.
  multiset{pq.left.x} + Elements(left) +
  Elements(pq.right) + multiset{pq.x};
==  // определение Elements, поскольку pq' -- это Node
  Elements(pq') + multiset{pq.x};
}
```

Аналогично доказывается четвертый и последний вариант:

```
} else {
  var right := ReplaceRoot(pq.right, y);
  var pq' := Node(pq.right.x, pq.left, right);
  assert pq' == ReplaceRoot(pq, y);
  ReplaceRootCorrect(pq.right, y);
  calc {
    Elements(pq) + multiset{y};
  ==  // определение Elements, поскольку pq -- это Node
    multiset{pq.x} + Elements(pq.left) +
    Elements(pq.right) + multiset{y};
  ==
    Elements(pq.right) + multiset{y} +
    Elements(pq.left) + multiset{pq.x};
  ==  { assert Elements(pq.right) + multiset{y}
           == Elements(right) + multiset{pq.right.x}; }
    multiset{pq.right.x} + Elements(pq.left) +
    Elements(right) + multiset{pq.x};
  == // определение Elements, поскольку pq' -- это Node
```

```
        Elements(pq') + multiset{pq.x};
      }
    }
  }
```

Доказательство заняло у нас некоторое время, зато теперь мы не только реализовали все функции в модуле `PriorityQueue`, но и доказали их корректность. Код получился довольно сложным и содержит больше случаев, чем может с уверенностью проверить человек-рецензент или набор тестов. С другой стороны, относительно простая спецификация и сопровождающее ее доказательство дают уверенность в том, что код написан правильно.

Врезка 10.0

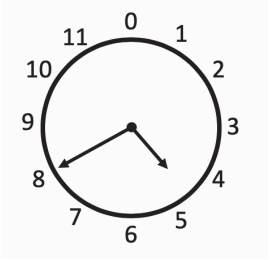
При использовании 12-часовой системы счисления времени суток многие путают 12 a. m. (по полуночи) и 12 p. m. (по полудни). Главный источник путаницы в том, что после 11 a. m. следует 12 p. m. Если в мысленном выражении

```
1 <= hour <= 12
```

заменить 12 на 0, то получится

```
0 <= hour < 12
```

И тогда вы с легкостью запомните, что 0 p. m. (по полудни) соответствует моменту времени за один час до 1 p. m.



Выбирайте инварианты разумно, и это поможет вам упростить архитектуру кода.

10.3. Создание внутренних спецификаций из внешних

В разделе 10.0.0 при разработке модуля `PriorityQueue` мы приняли решение, что все спецификации наших функций, за исключением предусловия непустоты `RemoveMin`, будут внешними. Преимущество этого подхода заключа-

ется в том, что нам не пришлось загромождать тело функции элементами доказательства. Однако есть и недостаток: функции не могут полагаться на инвариант структуры данных. Для некоторых функций это не имеет большого значения. Действительно, все функции в примере с очередью в главе 9 работают вообще без ссылки на инвариант структуры данных. Но для примера с деревом Брауна это имеет значение. Давайте посмотрим, как это сделать.

10.3.0. Оптимизация DeleteMin

Мы начали нашу функцию `DeleteMin` следующим образом:

```
function DeleteMin(pq: PQueue): PQueue
  requires !IsEmpty(pq)
{
  if pq.left == Leaf || pq.right == Leaf then
    pq.left
```

Для дерева, удовлетворяющего условию сбалансированности дерева Брауна, эта проверка **if** немного избыточна. Любое дерево Брауна удовлетворяет условию:

```
pq.left == Leaf ==> pq.right == Leaf
```

поэтому дизъюнкцию в условии **if** можно упростить до `pq.right == Leaf`. Другими словами, не существует случая, когда листом является только левое дерево, поэтому код может просто проверять правое дерево.

Упражнение 10.4

Сформулируйте и докажите лемму, которая показывает, что условие сбалансированности дерева Брауна влечет за собой вышеизложенное.

В результате возникает соблазн изменить реализацию `DeleteMin` и начать ее так:

```
function DeleteMin(pq: PQueue): PQueue
  requires !IsEmpty(pq)
{
  if pq.right == Leaf then
    pq.left
  else if pq.left.x <= pq.right.x then // ошибка: .x требует, чтобы pq.left
                                       // был узлом Node
```

но такой код вызывает сообщение об ошибке верификатора из-за следующей строки, которую я сюда включил. Поскольку эту функцию `DeleteMin` можно вызвать для любого непустого дерева, даже для деревьев, не удов-

летворяющих свойству сбалансированности дерева Брауна, верификатор сообщает, что код пытается получить доступ к элементу x из pq.left. (Кстати! Я только что дал вам ответ на упражнение 10.3.)

Чтобы оптимизировать наш код для деревьев Брауна, нужно задать предусловие, ограничивающее область применения этого кода деревьями Брауна. Этот аспект не учитывался в нашем решении с внешними спецификациями. Давайте изменим решение, хотя бы частично, и добавим предусловие в DeleteMin.

```
function DeleteMin(pq: PQueue): PQueue
  requires IsBalanced(pq) && !IsEmpty(pq)
{
  if pq.right == Leaf then
    pq.left
  else if pq.left.x <= pq.right.x then
    Node(pq.left.x, pq.right, DeleteMin(pq.left))
  else
    Node(pq.right.x, ReplaceRoot(pq.right, pq.left.x),
         DeleteMin(pq.left))
}
```

С таким изменением спецификаций тело функции верифицируется даже с помощью упрощенной проверки if. Отлично! Но верификатор теперь жалуется на вызов DeleteMin в RemoveMin. Давайте изменим спецификацию и этой функции. Попробуем так:

```
function RemoveMin(pq: PQueue): (int, PQueue)
  requires IsBalanced(pq) && !IsEmpty(pq)
```

Ошибка не исчезла, потому что наш модуль экспортирует RemoveMin, но не экспортирует IsBalanced. Мы могли бы добавить экспорт IsBalanced, но это обременит клиентов, использующих приоритетную очередь, еще одной концепцией. Поэтому давайте еще больше усилим предусловие:

```
function RemoveMin(pq: PQueue): (int, PQueue)
  requires Valid(pq) && !IsEmpty(pq)
```

Теперь все в порядке. Мы оптимизировали код в DeleteMin, усилили предусловие и распространили эти изменения в спецификациях на вызывающий код.

10.3.1. Спектр внутренних и внешних спецификаций

Чтобы применить оптимизацию в DeleteMin, нам пришлось усилить некоторые спецификации. Мы постарались ограничить количество дополнительных предусловий, кульминацией которых стало добавление Valid(pq) к предусловию экспортируемой функции RemoveMin. С эстетической точки зрения это не очень элегантно, потому что RemoveMin теперь является един-

ственной из экспортированных функций, в которой упоминается `Valid`. Другими словами, мы попали в ситуацию, когда интерфейс модуля использует смесь внутренних и внешних спецификаций.

Мы могли бы рассмотреть возможность сделать все спецификации полностью внутренними. Другими словами, вместо того чтобы написать спецификации не в отдельных леммах корректности, а непосредственно в функциях. В этом случае функцию `RemoveMin` можно объявить так:

```
function RemoveMin(pq: PQueue): (int, PQueue)
  requires Valid(pq) && !IsEmpty(pq)
  ensures var (y, pq') := RemoveMin(pq);
    Valid(pq') &&
    IsMin(y, Elements(pq)) &&
    Elements(pq') + multiset{y} == Elements(pq)
```

Область выбора между внутренними и внешними спецификациями – это целый спектр. Промежуточным моментом в пространстве проектирования является включение во внутренние спецификации инварианта `Valid` структуры данных и, насколько это возможно, исключение других спецификаций корректности. При таком подходе функцию `RemoveMin` можно объявить так:

```
function RemoveMin(pq: PQueue): (int, PQueue)
  requires Valid(pq) && !IsEmpty(pq)
  ensures var (y, pq') := RemoveMin(pq);
    Valid(pq')
```

10.3.2. Внешне внутренние и внутренне внешние спецификации

Независимо от выбора точки в спектре внутренних и внешних спецификаций, включив в функцию нетривиальное постусловие, мы столкнемся с недостатком внешней спецификации, заключающимся в необходимости засорять тело функции дополнительными вызовами лемм. Но есть вариант, дающий нам простой внутренний (точнее настолько внутренний, насколько мы пожелаем) интерфейс модуля, и в то же время позволяющий отделить определения функций от доказательств их корректности, – это наложение набора внутренних спецификаций функций поверх внешних. Самый простой способ реализовать это – записать два слоя как отдельные модули.

Прежде всего изменим имя модуля `PriorityQueue` на `PriorityQueueExtrinsic`:

```
module PriorityQueueExtrinsic {
  export
    provides PQueue, Empty, IsEmpty, Insert, RemoveMin
    provides Valid, Elements
    reveals IsMin
    provides EmptyCorrect, IsEmptyCorrect, InsertCorrect
```

```
    provides RemoveMinCorrect
    // остальная часть модуля осталась прежней...
}
```

Затем определим еще один модуль уровнем выше модуля `PriorityQueueExtrinsic`, где типы и функции определяются непосредственно через внешние аналоги. В этом модуле нет лемм о корректности, поскольку мы собираемся определить спецификации внутри функций. Дадим этому модулю простое имя `PriorityQueue`, поскольку, как ожидается, именно он будет использоваться клиентами. Вот он:

```
module PriorityQueue {
  export
    provides PQueue, Empty, IsEmpty, Insert, RemoveMin
    provides Valid, Elements
    reveals IsMin
    provides Extrinsic

  import Extrinsic = PriorityQueueExtrinsic

  type PQueue = Extrinsic.PQueue

  ghost predicate Valid(pq: PQueue) {
    Extrinsic.Valid(pq)
  }

  ghost function Elements(pq: PQueue): multiset<int> {
    Extrinsic.Elements(pq)
  }

  ghost predicate IsMin(m: int, s: multiset<int>) {
    Extrinsic.IsMin(m, s)
  }

  function Empty(): PQueue {
    Extrinsic.Empty()
  }

  predicate IsEmpty(pq: PQueue) {
    Extrinsic.IsEmpty(pq)
  }

  function Insert(pq: PQueue, y: int): PQueue {
    Extrinsic.Insert(pq, y)
  }

  function RemoveMin(pq: PQueue): (int, PQueue)
    requires Valid(pq) && !IsEmpty(pq)
```

```
  {
    Extrinsic.RemoveMin(pq)
  }
}
```

Обратите внимание, что в экспортируемом наборе модуля имеется символ `Extrinsic`. Это необходимо, чтобы экспортировать сигнатуру и тело функции `IsMin`, как в `PriorityQueueExtrinsic`. (Экспортируя сигнатуру `Extrinsic`, мы также даем возможность экспортировать не только сигнатуру, но и тело типа `PQueue`. Это может пригодиться в некоторых сложных ситуациях, когда ожидается, что клиенты захотят использовать оба модуля.)

> **Врезка 10.1**
>
> Модули PriorityQueue и PriorityQueueExtrinsic в этом примере объявлены как одноуровневые. Однако точно так же можно было бы сделать PriorityQueueExtrinsic вложенным, объявив его внутри PriorityQueue. В таком случае можно удалить объявление import из PriorityQueue, которое вводит локальное имя Extrinsic, и дать внутреннему модулю имя Extrinsic.

Наконец, добавим спецификации и доказательства. Для этого скопируем все спецификации в виде лемм о корректности из `PriorityQueueExtrinsic` в соответствующие функции в `PriorityQueue` и добавим вызовы лемм из функций:

```
function Empty(): PQueue
  ensures var pq := Empty();
    Valid(pq) &&
    Elements(pq) == multiset{}
{
  Extrinsic.EmptyCorrect();
  Extrinsic.Empty()
}

predicate IsEmpty(pq: PQueue)
  requires Valid(pq)
  ensures IsEmpty(pq) <==> Elements(pq) == multiset{}
{
  Extrinsic.IsEmptyCorrect(pq);
  Extrinsic.IsEmpty(pq)
}

function Insert(pq: PQueue, y: int): PQueue
  requires Valid(pq)
  ensures var pq' := Insert(pq, y);
```

```
    Valid(pq') &&
    Elements(pq') == multiset{y} + Elements(pq)
{
  Extrinsic.InsertCorrect(pq, y);
  Extrinsic.Insert(pq, y)
}

function RemoveMin(pq: PQueue): (int, PQueue)
  requires Valid(pq) && !IsEmpty(pq)
  ensures var (y, pq') := RemoveMin(pq);
    Valid(pq') &&
    IsMin(y, Elements(pq)) &&
    Elements(pq') + multiset{y} == Elements(pq)
{
  Extrinsic.RemoveMinCorrect(pq);
  Extrinsic.RemoveMin(pq)
}
```

В этом примере, показывающем, как наложить модуль с внутренними спецификациями поверх модуля с внешними спецификациями, я использовал полные внешние спецификации. Если вы выберете другую точку в спектре внутренних и внешних спецификаций, то она будет иметь ту же форму.

10.4. Итоги

Инвариант – это условие, которое всегда выполняется. В этой главе мы рассмотрели инвариант, описывающий условие согласованности деревьев Брауна: свойство кучи и свойство сбалансированности дерева Брауна. Мы использовали этот инвариант, чтобы доказать функциональную корректность операций очереди с приоритетами, а также что эти операции создают только сбалансированные деревья. Мы реализовали инвариант как предикат `Valid`. Точнее, `Valid` – это предикат, который выполняется, когда заданное скелетное значение удовлетворяет желаемому инварианту структуры данных. Затем мы использовали `Valid()` в пред- и постусловиях (для лемм корректности в разделе 10.1.0 и для встроенных функций в разделе 10.3.2).

Используя `Valid()` в пред- и постусловиях операций очереди с приоритетами, клиенты нашего модуля `PriorityQueue` уведомляются о существовании условия, что выполняются свойства структуры данных. Экспортируя только сигнатуру `Valid()` в интерфейсе модуля, мы абстрагируемся от деталей инварианта. По сути, это означает, что клиенты не смогут создавать свои деревья на пустом месте, поскольку клиент не знает фактической реализации `Valid()`. Вместо этого клиенты должны будут создавать очереди с приоритетами, используя только имеющиеся операции, которые мы предоставили в интерфейсе модуля. Таким образом, мы создали верифицированный интерфейс с желаемым сокрытием информации.

Если в вашей структуре есть свойство, которое, как вам кажется, присутствует всегда, вы можете использовать инвариант для его проверки. Для этого напишите и докажите лемму о том, что инвариант структуры данных (`Valid()`) подразумевает интересующее свойство (см. упражнение 10.4).

С точки зрения структуры и спецификации программы показанный мной пример имеет много общего с другими модулями, предоставляющими неизменяемые структуры данных. В частности, он фиксирует инвариант структуры данных в предикате `Valid` и использует некоторое количество (здесь – одну) функций абстракции (таких как, `Elements`) для описания эффекта различных операций, реализуемых модулем. Аналогичную картину, хотя и немного более сложную, мы увидим, когда доберемся до изменяемых структур данных в следующей части книги.

Упражнение 10.5

Вернитесь к модулю `BlockChain` из упражнения 9.4 и измените тип результата функции `Balance` с текущего **`int`** на **`nat`**. Чтобы доказать, что функция действительно возвращает неотрицательное число, вам придется использовать инвариант структуры данных.

Добавьте предикат

```
ghost predicate Valid(ledger: Ledger)
```

в модуль, как мы сделали это в данной главе. Используйте этот предикат в спецификациях функций `Init`, `Deposit`, `Withdraw` и `Balance` и докажите, что реализации функций соответствуют их спецификациям. Как обычно, напишите и верифицируйте тестовый код, чтобы убедиться, что ваши спецификации можно использовать так, как вы ожидаете.

Примечания

С деревьями Брауна [24] меня познакомила восхитительная книга Аарона Стампа (Aaron Stump) по верификации в Agda [120].

Предикаты, проверяющие выполнение свойств структуры данных, которые мы использовали в этой главе (и которые увидим в главах 16 и 17), позволяют описывать ожидаемое состояние структуры данных, не разглашая детали этого состояния клиентам. Так достигается сокрытие информации. Но можно ли скрыть еще больше информации? Благодаря выбранному подходу к созданию интерфейса нашего модуля клиент никогда не сможет получить недопустимую структуру. Так нужно ли в таком случае беспокоить клиентов наличием условия действительности? В общем случае ответ на этот вопрос зависит от того, есть ли у клиентов способы, позволяющие обнаружить мутации или взаимодействия между частями реализации. Мы написали наш модуль в функциональном стиле, где значения не изменяются, поэтому ответ заключается в том, что мы также можем скрыть суще-

ствование `Valid`. Для этой цели многие верификаторы (включая Dafny) поддерживают *типы подмножеств* (также известные как *подтипы предикатов* или *уточняющие типы*), но я не буду рассматривать их в этой книге.

Некоторые языки с поддержкой верификации, в том числе Coq [16], PVS [107], Agda [22], F* [53] и LiquidHaskell [123], основаны на таких типах подмножеств. В этих языках принято (или даже необходимо) формулировать инварианты структуры данных как части типов. Расширенные формы такого программирования и проверки рассматриваются в книге Адама Хлипалы (Adam Chlipala) «Certified Programming» [27].

Часть II

Императивные программы

В императивном программировании значение переменной может меняться во время выполнения программы. Это дает нам естественный способ представить развивающееся состояние реальной жизни, например список контактов в социальных сетях или скорость локомотива. Это также означает, что нам не обязательно немедленно вычислять желаемое значение и можно действовать поэтапно, постепенно изменяя значения переменных, пока не получится то, что нам нужно. Императивное программирование иногда превозносится как средство достижения эффективности, поскольку на этапе вычисления можно обновить только то, что необходимо, а не создавать копии всех частей состояния, которые не изменяются.

В этой части я покажу вам, как рассуждать о трех видах императивного состояния: локальных переменных, массивах значений и динамически выделяемых структурах данных. Общим для рассуждений о каждом из них является определенная форма *инварианта*, подобная инвариантам структур данных, которые мы видели в главе 10.

Глава 11

Циклы

Многие интересные вычисления можно описать как многократное повторение одних и тех же шагов. Распространенной конструкцией для этого является *цикл*. Существует множество форм циклов, но в этой главе, да и во всей книге, я буду рассматривать только одну форму цикла – **`while`**.

Центральное место в рассуждениях о любом цикле занимает *инвариант цикла*. Начнем со спецификаций циклов, затем перейдем к реализации циклов, обсуждению условия завершимости и множеству примеров.

11.0. Спецификации циклов

Спецификация цикла состоит из двух частей. *Инвариант цикла* ограничивает пространство состояний, в которых цикл может находиться. *Условие цикла* управляет продолжением выполнения шагов и его завершением.

В предварительном состоянии цикла существует условие соблюдения инварианта. Я буду называть его *условием использования цикла*. *Действие* цикла заключается в переходе из одного состояния, в котором сохраняется инвариант, в состояние, в котором сохраняются и инвариант, и отрицание условия продолжения.

11.0.0. Нетехнические примеры

Давайте посмотрим, как работает цикл. Вот иллюстративная спецификация цикла:

```
while not in Seattle
  invariant on the train
```

Все, что следует за ключевым словом **while**, является условием продолжения цикла, а все, что следует за ключевым словом **invariant**, – это инвариант. Цикл также имеет *тело цикла*, выполнение которого повторяется до тех пор, пока выполняется условие продолжения цикла. Однако в этом разделе я намеренно буду опускать тело цикла, чтобы мы могли сосредоточиться только на спецификации.

Чтобы использовать этот цикл, согласно условию использования цикла мы должны изначально находиться в поезде (`on the train`). Соответственно, нам нужно подготовить код, предшествующий циклу, чтобы попасть в поезд. По завершении цикла, как определяется инвариантом и отрицанием условия цикла, мы одновременно будем находиться в поезде и в Сиэтле. Вслед за циклом следует добавить код, который поможет нам покинуть поезд и начать наслаждаться пребыванием в городе.

Обратите внимание, что инвариант говорит только о том, что мы находимся в поезде, поэтому согласно спецификации цикла мы можем находиться в разных вагонах на входе в цикл и выходе из него. Также обратите внимание, что нам разрешено двигаться к Сиэтлу, пока мы находимся в поезде. То есть на входе в цикл должен сохраняться инвариант цикла, а условие цикла не имеет значения. По завершении цикла сохраняются как инвариант, так и отрицание условия цикла.

Вот еще один пример – цикл выполнения работы по дому:

```
while homework is not done
  invariant awake
```

Чтобы войти в этот цикл, вы должны пробудиться (`awake`), даже если все работы по дому (`homework`) уже сделаны. По завершении цикла вы уже будете проснувшимися и сделавшими всю работу по дому. Сравните эту спецификацию цикла со следующей:

```
while awake
  invariant homework is not done
```

Чтобы войти в этот цикл у вас должна иметься невыполненная работа по дому, а по окончании цикла вы уснете, а работа по дому так и не будет сделана.

11.0.1. Логика Флойда для спецификаций циклов

Выше я объяснил правила спецификации циклов простыми словами. Но мы можем формально описать их, используя логику Флойда, как это делалось в главе 2. Для логических выражений `B` и `J` семантика спецификации цикла задается тройкой Хоара:

```
{{ J }}
while B
  invariant J
{{ J && !B }}
```

Здесь говорится, что для входа в цикл необходимо доказать, что его инвариант изначально выполняется. Это условие использования цикла. После цикла инвариант и отрицание условия цикла сохраняются.

11.0.2. Числовые примеры

Вот пример спецификации цикла, в котором используются целые числа:

```
while x < 300
  invariant x % 2 == 0
```

Чтобы использовать этот цикл, нужно доказать, что `x` – четное число. По завершении цикла можно предположить, что `x` – по-прежнему четное число и `300 <= x`.

А что можно сказать об этом цикле?

```
while x % 2 == 1
  invariant 0 <= x <= 100
```

Этот цикл требует доказать, что изначально `x` находится в диапазоне от `0` до `100` включительно, а после завершения цикла можно быть уверенными, что `x` – четное число от `0` до `100` включительно.

Вот спецификация цикла с предшествующей инструкцией присваивания и последующей инструкцией **`assert`**:

```
x := 2;
while x < 50
  invariant x % 2 == 0
assert 50 <= x && x % 2 == 0;
```

Предварительное условия входа в цикл `x % 2 == 0` выполняется и доказуемо, поскольку в этот момент `x` имеет значение `2`. После выхода из цикла мы знаем, что инвариант и отрицание условия цикла выполняются. Именно это условие и проверяет утверждение **`assert`**, поэтому утверждение также доказуемо.

Вот еще один пример:

```
x := 0;
while x % 2 == 0
  invariant 0 <= x <= 20
assert x == 19; // недоказуемо
```

На входе в цикл x равен 0, поэтому инвариант, требующий, чтобы значение x находилось в диапазоне от 0 до 20, выполняется. Для доказательства утверждения **assert** нам нужно доказать:

```
0 <= x <= 20 && x % 2 != 0  ==>  x == 19
```

Мы не можем доказать это следствие. Например, x может быть равно 3 при выходе из цикла, поскольку 3 – нечетное число и находится в диапазоне от 0 до 20.

Упражнение 11.0

Для каждой из следующих спецификаций цикла укажите, выполняется ли условие использования цикла и можно ли доказать выполнение утверждения **assert**, следующего за циклом:

а)
```
x := 0;
while x != 100
  invariant true
assert x == 100;
```

б)
```
x := 20;
while 10 < x
  invariant x % 2 == 0
assert x == 10;
```

в)
```
x := 20;
while x < 20
  invariant x % 2 == 0
assert x == 20;
```

г)
```
x := 3;
while x < 2
  invariant x % 2 == 0
assert x % 2 == 0;
```

д)
```
if 50 < x < 100 {
  while x < 85
    invariant x % 2 == 0
  assert x < 85 && x % 2 == 1;
}
```

е)
```
if 0 <= x {
  while x % 2 == 0
    invariant x < 100
  assert 0 <= x;
}
```

ж)
```
x := 0;
while x < 100
  invariant 0 <= x < 100
assert x == 25;
```

11.0.3. Достижение равенства

При использовании циклов часто требуется, чтобы перед первой итерацией переменная получала значение, равное началу интервала и после последней итерации, – значение, равное концу интервала. Например, мы можем инициализировать переменную i значением 0 и выполнять цикл до тех пор, пока значение i не достигнет 100. В таких ситуациях мы часто называем переменную i *счетчиком цикла* или *переменной цикла*.

Вот пример такого цикла:

```
i := 0;
while i != 100
  invariant 0 <= i <= 100
assert i == 100;
```

Отталкиваясь только от отрицания условия цикла, мы можем доказать это утверждение **assert**, т. е. доказательство этого утверждения не зависит от инварианта. Например, в следующем фрагменте утверждение **assert** тоже доказуемо:

```
i := 0;
while i != 100
  invariant true
assert i == 100;
```

Другой вариант спецификации первого цикла:

```
i := 0;
while i < 100
  invariant 0 <= i <= 100
assert i == 100;
```

Это утверждение **assert** тоже доказуемо. Но здесь, в отличие от двух предыдущих примеров, условие цикла имеет вид `i < 100`, поэтому невозможно доказать утверждение, просто отрицая это условие (`100 <= i`). Вот тут-то и появляется инвариант `i <= 100`. Из `100 <= i` (отрицание условия цикла) и `i <= 100` (инвариант) можно заключить, что `i == 100`, и тем самым доказать утверждение **assert**.

Теперь рассмотрим четвертый пример:

```
i := 0;
while i < 100
  invariant true
assert i == 100; // недоказуемо
```

Это утверждение **assert** недоказуемо, потому что о цикле нам известно только, что **true** `&& 100 <= i`.

Из этих четырех небольших примеров следует вывод, что спецификацию цикла, достигающего определенного значения, можно написать несколькими способами. Если условие продолжения цикла имеет вид `i != 100`, то мы сразу можем доказать, что по завершении цикла `i` равно `100`. Если условие цикла имеет вид `i < 100`, то инвариант цикла дополнительно должен предусматривать `i <= 100`, чтобы мы могли доказать, что `i == 100` по завершении цикла.

Более того, как показывает мой опыт, понимание этих четырех примеров и причин, почему каждый из них верифицируется или не верифицируется, является ключом к работе с циклами. Все, что я сказал до сих пор, основано на *спецификации* цикла (инвариант и условие цикла). И действительно, в рассуждениях о цикле используется его спецификация, а не тело. Постарайтесь не забывать об этом, когда я буду знакомить с телами циклов в разделе 11.1.

Упражнение 11.1

Для каждой следующей программы укажите возможное значение `i` типа **`int`** после завершения цикла, показывающее, что утверждение недоказуемо:

а)
```
i := 0;
while i < 100
  invariant 0 <= i
assert i == 100;
```

б)
```
i := 100;
while 0 < i
  invariant true
assert i == 0;
```

в)
```
i := 0;
while i < 97
  invariant 0 <= i <= 99
assert i == 99;
```

г)
```
i := 22;
while i % 5 != 0
  invariant 10 <= i <= 100
assert i == 55;
```

Упражнение 11.2

Для каждой программы из упражнения 11.1 усильте инвариант так, чтобы он сохранялся при входе в цикл и был достаточным для доказательства утверждения **`assert`**.

11.0.4. Отношения между переменными

Циклы становятся гораздо интереснее, когда они включают более одной переменной. Вот спецификация цикла, где инвариант цикла связывает две переменные:

```
x, y := 0, 0;
while x < 300
  invariant 2 * x == 3 * y
assert 200 <= y;
```

Следуя правилу спецификации циклов, отношение между `x` и `y`, указанное в инварианте, сохраняется по завершении цикла, как и отрицание условия цикла, т. е. `300 <= x`. Это позволяет заключить, что по завершении цикла условие `200 <= y` выполняется, и доказать утверждение **`assert`**.

Вот еще один пример:

```
x, y := 0, 191;
while !(0 <= y < 7)
  invariant 7 * x + y == 191
assert x == 191 / 7 && y == 191 % 7;
```

Эта программа вычисляет целочисленное частное и остаток от деления `191` на `7`, как подтверждает (доказуемое) утверждение **`assert`**. Вариантом этой спецификации цикла является

```
x, y := 0, 191;
while 7 <= y
  invariant 0 <= y && 7 * x + y == 191
assert x == 191 / 7 && y == 191 % 7;
```

Инвариант этого цикла ограничивает y неотрицательными значениями, поэтому мы можем упростить условие цикла.

В качестве заключительного примера рассмотрим следующую программу, которая вычисляет сумму первых 33 натуральных чисел:

```
n, s := 0, 0;
while n != 33
  invariant s == n * (n - 1) / 2
```

Благодаря знакомому арифметическому тождеству инвариант сообщает, что s – это сумма первых n натуральных чисел. Мы можем сделать вывод, что по завершении цикла n равно 33 и, следовательно, что s – это сумма первых 33 натуральных чисел.

11.0.5. Фреймы циклов

Рассмотрим следующую спецификацию цикла, где цикл вычисляет r как целочисленный квадратный корень из N:

```
r, N := 0, 104;
while (r+1)*(r+1) <= N
  invariant 0 <= r && r*r <= N
assert 0 <= r && r*r <= N < (r+1)*(r+1);
```

Прежде чем продолжить, давайте остановимся ненадолго и оценим красоту этого фрагмента программы. Из того факта, что r является целочисленным квадратным корнем из N, следует, что N лежит в диапазоне от квадрата r до квадрата r+1, а само значение r неотрицательно. Нам нужно, чтобы эти три неравенства были доказуемыми после цикла. Чтобы выразить это в спецификации цикла, можно поместить два условия в инвариант и отрицание третьего условия в условие продолжения цикла. Какая красота!

Построив спецификацию цикла из частей утверждения **assert**, мы убедились, что утверждение в нашем примере доказуемо. Но можно ли заключить, что после цикла r равно 10? Если по завершении цикла N по-прежнему равно 104, то тогда, конечно же, r будет равно 10. Но в спецификации цикла не сказано, что N по-прежнему будет равно 104. Например, спецификация цикла допускает, что после выхода из цикла r и N будут иметь значения 2 и 5 соответственно.

Из этого наблюдения следует, что в наших спецификациях цикла есть явное упущение: они не говорят, какие переменные разрешено изменять, а какие должны оставаться неизменными. Часть спецификации, определяющая, что можно изменить, а что следует оставить без изменений, называется *фреймом* – в нашем контексте *фреймом цикла*. Мы могли бы явно

добавить фрейм цикла к его инварианту и условию продолжения, которые уже являются частью спецификации. Но не будем делать этого, потому что уже в следующем разделе я изложу вам свой план по выяснению фрейма цикла путем рассмотрения его реализации. А пока просто запомните, что существует такая вещь, как фрейм цикла, который является частью спецификации, и если я не оговариваю иного, то неявно фрейм цикла позволяет изменять все локальные переменные.

11.1. Реализации циклов

Если вам уже доводилось писать циклы (а я полагаю, что у вас есть такой опыт), то после прочтения предыдущего раздела у вас может возникнуть вопрос: являются ли циклы, о которых я говорю, теми циклами, с которыми вы знакомы? Да, это одно и то же. Но до сих пор я говорил только о *спецификациях* циклов, а циклы, которые вы писали, вероятно, являются *реализациями*. Давайте объединим эти два понятия. Поскольку сейчас мы переходим к более знакомой теме, не забывайте о том, что только что узнали о спецификациях циклов.

Цикл получил свое название из-за того, что его спецификация реализуется путем повторения шагов. В частности, цикл многократно выполняет свое тело, пока выполняется условие продолжения цикла. Тело цикла – это его *реализация*. Синтаксически реализация определяется в фигурных скобках после спецификации цикла.

11.1.0. Деление с остатком

В качестве первого примера вернемся к циклу из предыдущего раздела, вычисляющему частное и остаток от деления `191` на `7`. Вот спецификация этого цикла, включая инициализирующее присваивание перед циклом и утверждение **`assert`** после цикла:

```
x, y := 0, 191;
while 7 <= y
  invariant 0 <= y && 7 * x + y == 191
assert x == 191 / 7 && y == 191 % 7;
```

Тело цикла состоит из последовательности инструкций, поддерживающей инвариант цикла при условии, что выполняется условие цикла для исходного состояния. То есть, начиная с состояния, в котором выполняются и инвариант, и условие цикла, тело должно создать состояние, в котором сохраняется инвариант. Давайте подумаем, как это можно сделать для программы, вычисляющей остаток от деления, т. е. что можно подставить на место **`?`** в следующей тройке Хоара:

```
{{ 0 <= y && 7 * x + y == 191 && 7 <= y }}
?
{{ 0 <= y && 7 * x + y == 191 }}
```

Тело цикла

Заменить **?** можно, например, инструкцией увеличения x на 1. При этом, возможно, нам также придется изменить y. Используя вычисления слабейших предусловий для операций присваивания из раздела 2.3 и нотацию записи доказательств из раздела 2.3.1, мы имеем:

```
{{ 0 <= y && 7 * x + y == 191 && 7 <= y }}
?
{{ 0 <= y && 7 * (x + 1) + y == 191 }}
x := x + 1
{{ 0 <= y && 7 * x + y == 191 }}
```

Формулу в середине можно упростить:

```
  7 * (x + 1) + y == 191
=
  7 * x + 7 + y == 191
```

Если в этой формуле заменить y на `y - 7`, то мы получим то же условие, что и в начале тройки Хоара. Итак, у нас есть:

```
{{ 0 <= y && 7 * x + y == 191 && 7 <= y }}
{{ 0 <= y - 7 && 7 * x + 7 + (y - 7) == 191 }}
y := y - 7
{{ 0 <= y && 7 * x + 7 + y == 191 }}
{{ 0 <= y && 7 * (x + 1) + y == 191 }}
x := x + 1
{{ 0 <= y && 7 * x + y == 191 }}
```

Желательно перепроверить это вычисление, тем более что в направлении подстановки легко запутаться. Начиная с *последней* из этих строк и читая вверх, всякий раз сталкиваясь с присваиванием, мы заменяем переменную слева выражением справа (меняем x на `x + 1` и y на `y - 7`), и всякий раз, когда имеются две строки без программной инструкции между ними, мы должны проверить, что первая подразумевает вторую. (При необходимости прочитайте еще раз раздел 2.3.1.)

Возможно, такое объяснение показалось вам немного формальным. Давайте теперь повторим то же самое, но уже на конкретном примере. Мы должны написать инструкцию, которая поддерживает условие `7 * x + y == 191` (наряду с различными условиями диапазона и т. п.). Если мы увеличим x на 1, то из этой формулы следует, что нам нужно уменьшить y на 7. Вот что значит сохранить инвариант.

Вот наша программа:

```
x, y := 0, 191;
while 7 <= y
  invariant 0 <= y && 7 * x + y == 191
```

```
{
  y := y - 7;
  x := x + 1;
}
assert x == 191 / 7 && y == 191 % 7;
```

Рассматривая только тело цикла, трудно догадаться, что эта программа вычисляет целочисленный квадратный корень. Но благодаря инвариантам цикла (и помощи верификатора) мы можем быть уверены, что цикл выполняет постусловие программы.

Переходим к ответу

Реализовать тело цикла, осуществляющего деление с остатком, можно несколькими способами. Вот тройка Хоара, показывающая другой способ, удовлетворяющий инварианту:

```
{{ 0 <= y && 7 * x + y == 191 && 7 <= y }}
{{ true }}
{{ 0 <= 2 && 7 * 27 + 2 == 191 }}
x, y := 27, 2
{{ 0 <= y && 7 * x + y == 191 }}
```

Итак, перед нами еще один способ реализации нашего цикла:

```
x, y := 0, 191;
while 7 <= y
  invariant 0 <= y && 7 * x + y == 191
{
  x, y := 27, 2;
}
assert x == 191 / 7 && y == 191 % 7;
```

Ускоряем в два раза

Давайте еще раз вернемся к первой программе с телом цикла и рассмотрим какой-нибудь нерабочий случай. Например, попытаемся объединить две итерации цикла в одну. Вместо увеличения `x` на `1` и уменьшения `y` на `7` попробуем увеличить `x` на `2` и уменьшить `y` на `14`. Тогда тройка Хоара, выражающая доказательство корректности реализации цикла, будет выглядеть следующим образом:

```
{{ 0 <= y && 7 * x + y == 191 && 7 <= y }}
{{ 14 <= y && 7 * x + y == 191 }} // ошибка: это условие не следует
                                  // из предыдущего
{{ 0 <= y - 14 && 7 * (x + 2) + (y - 14) == 191 }}
x, y := x + 2, y - 14
{{ 0 <= y && 7 * x + y == 191 }}
```

Здесь я добавил еще две аннотации над инструкцией присваивания. Наша «более быстрая» реализация удовлетворяет условию `7 * x + y == 191`, но требует доказать соответствие условию `14 <= y`, которое не следует из предшествующего. Итак, эта попытка ускорить работу цикл некорректна.

Упражнение 11.3

Добавьте инструкцию **if** в тело цикла, где одна ветвь выполняет инструкцию `x, y := x + 2, y - 14`, а другая – инструкцию `x, y := x + 1, y - 7`. Какое условие нужно указать в инструкции **if**, чтобы сделать цикл корректным?

Самый простой способ сохранить соответствие какому-либо условию

Если требуется лишь обеспечить соответствие условию, которое уже выполняется, то нет более простого способа, чем ничего не делать. Такой подход действительно сохранит инвариант любого цикла. Но никакое количество повторений такого тела цикла не приблизит нас к состоянию, в котором условие цикла перестанет выполняться. А если этого не произойдет, то такой цикл никогда не завершится.

Завершение цикла важно, но обсуждение этого вопроса я отложу до раздела 11.2.

11.1.1. Формальность в отношении сохранения инварианта цикла

Для цикла

```
while B
  invariant J
{
  Body
}
```

проверяемое условие, необходимое для поддержания инварианта цикла, фиксируется следующей тройкой Хоара:

```
{{ J && B }}
Body
{{ J }}
```

Обратите внимание, что здесь нет повторения тела. Тройка Хоара содержит только предположение `J && B` и больше ничего! Чтобы доказать, что `J` имеет место после завершения цикла, нужно усилить инвариант. Конечно, если усилить инвариант цикла `J` до некоторого `J'`, то вам также придется доказать, что не только `J`, но и `J'` сохраняется после `Body`.

В дополнение к этому доказательству сохранения инварианта необходимо также доказать завершимость цикла. Как это сделать, я покажу в разделе 11.2.

11.1.2. Вычисление сумм

Рассмотрим еще один пример из предыдущего раздела, суммирующий первые 33 натуральных числа. Вот спецификация:

```
while n != 33
  invariant s == n * (n - 1) / 2
```

Чтобы реализовать этот цикл, нужно заменить **?** в следующей тройке Хоара:

```
{{ s == n * (n - 1) / 2 && n != 33 }}
?
{{ s == n * (n - 1) / 2 }}
```

Увеличение n на 1 кажется разумным шагом. Давайте выясним, как этот шаг повлияет на s:

```
  (s == n * (n - 1) / 2)⟦n := n + 1⟧
=      { подстановка }
  s == (n + 1) * n / 2
=
  s == (n*n + n) / 2
=
  s == (n*n - n + 2*n) / 2
=
  s == (n*n - n) / 2 + 2*n / 2
=
  s == n * (n - 1) / 2 + n
```

Согласно инварианту цикла первоначально s получает значение выражения n * (n - 1)/2, поэтому нам нужно увеличить s на n. Таким образом, программа целиком, включая присваивание значений переменным n и s перед циклом для соответствия инварианту, выглядит следующим образом:

```
n, s := 0, 0;
while n != 33
  invariant s == n * (n - 1) / 2
{
  s := s + n;
  n := n + 1;
}
```

Упражнение 11.4

Присвойте переменным другие начальные значения, корректные для приведенного выше цикла.

Упражнение 11.5

Напишите инициализирующую инструкцию присваивания и реализацию цикла для следующих спецификаций:

а)
```
while x < 300
    invariant x % 2 == 0
```

б)
```
while x % 2 == 1
    invariant 0 <= x <= 100
```

Упражнение 11.6

Взгляните на следующий фрагмент программы:

```
x := 0;
while x < 100
{
  x := x + 3;
}
assert x == 102;
```

Напишите инвариант цикла, который выполняется изначально, поддерживается телом цикла и позволяет доказать утверждение `assert` после цикла.

11.1.3. Вывод фреймов цикла

В разделе 11.0.5 я упоминал, что в спецификации также необходимо указать фрейм цикла, определяющий, что разрешено изменять, а что должно остаться неизменным. На данный момент у меня есть два замечания по этому поводу.

Во-первых, входные параметры в Dafny не могут изменяться. Отсюда следует, что любой входной параметр, упомянутый в спецификации цикла (или, если уж на то пошло, любой входной параметр, не указанный в спецификации цикла), не должен изменяться циклом.

Во-вторых, Dafny может определить, какие локальные переменные (и выходные параметры) находятся внутри фрейма цикла. Он делает это не на основании явной спецификации цикла, а путем анализа реализации цикла. Точнее говоря, локальная переменная находится внутри фрейма цикла, если его тело цикла содержит инструкцию присваивания переменной. На практике это означает, что, исследовав спецификацию и тело цикла, верификатор сможет определить, что любая локальная переменная, которой ничего не присваивается в теле цикла, сохранит свое первоначальное значение после выхода из цикла. А любую локальную переменную, которой цикл присваивает значение, нужно упомянуть в инварианте цикла, иначе

верификатор не будет иметь никакой информации о ее состоянии после цикла.

Как это делается, иллюстрирует следующий пример:

```
method LoopFrameExample(X: int, Y: int)
  requires 0 <= X <= Y
{
  var i, a, b := 0, X, Y;
  while i < 100 {
    i, b := i + 1, b + X;
  }
  assert a == X;
  assert Y <= b; // недоказуемо без инварианта цикла
}
```

В теле этого цикла ничего не присваивается переменной `a`, а переменная `X` является входным параметром (и потому ей тоже ничего не присваивается), поэтому утверждение **`assert`** `a == X` доказуемо. Напротив, переменной `b` присваивается некоторое значение в теле цикла, поэтому без инварианта цикла, упоминающего `b`, верификатор ничего не сможет сказать о состоянии `b` после цикла. Для доказательства второго утверждения **`assert`** нужно добавить

```
invariant Y <= b
```
или
```
invariant b == Y + i * X
```

в спецификации цикла. Такой инвариант цикла доказуем без дополнительного инварианта `0 <= X`, поскольку `X` является входным параметром и не может изменяться.

11.2. Завершимость цикла

Существует еще одно свойство циклов, которое необходимо доказать, – их завершимость. Мы должны показать, что само тело цикла завершится, что завершится любой рекурсивный вызов объемлющего метода из тела цикла, а также что итерации цикла в конечном итоге завершатся. Другими словами, мы должны показать невозможность бесконечного повторения или *бесконечного цикла*.

Чтобы показать невозможность бесконечного повторения, итерации цикла маркируются элементами фундированного порядка, так же как невозможность бесконечной рекурсии показывается путем маркировки вызовов функций и методов элементами фундированного порядка. Но вместо демонстрации уменьшения элемента в вызываемой функции по отношению к вызывающей мы должны показать его уменьшение при переходе от одной итерации к другой. Это достигается путем вычисления метки

следующей итерации в момент достижения границы тела цикла. Проще говоря, мы проверяем, вызывает ли тело уменьшение оценочной функции завершимости цикла.

Оценочная функция завершимости цикла объявляется с помощью инструкции **decrease**. Для цикла

```
while B
  invariant J
  decreases D
{
  Body
}
```

проверяемое условие, доказывающее завершимость, выраженное как тройка Хоара, выглядит так:

```
{{ J && B }}
ghost var d₀ := D;
Body
{{ d₀ ≻ D }}
```

Для удобства фиксации значения `D` на входе в тело цикла, чтобы иметь возможность снова обратиться к нему в постусловии тройки Хоара, я ввел призрачную локальную переменную d_0. Тройка Хоара гласит, что значение d_0 должно превышать значение `D` после выполнения тела цикла.

11.2.0. Завершимость деления с остатком

В разделе 11.1.0 мы рассмотрели несколько реализаций цикла, вычисляющего частное и остаток от деления чисел `191` и `7`. Давайте проверим завершимость некоторых попыток реализации этой спецификации цикла.

Простое тело

Тело нашего самого простого цикла:

```
y := y - 7;
x := x + 1;
```

Поскольку в каждой итерации значение `y`, выберем эту переменную в качестве оценки завершимости. Далее нам предстоит доказать:

```
{{ 0 <= y && 7 * x + y == 191 && 7 <= y }}
ghost var d0 := y;
y := y - 7;
x := x + 1;
{{ d0 ≻ y }}
```

Мы работаем с целыми числами, поэтому, согласно определению фундированного порядка $\succ$ о целых числах (раздел 3.2), мы должны доказать:

```
y < d₀ && 0 <= d₀
```

Первый конъюнкт выполняется, потому что `y == d₀ - 7`, а второй вытекает из инварианта `0 <= y` (и он также следует из условия продолжения цикла `7 <= y`). Готово!

Вот как мы запишем весь цикл – спецификацию, реализацию и инициализирующую инструкцию:

```
x, y := 0, 191;
while 7 <= y
  invariant 0 <= y && 7 * x + y == 191
  decreases y
{
  y := y - 7;
  x := x + 1;
}
```

Быстрое тело

Вспомним реализацию цикла деления с остатком, быстро возвращающую ответ:

```
x, y := 27, 2;
```

Что здесь уменьшается? Чтобы увидеть это более отчетливо, ниже приводится проверяемое условие в форме тройки Хоара, где два вопросительных знака нужно заменить одним и тем же выражением:

```
{{ 0 <= y && 7 * x + y == 191 && 7 <= y }}
ghost var d0 := ?;
x, y := 27, 2;
{{ d0 > ? }}
```

Обратите внимание, что тройка Хоара позволяет нам начать с `7 <= y`. Следовательно, `decreases y` позволяет доказать завершимость и этого цикла.

Смена направлений

Вот пример реализации цикла, которую мы еще не пробовали:

```
while 7 <= y
  invariant 0 <= y && 7 * x + y == 191
{
  x, y := x - 1, y + 7;
}
```

Вместо увеличения `x` и уменьшения `y` тело цикла действует в точности наоборот. Оно по-прежнему сохраняет инвариант. Но для доказательства

завершимости нужна нижняя граница x или верхняя граница y. У нас нет ни того ни другого. И действительно, если значение y изначально будет удовлетворять условию продолжения цикла, то оно будет удовлетворять ему всегда, поэтому этот цикл никогда не завершится.

Мы можем доказать, что этот цикл не завершается. Для этого инициализируем y значением, удовлетворяющим условию продолжения цикла, и добавим `7 <= y` как инвариант:

```
x, y := 0, 191;
while 7 <= y
  invariant 0 <= y && 7 * x + y == 191
  invariant 7 <= y
{
  x, y := x - 1, y + 7;
}
```

(По аналогии с множественными инструкциями **requires** и **ensures** несколько инструкций **invariant** имеет тот же эффект, что и одна инструкция, объединяющая все условия.) Эта программа не завершается, но она сохраняет инвариант. В общем случае всякий раз, сталкиваясь с ситуацией, когда инвариант подразумевает условие цикла, это означает, что цикл никогда не завершитсяй[1].

11.2.1. Инструкции decreases по умолчанию для циклов

В разделе 3.4 я отмечал, что Dafny автоматически добавляет инструкцию **decreases** по умолчанию в рекурсивные функции и методы, если она не указана явно. Аналогично Dafny добавляет инструкцию **decreases** по умолчанию в любые циклы, где она не задана явно.

При построении инструкции **decreases** по умолчанию Dafny руководствуется до безумия простым и удивительно эффективным правилом: если условие цикла является арифметическим сравнением `E < F` или `E <= F`, то по умолчанию используется

```
decreases F - E
```

А если условие цикла является арифметическим сравнением `E != F`, то по умолчанию используется абсолютная разность между `E` и `F`:

```
decreases if E < F then F - E else E - F
```

Несмотря на простоту правила, Dafny с успехом справляется с доказательством завершимости большинства циклов, не требуя от вас никаких действий.

Нет нужды в точности запоминать правила построения инструкции **decreases** по умолчанию, потому что в любой момент можно навести ука-

[1] Если в цикле нет инструкции принудительного прерывания итераций, такой как **return** или **break**.

затель мыши на ключевое слово **while** в IDE и увидеть, что сконструировал Dafny. Если вам не понравится инструкция **decreases** по умолчанию (что случается редко и только если значение по умолчанию не способно доказать завершимость вашего цикла), то вы сможете просто написать свою.

11.3. В заключение о правилах оформления циклов

Давайте еще раз окинем взглядом все проверяемые условия цикла. Для цикла

```
while B
  invariant J
  decreases D
{
  Body
}
```

контекст, используемый им, можно представить, как

```
{{ J }}
while B
  invariant J
  decreases D
{
  Body
}
{{ J && !B }}
```

Этот контекст не имеет ничего общего с `Body` (и с `D`, если уж на то пошло), и корректность реализации цикла сводится к доказательству тройки Хоара:

```
{{ J && B }}
ghost var d₀ := D;
Body
{{ J && d₀ > D }}
```

не имеющей ничего общего с многократным выполнением `Body`.

Упражнение 11.7

Для каждого из следующих методов напишите инвариант цикла и (явную) инструкцию **decreases**, позволяющую доказать корректность метода.

```
method UpWhileLess(N: int) returns (i: int)
  requires 0 <= N
```

```
  ensures i == N
{
  i := 0;
  while i < N {
    i := i + 1;
  }
}

method UpWhileNotEqual(N: int) returns (i: int)
  requires 0 <= N
  ensures i == N
{
  i := 0;
  while i != N {
    i := i + 1;
  }
}

method DownWhileNotEqual(N: int) returns (i: int)
  requires 0 <= N
  ensures i == 0
{
  i := N;
  while i != 0 {
    i := i - 1;
  }
}

method DownWhileGreater(N: int) returns (i: int)
  requires 0 <= N
  ensures i == 0
{
  i := N;
  while 0 < i {
    i := i - 1;
  }
}
```

Вы должны написать инварианты для всех циклов, несмотря на готовность Dafny взять на себя часть вашей работы: в двух примерах из четырех он сможет сделать это автоматически. Тем не менее вы все-таки напишите их.

11.4. Целочисленный квадратный корень

Я завершу эту главу еще одним примером – программой извлечения целочисленного квадратного корня, с которой мы познакомились в разделе 11.0.5. Начнем со спецификации метода:

```
method SquareRoot(N: nat) returns (r: nat)
  ensures r * r <= N < (r + 1) * (r + 1)
```

Обратите внимание, что я ограничил `N` диапазоном натуральных чисел, чтобы не застрять в попытке вычислить квадратный корень из отрицательного числа. Я также объявил `r` с типом **`nat`**, чтобы избавиться от необходимости писать `0 <= r` в постусловии, а затем и в инварианте цикла.

11.4.0. Подход к решению задачи

В следующих главах мы познакомимся с некоторыми методами конструирования циклов для решения задач определенного вида. Вот первый такой метод.

> **Метод конструирования циклов 11.0** (без конъюнкта)
>
> Чтобы решить задачу вида `A && B`, выберите инвариант цикла `A` и условие продолжения `!B`, т. е. используйте такую спецификацию цикла:
>
> ```
> while !B
>   invariant A
> ```

Задача вычисления квадратного корня состоит из двух конъюнктов. По аналогии с примерами выше используем `r * r <= N` в качестве инварианта и отрицание `N < (r + 1) * (r + 1)` в качестве условия продолжения цикла. Чтобы изначально установить соответствие инварианту, присвоим `0` переменной `r`:

```
{
  r := 0;
  while (r + 1) * (r + 1) <= N
    invariant r * r <= N
```

Этот цикл можно реализовать, увеличив значение `r`:

```
  {
    r := r + 1;
  }
}
```

Вот и все! Полученная программа фактически выполняет линейный поиск правильного значения `r`.

11.4.1. Более эффективная программа

В каждой итерации наша программа вычисляет выражение `(r + 1) * (r + 1)`. Чтобы увеличить быстродействие, можно подумать о вычислении выражения на основе результата, полученного в предыдущей итерации, а инварианты помогут нам убедиться, что мы правильно преобразовали программу. Мы сделаем это, используя прием, который называется *построением цикла с помощью инварианта*.

Я хотел бы добавить переменную `s`, значение которой всегда будет равно результату `(r + 1) * (r + 1)`. В таком случае мы сможем выразить условие цикла как `s <= N`. Добавим эту переменную в инвариант цикла, чтобы воплотить желание в реальность:

```
invariant s == (r + 1) * (r + 1)
```

Очень хорошо. Теперь, когда этот инвариант включен в спецификацию цикла, мы должны обеспечить соответствие ему перед началом цикла. Поскольку изначально `r` равно `0`, инвариант цикла прямо говорит нам, что мы должны инициализировать `s` значением `1`:

```
var s := 1;
```

Чтобы доказать корректность, мы также обязаны поддерживать инвариант в теле цикла. Тело увеличивает значение `r`, поэтому мы можем вычислить слабейшее предусловие этого присваивания относительно нашего нового инварианта. Пойдем *обратным проходом* от инварианта цикла, который требует установить `s` перед присваиванием нового значения переменной `r`:

```
{{ s == (r + 1 + 1) * (r + 1 + 1) }}
r := r + 1
{{ s == (r + 1) * (r + 1) }}
```

С точки зрения арифметики мы имеем:

```
  (r + 1 + 1) * (r + 1 + 1)
=
  r*r + 4*r + 4
=
  r*r + 2*r + 1 + 2*r + 3
=
  (r + 1) * (r + 1) + 2*r + 3
```

Благодаря инварианту мы знаем, что `s` равно `(r + 1) * (r + 1)` при входе в тело цикла, поэтому нам остается только увеличить `s` на `2 * r + 3`.

Мы сначала определили новый инвариант, а затем подумали о том, как обеспечить соответствие ему.

Итак, вот наша окончательная программа вычисления целочисленного квадратного корня:

```
method SquareRoot(N: nat) returns (r: nat)
  ensures r * r <= N < (r + 1) * (r + 1)
{
  r := 0;
  var s := 1;
  while s <= N
    invariant r * r <= N
    invariant s == (r + 1) * (r + 1)
  {
    s := s + 2*r + 3;
    r := r + 1;
  }
}
```

Использованный нами метод конструирования программы можно описать так:

> **Метод конструирования циклов 11.1** (Построение цикла с помощью инварианта)
>
> Если задачу можно упростить, имея заранее вычисленное значение `Q`, то введите новую переменную `q` с намерением установить и поддерживать инвариант `q == Q`.

Я подробно расскажу об этом методе в разделе 12.1.

11.5. Итоги

Цикл состоит из двух частей: спецификации и реализации. Спецификация, в свою очередь, состоит из трех частей:

- условия цикла, управляющего повторным выполнением итераций;
- инварианта цикла, ограничивающего состояние, доступное для итераций цикла;
- фрейма цикла, который обычно определяется неявно.

Добавляя спецификацию цикла, вы берете на себя обязательство доказать соответствие условиям *использования* цикла (т. е. вы должны доказать, что инвариант цикла выполняется перед его использованием). Это даст вам уверенность, что по завершении цикла будут выполняться его инвариант и отрицание условия цикла.

Реализация цикла – это блочная инструкция. *Реализуя* цикл, вы должны показать, что тело цикла сохраняет соответствие инварианту из состояния, в котором выполняется условие продолжения цикла. Чтобы показать, что итерации рано или поздно завершатся, вы также должны уменьшать в теле некоторую оценочную функцию завершимости.

Многие инварианты циклов следуют типичным шаблонам. В этой главе я определил два таких шаблона (как методы конструирования циклов) и

покажу другие в последующих главах. Вы можете попробовать применить эти шаблоны при разработке своих циклов и инвариантов.

Циклы являются отличительной чертой императивного программирования, а инварианты циклов – важной частью доказательства корректности программ. Теперь, после знакомства с несколькими примерами циклов и инвариантов, я бы рекомендовал вернуться к началу этой главы и еще раз прочитать раздел, описывающий спецификации циклов без тела. Если при первом прочтении вам показалось странным рассуждать о цикле, игнорируя его тело, то теперь вы, возможно, начнете понимать, как спецификация цикла отделяет рассуждения об использовании цикла от рассуждений о его реализации.

Упражнение 11.8

Используя тип данных `List` и функцию `Length` из главы 6, напишите цикл, создающий список с заданным значением `d`, повторяющимся `n` раз. Докажите постусловие метода, которое гласит, что возвращаемый список имеет длину `n`.

Упражнение 11.9

Напишите метод `Duplicate`, принимающий список (см. главу 6) и возвращающий список в два раза длиннее. Элементами нового списка могут быть любые значения по вашему выбору – в этом упражнении важна только длина списка. Докажите постусловие метода, которое гласит, что возвращаемый список действительно в два раза длиннее исходного.

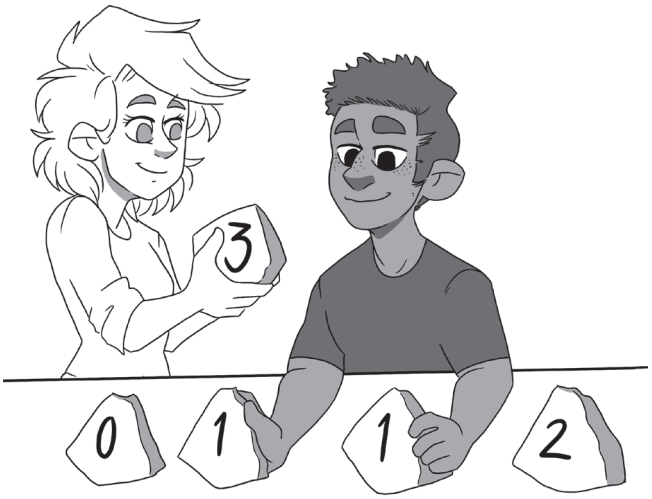
Примечания

Инварианты широко используются в доказательствах корректности программ. Например, в главе 10 мы видели инварианты структур данных, а в главе 16 познакомимся с инвариантами объектов. В этой главе были представлены инварианты циклов. С точки зрения обучения циклы служат удобным инструментом для освоения понятия инвариантов, поскольку не требуют писать много строк кода, чтобы поэкспериментировать с различными инвариантами цикла. Исследуя простые примеры из этой и следующей глав, постарайтесь понять, как инварианты используются для рассуждений о программах, не требуя знать, сколько итераций выполняет цикл.

Инварианты циклов являются наиболее распространенным способом рассуждений о циклах, но есть и другие способы доказательства корректности повторяющегося императивного поведения. Одно из мнений по этому поводу отражено в классической статье Эрика Хенера (Eric Hehner) [60].

Глава 12

Рекурсивные спецификации, итеративные программы



В спецификациях методов мы часто используем функции и определяем функции рекурсивно. Если реализовать такой метод с помощью цикла, то инвариант цикла тоже будет выражаться с помощью функций. В этой главе я приведу несколько примеров, демонстрирующих проблемы, которые возникают при таком соединении рекурсии и итерации.

12.0. Итеративное вычисление чисел Фибоначчи

В разделе 3.1 я определил знаменитую последовательность чисел Фибоначчи

```
0, 1, 1, 2, 3, 5, 8, ...
```

как рекурсивную функцию:

```
function Fib(n: nat): nat {
  if n < 2 then n else Fib(n - 2) + Fib(n - 1)
}
```

Эту функцию легко отладить и убедиться, что она корректно определяет последовательность чисел Фибоначчи, которая начинается с `0` и `1`, а все последующие числа определяются как сумма двух предыдущих. Однако такой способ вычисления чисел Фибоначчи не особенно эффективен. Давайте напишем гораздо более эффективную версию ($O(n)$ операций сложения для `Fib`(n)) с использованием цикла.

Используем функцию `Fib` в спецификации нашего метода `ComputeFib`:

```
method ComputeFib(n: nat) returns (x: nat)
  ensures x == Fib(n)
```

Обратите внимание, что `Fib` – это просто часть спецификации, и она не будет вызываться при выполнении `ComputeFib`. Другими словами, инструкция **`ensures`** является призрачной – она просто объясняет, что мы хотим проверить, и элиминируется компилятором. Фактически, учитывая, что мы не собираемся использовать функцию `Fib` в скомпилированной программе, имеет смысл объявить ее призрачной; именно так я и буду поступать в оставшейся части главы, определяя функции, предназначенные только для использования в спецификациях.

Мы реализуем `ComputeFib` с помощью цикла со счетчиком `i`, который будет изменяться в диапазоне от `0` до `n`. Вот как выглядит соответствующие инструкции инициализации и спецификация цикла:

```
{
  x := 0;
  var i := 0;
  while i != n
    invariant 0 <= i <= n
    invariant x == Fib(i)
}
```

Эта спецификация гласит, что по завершении цикла `i == n` и `x == Fib(n)`, а это значит, что мы на правильном пути: мы свели задачу реализации метода к задаче реализации цикла.

Только что мы применили краеугольный метод конструирования спецификации цикла на основе модификации постусловия: заменили константу `n` в постусловии счетчиком цикла `i`. Вот краткое описание этого метода.

> **Метод конструирования циклов 12.0** (Замена константы переменной)
>
> Для цикла с условием `...C...`, где `C` – выражение, которое остается постоянным на протяжении всего цикла, используйте переменную `k`, которую цикл будет изменять до тех пор, пока она не станет равной `C`, и сделайте `...k...` инвариантом цикла.

Применительно к нашей программе константа C в этом методе конструирования циклов – наш внутренний параметр n, а переменная k – счетчик цикла i.

В рамках реализации цикла мы сразу видим, что присваивание

```
i := i + 1;
```

поддерживает первый инвариант цикла и позволяет доказать завершимость.

Тело цикла также должно обновить x. Для большей эффективности мы можем отслеживать не одно, а два последовательных числа Фибоначчи. Одно мы сохраним в x, а другое в локальной переменной y. Но теперь у нас есть выбор. Мы можем спроектировать наш цикл с помощью инварианта:

```
invariant x == Fib(i) && y == Fib(i + 1)
```

или:

```
invariant y == Fib(i - 1) && x == Fib(i)
```

Последний не является корректным, когда i равно 0 (поскольку Fib принимает значения типа **nat**). Мы можем спасти ситуацию, записав этот инвариант как

```
invariant (i == 0 || y == Fib(i - 1)) && x == Fib(i)
```

Это не очень хорошая идея. Начав с более сложного инварианта, мы, скорее всего, получим более сложную программу. Инвариант цикла отражает структуру разрабатываемого цикла, поэтому, записав инвариант цикла до его реализации, можно сразу же заметить некоторые недостатки.

Итак, давайте воспользуемся инвариантом, где y – число Фибоначчи, следующее за x. Введем переменную y и инициализируем ее, чтобы удовлетворить инварианту:

```
var y := 1;
```

Мы уже решили, что будем увеличивать i в цикле, поэтому, чтобы получить необходимое условие перед обновлением i, пойдем в обратном направлении от инварианта цикла. Как мы видели ранее, для этого нужно заменить i на i + 1 в нужном нам условии после присваивания i. Это дает нам

```
{{ x == Fib(i) && y == Fib(i + 1) && i != n }}
?
{{ x == Fib(i + 1) && y == Fib(i + 1 + 1) }}
i := i + 1
{{ x == Fib(i) && y == Fib(i + 1) }}
```

где **?** показывает место, где нужно обновить x и y. Мы должны присвоить переменной x значение, равное Fib(i + 1). Это легко сделать, поскольку мы видим, что требуемое значение присваивается переменной y перед **?**. В то же время мы хотим присвоить переменной y значение, равное Fib(i + 2). По определению Fib это то же самое значение, что и Fib(i) + Fib(i + 1), и эти два члена хранятся в x и y перед **?**. Теперь мы знаем, что **?** следует заменить на

```
x, y := y, x + y;
```

Реализация ComputeFib завершена. Чтобы напомнить о том, что мы только что сделали, я покажу весь метод, включая тройки Хоара, которые мы использовали в доказательстве (за исключением доказательства завершимости, о котором Dafny позаботится автоматически в этой программе):

```
method ComputeFib(n: nat) returns (x: nat)
  ensures x == Fib(n)
{
  {{ true }}
  x := 0;
  var i := 0;
  var y := 1;
  {{ 0 <= i <= n && x == Fib(i) && y == Fib(i + 1) }}
  while i != n
    invariant 0 <= i <= n
    invariant x == Fib(i) && y == Fib(i + 1)
  {
    {{ 0 <= i <= n && x == Fib(i) && y == Fib(i + 1) && i != n }}
    {{ 0 <= i < n && x == Fib(i) && y == Fib(i + 1) }}
    {{ 0 <= i < n && y == Fib(i + 1) && x + y == Fib(i) + Fib(i + 1) }}
    x, y := y, x + y;
    {{ 0 <= i < n && x == Fib(i + 1) && y == Fib(i) + Fib(i + 1) }}
    {{ 0 <= i < n && x == Fib(i + 1) && y == Fib(i + 2) }}
    {{ 0 <= i + 1 <= n && x == Fib(i + 1) && y == Fib(i + 1 + 1) }}
    i := i + 1;
    {{ 0 <= i <= n && x == Fib(i) && y == Fib(i + 1) }}
  }
  {{ 0 <= i <= n && x == Fib(i) && y == Fib(i + 1) && i == n }}
  {{ x == Fib(n) }}
}
```

Упражнение 12.0

(*Усложненное*) Напишите императивную программу, которая вычисляет Fib(n), выполняя только $O(\log n)$ арифметических операций.

12.1. Квадрат числа Фибоначчи

В этом разделе я покажу мощный прием *построения цикла с помощью инварианта* (Метод конструирования циклов 11.1), в котором инварианты помогают постепенно совершенствовать программу. Мы уже использовали этот метод для небольшой оптимизации в разделе 11.4.1.

В качестве примера реализуем метод, вычисляющий квадрат числа Фибоначчи:

```
method SquareFib(N: nat) returns (x: nat)
  ensures x == Fib(N) * Fib(N)
```

Предположим, что программа будет выполняться на машине, не поддерживающей умножение – только сложение. Поэтому реализуем метод без использования умножения (но мы можем использовать умножение в спецификациях, поскольку они не компилируются). Я знаю, что только что описал метод `ComputeFib`, при реализации которого мы уже приняли определенные проектные решения, но давайте начнем заново и посмотрим, как работает построение цикла с помощью инварианта.

12.1.0. Простое начало

Нам уже знаком метод замены константы переменной, поэтому введем счетчик цикла `n` и напишем инвариант цикла, в котором константа постусловия `N` заменяется на `n`:

```
{
  x := 0;
  var n := 0;
  while n != N
    invariant 0 <= n <= N
    invariant x == Fib(n) * Fib(n)
}
```

Тело цикла будет увеличивать `n`, поэтому применим обратный проход, чтобы выяснить, что присвоить переменной `x`:

```
{{ x == Fib(n + 1) * Fib(n + 1) }}
n := n + 1
{{ x == Fib(n) * Fib(n) }}
```

По условию перед обновлением `n` мы должны присвоить переменной `x` квадрат `Fib(n + 1)`. В такой ситуации часто можно расширить функцию, применяемую к `n + 1`, и получить другое выражение через функцию, применяемую к `n`. Но мы не можем сделать этого с `Fib`, потому что ее рекурсивность определяется двумя предыдущими значениями, `Fib(n - 1)` и `Fib(n)`, и если `n - 1` даст отрицательное значение, мы не сможем вызвать для него `Fib`.

12.1.2. Желание

Но пусть вас это не волнует. Давайте просто пожелаем, чтобы квадрат `Fib(n + 1)` уже был вычислен в переменной, скажем, `y`. После фиксации желания в инварианте и определения начального значения `y` наша обновленная программа, удовлетворяющая этому инварианту, будет выглядеть так:

```
x := 0;
var n, y := 0, 1;
while n != N
  invariant 0 <= n <= N
  invariant x == Fib(n) * Fib(n)
  invariant y == Fib(n + 1) * Fib(n + 1)
{
  x, y := y, ?;
  n := n + 1;
}
```

Обратите внимание, что наша новая программа присваивает значение `y` переменной `x`, что и было причиной введения `y`.

Но какое значение присвоить переменной `y` в теле цикла, чтобы сохранить соответствие инварианту? Давайте снова выполним обратный проход:

```
{{ y == Fib(n + 2) * Fib(n + 2) }}
n := n + 1
{{ y == Fib(n + 1) * Fib(n + 1) }}
```

На этот раз, начиная с аргумента, имеющего значение не меньше `2`, мы вправе применить определение `Fib`. Поэтому перепишем выражение `Fib(n + 2) * Fib(n + 2)` в части, которые уже вычислили. Мы могли бы выполнить вычисления на бумаге или обратиться за помощью к верификатору, чтобы убедиться, что все сделали правильно. Чтобы сделать это на компьютере, необходимо опустить бессмысленное присваивание `y := ?`, которое я написал выше, – выяснив, что нужно присвоить переменной `y`, мы включим в программу правильную инструкцию присваивания `y`. Затем мы сможем написать следующее доказательное вычисление внутри тела цикла (о доказательных вычислениях я рассказывал в главе 5 и в части 1):

```
  calc {
    Fib(n + 2) * Fib(n + 2);
  ==  // определение Fib
    (Fib(n) + Fib(n + 1)) * (Fib(n) + Fib(n + 1));
  ==  // перекрестное умножение
    Fib(n) * Fib(n) + 2 * Fib(n) * Fib(n + 1) +
    Fib(n + 1) * Fib(n + 1);
  ==  // инвариант
    x + 2 * Fib(n) * Fib(n + 1) + y;
}
```

Инструкция **calc** доказала, что квадрат `Fib(n + 2)` можно выразить как сумму трех слагаемых. Первый член – это квадрат `Fib(n)`, который, согласно инварианту цикла, уже хранится в `x`. Третий член – это квадрат `Fib(n + 1)`, который при входе в тело цикла сохраняется в `y`. Если бы у нас была переменная, например `k`, для хранения среднего члена, то инструкция присваивания переменной `y` выглядела бы так:

```
y := x + k + y;
```

Инструкция **calc**, которую мы использовали в качестве блокнота, выполнила свою функцию, и теперь ее можно удалить. Но я советую оставить ее на месте, пока мы не закончим. На самом деле инструкцию **calc** можно вообще не удалять, поскольку она содержит понятное объяснение, для чего мы ввели `k`. Если вы решите сохранить ее в роли документации, то добавьте

```
==  // инвариант
  x + k + y;
```

в конец после введения `k`.

12.1.2. Еще одно желание

Наше желание привело к добавлению `k` с инвариантом

```
invariant k == 2 * Fib(n) * Fib(n + 1)
```

и начальным значением `0`. Значение, которое мы должны присвоить `k`, чтобы оно соответствовало приращению `n`, равно `2 * Fib(n + 1) * Fib(n + 2)`. Давайте напишем еще одну инструкцию **calc** в теле цикла, чтобы попытаться упростить это выражение:

```
calc {
  2 * Fib(n + 1) * Fib(n + 2);
==  // определение Fib
  2 * Fib(n + 1) * (Fib(n) + Fib(n + 1));
==  // распредлительное свойство арифметики
  2 * Fib(n + 1) * Fib(n) + 2 * Fib(n + 1) * Fib(n + 1);
}
```

Нам определенно везет! Первое из этих слагаемых – значение `k` на входе в тело цикла, а второе – в два раза больше значения `y`. То есть инструкция обновления `k` будет выглядеть как `k := k + y + y;`.

Итак, вот законченное тело нашего метода:

```
x := 0;
var n, y, k := 0, 1, 0;
while n != N
  invariant 0 <= n <= N
  invariant x == Fib(n) * Fib(n)
```

```
    invariant y == Fib(n + 1) * Fib(n + 1)
    invariant k == 2 * Fib(n) * Fib(n + 1)
  {
    x, y, k := y, x + k + y, k + y + y;
    n := n + 1;
  }
```

Инструкции **calc**, написанные нами во время разработки, были помещены в начало тела цикла, но я удалил их из этой окончательной версии программы.

Обратите внимание, что одновременное присваивание совершенно необходимо, потому что переменные обновляются относительно друг друга. В качестве альтернативы можно ввести временные переменные и выполнять присваивания последовательно:

```
var prevX := x;
var prevY := y;
x := prevY;
y := prevX + k + prevY;
k := k + prevY + prevY;
```

12.1.3. Рефлексия

Я почти уверен, что окончательная версия программы заставила вас удивиться. По крайней мере, я нахожу удивительным, что, в конце концов, все наши желания были удовлетворены уже имеющимися у нас значениями. Конечно, так бывает не всегда, тем не менее построение цикла с помощью инварианта – ценный прием, чтобы его попробовать.

Упражнение 12.1

Реализуйте

```
method Cube(n: nat) returns (c: nat)
  ensures c == n * n * n
```

с циклом, который повторяется `n` раз и использует только сложение (без умножения).

12.2. Степени двойки

Мы можем определить функцию, вычисляющую 2^n. Функция использует тот факт, что 2^0 `==` `1` и для любого другого показателя степени `n`: 2^n `==` `2 * 2^(n-1)`:

```
ghost function Power(n: nat): nat {
  if n == 0 then 1 else 2 * Power(n - 1)
}
```

Я намеренно объявил эту функцию призрачной, потому что мы будем использовать ее только в спецификациях.

Вот метод, вычисляющий `Power(n)`:

```
method ComputePower(n: nat) returns (p: nat)
  ensures p == Power(n)
```

Рассмотрим далее реализацию этого метода.

12.2.0. Обычный инвариант

Написать цикл, реализующий этот метод, несложно, поэтому давайте снова пройдемся по шагам, чтобы закрепить ваше понимание создания инвариантов и программ.

Главная наша идея – использовать цикл, в котором счетчик цикла `i` изменяется от `0` до `n`, сохраняя при этом соответствие условию `p == Power(i)`. Поскольку `i` начинается с `0`, инициализируем `p` значением `1`, чтобы обеспечить соответствие инварианту на входе в цикл.

```
{
  p := 1;
  var i := 0;
  while i != n
    invariant 0 <= i <= n
    invariant p == Power(i)
}
```

Упражнение 12.2

Как отреагирует верификатор, если:

а) заменить `p := 1` на `p := 2`?

б) заменить `p := 1` на `p := 2` и изменить инвариант на `p == Power(i + 1)`?

в) заменить `p := 1` на `p := 2` и `i := 0` на `i := 1`?

г) выполнить все действия, перечисленные в предыдущем пункте, и дополнительно заменить `i != n` на `i < n`?

Увеличим `i` на `1` в теле цикла. Это гарантирует завершимость и позволит нам аннотировать программу, чтобы сосредоточиться на обновлении `p` в цикле:

```
{{ 0 <= i < n && p == Power(i) }}
?
{{ 0 <= i + 1 <= n && p == Power(i + 1) }}
i := i + 1
      {{ 0 <= i <= n && p == Power(i) }}
```

Здесь и далее я сразу упрощу `0 <= i <= n && i != n` до полуоткрытого интервала `0 <= i < n`.

В аннотации, предшествующей вопросительному знаку, видно, что на входе в тело цикла p – это Power(i), и нам нужно присвоить ей Power(i + 1). По определению значение Power(i + 1) равно 2 * Power(i), поэтому инструкция обновления p должна выглядеть так:

```
p := 2 * p;
```

12.2.1. Альтернативный инвариант

Инвариант, который мы написали для p, фокусируется на том, что уже вычислено. Но это не единственный способ записи инварианта. Альтернативный способ – сосредоточиться на том, что еще предстоит сделать. Словами это можно выразить так: «Если умножить p на Power(n - i), то получится искомое». Соответствующая спецификация цикла записывается так:

```
p := 1;
var i := 0;
while i != n
  invariant 0 <= i <= n
  invariant p * Power(n - i) == Power(n)
```

Инструкции присваивания начальных значений переменным p и i приводят состояние в соответствие с инвариантом. После цикла i равно n, поэтому мы имеем:

```
p * Power(0) == Power(n)
```

По определению Power(0), левая часть этого уравнения упрощается до p, отсюда вытекает постусловие метода.

Конечно, не всегда можно сформулировать инварианты таким образом, но в некоторых случаях это единственный способ сформулировать некоторые инварианты.

> **Метод конструирования циклов 12.1** (Что еще предстоит сделать)
>
> Пытаясь решить задачу вида p == F(n), можно попробовать использовать счетчик цикла i, удовлетворяющий условию 0 <= i <= n, или инвариант «что сделано»:
>
> ```
> invariant p == F(i)
> ```
>
> или инвариант «что еще предстоит сделать»:
>
> ```
> invariant p ⊕ F(n - i) == F(n)
> ```
>
> где ⊕ – некоторая операция.

Упражнение 12.3

Обобщите функцию `Power` до любого основания. То есть определить функцию

```
ghost function Exp(b: nat, n: nat): nat
```

возвращающую b^n. Напишите метод `ComputeExp`, вычисляющий `Exp(b, n)` за $O(n)$ итераций.

Упражнение 12.4

Докажите следующие две леммы о функции `Exp` из упражнения 12.3:

```
lemma ExpAddExponent(b: nat, m: nat, n: nat)
  ensures Exp(b, m + n) == Exp(b, m) * Exp(b, n)

lemma ExpSquareBase(b: nat, n: nat)
  ensures Exp(b * b, n) == Exp(b, 2 * n)
```

Упражнение 12.5

(*Усложненное*) Напишите метод `FastExp`, вычисляющий функцию `Exp` из упражнения 12.3 за $O(\log n)$ итераций.

Здесь уместно дать несколько подсказок и высказать несколько предупреждений. Для применения инварианта цикла в форме «что еще предстоит сделать» используйте леммы из упражнения 12.4 и учтите следующий факт, касающийся целочисленного деления с остатком, справедливый для любого целого числа `k`:

```
k == 2 * (k / 2) + k % 2
```

Также имейте в виду, что это упражнение включает нелинейную арифметику. В линейной арифметике в операциях умножения всегда участвуют константы, например `3 * x`. Подобные выражения на самом деле являются просто удобным обозначением повторяющихся сложений, например `x + x + x`. В нелинейной арифметике в операции умножения участвует несколько переменных, например `x * y`. Нелинейная арифметика сложнее поддается автоматической верификации, а значит, вам потребуется проявить больше терпения. Чаще всего вам придется подождать несколько секунд, пока верификатор завершит работу, но иногда может понадобиться помочь ему достичь доказательства, добавляя некоторые арифметические утверждения, которые мы с вами принимаем как нечто само собой разумеющееся еще со школы.

12.3. Суммы

В следующем примере я использую функцию F, принимающую и возвращающую целое число. Пример не зависит от того, что делает F, поэтому я опущу ее тело. (Логики назвали бы это *неинтерпретируемой функцией*. Если вы решите скомпилировать и запустить этот пример, то добавьте тело в F. Но для простой верификации программы тело не нужно.)

```
function F(x: int): int
```

Врезка 12.0

Полуоткрытые интервалы очень удобны для рассуждений. Количество значений i в полуоткрытом интервале

```
2 <= i < 7
```

равно `7 - 2`, т. е. 5. Аналогично, поскольку диапазоны в Dafny определяют полуоткрытые интервалы, количество элементов в последовательности `a[2..7]` равно `7 - 2`. Ниже показано, как можно кратко записать вычисление суммы в полуоткрытом интервале:

$$2^2 + 3^2 + 4^2 + 5^2 + 6^2 = \sum_{i:=2}^{<7} i^2$$

Итак, чтобы получить размер полуоткрытого интервала, нужно вычесть нижнюю границу из верхней. Если интервал начинается с 0, то размер интервала равен его верхней границе. Например, следующий интервал содержит 23 значения i:

```
0 <= i < 23
```

12.3.0. Суммирование вверх и вниз

Функцию F можно применить к любому целому числу. Давайте применим ее к диапазону целых чисел от нижней границы lo до верхней hi и сложим полученные значения. В математической форме это записывается так:

$$\sum_{i:=lo}^{<hi} F(i).$$

Если в языке нет встроенной конструкции для ∑, то мы должны определить свою рекурсивную функцию для вычисления этого выражения. Сделать это можно двумя способами: рекурсивные вызовы могут смещать нижнюю границу вверх:

```
ghost function SumUp(lo: int, hi: int): int
  requires lo <= hi
```

```
    decreases hi - lo
  {
    if lo == hi then 0 else F(lo) + SumUp(lo + 1, hi)
  }
```

или верхнюю границу вниз:

```
  ghost function SumDown(lo: int, hi: int): int
    requires lo <= hi
    decreases hi - lo
  {
    if lo == hi then 0 else SumDown(lo, hi - 1) + F(hi - 1)
  }
```

Обратите внимание, что мы должны явно указать инструкцию **decreases** в обеих этих функциях.

В математической форме эти две ветви **else** выражают:

$$\sum_{i:=lo}^{>hi} F(i) = F(lo) + \sum_{i:=lo+1}^{>hi} F(i)$$

и

$$\sum_{i:=lo}^{>hi} F(i) = \left(\sum_{i:=lo}^{>hi-1} F(i) \right) + F(hi-1).$$

Если я написал их правильно, то обе функции вернут один и тот же результат. Давайте докажем, что это действительно так:

```
  lemma SameSums(lo: int, hi: int)
    requires lo <= hi
    ensures SumUp(lo, hi) == SumDown(lo, hi)
    decreases hi - lo
  {
    if lo != hi {
      PrependSumDown(lo, hi);
    }
  }

  lemma PrependSumDown(lo: int, hi: int)
    requires lo < hi
    ensures F(lo) + SumDown(lo + 1, hi) == SumDown(lo, hi)
    decreases hi - lo
  {
  }
```

Все обращения к индуктивным предположениям в этих двух леммах могут быть обработаны механизмом автоматической индукции Dafny, но для

этого нужно добавить инструкции **decreases** (совпадающие с инструкциями **decreases** в соответствующих функциях). Если вы забудете это сделать, то получите от верификатора сообщение о невозможности доказать леммы. Вы также обнаружите необходимость в этих инструкциях **decreases**, когда попытаетесь рекурсивно вызвать леммы в доказательствах, написанных вручную.

Упражнение 12.6

Как выглядят инструкции **decreases** по умолчанию для функций `SumUp` и `SumDown`? Объясните, почему они не способствуют доказательству завершимости.

Упражнение 12.7

Добавьте в обе леммы атрибут {`:induction` **false**} и напишите доказательства вручную.

Упражнение 12.8

Не опираясь на автоматическую индукцию, докажите:

```
lemma AppendSumUp(lo: int, hi: int)
  requires lo < hi
  ensures SumUp(lo, hi - 1) + F(hi - 1) == SumUp(lo, hi)
```

12.3.1. Вычисление сумм итеративным способом

Написать цикл вычисления суммы тоже можно двумя способами. Ниже приводится реализация цикла, исходящая из предположения, что `lo <= hi`, в которой счетчик цикла увеличивается. Я буду называть этот цикл `LoopUp`:

```
s, i := 0, lo;
while i != hi
  invariant lo <= i <= hi
{
  s := s + F(i);
  i := i + 1;
}
```

А вот реализация цикла, в которой счетчик цикла уменьшается. Я буду называть этот цикл `LoopDown`:

```
s, i := 0, hi;
while i != lo
  invariant lo <= i <= hi
{
  i := i - 1;
  s := s + F(i);
```

```
}
```

Чтобы доказать, что эти циклы вычисляют сумму и сохраняют ее в `s`, в каждый цикл нужно также добавить инвариант, описывающий свойства `s`.

12.3.2. Верификация суммирования

Сосредоточимся на `LoopUp`. Используя метод замены константы (`hi`) переменной (`i`), получаем разумный инвариант цикла, описывающий свойства `s`:

```
invariant s == SumUp(lo, i)
```

Этот инвариант выполняется на входе в цикл, потому что `i` инициализируется значением `lo`. Это позволяет нам заключить, что по завершении цикла выполнится условие `s == SumUp(lo, hi)`, поскольку в этот момент `i` получит значение `hi`. Но верификатор сообщает, что инвариант не выполняется телом цикла. ☹ Чтобы понять причину, напишем несколько аннотаций, действуя в обратном направлении от желаемого инварианта после выполнения цикла и используя замены при встрече с инструкциями присваивания:

```
{{ s == SumpUp(lo, i) }}
{{ s + F(i) == SumpUp(lo, i + 1) }}
s := s + F(i);
{{ s == SumpUp(lo, i + 1) }}
i := i + 1;
{{ s == SumpUp(lo, i) }}
```

Мы должны показать, что из первой аннотации следует вторая. Теперь проблема понятна: `SumUp(lo, i)` определяется в терминах `SumUp(lo + 1, i)`, а должна определяться в терминах `SumUp(lo, i + 1)`, т. е. суммирование идет не в ту сторону.

Есть три способа решить эту проблему.

Первый – использовать лемму, которая сообщает, что произойдет при увеличении верхней границы, передаваемой в `SumUp`. Если вызвать

```
AppendSumUp(lo, i + 1);
```

(см. упражнение 12.8) на входе в тело цикла (или `AppendSumUp(lo, i)` после увеличения `i`), верификация выполняется успешно. То же верно и для `LoopDown`: нам необходимо вызвать `PrependSumDown(i, hi)` в конце тела цикла (или `PrependSumDown(i - 1, hi)` перед уменьшением `i`).

Второй способ – изменить инвариант цикла: записать его в терминах `SumDown` вместо `SumUp`, поскольку определение `SumDown` прямо говорит, что происходит при увеличении верхней границы:

```
invariant s == SumDown(lo, i)
```

Как видите, LoopUp требует присутствия SumDown в своем инварианте. То есть, выполняя итерации с увеличением нижней границы, нужная нам функция рекурсивно возвращается вниз. И наоборот, чтобы доказать корректность LoopDown, нужен инвариант:

```
invariant s == SumUp(i, hi)
```

Это правило стоит того, чтобы его запомнить. Но есть еще одно решение, позволяющее использовать SumUp в LoopUp и SumDown в LoopDown.

Итак, третий способ заключается в том, чтобы заменить инвариант «что уже сделано» инвариантом *«что еще предстоит сделать»*, как я упоминал в разделе «Метод конструирования циклов 12.1». Для LoopUp инвариант принимает вид:

```
invariant s + SumUp(i, hi) == SumUp(lo, hi)
```

Инварианты «что сделано» встречаются гораздо чаще, поэтому вы могли не задумываться о такой формулировке. Но, взглянув на нее, можно заметить, что она выглядит вполне естественно. По сути, одна итерация цикла перемещает член F(i) из Sum(i, hi) в s.

Упражнение 12.9

Докажите следующую лемму:

```
lemma SumRanges(lo: int, mid: int, hi: int)
  requires lo <= mid <= hi
  ensures SumUp(lo, mid) + SumUp(mid, hi) == SumUp(lo, hi)
```

12.3.3. В заключение о программах суммирования

Для сравнения ниже приводятся различные версии метода суммирования:

```
method LoopUp0(lo: int, hi: int)
    returns (s: int)
  requires lo <= hi
  ensures s == SumUp(lo, hi)
{
  s := 0;
  var i := lo;
  while i != hi
    invariant lo <= i <= hi
    invariant s == SumUp(lo, i)
  {
    s := s + F(i);
    i := i + 1;
```

```
method LoopDown0(lo: int, hi: int)
    returns (s: int)
  requires lo <= hi
  ensures s == SumDown(lo, hi)
{
  s := 0;
  var i := hi;
  while i != lo
    invariant lo <= i <= hi
    invariant s == SumDown(i, hi)
  {
    i := i - 1;
    s := s + F(i);
```

```
    AppendSumUp(lo, i);
  }
}

method LoopUp1(lo: int, hi: int)
    returns (s: int)
  requires lo <= hi
  ensures s == SumDown(lo, hi)
{
  s := 0;
  var i := lo;
  while i != hi
    invariant lo <= i <= hi
    invariant s == SumDown(lo, i)
  {
    s := s + F(i);
    i := i + 1;
  }
}

method LoopUp2(lo: int, hi: int)
    returns (s: int)
  requires lo <= hi
  ensures s == SumUp(lo, hi)
{
  s := 0;
  var i := lo;
  while i != hi
    invariant lo <= i <= hi
    invariant s + SumUp(i, hi)
            == SumUp(lo, hi)
  {
    s := s + F(i);
    i := i + 1; s := s + F(i);
  }
}
```

```
    PrependSumDown(i, hi);
  }
}

method LoopDown1(lo: int, hi: int)
    returns (s: int)
  requires lo <= hi
  ensures s == SumUp(lo, hi)
{
  s := 0;
  var i := hi;
  while i != lo
    invariant lo <= i <= hi
    invariant s == SumUp(i, hi)
  {
    i := i - 1;
    s := s + F(i);
  }
}

method LoopDown2(lo: int, hi: int)
    returns (s: int)
  requires lo <= hi
  ensures s == SumDown(lo, hi)
{
  s := 0;
  var i := hi;
  while i != lo
    invariant lo <= i <= hi
    invariant SumDown(lo, i) + s
            == SumDown(lo, hi)
  {
    i := i - 1;
    s := s + F(i);
  }
}
```

Также в этих программах можно заметить, что циклы, в которых счетчик увеличивается, выполняют приращение счетчика цикла в конце, после того как остальная часть тела его использует. Это называется *постинкрементированием*. Циклы, в которых счетчик уменьшается, выполняют уменьшение счетчика цикла в начале, перед тем как остальная часть тела его использует. Это называется *преддекрементированием*. Использование постинкрементирования и преддекрементирования – обычная практика.

12.4. Итоги

При разработке императивных программ мы часто используем спецификации, которые задаются функциональными определениями. В таких случаях нужно убедиться, что направление рекурсии совпадает с направлением итераций. Любые несоответствия можно устранить, доказав соответствующие леммы или сформулировав инварианты цикла так, чтобы они соответствовали и рекурсивным определениям, и реализациям циклов.

Сравнивая циклы, движущиеся в разных направлениях, мы увидели, что в восходящих циклах используется постинкрементирование счетчика цикла, а в нисходящих – преддекрементирование.

Помимо написанных нами программ, я также представил два метода конструирования циклов. Знание этих методов необходимо для овладения доказательством корректности программ.

Примечания

В этой главе я показал инварианты «что уже сделано» (которые наиболее распространены) и «что еще предстоит сделать». Чтобы облегчить рассуждения о том, что еще предстоит сделать, можно использовать вызываемые аннотированными блоками инструкций. Как показывает Эрик Хенер (Eric Hehner), такие *аннотированные блоки инструкций* иногда представляют собой естественную альтернативу инструкциям **`while`** [62].

Глава 13

Массивы и поиск

Массив – это простая изменяемая структура данных. Доступ к его элементам осуществляется посредством числовых значений, называемых *индексами*, которые вычисляются во время выполнения. В этой главе мы посмотрим, как писать и верифицировать программы, использующие массивы, и откроем для себя множество полезных и интересных алгоритмов.

13.0. О массивах

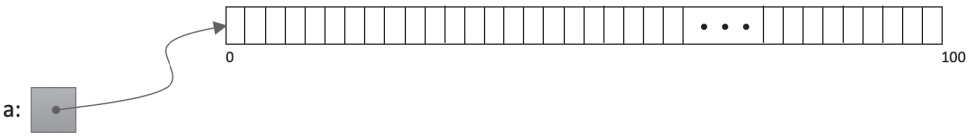
Я предполагаю, что вы уже встречались с массивами ранее. Но, чтобы убедиться, что мы говорим об одних и тех же типах массивов, рассмотрим некоторые важные детали.

13.0.0. Размещение массива в памяти и его длина

Массив хранит фиксированное количество элементов, адресуемых последовательными индексами, начиная с `0`. *Длина* массива определяется при его размещении в памяти. Следующая программная инструкция размещает в памяти массив длиной `100` и присваивает локальной переменной `a` ссылку на этот массив:

```
var a := new int[100];
```

Следующая диаграмма иллюстрирует связь переменной `a` с массивом, на который она ссылается:



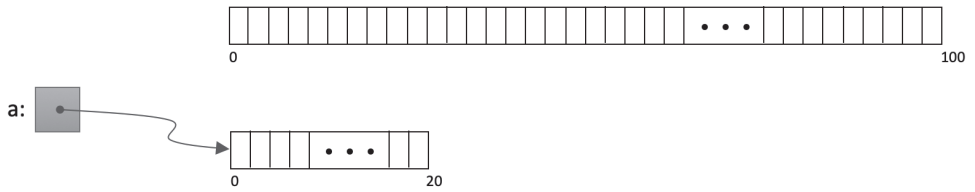
Получить длину массива `a` можно с помощью члена `Length`. Например, после предыдущего присваивания мы можем доказать:

```
assert a.Length == 100;
```

Каждый из 100 элементов имеет тип **`int`**, а сама переменная `a` имеет тип **`array<int>`**, не зависящий от длины `a`. Это означает, что мы можем обновить `a`, записав в нее ссылку на другой массив, который может иметь другую длину. Например,

```
a := new int[20];
```

разместит в памяти новый массив длиной `20` и запишет ссылку на него в переменную `a` (затерев прежнюю ссылку). Следующая диаграмма иллюстрирует новую ситуацию:



13.0.1. Элементы массива

Доступ к элементу с индексом `i` в массиве `a` осуществляется с помощью выражения `a[i]`. В отношении этого выражения мы должны доказать, что `i` является правильным индексом для `a`, т. е.:

```
0 <= i < a.Length
```

Такой способ доступа к массиву можно использовать в левой части инструкции присваивания. Например,

```
a[9] := a[9] + 5;
```

увеличит на `5` элемент с индексом `9`.

Обновление одного элемента массива не оказывает влияния на другие элементы. Например, утверждение **`assert`** в следующей программе является доказуемым:

```
a[6] := 2;
a[7] := 3;
assert a[6] == 2 && a[7] == 3;
```

Конечно, если два целочисленных выражения дают одно и то же значение, то их использование для индексирования обеспечит доступ к одному и тому же элементу массива. Например,

```
method TestArrayElements(j: nat, k: nat)
  requires j < 10 && k < 10
{
  var a := new int[10];
  a[j] := 60;
  a[k] := 65;
  if j == k {
    assert a[j] == 65;
  } else {
    assert a[j] == 60;
  }
}
```

Этот метод показывает, что вторая инструкция присваивания затрет значение, присвоенное первой инструкцией, если `j == k`, и никак не повлияет на него, если `j != k`.

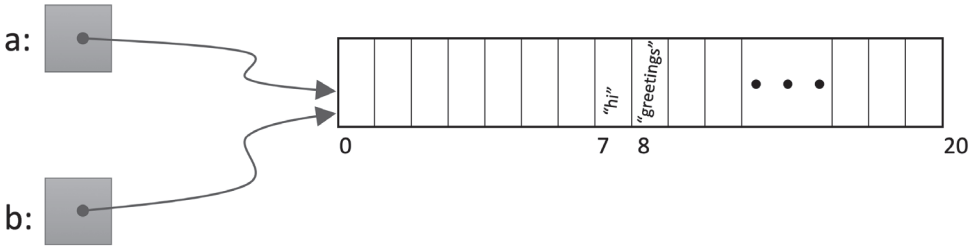
13.0.2. Массивы являются ссылками

Переменные-массивы в Dafny являются *ссылками* на свои элементы. Следовательно, копируя значение переменной-массива, вы фактически копируете ссылку на массив, а не его элементы. Эту особенность иллюстрирует следующий фрагмент программы:

```
var a := new string[20];
a[7] := "hello";
var b := a;
assert b[7] == "hello";
b[7] := "hi";
a[8] := "greetings";
assert a[7] == “hi” && b[8] == "greetings";
```

На диаграмме ниже показано состояние массива после выполнения этого фрагмента.

В этом примере в памяти хранится только один массив. Он имеет длину `20`, а его элементы – тип **`string`**. В локальную переменную `a` записана ссылка на массив, а элемент `7` получает значение «`hello`». Инструкция присваивания записывает в `b` ссылку на тот же массив в памяти, на который ссылается `a`. Соответственно, в оставшейся части примера, поскольку `a` и `b` равны (т. е. содержат ссылки на один и тот же массив), не имеет значения, какая локальная переменная используется для доступа к элементам массива.



Обратите также внимание, что `a` и `b` – это разные локальные переменные. Следовательно, если в `b` записать ссылку на другой массив, то это никак не повлияет на `a` и она по-прежнему будет ссылаться на первоначальный массив:

```
b := new string[8];
b[7] := “long time, no see”;
assert a[7] == “hi”;
assert a.Length == 20 && b.Length == 8;
```

13.0.3. Многомерные массивы

Массивы в Dafny могут иметь более одного измерения. Вот пример создания двумерного массива (обычно называемого *матрицей*):

```
var m := new bool[3, 10];
m[0, 9] := true;
m[1, 8] := false;
assert m.Length0 == 3 && m.Length1 == 10;
```

Элементы этой матрицы имеют тип **`bool`**, а переменная `m` – тип **`array2<bool>`**. Каждое измерение имеет длину, и эти длины можно получить с помощью членов `Length0` и `Length1`.

В основном я буду использовать одномерные массивы.

13.0.4. Последовательности

Близкими родственниками массивов являются *последовательности*. В отличие от массивов, которые могут изменяться и доступны через ссылки, последовательности являются неизменяемыми значениями, точно так же как логические значения, целые числа и значения типов данных **`datatype`**. По этой причине типы массивов часто называют *ссылочными типами*, а типы последовательностей *типами-значениями*.

Все элементы последовательности должны быть одного типа, как и элементы массивов. Конструктор типа последовательности записывается как **`seq`**, соответственно, **`seq<int>`** обозначает целочисленные последовательности, а **`seq<bool>`** – последовательности логических значений.

Поскольку массив является ссылкой на элементы, хранящиеся в куче (т. е. в динамической памяти), он создается с использованием оператора **new**, как мы видели выше. Самый простой способ записать значение последовательности – перечислить ее элементы в квадратных скобках. Например, `[]` обозначает пустую последовательность, `[58]` обозначает последовательность с одним целочисленным элементом `58`, а

```
["hey", "hola", "tjena"]
```

это последовательность из трех строк.

Индексация элементов последовательности осуществляется с использованием того же синтаксиса, что и элементов массивов. Например, если именем `greetings` обозначить последовательность строк, приведенную выше, то `greetings[2]` будет соответствовать элементу «`tjena`». Длина массива `a` определяется как `a.Length`, а длина последовательности `a` – как `|a|`. Например, `|[]|` даст `0`, а `|greetings|` даст `3`.

Последовательности можно объединять с помощью оператора `+`. Вот пример с некоторыми пятиугольными числами (pentagonal numbers):

```
[1, 5, 12] + [22, 35] == [1, 5, 12, 22, 35]
```

Другой способ получить последовательность – извлечь подпоследовательность из другой последовательности. Выражение `a[lo..hi]` извлечет подпоследовательность длиной `hi - lo` из последовательности `a`, взяв первые `hi` элементов из `a` и отбросив из результата первые `lo` элементов. Операция требует

```
0 <= lo <= hi <= |a|
```

Если нижняя граница равна `0`, то ее можно опустить, а если верхняя граница равна длине последовательности, ее тоже можно опустить. Вот несколько примеров, где я использую `p` для обозначения последовательности пяти пятиугольных чисел, перечисленных выше:

```
p[2..4] == [12, 22]
p[..2] == [1, 5]
p[2..] == [12, 22, 35]
greetings[..1] == ["hey"]
greetings[1..2] == ["hola"]
```

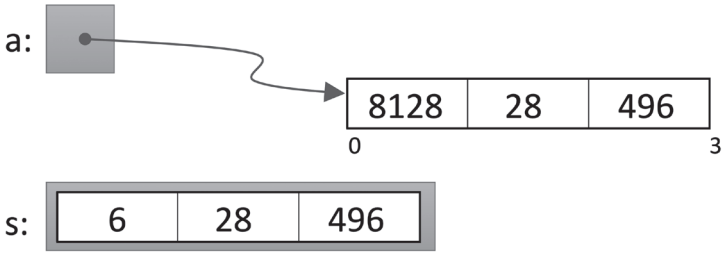
Операция, такая как `p[2..]` и применяемая к `p`, интерпретируется как «отбросить два первых элемента», а операция `p[..2]` – «взять два первых элемента.

Последовательность можно также получить из фрагмента массива. При этом используется тот же синтаксис, что и в операциях взятия и отбрасывания подпоследовательностей. Чтобы получить последовательность всех элементов массива `a`, можно опустить нижнюю и верхнюю границы, напи-

сав `a[..]`. Обратите внимание, что из текущих элементов массива создается новое значение последовательности, на которое не смогут повлиять будущие изменения в массиве. Например, все утверждения **`assert`** в следующем фрагменте кода верны:

```
var a := new int[3];
a[0], a[1], a[2] := 6, 28, 496;
assert a[..2] == [6, 28] && a[1..] == [28, 496];
var s := a[..];
assert s == [6, 28, 496];
a[0] := 8128;
assert s[0] == 6 && a[0] == 8128;
```

На следующей диаграмме показано окончательное состояние программы. Обратите внимание, что `a` содержит ссылку на массив, а `s` – значение последовательности.



Но хватит теоретических рассуждений о массивах и последовательностях. Пришло время заняться программами, которые их используют.

13.1. Линейный поиск

Рассмотрим метод поиска элемента в массиве. Чтобы сделать метод пригодным для использования в различных поисковых приложениях, параметризуем его предикатом `P` и напишем метод для поиска элемента массива, удовлетворяющего предикату `P`. Сигнатура метода будет выглядеть так:

```
method LinearSearch<T>(a: array<T>, P: T -> bool) returns (n: int)
```

Метод принимает параметр типа `T`, обозначающий тип элементов массива. По типу предиката `P` можно сказать, что он принимает аргумент типа `T` и возвращает логическое значение. У метода есть выходной параметр `n`, который мы будем использовать для возврата результатов поиска.

Раньше я не говорил, что функции могут использоваться в роли значений. Я не буду много говорить об этом и здесь, лишь отмечу, что если вы объявили функцию, то сможете использовать имя этой функции как обычное значение. Например, предположим, что вы объявили предикат

```
predicate NearPi(r: real) {
  3.14 <= r <= 3.15
}
```

и переменную `rls` типа **`array<real>`**, то в таком случае вы сможете вызвать указанный выше метод следующим образом:

```
var n := LinearSearch(rls, NearPi);
```

Обратите внимание: поскольку параметр `P` не является призрачным, значение, передаваемое в вызов `LinearSearch`, тоже не должно быть призрачным, поэтому `NearPi` объявляется как компилируемый предикат.

Реализации метода `LinearSearch`, которые мы рассмотрим далее, выполняют последовательный обход элементов массива. Такой алгоритм называется линейным поиском.

Я покажу несколько вариантов спецификации и соответствующие реализации, что позволит мне представить обобщенную конструкцию спецификаций для программ, работающих с массивами: логические кванторы.

13.1.0. Простая спецификация

Для разминки напишем простую спецификацию, возвращающую значение `n` в закрытом диапазоне `0 <= n <= a.Length`, которое строго меньше `a.Length` и представляет индекс элемента массива, удовлетворяющего предикату `P`:

```
method LinearSearch0<T>(a: array<T>, P: T -> bool) returns (n: int)
  ensures 0 <= n <= a.Length
  ensures n == a.Length || P(a[n])
```

В нашей реализации мы будем возвращать в выходном параметре `n` индекс цикла. Вот спецификация цикла:

```
{
  n := 0;
  while n != a.Length
    invariant 0 <= n <= a.Length
}
```

Этой спецификации цикла достаточно, чтобы доказать соответствие постусловию метода, поскольку оно утверждает, что по завершении цикла `n == a.Length`, и это значение допускается спецификацией метода.

Тело цикла будет увеличивать `n`, предварительно проверяя соответствие текущего элемента предикату `P`. Предположим, что проверка вернула **`true`**. Что мы должны сделать в этом случае? А в этом случае наша цель достигнута, и мы можем не беспокоиться об инвариантах цикла и других обязательствах. Проще говоря, мы можем прервать цикл и выйти из метода. Сделать это можно с помощью инструкции **`return`**, которая тут же завершает

выполнение тела метода. В точке вызова **return** мы должны доказать, что постусловие метода выполняется. С этой целью можно представить инструкцию **return** как переход к самому концу тела метода, где мы должны доказать, что постусловие метода выполняется.

Итак, вот полное тело метода `LinearSearch0`:

```
n := 0;
while n != a.Length
  invariant 0 <= n <= a.Length
{
  if P(a[n]) {
    return;
  }
  n := n + 1;
}
```

Вместо инструкции **return** для перехода в конец тела метода можно использовать инструкцию **break**, которая прерывает цикл и передает управление первой инструкции, следующей за циклом. Этот способ прерывания циклов часто встречается во многих программах. При этом вам не нужно доказывать, что инвариант цикла выполняется в момент вызова **break**, и разрешено прерывать выполнение цикла независимо от условия цикла. Я уже говорил, что инвариант и отрицание условия цикла будут выполняться сразу после цикла, но это утверждение не относится к случаю, когда работа цикла прерывается инструкцией **break**. Для каждой инструкции **break**, используемой в теле цикла, существует дополнительный набор путей через тело цикла к выходу из него. За исключением этих различий, **break** не создает никаких проблем, и верификатор имеет всю необходимую информацию о дополнительных путях выхода из цикла. И все же, чтобы сохранить единообразие, я буду избегать инструкции **break** в этой книге.

13.1.1. Необходимость оправдания неудачи

Спецификация метода `LinearSearch0` допускает реализацию, намного более простую, чем использование цикла: просто замените все тело метода единственной инструкцией `n := a.Length`. (Если бы существовал смайлик, изображающий тромбонные фанфары Винни Пуха, я бы вставил его сюда.) Не хотелось бы, чтобы наша спецификация позволяла реализации так легко сдаваться. Возвращая `a.Length`, метод сообщает, что в `a` нет элемента, удовлетворяющего предикату `P`. Для этого требуется логический квантор. Я буду использовать *квантор всеобщности*, который в Dafny записывается как **forall** и как `∀` – в логике. Он вводит связанную переменную и говорит, что условие выполняется для всех значений этой связанной переменной.

Вот пример квантора, говорящего обо всех целых числах `x`, где `Fib` – функция Фибоначчи (раздел 3.1):

```
forall x: int :: 5 <= x ==> 5 <= Fib(x)
```

Это выражение возвращает значение **true**, если *область действия* квантора – т. е. выражение справа от «::» – выполняется для каждого целого числа x, и значение **false** в противном случае.

Упражнение 13.0

Какое значение (**false** или **true**) имеет квантор в примере выше?

Что касается локальных переменных, то мы можем оставить тип «: **int**», если тип можно определить по остальной части выражения.

Вот улучшенная спецификация нашего метода линейного поиска:

```
method LinearSearch1<T>(a: array<T>, P: T -> bool) returns (n: int)
  ensures 0 <= n <= a.Length
  ensures n == a.Length || P(a[n])
  ensures n == a.Length ==>
    forall i :: 0 <= i < a.Length ==> !P(a[i])
```

Она говорит, что если в n возвращается a.Length, то для каждого i в диапазоне 0 <= i < a.Length выполняется условие !P(a[i]). Выглядит неплохо, потому что предотвращает реализацию, возвращающую a.Length, когда в массиве есть элемент, удовлетворяющий предикату P. А как будет выглядеть инвариант цикла после усовершенствования спецификации метода?

Давайте применим метод конструирования циклов 12.0 – *замену константы переменной*. По завершении цикла должно выполняться условие:

```
forall i :: 0 <= i < a.Length ==> !P(a[i])
```

У нас уже есть переменная n, которая будет равна константе a.Length после цикла, поэтому, заменив эту константу переменной n, мы приходим к следующему инварианту цикла:

```
invariant forall i :: 0 <= i < n ==> !P(a[i])
```

Реализация цикла будет точно такой же, как в LinearSearch0, но давайте подробно рассмотрим, почему она работает. Написав аннотации вокруг инструкции приращения n в теле цикла, мы получаем:

```
{{ forall i :: 0 <= i < n + 1 ==> !P(a[i]) }}
n := n + 1
{{ forall i :: 0 <= i < n ==> !P(a[i]) }}
```

Далее воспользуемся тремя преобразованиями равенства, которые часто встречаются при работе с кванторами. Поскольку n имеет тип **nat**, то существует арифметическое равенство:

```
  0 <= i < n + 1
=
  0 <= i < n || i == n
```

Второе равенство – это правило *разделения области значений квантифицированной переменной* (Range Split; см. упражнение B.7), которое в общем случае имеет вид:

```
  forall x :: A || B ==> C
=
  (forall x :: A ==> C) && (forall x :: B ==> C)
```

Мы применяем правило разделения области значений квантифицированной переменной с заменой `x`, `A`, `B` и `C` на `i`, `0 <= i < n`, `i == n` и `!P(a[i])` соответственно, что дает нам:

```
  forall i :: 0 <= i < n || i == n ==> !P(a[i])
=
  (forall i :: 0 <= i < n ==> !P(a[i])) &&
  (forall i :: i == n ==> !P(a[i]))
```

Третье равенство известно как *правило элиминации квантора всеобщности по одноэлементному множеству* (One-Point Rule; см. упражнение B.7) и дает нам возможность исключить квантор. Его общая форма:

```
  forall x :: x == E ==> A
=
  A[[x := E]]
```

где `E` – выражение, в котором не упоминается `x`. В последней строке используется подстановка (впервые представленная в разделе 2.3; дополнительно см. также упражнение B.6), которая заменяет на `E` все вхождения `x` в `A`. В нашей ситуации мы применяем правило элиминации квантора всеобщности по одноэлементному множеству с заменой `x`, `E` и `A` на `i`, `n` и `!P(a[i])` соответственно:

```
  forall i :: i == n ==> !P(a[i])
=
  !P(a[n])
```

Вернемся к аннотациям вокруг `n := n + 1`. Три приведенных выше равенства позволяют записать условие перед увеличением `n` следующим образом:

```
{{ (forall i :: 0 <= i < n ==> !P(a[i])) && !P(a[n]) }}
{{ forall i :: 0 <= i < n + 1 ==> !P(a[i]) }}
n := n + 1
{{ forall i :: 0 <= i < n ==> !P(a[i]) }}
```

Первый из конъюнктов выполняется при входе в тело цикла согласно инварианту, а второй выполняется после инструкции **`if`** (потому что ветвь `then` использует **`return`** для перехода к концу метода, если выполняется `P(a[n])`).

Это доказывает корректность `LinearSearch1`.

13.1.2. Поиск первого вхождения

Мы можем еще больше уточнить спецификацию нашего метода для случая, когда элемент найден, и гарантировать возврат первого вхождения элемента, удовлетворяющего `P`. С этой целью будем говорить, что среди первых `n` элементов массива нет элементов, удовлетворяющих предикату `P`. Постусловие, которое мы добавили в `LinearSearch1`, было его слабейшей формой, поэтому заменим эту инструкцию **`ensures`** новой:

```
method LinearSearch2<T>(a: array<T>, P: T -> bool) returns (n: int)
  ensures 0 <= n <= a.Length
  ensures n == a.Length || P(a[n])
  ensures forall i :: 0 <= i < n ==> !P(a[i])
```

Спецификация цикла и тело `LinearSearch1` можно оставить прежними в новом методе.

13.1.3. Зная, что искомый элемент существует

Давайте реализуем еще один вариант линейного поиска, в котором заранее известно, что искомый элемент присутствует в массиве. В этом случае мы можем упростить постусловие, удалив особый случай `n == a.Length`. Чтобы задать это новое знание, используем предусловие. Самый простой способ записи условия – использовать *квантор существования*, который в Dafny записывается как **`exists`** и `∃` в логике.

Вот пример применения этого квантора к целым числам:

```
exists x :: 0 <= x && Fib(x) == 143
```

Это выражение возвращает **`true`**, если существует целое число, для которого выполняется заданное условие, и **`false`** в противном случае.

Упражнение 13.1

Какое значение (**`false`** или **`true`**) имеет квантор в примере выше?

С квантором существования в роли предусловия наша новая спецификация линейного поиска выглядит следующим образом:

```
method LinearSearch3<T>(a: array<T>, P: T -> bool) returns (n: int)
  requires exists i :: 0 <= i < a.Length && P(a[i])
  ensures 0 <= n < a.Length && P(a[n])
```

Предусловие говорит, что существует индекс `i`, для которого выполняется `P(a[i])`, а постусловие говорит, что `n` – это индекс, для которого выполняется `P(a[n])`.

Важное отличие в формулировании кванторов всеобщности и существования заключается в основной связке области действия: в кванторе всеобщности используется импликация, а в кванторе существования –

конъюнкция. Например, чтобы сказать, что `P(a[i])` выполняется для *всех* индексов, мы пишем:

```
forall i :: 0 <= i < a.Length ==> P(a[i])
```

а чтобы сказать, что условие выполняется только для *некоторого* индекса, мы пишем:

```
exists i :: 0 <= i < a.Length && P(a[i])
```

Если заменить `==>` на `&&`, то выражение **`forall`** говорило бы, что все целые числа находятся в диапазоне от `0` до `a.Length`, что неверно (например, `-1` не входит в этот диапазон), поэтому выражение **`forall`** вернет значение **`false`**. И наоборот, если заменить `&&` на `==>`, то выражение **`exists`** будет утверждать, что существует целое число `i` такое, что *если* оно находится в диапазоне от `0` до `a.Length`, *то* и т. д. Это верно для любых целых чисел, не входящих в указанный диапазон, которых очень много (например, `-1` – одно из них), поэтому выражение **`exists`** вернет значение **`true`**.

Вас может удивить, что в `LinearSearch3` можно использовать ту же реализацию метода и инвариант цикла, что и в `LinearSearch1` и `LinearSearch2`. Почему? Выполнение постусловия в инструкции **`return`** не выглядит удивительным. Но то, что происходит после цикла, где отрицание условия цикла сообщает нам, что `n` имеет значение, равное `a.Length`, которое новая спецификация не позволяет возвращать, действительно выглядит удивительным. Мы также знаем, что после цикла выполняется инвариант, поэтому можем доказать выполнение следующего условия:

```
forall i :: 0 <= i < a.Length ==> !P(a[i])
```

Вот еще одно полезное свойство кванторов, известное как *закон Де Моргана* (см. упражнение B.8):

```
(forall x :: !R) == !(exists x :: R)
Из этого и других свойств булевой логики получаем:
  forall x :: A ==> B
=
  forall x :: !A || B
=      { закон Де Моргана }
  forall x :: !(A && !B)
=      { закон Де Моргана для кванторов }
  !exists x :: A && !B
Используя x, A и B как i, 0 <= i < a.Length и P(a[i]), получаем
  forall i :: 0 <= i < a.Length ==> !P(a[i])
=
  !exists i :: 0 <= i < a.Length && P(a[i])
```

где первая строка – это то, что мы получаем из инварианта и отрицания условия цикла, а второе выражение – это отрицание предусловия. Другими

словами, после цикла мы можем получить условие, которое является отрицанием предусловия. Это может означать только одно – поток выполнения никогда не достигнет этой точки!

13.1.4. Инвариант, указывающий, где искать

Существует другой интересный способ записи инварианта цикла для `LinearSearch3`. В начале каждой итерации мы можем утверждать, что искомый элемент имеет некоторый индекс `n` или выше:

```
exists i :: n <= i < a.Length && P(a[i])
```

Действуя в обратном направлении от этого инварианта, мы получаем следующие аннотации вокруг инструкции увеличения `n`:

```
{{ !P(a[n]) && exists i :: n <= i < a.Length && P(a[i]) }}
// разделение диапазона и правило одной точки
{{ !P(a[n]) && (P(a[n]) || exists i :: n + 1 <= i < a.Length && P(a[i])) }}
// логика
{{ exists i :: n + 1 <= i < a.Length && P(a[i]) }}
n := n + 1
{{ exists i :: n <= i < a.Length && P(a[i]) }}
```

Первый конъюнкт выполняется потому, что ветвь **`then`** выполняет **`return`**, а второй конъюнкт выполняется в силу инварианта при входе в цикл. Итак, цикл выше соответствует этому инварианту.

Мы можем внести еще два упрощения. Во-первых, квантор существования подразумевает `n < a.Length`, поскольку он гласит, что `i` находится в заданном диапазоне. Это означает, что мы можем усилить инвариант, в котором упоминается диапазон `n`:

```
0 <= n < a.Length
```

Условие цикла всегда оценивается в состоянии, в котором выполняется инвариант цикла. В этом диапазоне `n` условие `n != a.Length` можно упростить до **`true`**. Однако если мы сделаем это, то используемая в Dafny простая схема автоматического определения инструкции **`decreases`** по умолчанию в спецификации цикла (см. раздел 11.2.1) ни к чему не приведет, поэтому нам придется записать ее вручную.

Вот наша окончательная реализация `LinearSearch3`:

```
n := 0;
while true
  invariant 0 <= n < a.Length
  invariant exists i :: n <= i < a.Length && P(a[i])
  decreases a.Length - n
{
  if P(a[n]) {
```

```
      return;
    }
    n := n + 1;
  }
```

В этом инварианте цикла представлена разновидность замены константы переменной, но замена производится в предположении, а не в цели доказательства.

Метод конструирования циклов 13.0
(Замена константы переменной в предусловии)

Чтобы сконструировать постусловие из предусловия формы `...C...` для некоторой константы `C`, используйте инвариант цикла `...k...`, где `k` – локальная переменная, изменяемая циклом, чтобы постепенно сделать предположение более конкретным. Этот метод можно назвать *заменой константы переменной в предусловии*.

13.1.5. В заключение о свойствах кванторов

В этом разделе, посвященном линейному поиску, мы столкнулись с несколькими полезными свойствами кванторов. Вот краткое описание этих свойств в различных формах.

Разделение области значений квантифицированной переменной:

- `(forall x :: A || B ==> C) <==>`
  `(forall x :: A ==> C) && (forall x :: B ==> C);`
- `(exists x :: (A || B) && C) <==>`
  `(exists x :: A && C) || (exists x :: B && C).`

Правило элиминации квантора всеобщности по одноэлементному множеству:

- `(forall x :: x == E ==> A) <==> A[x := E];`
- `(exists x :: x == E && A) <==> A[x := E].`

Закон Де Моргана для кванторов:

- `(forall x :: A ==> B) <==> !(exists x :: A && !B);`
- `!(forall x :: A ==> !B) <==> (exists x :: A && B).`

Дополнительные свойства кванторов вы найдете в приложении B.

Упражнение 13.2

Напишите спецификацию для метода линейного поиска, который всегда возвращает значение, строго меньшее `a.Length`, и использует

отрицательное значение (вместо `a.Length`) как признак, что ни один элемент не удовлетворяет `P`.

13.2. Двоичный поиск

В этом разделе мы реализуем еще один алгоритм из кладовой информатики – двоичный поиск. Идея заключается в следующем: если входные данные отсортированы, то в каждой итерации цикла можно уменьшить пространство поиска вдвое. В этом примере мы будем использовать целочисленный массив и искать заданный ключ, а также попрактикуемся в использовании метода конструирования циклов 12.0, заключающегося в *замене константы переменной*.

13.2.0. Требование сортировки в спецификации

Для начала посмотрим, как сообщить, что входные данные отсортированы. Вот ответ:

```
forall i, j :: 0 <= i < j < a.Length ==> a[i] <= a[j]
```

В нем говорится, что если взять два индекса в массиве, `i` и `j`, где `i` находится «левее» `j`, то элементы, соответствующие этим двум индексам, будут упорядочены правильно. (Также было бы неплохо написать `i <= j` вместо `i < j`.)

Написанное мною не единственный способ выразить свойство упорядоченности элементов массива. Как вариант, можно также использовать вложенные кванторы по `i` и `j`. Например,

```
forall i :: 0 <= i ==>
  forall j :: i < j < a.Length ==> a[i] <= a[j]
```

Это вполне рабочий способ, но с невложенными кванторами обычно легче работать.

Другой способ сообщить об упорядоченности массива – показать, что любые два соседних элемента упорядочены правильно:

```
forall i :: 0 <= i < a.Length - 1 ==> a[i] <= a[i + 1]
```

Это выражение логически эквивалентно приведенным выше, но оно не так очевидно дает понять, что несмежные элементы тоже упорядочены правильно. Это свойство нам понадобится в двоичном поиске. Чтобы получить «транзитивное» свойство упорядоченности несмежных элементов, потребуются дополнительные усилия для проведения индуктивного доказательства. Поэтому старайтесь не выражать свойство упорядоченности на основе соседних элементов.

Упражнение 13.3

Напишите выражение с квантификаторами, сообщающее, что элементы в заданном целочисленном массиве строго увеличиваются.

Упражнение 13.4

Докажите следующую лемму, которая показывает, что транзитивное свойство упорядоченности можно получить из свойства упорядоченности соседних элементов.

```
lemma SortedTransitive(a: array<int>, i: int, j: int)
  requires forall k :: 0 <= k < a.Length - 1 ==> a[k] <= a[k+1]
  requires 0 <= i <= j < a.Length
  ensures a[i] <= a[j]
```

13.2.1. Постусловие двоичного поиска

Постусловие для двоичного поиска можно было бы написать по образу и подобию постусловия для линейного поиска, взяв за основу, например, постусловие `LinearSearch1` из раздела 13.1.1. Однако метод двоичного поиска легко может предоставить дополнительную информацию вызывающему коду. Мы можем вернуть «самую раннюю точку вставки», т. е. индекс самого левого вхождения ключа, если он существует в массиве, или индекс, куда вставить ключ, если его нет в массиве. Иными словами, можно вернуть длину самого длинного префикса, все элементы которого меньше искомого ключа, т. е. число `n`, сообщающее, сколько первых элементов массива строго меньше ключа, а остальные элементы – не меньше значения ключа.

Итак, вот спецификация метода двоичного поиска, который нам предстоит написать:

```
method BinarySearch(a: array<int>, key: int) returns (n: int)
  requires forall i, j :: 0 <= i < j < a.Length ==> a[i] <= a[j]
  ensures 0 <= n <= a.Length
  ensures forall i :: 0 <= i < n ==> a[i] < key
  ensures forall i :: n <= i < a.Length ==> key <= a[i]
```

Прежде чем перейти к реализации этого метода, мы можем опробовать спецификацию «самой ранней точки вставки», написав клиентский метод. Это даст нам некоторую уверенность в том, что написанная нами спецификация полностью соответствует нашим ожиданиям. Вот метод, который определяет, действительно ли данный ключ присутствует в отсортированном массиве:

```
method Contains(a: array<int>, key: int) returns (present: bool)
  requires forall i, j :: 0 <= i < j < a.Length ==> a[i] <= a[j]
  ensures present == exists i :: 0 <= i < a.Length && key == a[i]
{
```

```
  var n := BinarySearch(a, key);
  present := n < a.Length && a[n] == key;
}
```

13.2.2. Реализация

Главный инвариант двоичного поиска заключается в наличии подраздела массива, «окна» от `lo` до `hi`, где мы продолжаем поиск ключа. Все элементы, находящиеся слева от окна, слишком малы, чтобы быть искомым ключом; я назову этот подраздел массива «сегментом слишком малых значений». Все элементы, находящиеся справа от окна, напротив, слишком велики; я назову этот подраздел массива «сегментом слишком больших значений». Мы будем продолжать сжимать окно до тех пор, пока оно не достигнет размера `0`, после чего сегменты слишком малых и слишком больших значений встретятся. Уменьшая размер окна вдвое в каждой итерации, мы получаем логарифмическое количество итераций.

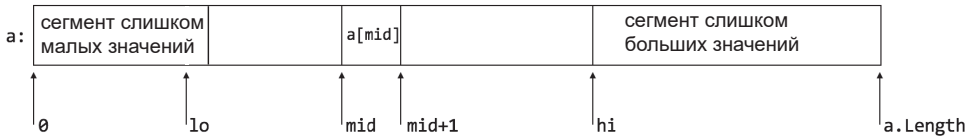
Вот наша спецификация цикла вместе с начальным и конечным присваиваниями. Обратите внимание, что `lo` и `hi` заменяют `n` в постусловии; т. е. мы дважды применяем метод конструирования циклов 12.0, выражающийся в *замене константы переменной*. После завершения цикла `lo` и `hi` равны, поэтому остальная часть инварианта и присваивание переменной `n` гарантируют соответствие постусловию метода.

```
{
  var lo, hi := 0, a.Length;
  while lo < hi
    invariant 0 <= lo <= hi <= a.Length
    invariant forall i :: 0 <= i < lo ==> a[i] < key
    invariant forall i :: hi <= i < a.Length ==> key <= a[i]
  n := lo;
}
```

В реализации цикла мы выбираем среднюю точку между `lo` и `hi` и сравниваем элемент в этой средней точке с ключом:

```
    var mid := (lo + hi) / 2;
    if a[mid] < key {
      ?
    } else {
      ?
    }
```

Мы можем использовать аннотации, как мы это делали раньше, чтобы получить точное представление о том, как должны выглядеть инструкции присваивания в позициях двух вопросительных знаков. Но давайте посмотрим, сможем ли мы вывести их из нашего понимания инвариантов. Вот диаграмма, делающая задачу более наглядной:



В ветви **then** инструкции **if** мы знаем, что рассматриваемый элемент (`a[mid]`) меньше ключа. Поскольку массив отсортирован, все элементы слева от `a[mid]` тоже меньше ключа. Это говорит о том, что нам следует увеличить сегмент слишком малых значений, включив в него элемент `a[mid]`. Это делается присваиванием:

```
lo := mid + 1;
```

В ветви **else** инструкции **if** мы знаем, что рассматриваемый элемент (`a[mid]`) не меньше ключа и, соответственно, все элементы справа от него. Это говорит о том, что нам следует увеличить сегмент слишком больших значений, включив в него элемент `a[mid]`. Это делается присваиванием:

```
hi := mid;
```

Обратите внимание на асимметрию этих двух инструкций присваивания. Она обусловлена тем, что `lo` – граница сегмента слишком малых значений; т. е. при условии `i < lo` индекс `i` находится в сегменте слишком малых значений.

Мы используем инструкцию **decreases** по умолчанию, `hi - lo`, определяющую размер окна. (Напомню, что, если вы захотите узнать, как выглядит инструкция **decreases** по умолчанию для конкретного цикла, наведите указатель мыши на ключевое слово **while** в Dafny IDE, и всплывающая подсказка даст вам эту информацию.) Тело цикла перемещает `a[mid]` из окна либо в сегмент слишком малых значений, либо в сегмент слишком больших значений, тем самым уменьшая размер окна (т. е. `hi - lo`).

Стоп, стоп, стоп! Не так быстро! Действительно ли `a[mid]` находится в окне? То есть действительно ли `lo <= mid < hi`? Диаграмма выше создает такое впечатление, но на ней сложно отразить каждый аспект задачи. Мы вычисляем среднее целое между `lo` и `hi`. Для любых целых чисел `lo` и `hi`, таких что `lo < hi`, следующее доказательное вычисление подтверждает, что `lo <= mid < hi`:

```
calc {
  lo;
==
  (lo + lo) / 2;
<=  { assert lo <= hi; }
  (lo + hi) / 2; // это mid
<   { assert lo < hi; }
```

```
  (hi + hi) / 2;
==
  hi;
}
```

Теперь мы точно знаем, что все в порядке!

Упражнение 13.5

Измените присваивание `lo := mid + 1;` на `lo := mid;`. Какое сообщение выведет верификатор? Создайте входные данные для `BinarySearch`, которые (если вы проигнорируете ошибку верификации, скомпилируете и запустите программу) вызовут ошибку, о которой сообщает верификатор.

Упражнение 13.6

Напишите спецификацию и реализацию двоичного поиска, возвращающую «последнюю точку вставки», т. е. позицию в массиве *после* последнего вхождения ключа.

Упражнение 13.7

Напишите спецификацию и реализацию двоичного поиска, возвращающую результат сразу после обнаружения любого вхождения ключа. Верните `-1`, если ключа нет в массиве.

Упражнение 13.8

Аналогично тому, как мы поступали в разделах 13.1.3 и 13.1.4, добавьте в `BinarySearch` предусловие, указывающее, что искомый ключ находится в заданном массиве. Затем упростите остальную часть спецификации и тело метода. Эта версия метода может вернуть индекс любого элемента массива, содержащего ключ.

13.3. Минимум

Теперь разработаем метод, отыскивающий минимальное значение в массиве. Мы будем двигаться постепенно, чтобы проиллюстрировать некоторые преимущества спецификации и верификации во время проектирования и разработки программ. Попутно мы познакомимся еще с двумя методами конструирования циклов.

13.3.0. Наименьшее значение – единственное

Наш метод принимает целочисленный массив `a` и возвращает целое число `m`, соответствующее самому маленькому значению из всех элементов в `a`. Это постусловие легко выразить с помощью квантора всеобщности:

```
method Min(a: array<int>) returns (m: int)
  ensures forall i :: 0 <= i < a.Length ==> m <= a[i]
```

Замена константы `a.Length` индексом цикла `n` (метод конструирования циклов 12.0) дает нам спецификацию цикла:

```
{
  var n := 0;
  while n != a.Length
    invariant 0 <= n <= a.Length
    invariant forall i :: 0 <= i < n ==> m <= a[i]
}
```

Тело цикла будет увеличивать `n`. Но прежде, если обнаружится, что `a[n]` меньше `m`, мы обновим `m`:

```
  {
    if a[n] < m {
      m := a[n];
    }
    n := n + 1;
}
```

Мы довольно быстро справились с заданием, верно?

13.3.1. Наименьшее значение – не единственное

Спецификация, которую мы написали, – это только часть истории. Уже в разделе 1.4.1 мы утверждали, что метод `Min` должен возвращать одно из входных данных. Теперь, когда у нас есть целый массив целых чисел, воспользуемся квантором существования, чтобы выразить это дополнительное постусловие:

```
ensures exists i :: 0 <= i < a.Length && m == a[i]
```

Чтобы добавить эту цель метода в спецификацию цикла, можно применить метод замены константы переменной (`a.Length` на `n`), но в данном случае можем поступить проще.

> **Метод конструирования циклов 13.1** (Всегда)
>
> Чтобы обеспечить выполнение постусловия `Q`, сделайте `Q` инвариантом цикла.

Очевидно, что если бы это было единственное постусловие, то было бы вообще бессмысленно повторять цикл. Этот метод полезен, когда имеются также другие постусловия, как в случае с нашим методом `Min`.

Используем новое постусловие как инвариант:

```
invariant exists i :: 0 <= i < a.Length && m == a[i]
```

Он гласит, что m всегда является одним из элементов a. Это условие поддерживается телом цикла, но верификатор жалуется, что оно не выполняется на входе в цикл. Если вы не заметили этого раньше, то теперь наверняка заметите, что наша программа не инициализирует m. В разделе 13.3.0 начальное значение m не играло никакой роли.

Упражнение 13.9

Объясните, почему начальное значение m не играло никакой роли в разделе 13.3.0. На входе в метод выходной параметр мог иметь произвольное значение. Опишите порядок выполнения метода, получающего трехэлементный массив `2, 10, 1`, если m получает первоначальное значение `-62081`.

Итак, нам нужно инициализировать m и убедиться, что это – элемент a. Как ребенка в кондитерской или гитариста в музыкальном магазине, вас может одолевать множество заманчивых идей. Например, можно выбрать первый элемент массива:

```
m := a[0];
```

или последний:

```
m := a[a.Length - 1];
```

или отстоящий на две трети от начала a:

```
m := a[2 * a.Length / 3];
```

Подойдет любой из этих вариантов, лишь бы массив содержал хоть какие-то элементы! Но это предусловие мы не предусмотрели. Давайте добавим его:

```
requires a.Length != 0
```

а затем вы сможете выбрать любой из вариантов инициализации m, показанных выше.

Упражнение 13.10

Мы можем инициализировать m элементом, отстоящим на две трети от начала a. А можно ли инициализировать элементом, который следует за тем, что отстоит на две трети от начала?

```
m := a[2 * a.Length / 3 + 1];
```

Если вы решите использовать традиционный выбор `m := a[0]`, то нет смысла повторять цикл для `n == 0`. Эту итерацию можно пропустить, инициализировав n значением 1.

Упражнение 13.11

Если вы начнете с `1`, как только что было предложено, то какое из трех вхождений числа `0` в инвариантах цикла можно заменить числом `1`? Объясните почему.

Упражнение 13.12

Напишите спецификацию и реализацию метода поиска максимального значения в массиве типа **`array<nat>`**. Допустите возможность получения пустого массива и в этом случае верните `0` как максимальное значение.

13.3.2. Краткий обзор пройденного пути

Мы начали этот пример, написав одно постусловие и код. Затем мы добавили второе постусловие и скорректировали инвариант цикла. Чтобы обеспечить соответствие инварианту на входе в цикл, мы обнаружили, что нам нужно добавить предусловие метода.

Это типичный процесс разработки кода. На первом этапе мы можем не знать, как должна выглядеть спецификация. В процессе дела мы выясняем, какие пред- и постусловия нужно изменить или добавить. Любое изменение в спецификации влияет на клиентов. Это веская причина писать явные спецификации, а также пользоваться автоматической верификацией клиентов. ☺

13.4. Количество совпадений

В спецификации и доказательстве следующей программы используются мультимножества. Как мы видели ранее (в разделе 10.2.4, в двух абзацах, где кратко упоминается термин *экстенсиональность*), доказательства с применением мультимножеств требуют ручного вмешательства, т. е. часто приходится вручную отлаживать доказательства, предоставляя верификатору все больше и больше деталей, пока он не сможет подтвердить верность доказательства.

Далее мы рассмотрим довольно интересную программу: она получает два отсортированных массива и определяет количество общих элементов. Учитывая, что элементы в массивах могут повторяться, мои слова выглядят немного двусмысленно, поэтому уточним условие. Сколько элементов одного массива можно сопоставить с равными элементами в другом массиве? Другими словами, если собрать элементы каждого массива в мультимножество, то каков будет размер пересечения этих двух мультимножеств? Вот спецификация этого метода на Dafny:

```
method CoincidenceCount(a: array<int>, b: array<int>) returns (c: nat)
  requires forall i, j :: 0 <= i < j < a.Length ==> a[i] <= a[j]
```

```
requires forall i, j :: 0 <= i < j < b.Length ==> b[i] <= b[j]
ensures c == |multiset(a[..]) * multiset(b[..])|
```

Два предусловия говорят, что `a` и `b` отсортированы (см. раздел 13.2.0). В постусловии используются новые для нас обозначения, поэтому рассмотрим их поближе.

13.4.0. Обозначения

Нам нужно представить два массива в виде мультимножеств. В Dafny нет встроенной функции преобразования массива в мультимножество, поэтому мы сначала превращаем массив в последовательность (см. раздел 13.0.4), а затем последовательность – в мультимножество.

Функция **`multiset`** превращает последовательность в мультимножество, оператор `*` обозначает пересечение мультимножеств, а вертикальные скобки определяют размер (т. е. мощность) мультимножества. Обратите внимание на круглые скобки после функции **`multiset`**. Если поставить фигурные скобки после ключевого слова **`multiset`**, то будет создано мультимножество из некоторых заданных элементов. Например, **`multiset`**{} – это пустое мультимножество, а **`multiset`**{x} – это мультимножество с единственным элементом `x`.

Теперь вы без труда прочитаете постусловие метода `CoincidenceCount`. Оно гласит, что `c` – это количество элементов в пересечении мультимножеств `a` и `b`.

13.4.1. Спецификация цикла

В реализации `CoincidenceCount` мы используем две переменные индекса цикла, `m` и `n`, чтобы отслеживать количество обработанных элементов в `a` и `b`. Соответствующий инвариант цикла выглядит так:

```
invariant 0 <= m <= a.Length && 0 <= n <= b.Length
```

Идея состоит в том, что цикл будет наращивать `m` или `n`, или оба индекса, в зависимости от результата сравнения следующих элементов в `a` и `b`. Проще всего будет использовать инвариант «что еще предстоит сделать» (но я советую также для сравнения попробовать инвариант «что уже сделано»):

```
invariant c + |multiset(a[m..]) * multiset(b[n..])|
      == |multiset(a[..]) * multiset(b[..])|
```

Условие завершимости будет вытекать из приращения индексов `m` и `n`, но, поскольку тело цикла может увеличивать только один из них, мы должны включить в инструкцию **`decreases`** не только `a.Length - m`, но и `b.Length - n`. Я объединил эти выражения посредством `+` (но точно так же их можно объединить в лексикографическую пару):

```
decreases a.Length - m + b.Length - n
```

Наконец, нам нужно условие цикла. До каких пор должны выполняться итерации? Пока выполняются *оба* условия `m < a.Length` и `n < b.Length`, или пока выполняется *хотя бы одно* из них? Обработав все элементы в одном массиве, мы найдем все возможные совпадения, поэтому на этом можно и остановиться.

В результате мы получаем следующую спецификацию цикла:

```
{
  c := 0;
  var m, n := 0, 0;
  while m < a.Length && n < b.Length
    invariant 0 <= m <= a.Length && 0 <= n <= b.Length
    invariant c + |multiset(a[m..]) * multiset(b[n..])|
          == |multiset(a[..]) * multiset(b[..])|
    decreases a.Length - m + b.Length - n
}
```

У верификатора получилось доказать постусловие метода по этой спецификации цикла, поэтому нам осталось только реализовать цикл.

13.4.2. Тело цикла

Предполагая, что присваивания `m := m + 1`, `n := n + 1` и `c := c + 1` будут играть важную роль в теле цикла, мы можем написать некоторые аннотации, как делали это в некоторых предыдущих примерах, а затем понять, как получить правильное решение. Но мы можем также подумать о том, как следует использовать эти присваивания, а затем с помощью верификатора проверить свою правоту.

Итак, тело цикла будет использовать следующие элементы `a` и `b`. Их можно сравнивать тремя способами, поэтому напишем условный оператор с тремя случаями:

```
  if
  case a[m] == b[n] =>
    ?
  case a[m] < b[n] =>
    ?
  case b[n] < a[m] =>
    ?
```

Если `a[m]` и `b[n]` равны, то это означает, что мы нашли еще одно совпадение, поэтому следует увеличить `c`, а затем перейти к следующим элементам, увеличив `m` и `n`:

```
    c, m, n := c + 1, m + 1, n + 1;
```

Если элемент a[m] меньше элемента b[n], то он меньше и всех остальных элементов в b (поскольку b отсортирован). Отсюда следует, что a[m] не совпадет ни с каким другим элементом в b, поэтому можно пропустить его, как показано ниже:

```
m := m + 1;
```

аналогично, когда b[n] меньше a[m]:

```
n := n + 1;
```

Мы сконструировали программу, полагаясь на свою интуицию и неформальные рассуждения. Это правда. Но обратите внимание, как здорово нам помог инвариант цикла. Мы начали с разработки правдоподобной спецификации цикла на основе спецификации метода, а после этого нам осталось только написать тело цикла. Для этого мы подумали о том, как изменить переменные, чтобы одновременно уменьшить оценку завершимости и обеспечить соответствие инварианту. Это намного проще, чем думать обо всех возможных последовательностях итераций. Поскольку задача упростилась, мы с большей уверенностью можем положиться на интуицию и неформальные рассуждения при написании тела цикла. Итак, спецификация цикла оказала нам существенную помощь.

Верно также и то, что мы еще не закончили.

13.4.3. Доказательство

Мы сконструировали тело цикла в CoincidenceCount, обдумав три случая. Но верификатор сообщает, что не может доказать соответствие инварианту относительно c. Это значит, что мы где-то допустили ошибку или верификатору нужна наша помощь. В любом случае мы должны написать доказательство вручную.

Для начала выясним, с каким из трех случаев в теле цикла возникли проблемы у верификатора. Скопируем инвариант, вызвавший сообщение верификатора, в инструкцию **assert** в конце каждого случая:

```
assert c + |multiset(a[m..]) * multiset(b[n..])|
    == |multiset(a[..]) * multiset(b[..])|;
```

Этот шаг удовлетворит верификатор. Причина в том, что поток управления программы благополучно преодолевает инструкцию **assert**, только если выполняется указанное условие. После того как мы поместили **assert** в конец каждого пути к концу тела цикла, любой поток управления, достигающий этой точки, будет удовлетворять инварианту цикла, поэтому все претензии к нему отпадают.

Но теперь мы получаем сообщение о невозможности доказать верность утверждений **assert**. Всех трех! Нам предстоит немного поработать. Разберемся с каждым утверждением по отдельности.

13.4.4. Случай совпадения

В случае `a[m] == b[n]` мы неофициально утверждали, что обнаружили совпадение и, следовательно, должны увеличить `c` и перейти к следующим элементам массивов. Другими словами, мы ожидаем, что пересечение элементов `a[m..]` и `b[n..]` должно равняться элементу `a[m]` вместе с пересечением остальных элементов `a[m+1..]` и `b[n+1..]`. Сформулируем это в утверждении в данной ветви **case**:

```
assert multiset(a[m..]) * multiset(b[n..])
    == multiset{a[m]} + (multiset(a[m+1..]) * multiset(b[n+1..]));
```

Верификатор сообщает об ошибке в этом утверждении, но больше не жалуется на утверждение о `c`. Похоже, что наши рассуждения правильные, и нам просто нужно убедить верификатор в справедливости этого утверждения.

Мы могли бы доказать проверяемое условие прямо здесь, в теле `CoincidenceCount`, но я предлагаю записать его в виде отдельной леммы, содержащей это утверждение в виде постусловия. В качестве предусловия леммы выберем из этой ветви **case** весь контекст, который понадобится лемме:

```
lemma MultisetIntersectionPrefix(a: array<int>, b: array<int>,
                                 m: nat, n: nat)
  requires m < a.Length && n < b.Length
  requires a[m] == b[n]
  ensures multiset(a[m..]) * multiset(b[n..])
       == multiset{a[m]} + (multiset(a[m+1..]) * multiset(b[n+1..]))
```

Чтобы убедиться в правильности написанного, вставим вызов этой леммы в ветвь **case**, с которой мы работаем. Из этой ветви мы можем также удалить утверждения **assert**, добавленные для отладки. Итак, первая ветвь **case** в `CoincidenceCount` будет выглядеть так:

```
  case a[m] == b[n] =>
    MultisetIntersectionPrefix(a, b, m, n);
    c, m, n := c + 1, m + 1, n + 1;
```

Теперь верификатор смог доказать верность этого пути через тело цикла, поэтому переключимся на доказательство новой леммы.

Напишем доказательное вычисление, как это часто бывает в леммах. Начнем с левой части постусловия леммы и попытаемся получить правую часть. Перед вычислениями я определю `E` для обозначения одноэлементного мультимножества `a[m]` для удобства:

```
{
  var E := multiset{a[m]};
  calc {
```

```
    multiset(a[m..]) * multiset(b[n..]);
Затем отсечем E от каждого мультимножества:
  ==
    (E + multiset(a[m+1..])) * (E + multiset(b[n+1..]));
```

но верификатор жалуется, что не может этого доказать. Не унывайте! Нам просто нужно дать верификатору более подробную информацию.

На этом этапе мы фактически сделали два маленьких шага. Сначала мы разделили последовательность `a[m..]` на последовательность с одним элементом `[a[m]]` и последовательность с остальными элементами `a[m+1..]` (аналогично мы разделили `b[n..]`). Затем изменили конструкцию мультимножества конкатенации этих двух последовательностей на объединение двух мультимножеств:

```
multiset(A + A') == multiset(A) + multiset(A')
```

где `A` и `A'` – это `[a[m]]` и `a[m+1..]` соответственно.

Первый шаг заменяет в аргументе функции (здесь, **`multiset`**) одно выражение, возвращающее последовательность, другим выражением, возвращающим ту же последовательность. Это должно помочь верификатору. Тот же прием, но применительно к мультимножествам, а не к последовательностям я упоминал в подразделе «Первый совет для доказательств, использующих мультимножества» в разделе 10.2.4. Давайте попробуем сформулировать это равенство в подсказке к шагу вычислений, который мы только что попробовали:

```
    multiset(a[m..]) * multiset(b[n..]);
==   { assert a[m..] == [a[m]] + a[m+1..]
           && b[n..] == [b[n]] + b[n+1..]; }
    (E + multiset(a[m+1..])) * (E + multiset(b[n+1..]));
```

Да, верификатор принял это доказательство! (Видимо, он сам разобрался со вторым шагом.)

Далее распределяем пересечение по объединению:

```
  ==   // дистрибутивность * по +
    E + (multiset(a[m+1..]) * multiset(b[n+1..]));
  }
}
```

На этом мы завершаем доказательство леммы, которую использовали, чтобы доказать, что первый вариант **`case`** в цикле `CoincidenceCount` соблюдает инвариант.

13.4.5. Первый случай несовпадения

В случае `a[m] < b[n]` мы утверждали, что можем исключить элемент `a[m]` из дальнейшего рассмотрения, поскольку он не может быть частью `b[n..]`.

В этой ветви **case** мы можем записать необходимое нам условие в виде утверждения **assert**:

```
assert multiset(a[m..]) * multiset(b[n..])
    == multiset(a[m+1..]) * multiset(b[n..]);
```

Верификатор отвергает это утверждение, но не отвергает больше утверждение о c. Это говорит о том, что мы должны доказать это утверждение.

Напишем лемму, фиксирующую нужное нам свойство, но вместо передачи b и n в виде отдельных параметров будем считать **multiset**(b[n..]) мультимножеством B:

```
lemma MultisetIntersectionAdvance(a: array<int>, m: nat,
                                  B: multiset<int>)
  requires m < a.Length && a[m] !in B
  ensures multiset(a[m..]) * B == multiset(a[m+1..]) * B
```

Чтобы убедиться, что лемма сформулирована правильно, подправим вторую ветвь **case** и вызовем эту лемму:

```
  case a[m] < b[n] =>
    MultisetIntersectionAdvance(a, m, multiset(b[n..]));
    m := m + 1;
```

Верификатор подтверждает этот вариант, поэтому далее мы переключаемся на доказательство леммы. Начнем доказательство, следуя аналогии с предыдущей леммой:

```
{
  var E := multiset{a[m]};
  calc {
    multiset(a[m..]) * B;
  ==   { assert a[m..] == [a[m]] + a[m+1..]; }
    (E + multiset(a[m+1..])) * B;
```

Далее используем закон дистрибутивности пересечения относительно объединения:

```
  ==   // дистрибутивность * по +
    (E * B) + (multiset(a[m+1..]) * B);
```

Теперь верификатор сможет самостоятельно восполнить недостающую часть доказательства, но, как мне кажется, будет нелишним включить в вычисления еще один шаг:

```
  ==   { assert E * B == multiset{}; }
    multiset(a[m+1..]) * B;
  }
}
```

Два варианта доказаны, остался еще один.

13.4.6. Второй случай несовпадения

Случай `b[n] < a[m]` симметричен только что рассмотренному случаю `a[m] < b[n]`. Но не будем писать еще одну похожую лемму, а воспользуемся предыдущей. Используем ее так:

```
MultisetIntersectionAdvance(b, n, multiset(a[m..]));
```

К сожалению, в этой третьей ветви **case** верификатор не смог доказать утверждение о `c`, поэтому нам придется написать доказательство вручную.

Разница между этим случаем и предыдущим в том, что `a` и `b` (а также `m` и `n`) поменялись местами. Вызвав ту же лемму, которую разработали выше, мы получаем свойство, где `b` находится в левом аргументе пересечения, а `a` – в правом. Мы с вами знаем, что операция пересечения коммутативна, поэтому порядок не должен иметь значения. Но это снова сравнение мультимножеств, которое используется в функции (здесь функция `|_|`, возвращающая мощность мультимножества), поэтому мы должны явно указать это сравнение, чтобы верификатор смог его использовать.

Заменим последнюю ветвь **case** в теле цикла этим доказательным вычислением (и присваиванием `n`):

```
case b[n] < a[m] =>
  calc {
    multiset(a[m..]) * multiset(b[n..]);
  ==   // операция объединения мультимножеств коммутативна
    multiset(b[n..]) * multiset(a[m..]);
==   { MultisetIntersectionAdvance(b, n,
                                   multiset(a[m..])); }
  multiset(b[n+1..]) * multiset(a[m..]);
==   // операция объединения мультимножеств коммутативна
  multiset(a[m..]) * multiset(b[n+1..]);
}
n := n + 1;
```

На этом доказательство метода `CoincidenceCount` завершено.

Упражнение 13.13

Напишите более симметричную лемму, которую можно использовать в обоих случаях несовпадения.

13.4.7. Краткий обзор пройденного пути

Чтобы получить метод, вычисляющий количество совпадений в двух отсортированных массивах, мы написали спецификацию метода (в терминах мультимножеств), затем разработали спецификацию цикла, а потом, когда осталось решить гораздо меньшую задачу, использовали неформальные рассуждения для реализации тела цикла (т. е. одной произвольной итерации цикла).

Для доказательства корректности тела цикла потребовало приложить гораздо больше усилий. Нам пришлось вручную представить различные этапы доказательства, в основном чтобы получить *экстенсиональность* последовательностей и мультимножеств. Я показал на примере, как отладить неудачную попытку верификатора автоматически создать все доказательство. В таких случаях проблема обычно кроется либо в наших собственных рассуждениях, либо в необходимости представить дополнительные доказательства, поэтому мы начали писать доказательство самостоятельно, последовав рекомендациям в главе 5 и во многих последующих главах.

13.5. Поиск точки на наклонной местности

Обсуждаемый нами алгоритм ищет ключ (точку) на наклонной поверхности. Роль поверхности играет прямоугольная область, где любой шаг на север или восток приведет к увеличению (или уменьшению) высоты местоположения. Обычно высоты на двухмерной карте изображаются *изолиниями*, соединяющими точки с одинаковой высотой. Для иллюстрации я нарисовал такие изолинии на координатной сетке на рис. 13.0.

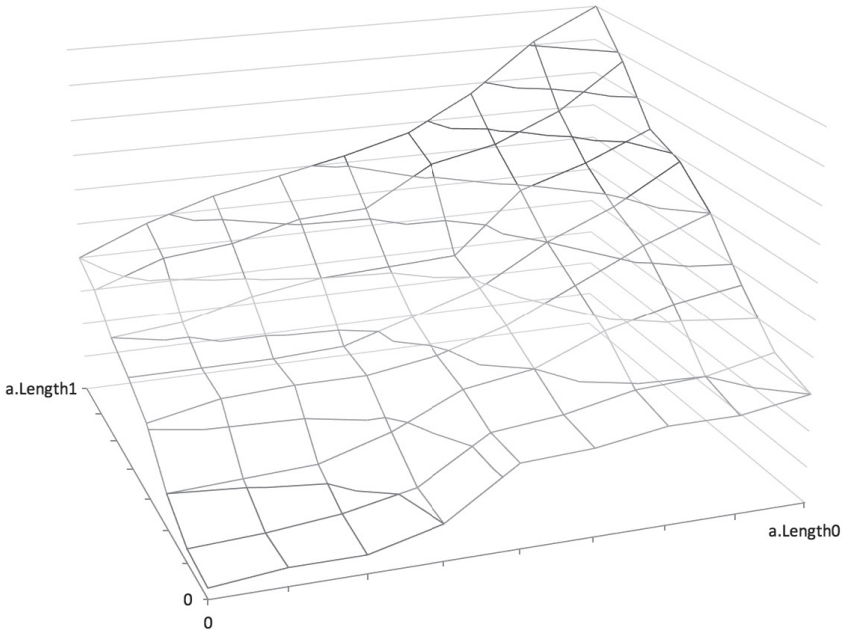


Рис. 13.0. Пример входных данных для поиска точки на наклонной местности. На диаграмме изображены изолинии, показывающие, где местность (входная матрица значений) пересекается с горизонталями различных высот

Самая низкая точка в этом примере находится в нижнем левом углу, а самая высокая – в верхнем правом. Координатная сетка напоминает двумерный массив (т. е. матрицу) с входными данными для поиска. Массив со-

держит целые числа, обозначающие высоту, а «ключ», который мы должны найти, – это точка, имеющая определенную высоту.

В терминах матрицы `a` наклон поверхности выражается следующими двумя кванторами:

```
(forall i, j, j' ::
  0 <= i < a.Length0 && 0 <= j < j' < a.Length1 ==>
    a[i,j] <= a[i,j']) &&
(forall i, i', j ::
  0 <= i < i' < a.Length0 && 0 <= j < a.Length1 ==>
    a[i,j] <= a[i',j])
```

Нам также известно, что искомая точка действительно существует. На основе этих фрагментов информации мы можем написать спецификацию реализуемого метода:

```
method SlopeSearch(a: array2<int>, key: int) returns (m: int, n: int)
  requires forall i, j, j' ::
    0 <= i < a.Length0 && 0 <= j < j' < a.Length1 ==>
      a[i,j] <= a[i,j']
  requires forall i, i', j ::
    0 <= i < i' < a.Length0 && 0 <= j < a.Length1 ==>
      a[i,j] <= a[i',j]
  requires exists i, j ::
    0 <= i < a.Length0 && 0 <= j < a.Length1 && a[i,j] == key
  ensures 0 <= m < a.Length0 && 0 <= n < a.Length1
  ensures a[m,n] == key
```

13.5.0. Начальная позиция поиска

Роль индекса в нашей программе будет играть не одно, а два целых числа, индексирующих два измерения в матрице. Первое, что приходит в голову, – начать обход значений в матрице с начала координат и итеративно увеличивать один или оба индекса. Через некоторое время поиск достигнет изолинии ключа, т. е. места, где значения с одной стороны меньше ключа, а с другой стороны равны или выше ключа. И что дальше? Продолжать ли поиск вдоль изолинии по часовой стрелке или против часовой стрелки? Самый простой способ последовать вдоль изолинии – начать с одного из ее концов. Мы можем для начала отыскать пересечение нужной нам изолинии с осью X или, наоборот, – с осью Y. Конкретный выбор неважен, поэтому начнем с поиска пересечения изолинии с осью Y.

Хорошо, а как найти пересечение требуемой изолинии с осью Y? Это можно сделать, применив линейный поиск от начала координат. Или использовав более быстрый алгоритм двоичного поиска. В этом примере я использую линейный поиск, но начну с противоположной от начала координат стороны (т. е. с верхнего левого угла сетки). Это даст нам простой

алгоритм с одним циклом, в котором каждый шаг будет выполняться либо вправо, либо вниз.

Инвариант перемещения индекса цикла вправо и вниз заключается в том, что искомый ключ находится в прямоугольнике, верхний левый угол которого является индексом нашего цикла.

13.5.1. Реализация

Я только что описал основные части алгоритма поиска точки на наклонной поверхности: начальную позицию, инвариант и порядок следования вдоль изолинии. Поскольку изолиния проходит между отдельными точками матрицы, с трудом верится, что наша программа случайно не пропустит ключ. Давайте напишем программу и посмотрим, сможем ли мы доказать ее корректность.

Сделаем выходные параметры `m,n` индексом цикла и будем поддерживать как инвариант корректность индекса `m,n`. Исходя из этого, легко написать условие цикла. Вот как выглядят инициализация и спецификация цикла:

```
{
  m, n := 0, a.Length1 - 1;
  while a[m,n] != key
    invariant 0 <= m < a.Length0 && 0 <= n < a.Length1
    invariant exists i, j ::
      m <= i < a.Length0 && 0 <= j <= n && a[i,j] == key
  decreases a.Length0 - m + n
}
```

Постусловие вытекает из первого инварианта и отрицания условия цикла. Фактически это пример метода конструирования циклов 11.0 *с помощью элиминации конъюнкта*. Второй инвариант цикла является примером *замены константы переменной в предусловии* (метод конструирования циклов 13.0). Поскольку мы решили, что `m,n` будет правильным индексом массива, а не границей нашего поиска, обратите внимание, что константа, которую заменяет переменная `n`, равна `a.Length1 - 1`.

Инструкция **`decreases`** говорит о нашем решении увеличивать `m` и уменьшать `n`.

В теле цикла нам нужно сравнить `a[m,n]` с ключом. Если `a[m,n]` меньше ключа, то мы переходим на более высокую отметку, увеличивая `m`. Если `a[m,n]` больше ключа, то мы переходим на более низкую отметку, уменьшая `n`:

```
{
  if a[m,n] < key {
    m := m + 1;
  } else {
    n := n - 1;
```

```
    }
  }
```

Этот код соблюдает инвариант и уменьшает меру завершимости, так что мы закончили!

13.6. Поиск каньона

Задача поиска каньона похожа на задачу поиска точки на наклонной поверхности, только на этот раз требуется найти точку не на наклонной поверхности, а в каньоне. В частности, нам даны два отсортированных массива, `a` и `b`, и мы должны найти минимальную абсолютную разность между элементом `a` и элементом `b`.

Для начала напишем спецификацию, затем немного подумаем над задачей и, наконец, реализуем метод с помощью цикла.

13.6.0. Спецификация метода

В задаче упоминается абсолютная разность, которую также можно назвать расстоянием между двумя значениями. Определим функцию, выражающую расстояние:

```
function Dist(x: int, y: int): nat {
  if x < y then y - x else x - y
}
```

Наш метод получает два непустых отсортированных массива и возвращает меру расстояния:

```
method CanyonSearch(a: array<int>, b: array<int>) returns (d: nat)
  requires a.Length != 0 && b.Length != 0
  requires forall i, j :: 0 <= i < j < a.Length ==> a[i] <= a[j]
  requires forall i, j :: 0 <= i < j < b.Length ==> b[i] <= b[j]
```

В качестве постусловия используем представление минимума, которое мы определили выше в разделе 13.3. Постусловие гласит, что минимум находится среди возможных значений и что минимум не превышает любого возможного значения:

```
ensures exists i, j ::
  0 <= i < a.Length && 0 <= j < b.Length && d == Dist(a[i], b[j])
ensures
  forall i, j :: 0 <= i < a.Length && 0 <= j < b.Length ==>
    d <= Dist(a[i], b[j])
```

13.6.1. О каньоне

Имея два массива, мы можем представить поверхность `a.Length * b.Length`, где элементом с индексом `i,j` является `Dist(a[i], b[j])`. Рассмотрим попе-

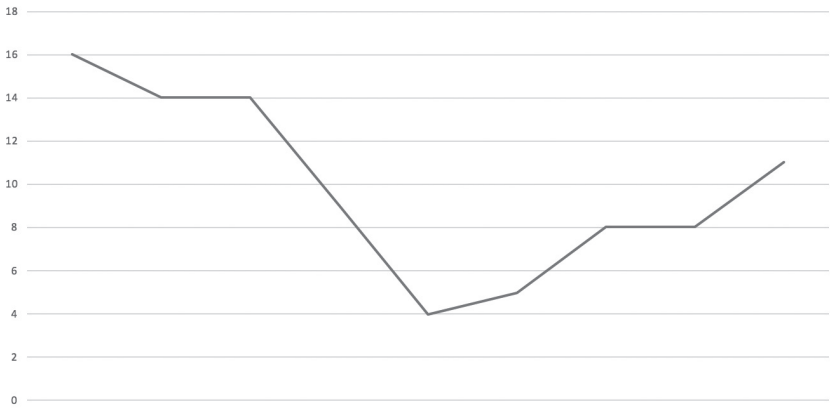
речное сечение этой матрицы, скажем вдоль оси `a`, где `j` фиксирован. Для примера предположим, что элементы `a` равны

```
3, 5, 5, 10, 23, 24, 27, 27, 30
```

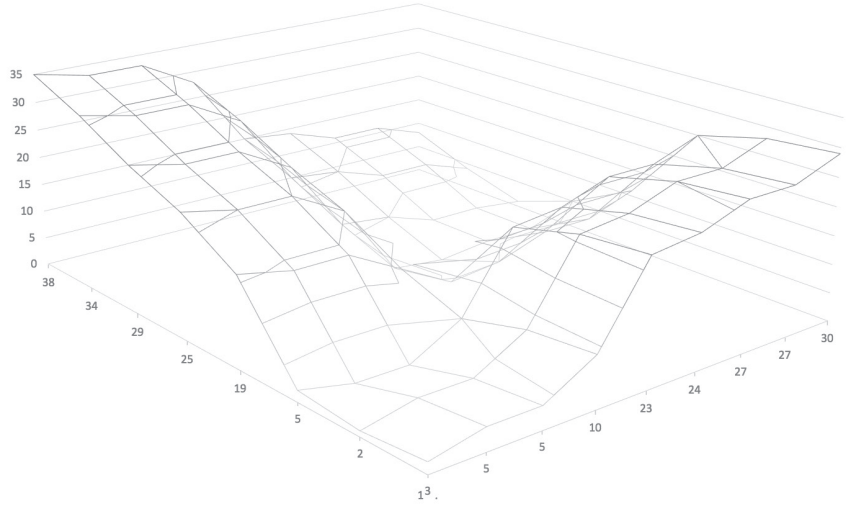
и элемент `b[j]` имеет значение `19`. Расстояния между значениями `a` и `b[j]` равны

```
16, 14, 14, 9, 4, 5, 8, 8, 11
```

которые можем изобразить на графике следующим образом:

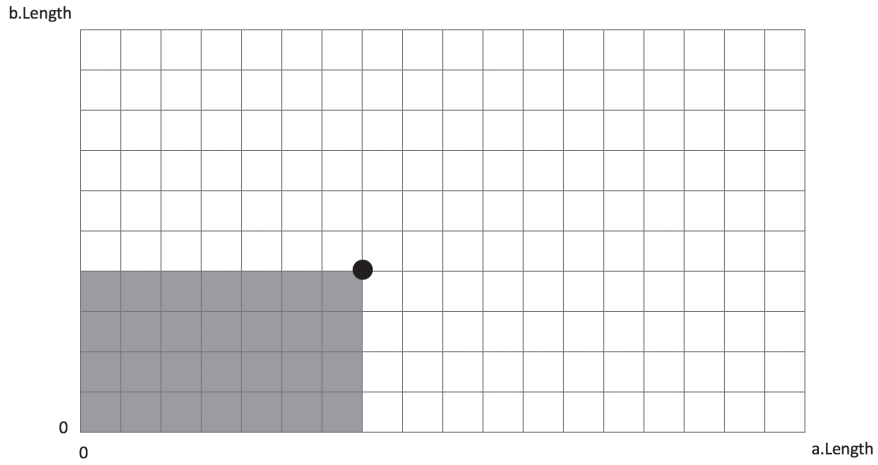


Эта диаграмма имеет форму буквы «V», нижняя точка которой находится там, где значение `a` ближе всего к `b[j]`. Это будет верно для каждого сечения в обоих измерениях. Таким образом, весь двумерный ландшафт выглядит как каньон, что и объясняет название алгоритма. Следующая диаграмма поможет вам визуально представить ситуацию:

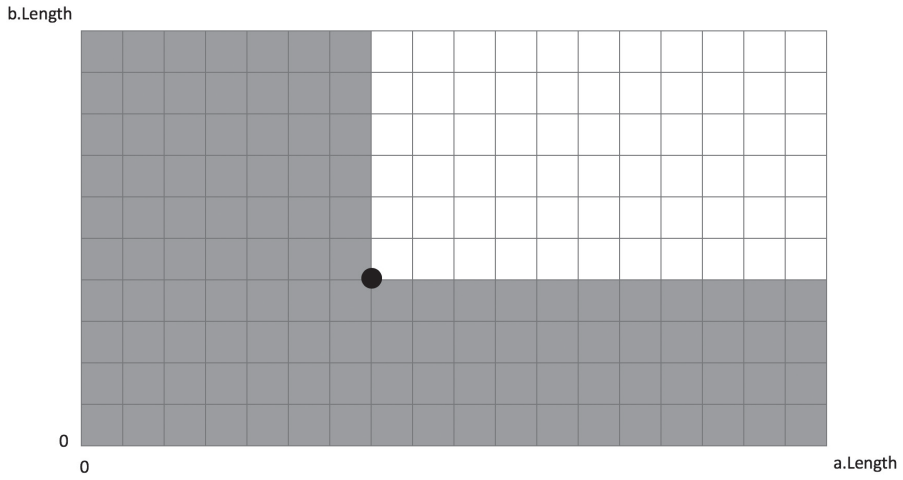


Форма каньона дает нам надежду, что мы сможем написать алгоритм, выполняющий поиск вдоль дна каньона и отыскивающий точку, в которой индекс цикла оказывается ближе всего к самой нижней точке из всех V-образных сечений.

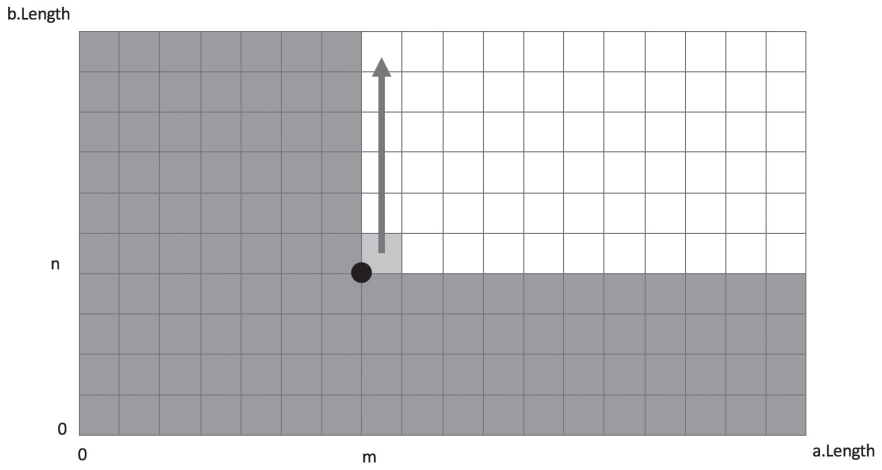
Для инварианта нужно включить в постусловие что-то похожее на квантор всеобщности. В постусловии говорится, что `d` – меньшее из расстояний между всеми парами точек. Мы могли бы указать инвариант, говорящий, что `d` находится ниже всех точек, где обе координаты меньше индекса цикла, т. е. ниже точек в заштрихованной области на следующей диаграмме:



Однако это не так просто, когда индекс цикла достигает края рассматриваемой местности. Лучше стремиться к инварианту, в котором `d` находится ниже всех точек в заштрихованной области на следующей диаграмме, координаты которых меньше индекса цикла:



Как соблюсти такой инвариант? Пусть `m,n` – индекс цикла. Если `a[m]` ниже `b[n]`, то `a[m]` ниже всех значений в `b`, начиная с индекса `n` и больше. На рисунке ниже светло-серый квадрат



представляет `Dist(a[m], b[n])`. Если `a[m] <= b[n]`, то расстояния растут при смещении вдоль стрелки. Значит `d` нужно обновлять, только если `Dist(a[m], b[n])` является новым минимумом, и тогда мы сможем соблюсти инвариант, увеличивая `m`. Аналогично если `b[n] <= a[m]`, то можно увеличить `n`.

13.6.2. Реализация метода

Теперь мы можем приступить к реализации метода с применением цикла. Используем индексы цикла `m` и `n`:

```
var m, n := 0, 0;
while m < a.Length && n < b.Length
  invariant 0 <= m <= a.Length && 0 <= n <= b.Length
  decreases a.Length - m + b.Length - n
```

Чтобы обеспечить соблюдение постусловия, описывающего свойство существования, *всегда* применяется метод конструирования циклов 13.1:

```
invariant exists i, j ::
  0 <= i < a.Length && 0 <= j < b.Length &&
    d == Dist(a[i], b[j])
```

Это означает, что мы должны инициализировать `d` расстоянием между любыми двумя значениями массива. Для этого вполне подойдет расстояние `Dist(a[0], b[0])`, поэтому запишем следующую инструкцию в начале тела метода перед циклом:

```
{
  d := Dist(a[0], b[0]);
```

Осталось добавить квантор всеобщности, присутствующий в постусловии метода. Выше мы определили, что он должен охватывать L-образную область. Сделаем это, исключив точки `i,j` такие, что

```
m <= i < a.Length && n <= j < b.Length
```

Соответственно, последний инвариант принимает вид:

```
    invariant forall i, j ::
      0 <= i < a.Length && 0 <= j < b.Length ==>
        d <= Dist(a[i], b[j]) || (m <= i && n <= j)
}
```

При определении этого инварианта мы применили известный метод.

> **Метод конструирования циклов 13.2** (Ослабление)
>
> Чтобы соблюсти постусловие `Q`, используйте инвариант цикла, ослабляющий `Q` предикатом `R`.

В теле цикла мы сначала обновим `d`, если текущая позиция является новым минимумом:

```
  {
    d := if Dist(b[n], a[m]) < d then Dist(b[n], a[m]) else d;
```

Остальная часть тела цикла увеличивает `m` или `n`, согласно нашим наблюдениям в разделе 13.6.1:

```
    if
    case a[m] <= b[n] =>
      m := m + 1;
    case b[n] <= a[m] =>
      n := n + 1;
  }
```

Верификатор доволен этой программой, а значит, мы закончили!

Упражнение 13.14

Написанный мной оператор **`if-case`** позволяет использовать любую альтернативу, если `a[m]` и `b[n]` равны. Однако если они равны, то `Dist(a[m], b[n]) == 0` и нет смысла продолжать поиск. Измените оператор **`if`**, чтобы отразить это наблюдение.

Упражнение 13.15

Пусть есть три отсортированных массива `a`, `b` и `c`, определите и реализуйте программу, отыскивающую наименьшее значение выражения:

```
Dist(x, y) + Dist(y, z) + Dist(z, x)
```

где `x,y,z` – индексы элементов `a,b,c` соответственно.

13.7. Голосование большинством

В последнем примере в этой главе я представлю интересный алгоритм Бойера и Мура под названием «голосование большинством». По условиям задачи дается набор голосов. Каждый голос отдается за одного кандидата, и выбирается кандидат, имеющий строгое большинство голосов. То есть за выбранного кандидата должно быть отдано строго больше половины голосов. Задача состоит в том, чтобы найти этого кандидата. И сделать это нужно за линейное время, т. е. проверив каждый голос только один раз.

Вместо массивов я буду использовать в этой программе последовательности (см. раздел 13.0.4). В рамках примера я покажу еще один метод спецификации, включающий призрачные параметры.

13.7.0. Подсчет вхождений

Вот функция для подсчета вхождений в последовательность:

```
function Count<T(==)>(a: seq<T>, lo: int, hi: int, x: T): nat
  requires 0 <= lo <= hi <= |a|
{
  if lo == hi then
    0
  else
    Count(a, lo, hi - 1, x) + if a[hi - 1] == x then 1 else 0
}
```

Она возвращает количество вхождений `x` в последовательность `a` от индекса `lo` до индекса `hi`. Поскольку эта компилируемая функция должна сравнивать элементы типа `T`, параметр типа ограничен типами, поддерживающими сравнение, на что указывает суффикс `(==)` (см. также раздел 6.5). Тело `Count` напоминает тело функции `SumDown` из раздела 12.3.0, на примере которой был продемонстрирован инвариант «что было сделано» (см. раздел 12.3).

Давайте ненадолго задержим наше внимание на этой функции и докажем несколько ее свойств. Одно из свойств гласит, что количество вхождений в подпоследовательность `a[lo..hi]` можно разделить на количество вхождений в смежных интервалах `a[lo..mid]` и в `a[mid..hi]`:

```
lemma SplitCount<T>(a: seq<T>, lo: int, mid: int, hi: int, x: T)
  requires 0 <= lo <= mid <= hi <= |a|
  ensures Count(a, lo, mid, x) + Count(a, mid, hi, x)
       == Count(a, lo, hi, x);
{
}
```

Доказательство выполняется механизмом автоматической индукции в Dafny.

Врезка 13.0

Два полуоткрытых интервала считаются смежными, если верхняя граница одного совпадает с нижней границей другого. Смежные интервалы можно объединить, отбросив две общие границы. Например,

```
a[3..9] + a[9..15] == a[3..15].
```

Это выражение выполняет конкатенацию 9 - 3 элементов, начиная с 3-го, и 15 - 9 элементов, начиная с 9-го, дающую 15 - 3 элементов, начиная с 3-го.

Другой пример: если `lo <= mid <= hi`, то два квантора в конъюнкции

```
(forall i :: lo <= i < mid ==> P(i)) &&
(forall i :: mid <= i < hi ==> P(i))
```

можно объединить в один:

```
forall i :: lo <= i < hi ==> P(i).
```

Иначе говоря, если `P` выполняется для интервалов от `lo` до `mid` и от `mid` до `hi` (напомню, что я использую слово «до» для обозначения полуоткрытых интервалов, см. врезку 1.0), то `P` выполняется на всем интервале от `lo` до `hi`. Суммирование выглядит так:

$$\sum_{i:=lo}^{>mid} F(i) + \sum_{i:=mid}^{>hi} F(i) = \sum_{i:=lo}^{>hi} F(i)$$

Другое свойство гласит, что число вхождений двух разных элементов в подпоследовательность `a[lo..hi]` не превосходит `hi - lo`, и его доказательство тоже осуществляется механизмом автоматической индукции Dafny без нашего вмешательства:

```
lemma DistinctCounts<T>(a: seq<T>, lo: int, hi: int, x: T, y: T)
  requires 0 <= lo <= hi <= |a|
  ensures x != y ==>
    Count(a, lo, hi, x) + Count(a, lo, hi, y) <= hi - lo
{
}
```

Я решил записать `x != y` как посылку импликации в постусловии леммы вместо предусловия леммы. Это означает, что лемму можно вызвать независимо от того, различны ли `x` и `y`, и, таким образом, вызывающая сторона

получает возможность использовать *противопоставление* этой импликации. (Положительные и отрицательные стороны этих двух стилей определения спецификаций в леммах вы найдете в разделе 7.1, в обсуждении леммы `LessTransitive`.)

Упражнение 13.16

Добавьте в обе леммы, приведенные выше, атрибут `{:induction false}`, чтобы отключить автоматическую индукцию. Затем напишите доказательство для каждой из них.

13.7.1. Спецификация метода

Чтобы конкретизировать задачу голосования большинством, определим предикат, который определяет значение понятия «строгое большинство». Определить, имеет ли `x` строгое большинство среди голосов от индекса `lo` до `hi`, можно так:

```
(hi - lo) / 2 < Count(a, lo, hi, x)
```

Однако мне проще интерпретировать выражение, имея `2 *` вместо `/ 2` слева, поэтому я определю предикат следующим образом:

```
  predicate HasMajority<T(==)>(a: seq<T>, lo: int, hi: int, x: T)
  requires 0 <= lo <= hi <= |a|
{
  hi - lo < 2 * Count(a, lo, hi, x)
}
```

Теперь напишем спецификацию метода, определяющего победителя голосования:

```
method FindWinner'<Candidate(==)>(a: seq<Candidate>)
      returns (k: Candidate)
  requires exists K :: HasMajority(a, 0, |a|, K)
  ensures HasMajority(a, 0, |a|, k)
```

Предусловие гласит, что победитель, избираемый строгим большинством голосов, обязательно существует. Как оказывается, назначение имени победителю дает дополнительное удобство при реализации метода, но область действия экзистенциально связанной переменной `K` ограничивается предусловием. Вот тут-то и пригодится прием использования призрачного параметра, о котором я рассказывал выше. Перепишем спецификацию метода:

```
method FindWinner<Candidate(==)>(a: seq<Candidate>, ghost K: Candidate)
      returns (k: Candidate)
  requires HasMajority(a, 0, |a|, K)
  ensures k == K
```

На первый взгляд эта спецификация выглядит немного странной, обещая найти победителя при условии, что победитель уже передан в виде параметра. Метод с такой спецификацией легко реализовать одним присваиванием:

```
k := K;
```

Но не все так просто. Параметр K является призрачным, и мы не можем присвоить его компилируемой переменной. Соответственно, реализация метода должна вычислять k каким-то другим способом. K используется только в спецификации. Давайте пока сосредоточимся на реализации метода, а позже я вернусь к вопросу (или к загадке, если хотите) о том, как вызывающая сторона сможет передать значение для K.

13.7.2. Разработка реализации

Следуя стандартным путем, используем индекс цикла, скажем n, который принимает значения в диапазоне от 0 до |a|. По ходу дела мы будем использовать k для отслеживания «ведущего кандидата», т. е. кандидата, получившего наибольшее количество из первых n голосов. Мы также используем переменную c для отслеживания количества голосов, отданных за k. Этот подход прекрасно работает, пока k продолжает оставаться ведущим кандидатом. Однако может случиться так, что какой-то кандидат наберет большинство голосов в начале последовательности, но позже его обгонит другой кандидат. Когда цикл обнаруживает, что ведущий кандидат больше не является ведущим, он должен изменить k, но как обновить c без повторного сканирования уже просмотренной части последовательности?

Красота предлагаемого алгоритма заключается в возможности игнорировать любую начальную часть последовательности, в которой нет кандидата, набравшего строгое большинство голосов. Рассмотрим следующую диаграмму, где показаны первые n и оставшиеся |a| - n голосов

Согласно условию за K отдано большинство голосов из |a|. Другими словами, плотность голосов за K в a больше 1/2. Если часть последовательности имеет K-плотность меньше 1/2, то остальная часть последовательности должна иметь K-плотность выше 1/2. Итак, если среди первых n элементов a нет кандидата с большинством голосов, мы можем просто игнорировать эти элементы, поскольку искомый победитель имеет большинство среди остальных элементов.

Вместо вычисления k и c в a[..n] мы вычислим их для a[lo..hi]. То есть, помимо переменных k и c, мы будем использовать еще две переменные: lo и hi. hi – это длина начальной части последовательности a, просмотренной к текущему моменту, а lo – длина начальной части последовательности a, в которой отсутствует искомый победитель.

13.7.3. Спецификация цикла

На основе диаграммы, нарисованной выше, напишем следующую спецификацию цикла:

```
{
  k := a[0];
  var lo, hi, c := 0, 1, 1;
  while hi < |a|
    invariant 0 <= lo <= hi <= |a|
    invariant c == Count(a, lo, hi, k)
    invariant HasMajority(a, lo, hi, k)
    invariant HasMajority(a, lo, |a|, K)
```

Но этого недостаточно для доказательства постусловия, поскольку верификатор выводит сообщение об ошибке (которое можно увидеть, добавив фигурную скобку }, закрывающую тело метода). Инвариант и отрицание условия цикла дают нам

```
HasMajority(a, lo, |a|, k) && HasMajority(a, lo, |a|, K)
```

Эта спецификация гласит, что K и k имеют большинство голосов в части последовательности. Интуитивно понятно, что только один кандидат может получить строгое большинство, поэтому имеющиеся у нас условия должны подразумевать k == K. Однако мы должны помочь верификатору увидеть это. К счастью, одна из лемм, которые мы написали в разделе 13.7.0, вместе с определением HasMajority выражает именно это. Поэтому закончим тело метода так:

```
  DistinctCounts(a, lo, |a|, k, K);
}
```

и теперь нам осталось лишь написать тело цикла.

13.7.4. Реализация цикла

Если следующий голос, который будет рассмотрен, отдан за ведущего кандидата, то нам остается только увеличить (индекс цикла и) текущий счетчик голосов, отданных за этого кандидата:

```
{
  if a[hi] == k {
    hi, c := hi + 1, c + 1;
```

Даже если a[hi] является голосом за другого кандидата, вполне возможно, что k останется лидером в a[lo..hi+1]:

```
} else if hi + 1 - lo < 2 * c {
  hi := hi + 1;
```

В оставшемся случае может обнаружиться, что в этом голосовании нет победителя со строгим большинством, поэтому мы проигнорируем часть a до hi + 1:

```
} else {
  hi := hi + 1;
```

Далее нам нужно сбросить переменные, чтобы начать поиск заново. Самая трудная часть для соблюдения инвариантов цикла – последняя. Давайте порассуждаем о ней.

На данный момент мы знаем, что k имеет ровно половину голосов в a[lo..hi]:

```
calc {
  true;
==>
  2 * Count(a, lo, hi, k) == hi - lo;
```

Это означает, что ни один кандидат, включая K, не может иметь более половины голосов: если K и k – это одно и то же, то K имеет ровно половину голосов; в противном случае лемма DistinctCounts из раздела 13.7.0 говорит, что у K не больше голосов, чем у другой половины:

```
==>   { DistinctCounts(a, lo, hi, k, K); }
  2 * Count(a, lo, hi, K) <= hi - lo;
```

Инвариант входа в цикл гласит, что K имеет большинство голосов в a[lo..|a|], поэтому, разделив диапазон на a[lo..hi] и a[hi..|a|], мы приходим к выводу, что K имеет большинство голосов в последнем из них:

```
==>   { SplitCount(a, lo, hi, |a|, K); }
  |a| - hi < 2 * Count(a, hi, |a|, K);
==   // определение HasMajority
  HasMajority(a, hi, |a|, K);
}
```

Если вы не хотите включать нить рассуждений в это доказательное вычисление, то можете заменить ее простыми вызовами двух лемм:

```
DistinctCounts(a, lo, hi, k, K);
SplitCount(a, lo, hi, |a|, K);
```

Теперь мы готовы написать окончательное присваивание в этой ветви инструкции **if** в теле цикла:

```
      k, lo, hi, c := a[hi], hi, hi + 1, 1;
    }
  }
```

На этом мы заканчиваем реализацию метода `FindWinner`.

13.7.5. Определение победителя, если он есть

В разделе 13.7.1 я говорил, что вернусь к загадке того, как код, вызывающий метод `FindWinner`, может передать `K` – значение, которое `FindWinner` должен вычислить. Сейчас я раскрою разгадку этой тайны.

Инструкция «присвоить такое значение, что»

Для этого я сначала должен представить инструкцию «присвоить такое значение, что» (assign-such-that) в языке Dafny. Она имеет форму

```
x :| P;
```

и присваивает переменной `x` такое значение, что выполняется предикат `P`. Проверяемым условием этой инструкции является существование значения, которое `x` может принять. Например, инструкция

```
x :| 0 <= x < N;
```

присвоит `x` произвольное натуральное число, меньшее `N`. Это возможно, только если `N` строго положительно, поэтому доказательство корректности данной инструкции заключается именно в этом. Говоря техническим языком, проверяемое условие имеет форму:

```
exists x :: 0 <= x < N
```

но в данном случае это означает, что `N` строго больше `0`.

Реализация FindWinner'

Инструкцию «присвоить такое значение, что» можно использовать в реализации метода `FindWinner'`, представленного в разделе 13.7.1, имеющего более традиционное предусловие и, более того, не имеющего призрачного входного параметра:

```
method FindWinner'<Candidate(==)>(a: seq<Candidate>)
      returns (k: Candidate)
  requires exists K :: HasMajority(a, 0, |a|, K)
  ensures HasMajority(a, 0, |a|, k)
{
  ghost var K :| HasMajority(a, 0, |a|, K);
  k := FindWinner(a, K);
}
```

Этот метод вводит призрачную переменную `K` и присваивает ей значение, удовлетворяющее предикату `HasMajority(a, 0, |a|, K)`. Это связано с

условием существования такого K, которое следует непосредственно из предусловия FindWinner'. После присваивания значения переменной K приведенный выше код тут же использует ее в вызове FindWinner.

Кстати, нет причин беспокоиться о том, как оператор «присвоить такое значение, что» находит требуемое значение K, поскольку это призрачное присваивание. Мы доказываем, что значение K существует, и это все, что нам нужно знать.

Поиск победителя

Метод FindWinner' демонстрирует полезную базовую идею, но как метод он не более полезен, чем FindWinner. Я имею в виду, что пока непонятно, как FindWinner' поможет доказать существование такого K. Раз уж мы зашли так далеко, давайте сделаем еще два шага и превратим алгоритм во что-то полезное, что мы сможем вызвать.

Внесем небольшие изменения в FindWinner, чтобы его можно было вызывать даже в отсутствие кандидата, набравшего строгое большинство голосов. Если такого кандидата нет, то мы ничего не можем сказать о возвращаемом значении. Наш первый шаг на пути к получению полезного метода – превратить FindWinner в новый метод SearchForWinner, принимающий еще один призрачный параметр hasWinner. Если hasWinner имеет значение true, то SearchForWinner будет действовать как FindWinner и вернет кандидата, получившего строгое большинство голосов. Если hasWinner имеет значение false, то у нас нет никаких гарантий относительно возвращаемого значения:

```
method SearchForWinner<Candidate(==)>(a: seq<Candidate>,
                                      ghost hasWinner: bool,
                                      ghost K: Candidate)
      returns (k: Candidate)
  requires |a| != 0
  requires hasWinner ==> HasMajority(a, 0, |a|, K)
  ensures hasWinner ==> k == K
{
  k := a[0];
  var lo, hi, c := 0, 1, 1;
  while hi < |a|
    invariant 0 <= lo <= hi <= |a|
    invariant c == Count(a, lo, hi, k)
    invariant HasMajority(a, lo, hi, k)
    invariant hasWinner ==> HasMajority(a, lo, |a|, K)
  {
    if a[hi] == k {
      hi, c := hi + 1, c + 1;
    } else if hi + 1 - lo < 2 * c {
      hi := hi + 1;
    } else {
```

```
        hi := hi + 1;
        DistinctCounts(a, lo, hi, k, K);
        SplitCount(a, lo, hi, |a|, K);
        if hi == |a| {
          return;
        }
        k, lo, hi, c := a[hi], hi, hi + 1, 1;
      }
    }
    DistinctCounts(a, lo, |a|, k, K);
  }
```

Различия между этим методом и `FindWinner` заключаются в следующем:

- предусловие `HasMajority(a, 0, |a|, K)` должно выполняться, только если `hasWinner` имеет значение **`true`**;
- предусловие `HasMajority` метода `FindWinner` подразумевает, что `a` – непустая последовательность, но поскольку мы ослабили это предусловие, то добавили в `SearchForWinner` дополнительное предусловие `|a| != 0`;
- постусловие `k == K` аналогично ослабляется до `hasWinner ==> k == K`;
- четвертый инвариант цикла также получает посылку `hasWinner`, но остальные инварианты такие же, как в `FindWinner`;
- сбросив переменные, чтобы начать поиск заново, мы должны сначала проверить – исследовали ли мы все элементы последовательности. В `FindWinner` в этом не было необходимости, поскольку предполагалось, что в последовательности есть кандидат, имеющий строгое большинство. Новый метод внезапно завершает цикл с помощью инструкции **`return`**, если все элементы последовательности были просмотрены.

Метод `SearchForWinner` можно вызвать с произвольным значением `K`, просто передав **`false`** в `hasWinner`. Но мы не должны поступать так всегда, потому что тогда мы ничему не научимся тому, что делает `SearchForWinner`.

Теперь сделаем второй шаг.

Не загадочное, но довольно фантастическое использование SearchForWinner

Теперь мы готовы написать метод без каких-либо предположений о входной последовательности, а также без призрачных параметров. Чтобы метод мог сообщить вызывающему коду о наличии победителя со строгим большинством голосов, я объявлю следующий тип:

```
datatype Result<Candidate> = NoWinner | Winner(Candidate)
```

Спецификация метода теперь выглядит следующим образом:

```
method DetermineElection<Candidate(==)>(a: seq<Candidate>)
      returns (result: Result<Candidate>)
  ensures match result
    case Winner(c) => HasMajority(a, 0, |a|, c)
    case NoWinner => !exists c :: HasMajority(a, 0, |a|, c)
```

Этот метод точно сообщает, есть ли кандидат, получивший строгое большинство голосов; и если есть, то возвращает этого кандидата.

Реализация начинается с проверки пустой последовательности:

```
{
  if |a| == 0 {
    return NoWinner;
  }
```

Это связано с тем, что метод `SearchForWinner` можно вызвать, только если `a` не пустая последовательность.

Далее реализация вводит призрачную переменную `hasWinner` и присваивает ей логическое значение, которое указывает, есть ли победитель:

```
  ghost var hasWinner := exists c :: HasMajority(a, 0, |a|, c);
```

Классная инструкция, согласитесь!

После этого реализация вводит еще одну призрачную переменную `w`, которой присваивается такое значение, что если есть победитель, то `w` и есть этот победитель, а если победителя нет, то `w` получает произвольное значение своего типа:

```
  ghost var w;
  if hasWinner {
    w :| HasMajority(a, 0, |a|, w);
  } else {
    w := a[0];
  }
```

Инструкция «присвоить такое значение, что» включает проверяемое условие существования значения для `w`, и оно следует из условия цикла `hasWinner` и инструкции присваивания `hasWinner`.

Имея эти две призрачные переменные (и зная, что `|a| != 0`), мы можем вызвать `SearchForWinner`:

```
  var c := SearchForWinner(a, hasWinner, w);
```

Значение `c` соответствует кандидату, набравшему строгое большинство голосов, если он есть. Поэтому было бы хорошо иметь возможность вернуть его:

```
  if hasWinner then Winner(c) else NoWinner
```

Но мы не можем так поступить, потому что `hasWinner` – это призрачная переменная. Вместо этого мы просматриваем последовательность еще раз, чтобы убедиться, действительно ли этот кандидат набрал строгое большинство голосов. Если это так, то `hasWinner` имеет значение **`true`**. А поскольку `SearchForWinner` гарантированно вернет кандидата со строгим большинством голосов, если таковой имеется, то: обнаружив, что `c` не является победителем, очевидно, что `hasWinner` должно иметь значение **`false`**. Наконец (барабанная дробь), наш полезный метод завершается так:

```
    return if HasMajority(a, 0, |a|, c) then
        Winner(c)
      else
        NoWinner;
  }
```

13.8. Итоги

В этой главе я показал множество примеров использования массивов и последовательностей. Все представленные здесь алгоритмы выполняют поиск по заданным данным, и их инварианты часто имеют форму «мы знаем, что ключа здесь нет: ...» или «мы знаем, что ключ здесь есть: ...». Еще одна общая черта этих программ: наличие в спецификациях кванторов, использующих индексы массивов.

Конечные реализации программ обманчиво просты. Мы не смогли бы убедиться в корректности многих из них без дополнительных доказательств. Верификатор проверил наши доказательства и убедился, что мы не упустили ни одного случая.

При разработке некоторых из этих программ, включая, казалось бы, тривиальный линейный поиск, мы потратили значительное время на обдумывание их спецификаций. Точность спецификации позволяет принимать решения в крайних случаях, например как и когда сигнализировать о сбое и следует ли указывать, какое вхождение ключа искать. После определения реализации верификатор проверяет ее соответствие спецификации.

В расширенном разделе 13.7 я продемонстрировал применение призрачных параметров и переменных. Призрачный параметр `K` в спецификации `MajorityVote` дал нам имя для представления неизвестного кандидата-победителя. Поскольку победитель, получивший строгое большинство голосов, уникален (как следует из леммы `DistinctCounts`), мы смогли сформулировать постусловие в довольно простом виде `k == K`. А в методе `DefinerElection` мы использовали призрачные конструкции для определения значений призрачных параметров `SearchForWinner`. Призрачные конструкции помогли нам в разработке программы, но окончательная ее версия компилируется и запускается без них.

Упражнение 13.17

Группа смежных элементов последовательности с равными значениями называется *плато*.

а. Определите призрачный предикат `Plateau(s, lo, hi)`, возвращающий **`true`**, если `s[lo..hi]` – это плато.

б. Определите спецификацию метода, который принимает последовательность `s` целых чисел и возвращает длину самого длинного плато в ней.

в. Реализуйте этот метод.

Упражнение 13.18

Получив три неубывающие целочисленные последовательности с общим элементом, верните этот общий элемент.

Примечания

Ошибка реализации метода двоичного поиска в библиотеке Java попала в заголовки газет, когда она была обнаружена лишь спустя несколько лет после ее появления. Проблема заключалась в арифметическом переполнении в инструкции

```
mid := (lo + hi) / 2;
```

если переменные имеют тип машинного целого числа, размер которого ограничен. Эта проблема отсутствует в разделе 13.2.2, поскольку тип **`int`** в Dafny не ограничен. Если в вашем языке используются ограниченные целые числа и верификатор проверяет арифметическое переполнение, то вы получите сообщение об ошибке в выражении справа от оператора присваивания. Если ваш язык использует целые числа со знаком и модульную арифметику (т. е. семантику *циклического переноса – wraparound*), то верификатор сообщит о возможности появления отрицательных индексов в выражении `a[mid]`. Язык Dafny включает объявления, позволяющие определять ограниченные целые числа, но я не использую их в этой книге. Если бы они использовались в разделе 13.2.2, то верификатор действительно сообщил бы о возможности арифметического переполнения.

Упражнение 13.19

Как можно изменить правую часть в инструкции присваивания, чтобы ни одно из подвыражений не вызывало переполнения.

Реализация поиска точки на наклонной местности в разделе 13.5 является линейной по сторонам двумерной сетки. Мы могли бы реализовать более эффективный метод, обобщив двоичный поиск, как показал Ричард Берд

(Richard Bird) [17]. Вслед за Грайсом [55] Берд назвал алгоритм Saddleback Search (поиск седловины).

Некоторые алгоритмы, представленные в разделах и упражнениях этой главы, используются в нескольких классических учебниках по доказательству корректности программ. Среди них – «The Science of Programming» Дэвида Гриса (David Gries), в котором дополнительно прослеживаются истоки таких алгоритмов [55].

Инструкция «присвоить такое значение, что» была введена Ральфом Бэком (Ralph Back) под названием *недетерминированное присваивание* (nondeterministic assignment) [10, 6]. В сочетании с предусловием она широко использовалось Кэрроллом Морганом (Carroll Morgan) в качестве *инструкции языка спецификаций* [93].

Глава 14

Изменение массивов

Одна из общих черт программ, которые мы исследовали в предыдущей главе, – они не меняют содержимое массивов. Но скоро мы изменим этому правилу.

Как вы помните, доступ к хранилищу, размещенному в куче, осуществляется через ссылки, поэтому само хранилище нельзя передать функциям и методам в виде аргумента. Однако спецификации должны иметь возможность идентифицировать соответствующие части кучи. Для этой цели используются *фреймы*.

В этой главе мы будем использовать очень простые фреймы, настолько простые, что я отложу их общее описание до главы 16. В разделе 14.0 я расскажу, что нужно знать о фреймах, используемых в этой и следующей главах. В разделе 14.1 мы попрактикуемся в написании простых программ, изменяющих массивы.

14.0. Простые фреймы

В этом разделе дается краткое руководство по работе с фреймами в простых программах, использующих массивы.

14.0.0. Инструкция modifies

Вернемся к методу `Min` из раздела 13.3.0. Задача, решаемая методом `Min`, – найти в массиве элемент с минимальным значением и вернуть его. Но что, если реализация метода *изменит* значение каждого элемента массива, скажем увеличит на `1330`, а затем вернет число `1330`? Такая реализация выполнит условие, указанное в инструкции **`ensures`**. То есть к моменту завершения метода минимальное значение в массиве действительно будет равно `1330`! 😳 Но допускает ли наша спецификация такую реализацию?

К счастью, нет. Спецификация метода описывает, что методу разрешено изменять, а что он должен оставить неизменным. Это выражается в форме *фрейма записи* метода, который указывается с помощью инструкции **modifies**. Поскольку метод `Min` не имеет инструкции **modifies** в спецификации (т. е. имеет пустую инструкцию **modifies**), ему запрещено вносить изменения в данный массив.

Использование фреймов регулируется набором собственных правил. Вот первое такое правило:

> Если метод должен изменять элементы массива, переданного в параметре `a`, то его спецификация должна включать инструкцию **modifies** `a`.

Например:

```
method SetEndPoints(a: array<int>, left: int, right: int)
  requires a.Length != 0
  modifies a
{
  a[0] := left;
  a[a.Length - 1] := right;
}
```

В отсутствие инструкции **modifies** тело этого метода не будет соответствовать своему контракту, и верификатор сообщит, что метод изменяет содержимое массива, который не указан в инструкции **modifies**.

Чтобы массив, на который ссылается `a`, можно было изменить, спецификация должна включать инструкцию **modifies** `a`. Например, реализация следующего метода соответствует его спецификации:

```
method Aliases(a: array<int>, b: array<int>)
  requires 100 <= a.Length
  modifies a
{
  a[0] := 10;
  var c := a;
  if b == a {
    b[10] := b[0] + 1;
  }
  c[20] := a[14] + 2;
}
```

потому что к моменту обновления `b[10]` и `c[20]` известно, что они ссылаются на элементы массива `a`. Однако без проверки **if** `b == a` верификатор обнаружил бы ошибку в инструкции присваивания `b[10]`.

Упражнение 14.0

Попробуйте добавить постусловие

```
ensures a[0] == left && a[a.Length - 1] == right
```

в метод `SetEndPoints`. Исправьте предусловие метода (внеся как можно меньше изменений), чтобы метод благополучно проверялся верификатором.

Упражнение 14.1

Добавьте инструкцию **`modifies`** `a` в спецификацию метода `Min` из раздела 13.3.0, а затем реализуйте метод в соответствии с решением «вернуть 1330!», описанным выше.

14.0.1. Старые значения

Представьте, что мы решили написать постусловие для метода, который увеличивает элементы заданного массива. В таком случае мы должны указать, что на выходе из метода значения элементов больше, чем на входе. Другими словами, постусловие должно ссылаться как на начальное, так и на конечное состояние метода. По этой причине постусловие метода является *предикатом с двумя состояниями*.

Выражение `E` в инструкции **`ensures`** обычно относится к значению `E` в конечном состоянии метода. Чтобы обратиться к значению `E` в начальном состоянии, нужно использовать инструкцию **`old`**`(E)`. Например, инструкция **`modifies`** в следующем методе гласит, что элементы массива могут изменяться, но постусловие ограничивает эти изменения увеличением `a[4]`, возможным уменьшением `a[6]` и требует, чтобы значение `a[8]` оставалось неизменным:

```
method UpdateElements(a: array<int>)
  requires a.Length == 10
  modifies a
  ensures old(a[4]) < a[4]
  ensures a[6] <= old(a[6])
  ensures a[8] == old(a[8])
{
  a[4], a[8] := a[4] + 3, a[8] + 1;
  a[7], a[8] := 516, a[8] - 1;
}
```

Если говорить точнее, то **`old`** влияет только на разыменование кучи в своем аргументе. Например, в следующем методе

```
method OldVsParameters(a: array<int>, i: int) returns (y: int)
  requires 0 <= i < a.Length
```

```
  modifies a
  ensures old(a[i] + y) == 25
```

инструкция **old**() не влияет на переменные a, i и y. То есть входные параметры a и i, на которые ссылается эта инструкция, обозначают свои начальные значения, а выходной параметр y обозначает свое конечное значение, как если бы **old** не использовалась; интерпретируется только разыменование кучи, обозначенное квадратными скобками.

Распространенной ошибкой является использование **old** с непреднамеренным аргументом, как иллюстрирует постусловие следующего метода:

```
method Increment(a: array<int>, i: int)
  requires 0 <= i < a.Length
  modifies a
  ensures a[i] == old(a)[i] + 1 // распространенная ошибка
{
  a[i] := a[i] + 1; // ошибка: нарушение постусловия
}
```

Поскольку в аргументе **old** нет разыменования кучи, выражение **old**(a) будет означать то же самое, что и просто a.

Упражнение 14.2

В спецификации Increment выше переместите круглые скобки в инструкции **old** так, чтобы тело метода соответствовало постусловию.

Упражнение 14.3

Напишите реализацию OldVsParameters, удовлетворяющую спецификации.

Кстати, только инструкция **ensures** в спецификации метода является предикатом с двумя состояниями. Инструкции **requires**, **modifies** и **decreases** всегда интерпретируются с позиции начального состояния метода.

14.0.2. Новые массивы

Цель инструкции **modifies** в методе M – разъяснить, какие части состояния вызывающего кода можно изменить с помощью M. Это не относится к массивам, созданным в M, потому что вызывающий код не имеет доступа к таким массивам перед вызовом M.

> Метод может создать новый массив и изменять его элементы без упоминания этого массива в инструкции **modifies**.

Например, метод

```
method NewArray() returns (a: array<int>)
  ensures a.Length == 20
```

```
{
  a := new int[20];
  var b := new int[30];
  a[6] := 216;
  b[7] := 343;
}
```

будет считаться корректным даже без инструкции **modifies**.

14.0.3. Свежие массивы

Рассмотрим метод, вызывающий NewArray():

```
method Caller() {
  var a := NewArray();
  a[8] := 512; // ошибка: изменение элементов a запрещено
}
```

Чтобы Caller мог изменять элементы a, он должен иметь инструкцию **modifies** a или доказать, что a был создан после входа в Caller. Последнее – как раз наш случай. Для доказательства мы должны усилить спецификацию NewArray и гарантировать, что массив, возвращаемый методом NewArray, будет создан внутри NewArray. После этого становится очевидно, что массив выделяется от имени Caller.

Такая спецификация записывается с использованием предиката **fresh**:

```
method NewArray() returns (a: array<int>)
  ensures fresh(a) && a.Length == 20
```

После добавления этой спецификации в NewArray метод Caller благополучно проходит верификацию.

14.0.4. Инструкция reads

Функция не может ничего изменять, поэтому у нее нет фрейма записи. Однако у функции есть *фрейм чтения*, который определяет зависимости функции от кучи и задается с помощью инструкции **reads**. Эта информация о зависимостях используется для выяснения влияния различных изменений в куче на значение функции (что особенно важно, если тело функции недоступно или если функция является рекурсивной).

Если функция обращается к элементам массива a, ее спецификация должна включать инструкцию **reads** a.

Следующая функция зависит от значений входного массива a, поэтому ее спецификация должна включать **reads** a:

```
predicate IsZeroArray(a: array<int>, lo: int, hi: int)
  requires 0 <= lo <= hi <= a.Length
  reads a
```

```
  decreases hi - lo
{
  lo == hi || (a[lo] == 0 && IsZeroArray(a, lo + 1, hi))
}
```

Обратите внимание, что инструкции **reads** сообщают, от каких частей изменяемой кучи зависит функция. То есть элементы последовательности и свойства значения некоторого типа данных могут быть использованы функцией без инструкции **reads** (что объясняет, почему мы не видели никаких инструкций **reads** в частях 0 и 1 книги). Например, версия `IsZeroArray` для последовательности не требует инструкции **reads**:

```
predicate IsZeroSeq(a: seq<int>, lo: int, hi: int)
  requires 0 <= lo <= hi <= |a|
  decreases hi - lo
{
  lo == hi || (a[lo] == 0 && IsZeroSeq(a, lo + 1, hi))
}
```

14.1. Простые изменения массивов

Рассмотрим несколько наглядных примеров простых изменений массивов.

14.1.0. Инициализация массива

Напишем метод, присваивающий всем элементам массива заданное значение. Вот спецификация метода:

```
method InitArray<T>(a: array<T>, d: T)
  modifies a
  ensures forall i :: 0 <= i < a.Length ==> a[i] == d
```

Для реализации этого метода определим цикл, инвариант которого получен путем замены константы `a.Length` в постусловии на индекс цикла:

```
{
  var n := 0;
  while n != a.Length
    invariant 0 <= n <= a.Length
    invariant forall i :: 0 <= i < n ==> a[i] == d
}
```

Мы давно не практиковались в обратном проходе от инварианта цикла через обновление его индекса `n := n + 1` (см. главы 2, 11 и 12). Давайте сделаем это для нашей простой программы:

```
{{ (forall i :: 0 <= i < n ==> a[i] == d) && a[n] == d }}
{{ forall i :: 0 <= i < n + 1 ==> a[i] == d }}
```

```
n := n + 1
{{ forall i :: 0 <= i < n ==> a[i] == d }}
```

Этот расчет сразу говорит нам, что значение `d` должно присваиваться элементу `a[n]` перед увеличением `n`. Итак, вот тело цикла, завершающее реализацию нашего метода:

```
  {
    a[n] := d;
    n := n + 1;
  }
```

14.1.1. Инициализация матрицы

Повторим метод `InitArray`, но на этот раз для матрицы:

```
method InitMatrix<T>(a: array2<T>, d: T)
  modifies a
  ensures forall i, j ::
    0 <= i < a.Length0 && 0 <= j < a.Length1 ==> a[i,j] == d
```

Постусловие упоминает оба измерения `a`. Это говорит о том, что цикл должен иметь два индекса. В предыдущей главе мы видели много подобных программ, но там нам посчастливилось обойтись единственным циклом, который каким-то образом перебирал индексы. Здесь у нас нет возможности сократить вычисления – мы должны просмотреть каждую из многих пар индексов. Это предполагает использование двух циклов, вложенных друг в друга.

В каждой итерации внешнего цикла мы будем инициализировать целую строку. Другими словами, после `m` итераций будет инициализировано `m` строк. Это отражено в инварианте цикла, который мы получаем, заменяя константу `a.Length0` в постусловии индексом цикла `m` (метод конструирования циклов 12.0):

```
{
  var m := 0;
  while m != a.Length0
    invariant 0 <= m <= a.Length0
    invariant forall i, j ::
      0 <= i < m && 0 <= j < a.Length1 ==> a[i,j] == d
}
```

Давайте также совершим обратный проход от инварианта цикла через обновление его индекса:

```
{{ (forall i, j :: 0 <= i < m && 0 <= j < a.Length1 ==> a[i,j] == d) &&
   (forall j :: 0 <= j < a.Length1 ==> a[m,j] == d) }}
//   применить правило одной точки ко второму квантору
```

```
{{ (forall i, j :: 0 <= i < m && 0 <= j < a.Length1 ==> a[i,j] == d) &&
   (forall i, j :: i == m && 0 <= j < a.Length1 ==> a[i,j] == d) }}
//   применить правило разделения диапазона
{{ forall i, j :: 0 <= i < m + 1 && 0 <= j < a.Length1 ==> a[i,j] == d }}
m := m + 1
{{ forall i, j :: 0 <= i < m && 0 <= j < a.Length1 ==> a[i,j] == d }}
```

Верхняя аннотация содержит два квантора, которые становятся постусловием внутреннего цикла. Легко забыть, что внутренний цикл должен установить *оба* этих квантора, а не только квантор `forall j`. Если в инварианте внутреннего цикла мы упомянем только строку `m`, то такая спецификация разрешит изменять любые другие строки.

Фреймы цикла, связанные с кучей

Основная причина только что упомянутой проблемы – это фреймы циклов, т. е. (обычно неявная) часть спецификации, определяющая, что разрешено изменять в теле цикла. Ранее я упоминал фреймы циклов в контексте локальных переменных (разделы 11.0.5 и 11.1.3), но здесь нам также необходимо учитывать, что спецификация цикла гласит о памяти, выделенной в куче, например об элементах массивов.

В спецификациях циклов, как и в спецификациях методов, тоже можно использовать инструкцию `modifies` и с ее помощью определять, какие хранилища, размещенные в куче, можно изменять. Однако инструкции `modifies` в циклах редко отличаются от инструкций `modifies` в объемлющих методах (или циклах, если говорить о вложенных циклах). Следовательно, если в цикле отсутствует явная инструкция `modifies`, то по умолчанию цикл получает инструкцию `modifies` объемлющего метода (или цикла).

Внутренний цикл

Вернемся к методу инициализации матрицы. Рассуждения выше привели нас к выводу, что внутренний цикл должен обеспечить выполнение двух кванторов в верхней аннотации. Первый квантор (`forall i, j`) уже выполняется при входе во внутренний цикл, поэтому внутреннему циклу остается только поддерживать выполнение этого условия. Мы применим к нему метод конструирования циклов 13.1 (*всегда*). Для поддержки выполнения второго квантора (`forall j`) мы заменим константу `a.Length1` индексом цикла `n`. В результате этого получаем:

```
{
  var n := 0;
  while n != a.Length1
    invariant 0 <= n <= a.Length1
    invariant forall i, j ::
      0 <= i < m && 0 <= j < a.Length1 ==> a[i,j] == d
    invariant forall j :: 0 <= j < n ==> a[m,j] == d
```

Оставшуюся часть реализации метода заполнить несложно:

```
    {
      a[m,n] := d;
      n := n + 1;
    }
    m := m + 1;
  }
```

14.1.2. Увеличение значений в массиве

Я покажу еще два примера того же уровня сложности, чтобы помочь вам закрепить эту базовую технику. Метод в следующем примере увеличивает каждый элемент заданного массива на 1. Вот его спецификация:

```
method IncrementArray(a: array<int>)
  modifies a
  ensures forall i :: 0 <= i < a.Length ==> a[i] == old(a[i]) + 1
```

Обратите внимание, что выражение `old(a[i])` обозначает `i`-й элемент массива на входе в метод. Как говорилось в разделе 14.0.1, выражение вида `old(a)[old(i)]` обозначает другое значение – поскольку выражение `old` не влияет на параметры и связанные переменные, оно просто равно `a[i]`. Именно квадратные скобки разыменовывают кучу, поэтому именно к квадратным скобкам должно применяться выражение `old`, что мы и делаем, записав `old(a[i])`.

Если попробовать заменить константу `a.Length` индексом цикла `n`, то получится следующий цикл:

```
{
  var n := 0;
  while n != a.Length
    invariant 0 <= n <= a.Length
    invariant forall i :: 0 <= i < n ==> a[i] == old(a[i]) + 1
  {
    a[n] := a[n] + 1;
    n := n + 1;
  } // ошибка: второй инвариант цикла не выполняется его телом
}
```

Однако в этом случае мы не можем обеспечить выполнение инварианта. Тело цикла увеличивает значение `a[n]` на единицу, и чтобы соблюсти инвариант, нам нужно знать:

```
a[n] + 1 == old(a[n]) + 1
```

Фрейм цикла совпадает с фреймом объемлющего метода, поэтому элементы `a` могут принимать в цикле любые значения, разрешенные инвариантом. Инвариант выше говорит только об элементах массива `a[..n]`, поэ-

тому нет никакой информации о связи между текущим значением a[n] и значением a[n] на входе в цикл.

Отладка верификации

Чтобы понять, в чем проблема, вспомните, что нужно смотреть только на спецификацию цикла и не поддаваться искушению прийти к другим заключениям о состояниях, которых может достичь тело цикла (глава 11). Если вы все еще не понимаете, в чем проблема, то попробуйте устранить ее, как описывается далее.

Для начала добавьте инструкцию **assert** с нарушаемым инвариантом в то место, где он должен выполняться. Теперь тело цикла будет выглядеть так:

```
a[n] := a[n] + 1;
n := n + 1;
assert forall i :: 0 <= i < n ==> a[i] == old(a[i]) + 1; // ошибка
```

В результате сообщение об ошибке должно было исчезнуть в инварианте и появиться в утверждении **assert**. Если бы так произошло, то мы могли бы доказать условие здесь, и с инвариантом было бы все в порядке.

Теперь переместите или, если хотите, вставьте эту инструкцию **assert** перед присваиванием переменной n и при этом замените n на n + 1 в утверждении, как мы делали это при вычислении слабейшего предусловия. В результате должно получиться:

```
a[n] := a[n] + 1;
assert forall i :: 0 <= i < n + 1 ==> a[i] == old(a[i]) + 1; // ошибка
n := n + 1;
```

Это утверждение **assert** перед присваиванием переменной n совпадает с утверждением, которое минуту назад стояло после присваивания, а значит, теперь верификатор должен жаловаться именно на это утверждение и больше ни на что. (Если у вас получится иначе, то вам нужно сосредоточить усилия по отладке, чтобы выяснить, почему ваш результат не совпадает с моим.)

Далее давайте вручную разделим этот квантор. Мы имеем диапазон 0 <= i < n + 1, поэтому преобразуем его в дизъюнкцию 0 <= i < n и i == n. Ко второму дизъюнкту применим правило элиминации квантора всеобщности по одноэлементному множеству (см. раздел 13.1.5 или приложение B, чтобы вспомнить эти правила кванторов). Представив результат в виде двух дополнительных инструкций **assert**, имеем:

```
a[n] := a[n] + 1;
assert forall i :: 0 <= i < n ==> a[i] == old(a[i]) + 1;
assert a[n] == old(a[n]) + 1; // ошибка
assert forall i :: 0 <= i < n + 1 ==> a[i] == old(a[i]) + 1;
n := n + 1;
```

Теперь верификатор сообщает, что не может доказать среднее утверждение. Поскольку предшествующая инструкция только что присвоила элементу `a[n]` новое значение, можно ожидать, что `a[n]` в нарушенном утверждении будет иметь то же значение, что имела правая часть инструкции (т. е. `a[n] + 1`) перед фактическим присваиванием. Давайте добавим утверждение **`assert`** для проверки этого условия:

```
assert a[n] + 1 == old(a[n]) + 1; // ошибка
a[n] := a[n] + 1;
assert forall i :: 0 <= i < n ==> a[i] == old(a[i]) + 1;
assert a[n] == old(a[n]) + 1;
assert forall i :: 0 <= i < n + 1 ==> a[i] == old(a[i]) + 1;
n := n + 1;
```

На этот раз верификатор жалуется на первое утверждение и ни на какое другое. Это говорит о том, что если мы сможем доказать соблюдение этого условия в точке, где находится утверждение **`assert`**, то верификация выполнится благополучно.

Я думаю, что ошибка в утверждении обусловлена недостатком информации, в частности мы должны показать верификатору, что в начале цикла выполняется условие `a[n] == old(a[n])`. Немного поразмышляв о цикле в целом, легко заметить, что элементы массива с индексом `n` и выше не были изменены циклом. Чтобы сообщить об этом факте верификатору, включим его в инвариант:

```
invariant forall i :: n <= i < a.Length ==> a[i] == old(a[i])
```

После этого все жалобы верификатора исчезнут. Новый инвариант выполняется в начале цикла, все утверждения соответствуют ему, и теперь можно доказать, что все инварианты выполняются в конце тела цикла. Поскольку инструкции **`assert`** были добавлены, только чтобы помочь диагностировать ситуацию, теперь, когда они выполнили свою задачу, их можно удалить.

Упражнение 14.4

Определите спецификацию и реализацию метода, увеличивающего на 1 каждый элемент матрицы.

14.1.3. Копирование массива

Скопируем элементы одного массива в другой:

```
method CopyArray(src: array, dst: array)
  requires src.Length == dst.Length
  modifies dst
  ensures forall i :: 0 <= i < src.Length ==> dst[i] == old(src[i])
```

Этот метод принимает два массива, один из которых указан в инструкции **modifies**, дающей методу разрешение изменять элементы dst. Точнее говоря, методу разрешено изменять элементы массива, на которые ссылается dst. Если src ссылается на тот же массив, что и dst, т. е. если src == dst, тогда методу будет разрешено изменять элементы массива с помощью ссылки src. Как я уже отмечал ранее (в разделе 13.0.2), инструкция **modifies** перечисляет массивы, которые разрешено изменять, но не заботится о том, какие выражения используются для ссылки на эти массивы.

Если src и dst ссылаются на один и тот же массив, то постусловие типа

```
ensures forall i :: 0 <= i < src.Length ==> dst[i] == src[i]
```

будет выполняться, даже если метод полностью изменит содержимое массива, потому что это постусловие просто гласит, что значение каждого элемента обрабатываемого массива равно самому себе. Вот почему мы пишем спецификацию **old**(src[i]). (В качестве альтернативы можно было бы добавить предусловие src != dst.)

Определившись со спецификацией метода, мы легко можем написать спецификацию и тело цикла, следуя шаблонам, которые видели раньше.

Обратите внимание, что, так же как и в ситуации, описанной в разделе 14.1.2, мы должны добавить инвариант цикла, гласящий, что элементы src не изменяются, чтобы справиться с ситуацией, когда src == dst:

```
{
  var n := 0;
  while n != src.Length
    invariant 0 <= n <= src.Length
    invariant forall i :: 0 <= i < n ==> dst[i] == old(src[i])
    invariant forall i :: 0 <= i < src.Length ==> src[i] == old(src[i])
  {
    dst[n] := src[n];
    n := n + 1;
  }
}
```

Потренируйтесь в определении инструкций **modifies** для циклов, выполнив следующие упражнения.

Упражнение 14.5

Добавьте src != dst как предусловие. Затем максимально упростите постусловие и инварианты (не меняя смысла постусловия).

Упражнение 14.6

Замените **old**(src[i]) на src[i] в постусловии метода CopyArray, как это гипотетически рассматривалось выше. Затем измените реализацию CopyArray так, чтобы он делал одинаковыми все элементы src, если src == dst.

Упражнение 14.7

Напишите спецификацию и реализацию метода, копирующего элементы из одной матрицы в другую того же размера. Разрешите использовать одну и ту же матрицу в роли источника и приемника и убедитесь, что в спецификации указано правильное значение для этого случая. (Напишите небольшую тестовую программу для проверки.)

Упражнение 14.8

Напишите спецификацию и реализацию метода

```
method DoubleArray(src: array<int>, dst: array<int>)
```

который для каждого `i` записывает в `dst[i]` значение `2 * src[i]`. Считайте, что заданные массивы имеют одинаковую длину, и допустите возможность, что они ссылаются на один и тот же массив в куче.

Упражнение 14.9

Напишите спецификацию и реализацию метода, меняющего местами элементы массива.

Упражнение 14.10

Напишите спецификацию и реализацию метода, который принимает и транспонирует квадратную матрицу.

Упражнение 14.11

Напишите спецификацию и реализацию метода, который выполняет поворот массива «влево». То есть то, что раньше хранилось в `a[(i + 1) % a.Length]`, сохраняется в `a[i]`.

Упражнение 14.12

Напишите спецификацию и реализацию метода, который выполняет поворот массива «вправо». То есть то, что раньше хранилось в `a[(i - 1) % a.Length]`, сохраняется в `a[i]`.

14.1.4. Операции с массивами без циклов

Циклы, подобные тем, что показаны выше в этом разделе, иллюстрируют приемы использования инвариантов циклов с кванторами и дают полезную практику для создания более сложных циклов. Кроме того, подобные циклы довольно утомительны – они концептуально просты, но требуют определять инварианты, которые выглядят сложнее, чем тела цикла, которые они описывают.

Поразмыслив над этим, вы заметите, что циклы не лучшая программная конструкция для выполнения простых и единообразных операций, и некоторые языки программирования могут предложить кое-что получше. Далее я покажу вам две такие конструкции, имеющиеся в Dafny.

Конструктор массива

При размещении массива в памяти можно задать функцию, определяющую начальные элементы, например:

```
a := new int[25](F);
```

где `F` – это функция от индексов нового массива (от `0` до `25` в этом примере), создающая желаемые значения для элементов нового массива (типа **`int`** в этом примере).

Чаще в роли `F` используется *встроенная функция*, известная как *лямбда-выражение*. Она имеет вид `x => E`, где `x` – формальный параметр, а `E` – выражение, определяющее тело функции. Например,

```
a := new int[25](i => i * i);
```

разместит в памяти массив длиной `25`, элементы которого инициализируются квадратами первых `25` чисел. Вот еще один часто встречающийся пример:

```
a := new int[n](_ => 0);
```

Эта инструкция создаст массив с длиной `n` и инициализирует нулем каждый его элемент. Как показано в примере, если формальный параметр встроенной функции не используется в теле встроенной функции, то его имя можно опустить и использовать символ-заполнитель _.

Для инициализации многомерных массивов следует использовать функции нескольких аргументов. Вот пример инициализации матрицы размером `50` на `50` – значением `d` диагональных элементов и значением `0` всех остальных:

```
m := new int[50, 50]((i, j) => if i == j then d else 0);
```

Обратите внимание, что этой встроенной функции параметры передаются в круглых скобках, – когда параметр только один, как в примере выше, круглые скобки можно опустить.

Встроенные функции со спецификациями

Чтобы разместить в памяти массив – копию другого массива, вы могли бы написать такой код:

```
b := new T[a.Length](i => a[i]); // ошибка доступа к a
```

Однако верификатор тут же сообщит об ошибке, потому что он проверяет корректность любых функций, в том числе и встроенных. В частности, чтобы определение `i => a[i]` считалось корректной функцией, нужно убедиться, что `i` находится в рамках определенного диапазона и доступ к `a` разрешен фреймом функции. Если бы это была именованная функция, то необходимость всего этого была бы очевидна (мы рассматривали предусловия функций на протяжении всей книги, а с фреймами чтения познакомились в разделе 14.0.4). Встроенные функции тоже могут иметь инструкции **`require`** и **`reads`**, и мы должны использовать их, чтобы реализовать этот пример. Вот корректная форма определения функции, которую принимает верификатор:

```
b := new T[a.Length](i requires 0 <= i < a.Length reads a =>
                     a[i]);
```

Код получился довольно длинным, но он избавляет нас от необходимости писать цикл и изобретать инвариант цикла.

Конструкторы последовательностей

Для создания последовательностей существует похожая конструкция. Выражение

```
seq(500, i => i % 2 == 0)
```

обозначает последовательность из 500 логических значений, четные элементы которой получают значение **`true`**, а нечетные – **`false`**.

Напомню, что последовательности – это значения, а массивы – ссылки на изменяемые элементы. Соответственно, конструктор последовательности – это выражение, а не инструкция размещения в памяти.

Агрегирующие инструкции

В Dafny есть *агрегирующая инструкция*, которую можно использовать для обновления элементов массива. Агрегирующую инструкцию можно рассматривать как обобщение инструкции одновременного присваивания. То есть если

```
a[5], a[7] := a[5] + 25, a[7] + 49;
```

изменяет два элемента массива, то агрегирующая инструкция

```
forall i | 0 <= i < a.Length {
  a[i] := a[i] + i * i;
}
```

похожим образом изменяет все элементы массива.

Несмотря на синтаксическое сходство с инструкциями **`for`** и **`foreach`** в таких языках, как Java или C#, инструкция **`forall`** *не является циклом*. Это обстоятельство имеет два следствия.

Во-первых, тело инструкции **forall** выполняется одновременно для всех значений связанной переменной. Такой подход помогает упростить формулировку некоторых вычислений. Например,

```
forall i | 0 <= i < a.Length && i % 2 == 0 {
  a[i] := a[i] + 1;
}
```

увеличивает элементы с четным индексом на 1, соответственно, метод IncrementArray из раздела 14.1.2 можно реализовать так:

```
forall i | 0 <= i < a.Length {
  a[i] := a[i] + 1;
}
И
forall i | 0 <= i < a.Length {
  a[i] := a[(i + 1) % a.Length];
}
```

поворачивает массив a влево (то же самое предлагалось сделать в упражнении 14.11 с помощью цикла). Однако некоторые операции, такие как суммирование, невозможно реализовать подобным образом, поскольку их нельзя применить сразу ко всем индексам.

Во-вторых, для рассуждений об инструкции **forall** не требуется инвариант цикла! Например, вы можете попрощаться с длинными и скучными инвариантами вложенных циклов, описанными в разделе 14.1.1, и вместо этого просто написать:

```
forall i, j | 0 <= i < a.Length0 && 0 <= j < a.Length1 {
  a[i,j] := d;
}
```

Чтобы компиляция прошла успешно, на инструкцию **forall** налагаются некоторые ограничения. Наиболее существенное: тело должно содержать только один оператор присваивания.

В будущем я буду использовать инструкцию **forall** везде, где это возможно, но мы увидим множество программ, где итерации действительно необходимы. И для таких программ вам пригодится умение писать инварианты цикла.

Упражнение 14.13

Выполните упражнение 14.7, используя инструкцию **forall**.

Упражнение 14.14

Выполните упражнение 14.8, используя инструкцию **forall**.

Упражнение 14.15

Выполните упражнение 14.9, используя инструкцию **`forall`**.

Упражнение 14.16

Выполните упражнение 14.10, используя инструкцию **`forall`**.

14.2. Итоги

В этой главе я представил *фреймы чтения и записи*, определяющие части кучи, от которых может зависеть функция и которые может изменять метод соответственно. Фреймы чтения объявляются с помощью инструкций **`reads`**, а фреймы записи – с помощью инструкций **`modifies`**, и обе они отмечают ссылки, ведущие в кучу.

Я также представил некоторые специальные конструкции для использования в спецификациях, когда изменяются элементы массива или поля объекта в куче. **`old`**`(E)` обозначает значение `E` в состоянии метода перед входом в него, **`fresh`**`(E)` говорит, что ссылка (или набор ссылок) `E` была выделена уже после входа в объемлющий метод.

Важно понимать, как определять спецификации и реализации циклов, которые разными способами изменяют элементы массивов. Я привел ряд примеров инициализации и простого изменения элементов массивов. Они наглядно показали, насколько важно указывать, что изменилось в куче, а что – нет.

Наконец, я продемонстрировал некоторые возможности языка, позволяющие вообще избавиться от циклов.

В следующей главе мы рассмотрим несколько программ, манипулирующих массивами более интересными способами.

Примечания

Разработка новаторского языка программирования Euclid [74] была нацелена на создание верифицируемых программ. Он содержал объявления пред- и постусловий, встроенные утверждения и инварианты модулей. Чтобы процедура могла получить доступ к переменной, объявленной вовне, процедура должна была явно *импортировать* эту переменную. С помощью такой инструкции импорта также можно указать, может ли процедура изменять переменную или только читать ее.

Глава 15

Сортировка на месте

В главе 8 мы познакомились со спецификациями и программами сортировки индуктивных списков. Поскольку списки – неизменяемые структуры данных, эти программы всегда возвращали отсортированные копии, не изменяя самих входных данных. В этой главе мы рассмотрим программы, переупорядочивающие элементы заданного массива, т. е. осуществляющие сортировку *in situ*, что на латыни означает «на месте».

15.0. Голландский национальный флаг

Для начала рассмотрим задачу сортировки массива с тремя значениями – тремя цветами: красным, белым и синим. Они будут сортироваться в порядке следования цветов на голландском национальном флаге (сначала красный, затем белый и, наконец, синий).

Начнем с представления цветов и способа их сравнения:

```
datatype Color = Red | White | Blue

ghost predicate Below(c: Color, d: Color) {
  c == Red || c == d || d == Blue
}
```

Поскольку мы предполагаем сортировать массив на месте, наш метод принимает только один параметр – ссылку на массив. Фрейм записи гласит, что метод может изменять содержимое массива. Как мы видели в раз-

деле 8.0, постусловие состоит из двух частей. Одна часть гласит, что на выходе элементы отсортированы, а другая – что мультимножество элементов в массиве не изменяется после исполнения метода. Используя обозначения, которые мы видели в разделе 13.4, запишем последнее условие с помощью преобразования элементов массива в последовательность и затем в мультимножество:

```
method DutchFlag(a: array<Color>)
  modifies a
  ensures forall i, j :: 0 <= i < j < a.Length ==> Below(a[i], a[j])
  ensures multiset(a[..]) == old(multiset(a[..]))
```

Идея алгоритма состоит в разделении входного массива на четыре смежных сегмента. Первый будет содержать только красные элементы, второй – только белые, а последний – только синие элементы. Оставшийся сегмент (третий, который я помещу между сегментами с белыми и синими элементами, хотя его так же можно поместить между сегментами с красными и белыми элементами) будет содержать элементы, которые в текущий момент еще не перенесены в соответствующий сегмент.

Для отслеживания четырех сегментов нам понадобятся три маркера. Они будут играть роль индексов циклов, и я назову их `r`, `w` и `b`. На следующей диаграмме показаны основные идеи, реализованные к данному моменту. Фактически диаграмма отражает инвариант цикла, который мы напишем.

Вот спецификация цикла:

```
{
  var r, w, b := 0, 0, a.Length;
  while w < b
    invariant 0 <= r <= w <= b <= a.Length
    invariant forall i :: 0 <= i < r ==> a[i] == Red
    invariant forall i :: r <= i < w ==> a[i] == White
    invariant forall i :: b <= i < a.Length ==> a[i] == Blue
    invariant multiset(a[..]) == old(multiset(a[..]))
}
```

Начальные присваивания переменным `r`, `w`, `b` очищают три цветных сегмента и помещают все элементы массива в неотсортированный сегмент. Условие цикла гласит, что цикл продолжается, пока в неотсортированном сегменте имеются элементы. Когда неотсортированный сегмент опустеет, инвариант и отрицание условия цикла будут подразумевать `w == b`, т. е. что

массив состоит исключительно из трех цветных сегментов, расположенных в правильном порядке. Последний инвариант использует метод конструирования циклов 13.1 (*всегда*).

Кстати, обратите внимание на последовательное использование полуоткрытых интервалов в диапазонах кванторов. Это следствие того, что `r,w,b` рассматриваются как разграничивающие сегменты и обычно так представляются диапазоны в информатике. Смежные диапазоны легко объединяются, когда верхняя граница одного диапазона является нижней границей следующего (см. также врезку 13.0). Действительно, когда `w == b`, то можно заметить, что три квантора учитывают все элементы массива.

Осталась одна задача: реализовать тело цикла. Каждая итерация сортирует очередной элемент, поэтому мы должны проверить один из элементов в неотсортированном сегменте. Очевидными кандидатами являются `a[w]` и `a[b-1]`, поскольку эти элементы примыкают к отсортированной области. Мы будем использовать `a[w]`, что дает нам следующую структуру тела цикла:

```
{
  match a[w]
  case Red => ?
  case White => ?
  case Blue => ?
}
```

Для замены трех знаков вопроса вам может пригодиться диаграмма, показанная выше.

Самое простое, когда `a[w]` – белый элемент. В этом случае нужно лишь включить его в сегмент с белыми элементами, что мы и делаем, увеличивая `w`:

```
  case White =>
    w := w + 1;
```

Если `a[w]` – красный элемент, то его нужно переместить в сегмент с красными элементами. Но там, где заканчивается красный, начинается белый сегмент, поэтому мы должны переместить первый белый элемент в конец белой области:

```
  case Red =>
    a[r], a[w] := a[w], a[r];
    r, w := r + 1, w + 1;
```

Если белый сегмент пустой, т. е. если `r == w`, то обмен значениями `a[r]` и `a[w]` ничего не переместит, и это нормально.

Наконец, если элемент `a[w]` имеет синий цвет, то нужно переместить его в `a[b-1]` и уменьшить `b`. Эта последовательность действий поместит синий элемент в синий сегмент и приведет к завершению. Однако, сделав только это, мы нарушим последний инвариант, потому что просто скопируем `a[w]`

и затрем элемент, ранее хранившийся в `a[b-1]`. Чтобы этого не произошло, нужно поменять местами `a[w]` и `a[b-1]`:

```
case Blue =>
  a[w], a[b-1] := a[b-1], a[w];
  b := b - 1;
```

На этом программа завершена.

Упражнение 15.0

Измените инвариант, чтобы поместить неотсортированный сегмент между красным и белым сегментами. Затем повторно реализуйте инициализацию и тело цикла.

Упражнение 15.1

Напишите спецификацию метода, сортирующего массив логических значений, при этом значение **`false`** должно считаться меньше, чем **`true`**. Реализуйте метод, используя цикл, который меняет местами элементы массива (подобно цветам национального флага Голландии).

Упражнение 15.2

Как и многие другие алгоритмы сортировки, метод `DutchFlag`, описанный выше, является упрощением того, что на самом деле происходит. Я представил каждый элемент как цвет. Чаще всего элементы массива представляют собой более сложные фрагменты данных, а цвет – всего лишь ключ, по которому сортируется массив. Определите предикат `CBelow`, сравнивающий два значения на основе их цвета:

```
ghost predicate CBelow<T>(color: T -> Color, x: T, y: T)
```

и реализуйте метод, сортирующий массив по цвету его элементов:

```
method DutchFlagKey<T>(a: array<T>, color: T -> Color)
  modifies a
  ensures forall i, j :: 0 <= i < j < a.Length ==>
    CBelow(color, a[i], a[j])
  ensures multiset(a[..]) == old(multiset(a[..]))
```

Упражнение 15.3

(Усложненное) Алгоритмы сортировки обычно используют операцию перестановки для изменения порядка элементов данных. Но сортировку массива из трех цветов (без каких-либо других полезных данных) можно выполнить и без перестановки: сначала можно просканировать содержимое массива и подсчитать количество вхожде-

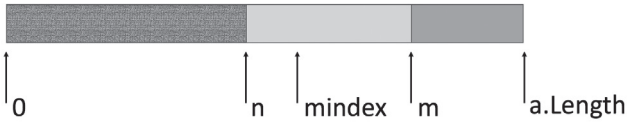
ний каждого цвета, а затем сформировать результат, соответствующий этим трем счетчикам, записав соответствующее количество красных, белых и синих цветов соответственно. Попробуйте реализовать и верифицировать метод, действующий таким способом.

15.1. Сортировка выбором

Сортировка выбором – это алгоритм сортировки, снова и снова отыскивающий минимальный среди неотсортированных элементов, а затем добавляющий его в сегмент с отсортированными элементами. Его спецификация подобна спецификации `DutchFlag` из предыдущего раздела, но здесь я использую целые числа вместо цветов:

```
method SelectionSort(a: array<int>)
  modifies a
  ensures forall i, j :: 0 <= i < j < a.Length ==> a[i] <= a[j]
  ensures multiset(a[..]) == old(multiset(a[..]))
```

У нас будет два вложенных цикла. Если `n` – индекс цикла, то инвариантом внешнего цикла является условие упорядоченности первых `n` элементов массива, как показывает следующая диаграмма:



Внутренний цикл сканирует оставшуюся часть массива в поисках минимума. Как мы видели ранее, минимум можно представить как находящийся ниже всех проверенных элементов и как один из элементов, подлежащих проверке. Мы должны отслеживать, где в массиве находится минимум, и для этой цели используем переменную `mindex`. Если гарантировать, что `mindex` является индексом в неотсортированной части массива, то из этого следует, что `a[mindex]` является одним из элементов, подлежащих проверке. Эта ситуация показана на диаграмме выше, где `m` – индекс внутреннего цикла.

Теперь, обозначив главную идею, перейдем к реализации метода.

Используем индекс цикла `n` и превратим первое постусловие в инвариант, заменив константу `a.Length` переменной `n`, а другое постусловие превратим в инвариант «всегда» (метод конструирования циклов 13.1):

```
{
  var n := 0;
  while n != a.Length
    invariant 0 <= n <= a.Length
    invariant forall i, j :: 0 <= i < j < n ==> a[i] <= a[j]
```

```
    invariant multiset(a[..]) == old(multiset(a[..]))
}
```

Теперь перейдем к внутреннему циклу. Мы уже написали несколько подобных циклов (включая программу поиска минимума в разделе 13.3), поэтому можем сразу же написать:

```
var mindex, m := n, n;
while m != a.Length
  invariant n <= m <= a.Length && n <= mindex < a.Length
  invariant forall i :: n <= i < m ==> a[mindex] <= a[i]
{
  if a[m] < a[mindex] {
    mindex := m;
  }
  m := m + 1;
}
```

и даже немного улучшить реализацию: если гарантировать, что `mindex` всегда будет меньше `m`, то мы можем записать первый инвариант так:

```
  invariant n <= mindex < m <= a.Length
```

Однако верификатор сообщит, что условие `mindex < m` не выполняется на входе в цикл. Но мы можем это исправить, присвоив `m` начальное значение `n + 1`, которое, как известно, не превышает `a.Length`:

```
var mindex, m := n, n + 1;
```

И действительно, нет смысла повторять внутренний цикл для `m == n`, поскольку `mindex` инициализируется значением `n`, которое, очевидно, является индексом минимального элемента в `a[n..n+1]`.

Отлично. После завершения внутреннего цикла `mindex` будет содержать индекс следующего минимального элемента, поэтому нам остается только поменять местами `a[n]` и `a[mindex]`:

```
a[n], a[mindex] := a[mindex], a[n];
n := n + 1;
```

Однако тут что-то не так! Верификатор сообщает, что цикл не выполняет инвариант цикла с квантором всеобщности. На входе в тело цикла этот инвариант сообщает нам, что отсортированы первые `n` элементов, а теперь мы говорим, что отсортированы первые `n + 1` элементов.

Одним из возможных виновников является внутренний цикл. Ранее (при инициализации матрицы в разделе 14.1.1) мы видели, что внутренний цикл иногда должен включать в себя некоторые инварианты внешнего цикла. Но это не относится к нашему случаю, потому что здесь внутренний цикл не изменяет ничего в массиве или в куче; верификатор определяет

это синтаксически, и поэтому ему автоматически становится известно, что отсортированный сегмент массива остается отсортированным.

Упражнение 15.4

Альтернативный способ реализации внутреннего цикла – заменить любой новый минимум в `a[n]`, как только он будет обнаружен. Это означает, что внутренний цикл будет изменять массив, поэтому ему потребуются некоторые инварианты из внешнего цикла. Исправив ситуацию с текущим алгоритмом, мы попробуем изменить внутренний цикл, как я только что описал, и верифицируем его. (Сразу отмечу, что эта альтернативная программа хуже, но как упражнение она позволит вам попрактиковаться в работе с инвариантами.)

И что дальше? Мы можем проверить, считает ли верификатор первые `n` элементов отсортированными, добавив следующее утверждение **`assert`** сразу после перестановки и *перед* увеличением `n`:

```
assert forall i, j :: 0 <= i < j < n ==> a[i] <= a[j];
```

Верификатор подтверждает корректность этого утверждения. Это может означать только одно: верификатор не уверен, что новое значение `a[n]` упорядочено в `a[..n]`.

При работаете над доказательствами корректности программ подобные ситуации возникают постоянно. Мы записываем предполагаемые решения в виде инвариантов, и они определяют реализацию программы. Какая-то часть дизайна всегда остается в нашей голове. Либо мы правильно продумали ситуацию, но не додумались записать условия в виде инвариантов, либо упустили какую-то часть конструкции, которая может быть или не быть корректной. В любом случае единственный способ привести убедительный аргумент в пользу корректности – сформулировать его как инвариант.

В нашем случае не хватает свойства, показывающего, что элементы отсортированного сегмента массива не просто отсортированы между собой, но и находятся в конечном положении. Другими словами, все отсортированные элементы находятся ниже неотсортированных элементов. Это важнейшее свойство алгоритма сортировки выбором. Запишем его как инвариант внешнего цикла:

```
invariant forall i, j :: 0 <= i < n <= j < a.Length ==> a[i] <= a[j]
```

Это свойство пригодится нам и в других алгоритмах сортировки, поэтому определим его как предикат:

```
ghost predicate SplitPoint(a: array<int>, n: int)
  requires 0 <= n <= a.Length
  reads a
{
```

```
    forall i, j :: 0 <= i < n <= j < a.Length ==> a[i] <= a[j]
  }
```

Если вы решите ввести этот предикат, то сформулируйте последний инвариант внешнего цикла как:

```
    invariant SplitPoint(a, n)
```

С помощью этого инварианта, заданного как квантор или предикат `SplitPoint`, мы успешно верифицировали `SelectionSort`.

15.2. Быстрая сортировка

Последняя процедура сортировки, которую мы рассмотрим в этой книге, – быстрая сортировка. Она замечательно работает на практике, но верифицировать ее намного сложнее, чем другие процедуры сортировки, представленные выше. Сложность сводится к тому, чтобы правильно указать, какие части массива изменяются и какие условия продолжают выполняться, несмотря на изменения.

15.2.0. Спецификация метода

Сначала все выглядит просто: спецификация `Quicksort` выглядит так же, как спецификация `SelectionSort`:

```
method Quicksort(a: array<int>)
  modifies a
  ensures forall i, j :: 0 <= i < j < a.Length ==> a[i] <= a[j]
  ensures multiset(a[..]) == old(multiset(a[..]))
```

`Quicksort` делит входной массив на более мелкие сегменты и рекурсивно сортирует их. Чтобы указать границы сортируемого сегмента, используем вспомогательный метод, который будет вызываться основным методом `Quicksort` с границами `0` и `a.Length`:

```
{
  QuicksortAux(a, 0, a.Length);
}
```

Теперь перейдем к спецификации `QuicksortAux`. Вот части спецификации, которые, очевидно, нам понадобятся:

```
method QuicksortAux(a: array<int>, lo: int, hi: int)
  requires 0 <= lo <= hi <= a.Length
  modifies a
  ensures forall i, j :: lo <= i < j < hi ==> a[i] <= a[j]
  decreases hi - lo
```

Нам также понадобится постусловие, гласящее, что мультимножество элементов массива не изменилось, и потребуется более точная информа-

ция о том, какие элементы массива вообще изменяются. Чтобы выразить `a[i] == old(a[i])` для индексов `i` вне диапазона от `lo` до `hi`, используем квантор. Нам придется несколько раз написать этот квантор и свойство **`multiset`**, поэтому определим предикат, который можно использовать повторно при необходимости.

15.2.1. Предикаты с двумя состояниями

Этот предикат отличается от любого другого из тех, что мы видели до сих пор, поскольку он должен связывать текущее состояние метода с его предыдущим состоянием. Для этой цели в Dafny имеются *предикаты с двумя состояниями*. Они способны читать две кучи: текущую кучу и некоторую кучу в прошлом. Доступ к куче в прошлом осуществляется с помощью выражений **`old`**, в точности как в телах методов и постусловиях. Вызовы предиката с двумя состояниями фактически передаются в кучу в прошлом. Это все, что мы должны знать о предикатах с двумя состояниями, чтобы использовать их в `Quicksort`.

Вот наш предикат с двумя состояниями:

```
twostate predicate SwapFrame(a: array<int>, lo: int, hi: int)
  requires 0 <= lo <= hi <= a.Length
  reads a
{
  (forall i :: 0 <= i < lo || hi <= i < a.Length ==>
    a[i] == old(a[i])) &&
  multiset(a[..]) == old(multiset(a[..]))
}
```

В первую очередь используем этот предикат в постусловии `QuicksortAux`:

```
  ensures SwapFrame(a, lo, hi)
```

15.2.2. Основной алгоритм

Идея алгоритма быстрой сортировки проста. Сначала среди элементов, подлежащих сортировке, выбирается элемент, известный как *опорный элемент* (pivot). Если взять в качестве примера сортировку цветов национального флага Голландии, то представьте, что опорный элемент – это белый цвет, элементы меньше опорного элемента – красные, а элементы больше опорной точки – синие. Затем выполняется сортировка по цвету, как было показано в разделе 15.0. В алгоритме быстрой сортировки подпрограмму, выбирающую опорный элемент, обычно называют `Partition`. Метод `Partition` в нашем примере будет возвращать индекс опорного элемента. Затем выполняется рекурсивная сортировка элементов слева и справа от опорного элемента.

Вот тело `QuicksortAux`, выполняющее только что описанные действия:

```
{
  if 2 <= hi - lo {
    var p := Partition(a, lo, hi);
    QuicksortAux(a, lo, p);
    QuicksortAux(a, p + 1, hi);
  }
}
```

Увы, даже после определения спецификации `Partition` (что мы и сделаем далее) верификация `QuickSortAux` терпит неудачу. Проблема в том, что наш предикат `SwapFrame` не дает необходимой информации об отношении элементов. Можно было бы попытаться доказать некоторые леммы о `SwapFrame`, но такие леммы должны быть *леммами о двух состояниях*, которые не рассматриваются в этой книге. Поэтому мы используем предикат `SplitPoint`, который я представил в разделе 15.1 вместе с описанием сортировки выбором.

15.2.3. Использование SplitPoint

В качестве последнего улучшения спецификации `QuicksortAux` введем пред- и постусловия, требующие, чтобы `lo` и `hi` были точками разделения. Предусловие гласит, что элементы `a[..lo]` находятся ниже элементов `a[lo..hi]`, а те, в свою очередь, ниже элементов `a[hi..]`. А суть постусловия сводится к тому, что метод не меняет местами элементы, находящиеся по разные стороны от точек разделения.

Вот что мы добавим в спецификацию `QuicksortAux`:

```
  requires SplitPoint(a, lo) && SplitPoint(a, hi)
  ensures SplitPoint(a, lo) && SplitPoint(a, hi)
```

15.2.4. Метод Partition

Пришло время написать спецификацию и реализацию `Partition`. По сути, это алгоритм сортировки цветов голландского национального флага, но его формулировка во многом отличается от той, что мы видели в разделе 15.0. Одно из отличий в спецификации состоит в том, что теперь сортируется только часть массива, другое – в том, что теперь мы работаем не с цветами флага (но см. упражнение 15.2), а третье – в том, что `Partition` возвращает индекс опорного элемента. Одно из отличий в реализации состоит в необходимости выбрать опорный элемент, а другое – в сохранении опорного элемента отдельно при сортировке «красных» и «синих» элементов.

Спецификация получилась довольно большой:

```
method Partition(a: array<int>, lo: int, hi: int) returns (p: int)
  requires 0 <= lo < hi <= a.Length
  requires SplitPoint(a, lo) && SplitPoint(a, hi)
  modifies a
```

```
  ensures lo <= p < hi
  ensures forall i :: lo <= i < p ==> a[i] < a[p]
  ensures forall i :: p <= i < hi ==> a[p] <= a[i]
  ensures SplitPoint(a, lo) && SplitPoint(a, hi)
  ensures SwapFrame(a, lo, hi)
```

В дополнение к предикатам `SplitPoint` и `SwapFrame` эта спецификация, подобно спецификации `QuickSortAux`, гласит, что метод возвращает в качестве `p` индекс внутри `a[lo..hi]`. Это важно для рекурсивных вызовов `QuicksortAux`. Постусловие также определяет, как элементы сегмента `a[lo..hi]` связаны с опорным элементом `a[p]`.

15.2.5. Реализация Partition

Для завершения алгоритма быстрой сортировки нам осталось лишь реализовать метод `Partition`. Сделаем это в три шага.

На первом шаге выберем опорный элемент, чтобы разбить массив на сегменты. Выбор опорного элемента влияет на производительность, и, чтобы сделать правильный выбор, обычно используют эвристики. Для доказательства корректности не важно, какой элемент будет выбран, поэтому просто выберем `a[lo]`:

```
{
  var pivot := a[lo];
```

Вторым шагом напишем цикл, переупорядочивающий элементы в `a[lo+1..hi]` на сегменты меньше-опорного-элемента и больше-или-равные-опорному-элементу. По аналогии с голландским национальным флагом используем два маркера, `m` и `n`, соответственно, в `a[lo+1..m]` будут находиться значения меньше опорного элемента, в `a[m..n]` – еще не исследованные элементы и в `a[n..hi]` – равные или больше опорного элемента. Спецификация цикла тоже получилась большой, как и спецификация объемлющего метода:

```
  var m, n := lo + 1, hi;
  while m < n
    invariant lo + 1 <= m <= n <= hi
    invariant a[lo] == pivot
    invariant forall i :: lo + 1 <= i < m ==> a[i] < pivot
    invariant forall i :: n <= i < hi ==> pivot <= a[i]
    invariant SplitPoint(a, lo) && SplitPoint(a, hi)
    invariant SwapFrame(a, lo, hi)
```

Реализация цикла проверяет следующий элемент и либо расширяет нижний сегмент, либо меняет местами элементы в нижнем и верхнем сегментах:

```
  {
    if a[m] < pivot {
```

```
      m := m + 1;
    } else {
      a[m], a[n-1] := a[n-1], a[m];
      n := n - 1;
    }
  }
```

Третьим шагом вставим опорный элемент в область тех элементов, которые мы разделили. Для этого поменяем местами опорный элемент с соседним элементом в верхней области, а затем вернем полученный индекс:

```
  a[lo], a[m - 1] := a[m - 1], a[lo];
  return m - 1;
}
```

Я лишь кратко рассказал об этом алгоритме, характеризующемся его большими спецификациями. Теперь, когда мы дошли до конца, я предлагаю продолжить знакомство с тем, что вы только что видели, вернуться назад и попробовать закомментировать разные части спецификаций, чтобы увидеть, где верификация терпит неудачу.

Упражнение 15.5

Одна из эвристик выбора опорного элемента – взять крайний левый, средний и крайний правый элементы из `a[lo..hi]` и выбрать тот, что по своей величине находится двумя другими. Добавьте в `Partition` код, реализующий этот выбор.

Упражнение 15.6

Напишите метод, сортирующий целочисленный массив, вставляя его элементы в приоритетную очередь, разработанную в главе 10, а затем последовательно удаляющий самый маленький оставшийся элемент.

15.3. Итоги

В этой главе я показал три алгоритма, сортирующих заданный массив путем перестановки его элементов. По пути мы встретили фреймы чтения и записи и предикаты с двумя состояниями. Повсюду мы выражали интересующие нас свойства, используя кванторы всеобщности.

Основная цель этой и нескольких предыдущих глав – показать, как можно рассуждать об алгоритмах, используя инварианты. При конструировании циклов очень важно обращать внимание на начальное и конечное состояния каждой итерации. Инвариант цикла как раз и служит для выражения этих состояний и позволяет сузить фокус внимания с бесконечного числа итераций цикла до поведения одной произвольной итерации. Указав

условие, которое должно выполняться в начале каждой итерации, вы получите четкое руководство по реализации тела цикла.

Инварианты не уникальны для циклов. Например, в главе 10 мы видели инварианты для структур данных, а в последующих главах мы увидим инварианты для объектов.

Вот несколько упражнений, в которых вам будет предложено написать свои доказательства алгоритмов сортировки. Каждое из них включает множество деталей, поэтому не надейтесь быстро покончить с отладкой программ и доказательствами их корректности.

Упражнение 15.7

Операция слияния является центральной в алгоритме сортировки слиянием (функциональную версию которой мы видели в разделе 8.2). Используя массивы, напишите спецификацию и реализацию метода

```
method Merge(a: array<int>, b: array<int>) returns (c: array<int>)
```

который принимает два отсортированных массива, `a` и `b`, и возвращает новый отсортированный массив `c`, содержащий элементы из `a` и `b`.

Упражнение 15.8

В разделе 8.1 мы реализовали функциональную версию сортировки вставками для списка. Теперь вам предлагается написать спецификацию и реализацию метода, выполняющего сортировку вставками для целочисленного массива. Сортировка вставками поочередно вставляет каждый элемент в отсортированный префикс массива.

Вам понадобятся два вложенных цикла. В этом упражнении выполняйте перестановку в каждой итерации внутреннего цикла (проще говоря, используйте метод конструирования циклов 13.1 для выполнения условия «те же самые элементы»).

Упражнение 15.9

Реализуйте сортировку вставками для целочисленного массива, но (в отличие от упражнения 15.8) не выполняйте перестановку во внутреннем цикле. Вместо этого непосредственно перед входом во внутренний цикл сохраните значение вставляемого элемента:

```
var x := a[i];
```

Затем в каждой итерации внутреннего цикла просто скопируйте следующий элемент на место:

```
while // ...
  a[j] := a[j - 1];
```

По завершении внутреннего цикла запишите сохраненное значение на место:

```
a[j] := x;
```

Эти шаги приведут к перемещению элемента на место и сэкономят время, поскольку в каждой итерации будет выполняться только «половина перестановки».

Подсказка: чтобы доказать свойство «те же самые элементы», вашему внутреннему циклу потребуется инвариант:

```
invariant multiset(a[..][j := x]) == multiset(old(a[..]))
```

где выражение обновления последовательности `s[j := x]` обозначает последовательность, подобную `s`, в которой элемент `j` заменен значением `x`.

Примечания

Быструю сортировку изобрел Чарльз Энтони Ричард Хоар (C. A. R. Hoare), который также ввел нотацию троек, используемую нами при рассуждении о программах. Спустя годы после изобретения Quicksort стал одним из первых алгоритмов сортировки, прошедших формальную верификацию [51]. Теперь, более полувека спустя, интересно прочитать замечания о перспективах автоматизации подобной верификации, приводившиеся в конце этой статьи.

Еще раньше формальное доказательство корректности было разработано для алгоритма Treesort 3 [88] – версии пирамидальной сортировки, предложенной Робертом Флойдом (Robert W. Floyd).

Существуют также формальные доказательства корректности для многих других алгоритмов сортировки. Например, более современные доказательства корректности сортировки вставками, быстрой сортировки и пирамидальной сортировки были формализованы в системе Coq [49], а сортировка подсчетом (см. также упражнение 15.3) и поразрядная сортировка были проверены в системе KeY [37].

В разделе «Примечания» главы 8 я упомянул некоторые функциональные алгоритмы сортировки. Галерея верифицированных программ Why3, о которой я там упоминал, содержит также несколько императивных алгоритмов сортировки [20].

При попытке формально верифицировать алгоритм сортировки TimSort из стандартной библиотеки Java была выявлена ошибка в реализации [38]. Эта ошибка затем была исправлена, а исправленный алгоритм повторно верифицирован в системе KeY.

Для удобства при разработке быстрой сортировки в разделе 15.2 мы объявили и использовали предикат `SwapFrame` с двумя состояниями (**`twostate`**). Такие предикаты поддерживаются некоторыми верификаторами (напри-

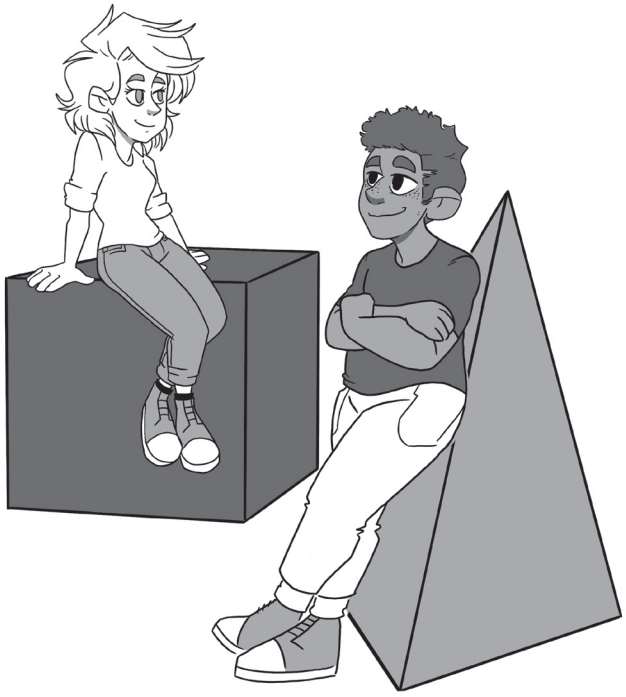
мер, Frama-C [14] поддерживает пользовательские предикаты, связывающие два или более состояний памяти), но далеко не всеми. К счастью, нашу программу `Quicksort` легко написать без такого предиката, что и будет предложено в следующем упражнении.

Упражнение 15.10

В программе `Quicksort` из раздела 15.2 замените все вызовы предиката `SwapFrame` с двумя состояниями телом `SwapFrame` (параметризовав его соответствующим образом), а затем полностью удалите объявление `SwapFrame`. Убедитесь, что обновленная программа успешно проходит верификацию.

Глава 16

Объекты



Объект – это абстракция состояния с идентичностью. В языках с поддержкой классов (включая Dafny) каждый объект создается как экземпляр *класса*. Доступ к объекту всегда осуществляется по *ссылке* (так же как к массивам). Класс определяет компоненты состояния своих объектов, которые называются *полями*, а также операции над объектами – функции и методы. В совокупности поля, функции и методы класса называются *членами* класса. Идентичность объекта определяется его ссылкой, а не значением его полей.

В этой главе я приведу три примера использования объектов и покажу, как писать спецификации объектов и рассуждать о них. Двумя важными ингредиентами являются *инварианты* структур классов и *фреймы*.

16.0. Контрольные суммы

Напишем класс `ChecksumMachine`, объекты которого вычисляют контрольные суммы для некоторых данных. Клиенты могут предоставлять данные по частям и постепенно вычислять контрольную сумму. А так как в этом случае необходимо сохранять состояние, используем объекты.

Роль данных будет играть строка, т. е. последовательность символов (Dafny определяет тип **`string`** как синоним **`seq<char>`**). Контрольная сумма будет вычисляться как простой хеш-код: остаток от деления на 137 суммы целочисленных значений кодов символов в строке. Выражение `ch` **`as int`** преобразует символ `ch` в целочисленное значение. Вот определение нашей

хеш-функции. Она объявлена призрачной, потому что будет вызываться только в спецификациях:

```
ghost function Hash(s: string): int {
  SumChars(s) % 137
}

ghost function SumChars(s: string): int {
  if |s| == 0 then 0 else
    var last := |s| - 1;
    SumChars(s[..last]) + s[last] as int
}
```

16.0.0. Спецификация

Вот как выглядит определение нашего класса с точки зрения клиентов:

```
class ChecksumMachine {
  var data: string
  constructor ()
    ensures data == “”
  method Append(d: string)
    modifies this
    ensures data == old(data) + d
  function Checksum(): int
    reads this
    ensures Checksum() == Hash(data)
}
```

Это объявление представляет `ChecksumMachine` как класс с одним полем `data` для накопления фрагментов данных, предоставляемых клиентами через метод `Append`. Чтобы обратиться к члену класса, нужна ссылка на объект. Например, если `m` является ссылкой на объект `ChecksumMachine`, то `m.data` обозначает доступ к полю `data` этого объекта.

Класс предоставляет *конструктор* для инициализации объекта при его создании. Конструкторов может быть несколько, и им можно дать имена, как и методам. Часто классы имеют *анонимный* конструктор, как в этом примере. Конструкторы могут принимать входные параметры, но в нашем примере параметры не нужны. Постусловие конструктора сообщает вызывающему коду о состоянии объекта после создания.

Форма метода `Append` и функции `Checksum` должна казаться знакомой, но есть одно отличие от виденного нами ранее. Эти метод и функция объявлены как члены типа `ChecksumMachine`, поэтому они неявно принимают дополнительный параметр, известный как *приемник* и доступный под именем **`this`**. Его можно видеть в инструкции **`modifies`** в спецификации метода `Append`, сообщающей, что `Append` может изменять поля **`this`**, и в инструкции **`reads`** в

спецификации функции Checksum, указывающей, что Checksum может зависеть от полей **this**.

Неявный характер параметра **this** обеспечивает естественный способ обращения к полям и методам класса. Спецификации в примере показывают несколько вариантов использования выражения data, которое является сокращенной формой записи **this**.data.

Метод Append может изменять состояние своего параметра-приемника, а его постусловие связывает значения **this**.data до и после. Значение выражения, описывающего состояние до выполнения операции, обозначается с помощью слова **old**, представленного в разделе 14.0.1. Как я уже упоминал в том разделе, **old** влияет только на разыменование кучи в выражении аргумента. Для объектов это означает (явную или неявную) точку «.», разыменовывающую поля. Например, **old**(this.data) и **old**(data) относятся к начальному значению поля data в **this**, а выражение **old**(**this**).data применяет **old** к аргументу, не разыменовывающему кучу, поэтому его результат совпадает с результатом выражения **this**.data.

Я использовал постусловие с функцией Checksum() вместо предоставления ее тела. Это позволяет при реализации функции предоставить в качестве тела функции альтернативное, но эквивалентное выражение. Ссылка на Checksum() в постусловии относится к значению, возвращаемому функцией (см. главу 6).

Упражнение 16.0

Определите тела конструктора, метода и функции в ChecksumMachine.

16.0.1. Тестовая программа

Прежде чем обсуждать реализацию ChecksumMachine, напишем тестовую программу (см. раздел 9.3.3 и упражнение 10.0). Тестовая программа показывает, как клиент использует класс, и проверяет адекватность наших спецификаций:

```
method ChecksumMachineTestHarness() {
  var m := new ChecksumMachine();
  m.Append(“green”);
  m.Append(“grass”);
  var c := m.Checksum();
  print “Checksum is “, c, “\n”;
}
```

Этот метод размещает в памяти объект ChecksumMachine, вызывает его конструктор и сохраняет ссылку на объект в локальной переменной m. Затем к ссылке m на объект применяется операция Append. Обратите внимание, что перед именем вызываемого метода добавляется ссылка m, определяющая

значение для параметра приемника. В заключение тестовая программа получает и выводит контрольную сумму строки «greengrass».

Давайте поразмышляем, как работает фрейминг в этом примере. Тестовый метод ChecksumMachineTestHarness не имеет никаких собственных инструкций **modifies**. Это означает, что он не может изменить состояние прежде существовавшего объекта. Расширяя правило о фреймах, представленное в разделе 14.0.2, получаем:

> Метод может создать новый массив или объект и изменять его состояние (т. е. элементы массива или поля объекта) без упоминания этого массива или объекта в инструкции **modifies**.

Приведенная выше тестовая программа создает объект ChecksumMachine и поэтому может изменять состояние этого объекта. В спецификации метода Append говорится, что он может изменять состояние параметра приемника, который тестовая программа передала как m. Отсюда мы заключаем, что ChecksumMachineTestHarness удовлетворяет своей спецификации.

16.0.2. Инвариант

Как предполагалось изначально, класс ChecksumMachine должен вычислять контрольную сумму инкрементально. То есть мы должны хранить контрольную сумму, вычисленную к настоящему моменту, и обновлять ее при каждом вызове Append. Это избавит нас от необходимости хранить конкатенацию всех строк, добавленных во время выполнения, и функция Checksum будет вычислять текущую контрольную сумму.

Для этого добавим поле текущей контрольной суммы. Внутри класса объявим его так:

```
var cs: int
```

Мы должны продолжить использовать data в спецификациях, но не в скомпилированной программе, поэтому сделаем это поле призрачным:

```
ghost var data: string
```

Следующий шаг немного сложнее. Мы должны зафиксировать наше решение, чтобы сохранить отношение:

```
cs == Hash(data)
```

Это *инвариант* каждого объекта. Мы уже видели подобные инварианты на примере структур данных в главе 10, где наши структуры были неизменяемыми и определялись как объявления **datatype**. Здесь мы поступим точно так же и объявим предикат Valid(), возвращающий **true**, если объект удовлетворяет своему инварианту. Чтобы сделать этот предикат инвариантом объекта, мы должны использовать его в постусловии конструктора, а также в пред- и постусловии каждого метода. Другими словами, конструк-

тор обеспечивает соответствие предикату `Valid()`, а все методы сохраняют это соответствие.

Вот наше определение предиката `Valid()`:

```
ghost predicate Valid()
  reads this
{
  cs == Hash(data)
}
```

Поскольку функция `Valid()` читает поля **`this`**, она должна включать инструкцию **`read`**. Подобные фреймы чтения для массивов я описал в разделе 14.0.4. Вот то же правило фреймов, но применительно к объектам:

Если функция обращается к полям объекта `o`, ее спецификация должна включать инструкцию **`reads`** `o`.

> **Врезка 16.0**
>
> В этой главе я привожу правила фреймов для объектов, которые почти целиком совпадают с аналогичными правилами для массивов. Если бы я сначала описывал объекты, а потом массивы, то мог бы просто указать, что массивы – это объекты, поддерживающие особый синтаксис. Элементы массива – это «поля» объекта массива, а «имена полей» для элементов массивов не являются литералами, а вычисляются с помощью целочисленных выражений.

Вот обновленная версия той части нашего класса, которую должны видеть клиенты:

```
class ChecksumMachine {
  ghost var data: string
  ghost predicate Valid()
    reads this
  constructor ()
    ensures Valid() && data == ""
  method Append(d: string)
    requires Valid()
    modifies this
    ensures Valid() && data == old(data) + d
  function Checksum(): int
    requires Valid()
    reads this
    ensures Checksum() == Hash(data)
}
```

Обратите внимание, что реализация cs – это частное решение, поэтому я исключил его из объявления выше. Более того, cs не упоминается в спецификациях. Это хорошая практика сокрытия информации: она освобождает клиентов от необходимости знать подобные детали и позволяет свободно изменять эти детали в будущем (я подробно обсуждал тему сокрытия информации в главе 9, а модуль для текущего примера я объявлю в разделе 16.0.4).

Внося изменения в спецификации, рекомендуется также обновить тестовый код. В нашем случае добавление Valid() не влияет на тестовую программу, которая по-прежнему верифицируется автоматически.

16.0.3. Реализация

Спроектировав класс, перейдем к реализации конструктора, метода и функции.

Конструктор и функция Checksum реализуются просто:

```
constructor ()
  ensures Valid() && data == ""
{
  data, cs := "", 0;
}

function Checksum(): int
  requires Valid()
  reads this
  ensures Checksum() == Hash(data)
{
  cs
}
```

Конструктор изменяет поля **this**, но **this** считается объектом, созданным внутри конструктора, поэтому инструкция **modifies** не нужна.

> Конструктор может присваивать значения полям конструируемого объекта **this**, не упоминая **this** в инструкции **modifies**.

Метод Append можно реализовать с помощью цикла, который обрабатывает строку из параметра d по одному символу за раз:

```
method Append(d: string)
  requires Valid()
  modifies this
  ensures Valid() && data == old(data) + d
{
  var i := 0;
  while i != |d|
    invariant 0 <= i <= |d|
```

```
    invariant Valid()
    invariant data == old(data) + d[..i]
  {
    cs := (cs + d[i] as int) % 137;
    data := data + [d[i]];
    i := i + 1;
  }
}
```

Упражнение 16.1

(*Усложненное*) В начале этого раздела я определил рекурсивную функцию `SumChars`, вычисляющую контрольную сумму для заданного префикса строки (подобно функции `SumDown` в разделе 12.3.0). Она хорошо вписывается в приведенный выше цикл в `Append`. Измените определение функции `SumChars`, чтобы она вычисляла контрольную сумму для заданного суффикса строки (подобно `SumUp` в разделе 12.3.0). Затем измените тело метода `Append`, чтобы оно успешно верифицировалось. (Внимание: это упражнение требует некоторого терпения, потому что для его выполнения нужно добавить инструкции **`assert`** для проверки некоторых свойств последовательностей.)

16.0.4. Модуль

Выше я довольно неформально высказался о том, что «видят клиенты» в нашем классе. Давайте завершим пример, заключив код в модуль и добавив экспортируемый набор. Чтобы освежить память о модулях и экспортируемых наборах, можно вернуться к разделу 9.2.

```
module Checksums {
  export
    reveals Hash, SumChars, ChecksumMachine
    provides ChecksumMachine.data, ChecksumMachine.Valid
    provides ChecksumMachine.Append, ChecksumMachine.Checksum

  // определения, представленные выше в этом разделе...
}
```

Здесь я экспортировал тело `Hash` и `SumChars`. Если вы решите ограничить доступность тела `Hash` рамками модуля, то перенесите `Hash` в раздел **`provides`** и вообще опустите `SumChars`.

Здесь также экспортируется тело `ChecksumMachine`, тем самым мы сообщаем импортерам, что этот тип является классом. При экспортировании тела класса его анонимный конструктор (если таковой имеется) автоматически добавляется в экспортируемый набор. Любые другие элементы класса, которые должны быть доступны клиентам, необходимо явно перечислить

в экспортируемом наборе. Например, экспортируемый набор выше открывает доступ к полю data, тогда как поле cs не экспортируется и потому недоступно вне модуля.

16.0.5. Итоги

В этом примере мы начали с простого класса, содержащего компилируемое поле data, которое используется и в спецификациях, и (см. упражнение 16.0) в реализациях.

Затем добавили компилируемое поле cs и предикат Valid(), чтобы установить инвариантную связь между data и cs. Одновременно мы изменили спецификации, согласно которым конструктор обеспечивает соответствие предикату Valid(), функция требует соответствия предикату Valid(), а метод поддерживает его. Потом мы изменили реализации, использовав в них только поле cs. Это позволило нам сделать поле data призрачным. Наконец, мы написали модуль с экспортируемым набором для нашего класса.

Упражнение 16.2

Напишите вариант модуля Checksums, в котором класс отслеживает SumChars(data) вместо Hash(data). То есть, в отличие от приведенной выше программы, которая использует поле cs с инвариантом cs == Hash(data), используйте поле sum с инвариантом sum == SumChars(data). Интерфейс модуля, доступный клиенту, должен остаться неизменным.

Упражнение 16.3

Программа из упражнения 16.2 выполняет одну операцию вычисления остатка от деления в каждом вызове функции Checksum(). Соответственно, если функция вызывается дважды без изменения поля data, она повторно вычислит значение sum % 137. Если операция (в данном случае операция вычисления остатка от деления) является дорогостоящей, то можно использовать технику мемоизации для сохранения и повторного использования вычисленного значения. Измените решение в упражнении 16.2, включив в него мемоизацию.

В частности, замените функцию Checksum() одноименным методом, чтобы он мог изменять состояние объекта ChecksumMachine:

```
method Checksum() returns (checksum: int)
  requires Valid()
  modifies this
  ensures Valid() && data == old(data)
  ensures checksum == Hash(data)
```

Затем добавьте поле **var** ocs: Option**<int>**, где Option – тип

```
datatype Option<T> = None | Some(T)
```

Инвариант будет заключаться в том, что `ocs` получает значение `None` (если нет мемоизованного результата) или `Some(sum % 137)`. Метод `Checksum()` должен вернуть сохраненную контрольную сумму, если она была вычислена ранее, или, в противном случае, вычислить и сохранить ее перед возвратом.

Упражнение 16.4

Напишите императивную версию программы `BlockChain` из упражнения 10.5. В частности, объявите `Ledger` классом:

```
class Ledger {
  var transactions: List<int>
  var balance: nat

  ghost predicate Valid()
    reads this
  {
    balance == Sum(transactions)
  }
  // ...
}
```

где `List` – это тип данных из раздела 6.0, а `Sum` – рекурсивная призрачная функция, которую вы напишете для сложения чисел в заданном списке `List<int>` (подсказка: `Sum` будет выглядеть подобно функции `Length` из раздела 6.1). В вашем классе также должны быть определены конструктор и два метода, `Deposit` и `Withdraw`, изменяющие объект. Напишите их спецификации с точки зрения `Valid()` и `balance`. Также напишите и верифицируйте тестовую программу, чтобы убедиться, что ваши спецификации соответствуют требованиям. Реализуйте и верифицируйте класс.

16.1. Токенизатор

В программировании часто приходится решать задачу преобразования групп небольших элементов в более крупные элементы. Один из таких примеров – *токенизатор*, который анализирует поток символов и преобразует его в поток строк различных синтаксических категорий. В этом разделе я представлю класс, реализующий простой токенизатор. Он читает символы из фиксированной строки и разбивает их на токены, которые возвращает по одному. В примере используются простые инварианты, которые мы определим как предикат, проверяющий выполнение свойств структуры данных, по аналогии с предыдущим разделом.

16.1.0. Синтаксические категории

Начнем с определения синтаксических категорий:

```
datatype Category = Identifier | Number | Operator
                  | Whitespace | Error | End
```

Любой набор входных символов, не соответствующий первым четырем категориям, считается ошибкой. Категория `End` означает конец входных данных.

Далее нам нужен способ, с помощью которого можно узнать, какой категории соответствует символ. Категории, распознаваемые этим простым токенизатором, не имеют общих символов (но см. упражнение 16.6). Мы не собираемся когда-либо применять этот предикат к `End`, поэтому исключим его, используя предусловие:

```
predicate Is(ch: char, cat: Category)
  requires cat != End
  decreases cat == Error
{
  match cat
  case Whitespace => ch in " \t\r\n"
  case Identifier => 'A' <= ch <= 'Z' || 'a' <= ch <= 'z'
  case Number => '0' <= ch <= '9'
  case Operator => ch in "+-*/%!=><~^&|"
  case Error =>
    !Is(ch, Identifier) && !Is(ch, Number) &&
    !Is(ch, Operator) && !Is(ch, Whitespace)
}
```

Две альтернативы используют **`in`** для проверки принадлежности к последовательности символов, т. е. к строке. Инструкция **`decreases`** необходима для доказательства завершимости рекурсии. Она уменьшает логическое значение **`true`** выражения `cat == Error` до **`false`** (см. раздел 3.2).

Упражнение 16.5

Что произойдет в предикате `Is`, если:

а) удалить предусловие?

б) удалить инструкцию **`decreases`**?

16.1.1. Класс и инвариант

Экземпляр класса `Tokenizer` хранит исходную строку в своем поле. Поскольку это поле остается неизменным на протяжении всего существования объекта, его можно объявить *неизменяемым* с помощью ключевого слова **`const`**. Неизменяемое поле определяется конструктором и впоследствии никогда не изменяется:

```
class Tokenizer {
  const source: string
}
```

Остальные части этого примера тоже объявлены внутри этого класса.

Единственная другая часть состояния токенизатора – поле `n`, используемое для отслеживания количества символов, обработанных на данный момент, т. е. в `n` всегда будет иметь значение в диапазоне от `0` до `|source|`, поэтому отразим этот факт в предикате, который проверяет выполнение свойств структуры данных и и фиксирует инвариант объекта:

```
  var n: nat
  ghost predicate Valid()
    reads this
  {
    n <= |source|
  }
```

Я решил объявить `n` как поле типа **`nat`**. Его также можно объявить с типом **`int`**, но тогда инвариант должен указывать нижнюю границу `0 <= n`.

Чтобы создать объект `Tokenizer`, нужна исходная строка, которая будет разбита на токены:

```
  constructor (s: string)
    ensures Valid() && source == s && n == 0
  {
    source, n := s, 0;
  }
```

16.1.2. Спецификация метода Read

Остается самая интересная часть класса: метод `Read()`, возвращающий следующий токен. По задумке метод `Read` должен отбрасывать токены, состоящие из одних пробельных символов. Выходными параметрами метода являются категория возвращаемого токена и строка, представляющая сам токен. Чтобы сделать спецификацию более точной, предусмотрим возможность говорить о позиции в исходной строке, где начинается возвращаемый токен. Если нет необходимости возвращать эту позицию во время выполнения, то этот выходной параметр можно объявить призрачным:

```
method Read() returns (cat: Category, ghost p: nat, token: string)
```

Поддерживать или не поддерживать выполнение свойств структуры данных

Спецификация `Read` начинается так же, как спецификация метода `Append`, изменяющего состояние объекта и представленного в предыдущем разделе:

```
requires Valid()
modifies this
ensures Valid()
```

Это характерно для методов, изменяющих состояние объектов. Размышляя о методе `Read`, мы должны подумать о том, что должно произойти, когда будет достигнут конец строки или обнаружена ошибка. Как я писал выше, `Read` всегда обеспечивает выполнение свойств структуры данных, а выполнение свойств структуры данных – это все, что требуется для вызова. Следовательно, `Read` можно вызвать даже после того, как он обнаружит конец строки или ошибку.

Если бы мы хотели сделать это по-другому, то могли бы сделать так, чтобы предусловие не выполнялось, если предыдущий вызов `Read()` вернул `End` или `Error`. Но как обратиться к предыдущему возвращаемому значению `Read()`? Один из способов – ввести призрачное поле `ReadIsCallable`, присвоить ему значение **`false`**, когда `Read` возвращает `End` или `Error`, и потребовать `ReadIsCallable == true` в качестве предусловия для `Read`.

Более простая и столь же эффективная альтернатива – не обещать в `Read` обеспечивать выполнение свойств структуры данных после достижения конца строки или обнаружения ошибки. Для этого можно просто изменить постусловие выше на

```
ensures cat == End || cat == Error || Valid()
```

Это делает токенизатор непригодным для использования после достижения конца строки или обнаружения ошибки, потому что единственный способ для вызывающего кода выяснить соответствие предикату `Valid()` после вызова – сравнить возвращаемую категорию с `End` и `Error`.

Стоит подчеркнуть, что это неполная спецификация. То есть возможно, что на самом деле соответствие `Valid()` сохраняется даже после достижения конца строки или обнаружения ошибки. Спецификация, такая как

```
ensures cat == End || cat == Error ==> !Valid()
```

гласит, что в данном случае условие, что выполняются свойства структуры данных, не выполняется. Такая спецификация никому не нужна, поэтому лучше использовать неполную спецификацию свойств структуры данных.

Что ж, продолжим использовать первое постусловие, написанное выше, утверждающее, что `Read` всегда поддерживает выполнение свойств структуры данных.

Постусловие

Постусловия должны приносить пользу вызывающей стороне, и каждое постусловие создает обязательство по реализации. Нецелесообразно включать в постусловие все, что истинно после вызова, потому что некоторые

из этих вещей являются частными решениями реализации. Что касается метода Read, то может возникнуть ощущение, что мы пытаемся включить в постусловие все, что только получается придумать, потому что я буду писать много условий. Я полагаю, что все эти условия помогают нам продумать реализацию метода. Но если вы считаете иначе, то можете попробовать написать меньше постусловий, чем я.

Я уже упоминал, что токенизатор будет отбрасывать токены, состоящие из пробельных символов. Запишем это как постусловие:

```
ensures cat != Whitespace
```

Я упомянул, что мы собираемся вернуть выходной параметр p с позицией в исходной строке после пропуска пробелов. Таким образом, мы имеем

```
ensures old(n) <= p <= n <= |source|
```

Категория End относится к концу исходной строки. Она возвращается, когда p равно |source|. Фактически это единственный случай, когда возвращается End:

```
ensures cat == End <==> p == |source|
```

Если пропуск пробелов приводит к концу строки или к ошибке, то мы должны прекратить синтаксический анализ. То есть у нас будет выполняться условие p == n. Для всех остальных категорий мы выполним дополнительный анализ:

```
ensures cat == End || cat == Error <==> p == n
```

Давайте теперь свяжем символы в строке с категориями. Во-первых, source[old(n)..p] будет пробелом:

```
ensures forall i :: old(n) <= i < p ==>
  Is(source[i], Whitespace)
```

Следующий фрагмент строки source[p..n] будет относиться к возвращаемой категории. Обратите внимание, что это также имеет место, когда категория End или Error, потому что тогда p == n:

```
ensures forall i :: p <= i < n ==> Is(source[i], cat)
```

Идея состоит в том, чтобы разбить входные данные на максимально длинные токены. Она фиксируется следующим постусловием, которое гласит, что то, что следует за n, является другой категорией:

```
ensures p < n ==> n == |source| || !Is(source[n], cat)
```

Посылка p < n в этом постусловии необходима, потому что остальная часть условия не выполняется, когда возвращается категория Error.

Наконец, мы сообщаем, что возвращается в token:

```
  ensures token == source[p..n]
```

Постусловие получилось длинным и содержащим множество деталей. Хорошо, что у нас есть верификатор, который может помочь проверить и доказать корректность реализации всех деталей.

16.1.3. Реализация Read

Реализация `Read` состоит из трех частей: первая перешагивает через пробелы, вторая определяет синтаксическую категорию, и третья сканирует токен.

Первая часть выполняется циклом. Мы должны задать множество постусловий, поэтому включим инварианты цикла для тех случаев, которые относятся к `p` и `source[old(n)..p]` и пробелам:

```
{
  // пропуск пробелов
  while n != |source| && Is(source[n], Whitespace)
    invariant old(n) <= n <= |source|
    invariant forall i :: old(n) <= i < n ==>
      Is(source[i], Whitespace)
  {
    n := n + 1;
  }
  p := n;
```

Просмотрев постусловия метода, можно заметить, что инварианты этого цикла учитывают все постусловия, касающиеся этой части тела метода.

Далее мы определяем значение `cat`:

```
  // определение синтаксической категории
  if n == |source| {
    return End, p, "";
  } else if Is(source[n], Identifier) {
    cat := Identifier;
  } else if Is(source[n], Number) {
    cat := Number;
  } else if Is(source[n], Operator) {
    cat := Operator;
  } else {
    return Error, p, "";
  }
```

В двух случаях возвращается `p == n`. Постусловие гласит, что в таких случаях выходной параметр `token` должен содержать пустую строку.

Наконец, мы увеличиваем `n`, пока символы продолжают соответствовать категории `cat`:

```
    // чтение токена
    var start := n;
    n := n + 1;
    while n != |source| && Is(source[n], cat)
      invariant p <= n <= |source|
      invariant forall i :: p <= i < n ==> Is(source[i], cat)
    {
      n := n + 1;
    }
    token := source[start..n];
  }
```

Оба инварианта цикла используют метод конструирования циклов 13.1 (*всегда*). Последние два постусловия следуют из отрицания условия цикла и значения, присваиваемого выходному параметру `token` соответственно.

Упражнение 16.6

Измените в примере `Tokenizer` обработку первого символа токена. В частности, замените предикат `Is` двумя предикатами: `IsStart(ch, cat)`, который возвращает **`true`**, если символ `ch` – допустимый начальный символ категории `cat`, и `IsFollow(ch, cat)`, который возвращает **`true`**, если `ch` – допустимый неначальный символ категории `cat`. Используйте их, чтобы позволить токенам из категории `Identifier` начинаться с буквы и содержать любые буквы и цифры. Затем обновите спецификацию и реализацию метода `Read()`.

Упражнение 16.7

Текущая реализация `Tokenizer` сохраняет заданную строку навсегда. Измените реализацию класса так, чтобы можно было сократить размер хранимой строки с помощью метода, который удаляет из исходной строки уже обработанный префикс. В частности, объявите поле `source` призрачным и добавьте изменяемое поле `suffix` с инвариантом `suffix == source[m..]`, где `m` – новое поле. Например, вы можете ввести поля `m` и `j` такие, что `m + j == n`, и в этом случае вы также можете объявить поле `n` как призрачное. Измените реализацию `Read` соответствующим образом.

16.2. Простые агрегатные объекты

Объекты часто реализуются в терминах других объектов. Такие объекты называются *агрегатными объектами*, и им будет посвящена оставшаяся часть книги. Я представлю их в двух разделах. В этом разделе мы создадим агрегатный объект, состоящий из объектов простых типов, которые мы видели в двух предыдущих разделах. Затем в разделе 16.3 мы создадим агрегатный объект из других агрегатных объектов.

И в этом, и в следующем разделе я использую в качестве примера кофеварку, состоящую из кофемолки и резервуара для воды. То есть кофеварка – это агрегатный объект, *составляющими объектами* (или частями) которого являются кофемолка и резервуар для воды. Этот пример позволит нам сосредоточиться на структуре агрегатного объекта. Полученный класс не имеет большой практической ценности, но, изучив технику создания спецификаций здесь, в следующей главе мы напишем несколько полезных программ.

16.2.0. Составляющие объекты с простыми фреймами

Класс кофеварки, который мы напишем, использует два других класса. Вот как с точки зрения кофеварки выглядят эти классы, имеющие форму, подобную той, что мы видели в двух предыдущих разделах:

```
class Grinder {
  var HasBeans: bool
  ghost predicate Valid()
    reads this
  constructor ()
    ensures Valid()
  method AddBeans()
    requires Valid()
    modifies this
    ensures Valid() && HasBeans
  method Grind()
    requires Valid() && HasBeans
    modifies this
    ensures Valid()
}

class WaterTank {
  var Level: nat
  ghost predicate Valid()
    reads this
  constructor ()
    ensures Valid()
  method Fill()
    requires Valid()
    modifies this
    ensures Valid() && Level == 10
  method Use()
    requires Valid() && Level != 0
    modifies this
    ensures Valid() && Level == old(Level) - 1
}
```

Как видите, каждый класс объявляет предикат, проверяющий выполнение свойств структуры данных и используемый в качестве инварианта объекта. Детали предиката не важны для кофеварки, за исключением того факта, что `Valid()` зависит только от состояния **`this`**.

Мы воспользуемся еще одним классом `Cup`, который будет использоваться в выходных данных кофеварки. Назначение класса `Cup` во многом надуманное. Его главная цель – наводить на мысль о некотором полезном результате. И действительно, говоря о классе `Cup`, я буду упоминать лишь, что он имеет конструктор:

```
class Cup {
  constructor ()
}
```

16.2.1. Версия 0 класса CoffeeMaker

В дополнение к предикату, проверяющему выполнение свойств структуры данных, и конструктору наш класс кофеварки имеет функцию `Ready()`, которая сообщает, имеются ли в кофеварке в данный момент все ингредиенты, необходимые для приготовления чашки кофе, метод `Restock()`, загружающий ингредиенты, и метод `Dispense()`, который готовит чашку кофе при условии, что в кофеварку загружены все необходимые ингредиенты:

```
class CoffeeMaker {
  ghost predicate Valid()
    reads this
  constructor ()
    ensures Valid()
  predicate Ready()
    requires Valid()
    reads this
  method Restock()
    requires Valid()
    modifies this
    ensures Valid() && Ready()
  method Dispense(double: bool) returns (c: Cup)
    requires Valid() && Ready()
    modifies this
    ensures Valid()
}
```

Как видите, в этом классе предикат `Valid()` используется как инвариант объекта, следуя стилю, который мы видели в предыдущих разделах этой главы. Функцию `Ready()` можно вызвать перед `Dispense()`, а `Restock()` гарантирует возможность выполнить один вызов `Dispense()`.

Приступая к реализации этого класса, добавим поля для хранения кофемолки и резервуара для воды:

```
var g: Grinder
var w: WaterTank
```

Кофеварка находится в состоянии, когда выполняются свойства структуры данных, если есть оба этих объекта. Это отражено в предикате, проверяющем выполнение свойств класса `CoffeeMaker`:

```
ghost predicate Valid()
  reads this
{
  g.Valid() && w.Valid() // ошибка: инструкция reads неполная
}
```

Увы, такое положение вещей не устраивает верификатора. Предикату `Valid()` разрешено читать только поля **`this`**, но на самом деле он также читает поля **`this`**`.g` и **`this`**`.w`, которые читаются предикатами `g.Valid()` и `w.Valid()`. Чтобы учесть эти дополнительные зависимости, можно скорректировать инструкцию **`reads`** в `Valid()` следующим образом:

```
ghost predicate Valid()
  reads this, g, w
```

Внесем такое же изменение в инструкцию **`reads`** в `Ready()` и изменим инструкцию **`modifies`** в `Restock()` и `Dispense()`, как показано ниже:

```
modifies this, g, w
```

Посредством этих расширенных фреймов мы дали классу кофеварки возможность читать и изменять два составляющих ее объекта. Но при этом мы нарушили важные принципы сокрытия информации: использование одной кофемолки (`Grinder`) и одного резервуара для воды (`WaterTank`) – это частное решение организации класса `CoffeeMaker`. Для нас нежелательно, чтобы на клиентов класса `CoffeeMaker` влияли особенности его реализации, такие как использование другого набора составляющих объектов.

Соответственно, фреймы должны учитывать составляющие объекты, на которые ссылаются поля `g` и `w`, но эти поля не должны упоминаться в инструкциях **`reads`** и **`modifies`**. Эту проблему с фреймами легко решить с применением абстракции. Мы абстрагируем составляющие объекты в *набор представлений*.

16.2.2. Наборы представлений

Набор представлений (representation set) объекта `o` – это набор объектов, используемых для реализации абстракции, предоставляемой объектом `o`. Такой набор всегда включает сам объект `o`, а также объекты, составляющие его. Для объекта `o` класса `CoffeeMaker` набор представлений выглядит так: `{o, o.g, o.w}`. Клиенту, который использует `o`, вызывая его методы и функции, не обязательно знать, как реализован `o`. Но для клиента желательно знать, что `o` может быть реализован с использованием других объектов.

Пусть Repr обозначает набор представлений объекта CoffeeMaker. Мы должны перечислить весь этот набор объектов в инструкциях **modifies** и **reads**. У нас уже есть опыт перечисления отдельных объектов в таких предложениях, но Dafny также позволяет включать в инструкции **modifies** и **reads** выражения, обозначающие наборы объектов. Итак, если Repr является таким набором, то мы просто должны перечислить Repr в этих инструкциях спецификации фреймов.

Чтобы получить следующую версию класса CoffeeMaker, объявим Repr как поле в классе:

```
ghost var Repr: set<object>
```

Поле Repr нам нужно только для целей спецификации, поэтому оно объявлено призрачным. Тип Repr – это множество (**set**) объектов (**<object>**), поэтому Repr может содержать ссылки на объекты любого типа (включая массивы). Это удобно, потому что в набор представлений нашей кофеварки входят кофеварка (CoffeeMaker), кофемолка (Grinder) и резервуар для воды (WaterTank).

Далее изменим спецификации методов Restock и Dispense и укажем Repr в их фреймах записи:

```
modifies Repr
```

Аналогично изменим спецификацию фрейма чтения в Ready:

```
reads Repr
```

Далее, нам нужно где-то указать, какие объекты входят в Repr. Так как этот набор становится частью инварианта объекта, изменим тело Valid():

```
this in Repr &&
g in Repr && g.Valid() &&
w in Repr && w.Valid()
```

Этот код говорит, что **this**, g и w являются частью Repr. То же самое можно было бы выразить чуть иначе:

```
Repr == {this, g, w} &&
g.Valid() && w.Valid()
```

Однако обычно для объектов, включенных в Repr, указывается только нижняя граница, поэтому я буду придерживаться первого варианта.

Это было тело предиката Valid(), согласно которому Repr – это набор представлений CoffeeMaker. Точнее говоря, тело Valid() сообщает, что поле Repr (типом которого является множество объектов) допустимого объекта CoffeeMaker содержит объекты, на которые ссылаются поля g и w.

А как насчет спецификации фрейма в предикате Valid? Фрейм чтения Valid должен включать те объекты, от полей которых зависит предикат. Тело Valid читает поля объекта **this** (в частности, поля Repr, g и w), а также поля объектов g и w (вызовы g.Valid() и w.Valid()). Эти три объекта находятся

в наборе представлений, на что указывает замечание верификатора относительно следующего фрейма чтения для Valid:

```
reads Repr // ошибка: инструкция reads неполная
```

Это не совсем правильно. Этот вариант *будет* работать, когда выполняется условие Valid(), поскольку Valid() подразумевает, что **this**, g и w находятся в Repr. Но Valid() – это всего лишь функция, возвращающая **false** или **true**. Инструкция **reads** в функции Valid() должна быть корректной, даже когда Valid() возвращает **false**. Эта проблема имеет простое решение: достаточно включить **this** в инструкцию **reads**. Вот правильное определение предиката Valid():

```
ghost predicate Valid()
  reads this, Repr
{
  this in Repr &&
  g in Repr && g.Valid() &&
  w in Repr && w.Valid()
}
```

Поскольку инструкция **reads** в Valid является такой особенной, рассмотрим подробнее ее зависимости.

- Инструкция **reads** ссылается на **this**.Repr. Это законно, потому что мы включили **this** в инструкцию **reads**.
- Тело предиката ссылается на **this**.Repr. Это законно по той же причине: потому что мы включили **this** в инструкцию **reads**.
- Тело предиката ссылается на **this**.g и **this**.w. Это законно, потому что мы включили **this** в инструкцию **reads**.
- Тело предиката зависит от g.Valid(), *если* два предыдущих конъюнкта имеют значение **true**. То есть, поскольку && является оператором, вычисляемым по короткой схеме, третий конъюнкт вычисляется, только если первый и второй конъюнкты вернули **true**. Второй конъюнкт – это g в Repr, поэтому если он возвращает **true**, то это означает, что Repr содержит объект g. Поскольку набор Repr указан в инструкции **reads** предиката Valid, тело этого предиката может читать поля любого объекта в Repr. Итак, третьему конъюнкту разрешено читать поля g.
- Тело предиката зависит от w.Valid(), *если* четыре предыдущих конъюнкта возвращают **true**. Поскольку эти четыре конъюнкта подразумевают, что w находится в Repr, пятому конъюнкту разрешено читать поля w.

Обратите внимание, что если поменять местами второй и третий конъюнкты, то тело Valid сможет читать поля g, даже если g не будет включен в

Repr, поэтому в такой ситуации верификатор сообщит об ошибке «инструкция **reads** неполная».

16.2.3. Реализация класса

Разобравшись с фреймами чтения и записи, перейдем к реализациям методов и других членов класса.

Вот конструктор:

```
constructor ()
  ensures Valid()
{
  g := new Grinder();
  w := new WaterTank();
  Repr := {this, g, w};
}
```

Обратите внимание, что Repr – это такое же поле, как и любое другое. (Да, оно объявлено призрачным, но верификатор не замечает этого отличия.) Это означает, что мы должны инициализировать Repr. Чтобы задать постусловие конструктора, т. е. Valid(), инициализируем Repr множеством {this, g, w}. Этим я подчеркиваю, что, даже притом что инструкции **reads** и **modifies** являются частью спецификаций, сам язык Dafny интерпретирует Repr как самое обычное поле. Он также не придает особого смысла предикату Valid(). Мы рассуждаем о программах с точки зрения инвариантов объектов и наборов представлений и определяем их с помощью предиката, которому дали имя Valid(), и призрачного поля с именем Repr.

Далее показана реализация предиката Ready, который позволяет кофеварке готовить одну или две порции кофе:

```
predicate Ready()
  requires Valid()
  reads Repr
{
  g.HasBeans && 2 <= w.Level
}
```

Наконец, реализуем два метода CoffeeMaker:

```
method Restock()
  requires Valid()
  modifies Repr
  ensures Valid() && Ready()
{
  g.AddBeans();
  w.Fill();
}
```

```
  method Dispense(double: bool) returns (c: Cup)
    requires Valid() && Ready()
    modifies Repr
    ensures Valid()
  {
    g.Grind();
    if double {
      w.Use(); w.Use();
    } else {
      w.Use();
    }
    c := new Cup();
  }
```

16.2.4. Тестовая программа

Я уже не раз подчеркивал важность создания тестовых программ, помогающих проверить удобство наших спецификаций. Давайте напишем такую программу и для нашей кофеварки. Вот один из простых способов проверить, как можно получить чашку кофе:

```
method CoffeeTestHarness() {
  var cm := new CoffeeMaker();
  cm.Restock(); // нарушает спецификацию modifies
  var c := cm.Dispense(true); // нарушает спецификацию modifies
}
```

Ай-яй-яй! Мы не смогли использовать нашу кофеварку даже для такого простого действа! Проблема, о которой сообщает верификатор, заключается в том, что вызов `Restock` может изменить объекты, но в спецификации `CoffeeTestHarness` о такой возможности ничего не говорится. В тестовом методе неявно присутствует пустая инструкция **`modifies`**, разрешающая изменять поля только вновь создаваемых объектов, т. е. объектов, созданных после входа в метод (эти правила обсуждались в разделе 16.0). Инструкция **`modifies`** в `Restock` сообщает нам, что этот метод может изменять поля любых объектов, перечисленных `cm.Repr`. Что содержит `cm.Repr` во время вызова `Restock`? Являются ли объекты в `cm.Repr` новыми? Текущие характеристики ничего не говорят о новизне `Repr`. Поскольку поле `Repr` будут использовать клиенты нашей кофеварки в своих рассуждениях, мы должны включить в спецификации `CoffeeMaker` некоторую информацию о `Repr`.

16.2.5. Технические характеристики Repr

Давайте посмотрим, как писать спецификации набора представлений.

Конструктор

Сначала обратим наше внимание на конструктор. Что мы можем сказать о Repr в постусловии? Разрабатывая тестовый код, мы узнали, что вызывающий код волнует, когда были выделены объекты в Repr. С этой целью добавим в конструктор второй версии CoffeeMaker следующее постусловие:

```
ensures fresh(Repr)
```

В разделе 14.0.3 мы уже кратко познакомились с предикатом **fresh**, применив его к массиву. Здесь мы передаем функции **fresh** набор объектов, говоря тем самым, что все объекты в этом наборе создаются заново. Добавив это постусловие в конструктор, мы устранили ошибку, обнаруживаемую верификатором в точке вызова Restock в нашей тестовой программе. Отлично!

Соблюдение условия свежести

Итак, первую ошибку мы устранили, но верификатор по-прежнему жалуется на вызов второго метода – Dispense, – утверждая, что он может изменить больше объектов, чем разрешено спецификацией в CoffeeTestHarness. И снова проблем заключается в недостатке информации о Repr. Опишу три решения.

Полное отсутствие изменений. Один из возможных способов указать влияние методов CoffeeMaker на Repr, – сообщить об отсутствие любых изменений при их выполнении. У нас есть два метода, Restock и Dispense, которые никак не изменяют объекты, перечисленные в Repr. Соответственно, мы можем добавить в них следующее постусловие:

```
ensures Repr == old(Repr)
```

Благодаря добавлению этих постусловий в спецификации наша небольшая тестовая программа наконец-то успешно верифицируется.

Неизменяемые поля. Альтернативный способ указать влияние методов на Repr – сообщить, что Repr никогда не изменяется после выполнения конструктора. Это означает, что поле Repr можно объявить *неизменяемым*:

```
ghost const Repr: set<object>
```

Поля (призрачные или компилируемые), не предназначенные для изменения, полезно объявить неизменяемыми, добавив спецификатор **const**. Это удержит нас от случайных попыток присвоить значения таким полям, а также избавит от необходимости писать постусловия, утверждающие, что поля не изменяются.

Свежие объекты. Два предыдущих способа указать влияние методов на Repr прекрасно подходят для нашего класса CoffeeMaker. Однако объявление поля Repr неизменяемым ограничивает реализацию класса. Например, представьте, что после создания объекта кофеварки возникнет необходимость заменить кофемолку новым объектом Grinder (например, после

каждой 1000 чашек кофе). В текущем классе `CoffeeMaker` нет такой необходимости, но подобная расширяемость даст будущим версиям класса возможность включать новые составляющие объекты в уже существующий агрегатный объект.

Третий и предпочтительный способ сообщить, что метод может изменять `Repr`, – добавить в постусловие указание на возможность добавлять объекты в `Repr`, причем только новые объекты, созданные внутри метода. Другими словами, на выходе из метода значение `Repr` может содержать любые или все объекты, содержавшиеся в `Repr` на входе в метод, и все дополнительные объекты, добавленные в `Repr` внутри метода. Вот как записать такую спецификацию:

```
ensures fresh(Repr - old(Repr))
```

Эта спецификация тоже обеспечивает успешную верификацию тестового кода.

Удаление, захват и освобождение составляющих объектов

Чаще всего составляющие объекты создаются и хранятся агрегатным объектом, который их использует. Так действуют агрегатные объекты, которые мы уже видели. Но как быть, если агрегатному объекту потребуется перестать использовать один из своих составляющих объектов? А если потребуется настроить агрегатный объект с помощью составляющего объекта, предоставленного клиентом? Или агрегатный объект должен будет отдать составляющий объект своему клиенту? Давайте рассмотрим пару примеров, чтобы увидеть, как можно скорректировать спецификации и код для таких ситуаций.

В качестве первого примера рассмотрим замену кофемолки в кофеварке. То есть вместо использования одной и той же кофемолки кофеварка может удалить одну кофемолку и добавить новую. Вот как я написал бы соответствующие спецификацию и код:

```
method ChangeGrinder()
  requires Valid()
  modifies Repr
  ensures Valid() && fresh(Repr - old(Repr))
{
  g := new Grinder();
  Repr := Repr + {g};
}
```

Это – *стандартная* спецификация для метода, вносящего изменения. Тело метода создает новый объект `Grinder`, а затем – запомните этот шаг! – добавляет его в `Repr`. Но что случилось с предыдущим значением `g`? Обычно предыдущий объект `Grinder` был доступен только через это поле конкретного экземпляра `CoffeeMaker`. При присваивании нового значения полю `g` предыдущий объект `Grinder` становится недоступным, и поэтому система

времени выполнения (в данном случае сборщик мусора в Dafny) в какой-то момент освободит память, занятую этим объектом.

Тело метода `ChangeGrinder` не удаляет предыдущий объект `Grinder` из `Repr`. Это типичная ситуация, особенно если предполагается, что этот объект станет недоступным в выполняющейся программе. Но мы могли бы удалить старый объект `g` из `Repr`, если в этом будет какой-то смысл. Это одна из причин, почему предикат, проверяющий выполнение свойств структуры данных, дает только нижнюю границу элементов в `Repr`.

В качестве второго примера рассмотрим метод, который принимает объект `Grinder` от клиента:

```
method InstallCustomGrinder(grinder: Grinder)
  requires Valid() && grinder.Valid()
  modifies Repr
  ensures Valid() && fresh(Repr - old(Repr) - {grinder})
{
  g := grinder;
  Repr := Repr + {g};
}
```

Чтобы выполнить первое постусловие `Valid()`, должны выполняться свойства объекта `grinder`. Это следует из предусловия. Что касается свежести объектов, добавленных в `Repr`, мы не можем задать типичное постусловие **`fresh`**`(Repr - old(Repr))`, потому что данный объект `Grinder` был создан до вызова `InstallCustomGrinder`. Поэтому спецификация вычитает `{grinder}` из множества свежих дополнений к `Repr`. Это постусловие гласит, что `grinder` *захватывается* объектом `CoffeeMaker`.

Точнее говоря, спецификация *позволяет* захватить `grinder`. Мы могли бы добавить постусловие `grinder` **`in`** `Repr`, но обычно это бесполезно, поэтому такое постусловие лучше опустить.

Упражнение 16.8

Напишите спецификацию и реализацию метода, который *освобождает* (т. е. исключает из `Repr` и возвращает) кофемолку из кофеварки, заменяя ее новой, созданной в этом же методе. Ваша спецификация должна обеспечивать успешную верификацию следующего тестового кода:

```
method RemoveGrinderHarness() {
  var cm := new CoffeeMaker();
  var grinder := cm.RemoveGrinder();
  cm.Restock();
  grinder.AddBeans();
}
```

16.2.6. Кратко об идиомах спецификаций

Программирование с использованием агрегатных объектов – непростая задача. В этом разделе я показал, как писать спецификации для агрегатных объектов, содержащих составляющие объекты простых типов, которые мы видели в разделах 16.0 и 16.1. Мы должны помнить о множестве деталей, поэтому определение спецификаций и верификация программ с агрегатными объектами требует терпения.

Два основных строительных блока – инварианты объектов и наборы представлений.

В идиоматических спецификациях, которые я использую, инвариант объекта определяется как предикат, который проверяет выполнение свойств структуры данных и которому по соглашению дается имя `Valid()`. Предикат, проверяющий выполнение свойств структуры, обычно используется в постусловиях каждого конструктора, в предусловиях каждой функции, а также в пред- и постусловиях каждого метода.

Наборы представлений в идиоматических спецификациях объявляются как призрачное поле `Repr`, а соответствующие объекты `Repr` указываются и поддерживаются как инварианты объектов (т. е. в предикате достоверности). Обычно методы имеют спецификацию фрейма **`modifies`** `Repr`, а функции – спецификацию фрейма **`reads`** `Repr`, за исключением функции `Valid()`, спецификация фрейма которой задается как **`reads this`**, `Repr`. Обычно конструкторы имеют постусловие **`fresh`**`(Repr)`, а методы – постусловие **`fresh`**`(Repr - `**`old`**`(Repr))`.

Согласно этой идиоме фреймом объекта является `Repr`. Поскольку `Repr` – это переменная, которая может динамически изменяться с течением времени (например, расширяться при включении вновь создаваемых объектов), я использую идиому спецификации, которая называется *динамическими фреймами* [69]. (Название языка программирования Dafny происходит от перестановки некоторых букв в словосочетании Dyn^{amic} f^{r}a^{mes} – динамические фреймы.)

Упражнение 16.9

В разделе 16.0 мы написали класс `ChecksumMachine` с простыми фреймами (в частности, **`reads this`** и **`modifies this`**). Добавьте в этот класс призрачную переменную `Repr`, которая будет обозначать набор представлений. Затем измените спецификации класса так, чтобы они соответствовали идиоме, представленной в этом разделе.

16.3. Сложные агрегатные объекты

Согласно определениям классов в предыдущем разделе объект `CoffeeMaker` является агрегатным объектом, содержащим составляющие его объекты `Grinder` и `WaterTank`.

Набор представлений объекта `CoffeeMaker` содержит не только объект `CoffeeMaker`, но и два составляющих его объекта. Мы определили набор представлений как призрачное поле `Repr` в классе `CoffeeMaker`. Поле `Repr` абстрагирует составляющие объекты, поэтому клиентам `CoffeeMaker` не нужно или не нужен подробный перечень объектов, составляющих `CoffeeMaker`. Как следствие, `Repr` является динамическим фреймом `CoffeeMaker`.

В предыдущем разделе я представил `Grinder` и `WaterTank` как объекты с простыми фреймами. Обычно такие объекты сами являются агрегатными. В этом разделе я вновь вернусь к примеру `CoffeeMaker`, но на этот раз с `Grinder` и `WaterTank`, имеющими динамические фреймы.

16.3.0. Составляющие объекты с динамическими фреймами

Вот как выглядят классы `Grinder` и `WaterTank` с точки зрения `CoffeeMaker`. Отличие от классов, представленных в разделе 16.2.0, заключается в добавлении поля `Repr` в спецификации фреймов каждого класса, в которых упоминается `Repr`, и постусловия о свежести `Repr`:

```
class Grinder {
  var HasBeans: bool
  ghost var Repr: set<object>
  ghost predicate Valid()
    reads this, Repr
  constructor ()
    ensures Valid() && fresh(Repr)
  method AddBeans()
    requires Valid()
    modifies Repr
    ensures Valid() && fresh(Repr - old(Repr)) && HasBeans
  method Grind()
    requires Valid() && HasBeans
    modifies Repr
    ensures Valid() && fresh(Repr - old(Repr))
}

class WaterTank {
  var Level: nat
  ghost var Repr: set<object>
  ghost predicate Valid()
  reads this, Repr
  constructor ()
    ensures Valid() && fresh(Repr)
  method Fill()
    requires Valid()
    modifies Repr
    ensures Valid() && fresh(Repr - old(Repr)) && Level == 10
```

```
  method Use()
    requires Valid() && Level != 0
    modifies Repr
    ensures Valid() && fresh(Repr - old(Repr)) &&
      Level == old(Level) - 1
}
```

16.3.1. Инвариант CoffeeMaker: подмножества

Набор представлений `CoffeeMaker` теперь включает целые наборы представлений составляющих его объектов. Поэтому изменим тело `Valid()`, как показано ниже:

```
    this in Repr &&
    g in Repr && g.Repr <= Repr && g.Valid() &&
    w in Repr && w.Repr <= Repr && w.Valid()
```

16.3.2. Два этапа выполнения конструкторов

Обновленный конструктор `CoffeeMaker` должен выглядеть как-то так:

```
  constructor ()
    ensures Valid() && fresh(Repr)
  {
    g := new Grinder();
    w := new WaterTank();
    Repr := {this, g, w} + g.Repr + w.Repr; // недопустимое использование
                                            // полей на первом этапе
}
```

Однако верификатор Dafny не принимает этот код. Конструкторы в классах Dafny выполняются в два этапа. На первом этапе полям объекта присваиваются конкретные значения соответствующих типов и задаются значения для неизменяемых полей. Если начальное значение поля не важно для вас и поле имеет тип, который компилятор знает, как инициализировать (так называемый *тип с автоматической инициализацией*), то такому полю можно не присваивать какое-либо значение в конструкторе. На этом первом этапе объект все еще находится в стадии создания, поэтому существуют ограничения на доступность некоторых операций с полями.

На первом этапе выполнения конструктора допускается присваивать значение полю `g` и использовать `g` в выражении инициализации `Repr`, но запрещено использовать выражение `g.Repr` (т. е. **`this`**`.g.Repr`). Вот один из вариантов, как можно переписать приведенный выше код, чтобы он не вызывал возмущения верификатора на первом этапе конструктора:

```
    var gg := new Grinder();
    var ww := new WaterTank();
```

```
g, w := gg, ww;
Repr := {this, g, w} + gg.Repr + ww.Repr;
```

Выражение `gg.Repr` позволяет избежать использования возможно неинициализированных полей **this**, поэтому оно считается допустимым на первом этапе.

Другой вариант – перенести присваивание назначения переменной `Repr` на второй этап. Поскольку типом `Repr` является множество (который является типом с автоматической инициализацией), компилятор сможет автоматически присвоить начальное значение `Repr` на первом этапе. Операция присваивания на втором этапе затем заменит произвольное начальное значение, предоставленное компилятором. Реализация второго этапа выполнения конструктора обычно остается пустой, но вы можете определить его, использовав специальную инструкцию **new;**, чтобы обозначить конец первого и начало второго этапа. На втором этапе выполнения конструктора все поля уже имеют значения соответствующих типов, поэтому их можно использовать без ограничений. Как результат, мы можем написать тело конструктора `CoffeeMaker` по-другому:

```
g := new Grinder();
w := new WaterTank();
new;
Repr := {this, g, w} + g.Repr + w.Repr;
```

При необходимости вы также можете присвоить некоторое значение `Repr` на первом этапе, а затем заменить или обновить это поле на втором этапе. Например, ниже показан еще один законный способ определить тело конструктора `CoffeeMaker`:

```
g := new Grinder();
w := new WaterTank();
Repr := {this, g, w};
new;
Repr := Repr + g.Repr + w.Repr;
```

16.3.3. Разделение наборов представлений

Предикат `Ready()` аналогичен описанному в разделе 16.2, но метод `Restock()` создает ряд новых проблем. Давайте рассмотрим их:

```
method Restock()
  requires Valid()
  modifies Repr
  ensures Valid() && fresh(Repr - old(Repr)) && Ready()
{
  g.AddBeans();
  w.Fill(); // нарушение предусловия; нарушение инструкции modifies
} // нарушение постусловия
```

Верификатор сообщает о нескольких ошибках, первая из которых – нарушение предусловия при вызове `Fill`. Первое (и второе, и третье...) столкновение с такой ситуацией может привести в замешательство. Поэтому позвольте мне подробно объяснить процесс выяснения причин происходящего.

Начнем с нарушения предусловия `Fill`. Для отладки добавим утверждение **`assert`** с этим условием непосредственно перед вызовом `Fill`:

```
g.AddBeans();
assert w.Valid(); // нарушение утверждения
w.Fill(); // нарушение инструкции modifies
```

Верификатор немедленно сообщает нам о возможном нарушении утверждения **`assert`**, но для любого случая, соответствующего условию в утверждении, предусловие `Fill` выполняется (т. е. сообщение о нарушении предусловия заменяется сообщением о нарушении утверждения). Это подтверждает, что мы правильно записали условие в утверждении. Имея перед глазами условие `w.Valid()`, нам будет проще понять, почему оно может дать сбой.

Допустимость резервуара для воды `w` должна зависеть от допустимости кофеварки. То есть мы ожидаем, что условие `w.Valid()` выполняется на входе в метод. Давайте проверим это с помощью верификатора:

```
assert w.Valid();
g.AddBeans();
assert w.Valid(); // нарушение утверждения
w.Fill(); // нарушение инструкции modifies
```

Верификатор подтверждает выполнение условия `w.Valid()` в начале метода, но не может подтвердить его после вызова `AddBeans`. Из этого следует, что вызов `AddBeans` влияет на `w.Valid()`. Это удивительно, поскольку, по нашему представлению, кофемолка должна работать независимо от резервуара для воды.

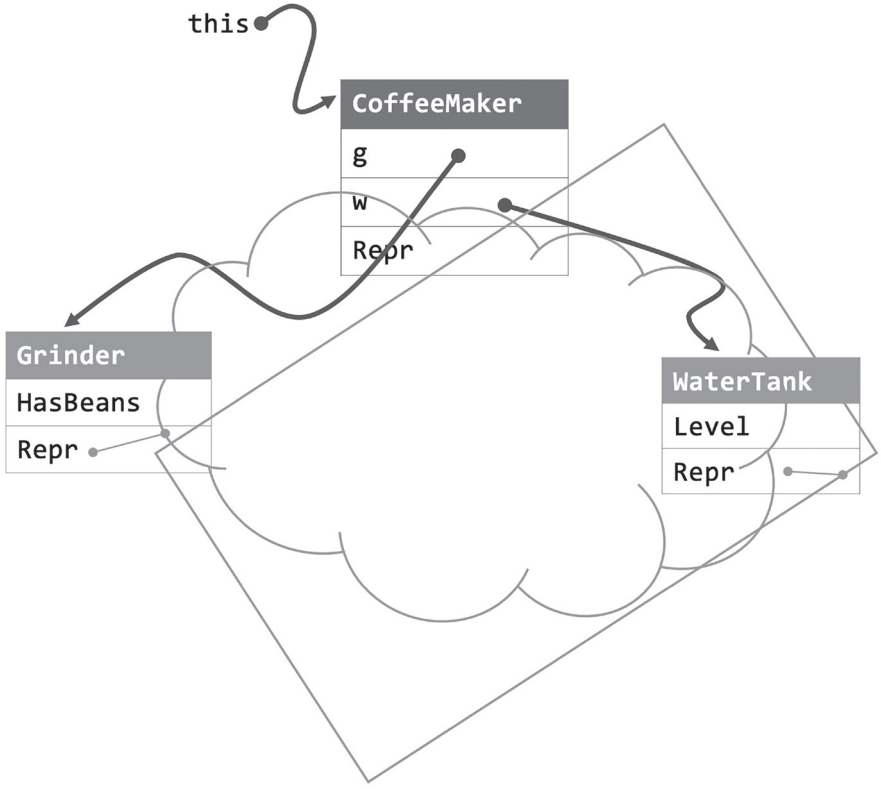
Причина во фреймах чтения и записи. Вызов `AddBeans` может изменить состояние объектов, если судить по инструкции **`modifies`** в его спецификации, а именно объектов в `g.Repr`. Функция `Valid` класса `WaterTank` зависит от состояния объектов в ее инструкции **`reads`**, а именно объектов в `w.Repr`. Таким образом, единственный способ, которым `g.AddBeans()` может повлиять на значение `w.Valid()`, – это пересечение между `g.Repr` и `w.Repr`. Давайте спросим об этом верификатора:

```
assert w.Valid();
assert g.Repr !! w.Repr; // нарушение утверждения
g.AddBeans();
assert w.Valid(); // нарушение утверждения
w.Fill(); // нарушение инструкции modifies
```

Оператор `!!` (имеющий более высокий приоритет, чем `&&` и `||`) говорит об отсутствии пересечений между его аргументами.

Новое утверждение **`assert`** вызывает ошибку верификации, показывая, что условие, которое должно выполняться всегда, еще не является частью наших спецификаций. Добавим это условие непересекаемости в инвариант `CoffeeMaker`. То есть добавим `g.Repr !! w.Repr` в тело `Valid` в классе `CoffeeMaker`.

К сожалению, добавленное нами условие непересекаемости не позволяет доказать выполнение условия `w.Valid()` после возврата из `g.AddBeans()`. Визуализация в виде диаграммы может помочь нам обдумать этот факт. На следующем рисунке показаны три известных нам объекта: кофеварка (`CoffeeMaker`), кофемолка (`Grinder`) и резервуар для воды (`WaterTank`). На нем набор `g.Repr` изображен в виде облака, а набор `w.Repr` – в виде прямоугольника.



Что мы знаем о наборах `g.Repr` и `w.Repr`? Если добавить следующие утверждения **`assert`** в начало метода `Restock`, то можно увидеть, как мало мы указали:

```
assert this !in g.Repr; // нарушение утверждения
assert g in g.Repr; // нарушение утверждения
assert w !in g.Repr; // нарушение утверждения
```

По сложившемуся у нас в головах представлению о `g.Repr` все три утверждения должны быть верны, но ни одно из этих условий мы не отразили в наших спецификациях. К сожалению, это распространенное явление – легко подразумевать какое-то свойство как само собой разумеющееся, но забыть отразить его в инвариантах или других спецификациях. В дополнение к свойству непересекаемости, которое мы выяснили выше, добавим в наши спецификации и эти три условия, чтобы обеспечить доказуемость утверждений.

Свойство **`this`** `!in g.Repr` необходимо, чтобы можно было положиться на неизменность значения **`this`**`.w` при вызове `g.AddBeans()`. Это свойство лучше всего включить в инвариант `CoffeeMaker`. Тогда тело предиката `Valid()` в классе `CoffeeMaker` будет выглядеть так:

```
this in Repr &&
g in Repr && g.Repr <= Repr && this !in g.Repr && g.Valid() &&
w in Repr && w.Repr <= Repr && this !in w.Repr && w.Valid() &&
g.Repr !! w.Repr
```

Врезка 16.1

Для двух множеств `A` и `B` выражение `A !! B` эквивалентно утверждению, что пересечение `A` и `B` – пустое, т. е.:

```
A * B == {}
```

Однако, помимо большей краткости, оператор `!!` имеет еще одно преимущество: он может принимать больше двух аргументов. Таким образом, он выражает, что все его аргументы попарно непересекающиеся. Например, если `C` – еще одно множество, то можно записать:

```
A !! B !! C
```

чтобы сообщить, что эти три множества попарно не пересекаются. То же условие с использованием пересечений записывается так:

```
A * B == {} && B * C == {} && C * A == {}
```

Я уже говорил выше, что объект всегда является частью своего собственного набора представлений. Поэтому в телах предикатов достоверности мы отражали это в `Repr`. Однако если экспортируемый набор включающего модуля скрывает тело `Valid` от клиентов, то клиенты не смогут использовать это свойство. Чтобы исправить ситуацию, вернемся к предикатам достоверности классов `Grinder` и `WaterTank` из раздела 16.2.0 и перепишем их так:

```
ghost predicate Valid()
  reads this, Repr
  ensures Valid() ==> this in Repr
```

Упоминание `Valid()` в собственном постусловии относится к значению, возвращаемому функцией (см. главу 6). Тело `Valid` по-прежнему скрыто от клиентов, но постусловие видно им. Соответственно, это постусловие будет частью наших предикатов, проверяющих выполнение свойств структуры данных, поэтому давайте сразу же добавим его в `Valid` класса `CoffeeMaker`.

Упражнение 16.10

> Легко опечататься и записать **`ensures this in`** `Repr` в предикате, проверяющем выполнение свойств структуры данных. Что означает это постусловие и чем оно отличается от постусловия, которое я написал выше?

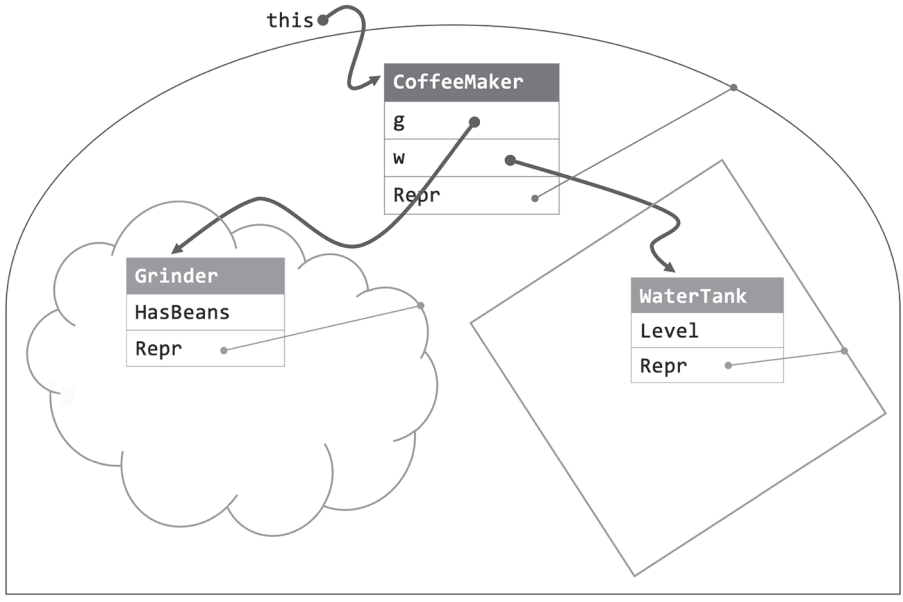
С новым постусловием в предикатах, проверяющих выполнение свойств структур данных, инвариантом

```
this !in g.Repr
```

для любого составляющего объекта `g` и инвариантом непересекаемости

```
g.Repr !! w.Repr
```

составляющих объектов `g` и `w` мы можем более точно изобразить наши объекты:



Теперь, как показывает эта диаграмма, мы точно знаем, что **this** отсутствует и в g.Repr, и в w.Repr (из-за инварианта CoffeeMaker), мы знаем, что g находится в g.Repr, а w – в w.Repr (это следует из постусловий предикатов, проверяющих выполнение свойств структур данных в Grinder и WaterTank, а также из того факта, что g.Valid() и w.Valid() являются частью инварианта CoffeeMaker), и мы знаем, что g.Repr и w.Repr не перекрываются (из-за инварианта непересекаемости в CoffeeMaker). На этой диаграмме я также изобразил **this**.Repr, содержащий **this**, и g.Repr и w.Repr.

Как эти улучшения влияют на наше доказательство корректности Restock? Вот что мы имеем на данный момент:

```
method Restock()
  requires Valid()
  modifies Repr
  ensures Valid() && fresh(Repr - old(Repr)) && Ready()
{
  g.AddBeans();
  w.Fill();
} // нарушение постусловия
```

Недоказанное постусловие – это Valid(), точнее верификатор выделяет подмножество условий g.Repr <= Repr и w.Repr <= Repr как недоказуемые. Но если вызовы AddBeans и Fill приводят к расширению g.Repr и w.Repr, то нам, конечно, нужно расширить и this.Repr. Это легко сделать, добавив

```
Repr := Repr + g.Repr + w.Repr;
```

в конец тела Restock. Теперь Restock успешно проходит верификацию!

16.3.4. Обновление Dispense

После проделанной нами работы по спецификации отношений между различными фреймами при верификации метода Dispense возникает только одна проблема. И она нам хорошо известна: не обеспечиваются отношения между подмножествами в наборе представлений. Исправляется эта проблема так же, как и в случае с Restock: нужно изменить Repr так, чтобы отразить любые изменения g.Repr и w.Repr. Добавив

```
Repr := Repr + g.Repr + w.Repr;
```

мы обеспечим благополучную верификацию CoffeeMaker.

Упражнение 16.11

Обновите методы ChangeGrinder и InstallCustomGrinder из раздела 16.2.4 и метод RemoveGrinder из упражнения 16.8, чтобы обеспечить их совместимость с новой версией CoffeeMaker, разработанной в этом разделе.

16.4. Итоги

При написании спецификаций для агрегатных объектов требуется помнить о множестве деталей. Я постарался подробно объяснить каждую часть разработанной спецификации. Самое замечательное, что показанные здесь приемы применимы в аналогичных формах и к другим агрегатным объектам. Фактически спецификации настолько схожи по форме, что их можно назвать идиоматическими.

А теперь позвольте мне подвести итог этим идиоматическим спецификациям динамических фреймов, описав типичную идиому написания спецификаций для агрегатных объектов, которая пригодится вам при реализации ваших классов.

16.4.0. Набор представлений

Для хранения набора представлений класс использует призрачную переменную:

```
ghost var Repr: set<object>
```

16.4.1. Инвариант

Класс также определяет призрачный предикат, который проверяет выполнение свойств структуры данных и задает инвариант объекта. Он имеет форму:

```
ghost predicate Valid()
  reads this, Repr
  ensures Valid() ==> this in Repr
{
  this in Repr &&
  // ...
}
```

Для каждого поля `a`, представляющего объект с простым фреймом, условие, что выполняются свойства структуры данных, также включает:

```
  a in Repr && a.Valid()
```

А для каждого поля `b`, представляющего объект с динамическим фреймом, условие, что выполняются свойства структуры данных, также включает:

```
  b in Repr && b.Repr <= Repr && this !in b.Repr && b.Valid()
```

Кроме того, частью предиката, проверяющего выполнение свойств структуры данных, являются различные условия непересекаемости. Для составляющих объектов `a0` и `a1` с простыми фреймами и составляющих

объектов `b0` и `b1` с динамическими фреймами условия непересекаемости можно выразить формулой:

```
a0 != a1 &&
{a0, a1} !! b0.Repr !! b1.Repr
```

16.4.2. Конструктор

Конструктор обеспечивает выполнение свойств создаваемого объекта и актуальность набора представлений:

```
constructor ()
  ensures Valid() && fresh(Repr)
```

Для составляющих объектов `a` и `b` с простым и динамическим фреймом соответственно тело конструктора инициализирует `Repr` инструкцией:

```
Repr := {this, a, b} + b.Repr;
```

Эта инструкция должна определяться во втором этапе выполнения конструктора, т. е. после **`new;`**.

16.4.3. Функции

Функции (кроме `Valid`, которая определяет `Repr` для допустимых объектов) должны содержать следующие предусловие и фрейм чтения:

```
function F(x: X): Y
  requires Valid()
  reads Repr
```

16.4.4. Методы

Мутирующий метод (метод, изменяющий текущее состояние) включает `Repr` в фрейм записи, требует и поддерживает выполнение свойств, а также гарантирует, что в `Repr` будут добавляться только объекты, вновь созданные внутри метода:

```
method M(x: X) returns (y: Y)
  requires Valid()
  modifies Repr
  ensures Valid() && fresh(Repr - old(Repr))
```

Иногда методы (вместо функций) используются для вычисления некоторых результатов без изменения состояния агрегатного объекта. В спецификациях таких *методов запроса* не используются инструкции **`modifies`** и **`ensures`**.

Упражнение 16.12

Опробуйте код, представленный в этой главе, вводя его в IDE, чтобы увидеть, какие ошибки возникают в тех местах, где наши специфи-

кации или код оказались недостаточно полными. Это поможет вам познакомиться с конкретными сообщениями об ошибках и лучше понять, как реагировать на подобные сообщения в ваших собственных программах.

Примечания

Динамические фреймы были изобретены Яннисом Кассиосом (Yannis Kassios) [69]. Первый простой верификатор, поддерживавший их, использовал призрачные функции для оценки наборов представлений [116]. В отличие от него для анализа динамических фреймов в Dafny используется призрачное поле (`Repr`). Это обстоятельство требует от программиста явно обновлять призрачное поле, но дает автоматизированному верификатору больше информации о том, когда на значения функций абстракции (таких как `Valid()`) не влияют несвязанные обновления состояния.

Для верификации программ, обновляющих состояние кучи, существуют альтернативные методы. В обзорной статье по поведенческим спецификациям [59] такие методы классифицируются по способу определения наборов представлений.

- Наборы *явных* представлений определяются программистом. Примерами могут служить динамические фреймы и инструмент Dafny.
- Наборы *неявных* представлений задаются предикатами специализированной логики. Примерами могут служить логика разделения [114, 103] и инструменты VeriFast [64] и Viper [98].
- *Предопределенные* наборы представлений получаются из предопределенных топологий кучи (обычно кучи имеют древовидную топологию). Примерами могут служить отношения *владения* в Spec# [83] и VCC [30], а также регионы в WhyML [20].

Наборы явных представлений обеспечивают гибкость спецификаций (как мы увидим в примере с итератором в разделе 17.3, где наборы представлений могут пересекаться). Еще одно преимущество динамических фреймов: они не требуют слишком много логики – идиома, представленная в этой книге, требует только наличия инструкций **`modifies`** и **`reads`**, наборов объектов и призрачных переменных.

Специализированные приемы, в свою очередь, могут обеспечить большую упорядоченность. Например, стандартные спецификации (`Repr`, `Valid()`, ...), потребовавшиеся в примере с `CoffeeMaker` в разделе 16.3, в Spec# обычно заменяются простым объявлением **`invariant`** в классе и аннотацией `[Rep]` перед каждым полем, содержащим составляющий объект [83]. Другой пример: методы, использующие наборы неявных представлений, не требуют наличия инструкции **`modifies`** и предикатов **`fresh`**. Кроме того, более строгое отслеживание эффектов чтения и записи часто лучше подходит для

реализации параллельных вычислений (см., например, [25, 84, 65]). Дополнительную информацию ищите в обзорном документе [59].

В этой (и следующей) главе, а также в главе 10 инварианты структуры данных записываются с помощью предикатов, проверяющих выполнение свойств структуры данных. Этот подход проясняет, где должны храниться инварианты, и оказывается довольно гибким, поскольку для верификации вспомогательных методов, в которых инвариант не выполняется на входе или выходе, достаточно просто исключить из спецификации предикат, проверяющий выполнение свойств структуры данных. Этот подход уходит корнями в *парадигму состояния/валидности* механизма статической проверки Extended Static Checker в Modula-3 [39].

В типичных случаях проще использовать выделенное объявление **`invariant`** в классе. Многие языки, включая Eiffel [89], JML [66] и Spec# [83], напрямую поддерживают такие объявления. Увы, в агрегатных объектах простота таких объявлений может оказаться обманчивой (см., например, [97, 112]).

Глава 17

Динамические структуры данных

В этой главе я познакомлю вас с четырьмя практическими примерами, в которых мы определим и верифицируем программы со структурами данных, выделяемыми в динамической памяти (в куче). Мы увидим некоторые агрегатные объекты, наборы представлений которых могут меняться в процессе выполнения. Два примера реализуют массивы со специальными свойствами, а два других реализуют ассоциативный массив с целочисленными ключами и итератор по таким массивам.

17.0. Ленивая инициализация массивов

Получение и изменение элемента массива по заданному индексу может выполняться за постоянное время. Выделение памяти, необходимой для массива, тоже может выполняться за постоянное время, если механизм распределения памяти имеет непрерывную область неиспользуемой памяти и может зарезервировать в ней необходимый фрагмент. Что можно сказать о создании массива (скажем, размера `n`) и инициализации каждого его элемента заданным значением (скажем, `d`)? Самый простой способ сделать это – использовать время, линейно-пропорциональное `n`. Однако при наличии некоторого дополнительного хранилища создание инициализированного массива тоже можно выполнить за постоянное время. Именно этим мы и займемся в этом разделе. Точнее, мы построим абстракцию массива, использующую постоянное количество целочисленных операций для создания, получения и изменения любого элемента массива.

17.0.0. Базовая спецификация

Вот спецификация класса, который мы реализуем:

```
class LazyArray<T(0)> {
  ghost var Elements: seq<T>
  ghost var Repr: set<object>
  ghost predicate Valid()
    reads this, Repr
    ensures Valid() ==> this in Repr
  constructor (length: nat, initial: T)
    ensures Valid() && fresh(Repr)
    ensures |Elements| == length
    ensures forall i :: 0 <= i < |Elements| ==>
      Elements[i] == initial
  function Get(i: int): T
    requires Valid() && 0 <= i < |Elements|
    reads Repr
    ensures Get(i) == Elements[i]
  method Update(i: int, x: T)
    requires Valid() && 0 <= i < |Elements|
    modifies Repr
    ensures Valid() && fresh(Repr - old(Repr))
    ensures Elements == old(Elements)[i := x]
}
```

Это объявление класса следует той же базовой структуре, к которой мы пришли в конце предыдущей главы, но я должен отметить несколько деталей.

Первая деталь – это поле `Elements`. Как и поле `Repr`, которое мы использовали здесь и в предыдущей главе, `Elements` – это *поле абстракции*, играющее ту же роль, что и функции абстракции в разделах 9.3.1 и 10.0.0. Это поле используется в спецификациях для описания действий, выполняемых классом. Поскольку поле является призрачным, класс может реализовать хранение массива любым способом. Мы могли бы определить `Elements` как функцию, но, так как мы имеем дело с изменяемыми структурами данных, определение `Elements` в виде поля упростит доказательство отсутствия взаимовлияния несвязанных изменений в куче.

Вторая деталь заключается в параметре `initial`, который принимает конструктор и использует для инициализации каждого элемента `Elements`. Другими словами, призрачное поле `Elements` быстро инициализируется начальными значениями `initial`.

Третья деталь заключается в использовании выражения *обновления последовательности* в постусловии метода `Update`. Для любого индекса `i` в последовательности `q` и любого значения `x` того же типа, что и элементы `q`, выражение `q[i := x]` обозначает последовательность, подобную `q`, за исключением того, что для индекса `i` оно возвращает `x`.

Наконец, самая малопонятная деталь заключается в параметре типа `T` с характеристикой `(0)`, который принимает класс. Эта характеристика ограничивает `T` типами, экземпляры которых могут инициализироваться ком-

пилятором значениями по умолчанию. Такие типы известны как *типы с автоматической инициализацией*. Для целей настоящего примера нам достаточно знать, что ограничение `T` типами с автоматической инициализацией позволяет создать массив значений `T` без необходимости явно инициализировать его элементы (что потребовало бы линейного времени, и мы можем создать наш `LazyArray` за постоянное время).

Упражнение 17.0

Напишите модуль на основе класса и определите соответствующий экспортируемый набор.

17.0.1. Тестовая программа

Как обычно, чтобы убедиться в корректности нашей спецификации, напишем небольшую тестовую программу. Тесты, выполняемые с помощью следующего метода, используют инициализацию, предоставляемую конструктором, и проверяют возможность многократного изменения элементов без влияния на второй объект `LazyArray`:

```
method LazyArrayTestHarness() {
  var a := new LazyArray(300, 4);
  assert a.Get(100) == a.Get(101) == 4;
  a.Update(100, 9);
  a.Update(102, 1);
  assert a.Get(100) == 9 && a.Get(101) == 4 && a.Get(102) == 1;

  var b := new LazyArray(200, 7);
  assert a.Get(100) == 9 && b.Get(100) == 7;
  a.Update(100, 2);
  assert a.Get(100) == 2 && b.Get(100) == 7;
}
```

Он успешно верифицируется, что дает нам некоторую уверенность в верности нашей спецификации.

17.0.2. Неизменяемое состояние

Прежде чем продолжить работу над реализацией, я хочу представить полезный прием определения спецификации. В нашем классе есть только один метод `Update`, и из его спецификации видно, что длина `Elements` никогда не меняется. При желании мы можем ввести имя для этой длины, скажем `N`. Это позволит нам сократить выражение `|Elements|` до `N`. Поскольку длина никогда не меняется, то `N` можно объявить неизменяемым полем:

```
  ghost const N: nat
```

Это объявление сразу дает понять, что длина никогда не меняется, и устраняет необходимость писать спецификацию типа, такую как:

```
|Elements| == |old(Elements)|
```

Связь между `N` и `|Elements|` указывается как инвариант объекта, т. е. как конъюнкт в теле предиката, проверяющего выполнение свойств структуры данных. Мы должны сообщить клиентам об этой связи, но сохранить в тайне другие детали тела `Valid`, поэтому поместим это условие в постусловие `Valid`:

```
ghost predicate Valid()
  reads this, Repr
  ensures Valid() ==> this in Repr && N == |Elements|
```

Как видите, введение неизменяемого призрачного поля и приравнивание его к выражению – это довольно удобный способ показать, что выражение никогда не изменяется.

Пока мы не закончили со спецификацией и недалеко ушли от **`const`**, внесем еще одно изменение. Как оказывается, набор представлений, который мы будем использовать для `LazyArray`, определяется при создании и никогда не меняется в дальнейшем. По этой причине мы можем объявить `Repr` неизменяемым полем:

```
ghost const Repr: set<object>
```

Такое обязательство не изменять набор представлений поможет нам сэкономить на спецификации свежести `Repr` в методах (см. раздел 16.2.4).

Упражнение 17.1

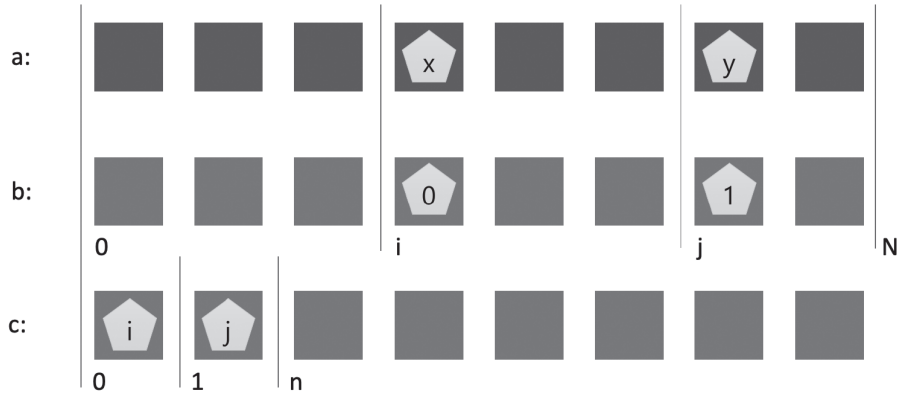
Обновите экспортируемый набор из упражнения 17.0, чтобы учесть изменения в классе, которые мы только что внесли.

17.0.3. Базовая структура данных

Для хранения значений элементов, установленных с помощью `Update`, мы будем использовать один базовый массив – `a`. Поскольку конструктор не имеет времени на инициализацию элементов массива `a`, нам потребуется применить смекалку, чтобы реализовать `Get`, выполняющийся за постоянное время.

Вот что мы сделаем. В дополнение к `a` мы используем еще два массива целых чисел, `b` и `c`, и одно целое число `n`. Целое число `n` будет хранить количество различных элементов, измененных функцией `Update`, а также количество инициализированных крайних левых элементов в `c`. Чтобы зафиксировать факт изменения `i`-го элемента, мы присвоим элементу `b[i]` число в диапазоне от `0` до `n` (т. е. `b[i]` будет хранить индекс элемента в `c`, который мы инициализировали), а в `c` сохраним `i`. Если `Get` будет вызван для получения значения неинициализированного элемента в `a`, мы вернем значение `initial`, переданное конструктору и хранящееся в поле `default`.

Все это иллюстрирует следующая диаграмма:



На диаграмме показано состояние для случая, когда `n` равно `2`. Здесь изменены 2 элемента в `a`: `a[i]` получил значение `x`, а `a[j]` – значение `y`. Соответствующие элементы в `b` содержат индексы в `c`: `b[i]` содержит `0` и `b[j]` содержит `1`. Элементы с этими индексами в `c` ссылаются на соответствующие элементы в `b`: `c[0]` хранит `i` и `c[1]` хранит `j`. Таким образом, мы имеем `c[b[i]] == i` и `c[b[j]] == j`.

Описанная ситуация не может сложиться случайно для неинициализированных значений. Если элемент `a[k]` не был инициализирован, то значит, элемент `b[k]` тоже не был инициализирован. Если значение `b[k]` не находится в диапазоне от `0` до `n`, то очевидно, что `b[k]` не был инициализирован, и это говорит нам о том, что элемент `a[k]` тоже не был инициализирован. А если неинициализированное значение `b[k]` содержит индекс инициализированного элемента в `c`, то мы все равно можем обнаружить, что `b[k]` не был инициализирован, потому что элемент `c[b[k]]` не будет равен `k`.

Вот определения всех этих полей в нашей реализации:

```
const default: T
const a: array<T>
const b: array<int>
const c: array<int>
var n: int
```

Следующий предикат позволит нам определить, был ли инициализирован индекс `i`:

```
predicate IsInitialized(i: int)
  requires 0 <= i < |Elements| == b.Length == c.Length
  requires 0 <= n <= |Elements|
  reads this, b, c
{
  0 <= b[i] < n && c[b[i]] == i
}
```

Обратите внимание, что в этом предикате отсутствуют объявления:

```
requires Valid()
reads Repr
```

типичные для функций, которые видят клиенты. Вместо этого предусловие включает только условия, необходимые для четкого определения `IsInitialized`. Аналогично фрейм чтения `IsInitialized` задается непосредственно в терминах объектов реализации. Это вполне допустимо, потому что `IsInitialized` является частью реализации и не экспортируется клиентам.

Фактически мы *должны* были объявить предикат без упоминания `Valid()`, поскольку собираемся использовать `IsInitialized` в определении `Valid()`. Центральная часть инварианта объекта – это следующая связь между нашими многочисленными полями:

```
forall i :: 0 <= i < N ==>
  Elements[i] == if IsInitialized(i) then a[i] else default
```

При таком использовании `IsInitialized` внутри `Valid` мы не можем позволить `IsInitialized` ссылаться на `Valid`, потому что тогда не сможем доказать завершимость между взаимозависимыми предикатами.

17.0.4. Инвариант объекта

Инвариант объекта `LazyArray` включает в себя множество условий – и инвариантов классов, и инвариантов фрейма. Как обычно, запишем их в теле `Valid()`. Инварианты фрейма гласят, что массивы `a`, `b` и `c` являются частью набора представлений и различны:

```
  ghost predicate Valid()
    reads this, Repr
    ensures Valid() ==> this in Repr && N == |Elements|
  {
    this in Repr &&
    a in Repr && b in Repr && c in Repr &&
    a != b && b != c && c != a &&
```

Затем утверждается, что длины массивов совпадают с длиной `Elements` в `LazyArray`, и указывается, что `N` равно этой длине:

```
    N == |Elements| == a.Length == b.Length == c.Length &&
```

Длина (`n`) инициализированной начальной части массива `c` находится в диапазоне от `0` до `N`:

```
    0 <= n <= N &&
```

И наконец, центральная часть инварианта объекта, которая связывает поле абстракции `Elements` с компилируемыми полями:

```
    (forall i :: 0 <= i < N ==>
      Elements[i] == if IsInitialized(i) then a[i] else default)
  }
```

17.0.5. Реализация

Реализация конструктора выглядит просто:

```
  constructor (length: nat, initial: T)
    ensures Valid() && fresh(Repr)
    ensures N == length && forall i :: 0 <= i < N ==>
      Elements[i] == initial
  {
    N := length;
    Elements := seq(length, _ => initial);
    default := initial;
    a, b, c := new T[length], new int[length], new int[length];
    n := 0;
    Repr := {this, a, b, c};
  }
```

С учетом центральной части инварианта реализация функции `Get` также получилась простой:

```
  function Get(i: int): T
    requires Valid() && 0 <= i < N
    reads Repr
    ensures Get(i) == Elements[i]
  {
    if IsInitialized(i) then a[i] else default
  }
```

Метод `Update` тоже использует предикат `IsInitialized` и должен соответствовать центральной части инварианта. При достаточном везении реализация этого метода тоже получится простой:

```
  method Update(i: int, x: T)
    requires Valid() && 0 <= i < N
    modifies Repr
    ensures Valid() && Elements == old(Elements)[i := x]
  {
    if !IsInitialized(i) {
      b[i], c[n] := n, i; // выход индекса за границы массива
      n := n + 1;
    }
    a[i] := x;
    Elements := Elements[i := x];
  }
```

Обратите внимание, что `Update` изменяет не только компилируемую структуру данных, но и поле абстракции `Elements`. Это необходимо для сохранения инварианта объекта и гарантии выполнения постусловия.

К счастью или нет, но метод `Update` не верифицируется. Верификатор сообщает о возможном выходе индекса за пределы массива (`c[n]`). Инвариант объекта сообщает нам, что `0 <= n <= N == c.Length`, но это условие оставляет возможность, что `n` может быть равно `c.Length`. Нам нужно доказать – себе и верификатору, – что `n < N` выполняется всегда, когда существует индекс `i`, соответствующий инициализируемому элементу.

Вот как мы разрешим эту ситуацию. Во-первых, сформулируем лемму, определяющую необходимое нам свойство (и которое, по нашему мнению, выполняется):

```
lemma ThereIsRoom(i: nat)
  requires Valid() && 0 <= i < N && !IsInitialized(i)
  ensures n < N
```

Во-вторых, вызовем эту лемму в `Update` непосредственно перед присваиванием `c[n]`:

```
  if !IsInitialized(i) {
    ThereIsRoom(i);
    b[i], c[n] := n, i;
```

После этого верификатор перестает жаловаться на возможный выход индекса за пределы допустимого диапазона. И в-третьих, докажем лемму.

Докажите лемму. Да, о том, что…

17.0.6. Доказательство, что в массиве есть место

Мы легко можем доказать самим себе, что в `c` есть место (т. е., что `n < c.Length`), когда в `a` есть неинициализированный элемент (т. е. когда для некоторого `i` выполняется условие `!IsInitialized(i)`). Основные рассуждения выглядят следующим образом (в этих выражениях две вертикальные черты по краям обозначают мощность множества, а вертикальная черта в середине является частью выражения, генерирующего множество):

```
  n
== // n равно числу инициализированных элементов в a
  |set j | 0 <= j < N && IsInitialized(j)|
<   // !IsInitialized(i), поэтому i входит в множество ниже,
    // но не в множество выше
  |set j | 0 <= j < N|
==  // это множество имеет N элементов
  N
```

К сожалению, верификатору требуется дополнительная помощь для доказательства каждого из этих шагов. Давайте рассмотрим каждый шаг подробнее и сделаем это в обратном порядке, снизу вверх.

Множества натуральных чисел

Индукция – самый простой способ доказать, что множество целых чисел `set j | 0 <= j < k` имеет `k` элементов. Это можно сделать с помощью рекурсивной функции. Я предлагаю написать эту функцию на самом верхнем уровне модуля, вне класса `LazyArray`, но ее можно также сделать членом класса. (А еще лучше собрать подобные функции и леммы в библиотечный модуль, содержащий полезный набор свойств.)

```
ghost function Upto(k: nat): set<int>
  ensures forall i :: i in Upto(k) <==> 0 <= i < k
  ensures |Upto(k)| == k
{
  if k == 0 then {} else Upto(k - 1) + {k - 1}
}
```

Первое постусловие функции определяет состав получаемого множества, а второе – его мощность.

Мощности собственных подмножеств

Верификатор знает лишь ограниченное количество свойств множеств, поэтому мы должны помочь ему. Нам нужно следующее свойство: если множество `u` является собственным подмножеством множества `U`, то `|u| < |U|`. Сформулируем и докажем это свойство следующим образом:

```
lemma SetCardinalities(u: set<int>, U: set<int>, x: int)
  requires u <= U
  ensures |u| <= |U|
  ensures x !in u && x in U ==> |u| < |U|
{
  if u == {} {
  } else {
    var y :| y in u;
    SetCardinalities(u - {y}, U - {y}, x);
  }
}
```

Инструкция «присвоить такое значение, что» `y :| y in u` (см. раздел 13.7.5) присваивает `y` произвольное значение, удовлетворяющее предикату `y in u`.

Мощность множества инициализированных элементов

Чтобы соотнести `n` с мощностью множества инициализированных элементов, можно позволить каждому объекту `LazyArray` поддерживать этот набор как призрачное поле и задавать инвариант его мощности. В таком случае нам останется только организовать одновременное изменение этого множества и `n`.

Для этой цели введем поле `s`:

```
ghost var s: set<int>
```

Чтобы описать значение s и связать его мощность с n, добавим два конъюнкта в предикат, проверяющий выполнение свойств структуры данных:

```
s == (set i | 0 <= i < N && IsInitialized(i)) &&
n == |s|
```

Чтобы изначально установить соответствие этим условиям, заменим присваивание `n := 0;` в конструкторе на:

```
s, n := {}, 0;
```

А чтобы обеспечить соответствие двум условиям как инвариантам объекта, заменим присваивание `n := n + 1;` в методе Update на:

```
s, n := s + {i}, n + 1;
```

Доказательство леммы

Настроив эти три ингредиента, напишем доказательство леммы ThereIsRoom:

```
lemma ThereIsRoom(i: nat)
  requires Valid() && 0 <= i < N && !IsInitialized(i)
  ensures n < N
{
  var S := Upto(N);
  SetCardinalities(s, S, i);
}
```

На этом разработка и верификация класса LazyArray завершены.

17.1. Расширяемый массив

В этом разделе мы напишем класс, представляющий расширяемый массив. Помимо конструктора, класс поддерживает обычные операции получения и изменения элементов, а также операцию Append, которая увеличивает длину массива и помещает заданное значение в новый элемент в конце массива. Мы разработаем операцию Append так, чтобы она не копировала предыдущие элементы массива. Это обеспечит логарифмическое время выполнения каждой операции нашего класса в зависимости от количества сохраненных элементов.

17.1.0. Спецификация

Определим класс ExtensibleArray, параметризованный типом элемента. Вот как будут видеть клиенты наш класс, объявленный с использованием стандартной идиомы спецификации с одним абстрактным полем (Elements):

```
class ExtensibleArray<T> {
  ghost var Elements: seq<T>
  ghost var Repr: set<object>
  ghost predicate Valid()
    reads this, Repr
    ensures Valid() ==> this in Repr
  constructor ()
    ensures Valid() && fresh(Repr) && Elements == []
  function Get(i: int): T
    requires Valid() && 0 <= i < |Elements|
    ensures Get(i) == Elements[i]
    reads Repr
  method Update(i: int, t: T)
    requires Valid() && 0 <= i < |Elements|
    modifies Repr
    ensures Valid() && fresh(Repr - old(Repr))
    ensures Elements == old(Elements)[i := t]
  method Append(t: T)
    requires Valid()
    modifies Repr
    ensures Valid() && fresh(Repr - old(Repr))
    ensures Elements == old(Elements) + [t]
}
```

Обратите внимание, что в постусловии метода `Append` используется стандартный прием:

```
fresh(Repr - old(Repr))
```

поскольку ожидается, что набор представлений будет меняться со временем.

Упражнение 17.2

Напишите тестовую программу для класса `ExtensibleArray`.

17.1.1. Структура данных

Основная идея заключается в хранении элементов расширяемого массива в массивах фиксированной длины. Я буду использовать массивы с длиной `256`. Существует «передовой» (front) массив, куда помещаются самые последние добавляемые элементы. После заполнения передовой массив переносится в «хранилище» (depot) – расширяемую коллекцию массивов. В качестве структуры данных для организации такого хранилища мы будем использовать расширяемый массив массивов! То есть `ExtensibleArray<T>` может содержать `ExtensibleArray<array<T>>`. Эту конструкцию иллюстрирует диаграмма на рис. 17.0.

При разработке этого проекта мы должны принять несколько важных решений, и лучше подумать о них до того, как мы приступим к написанию инварианта объекта, не говоря уже о реализации. Для выбора между альтернативными дизайнами может потребовать написать несколько пробных версий, поэтому при разработке собственных программ вы можете несколько раз менять объявления и инварианты, прежде чем остановиться на чем-то, что вас устраивает.

Типы, допускающие значение null

Одно из решений (достаточно распространенное, чтобы обсудить его в отдельном подразделе) касается представления пустого хранилища. После создания `ExtensibleArray<T>` мы можем столкнуться с проблемой создания объекта `ExtensibleArray<`**`array`**`<T>>` для хранилища, поскольку это сделает конструктор `ExtensibleArray` рекурсивным. У нас нет достойной оценочной функции завершимости для этой рекурсии, потому что если каждый вызов конструктора будет выполнять рекурсивный вызов, тогда у нас действительно получится бесконечная рекурсия.

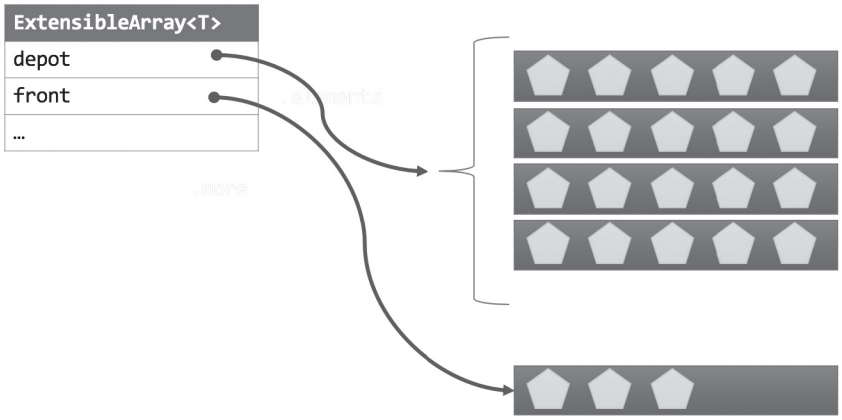


Рис. 17.0. Структура данных расширяемого массива

Вместо этого будем создавать объект хранилища `ExtensibleArray<`**`array`**`<T>>` только тогда, когда он действительно понадобится. В таком случае поле хранилища должно иметь такой тип, чтобы избежать необходимости размещать объект. Например, мы могли бы объявить такой тип:

```
Option<ExtensibleArray<array<T>>>
```

где `Option` – это следующий тип данных:

```
datatype Option<A> = None | Some(A)
```

Однако объявлять такой тип совершенно необязательно, потому что каждый класс `C` в Dafny уже порождает *два* типа: тип с именем `C` и тип с

именем C?. Значение типа C является ссылкой на экземпляр класса C. Мы использовали этот тип (для ссылок на массивы и объекты) во всех главах этой части книги. Значение типа C? является либо ссылкой, либо специальным значением **null**.

Обратите внимание, что знак вопроса ? является частью имени типа, а не оператором, применяемым к типу. То, что я сказал здесь о классах, верно и для типов массивов. Например, значение типа array?<T> является либо ссылкой на массив элементов типа T, либо специальным значением **null**.

Итак, объявив поле хранилища depot с типом

```
ExtensibleArray?<array<T>>
```

мы можем инициализировать это поле значением **null** и не вызывать конструктор рекурсивно.

Другие решения по дизайну

Второе решение, которое необходимо принять, – делать ли передовой массив частью хранилища, или отделить их друг от друга. Давайте разделим их. Такая организация выглядит понятнее, поскольку мы можем считать, что каждый массив в хранилище заполнен до отказа и содержит 256 элементов. И к тому же нам не придется создавать объект хранилища, пока мы не заполним самый первый массив.

Еще одно важное решение – добавлять ли передовой массив в хранилище сразу после заполнения последнего 256-го элемента, или подождать, пока будет сделан еще один запрос на добавление. Давайте выберем первый вариант ответа. Это означает, что передовой массив никогда не будет оставаться передовым сразу после заполнения, а все массивы в хранилище будут заполнены.

Третье решение – когда выделить передовой массив. Давайте выделим его в тот момент, когда появится первый добавляемый элемент. Если передовой массив не нужен, мы будем хранить в его поле значение **null**.

Вот объявления полей для передового массива и хранилища:

```
var front: array?<T>
var depot: ExtensibleArray?<array<T>>
```

Добавим поле для хранения общего количества элементов в расширяемом массиве:

```
var length: int
```

Также будет удобно иметь поле, содержащее количество элементов, хранящихся в хранилище, потому что это позволит нам выбирать между поиском в передовом массиве или в хранилище:

```
var M: int // сокращение для:
           // if depot == null then 0 else 256 * |depot.Elements|
```

17.1.2. Инвариант объекта

Приняв эти решения, напишем инвариант объекта.

Инварианты фрейма

Начнем с инвариантов фрейма. Как всегда, **this** входит в набор представлений, поэтому начнем заполнять тело `Valid()` следующим образом:

```
{
  this in Repr &&
```

Передовой массив находится в наборе представлений, если только он не равен **null**. А если он не равен **null**, то его длина всегда равна 256:

```
(front != null ==>
  front.Length == 256 && front in Repr) &&
```

О хранилище можно сказать еще многое. Во-первых, стандартные свойства фрейма:

```
(depot != null ==>
  depot in Repr && depot.Repr <= Repr &&
  this !in depot.Repr && depot.Valid() &&
```

Наборы представлений `front` (который сам по себе) и `depot` (который является `depot.Repr`) не пересекаются:

```
  front !in depot.Repr &&
```

Если не включить все эти инварианты фрейма, то верификация вряд ли завершится успехом. Поэтому нелишним будет потратить дополнительную минуту на то, чтобы тщательно подумать, включили ли вы эти стандартные части и о различных способах пересечения наборов представлений (используйте раздел 16.4 в качестве контрольного списка).

Также есть еще кое-что, что можно сказать о массивах в хранилище. Каждый такой массив имеет длину 256:

```
  forall j :: 0 <= j < |depot.Elements| ==>
    depot.Elements[j].Length == 256 &&
```

Кроме того, каждый такой массив находится в наборе представлений `ExtensibleArray`, но вне набора представлений `depot`:

```
    depot.Elements[j] in Repr &&
    depot.Elements[j] !in depot.Repr &&
```

Наконец, каждый такой массив отличается от `front` и от всех остальных массивов в хранилище:

```
    depot.Elements[j] != front &&
    forall k :: 0 <= k < |depot.Elements| && k != j ==>
      depot.Elements[j] != depot.Elements[k]) &&
```

Инварианты длины

Мы решили добавить переменную `M` для хранения общего количества элементов, имеющихся в хранилище. Мы тоже решили присваивать переменной `depot` значение **`null`**, если в хранилище нет элементов. Это дает нам следующий инвариант:

```
M == (if depot == null then 0 else 256 * |depot.Elements|) &&
```

Остальные `length - M` элементов хранятся в передовом массиве `front`. Мы решили, что `front` будет иметь значение **`null`**, если в передовом массиве нет элементов, и перемещаться в хранилище сразу после добавления 256-го элемента. Это дает нам следующие два инварианта:

```
M <= length < M + 256 &&
(length == M <==> front == null) &&
```

Инварианты элементов

Количество элементов в расширяемом массиве хранится в `length`:

```
length == |Elements| &&
```

Первые `M` из этих элементов хранятся в хранилище, по 256 в каждом составляющем его массиве:

```
(forall i :: 0 <= i < M ==>
  Elements[i] == depot.Elements[i / 256][i % 256]) &&
```

а остальные – в массиве `front`:

```
(forall i :: M <= i < length ==>
  Elements[i] == front[i - M])
}
```

На этом мы завершаем определение инвариантов объекта. Теперь реализуем операции.

17.1.3. Реализация

Нам нужно реализовать четыре операции: конструктор, функцию `Get` и методы `Update` и `Append`. Мы рассмотрели их спецификации в разделе 17.1.0.

Конструктор должен инициализировать `Elements` значением `[]`, а инвариант сообщает нам, какие значения присвоить другим полям:

```
{
  front, depot := null, null;
  length, M := 0, 0;
  Elements, Repr := [], {this};
}
```

Функция Get просматривает front, если заданный индекс равен хотя бы M. В противном случае она находит соответствующий массив в хранилище и выбирает требуемый элемент из него:

```
{
  if M <= i then
    front[i - M]
  else
    var arr := depot.Get(i / 256);
    arr[i % 256]
}
```

Здесь мы имеем рекурсивный вызов Get, поэтому нам нужно доказать завершимость. Если функция имеет такую инструкцию **reads**, но не имеет инструкции **decreases**, то Dafny использует оценочную функцию завершимости по умолчанию с данным фреймом чтения (за которым следуют лексикографические компоненты по одному для каждого параметра функции, как мы видели ранее). Итак, инструкция **decreases** по умолчанию для Get – это лексикографический кортеж Repr, i (в чем можно убедиться, наведя указатель мыши на имя функции в Dafny IDE). Поскольку набор представлений для depot является строгим подмножеством множества представлений **this**, первый из этих лексикографических компонентов уменьшается при рекурсивном вызове. Другими словами, Dafny автоматически доказывает завершимость Get.

Реализация метода Update имеет ту же структуру, что и Get, но дополнительно изменяет элементы:

```
{
  if M <= i {
    front[i - M] := t;
  } else {
    var arr := depot.Get(i / 256);
    arr[i % 256] := t;
  }
  Elements := Elements[i := t];
}
```

Метод Append должен поместить заданное значение в новый элемент в массиве front. Но front может иметь значение **null**, и в этом случае должен быть размещен новый массив:

```
{
  if front == null {
    front := new T[256](_ => t);
    Repr := Repr + {front};
  }
```

Поскольку о параметре типа `T` ничего не известно, Dafny требует от нас указать начальные значения для элементов этого нового массива. К счастью, у метода `Append` есть такой элемент, а именно `t`, поэтому метод использует его для инициализации каждого элемента нового массива. (Альтернативное решение: объявить параметр типа `T` с характеристикой автоинициализации (`0`), как это было сделано с параметром типа `LazyArray` в разделе 17.0.0.)

Теперь `front` имеет значение, отличное от **`null`**, и мы можем поместить новый элемент на его место, а затем обновить `length` и `Elements`:

```
front[length - M] := t;
length := length + 1;
Elements := Elements + [t];
```

Давайте приостановимся здесь и подождем отклика верификатора. Он сообщит, что постусловие `Valid()` не может быть подтверждено, и выделит условие `length < M + 256` в `Valid()`, которое он не может доказать. Действительно, после увеличения значения `length` оно может равняться `256`. Если это так, то нам нужно переместить `front` в хранилище `depot`, а затем присвоить переменной `front` значение **`null`**. Вот эта проверка:

```
if length == M + 256 {
```

Если хранилище пустое, то `depot` равно **`null`**, как утверждает инвариант нашего объекта. Итак, начнем с выделения `depot`, если это необходимо:

```
    if depot == null {
      depot := new ExtensibleArray();
      Repr := Repr + depot.Repr;
    }
```

Предполагается, что параметр типа `ExtensibleArray` имеет значение **`array`**`<T>` (поэтому результат **`new`** будет соответствовать объявленному типу `depot`), но вы можете указать этот тип явно, если хотите видеть его в тексте программы.

Поскольку теперь значение `depot` не равно **`null`**, добавим передовой массив в хранилище и скорректируем оставшиеся поля:

```
    depot.Append(front);
    Repr := Repr + depot.Repr;
    M := M + 256;
    front := null;
  }
}
```

Здесь мы получили от верификатора одно сообщение об ошибке, а именно что он не может подтвердить завершимость рекурсивного вызова `Append`. Нам нужно предоставить оценочную функцию завершимости. Первая

мысль – добавить инструкцию **decreases** Repr, поскольку она очень хорошо сработала для Get. Однако было бы неверно считать, что depot.Repr во время рекурсивного вызова меньше, чем **this**.Repr на входе в Append. В частности, если на входе в Append переменная depot имела значение **null**, то исходный **this**.Repr не будет связан с depot.Repr во время вызова. Поэтому используем другую меру завершимости:

```
decreases |Elements|
```

Она действительно уменьшается, и на этом мы завершаем реализацию ExtensibleArray.

17.1.4. Итоги

Класс ExtensibleArray – это минное поле с трудноуловимыми пограничными условиями. В любом программном модуле такие условия обязательно должны документироваться. Мы начали проектирование с идеи (показанной на диаграмме в разделе 17.1.1). Затем подумали о решениях, которые нужно будет принять; например, о том, когда массивы могут быть полными или пустыми. Мы задокументировали решения, записав их в виде инварианта объекта, а после этого относительно легко написали код с помощью верификатора.

17.2. Двоичное дерево поиска для ассоциативного массива

В этом разделе мы реализуем преобразование целых чисел в значения некоторого произвольного типа Data. Реализация использует *двоичное дерево поиска*, которое представляет собой двоичное дерево, инфиксный обход которого дает отсортированный список.

Пример включает функцию Lookup, которая возвращает данные, соответствующие заданному целому числу, если таковое имеется. Чтобы сообщить, имеются ли данные, результат Lookup имеет стандартный тип данных Option, определенный следующим образом:

```
datatype Option<T> = None | Some(T)
```

17.2.0. Спецификация

Как обычно, начнем со спецификации, чтобы было понятно, к чему мы стремимся. Вот как выглядит класс с точки зрения клиента:

```
class BinarySearchTree<Data> {
  ghost var M: map<int, Data>
  ghost var Repr: set<object>
  ghost predicate Valid()
    reads this, Repr
    ensures Valid() ==> this in Repr
```

```
  constructor ()
    ensures Valid() && fresh(Repr)
    ensures M == map[]
  function Lookup(key: int): Option<Data>
    requires Valid()
    reads Repr
    ensures key in M.Keys ==> Lookup(key) == Some(M[key])
    ensures key !in M.Keys ==> Lookup(key) == None
  method Add(key: int, value: Data)
    requires Valid()
    modifies Repr
    ensures Valid() && fresh(Repr - old(Repr))
    ensures M == old(M)[key := value]
  method Remove(key: int)
    requires Valid()
    modifies Repr
    ensures Valid() && fresh(Repr - old(Repr))
    ensures M == old(M) - {key}
}
```

Класс `BinarySearchTree` параметризуется типом `Data`. Абстрактно экземпляр класса представляет собой ассоциативный массив с целочисленными ключами и значениями `Data`. Этот ассоциативный массив объявлен как призрачное поле `M`.

Следуя стандартной идиоме спецификации, класс объявляет призрачное поле `Repr` для хранения набора представлений и предикат `Valid()` для реализации инварианта объекта. Первая строка спецификации конструктора, первые две строки функции `Lookup` и первые три строки методов `Add` и `Remove` – это стандартные спецификации, использующие `Repr` и `Valid()`.

Остальные строки спецификаций членов относятся к двоичному дереву поиска. Конструктор создает дерево, представляющее пустой ассоциативный массив. Функция `Lookup` возвращает значение `Data`, соответствующее заданному ключу, завернутое в вариант `Some` возвращаемого типа. Если искомый ключ отсутствует в ассоциативном массиве, то `Lookup` возвращает `None`. Метод `Add` добавляет или изменяет значение `Data`, связанное с данным ключом. Метод `Remove` удаляет любое значение `Data`, связанное с данным ключом, оставляя все остальные записи в неприкосновенности. (Выражение вычитания из ассоциативного массива `m - s` обозначает ассоциативный массив `m` с удаленными ключами из `s`.)

17.2.1. Реализация

Реализация двоичного дерева поиска использует экземпляры еще одного класса: `Node`. Чтобы упорядочить объекты `Node` в древовидную структуру, класс `Node` объявляет поля `left` и `right`, ссылающиеся на другие объекты `Node`. Классу `Node` нужен способ представления отсутствия левого или правого дочернего элемента, а классу `BinarySearchTree` – способ представления отсут-

ствия каких-либо узлов вообще. Самое простое решение – использовать специальное значение **null**, которое является частью типа Node?.

Как я объяснял в разделе 17.1.1, для каждого класса, такого как Node, Dafny создает два типа. Один из этих типов имеет имя Node и представляет ссылку на объект Node. Другой имеет имя Node? и представляет ссылку на объект Node, которая может также иметь специальное значение **null**. Мы могли бы использовать тип Option<Node<Data>> для представления возможных ссылок на объекты Node<Data>, но пользоваться типом Node?<Data> немного проще, потому что при этом не нужно явно вызывать деструктор для получения ссылки.

Класс BinarySearchTree объявляет поле, которое ссылается на корневой узел дерева:

```
var root: Node?<Data>
```

Как мы вскоре увидим (в разделе 17.2.2), класс Node, как и BinarySearchTree, имеет члены M, Repr и Valid(). Используя их, запишем инвариант объекта двоичного дерева поиска:

```
ghost predicate Valid()
  reads this, Repr
  ensures Valid() ==> this in Repr
{
  this in Repr &&
  (|M.Keys| == 0 ==>
    root == null) &&
  (|M.Keys| != 0 ==>
    root in Repr && root.Repr <= Repr && this !in root.Repr &&
    root.Valid() &&
    root.M == M)
}
```

Упражнение 17.3

Реализуйте конструктор класса BinarySearchTree.

Учитывая определение класса Node, реализации членов Lookup, Add и Remove выглядят очень просто. Вы найдете их в упражнениях 17.5, 17.7 и 17.9 ниже. Далее мы сосредоточимся на классе Node.

17.2.2. Инвариант Node

В дополнение к полям M и Repr, которые играют те же роли, что и в BinarySearchTree, класс Node объявляет поля для хранения одной пары ключ–значение и возможных ссылок на дочерние узлы:

```
class Node<Data> {
  ghost var M: map<int, Data>
  ghost var Repr: set<object>
```

```
var key: int
var value: Data
var left: Node?<Data>
var right: Node?<Data>
```

Самая сложная часть класса `Node` – разработка его инварианта, в котором можно по невнимательности упустить некоторые важные части. Сначала я покажу его целиком, а потом подробно опишу каждую часть:

```
ghost predicate Valid()
  reads this, Repr
  ensures Valid() ==> this in Repr
{
  this in Repr &&
  (left != null ==>
    left in Repr && left.Repr <= Repr &&
    this !in left.Repr &&
    left.Valid() &&
    Less(left.M.Keys, {key})) &&
  (right != null ==>
    right in Repr && right.Repr <= Repr &&
    this !in right.Repr &&
    right.Valid() &&
    Less({key}, right.M.Keys)) &&
  (left != null && right != null ==>
    left.Repr !! right.Repr) &&
  M == Union(Union(map[key := value], left), right)
}
```

Бóльшая часть этого определения связана со структурой агрегатного объекта `Node`. Она соответствует стандартной идиоме спецификации, кратко изложенной в разделе 16.4, но обратите внимание, что все, что связано с `left`, защищено условием `left != null`. Аналогичное условие защищает все, что связано с `right`. А условие непересекаемости `left.Repr !! right.Repr` защищено условием, согласно которому `left` и `right` не равны значению `null`. Я уже упоминал об этом раньше, но отмечу еще раз: просмотрите эти определения дважды, сравните их с тем, что показано в разделе 16.4, и подумайте об особых потребностях вашей структуры данных. Уверенность в наличии правильных определений структуры с самого начала избавит вас от многих разочарований позже, когда вы приступите к реализации и использованию класса.

Помимо частей, связанных со структурой, в инварианте есть еще три строки:

```
Less(left.M.Keys, {key})
Less({key}, right.M.Keys)
M == Union(Union(map[key := value], left), right)
```

Первые две говорят об упорядоченности двоичного дерева поиска. Чтобы записать эти условия симметрично для левой и правой ветвей, я определил предикат `Less(a, b)`, который гласит, что каждое значение в наборе `a` меньше любого значения в `b`:

```
ghost predicate Less(a: set<int>, b: set<int>) {
  forall x, y :: x in a && y in b ==> x < y
}
```

Последняя строка в предикате `Node`, проверяющем выполнение свойств, описывает значение ассоциативного массива `M`. А `M` узла – это комбинация `left.M`, `right.M` и одноэлементного ассоциативного массива **`map`**`[key := value]`. Чтобы выразить эту комбинацию, я определил функцию `Union(m, n)`, которая расширяет ассоциативный массив `m` ассоциативным массивом `n.M`, позволяя `n` иметь значение **`null`**:

```
ghost function Union<Data>(m: map<int, Data>,
                           n: Node?<Data>): map<int, Data>
  reads n
{
  if n == null then m else m + n.M
}
```

Функции `Less` и `Union` можно определить внутри класса `Node`. Однако, поскольку они не зависят от аргумента-получателя, я думаю, что они лучше будут смотреться на уровне модуля, т. е. лучше объявить их на том же уровне модуля, что и класс `Node`.

Теперь, определив инвариант `Node`, реализуем методы и функции этого класса.

17.2.3. Реализация Node

В классе `Node` нам нужно определить конструктор, функцию `Lookup` и методы `Add` и `Remove`, которые вызываются для выполнения основной работы при вызове соответствующих процедур в классе `BinarySearchTree`. Функция и методы определяются как рекурсивные, выполняющие обход структуры данных `Node`. Далее я покажу спецификации, а большую часть реализаций оставлю вам в качестве самостоятельного упражнения.

Конструктор

Конструктор `Node` принимает пару ключ–значение и имеет стандартную спецификацию, дополненную постусловием о начальном значении `M`:

```
  constructor (key: int, value: Data)
    ensures Valid() && fresh(Repr)
    ensures M == map[key := value]
```

Упражнение 17.4

Реализуйте конструктор `Node`.

Lookup

Спецификация `Lookup` имеет ту же структуру, что и спецификация функции `Lookup` в классе `BinarySearchTree`:

```
function Lookup(key: int): Option<Data>
  requires Valid()
  reads Repr
  ensures key in M.Keys ==> Lookup(key) == Some(M[key])
  ensures key !in M.Keys ==> Lookup(key) == None
```

Упражнение 17.5

Реализуйте функцию `Lookup` класса `BinarySearchTree`.

Упражнение 17.6

Реализуйте функцию `Lookup` класса `Node`.

Add

Спецификация метода `Add` тоже имеет ту же структуру, что и спецификация метода `Add` в классе `BinarySearchTree`:

```
method Add(key: int, value: Data)
  requires Valid()
  modifies Repr
  ensures Valid() && fresh(Repr - old(Repr))
  ensures M == old(M)[key := value]
```

Упражнение 17.7

Реализуйте функцию `Add` класса `BinarySearchTree`. Подсказка: в этой реализации вам потребуется обновить поле `Repr` (иначе реализация не будет соответствовать спецификации, и верификатор сообщит об ошибке), т. е. тело метода должно завершаться строкой:

```
Repr := Repr + root.Repr;
```

Прежде чем дать вам упражнение по реализации `Add` в классе `Node`, я хочу сказать кое-что о завершимости его рекурсивных вызовов. В этих рекурсивных вызовах (немного подскажу вам):

```
left.Add(key, value);
```

и

```
right.Add(key, value);
```

где `key` и `value` – это параметры самого метода `Add`. В качестве оценочной функции завершимости по умолчанию для `Add` Dafny выберет

```
decreases key
```

но эта оценка не уменьшается в рекурсивных вызовах. Деревья с корнями в `left` и `right` содержатся в дереве с корнем в **`this`**, поэтому можно попробовать использовать некоторую меру завершимости, основанную на размере или форме этих деревьев. К счастью, у нас уже есть такое значение – набор представлений каждого из этих деревьев, который хранится в поле `Repr`. Поэтому, чтобы доказать завершимость рекурсии `Add`, можно использовать

```
decreases Repr
```

но нам придется явно добавить эту инструкцию, чтобы переопределить значение по умолчанию, которое выбирает Dafny.

Теперь у вас может возникнуть вопрос: почему нам не пришлось писать явно подобную инструкцию **`decreases`** для реализации `Lookup` (упражнение 17.6)? Все просто: потому что для функции, зависящей от состояния объекта, Dafny добавляет в инструкцию **`decreases`** по умолчанию набор объектов, перечисленных в инструкции **`reads`**. Функция `Lookup` объявлена с инструкцией **`reads`** `Repr`, поэтому, чтобы учесть ее зависимость от состояния, Dafny использует инструкцию **`decreases`** по умолчанию

```
decreases Repr, key
```

где `key` – формальный параметр. Рекурсивные вызовы `Lookup` уменьшают эту оценку, поэтому завершимость подтверждалась автоматически (если бы завершимость была столь же «очевидна» для вас, как и для проверяющего, то вы, возможно, даже бы не подумали о необходимости ее доказывать).

Чтобы узнать, какую инструкцию **`decreases`** по умолчанию использует верификатор, просто наведите указатель мыши на имя метода или функции в IDE, содержащее рекурсивный (или взаимно рекурсивный) вызов, и вы увидите эту инструкцию во всплывающей подсказке.

Упражнение 17.8

Реализуйте функцию `Add` класса `Node`.

Подсказка: помните об инструкции **`decreases`**, о которой я только что рассказал.

Remove

Наконец, давайте определим метод `Remove` в классе `Node`. Этот метод самый сложный, потому что должен учитывать представление пустого ассоциативного массива значением **`null`**. Наш метод будет возвращать ссылку на обновленное дерево или **`null`**, если из дерева был удален последний элемент:

```
method Remove(key: int) returns (n: Node?<Data>)
  requires Valid()
  modifies Repr
  ensures n != null ==> n.Valid() && n.Repr <= old(Repr)
  ensures var newMap := old(M) - {key};
    (|newMap.Keys| == 0 ==> n == null) &&
    (|newMap.Keys| != 0 ==> n != null && n.M == newMap)
```

Обратите внимание, что эта спецификация немного отличается от стандартной спецификации метода. Вместо того чтобы гарантировать **this**.`Valid()` после завершения, этот метод гарантирует `n.Valid()`, но только если `n` не равно нулю. Кроме того, если `n` не равно нулю, метод может гарантировать более стандартное условие:

```
fresh(n.Repr - old(this.Repr))
```

Однако, чтобы немного упростить ситуацию, я выбрал более сильное постусловие, которое не позволяет `Remove` размещать дополнительное состояние (см. упражнение 17.12.)

Упражнение 17.9

Реализуйте функцию `Remove` класса `BinarySearchTree`.

Реализация `Remove` в классе `Node` требует больше усилий, поэтому я опишу ее в отдельном подразделе.

17.2.4. Реализация Remove в классе Node

Как я уже отмечал выше, обсуждая реализацию `Add`, рекурсивная природа `Remove` потребует явно указать инструкцию **decreases**. Мы столкнемся с этим требованием чуть позже, но тогда нас будут занимать совсем другие мысли, поэтому я предлагаю сразу добавить:

```
  decreases Repr
```

В реализации `Remove` мы должны учесть четыре основных случая: удаляемый ключ может храниться непосредственно в **this**, в левом или правом поддереве, или его вообще может не быть в дереве **this**. Начнем с самого сложного:

```
{
  if key == this.key {
```

Мы добрались до узла `Node`, содержащего пару ключ–значение, которую нужно удалить. Дальнейшие действия зависят от наличия поддеревьев в **this**. Если левого поддерева нет, то можно просто вернуть значение `right`, каким бы оно и было (даже **null**):

```
    if left == null {
      return right;
```

Аналогично если нет правого поддерева, то можно сразу же вернуть `left`:

```
    } else if right == null {
      return left;
```

Если есть оба поддерева, то мы должны выполнить некоторые операции с этими деревьями. Наш план прост: переместим в **this** пару ключ–значение с наименьшим ключом в правом поддереве. Сделаем это в три этапа – найдем минимальный ключ в `right`, сохраним его вместе со значением в **this**, а затем удалим этот минимальный ключ из правого поддерева:

```
    } else {
      var (k, val) := right.Min();
      this.key, this.value := k, val;
      right := right.Remove(k);
    }
```

Вот определение функции `Min`, постусловие которой аналогично тому, что мы видели ранее в книге для аналогичных функций поиска минимумов:

```
  function Min(): (int, Data)
    requires Valid()
    reads Repr
    ensures var (k, val) := Min();
      k in M.Keys && M[k] == val &&
      forall k' :: k' in M.Keys ==> k <= k'
  {
    if left == null then
      (key, value)
    else
      left.Min()
  }
```

Функция `Min` возвращает пару ключ–значение в виде кортежа, который является встроенным типом данных. Обратите внимание, что этот кортеж легко разложить на составляющие, как показано в теле метода `Remove` и в спецификации `Min`.

Упражнение 17.10

Зачем в `Min` нужно предусловие?

Продолжим реализацию `Remove`. Теперь у нас остались случаи, когда искомый ключ может храниться только в левом или только в правом поддереве:

```
    } else if key < this.key && left != null {
      left := left.Remove(key);
    } else if this.key < key && right != null {
```

```
      right := right.Remove(key);
    }
```

В четвертом случае, когда искомый ключ может храниться только в левом или только в правом поддереве, но это поддерево пустое, а значит, данный ключ отсутствует в дереве **this**, метод Remove ничего не должен делать.

В завершение обновляем M и возвращаем узел-получатель:

```
    M := M - {key};
    return this;
  }
```

Написанный нами код выглядит безупречно, но верификатор жалуется на постусловие, где |newMap.Keys| == 0. Поразмыслив, можно заметить, что ассоциативный массив newMap может быть пустым, только если перед вызовом метода он содержал единственный элемент с искомым ключом key. Такое возможно, только если оба поля, left и right, содержат **null**. Это должно следовать из того, что условие выполнения свойств структуры данных говорит нам о M:

```
M == Union(Union(map[key := value], left), right)
```

Функция Union предполагает операцию слияния ассоциативных массивов (+), для которой мы позже получаем набор .Keys, поэтому, возможно, верификатору не хватает информации о кардинальности. Давайте поможем ему и посмотрим, сможет ли он после этого построить доказательство. (В конце концов, мы же уверены в верности реализации Remove? Или не уверены?)

Постусловие в Union

Добавим в Union следующее постусловие:

```
  ensures var u := Union(m, n);
    |m.Keys| <= |u.Keys|
```

Ура! После добавления этого постусловия верификатор смог доказать корректность метода Remove. Но он не смог доказать само постусловие, поэтому нам придется помочь ему еще раз.

Взглянув на тело Union, можно заметить, что возвращаемый ассоциативный массив содержит все пары ключ–значение данного массива m. Посмотрим, сможет ли верификатор это подтвердить. Чтобы задать вопрос верификатору, можно добавить утверждение **assert**, но в данном случае у нас нет подходящего для этого места. Давайте дадим имя генератору ассоциативного массива в ветви **else** и напишем утверждение **assert** с использованием этого имени, прежде чем вернуть результат. Вот новое тело для Union:

```
  if n == null then
    m
```

```
  else
    var u := m + n.M;
    assert m.Keys <= u.Keys;
    u
```

Верификатор удовлетворен условием в **assert**, но теперь жалуется на похожее условие в постусловии. Судя по всему, верификатор не видит связи между подмножеством и мощностью. Что ж, давайте сформулируем необходимое нам свойство в виде леммы:

```
lemma SubsetCardinality(a: set, b: set)
  requires a <= b
  ensures |a| <= |b|
```

и добавим вызов этой леммы в тело `Union`:

```
  if n == null then
    m
  else
    var u := m + n.M;
    SubsetCardinality(m.Keys, u.Keys);
    u
```

Ура! Теперь постусловие `Union` благополучно верифицируется.

Лемма о мощности множества

Чтобы доказать лемму `SubsetCardinality`, попробуем применить индуктивное доказательство, которое рекурсивно вызывает лемму на немного меньшем множестве `b`:

```
{
  if a != b {
    var x :| x in b - a;
    SubsetCardinality(a, b - {x});
  }
}
```

Ура! Верификатор знает достаточно об этих операциях с множествами, чтобы доказать лемму, которая нам понадобилась для доказательства постусловия `Union`, необходимого для доказательства корректности метода `Remove` в классе `Node`, который мы вызвали из метода `Remove` в классе `BinarySearchTree`!

Упражнение 17.11

Напишите альтернативное доказательство леммы `SubsetCardinality`, которая удаляет элемент из `a` и `b` при рекурсивном вызове.

Упражнение 17.12

Замените первое постусловие `Node.Remove` на

```
ensures n != null ==> n.Valid() && fresh(n.Repr - old(Repr))
```

и скорректируйте реализацию метода, чтобы она благополучно верифицировалась.

17.3. Итератор для ассоциативного массива

В некоторых примерах в этой книге представлены реализации различных типов контейнеров, например множеств, массивов и ассоциативных массивов. Обычная операция над контейнером – перебор его элементов. В этом разделе мы напишем спецификацию и реализацию класса `Iterator`, который имеет метод `GetNext()`, последовательно возвращающий элементы `BinarySearchTree`, реализованного в разделе 17.2.

Главная особенность этого примера – итератор напрямую обращается к базовому контейнеру `BinarySearchTree`. Это типично для итераторов. Зависимость от структуры данных контейнера приводит к эффективной реализации инкрементного метода `GetNext()`, но вызывает беспокойство ситуация, когда контейнер изменяется в ходе итераций. Спецификация, которую я покажу, фактически делает бесполезным любой объект-итератор при изменении контейнера. Это легко достигается путем определения зависимости действительности итератора от набора представлений контейнера (или, точнее, не исключая возможности зависимости действительности итератора от набора представлений контейнера).

17.3.0. Спецификация

Вот как выглядит класс `Iterator` с точки зрения клиента:

```
class Iterator<Data> {
  const bst: BinarySearchTree<Data>
  ghost var RemainingKeys: set<int>
  ghost const Repr: set<object>
  ghost predicate Valid()
    reads this, Repr
    ensures Valid() ==>
      this in Repr && bst.Valid() && RemainingKeys <= bst.M.Keys
  constructor (bst: BinarySearchTree<Data>)
    requires bst.Valid()
    ensures Valid() && fresh(Repr - bst.Repr)
    ensures this.bst == bst && RemainingKeys == bst.M.Keys
  method GetNext() returns (r: Option<(int, Data)>)
    requires Valid()
    modifies Repr - bst.Repr
    ensures Valid()
```

```
    ensures match r
      case None =>
        old(RemainingKeys) == RemainingKeys == {}
      case Some((k, val)) =>
        k in old(RemainingKeys) && bst.M[k] == val &&
        RemainingKeys == old(RemainingKeys) - {k}
}
```

Как следует из объявления поля `bst` константой, итератор связан с фиксированным двоичным деревом `BinarySearchTree`. Состояние итератора включает `RemainingKeys` – подмножество ключей контейнера, которые итератор еще не вернул. Конструктор итератора инициализирует `RemainingKeys` полным набором ключей контейнера, и каждый вызов `GetNext()` удаляет из набора и возвращает один из этих ключей. Постусловие `Valid()` сообщает клиентам, что выполнение свойств итератора подразумевает выполнение свойств контейнера, а также связь между подмножествами `RemainingKeys` и `bst.M.Keys`.

Чтобы сообщить о том, что ключи были исчерпаны, `GetNext()` возвращает `None`, где тип `Option` – это тип данных, определенный в начале раздела 17.2. Постусловие гласит, что `None` возвращается только тогда, когда набор `RemainingKeys` пуст. В противном случае `GetNext()` возвращает пару `(k, val)`, где `bst.M[k] == val`. Обратите внимание, что `GetNext()` не дает никаких гарантий относительно порядка, в каком будут возвращаться пары ключ–значение (см. упражнение 17.14).

Набор представлений `Repr` итератора указывается как новый (**`fresh`**) при создании итератора, за исключением возможного пересечения с набором представлений контейнера. Это позволяет свойству допустимости итератора зависеть от состояния структуры данных контейнера. Эта зависимость не позволяет гарантировать корректность итератора после изменения `bst.Repr`. 😳

Набор представлений итератора никогда не меняется (т. е. не меняется сам набор, но состояние объектов в наборе представлений может меняться). Поэтому я объявил `Repr` как константу (см. примечание о неизменяемых полях в разделе 16.2.4). Методу `GetNext()` разрешено изменять состояние той части набора представлений итератора, которая находится за пределами набора представлений контейнера. Другими словами, итератор может читать структуру данных контейнера, но не изменять ее.

17.3.1. Тестовая программа

Чтобы проверить пригодность спецификации итератора для использования, рассмотрим следующий метод:

```
method TestHarness<Data>(a: BinarySearchTree, b: BinarySearchTree,
                         d: Data)
  requires a.Valid() && b.Valid() && a.Repr !! b.Repr
```

```
    modifies a.Repr
  {
    var iter0 := new Iterator(a);
    var iter1 := new Iterator(a);
    var iter2 := new Iterator(b);
    var o := iter0.GetNext();
    o := iter0.GetNext();
    o := iter1.GetNext();
    o := iter0.GetNext();
    o := iter2.GetNext();
    a.Add(15, d);
    o := iter2.GetNext(); // пока все хорошо
    o := iter0.GetNext(); // ошибка: контейнер был изменен
  }
```

Эта тестовая программа сначала создает два действительных и непересекающихся контейнера, `a` и `b`, а затем два итератора: один для `a` и один для `b`. Пока два контейнера не изменяются, мы можем вызывать метод `GetNext()` для любого итератора. После того как контейнер `a` был изменен (вызовом `Add`), итератор для `b` все еще можно использовать. Однако невозможно доказать выполнение предусловия `iter0.Valid()` для последнего вызова `GetNext()` в этом примере, потому что `iter0.Valid()` зависит от состояния, измененного вызовом `Add`. Таким образом, вызов `Add` для `a` делает итераторы `iter0` и `iter1` непригодными для использования.

17.3.2. Стек оставшихся узлов

Реализация итератора выполняет обход узлов в контейнере. Она делает это постепенно, поэтому при каждом вызове `GetNext` итератор отыскивает следующий непосещенный узел. Чтобы не сбиться в процессе работы, итератор поддерживает стек узлов. Для каждого узла в стеке итератор должен посетить пару ключ–значение в узле, а также правое поддерево узла.

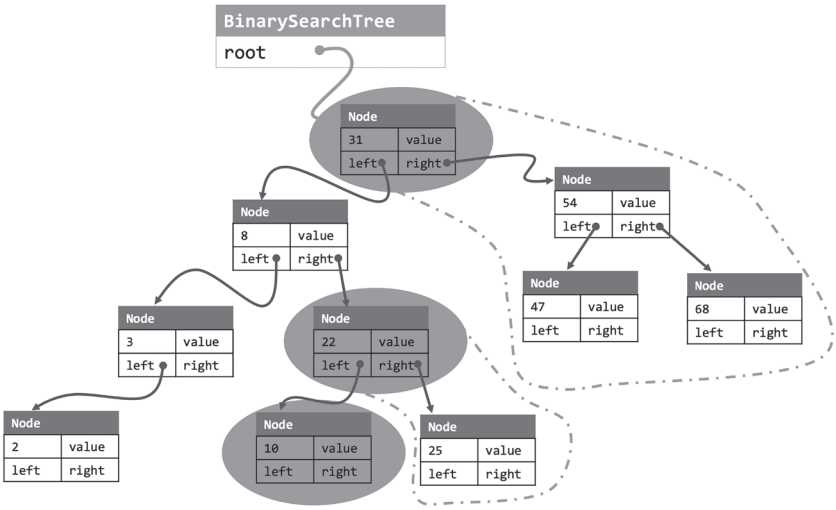
На следующей диаграмме (см. рис. ниже) показано двоичное дерево поиска и состояние итератора.

Стек этого итератора состоит из трех узлов, выделенных на диаграмме затененными овалами. Этому итератору остается вернуть узел (ключ которого равен) `10`, затем узел `22` и его правое поддерево, а затем узел `31` и его правое поддерево.

Для представления стека используем следующий тип данных:

```
datatype List<T> = Nil | Cons(T, List<T>)
```

Учитывая, что список `List` является неизменяемым, мы можем не беспокоиться об определении фрейма для него. Более того, поскольку значения типов данных всегда конечны, при обходе этого списка легко доказать завершимость.



Добавим в класс `Iterator` поле для хранения стека:

```
var stack: List<Node<Data>>
```

Позвольте мне упомянуть две альтернативы типу данных `List`. Одна из альтернатив – использование экземпляров некоторого класса узлов связанного списка. Если поля `head` и `tail` такого класса объявлены неизменяемыми (с использованием **`const`**), то нам не придется волноваться о фреймах. Однако нам все равно придется размещать такие узлы с помощью **`new`**, что недопустимо для функций, и придется написать больше кода для подтверждения завершимости обхода списка. Второй альтернативой может быть использование встроенного в Dafny типа **`seq`**. Это упрощает определение фреймов и доказательство завершимости, но работать с целочисленными индексами в такой последовательности сложнее, поскольку доступ к стеку осуществляется только с одного конца.

17.3.3. Инвариант итератора

Объявим инвариант итератора, используя предикат `Valid` как обычно:

```
ghost predicate Valid()
  reads this, Repr
  ensures Valid() ==>
    this in Repr && bst.Valid() &&
    RemainingKeys <= bst.M.Keys
{
  this in Repr &&
  bst in Repr && bst.Repr <= Repr && this !in bst.Repr &&
  bst.Valid() &&
  RemainingKeys <= bst.M.Keys &&
```

```
    SValid(stack, RemainingKeys)
}
```

где последняя конъюнкция гласит, что `stack` корректно представляет одну пару ключ–значение для каждого ключа в `RemainingKeys`.

Предикат `SValid` объявляется следующим образом:

```
ghost predicate SValid(st: List<Node<Data>>, R: set<int>)
  reads bst, bst.Repr
```

Инструкция **`reads`** в этом предикате выражает его зависимость от объектов, представляющих `bst`. Чтобы упомянуть **`this`**`.bst.Repr` в инструкции **`reads`**, она должна также включать **`this`**`.bst`, потому что состояние объекта, на который ссылается **`this`**`.bst`, влияет на то, какой набор объектов обозначает **`this`**`.bst.Repr`. Используя те же рассуждения, можно было бы подумать, что, для того чтобы упомянуть **`this`**`.bst` в инструкции **`reads`**, она также должна включать **`this`**. Однако это не так, потому что `bst` – неизменяемое поле. Таким образом, поскольку состояние не может измениться и привести к тому, что **`this`**`.bst` будет ссылаться на другой объект, функции разрешено ссылаться на него без упоминая в инструкции **`reads`**.

Определение `SValid` поддерживает два случая, в зависимости от значения `st`. Первый из них – `Nil`:

```
{
  match st
  case Nil =>
    R == {}
```

Пустой стек `Nil` корректно представляет одну пару ключ–значение для каждого ключа в `R`, когда `R` пустой. Другой случай, `Cons`, заслуживает более подробного обсуждения. Прежде всего это инвариант структуры, который гласит, что узел `node` и его представление являются частью `bst.Repr` и `node`. Допустим:

```
  case Cons(node, next) =>
    node in bst.Repr && node.Repr <= bst.Repr && node.Valid() &&
```

Следующий конъюнкт гласит, что `node.M` связан с `bst.M`, т. е. каждая пара ключ–значение, представленная узлом `node`, – одна из представления в `bst`:

```
    (forall k :: k in node.M.Keys ==>
      k in bst.M.Keys && bst.M[k] == node.M[k]) &&
```

Далее предикат `SValid` определяет связанную переменную `m` как пару ключ–значение, хранящуюся непосредственно в узле `node`, в сочетании с парами ключ–значение, представленными правым поддеревом `node`:

```
    var m := Union(map[node.key := node.value], node.right);
```

Теперь у нас есть стек Cons(node, next), представляющий пары ключ–значение в m.Keys и любые пары ключ–значение, представленные стеком next. Таким образом, последние два конъюнкта в SValid гласят, что R можно разделить на m.Keys и R - m.Keys так, что node и next будут представлять соответствующие пары ключ–значение:

```
      m.Keys <= R &&
      SValid(next, R - m.Keys)
  }
```

17.3.4. Добавление узла в стек

Далее нам нужно реализовать конструктор и метод GetNext в классе Iterator. Оба вталкивают узел Node в стек. Начнем с метода Push, это самая сложная часть предстоящей работы.

Спецификация

Вызов Push(nn) должен втолкнуть в стек все пары ключ–значение, представленные деревом с корнем в узле nn. Параметр nn может иметь значение **null**, и в этом случае в стек ничего не помещается:

```
method Push(nn: Node?<Data>)
```

Для методов, вызываемых клиентами, обычно предполагается предусловие Valid(). Однако Push – это рабочий метод, который вызывается, когда инвариант может быть временно нарушен. Поэтому мы потребуем выполнения не всех условий в Valid(), а только тех из них, которые действительно необходимы в Push, а именно: выполнение свойств контейнера, объект итератора не является частью набора представлений контейнера (мы собираемся изменить поля объекта итератора и должны гарантировать, что это не нарушит выполнение свойств контейнера), и стек правильно представляет ключи в RemainingKeys:

```
requires bst.Valid() && this !in bst.Repr
requires SValid(stack, RemainingKeys)
```

Нам также понадобятся три свойства параметра nn, когда он не равен **null**. Во-первых, нам нужно, чтобы nn был частью набора представлений bst и чтобы nn был допустимым деревом:

```
requires nn != null ==>
  nn in bst.Repr && nn.Repr <= bst.Repr && nn.Valid()
```

Во-вторых, ассоциативный массив, представленный в nn, должен согласовываться с ассоциативным массивом, представленным в bst:

```
requires nn != null ==>
  forall k :: k in nn.M.Keys ==>
    k in bst.M.Keys && bst.M[k] == nn.M[k]
```

В-третьих, мы собираемся поместить `nn.M.Keys` в стек, который уже представляет `RemainingKeys`, и хотим, чтобы стек мог хранить не более одного экземпляра ключа, поэтому нам нужно, чтобы эти наборы не пересекались:

```
requires nn != null ==> RemainingKeys !! nn.M.Keys
```

Уф! Продолжим: метод `Push` должен изменять объект итератора

```
modifies this
```

поддерживать инвариантное отношение между стеком и `RemainingKeys`

```
ensures SValid(stack, RemainingKeys)
```

с расширением `RemainingKeys` за счет `nn.M.Keys`:

```
ensures RemainingKeys
    == old(RemainingKeys) + if nn == null then {} else nn.M.Keys
```

Тело

С функциональной точки зрения метод `Push` должен втолкнуть `nn` и его левые дочерние элементы в стек. Например, вызов `Push(bst.root)` для пустого стека поместит в этот стек узлы, доступные по ссылкам `.left` из `bst.root`, и создаст состояние, показанное на рис. 17.1. Это можно было бы реализовать, используя рекурсивный подход, но я предлагаю написать итеративное решение, чтобы лишний раз потренироваться в написании спецификации цикла.

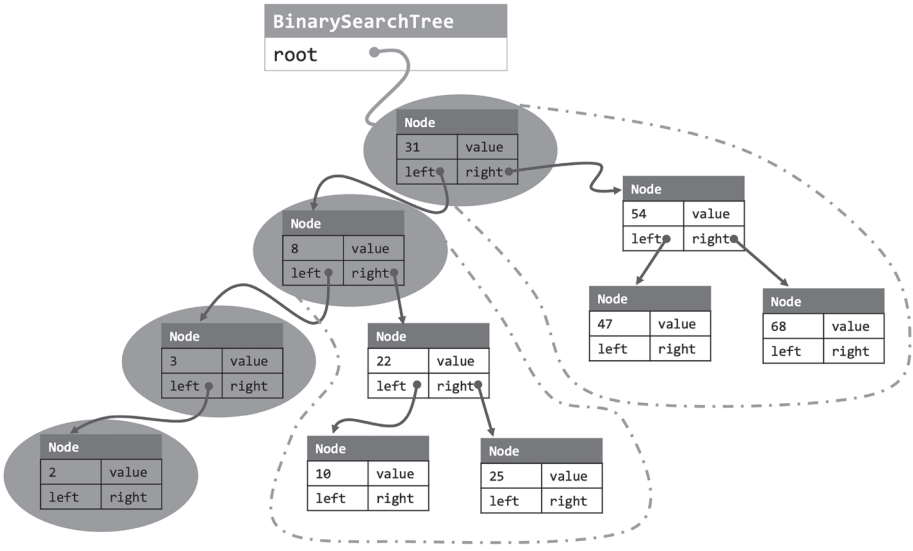


Рис. 17.1. Вызов Push(bst.root) для дерева bst поместит в стек узлы, выделенные затененными овалами

Итак, вместо рекурсивного вызова Push с новым значением nn мы используем цикл, каждая итерация которого получает новое значение nn. Другими словами, мы используем nn в качестве индекса цикла. Однако nn является формальным входным параметром, поэтому мы не можем изменять его в теле метода. Поэтому введем для этой цели локальную переменную:

```
{
  var node := nn;
```

Цикл будет двигаться вперед к последнему постусловию, уменьшая в каждой итерации набор, заданный выражением **if-then-else**. Для удобства дадим имя этому выражению:

```
  ghost var Final := if node == null then {} else node.M.Keys;
```

Запишем то, что осталось сделать, в призрачную переменную:

```
  ghost var S := Final;
```

Спецификация цикла

Используя node, Final и S, составим спецификацию цикла из частей спецификации метода. Итерации должны продолжаться, пока не будут последовательно обработаны все левые дочерние элементы nn:

```
  while node != null
```

Этот метод должен гарантировать сохранение допустимости стека, поэтому заставим каждую итерацию делать одно и то же (т. е. используем метод конструирования циклов 13.1):

```
    invariant SValid(stack, RemainingKeys)
```

Нам понадобятся те же три свойства узла nn, что и методу. Вот два из них:

```
    invariant node != null ==>
      node in bst.Repr && node.Repr <= bst.Repr && node.Valid()
    invariant node != null ==>
      forall k :: k in node.M.Keys ==>
        k in bst.M.Keys && bst.M[k] == node.M[k]
```

В качестве третьего свойства используем локальную переменную S, значение которой также задается инвариантом:

```
    invariant S == if node == null then {} else node.M.Keys
    invariant RemainingKeys !! S
```

Наконец, превратим последнее постусловие в инвариант «что еще предстоит сделать» (метод конструирования циклов 12.1), согласно которому S уменьшается в каждой итерации цикла:

```
    invariant RemainingKeys + S == old(RemainingKeys) + Final
    decreases S
```

Тело цикла

Теперь осталось только написать тело цикла. Давайте определим призрачную переменную `m`, которая будет обозначать пары ключ–значение, помещаемые в стек в ходе итераций:

```
  {
    ghost var m := Union(map[node.key := node.value], node.right);
```

Теперь просто поместим узел `node` в стек и перейдем к его левому дочернему узлу:

```
    stack, node := Cons(node, stack), node.left;
```

Наконец, переместим ключи в `m.Keys` из `S` в `RemainingKeys` и на этом завершим цикл и весь метод `Push`:

```
    RemainingKeys, S := RemainingKeys + m.Keys, S - m.Keys;
  }
}
```

Упражнение 17.13

Напишите рекурсивную реализацию `Push`.

17.3.5. Конструктор

Метод `Push` получился довольно сложным. Конструктор, по сравнению с ним, выглядит попроще: он инициализирует поля, представляющие пустой стек, а затем вызывает `Push`, чтобы добавить в стек все пары ключ–значение из заданного контейнера:

```
constructor (bst: BinarySearchTree<Data>)
  requires bst.Valid()
  ensures Valid() && fresh(Repr - bst.Repr)
  ensures this.bst == bst && RemainingKeys == bst.M.Keys
{
  this.bst := bst;
  stack, RemainingKeys := Nil, {};
  Repr := {this} + bst.Repr;
  new;
  Push(bst.root);
}
```

Обратите внимание, что вызов `Push` производится на втором этапе выполнения конструктора, т. е. после `new;`. До этого момента константные

поля итератора (которые используются методом Push) недоступны (см. раздел 16.3.2).

17.3.6. Метод GetNext

Напомню, как выглядит сигнатура метода GetNext (его спецификацию вы найдете в разделе 17.3.0):

```
method GetNext() returns (r: Option<(int, Data)>)
```

Тело реализует два случая: когда все пары ключ–значение были возвращены и когда еще остались пары, которые можно вернуть. Первый случай прост:

```
{
  match stack
  case Nil =>
    return None;
```

Во втором случае метод подготавливает к возврату пару ключ–значение, находящуюся на вершине стека:

```
  case Cons(node, next) =>
    r := Some((node.key, node.value));
```

(Обратите внимание на двойные круглые скобки. Внешние скобки ограничивают список аргументов конструктора Some, а внутренние создают кортеж из двух элементов.)

Затем мы извлекаем узел node из стека (путем выбора списка, содержащего верхний узел и хвост стека) и вносим соответствующие корректировки в RemainingKeys:

```
    ghost var m := Union(map[node.key := node.value], node.right);
    stack, RemainingKeys := next, RemainingKeys - m.Keys;
```

Последняя операция исключает из RemainingKeys слишком много, а поскольку спецификация GetNext гласит, что удалить следует только одну возвращаемую пару ключ–значение, мы помещаем правое поддерево узла node обратно в стек, для чего достаточно сделать единственный вызов Push:

```
    Push(node.right);
}
```

На этом мы завершаем реализацию класса Iterator.

Упражнение 17.14

Усильте спецификацию GetNext() так, чтобы она гарантировала возврат пары ключ–значение с наименьшим ключом из всех оставшихся. Затем внесите необходимые изменения в реализацию и доказательство.

17.4. Итоги

В этой главе я показал четыре примера программ, использующих динамические структуры данных, содержимое которых может меняться с течением времени. Две из них были разновидностями массивов, а остальные – реализацией ассоциативного массива и связанного с ним итератора. Во всех этих примерах используется шаблон спецификации с инвариантом объекта в предикате `Valid()`, который добавляется как предикат в пред- и постусловие каждого метода, доступного клиентам. Фреймы, затрагивающие части кучи, которые используются или модифицируются, задаются путем определения набора представлений агрегатного объекта. Гибкость динамических фреймов нашла хорошее применение в классе `Iterator`, где фреймы нескольких итераторов и самой коллекции могут перекрываться.

Примечания

Пример отложенной инициализации массива в разделе 17.0 взят из учебника по классическим алгоритмам (упражнение 2.12 в [1]).

Функциональные языки всегда поддерживали типы `Option`, позволяющие отличить присутствующие значения от отсутствующих. Поскольку тип `Option<T>` отличается от `T`, типы программы ясно дают понять, где значение может отсутствовать. В традиционных императивных языках для обозначения отсутствующих значений вместо этого используют специальное ссылочное значение **`null`**, но – ужас! 😩 – без возможности декларативно отличить «определенно, присутствует» от «возможно, отсутствует». Первым императивным языком, позволяющим различать ссылочные типы, которые могут и не могут иметь пустые значения, был Spec# [46, 47] – расширение языка C#.

Безопасность ссылочных типов, не поддерживающих значение **`null`** (и, если уж на то пошло, любого типа, для которого компилятор не может предоставить значение по умолчанию) достигается за счет ограничения возможностей конструирования объектов и других составных значений. Например, двухэтапные конструкторы в Dafny обеспечивают инициализацию всех непустых полей, прежде чем новый объект можно будет использовать.

Пример `Iterator` в разделе 17.3 решает проблему, известную программистам на Java как `ConcurrentModificationException`. Это исключение генерируется при попытке изменить коллекцию, для которой имеется активный итератор. Чтобы глубже понять проблему, начнем с самого начала: общедоступный интерфейс коллекции предлагает операции по извлечению определенных элементов из коллекции. Каждая такая операция выполняет обход содержимого структуры данных сверху вниз. Чтобы получить несколько элементов, может оказаться более эффективным перемещать курсор по структуре, а не выполнять обход данных с самого начала всякий раз. Именно этой цели служат итераторы коллекций. Далее, операция, изменяющая коллекцию, приведет к изменению базовой структуры данных в коллек-

ции. После такого изменения невозможно сказать, что будет означать ранее полученный курсор. По этой причине операция `next()` итератора в Java начинается с проверки целостности курсора (например, путем просмотра некоторого счетчика времени выполнения, который увеличивается при каждом обновлении коллекции) и выдает исключение `ConcurrentModificationException`, если коллекция изменилась.

Как было показано в разделе 17.3.1, спецификации позволяют верификатору обнаружить такую ситуацию. Проще говоря, допустимость итератора зависит от коллекции, поэтому любое изменение коллекции делает итератор недействительным.

Есть еще несколько замечаний о взаимодействии между коллекциями и их итераторами. Наша реализация позволяет создавать любое количество итераторов для коллекции. Каждый итератор зависит от коллекции, но сами итераторы не зависят друг от друга. Соответственно, итераторы можно использовать независимо друг от друга. Если программе понадобится изменить коллекцию, то мы должны доказать, что коллекция по-прежнему допустима. В частности, нет необходимости каким-то образом сообщать итераторам о том, что коллекция вот-вот обновится. Вместо этого итератор вовлекается в проверку только тогда, когда он используется. В этот момент необходимо доказать, что итератор все еще допустим, а с нашими спецификациями это возможно только в том случае, если можно доказать, что в коллекции не было никаких изменений. Все это можно увидеть в тестовой программе в разделе 17.3.1, так что вернитесь туда и взгляните еще раз. Обратите внимание на предусловия каждой операции и на то, что требуется для доказательства ее корректности.

Динамические фреймы позволяют гибко перекрывать наборы представлений, что довольно удобно, как показал пример с итератором. При использовании методов доказательства с неявными или предопределенными наборами представлений (см. раздел «Примечания» в конце главы 16) нет необходимости включать в спецификации инструкции **`modifies`** и **`reads`**. Однако создание итераторов требует дополнительных усилий, поскольку активный итератор должен иметь «разрешение на доступ» или «владеть» коллекцией. Передачу таких разрешений или прав владения можно организовать с помощью призрачного кода. Также необходимо предусмотреть действия на случай изменения коллекции, чтобы объявить итератор недействительным, потому что тогда коллекция должна восстановить все свои разрешения или право владения, чтобы выполнить дальнейшие операции изменения.

Справочный материал

Приложение A

Синтаксис Dafny

В этом приложении показаны фрагменты синтаксиса Dafny. Цель этого приложения – помочь вам вспомнить или узнать, как использовать различные конструкции в Dafny. Но имейте в виду, что это не исчерпывающий справочник по синтаксису языка программирования. Поэтому фрагменты приводятся без особых пояснений. Подробную информацию о синтаксисе ищите в справочном руководстве по Dafny [36].

A.0. Объявления

Программа на Dafny организуется как иерархия вложенных модулей. Зависимости между модулями определяются объявлениями **`import`**. Эти объявления не должны создавать циклических зависимостей. Экспортируемый набор модуля определяет, какие объявления, сделанные внутри модуля, будут видны импортирующему коду:

```
module MyModule {
  export
    provides A, B, C
    reveals D, E, F
  import L = LibraryA // L -- это локальный псевдоним для
                      // импортируемого модуля LibraryA
  import LibraryB    // сокращенная форма инструкции import LibraryB = LibraryB

  // далее следуют объявления типов и членов уровня модуля...
}
```

Самый внешний модуль программы является неявным. Поэтому в небольших программах методы и функции можно не заключать в объявление **`module`**.

A.0.0. Типы и объявления типов

Вот несколько примеров объявлений типов:

```
datatype Color = Brown | Blue | Hazel | Green
datatype Unary = Zero | Suc(Unary)
datatype List<X> = Nil | Cons(head: X, tail: List<X>)

class C<X> {
  // объявления членов класса...
}

type OpaqueType

type TypeSynonym = int
```

`X` в этих примерах является параметром типа.

Примеры типов:

```
bool        int         nat           real
set<X>      seq<X>      multiset<X>   map<X, Y>
char        string      X -> Y
()          (X, Y)      (X, Y, Z)
array<X>    array?<X>   array2<X>
object      object?     MyClass<X>    MyClass?<X>
```

Типы, показанные здесь в круглых скобках, обозначают кортежи с 0, 2 и 3 элементами.

А.0.1. Объявления членов

```
method M(a: A, b: B) returns (c: C, d: D)
  requires Pre
  modifies obj0, obj1, objectSet
  ensures Post // old(E) ссылается на значение E в точке входа в метод
  decreases E0, E1, E2
```

Объявления **constructor** (в классах) и **lemma** имеют тот же синтаксис, что и **method**. Чтобы объявить анонимный конструктор, просто опустите его имя:

```
function F(a: A, b: B): C
  requires Pre
  reads obj0, obj1, objectSet
  ensures Post // F(a, b) ссылается на результат функции
  decreases E0, E1, E2
```

Если `C` имеет тип **bool**, то первую строку объявления функции можно записать так:

```
predicate F(a: A, b: B)
```

Объявления полей и констант:

```
var b: B // изменяемые поля можно использовать только в классах
```

```
const n: nat
const greeting: string := "hello"
const year := 1402
```

Объявления **function**, **predicate**, **var** и **const** можно предварять ключевым словом **ghost**.

А.1. Инструкции

Каждая простая инструкция должна заканчиваться точкой с запятой (;). Составная инструкция с телом, заключенным в фигурные скобки, не должна заканчиваться точкой с запятой (;).

Объявление локальных переменных:

```
var x: X;
```

Часть «:X» можно опустить, если тип переменной легко определяется из контекста. При объявлении нескольких переменных двоеточие (:) имеет более высокий приоритет, чем запятая (,). То есть

```
var x, y: Y;
```

определит переменную y с типом Y, а тип переменной x будет определен из контекста автоматически.

Присваивание:

```
x := E;      // := произносится как «присвоить» или «назначить» (но НЕ «равно»!)
x, y := E, F; // одновременное присваивание
x :| E;       // присвоит переменной x значение, удовлетворяющее условию E
              // (присвоить такое значение, что)
```

Объявление переменной и присваивание ей значения можно объединить в одну инструкцию, например **var** x := E;.

Динамическое выделение объектов и массивов:

```
c := new C(...);
a := new T[n];
a := new T[n](i => ...);
```

Вызовы методов с 0, 1 и 2 выходными параметрами:

```
MethodWithNoResults(E, F);
x := MethodWithOneResult(E, F);
x, y := MethodWithTwoResults(E, F);
```

Другие простые инструкции:

```
assert E;     return;     return E, F, G;     new;
```

Некоторые составные инструкции:

```
if E {
  // инструкции...
} else {
  // инструкции...
}

if {
  case E0 => // инструкции...
  case E1 => // инструкции...
}

match E {
  case Pattern0(x, y) => // инструкции...
  case Pattern1(z, _) => // инструкции...
}

while Guard
  invariant Inv
  modifies obj0, obj1, objectSet
  decreases E0, E1, E2
{
  // инструкции...
}

forall x: X | Range {
  // инструкция присваивания
}

calc {
  E0;
== { assert HintWhyE0EqualsE1; }
  E1;
== { LemmaThatExplainsWhyE1EqualsE2(); }
  E2;
}
```

В инструкции **if** (в отличие от *выражения* **if-then-else**) ветвь **else** необязательна, но фигурные скобки обязательны. Если инструкция **if-case** или **match** стоит последней в списке инструкций, то фигурные скобки, окружающие варианты **case**, можно опустить. Без фигурных скобок варианты **case** стилистически не выделяются отступами и помещаются на одном уровне с ключевым словом **if** или **match**. Инструкция **forall** – это агрегатная инструкция, которая выполняет заданную инструкцию присваивания для каждого значения `x`, удовлетворяющего диапазону. Инструкция **calc** используется для написания структурированного доказательства.

А.2. Выражения

В табл. А.0 показаны наиболее часто используемые операторы. Операторы, перечисленные в одной строке таблицы, имеют одинаковый приоритет, при этом в табл. А.0 операторы упорядочены по возрастанию приоритетов.

Таблица А.0. Операторы и их приоритеты

Операторы	Значение
`<==>`	«тогда и только тогда, когда» (самый низкий приоритет)
`==>` `<==`	импликация, обратная импликация
`&&` `||`	логические И и ИЛИ
`==` `!=`	равно, не равно
`<` `<=` `=>` `>`	меньше, меньше или равно, больше или равно, больше
`in` `!in`	принадлежит, не принадлежит (коллекции)
`!!`	не пересекаются
`+` `-`	сложение/объединение/конкатенация/слияние, вычитание
`*` `/`	умножение/пересечение, деление, остаток от деления
`_ as int`	преобразование в целое число
`!` `-`	логическое НЕ, унарный минус
`_.x`	доступ к члену
`_[_]` `_[_ := _]`	выбор элемента, изменение элемента
`_[_ .. _]`	поддиапазон
`_[.. _]` `_[_ ..]`	взять, отбросить
`_[..]`	преобразовать массив элементов в последовательность

Для множеств `<=` (меньше или равно) обозначает подмножество, `+` (плюс) обозначает объединение, `*` (звездочка) обозначает пересечение и `-` (минус) обозначает разность множеств. Для мультимножеств эти операторы обозначают аналогичные операции с мультимножествами. Для последовательностей `<=` (меньше или равно) обозначает префикс, а `+` (плюс) обозначает конкатенацию. Для ассоциативных массивов `+` (плюс) обозначает объединение ассоциативных массивов (где правый операнд имеет более высокий приоритет) и `-` (минус) обозначает вычитание области ассоциативного массива. Оператор `<` (меньше) является более строгой версией `<=` (меньше или равно).

В выражении выбора члена `E.x`, `E` – это выражение (обычно значение ссылки или типа данных), а `x` – член типа `E`.

Выражение `E[J]` выбирает элемент `J` из `E`, где `E` – это массив, последовательность или ассоциативный массив, а `J` обозначает индекс массива/последовательности или ключ в ассоциативном массиве. Для мультимножества `E`, `E[J]` обозначает кратность элемента `J`. Элементы кортежа выбира-

ются с использованием элементов с числовыми именами; например, члены кортежа E с тремя элементами выбираются выражениями E.0, E.1 и E.2.

Если E – последовательность, ассоциативный массив или мультимножество, то выражение обновления E[J := V] возвращает коллекцию, подобную E, за исключением того, что в новой коллекции элемент J, ключ J или кратность J соответственно были заменены на V.

Для массива или последовательности E выражение выбора подпоследовательности E[lo..hi] представляет собой последовательность элементов hi - lo из E, начиная с lo. Если нижняя граница опущена, то по умолчанию считается, что она равна 0, а если опущена верхняя граница, то по умолчанию используется длина массива или последовательности. Для массива E выражение E[..] дает то же значение, что и E[0..E.Length], т. е. последовательность всех элементов E.

Если E – множество, мультимножество или последовательность, то выражение |E| обозначает общее количество элементов в E (которое известно как *мощность* множества или мультимножества и *длина* последовательности). Выражение E.Keys обозначает множество ключей в ассоциативном массиве E. Количество элементов в массиве E обозначается как E.Length, а длины измерений двумерного массива E можно получить как E.Length0 и E.Length1.

В следующей таблице показаны кортежи, отображения множеств, отображения мультимножеств, отображения последовательностей и отображения ассоциативных массивов с 0 и 3 элементами (или меньше для множества, если некоторые из a, b и c равны):

```
()              (a, b, c)
{}              {a, b, c}
multiset{}      multiset{a, b, c}
[]              [a, b, c]
map[]           map[x := a, y := b, z := c]
```

Вот некоторые литералы и другие выражения:

```
44      1.618   'D'       "hello"
this    null    old(E)    fresh(E)

seq(E, i => ...) // генератор последовательности

if E then E0 else E1

match E {
  case Pattern0(x, y) => E0
  case Pattern1(z, _) => E1
}

assert E0; E1   // аналогично E1, но сначала проверяется утверждение E0
```

```
MyLemma(); E    // аналогично E, но сначала вызывается MyLemma()

var x := E0; E1 // произносится как «присвоить переменной x значение E0 в E1»

set x: X | Range
forall x: X :: Expr // Expr обычно имеет форму E0 ==> E1
exists x: X :: Expr // в Expr часто используются &&, реже ==>
```

В выражении **if-then-else** (в отличие от инструкции **if**) ветвь **else** обязательна, а выражения `E0` и `E1` не заключаются в фигурные скобки.

Если выражения **match** не вкладываются друг в друга, то фигурные скобки можно опустить. Без фигурных скобок варианты **case** стилистически не выделяются отступами и помещаются на одном уровне с ключевым словом **match**.

Приложение B

Булева алгебра

В этом приложении приводится краткий обзор булевой алгебры и некоторых доказательств.

B.0. Булевы значения и отрицание

Логический тип имеет два значения: **`false`** и **`true`**.

С этими двумя значениями тесно связана унарная операция *отрицания*, также известная как «логическое НЕ». Во многих языках программирования она записывается как восклицательный знак (`!`), а в математике – как ¬. Таким образом, мы имеем:

```
!true  =  false
```

Отрицание – это *инволюция*, т. е. оно является обратным самому себе:

```
Двойное отрицание: !!X = X
```

В примере выше и во всем этом приложении переменные, такие как `X`, следует считать произвольным выражением соответствующего типа (здесь – логического).

Правило двойного отрицания гласит, что **`true`** и `!!`**`true`** – это одно и то же значение. Приведенное выше равенство говорит, что `!`**`true`** – это `false`, поэтому `!!`**`true`** – это `!`**`false`**. Отсюда следует, что:

```
true  =  !false
```

Когда булево выражение равно **`true`**, мы говорим, что оно *выполняется*.

B.1. Конъюнкция

Конъюнкция, широко известная как «логическое И» (или «логическое умножение»), – это двухместный оператор, который записывается как `&&` или `^`. Выражение `X && Y` истинно только тогда, когда истинны оба конъюнкта, `X` и `Y` (т. е. оба имеют значение **`true`**). Таким образом, **`true`** – это *левый единичный*

элемент в операции `&&`, а **`false`**, соответственно, – это *левый нулевой элемент* операции `&&`:

```
Единичный элемент:  true && X  =  X
Нулевой элемент:    false && X  =  false
```

Конъюнкция – *идемпотентна*, т. е. повторное применение к одному и тому же операнду не меняет значения результата; *коммутативна*, т. е. порядок операндов не меняет значения; и *ассоциативна*, т. е. круглые скобки можно перемещать:

```
Идемпотентность:    X && X  =  X
Коммутативность:    X && Y  =  Y && X
Ассоциативность:    X && (Y && Z)  =  (X && Y) && Z
```

Поскольку расположение круглых скобок не имеет значения для ассоциативного оператора, мы обычно их опускаем. В частности, мы пишем просто `X && Y && Z` для любого из двух выражений в предыдущей строке.

В силу коммутативности левый единичный элемент операции `&&` также является правым единичным элементом, а левый нулевой элемент – правым нулевым элементом.

Следующие свойства устанавливают связь между `!` и `&&`:

```
Дистрибутивность:            (X && !(Y && Z))  =  !(X && !Y) && !(X && !Z)
Закон исключенного третьего:          X && !X  =  false
```

Последнее из них гласит, что `X` и `!X` не могут быть оба равны **`true`**.

Упражнение B.0

а. Подставьте **`false`** вместо `X` в свойство дистрибутивности и используйте предыдущие свойства, чтобы показать выполнение свойства дистрибутивности.

б. Подставьте значение **`true`** вместо `Y` и `Z` в свойство дистрибутивности и используйте предыдущие свойства, чтобы показать выполнение свойства дистрибутивности.

B.2. Предикаты и корректность определений

Рассуждая о программах, мы пишем выражения, которые *в данном состоянии программы* могут иметь значение **`false`** или **`true`**. Такие логические выражения называются *предикатами*. Например, `a < 100` – это предикат, как и его отрицание `!(a < 100)` (т. е. `100 <= a`). Формулы, которые я написал выше (и все подобные формулы в этом приложении), также верны, когда `X`, `Y` и `Z` являются предикатами. Поэтому иногда можно услышать, как «булеву алгебру» называют «исчислением предикатов»; в этой книге я использую эти термины как взаимозаменяемые, обознача-

ющие одно и то же (но, если вы хотите более четко понять разницу, см., например, [42]).

Работая с предикатами, нас может волновать, насколько *корректно определены* выражения. Например, предикат `a == c / d` определен только в том случае, если `d` не равно нулю. Большинство языков программирования (включая Dafny) предлагают несколько ограниченную систему обозначений, обеспечивающую корректность определяемых выражений: логические операторы, вычисляемые по короткой схеме, `&&`, `||` и `==>`. Для этих операторов корректность определения правого операнда зависит от *значения* левого операнда.

Например, выражение

```
100 <= d && a == c / d
```

корректно определено, поскольку правый операнд `a == c / d` корректно определен, если выполняется условие `100 <= d`. А выражение

```
a == c/d && 100 <= d
```

не является корректно определенным. Другими словами, говоря о корректности, мы читаем формулы слева направо. Поэтому, хотя значение `X && Y` всегда дает тот же результат, что и `Y && X`, требования корректности могут помешать использовать один из этих вариантов (или оба).

Понятие корректности я уточнил в разделе 2.12. В оставшейся части приложения я будут использовать только значения логических выражений, на которые не влияет природа операторов, вычисляемых по короткой схеме.

B.3. Дизъюнкция и правила доказательства

Дизъюнкция, известная также как «логическое ИЛИ» (или «логическое сложение»), – это двухместный оператор, обозначаемый как `||` или `∨`. Выражение `X || Y` истинно только тогда, когда хотя бы один из дизъюнктов, `X` или `Y`, является истинным. Дизъюнкцию можно определить в терминах отрицания и конъюнкции, используя правило, приписываемое Де Моргану:

```
Закон Де Моргана:   !(X && Y)  =  !X || !Y
```

Из этого правила вытекает несколько свойств:

```
Единичный элемент:     false || X  =  X
Нулевой элемент:       true || X  =  true
Идемпотентность:       X || X  =  X
Коммутативность:       X || Y  =  Y || X
Ассоциативность:       X || (Y || Z)  =  (X || Y) || Z
```

Вследствие коммутативности левый единичный и левый нулевой элементы также являются правым единичным и правым нулевым элементами соответственно.

Напишем доказательство. Докажем альтернативную формулировку закона Де Моргана:

```
Закон Де Моргана: !(X || Y) = !X && !Y
```

Запишем доказательство в виде последовательности шагов, каждый из которых использует предыдущее определение или ранее доказанное свойство.

```
  !(X || Y)
=      { Двойное отрицание (применяется в 2 местах) }
  !(!!X || !!Y)
=      { Первая формулировка закона Де Моргана, приведенная выше }
  !!(!X && !Y)
= { Двойное отрицание }
  !X && !Y
```

Поскольку на каждом шаге этого «доказательного вычисления» равенство выполняется, мы приходим к заключению, что первая строка равна последней. Таким образом, мы доказали альтернативную формулировку закона Де Моргана.

Напишем еще одно доказательство, на этот раз для свойства ассоциативности ||:

```
  X || (Y || Z)
=    { Двойное отрицание (применяется в 2 местах) }
  !!X || !!(Y || Z)
=    { Закон Де Моргана }
  !(!X && !(Y || Z))
=    { Закон Де Моргана (альтернативная формулировка) }
  !(!X && (!Y && !Z))
=    { Ассоциативность операции && }
  !((!X && !Y) && !Z)
=    { Закон Де Моргана (альтернативная формулировка) }
  !(!(X || Y) && !Z)
=   { Закон Де Моргана }
  !!(X || Y) || !!Z
=    { Двойное отрицание (применяется в 2 местах) }
  (X || Y) || Z
```

Упражнение В.1

В разделе В.0 утверждается, что **`true`** равно `!`**`false`**. Напишите доказательство в виде доказательных утверждений.

Упражнение В.2

Докажите свойства единичного элемента, нулевого элемента, идемпотентности и коммутативности для ||.

Упражнение B.3

Докажите два дополнительных варианта закона Де Моргана:

а) `X || Y = !(!X && !Y)`;

б) `X && Y = !(!X || !Y)`.

Логическое И дистрибутивно относительно логического ИЛИ, а логическое ИЛИ дистрибутивно относительно логического И:

```
Дистрибутивность:   X || (Y && Z)  =  (X || Y) && (X || Z)
Дистрибутивность:   X && (Y || Z)  =  (X && Y) || (X && Z)
```

Вот доказательство первого из них:

```
  X || (Y && Z)
=    { Упражнение B.3 }
  !(!X && !(Y && Z)
=    { Свойства дистрибутивности ! и && }
  !(!X && !Y) && !(!X && !Z)
=    { Закон Де Моргана (применяется в 2 местах) }
  (X || Y) && (X || Z)
```

Упражнение B.4

Докажите второе свойство дистрибутивности (логическое И дистрибутивно относительно логического ИЛИ).

Наконец, вот альтернативная формулировка закона исключенного третьего:

```
Закон исключенного третьего:   X || !X  =  true
```

Упражнение B.5

Докажите эту формулировку закона исключенного третьего.

B.4. Импликация

Логическая импликация (или логическое следование) – это двухместный оператор, обозначаемый как `==>` или ⇒. Выражение `X ==> Y` истинно только тогда, когда `X` ложно или `Y` истинно. Другими словами, `X ==> Y` говорит, что `Y` истинно всегда, когда `X` истинно. Его можно читать как «`X` подразумевает `Y`», «`X` истинно, только если `Y` истинно», «`X` сильнее `Y`» или «`Y` слабее `X`». В выражении `X ==> Y` операнд `X` называют *посылкой*, а `Y` – *заключением*.

```
Импликация: X ==> Y = !X || Y
```

Оператор `==>` имеет меньший приоритет, чем `&&` и `||`. Это означает, что такое выражение, как

```
(W || X)  ==>  (Y && Z)
```

можно записать без скобок:

```
W || X  ==>  Y && Z
```

По этой причине в синтаксисе Dafny оператор `==>` имеет ширину 3 символа; визуально это говорит, что `==>` охватывает более широкую область в выражении, чем операнды `&&` и `||`, как бы подсказывая, что сначала применяются операторы, операнды которых ближе друг к другу, т. е. отделены меньшим количеством символов. Позвольте мне еще больше преувеличить этот эффект, записав выражение следующим образом:

```
W || X    ==>    Y && Z
```

Оператор `==>` – правоассоциативный, т. е.

```
X ==> Y ==> Z
```

означает то же, что и:

```
X ==> (Y ==> Z)
```

Вот несколько замечательных свойств импликации – настолько замечательных, что у каждого есть имя (а у одного даже латинское название 😊):

```
Modus Ponens (правило вывода):       X && (X ==> Y)  =  X && Y
Контрапозиция (противопоставление):  X ==> Y  =  !Y ==> !X
Преобразование конъюнкции посылок в дизъюнкцию заключений:
  X && Y ==> Z  =  X ==> !Y || Z
Дистрибутивность:                    X || Y ==> Z  =  (X ==> Z) && (Y ==> Z)
```

Упражнение В.6

Докажите свойства:

а) Modus Ponens;

б) контрапозиции;

в) преобразования конъюнкции посылок в дизъюнкцию заключений;

г) дистрибутивности;

д) `X || (!X ==> Y) = X || Y`;

е) `X ==> Y && Z = (X ==> Y) && (X ==> Z)`.

В.5. Доказательство истинности импликации

Вот еще два замечательных свойства импликации:

```
Рефлексивность: X ==> X
Транзитивность: (X ==> Y) && (Y ==> Z) ==> (X ==> Z)
```

Докажем эти свойства. Обратите внимание, что каждая из формул имеет вид импликации. Это обычное явление для проверяемых условий, с которыми мы сталкиваемся, поэтому рассмотрим некоторые методы их доказательства.

Самый простой способ показать, что формула импликации всегда выполнима, – доказать, что она истинна (равна **true**). Например, мы можем доказать свойство рефлексивности с помощью следующего доказательного вычисления:

```
  X ==> X
=    { Определение ==> }
  !X || X
=    { Закон исключения третьего }
  true
```

Иногда формула импликации, которую требуется доказать, имеет большой размер, поэтому начинать доказательное вычисление на основе всей формулы может оказаться трудной задачей. Второй метод доказательства импликации типа `A ==> B` состоит в том, чтобы «предположить `A`», а затем доказать `B`. Это равносильно доказательству того, что условие `B` выполняется, с использованием свойств `A` в этом доказательстве. (Я не буду приводить здесь все формальные подробности, но следующий пример должен показаться вам достаточно иллюстративным.) Третий метод доказательства импликации типа `A ==> B` состоит в том, чтобы начать доказательное вычисление с `A` и, используя шаги равенства и импликации, в конечном итоге прийти к `B`. Используем второй и третий методы для доказательства свойства транзитивности: предположим, что `(X ==> Y) && (Y ==> Z)`, а затем докажем `X ==> Z`:

```
  X
==>    { Использование предположения X ==> Y }
  Y
==>    { Использование предположения Y ==> Z }
  Z
```

Упражнение B.7

Докажите истинность всех следующих формул:

а) `(P(x) && Q(y) ==> R(x, y)) && !R(x, y) && P(x)  ==>  !Q(y);`

б) `(5 < x ==> y == 10) && y < 7 && (y < 1000 ==> x <= 5)  ==>  x == 5.`

Для доказательства формулы «б» используйте некоторые простые арифметические свойства.

B.6. Свободные переменные и подстановка

Переменные, встречающиеся в выражении, могут быть *свободными* или *связанными*. Связанная переменная – это переменная, область видимости которой ограничена некоторой частью выражения, и она недоступна на уровне всего выражения. Вот типичный пример:

$$k + \sum_{i:=0}^{<n} F(i, j).$$

Здесь присутствуют четыре переменные: `i`, `j`, `k` и `n`. Из них `i` – связанная переменная, поскольку ее область видимости ограничена суммированием, тогда как переменные `j`, `k` и `n` – свободные. Другой пример:

```
x < 100 && 20 <= Fib(y)
```

Здесь обе переменные, `x` и `y`, – свободные.

Связанную переменную можно переименовать, т. е. дать ей любое другое имя, которое не используется. Например, выражение в первом примере выше можно без изменения смысла записать как:

$$k + \sum_{w:=0}^{<n} F(w, j).$$

Здесь переменная `i` была переименована в `w`. Операцию изменения имени связанной переменной по малопонятным причинам называют *альфа-переименованием*. Тот факт, что альфа-переименование никогда не меняет смысла выражения, отражает идею о том, что связанные переменные являются «частными» для той ограниченной области, в которой они определены.

Для любых выражений `E` и `F` и переменной `x` выражение подстановки `E[x := F]` обозначает выражение `E`, в котором все вхождения свободной переменной `x` заменены на `F`. Например:

```
(x < 100 && 20 <= Fib(y))[[y := 2*x + y]] =
  x < 100 && 20 <= Fib(2*x + y)
```

Однако есть одно ограничение: замену (подстановку) можно применять только в том случае, если она не приводит к *захвату переменной*. Захват переменной происходит, когда свободная переменная `F` становится связанной переменной результата подстановки. Например, если бы мы попытались применить подстановку `[j := 2*i]` к первому примеру в этом разделе, то получили бы:

$$k + \sum_{i:=0}^{<n} F(i, 2*i),$$

где `i` в постановке стала привязанной к суммированию. Такая подстановка не допускается, потому что конкретные имена, используемые для связанных переменных, не должны иметь значения. Дело в том, что `i` в `[j := 2*i]` относится к некоторой переменной, которую не следует путать со связанной переменной `i` в подставляемом выражении.

Чтобы применить подстановку `[x := F]` к выражению `E`, не опасаясь захвата переменных, можно сначала выполнить альфа-переименование для всех связанных переменных в `E`. Например, чтобы применить подстановку `[j := 2*i]` к первому примеру в этом разделе, можно сначала переименовать связанную переменную `i` в `w`, а затем применить подстановку. Результат будет выглядеть так:

$$k + \sum_{w:=0}^{<n} F(w, 2 * i).$$

Выполняя подстановку, мы всегда (явно или неявно) применяем альфа-переименование, чтобы избежать проблем с захватом переменных. Иногда в таких случаях можно услышать фразу: «Подстановка без риска захвата».

Упражнение B.8

Выполните подстановку `[x := x - y]` в следующих выражениях. Не забудьте сначала применить альфа-переименование, если это необходимо:

а) `2 * x * t < 100`;

б) `y + x`;

в) `x + y`;

г) `t + 5`;

д) $\int_a^b \sin(x + y)\, dx$;

е) $\int_a^b \sin(x + y)\, dy$;

ж) $\int_x^y \sin z\, dz$.

Подстановку также можно производить сразу для нескольких переменных. Например, `E[x, y := F, G]` требует одновременной замены всех вхождений свободной переменной `x` на `F` и всех вхождений свободной переменной `y` на `G`. Например:

```
(F(x) + G(y))[[x, y := x + y, x - y]]   =   F(x + y) + G(x - y)
```

Упражнение B.9

Примените подстановку `[x, y, z := x + y, 10, x - y]` к каждому из следующих выражений:

а) `y + 4 * x`;

б) `Factorial(y + t) + z`;

в) $x^2 + y^2 == z^2$.

B.7. Квантор всеобщности

Квантор всеобщности – это своего рода «конъюнкция на стероидах». Его используют, когда нужно выразить, что предикат `P(x)` выполняется для всех значений `x`. Этот квантор записывается так:

```
forall x :: P(x)
```

или, на языке математики: $\forall x \cdot P(x)$. Переменная *x* привязана к выражению. Обычно тип *x* можно определить из контекста, и поэтому он опускается, однако при необходимости его можно указать явно:

```
forall x: T :: P(x)
```

Вот некоторые важные свойства, которыми обладает квантор всеобщности:

```
Дистрибутивность:
  (forall x :: E && F) = (forall x :: E) && (forall x :: F)
Правило элиминации квантора всеобщности по одноэлементному множеству:
  (forall x :: x == F ==> E)  =  E[[x := F]]
  при условии, что x не является свободной переменной в F
Неиспользуемая переменная:
  (forall x :: E)  =  E
  при условии, что x не является свободной переменной в E и тип x -- не пустой
Константа:
  (forall x :: true)  =  true
```

Упражнение B.10

Докажите следующее свойство, которое часто называют разделением *области значений квантифицированной переменной* (Range Split):

```
(forall x :: P || Q ==> R) =
  (forall x :: P ==> R) && (forall x :: Q ==> R)
```

Упражнение B.11

Пусть тип `x` не пустой и `x` не является свободной переменной в `E`, `F` или `P`. Докажите:

а) `(forall x :: x == E || x == F ==> R) =`
`R[[x := E]] && R[[x := F]];`

б) `P && (forall x :: Q)  =  (forall x :: P && Q)`.

Упражнение B.12

Докажите:

а) `!forall x :: false`
`при условии, что тип x не пустой;`

б) `(forall x :: false ==> P);`

в) `(forall x :: P) ==> P[[x := E].`

Упражнение B.13

а. Докажите:

```
(forall x :: P) || (forall x :: Q) ==> (forall x :: P || Q)
```

б. Приведите примеры `P` и `Q`, для которых следующее условие не выполняется:

```
(forall x :: P || Q) ==> (forall x :: P) || (forall x :: Q)
```

Упражнение B.14

Пусть тип `x` не пустой и `x` не встречается в `P`. Докажите:

а) `P || (forall x :: Q)  =  (forall x :: P || Q);`

б) `P ==> (forall x :: Q)  =  (forall x :: P ==> Q).`

Упражнение B.15

Учитывая, что `m < n`, докажите:

а) `(forall i :: m <= i < n ==> P(i)) =`
`P(m) && (forall i :: m + 1 <= i < n ==> P(i));`

б) `(forall i :: m <= i < n ==> P(i)) =`
`(forall i :: m <= i < n - 1 ==> P(i)) && P(n - 1);`

в) `(forall i :: 0 <= i < 3 ==> P(i)) =`
`P(0) && P(1) && P(2).`

B.8. Квантор существования

Пару квантору всеобщности составляет *квантор существования*, своего рода «дизъюнкция на стероидах». Его используют, когда нужно выразить, что предикат `P(x)` выполняется для *некоторого* значения `x`. Этот квантор записывается так:

```
exists x :: P(x)
```

или, на языке математики: $\exists x \cdot P(x)$. Его можно определить через обобщенную форму закона Де Моргана, сформулированную ниже двумя разными способами:

Закон де Моргана для кванторов:

```
!(forall x :: P)  =  (exists x :: !P)
!(exists x :: P)  =  (forall x :: !P)
```

Упражнение В.16

Докажите вторую формулировку закона Де Моргана для кванторов на основе первой.

Квантор существования имеет свойства, аналогичные свойствам квантора всеобщности.

Упражнение В.17

Докажите:

а) `(exists x :: E && F) ==>`
`(exists x :: E) && (exists x :: F);`

б) `(exists x :: x == F && E)  =  E[[x := F]]`
при условии, что `x` не является свободной переменной в `F`;

в) `(exists x :: E && F)  =  E && exists x :: F`
при условии, что `x` не является свободной переменной в `E` и тип `x` не пустой;

г) `(exists x :: E)  =  E`
при условии, что `x` не является свободной переменной в `E` и тип `x` не пустой;

д) `(exists x :: false)  =  false`.

Упражнение В.18

Пусть `x` не является свободной переменной в `P`. Докажите:

```
P || (exists x :: Q) = (exists x :: P || Q)
```

Упражнение В.19

Докажите:

а) `exists x :: true`
при условии, что тип `x` не пустой;

б) `!(exists x :: false && P);`

в) `P[[x := E]] ==> (exists x :: P)`.

Упражнение В.20

Докажите:

а) `(exists x :: P) || (exists x :: Q) ==>`
`(exists x :: P || Q);`

б) `(exists x :: P || Q) ==>`
`(exists x :: P) || (exists x :: Q).`

Упражнение B.21

Пусть тип `x` не пустой и `x` не встречается в `P`. Докажите:

а) `P || (exists x :: Q) = (exists x :: P || Q);`

б) `P ==> (exists x :: Q) = (exists x :: P ==> Q).`

Упражнение B.22

Учитывая, что `m < n`, докажите:

а) `(exists i :: m <= i < n && P(i)) =`
`P(m) || (exists i :: m + 1 <= i < n && P(i));`

б) `(exists i :: m <= i < n && P(i)) =`
`(exists i :: m <= i < n - 1 && P(i)) || P(n - 1);`

в) `(exists i :: 0 <= i < 3 && P(i)) =`
`P(0) || P(1) || P(2).`

Упражнение B.23

Пусть `x` не является свободной переменной в `Q`. Докажите:

`(exists x :: P) ==> Q = (forall x :: P ==> Q)`

Упражнение B.24

Для `x` и `y` целочисленного типа докажите:

а) `exists x :: x * x == 9;`

б) `forall x :: 0 <= x < 100 ==>`
`exists y :: 0 <= y < 10 && y * y <= x < (y + 1) * (y + 1);`

в) `exists x :: forall y :: 0 <= y ==> xy == x;`

г) `forall x :: exists y :: x < y.`

Примечания

Математики любят экспериментировать с определениями разных видов «логики». Это помогает им понять, насколько выразительными могут быть базовые принципы, и позволяет сравнивать выразительность различных видов «логики». Результаты таких экспериментов повлияли на разработку языков программирования и систем типов. Например, *линейная логика* повлияла на системы типов, которые позволяют отслеживать расходование ресурсов (например, памяти) [124, 115].

Несмотря на всю очевидность для простых логических значений, *закон исключенного третьего* часто оказывается нереализованным в некоторых

системах доказательства корректности для предикатов или типов. Системы, где это закон не реализован, называются *конструктивными* [34, 111], а системы, включающие его, – *интуиционистскими*. В Dafny закон исключенного третьего реализован.

Приложение C

Решения некоторых упражнений

Ответ на упражнение 2.34

Да, при условии, что это также приведет к сбою вашей программы. Например, из формулы

```
SP⟦assert true, CandyCrushScore == 50⟧ =
  CandyCrushScore == 50
```

мы знаем, что *если* начать счет с `50` и *если* программа завершится без сбоя, то результат останется равным `50`. Однако формула ничего не говорит о конечном состоянии, если инструкция завершится сбоем, а *SP* не сообщает нам, когда это произойдет.

Ответ на упражнение 2.35

```
NOCRASH⟦x := E, P⟧ =
  true
NOCRASH⟦assert E, P⟧ =
  P ==> E
NOCRASH⟦S;T, P⟧ =
  NOCRASH⟦S, P⟧ && NOCRASH⟦T, SP⟦S, P⟧ ⟧
NOCRASH⟦if B { S } else { T }, P⟧ =
  NOCRASH⟦S, P && B⟧ && NOCRASH⟦T, P && !B⟧
```

Ответ на упражнение 3.10

а. `x ≻ x - 2`.
б. `x - 2 < x && 0 <= x`.

Ответ на упражнение 3.13

Для `Outer` используйте **`decreases`** `a`.
Для `Inner` используйте **`decreases`** `a, b`.

Ответ на упражнение 4.1

```
function ReverseColors(t: BYTree): BYTree {
  match t
  case BlueLeaf => YellowLeaf
  case YellowLeaf => BlueLeaf
 case Node(left, right) => Node(ReverseColors(left), ReverseColors(right))
}
```

Ответ на упражнение 4.5

Предусловие деструктора можно удовлетворить с использованием выражения **if-then-else**:

```
if t.Node? then t.left == u else false
```

или с помощью оператора &&, вычисляемого по короткой схеме:

```
t.Node? && t.left == u
```

Обратите внимание, что проверка t.Node? должна производиться до использования деструктора; было бы ошибкой написать:

```
t.left == u && t.Node? // ошибка
```

Ответ на упражнение 5.6

```
lemma Ack1(n: nat)
  ensures Ack(1, n) == n + 2
{
  if n == 0 {
    // тривиальный случай
  } else {
    calc {
      Ack(1, n);
    ==  // определение Ack
      Ack(0, Ack(1, n - 1));
    ==  // определение Ack(0, _)
      Ack(1, n - 1) + 1;
    ==  { Ack1(n - 1); } // индуктивное предположение
      (n - 1) + 2 + 1;
    ==  // арифметика
      n + 2;
    }
  }
}
```

Ответ на упражнение 5.13

Следующее доказательство подробно разъясняет каждый случай:

```
lemma {:induction false} OceanizeIdempotent(t: BYTree)
  ensures Oceanize(Oceanize(t)) == Oceanize(t)
{
  match t
  case BlueLeaf =>
    calc {
      Oceanize(Oceanize(BlueLeaf));
    ==  // определение Oceanize
      Oceanize(BlueLeaf);
    }
  case YellowLeaf =>
    calc {
      Oceanize(Oceanize(YellowLeaf));
    ==  // определение Oceanize
      Oceanize(BlueLeaf);
    ==  // определение Oceanize
      BlueLeaf;
    ==  // определение Oceanize
      Oceanize(YellowLeaf);
    }
  case Node(left, right) =>
    calc {
      Oceanize(Oceanize(Node(left, right)));
    ==  // определение Oceanize
      Oceanize(Node(Oceanize(left), Oceanize(right)));
    ==  // определение Oceanize
      Node(Oceanize(Oceanize(left)), Oceanize(Oceanize(right)));
    ==  { OceanizeIdempotent(left); }
      Node(Oceanize(left), Oceanize(Oceanize(right)));
    ==  { OceanizeIdempotent(right); }
      Node(Oceanize(left), Oceanize(right));
    ==  // определение Oceanize
      Oceanize(Node(left, right));
  }
}
```

Ответ на упражнение 5.14

```
lemma {:induction false} OceanizeUpsBlueCount(t: BYTree)
  ensures BlueCount(t) <= BlueCount(Oceanize(t))
{
  match t
  case BlueLeaf =>
  case YellowLeaf =>
  case Node(left, right) =>
    calc {
```

```
      BlueCount(Node(left, right));
    ==  // определение BlueCount
      BlueCount(left) + BlueCount(right);
    <=  { OceanizeUpsBlueCount(left); }
      BlueCount(Oceanize(left)) + BlueCount(right);
    <=  { OceanizeUpsBlueCount(right); }
      BlueCount(Oceanize(left)) + BlueCount(Oceanize(right));
    == // определение BlueCount
      BlueCount(Node(Oceanize(left), Oceanize(right)));
    ==  // определение Oceanize
      BlueCount(Oceanize(Node(left, right)));
  }
}
```

Ответ на упражнение 5.15

Вот инструкция **calc** для случая `Cons`:

```
calc {
  EvalList(SubstituteList(args, n, c), op, env);
==  // args == Cons(e, tail)
  EvalList(SubstituteList(Cons(e, tail), n, c), op, env);
==  // определение SubstituteList
  EvalList(Cons(Substitute(e, n, c),
           SubstituteList(tail, n, c)), op, env);
==  // определение EvalList
  var v0, v1 :=
    Eval(Substitute(e, n, c), env),
    EvalList(SubstituteList(tail, n, c), op, env);
  match op
  case Add => v0 + v1
  case Mul => v0 * v1;
==  { EvalSubstitute(e, n, c, env); }
  var v0, v1 :=
    Eval(e, env[n := c]),
    EvalList(SubstituteList(tail, n, c), op, env);
  match op
  case Add => v0 + v1
  case Mul => v0 * v1;
==  { EvalSubstituteList(tail, op, n, c, env); }
  var v0, v1 :=
    Eval(e, env[n := c]),
    EvalList(tail, op, env[n := c]);
  match op
  case Add => v0 + v1
  case Mul => v0 * v1;
==  // определение EvalList
```

```
    EvalList(Cons(e, tail), op, env[n := c]);
  ==  // args == Cons(e, tail)
    EvalList(args, op, env[n := c]);
  }
```

Ответ на упражнение 5.16

```
lemma EvalEnv(e: Expr, n: string, env: map<string, nat>)
  requires n in env.Keys
  ensures Eval(e, env) == Eval(Substitute(e, n, env[n]), env)
{
  EvalSubstitute(e, n, env[n], env);
  assert env == env[n := env[n]]; // требуется для расширяемости типов
}
```

Ответ на упражнение 6.0

```
function Length'<T>(xs: List<T>): nat {
  if xs == Nil then 0 else 1 + Length'(xs.tail)
}

lemma LengthLength'<T>(xs: List<T>)
  ensures Length(xs) == Length'(xs)
{
}
```

Dafny автоматически доказывает эту лемму, вам нужно лишь написать пустое тело леммы: {}.

Ответ на упражнение 6.3

```
lemma {:induction false} AppendNil<T>(xs: List<T>)
  ensures Append(xs, Nil) == xs
{
  match xs
  case Nil =>
  case Cons(x, tail) =>
    calc {
      Append(xs, Nil);
    ==  // определение Append
      Cons(x, Append(tail, Nil));
    ==  { AppendNil(tail); }
      Cons(x, tail);
    ==
      xs;
  }
}
```

Ответ на упражнение 6.6

```
lemma UnitsAreTheSame()
  ensures L == R
{
  calc {
    L;
  ==  { RightUnit(L); }
    F(L, R);
  ==  { LeftUnit(R); }
    R;
  }
}
```

Ответ на упражнение 6.8

```
function LiberalTake<T>(xs: List<T>, n: nat): List<T>
{
  if n == 0 || xs == Nil then
    Nil
  else
    Cons(xs.head, LiberalTake(xs.tail, n - 1))
}

function LiberalDrop<T>(xs: List<T>, n: nat): List<T>
{
  if n == 0 || xs == Nil then
    xs
  else
    LiberalDrop(xs.tail, n - 1)
}

lemma TakesDrops<T>(xs: List<T>, n: nat)
  requires n <= Length(xs)
  ensures Take(xs, n) == LiberalTake(xs, n)
  ensures Drop(xs, n) == LiberalDrop(xs, n)
{
}
```

Ответ на упражнение 6.14

В постусловии `AtFind` вторым аргументом в вызове функции `At` является `Find(xs, y)`. Внутренняя спецификация `Find` сообщает нам, что `Find(xs, y)` не превышает `Length(xs)`. Более того, поскольку оператор `||` вычисляется по короткой схеме, `At` вызывается только в том случае, если первый дизъюнкт (`Find(xs, y) == Length(xs)`) не выполняется. Эти факты подразумевают соответствие предусловию вызова `At`.

Для `BeforeFind` вторым аргументом в вызове функции At является `i`. Поскольку оператор `==>` вычисляется по короткой схеме, `At` вызывается только в том случае, если выполняется посылка `i < Find(xs, y)`. Таким образом, внутренняя спецификация `Find` позволяет нам обеспечить выполнение предусловия вызова `At`.

Ответ на упражнение 7.2

```
predicate Less(x: Unary, y: Unary) {
  y != Zero && (x == Zero || Less(x.pred, y.pred))
}
```

Ответ на упражнение 7.3

```
lemma LessTrichotomous(x: Unary, y: Unary)
  // 1 или 3 из них истинно:
  ensures Less(x, y) <==> x == y <==> Less(y, x)
  // не все 3 истинны:
  ensures !(Less(x, y) && x == y && Less(y, x))
{
}
```

Обратите внимание, что в этом решении используется тот факт, что `<==>` является ассоциативным, а не последовательным. То есть в

```
Less(x, y) <==> x == y <==> Less(y, x)
```

можно, например, добавить в круглые скобки:

```
Less(x, y) <==> (x == y <==> Less(y, x))
```

Ответ на упражнение 7.5

```
lemma {:induction false} AddCorrect(x: Unary, y: Unary)
  ensures UnaryToNat(Add(x, y)) == UnaryToNat(x) + UnaryToNat(y)
{
  match y
  case Zero =>
  case Suc(y') =>
    calc {
      UnaryToNat(Add(x, y));
    ==  // y == Suc(y')
      UnaryToNat(Add(x, Suc(y')));
    ==  // определение Add
      UnaryToNat(Suc(Add(x, y')));
    ==  // определение UnaryToNat
      1 + UnaryToNat(Add(x, y'));
    ==  { AddCorrect(x, y'); }
      1 + UnaryToNat(x) + UnaryToNat(y');
```

```
    ==  // определение UnaryToNat
      UnaryToNat(x) + UnaryToNat(Suc(y'));
    ==  // y == Suc(y')
      UnaryToNat(x) + UnaryToNat(y);
    }
}

lemma {:induction false} SucAdd(x: Unary, y: Unary)
  ensures Suc(Add(x, y)) == Add(Suc(x), y)
{
  match y
  case Zero =>
  case Suc(y') =>
    calc {
      Suc(Add(x, Suc(y')));
    ==  // определение Add
      Suc(Suc(Add(x, y')));
    ==  { SucAdd(x, y'); }
      Suc(Add(Suc(x), y'));
    ==  // определение Add
      Add(Suc(x), Suc(y'));
    }
}

lemma {:induction false} AddZero(x: Unary)
  ensures Add(Zero, x) == x
{
  match x
  case Zero =>
  case Suc(x') =>
    calc {
      Add(Zero, Suc(x'));
    ==  // определение Add
      Suc(Add(Zero, x'));
    ==  { AddZero(x'); }
      Suc(x');
    }
}
```

Ответ на упражнение 9.0

```
module ImmutableQueue {
  import LL = ListLibrary

  type Queue<A>
  function Empty(): Queue
  function Enqueue<A>(q: Queue, a: A): Queue
```

```
  function Dequeue<A>(q: Queue): (A, Queue)
    requires q != Empty<A>()

  ghost function Length(q: Queue): nat
  lemma EmptyCorrect<A>()
    ensures Length(Empty<A>()) == 0
  lemma EnqueueCorrect<A>(q: Queue, x: A)
    ensures Length(Enqueue(q, x)) == Length(q) + 1
  lemma DequeueCorrect(q: Queue)
    requires q != Empty()
    ensures Length(Dequeue(q).1) == Length(q) - 1
}
```

Ответ на упражнение 9.5

В `Client()` мы записали это равенство в утверждении **`assert`**, которое является призрачной инструкцией. В призрачных контекстах Dafny поддерживает сравнение на равенство для всех типов.

Ответ на упражнение 9.6

```
module QueueExtender {
  import IQ = ImmutableQueue
  function TryDequeue<A>(q: IQ.Queue, default: A): (A, IQ.Queue)
  {
    if IQ.IsEmpty(q) then (default, q) else IQ.Dequeue(q)
  }
}
```

Ответ на упражнение 10.0

Чтобы доказать корректность этой программы, нужно вспомогательное утверждение **`assert`**, которое напомнит верификатору сведения о `Elements(pq)` после двух вставок:

```
module PriorityQueueTestHarness {
  import PQ = PriorityQueue
  method Test(x: int, y: int) {
    PQ.EmptyCorrect(); var pq := PQ.Empty();
    PQ.InsertCorrect(pq, x);  pq := PQ.Insert(pq, x);
    PQ.InsertCorrect(pq, y);  pq := PQ.Insert(pq, y);
    assert PQ.Elements(pq) == multiset{x,y};
    PQ.IsEmptyCorrect(pq); PQ.RemoveMinCorrect(pq);
    var (a, pq') := PQ.RemoveMin(pq);
    PQ.IsEmptyCorrect(pq'); PQ.RemoveMinCorrect(pq');
    var (b, pq') := PQ.RemoveMin(pq');
    assert {a,b} == {x,y} && a <= b;
  }
}
```

Все эти леммы помогают достичь желаемого результата, но излишне загромождают код и делают его не очень опрятным. Мы займемся упрощением в разделе 10.3.

Ответ на упражнение 10.2

```
lemma {:induction false} BinaryHeapStoresMinimum(pq: PQueue, y: int)
  requires IsBinaryHeap(pq) && y in Elements(pq)
  ensures pq.x <= y
{
  // Согласно определению Elements, рассмотрим три случая,
  // к которым приводит «y in Elements(pq)».
  if
  case y == pq.x =>
    // тривиальный случай
  case y in Elements(pq.left) =>
    calc {
      pq.x;
    <=  // определение IsBinaryHeap
      pq.left.x;
    <=  { BinaryHeapStoresMinimum(pq.left, y); }
      y;
    }
  case y in Elements(pq.right) =>
    calc {
      pq.x;
    <=  // определение IsBinaryHeap
      pq.right.x;
    <=  { BinaryHeapStoresMinimum(pq.right, y); }
      y;
    }
}
```

Ответ на упражнение 10.3

Смотрите раздел 10.3.0.

Ответ на упражнение 10.4

```
lemma BalanceConsequence(pq: PQueue)
  requires !IsEmpty(pq) && IsBalanced(pq)
  ensures pq.left == Leaf ==> pq.right == Leaf
{
}
```

Ответ на упражнение 11.7

```
UpWhileLess:
  invariant i <= N
```

```
    decreases N - i

  UpWhileNotEqual:
    invariant i <= N
    decreases N - i

  DownWhileNotEqual:
    invariant 0 <= i
    decreases i

  DownWhileGreater:
    invariant 0 <= i
    decreases i
```

Ответ на упражнение 12.5

```
method FastExp(b: nat, n: nat) returns (p: nat)
  ensures p == Exp(b, n)
{
  p := 1;
  var d, k := b, n;
  while k != 0
    invariant p * Exp(d, k) == Exp(b, n)
  {
    calc {
      Exp(d, k);
    ==  { assert k == 2 * (k / 2) + k % 2; }
      Exp(d, 2 * (k / 2) + k % 2);
    ==  { ExpAddExponent(d, k / 2 * 2, k % 2); }
      Exp(d, 2 * (k / 2)) * Exp(d, k % 2);
    ==  { ExpSquareBase(d, k / 2); }
      Exp(d * d, k / 2) * Exp(d, k % 2);
    }
    if k % 2 == 1 {
      assert Exp(d, k % 2) == d;
      p := p * d;
    }
    d, k := d * d, k / 2;
  }
}
```

Ответ на упражнение 12.8

```
lemma {:induction false} AppendSumUp(lo: int, hi: int)
  requires lo < hi
  ensures SumUp(lo, hi - 1) + F(hi - 1) == SumUp(lo, hi)
  decreases hi - lo
{
```

```
    if lo == hi - 1 {
    } else {
      AppendSumUp(lo + 1, hi);
    }
  }
```

Ответ на упражнение 13.3

```
forall i, j :: 0 <= i < j < a.Length ==> a[i] < a[j]
```

Ответ на упражнение 13.14

Добавьте случай:

```
case a[m] == b[n] =>
  return 0;
```

и (необязательно) во всех остальных условиях замените `<=` на `<`.

Ответ на упражнение 13.19

Поскольку известно, что `lo` и `hi` по меньшей мере равны `0`, можно написать:

```
mid := lo + (hi - lo) / 2;
```

Ответ на упражнение 14.0

Добавьте:

```
requires a.Length == 1 ==> left == right
```

Ответ на упражнение 14.2

Следующее постусловие обеспечит успешную верификацию метода:

```
ensures a[i] == old(a[i]) + 1
```

Ответ на упражнение 14.3

Следующее присваивание реализует спецификацию `OldVsParameters`:

```
y := 25 - a[i];
```

Ответ на упражнение 14.8

Подсказка: метод `DoubleArray` похож на `CopyArray`, за исключением двух отличий. Одно из них: вставка `2 *` перед членом `src` в спецификации метода, а также в спецификации и в теле цикла. Другое отличие: инвариант, который гласит о неизменности элементов `src`, должен определять только индексы в диапазоне `n <= i < src.Length`.

Ответ на упражнение 14.13

```
method CopyMatrix<T>(src: array2<T>, dst: array2<T>)
  requires src.Length0 == dst.Length0
  requires src.Length1 == dst.Length1
  modifies dst
  ensures forall i, j :: 0 <= i < dst.Length0 && 0 <= j < dst.Length1 ==>
    dst[i, j] == old(src[i, j])
{
  forall i, j | 0 <= i < dst.Length0 && 0 <= j < dst.Length1 {
    dst[i, j] := src[i, j];
  }
}

method TestHarness() {
  var m := new int[2, 1];
  m[0, 0], m[1, 0] := 5, 7;
  CopyMatrix(m, m);
  // следующее утверждение не выполнится, если забыть
  // добавить 'old' в спецификацию CopyMatrix
  assert m[1, 0] == 7;
  var n := new int[2, 1];
  CopyMatrix(m, n);
  assert m[1, 0] == n[1, 0] == 7;
}
```

Ответ на упражнение 15.5

Вставьте следующий код перед присваиванием переменной `pivot`:

```
var p0, p1, p2 := a[lo], a[(lo + hi) / 2], a[hi - 1];
if {
  case p0 <= p1 <= p2 || p2 <= p1 <= p0 =>
    a[(lo + hi) / 2], a[lo] := a[lo], a[(lo + hi) / 2];
  case p1 <= p2 <= p0 || p0 <= p2 <= p1 =>
    a[hi - 1], a[lo] := a[lo], a[hi - 1];
  case p2 <= p0 <= p1 || p1 <= p0 <= p2 =>
    // ничего не делать
}
```

Ответ на упражнение 15.7

Вот три подсказки. Первая – хорошая спецификация:

```
requires IsSorted(a, 0, a.Length) && IsSorted(b, 0, b.Length)
ensures fresh(c) && c.Length == a.Length + b.Length
ensures IsSorted(c, 0, c.Length)
ensures multiset(c[..]) == multiset(a[..]) + multiset(b[..])
```

Вам нужно определить предикат `IsSorted`. Постусловие **`fresh`**`(c)` сообщает вызывающей стороне, что возвращаемый массив не зависит от любого другого массива, который может иметься у вызывающей стороны. Условие `c.Length == a.Length + b.Length` следует из последнего постусловия, но доказательство требует индукции, поэтому вызывающей стороне, вероятно, понравится условие, указанное непосредственно в постусловии.

Вторая подсказка: для реализации используйте метод конструирования циклов 12.0 (*замена константы переменной*), чтобы заменить неявные константы верхней границы `a.Length`, `b.Length` и `c.Length` в `a[..]`, `b[..]` и `c[..]` переменными, скажем `i`, `j` и `k`. Дополнительно в последней строке реализации помогите верификатору подсказкой:

```
assert a[..i] == a[..] && b[..j] == b[..] && c[..k] == c[..];
```

Третья подсказка: добавьте в инвариант цикла условие, аналогичное предикату `SplitPoint`, который мы использовали для сортировки выбором и быстрой сортировки. Я предлагаю что-нибудь вроде:

```
predicate Below(a: seq<int>, b: seq<int>) {
  forall i, j :: 0 <= i < |a| && 0 <= j < |b| ==> a[i] <= b[j]
}
```

Удачи!

Ответ на упражнение 16.8

```
method RemoveGrinder() returns (grinder: Grinder)
  requires Valid()
  modifies Repr
  ensures Valid() && fresh(Repr - old(Repr))
  ensures grinder.Valid() && grinder in old(Repr) - Repr
{
  grinder := g;
  g := new Grinder();
  Repr := Repr + {g} - {grinder};
}
```

Ответ на упражнение 17.0

Вот хороший экспортируемый набор для модуля:

```
export
  reveals LazyArray
  provides LazyArray.Elements, LazyArray.Repr
  provides LazyArray.Valid
  provides LazyArray.Get, LazyArray.Update
```

Ответ на упражнение 17.3

Тело конструктора:

```
M, Repr, root := map[], {this}, null;
```

Ответ на упражнение 17.7

Тело `BinarySearchTree.Add`:

```
if root == null {
  root := new Node(key, value);
} else {
  root.Add(key, value);
}
M := root.M;
Repr := Repr + root.Repr;
```

Ответ на упражнение 17.10

Без информации о выполнении свойств узла ничего не известно об отношении между полями **this** (что необходимо, например, для доказательства постусловия `k in M.Keys`) или об отношении между полями **this** и полями `left` (что необходимо, например, для доказательства завершимости рекурсивного вызова).

Рекомендуемая литература

[0] Jean-Raymond Abrial. «Modeling in Event-B: System and Software Engineering». Cambridge University Press, 2010.

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. «The Design and Analysis of Computer Algorithms». Addison-Wesley, 1974[1].

[2] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. «Deductive Software Verification–The KeY Book–From Theory to Practice», volume 10001 of Lecture Notes in Computer Science. Springer, 2016.

[3] Krzysztof R. Apt. «Ten years of Hoare's logic: A survey – Part I». ACM Transactions on Programming Languages and Systems, 3(4): 431–483, October 1981.

[4] Linard Arquint, João C. Pereira, Peter Müller, and FelixWolf. Gobra. *https://www.pm.inf.ethz.ch/research/gobra.html*.

[5] Vytautas Astrauskas, Aurel Bílý, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. Prusti. *https://www.pm.inf.ethz.ch/research/prusti.html*.

[6] Ralph-Johan Back. «Proving total correctness of nondeterministic programs in infinitary logic». Acta Informatica, 15: 233–249, 1981.

[7] Ralph-Johan Back. «A method for refining atomicity in parallel algorithms». In Eddy Odijk, Martin Rem, and Jean-Claude Syre, editors, *PARLE '89: Parallel Architectures and Languages Europe, Volume II: Parallel Languages*, volume 366 of *Lecture Notes in Computer Science*, pages 199–216. Springer, 1989.

[8] Ralph-Johan Back and Joakim von Wright. «Duality in specification languages: A lattice-theoretical approach». Acta Informatica, 27(7): 583–625, 1990.

[9] Ralph-Johan Back and Joakim von Wright. «Refinement Calculus: A Systematic Introduction». Graduate Texts in Computer Science. Springer-Verlag, 1998.

[10] Ralph-Johan R. Back. «On the Correctness of Refinement Steps in Program Development». PhD thesis, University of Helsinki, 1978. Report A-1978-4.

[11] Roland Backhouse, editor. «Special issue on The Calculational Method», volume 53(3). Information Processing Letters, February 1995.

[12] Roland C. Backhouse. «Program Construction and Verification». Series in Computer Science. Prentice-Hall International, 1986.

[13] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. «Specification and verification: The Spec# experience». *Communications of the ACM*, 54(6): 81–91, June 2011.

[14] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and NickyWilliams. «The dogged pursuit of bug-free C programs: the Frama-C software analysis platform». *Communications of the ACM*, 64(8): 56–68, 2021.

[15] Gertrud Bauer and Markus Wenzel. «Calculational reasoning revisited (an Isabelle/Isar experience)». In Richard J. Boulton and Paul B. Jackson, editors, *Theo-*

[1] *Ахо А. В.; Хопкрофт Д. Э., Ульман Д. Д.* Разработка и анализ компьютерных алгоритмов. Вильямс, 2021. ISBN: 978-5-907203-27-3. – *Прим. перев.*

rem Proving in Higher Order Logics, 14th International Conference, TPHOLs 2001, volume 2152 of *Lecture Notes in Computer Science*, page 7590. Springer, 2001.

[16] Yves Bertot and Pierre Castéran. «Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions». Texts in Theoretical Computer Science. Springer, 2004.

[17] Richard Bird. «Pearls of Functional Algorithm Design». Cambridge University Press, 2010[1].

[18] Richard S. Bird. «Algebraic identities for program calculation». *The Computer Journal*, 32(2): 122–126, 1989.

[19] Garrett Birkhoff. «Lattice Theory». Colloquium Publications, Volume 25. American Mathematical Society, 1967.

[20] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. «Why3 – Where programs meet provers». *https://why3.lri.fr*.

[21] François Bourdoncle. «Abstract debugging of higher-order imperative languages». In Robert Cartwright, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI)*, pages 46–55. ACM, 1993.

[22] Ana Bove, Peter Dybjer, and Ulf Norell. «A brief overview of Agda – A functional language with dependent types». In Stefan Berghofer, Tobias Nipkow, ChristianUrban, and MakariusWenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78. Springer, 2009.

[23] Robert S. Boyer and J Strother Moore. «A Computational Logic». ACM Monograph Series. Academic Press, Inc., 1979.

[24] W. Braun and M. Rem. «A logarithmic implementation of flexible arrays». Memorandum MR83/4, University of Technology Eindhoven, 1983.

[25] Stephen Brookes and PeterW. O'Hearn. «Concurrent separation logic». *ACM SIGLOG News*, 3(3): 47–65, 2016.

[26] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. «An overview of JML tools and applications». *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3): 212–232, 2005.

[27] Adam Chlipala. «Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant». MIT Press, 2013.

[28] Robin Cockett and Tom Fukushima. «About Charity». Yellow Series Report 92/480/18, Department of Computer Science, The University of Calgary, June 1992.

[29] Edward Cohen. «Programming in the 1990s: An Introduction to the Calculation of Programs». Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[30] Ernie Cohen, Michał Moskal,Wolfram Schulte, and Stephan Tobies. «Local verification of global invariants in concurrent programs». In Tayssir Touili, Byron Cook, and Paul B. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010*, volume 6174 of *Lecture Notes in Computer Science*, pages 480–494. Springer, 2010.

[1] *Бёрд Р.* Жемчужины проектирования алгоритмов: функциональный подход. ДМК Пресс, 2023. ISBN: 978-5-89818-555-8. – *Прим. перев.*

[31] David R. Cok, Gary T. Leavens, and Mattias Ulbrich. JML Reference Manual (2nd edition), 2021. *https://www.openjml.org/documentation/JML_Reference_Manual.pdf*.
[32] David R. Cok and Serdar Tasiran. «Practical methods for reasoning about Java 8's functional programming features». In Ruzica Piskac and Philipp Rümmer, editors, *Verified Software. Theories, Tools, and Experiments – 10th International Conference, VSTTE 2018*, volume 11294 of *Lecture Notes in Computer Science*, pages 267–278. Springer, 2018.
[33] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. «Termination proofs for systems code». In Michael I. Schwartzbach and Thomas Ball, editors, *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 415–426. ACM, 2006.
[34] Thierry Coquand and Gérard P. Huet. «The calculus of constructions. Information and Computation», 76(2/3): 95–120, 1988.
[35] Dafny – язык программирования с поддержкой верификации программ. *https://dafny.org*.
[36] Документация Dafny, 2022. *https://dafny.org/dafny*.
[37] Stijn de Gouw, Frank S. de Boer, and Jurriaan Rot. «Proof pearl: The KeY to correct and stable sorting». *Journal of Automated Reasoning*, 53(2): 129–139, 2014.
[38] Stijn de Gouw, Jurriaan Rot, Frank S. de Boer, Richard Bubel, and Reiner Hähnle. «OpenJDK's java.utils.Collection.sort() is broken: The good, the bad and the worst case». In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification–27th International Conference, CAV 2015*, volume 9206 of *Lecture Notes in Computer Science*, pages 273–289. Springer, 2015.
[39] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. «Extended static checking». Research Report 159, Compaq Systems Research Center, 1998.
[40] EdsgerW. Dijkstra. «A Discipline of Programming». Prentice Hall, Englewood Cliffs, NJ, 1976[1].
[41] EdsgerW. Dijkstra andW. H. J. Feijen. «A Method of Programming». Addison-Wesley, 1988.
[42] Edsger W. Dijkstra and Carel S. Scholten. «Predicate Calculus and Program Semantics». Texts and Monographs in Computer Science. Springer-Verlag, 1990.
[43] Claire Dross, Pavlos Efstathopoulos, David Lesens, David Mentré, and Yannick Moy. «Rail, space, security: Three case studies for SPARK 2014». In *7th European Congress on Embedded Real Time Software and Systems* (ERTS2 2014), 2014.
[44] Eiffel Software. Eiffel. *https://www.eiffel.com*.
[45] Marco Eilers and Peter Müller. Nagini. *https://www.pm.inf.ethz.ch/research/nagini.html*.
[46] Manuel Fähndrich and K. Rustan M. Leino. «Declaring and checking non-null types in an object-oriented language». In Ron Crocker and Guy L. Steele Jr., editors, *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003*, pages 302–312. ACM, 2003.
[47] Manuel Fähndrich and Songtao Xia. «Establishing object invariants with delayed types». In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Con-*

[1] *Дейкстра Э. Дисциплина программирования. Матем.обеспечение ЭВМ*. RUGRAM, 2013. ISBN: 978-5-458-33419-8. – *Прим. перев.*

ference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, pages 337–350. ACM, 2007.
[48] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. «The spirit of ghost code. Formal Methods in System Design», 48(3): 152–174, 2016.
[49] Jean-Christophe Filliâtre and Nicolas Magaud. «Certification of sorting algorithms in the Coq system». In *Theorem Proving in Higher Order Logics: Emerging Trends*, 1999.
[50] Robert W. Floyd. «Assigning meanings to programs». *Proceedings Symposium on Applied Mathematics,* 19: 19–31, 1967.
[51] M. Foley and C. A. R. Hoare. «Proof of a recursive program: Quicksort». *The Computer Journal*, 14(4): 391–395, 1971.
[52] Four ferries. *https://fourferries.com*, 2022.
[53] F*. *https://www.fstar-lang.org*.
[54] James Gosling, Bill Joy, and Guy Steele. «The Java™Language Specification». Addison-Wesley, 1996.
[55] David Gries. «The Science of Programming». Texts and Monographs inComputer Science. Springer-Verlag, 1981.
[56] David Gries and Fred B. Schneider. «A Logical Approach to Discrete Math». Texts and Monographs in *Computer Science*. Springer-Verlag, 1994.
[57] John V. Guttag and James J. Horning, editors. «Larch: Languages and Tools for Formal Specification». Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M.Wing.
[58] David Harel. «Proving the correctness of regular deterministic programs: A unifying survey using dynamic logic». *Theoretical Computer Science*, 12: 61–81, 1980.
[59] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and MatthewJ. Parkinson. «Behavioral interface specification languages». *ACM Computing Surveys*, 44(3): 16:1–16:58, 2012.
[60] Eric C. R. Hehner. «do considered od: A contribution to the programming calculus». *Acta Informatica*, 11: 287–304, 1979.
[61] Eric C. R. Hehner. «A Practical Theory of Programming». Texts and Monographs in Computer Science. Springer-Verlag, 1993.
[62] Eric C. R. Hehner. «Specified blocks». In Bertrand Meyer and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005*, volume 4171 of *Lecture Notes in Computer Science*, pages 384–391. Springer, 2005.
[63] C. A. R. Hoare. «An axiomatic basis for computer programming». *Communications of the ACM*, 12(10): 576–580,583, October 1969.
[64] Bart Jacobs and Frank Piessens. «The VeriFast program verifier». Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, August 2008.
[65] Bart Jacobs, Frank Piessens, K. Rustan M. Leino, and Wolfram Schulte. «Safe concurrency for aggregate objects with invariants». In Bernhard K. Aichernig and Bernhard Beckert, editors, *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005)*, pages 137–147. IEEE Computer Society, 2005.

[66] The Java Modeling Language (JML). *http://www.jmlspecs.org*.
[67] Cliff B. Jones. «Systematic Software Development using VDM». Series in Computer Science. Prentice Hall International, second edition, 1990.
[68] Anne Kaldewaij. «Programming: The Derivation of Algorithms». Series in Computer Science. Prentice Hall International, 1990.
[69] Ioannis T. Kassios. «Dynamic frames: Support for framing, dependencies and sharing without restrictions». In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283. Springer, August 2006.
[70] Shmuel Katz and Zohar Manna. «A closer look at termination». *Acta Informatica*, 5: 333–352, 1975.
[71] Matt Kaufmann and J Strother Moore. ACL2 version 8.4. *https://www.cs.utexas.edu/users/moore/acl2*.
[72] Leslie Lamport. «How to write a proof». *The American Mathematical Monthly*, 102(7): 600–608, 1995.
[73] Leslie Lamport. Домашняя страница TLA+. *https://lamport.azurewebsites.net/tla/tla.html*, March 2022.
[74] Butler W. Lampson, James J. Horning, Ralph L. London, James G. Mitchell, and Gerald L. Popek. «Report on the programming language Euclid». Technical Report CSL-81-12, Xerox PARC, October 1981.
[75] Lean theorem prover. *https://leanprover.github.io*.
[76] K. Rustan M. Leino. «Dafny: An automatic program verifier for functional correctness». In Edmund M. Clarke and Andrei Voronkov, editors, *LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, April 2010.
[77] K. Rustan M. Leino. «Automating induction with an SMT solver». In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation – 13th International Conference, VMCAI 2012*, volume 7148 of *Lecture Notes in Computer Science*, pages 315–331. Springer, 2012.
[78] K. Rustan M. Leino. «Accessible software verification with Dafny». *IEEE Software*, 34(6): 94–97, 2017.
[79] K. Rustan M. Leino and Paqui Lucio. «An assertional proof of the stability and correctness of natural mergesort». *ACM Transactions on Computational Logic*, 17(1):6:1–6:22, 2015.
[80] K. Rustan M. Leino and Daniel Matichuk. «Modular verification scopes via export sets and translucent exports». In Peter Müller and Ina Schaefer, editors, *Principled Software Development – Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of his 60th Birthday*, pages 185–202. Springer, 2018.
[81] K. Rustan M. Leino and Rosemary Monahan. «Reasoning about comprehensions with first-order SMT solvers». In Sung Y. Shin and Sascha Ossowski, editors, *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC)*, pages 615–622. ACM, March 2009.
[82] K. Rustan M. Leino and Michał Moskal. «Usable auto-active verification». In Tom Ball, Lenore Zuck, and N. Shankar, editors, *Workshop on Usable Verification*, November 2010. *https://fm.csl.sri.com/UV10/index.shtml*.
[83] K. Rustan M. Leino and Peter Müller. «Using the Spec# language, methodology, and tools to write bug-free programs». In Peter Müller, editor, *Advanced Lec-*

tures on Software Engineering, LASER Summer School 2007/2008, volume 6029 of *Lecture Notes in Computer Science*, pages 91–139. Springer, 2008.

[84] K. Rustan M. Leino and Peter Müller. «A basis for verifying multi-threaded programs». In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009*, volume 5502 of *Lecture Notes in Computer Science*, pages 378–393. Springer, 2009.

[85] K. Rustan M. Leino and Nadia Polikarpova. «Verified calculations». In Ernie Cohen and Andrey Rybalchenko, editors, *Verified Software: Theories, Tools, Experiments – 5th International Conference, VSTTE 2013*, Revised Selected Papers, volume 8164 of *Lecture Notes in Computer Science*, pages 170–190. Springer, 2014.

[86] LiquidHaskell. *https://ucsd-progsys.github.io/liquidhaskell-blog*.

[87] Barbara Liskov and John Guttag. «Abstraction and Specification in Program Development». MIT Electrical Engineering and Computer Science Series. MIT Press, 1986.

[88] Ralph L. London. «Certification of algorithm 245 [M1]: Treesort 3: Proof of algorithms–A new kind of certification». *Communications of the ACM*, 13(6): 371–373, 1970.

[89] Bertrand Meyer. «Object-oriented Software Construction». Series in Computer Science. Prentice Hall International, 1988[1].

[90] Robin Milner, Mads Tofte, and Robert Harper. «The Definition of Standard ML». MIT Press, 1990.

[91] J Strother Moore and Claus-PeterWirth. «Automation of mathematical induction as part of the history of logic». Technical Report SR201302, SEKI, 2014.

[92] Carroll Morgan. «The specification statement». *ACM Transactions on Programming Languages and Systems*, 10(3): 403–419, July 1988.

[93] Carroll Morgan. «Programming from Specifications». Series in Computer Science. Prentice Hall International, second edition, 1994.

[94] Carroll Morgan. «(in-)formal methods: The lost art–A users' manual». In Zhiming Liu and Zili Zhang, editors, *Engineering Trustworthy Software Systems – First International School, SETSS 2014. Tutorial Lectures*, volume 9506 of *Lecture Notes in Computer Science*, pages 1–79. Springer, 2016.

[95] Carroll C. Morgan. «Data refinement using miracles». *Information Processing Letters*, 26(5): 243–246, January 1988.

[96] Joseph M. Morris. «A theoretical basis for stepwise refinement and the programming calculus». *Science of Computer Programming*, 9(3): 287–306, December 1987.

[97] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. «Modular invariants for layered object structures». *Science of Computer Programming*, 62(3): 253–286, 2006.

[98] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. «Viper: A verification infrastructure for permission-based reasoning». In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation – 17th International Conference, VMCAI 2016*, volume 9583 of *Lecture Notes in Computer Science*, pages 41–62. Springer, 2016.

[1] *Мейер Б.* Объектно-ориентированное конструирование программных систем. Русская редакция, 2004. ISBN 5-7502-0255-0, 0-13-62155-4. – *Прим. перев.*

[99] Greg Nelson. «A generalization of Dijkstra's calculus». *ACM Transactions on Programming Languages and Systems*, 11(4): 517–561, October 1989.

[100] Greg Nelson, editor. «Systems Programming with Modula-3». Series in Innovative Technology. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[101] Greg Nelson. «LIM and Nanoweb». Technical Report HPL-2005-41, HP Labs, February 2005. *https://www.hpl.hp.com/techreports/2005/HPL-2005-41.pdf*.

[102] Tobias Nipkow. «Structured proofs in Isar/HOL». In Herman Geuvers and FreekWiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002*, volume 2646 of *Lecture Notes in Computer Science*, pages 259–278. Springer, 2002.

[103] Peter O'Hearn. «Separation logic». *Communications of the ACM*, 62(2): 86–95, 2019.

[104] Chris Okasaki. «Purely Functional Data Structures». Cambridge University Press, 1998[1].

[105] «OpenJML– Does your program do what it is supposed to do?» *https://www.openjml.org*.

[106] Susan Owicki and David Gries. «An axiomatic proof technique for parallel programs I». *Acta Informatica*, 6: 319–340, 1976.

[107] Sam Owre, John M. Rushby, N. Shankar, and DavidW. J. Stringer-Calvert. PVS system. *https://pvs.csl.sri.com*, January 2021.

[108] D. L. Parnas. «On the criteria to be used in decomposing systems into modules». *Communications of the ACM*, 15(12): 1053–1058, December 1972.

[109] David J. Pearce, Lindsay Groves, and Mark Utting. «Whiley – Aprogramming language with extended static checking». *http://whiley.org*.

[110] Simon Peyton Jones, John Hughes, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and PhilipWadler. «Haskell 98 report». *https://www.haskell.org/onlinereport*, February 1999.

[111] Frank Pfenning and Christine Paulin-Mohring. «Inductively defined types in the calculus of constructions». In Michael G. Main, Austin Melton, Michael W. Mislove, and David A. Schmidt, editors, *Mathematical Foundations of Programming Semantics, 5th International Conference*, volume 442 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 1989.

[112] Nadia Polikarpova, Julian Tschannen, Carlo A. Furia, and Bertrand Meyer. «Flexible invariants through semantic collaboration». In Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods – 19th International Symposium*, volume 8442 of *Lecture Notes in Computer Science*, pages 514–530. Springer, 2014.

[113] John C. Reynolds. «The Craft of Programming». Series in Computer Science. Prentice-Hall International, 1981.

[114] John C. Reynolds. «Separation logic: Alogic for shared mutable data structures». In *17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74. IEEE Computer Society, 2002.

[1] *Окасаки К.* Чисто функциональные структуры данных. ДМК Пресс, 2016. ISBN: 978-5-97060-233-1, 978-0-5216635-0-2. – *Прим. перев.*

[115] Rust – язык программирования, позволяющий создавать надежное и эффективное программное обеспечение. *https://www.rustlang.org*.

[116] Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. «Automatic verifier for Java-like programs based on dynamic frames». In José Luiz Fiadeiro and Paola Inverardi, editors, *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008*, volume 4961 of *Lecture Notes in Computer Science*, pages 261–275. Springer, March–April 2008.

[117] SPARK 2014 – расширение возможностей безопасного и надежного программирования. *https://www.adacore.com/about-spark*, 2002.

[118] Stainless – формальная верификация программ на Scala. *https://stainless.epfl.ch*.

[119] Christian Sternagel. «Proof pearl – A mechanized proof of GHC's mergesort». *Journal of Automated Reasoning*, 51(4): 357–370, 2013.

[120] Aaron Stump. «Verified Functional Programming in Agda». ACM and Morgan & Claypool, New York, NY, USA, 2016.

[121] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Nadia Polikarpova. «AutoProof: Auto-active functional verification of object-oriented programs». In *Tools and Algorithms for the Construction and Analysis of Systems – 21st International Conference, TACAS 2015*, volume 9035 of *Lecture Notes in Computer Science*, pages 566–580. Springer, 2015.

[122] Jan L. A. van de Snepscheut. «What computing is all about». Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[123] Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn, and Graham Hutton. «Theorem proving for all: Equational reasoning in Liquid Haskell (functional pearl)». In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018*, pages 132–144. ACM, 2018.

[124] Philip Wadler. «Linear types can change the world!» In Manfred Broy and Cliff B. Jones, editors, *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods*, page 561. North-Holland, April 1990.

[125] Markus Wenzel. «Isabelle/Isar – A versatile environment for human-readable formal proof documents». PhD thesis, Institut für Informatik, Technische Universität München, 2002.

[126] ThomasWies, Damien Zufferey, Siddharth Krishna, and Ruzica Piskac. GRASShopper. *https://cs.nyu.edu/~wies/software/grasshopper*.

Предметный указатель

Л

М

Н

О

П

P

C

T

У

Ф

Х

Ц

Ч

Ш

Э

Я

К. Рустан М. Лейно

Доказательство корректности программ

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод с английского *Киселев А. Н.*
Научный редактор *Кондратьев Д. А.*
Корректор *Абросимова Л. А.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Печать цифровая. Усл. печ. л. 43.06.
Тираж 100 экз.

Веб-сайт издательства: **www.dmkpress.com**

Данная книга учит формально рассуждать о компьютерных программах, используя последовательный подход и язык программирования Dafny, поддерживающий верификацию.
Показано, как писать спецификации для программ, как удовлетворить требования этих спецификаций и как писать доказательства корректности программ относительно спецификаций. Автор сначала представляет теоретические предпосылки, лежащие в основе рассуждений о программном коде, а затем постепенно переходит к реальным примерам, использующим объекты, структуры данных и нетривиальную рекурсию.

Особенности книги:

- написана простым и понятным языком;
- постепенно вводит все более сложные понятия;
- наглядно демонстрирует, как писать доказательства, а также как задавать и верифицировать функциональные и императивные программы;
- приводит примеры программного кода на реальном языке программирования, а не псевдокод;
- содержит забавные иллюстрации и практические упражнения.

Издание будет полезно студентам вузов, преподавателям, исследователям в области формальной верификации, а также сотрудникам компаний, применяющих дедуктивную верификацию на практике.

К. Рустан М. Лейно

Старший научный сотрудник в отделе Automated Reasoning Group в Amazon Web Services, член ACM, член IFIP и лауреат премии CAV.

The MIT Press

ДМК ПРЕСС ИЗДАТЕЛЬСТВО

www.дмк.рф

Страница книги на dmkpress.com

ISBN 978-5-93700-199-3

9 785937 001993 >