Лекция 19

Вычислимые функции

Как устроены компьютерные программы в общем случае? Они получают некоторый набор данных на вход, совершают некоторые действия и возвращают некоторый набор данных на выход – последнее происходит, если программа успешно завершила работу. Если мы дополнительно потребуем, чтобы программа при исполнении не использовала сторонние данные, то есть работала только со входными данными и рабочей памятью, которая изначально пуста, то мы получим, что каждая программа *реализует* некоторую функцию.

Это ограничение весьма существенно – нам требуется, чтобы каждая программа соответствовала некоторому алгоритму. В то же время программа, обращающаяся к сторонней области памяти, может, например, вызвать генератор случайных чисел, что может сделать её поведение непоследовательным при обработке одних и тех же данных в разное время.

Таким образом, каждая программа *реализует* некоторую функцию. Поскольку наборы входных и выходных данных у разных программ могут сильно различаться, мы полагаем, что программы получают на вход и возвращают на выходе некоторый набор битов – строки конечной длины над алфавитом $\{0, 1\}$. Множество всех таких строк мы обозначаем $\{0, 1\}^*$; также это множество содержит и строку нулевой длины – пустую строку λ. Итак, каждая программа реализует частично определённую функцию $\{0, 1\}^* \to \{0, 1\}^*$; здесь мы отступаем от смысла ранее введённого обозначения $f : A \to B$, означающего, что функция f всюду определена на множестве A.

Вычислимые функции являются математической абстракцией функций, реализуемых программами на используемых ныне языках программирования. Их формальное определение требует особой аккуратности – просто зафиксировать какой-то конкретный язык программирования и объявить, что функции, реализуемые программами на нём, образуют класс вычислимых функций, увы, не годится. Например, выше нам уже пришлось делать оговорки, дабы запретить использование генератора случайных чисел. Другой проблемой является практический взгляд на программы: в реальной жизни мы запускаем программы на разных устройствах с разными вычислительными возможностями. Получается, что классы функций, реализуемых некоторым реальным языком программирования, могут различаться в зависимости

от вычислительных возможностей компьютера, на котором запускаются программы.

Но если даже мы возьмём самый мощный компьютер на планете, или даже мысленно объединим все вычислительные устройства земли в один кластер, то мы получим, что программа может использовать только ограниченное число битов памяти. Значит, реальные программы в реальной жизни реализуют только функции из конечного множества в конечное, что с точки зрения математики не очень интересно – такие функции можно задать конечными таблицами. Оценка времени работы реальных программ с точки зрения скорости роста тоже слабо осмыслена: не трудно показать, что все они работают не более чем некоторое конечное количество времени (если, конечно, останавливаются).

Поэтому для определения вычислимых функций мы идеализируем языки программирования (а если быть точнее, то модель вычислений, исполняющую программы) и разрешаем им использовать память любого размера, но конечную на каждом шаге исполнения, а также работать сколь угодно долго. Увы, это далеко не единственные ограничения, которые мы используем. Все их перечислить довольно трудно, и мы не будем утомлять читателя попыткой перечислить их все, не дойдя до сути дела. Раскроем же, наконец, наши цели.

Оказывается, что класс функций, реализуемых программами на идеализированных языках программирования, не зависит от выбора языка программирования (если он достаточно сильный, что верно для абсолютного большинства современных языков). Поэтому мы называем этот класс – *классом вычислимых функций*, не упоминая конкретный язык. Читатель в качестве языка программирования может держать в голове язык Си или YFPL[1]. Объяснить совпадение классов с философской точки зрения нетрудно: для каждого языка программирования существует компилятор, который транслирует программы на этом языке в машинный код, который в свою очередь является программой на довольно примитивном языке программирования. Интерпретатор машинного кода, как правило, довольно несложно реализовать на YFPL. Значит, используя исходный компилятор языка YFPL-1 (уже записанный в виде машинного кода) и интерпретатор машинного кода, написанный на языке YFPL-2, можно получить интерпретатор языка YFPL-1, написанный на языке YFPL-2.

Почему бы нам, следуя этой философии, не определить вычислимые функции как функции, реализуемые некоторым аналогом машинного кода? На самом деле, это и есть формальное определение вычислимых функций, которое мы и будем использовать. В качестве аналога машинного кода мы используем машины Тьюринга, которые описываем в следующей главе. Почему же мы начинаем с абстрактных разговоров о вычислимых функциях вместо конкретики? Потому что мы хотим, чтобы читатель сосредоточился на некоторых общих нетривиальных особенностях всех достаточно сильных языков программирования, не опускаясь при этом на слишком формальный уровень изложения, вызванный использованием машин Тьюринга. Представьте, что было бы, если бы Вы свой первый курс по программированию на-

[1] Your Favorite Programming Language

чали с написания чего-то на ассемблере.

Какие же общие особенности есть у всех языков программирования? Оказывается, что некоторые задачи невозможно в принципе решить алгоритмически, то есть написать программу, которая решит эту задачу. К таким задачам относится, например, «проблема останова» – проверка остановки компьютерной программы по её входу. Более того, в итоге мы докажем теорему Успенского-Райса, которая гласит, что любое нетривиальное свойство вычислимой функции нельзя проверить алгоритмически. Неформально это означает, что нельзя по коду программы предсказать её поведение, используя для этого другую программу.

Из позитивных свойств есть совершенно неожиданные. Оказывается, что на любом достаточно сильном языке программирования существует программа, выводящая собственный текст. Кроме того, существует большое семейство программ, которые в процессе работы могут запускать себя (возможно на другом входе), а дальше по результатам исполнения продолжать вычисления. Эти контринтуитивные свойства вытекают напрямую из теоремы о неподвижной точке.

Формально под «достаточно сильным» языком программирования мы понимаем язык, на котором возможно написать интерпретатор машин Тьюринга. Такие языки программирования называют *Тьюринг-полными*.

Приведём ещё один довод в пользу выбранного нами подхода изложения, который по аналогии с наивной теорией множеств называют наивной теорией алгоритмов – в обоих случаях отсутсвует формальное определение предмета исследования. В современных курсах, посвящённых эффективным алгоритмам и вычислительной сложности, часто не опускаются на уровень изложения в терминах машин Тьюринга. Используемые там объекты и понятия, такие как «задача разрешения», «полиномиальная m-сводимость», либо совпадают с понятиями, возникающими при нашем подходе, либо получаются из них путём введения дополнительных ограничений, скажем, вместо вычислимых функций исследуют функции, вычислимые за полиномиальное время.

Завершим затянувшееся введение к этой главе обсуждением связи вычислимых функций и алгоритмов. Понятие алгоритма в последние годы настолько влилось в нашу жизнь, что его использование, как правило, не требует разъяснений. Под алгоритмом понимают чёткий конечный набор инструкций, исполнение которых (в идеале) не зависит от исполнителя. Пожалуй, наиболее формальным описанием алгоритма в наши дни служат компьютерные программы (с некоторыми оговорками). Поэтому вычислимые функции – это функции, которые реализуются алгоритмами, а формальным определением алгоритма является машина Тьюринга.

19.1 Правила игры и некоторые свойства вычислимых функций

Напомним, что читателю, предпочитающему отталкиваться от формальных определений, следует изучить сначала следующую главу и понимать под вычислимыми функциями функции, реализуемые машинами Тьюринга. Читателю, согласившемуся с выбранным нами порядком изложения следует держать в голове свой лю-

бимый язык программирования YFPL и опираться на свою интуицию, когда мы сообщаем ему о фактах, справедливость которых не доказываем, но объясняем их смысл в терминах языков программирования. Далее на этих фактах мы уже строим формальные доказательства о вычислимых функциях. Можно было бы назвать их аксиомами, однако под аксиомами понимают исчерпывающий набор свойств, в то время как мы приводим только их часть. Знакомству с этими базовыми свойствами и посвящён этот раздел.

Договоримся, что в YFPL есть несколько типов данных, среди которых есть двоичные строки и натуральные числа – полный список приведён в следующем разделе. Разрешено использовать циклы, арифметические и логические операции. Мы будем считать, что все арифметические функции вычислимы, равно как и функции для вычисления которых существует алгоритм (в смысле чёткого набора инструкций).

Теперь нам следует вернуться к тем ограничениям, которые необходимо наложить на идеализированные языки программирования, чтобы наши разговоры о реализуемых ими функциях имели нужный нам смысл. Как мы уже говорили выше, программа на каждом шаге вычисления может использовать только память конечного размера; тем не менее, количество используемой памяти может неограниченно расти. Такие ограничения приводят, в частности, к ограничению использования типов: например, мы не можем использовать произвольные иррациональные числа в процессе вычисления, поскольку для их описания нужно задать бесконечную последовательность цифр. Однако мы можем использовать рациональные числа, но для них представление в виде бесконечных дробей не годится – нам следует использовать представление рациональных чисел в виде пары из целого и натурального чисел (последнее не равно нулю). В общем случае мы можем использовать любой тип данных, объекты которого имеют конечное описание в виде набора битов. Такие объекты называют *конструктивными.*

19.1.1 Кодирование конструктивных объектов

Мы зафиксируем базовые типы, которые доступны YFPL для объявления переменных и массивов элементов этих типов. Приведём их список:

- двоичные строки $\{0,1\}^*$;
- натуральные числа $\mathbb{N}$;
- конечные наборы натуральных чисел $\mathbb{N}^k = \underbrace{\mathbb{N} \times \mathbb{N} \times \ldots \times \mathbb{N}}_{k}$;
- конечные наборов натуральных чисел произвольной длины

  $$\{(x_1, \ldots, x_k) \mid k \geqslant 1,\, x_i \in \mathbb{N}\} = \bigcup_{k=1}^{\infty} \mathbb{N}^k,$$

  обозначим это множество $\mathbb{N}^*$.

Выше мы ввели вычислимые функции как частично определённые функции $\{0,1\}^* \to \{0,1\}^*$, реализуемые программами на YFPL. Описав другие типы, мы можем теперь определить вычислимые функции $A \to B$, где A и B принимают значения $\{0,1\}^*, \mathbb{N}^*, \mathbb{N}^k, k \geqslant 1$. Насколько влияет выбор множеств A и B на получаемый класс вычислимых функций? Следующее утверждение объясняет, что он не является существенным.

Утверждение 19.1. *Пусть A и B – множества из класса $\{\{0,1\}^*, \mathbb{N}^*, \mathbb{N}^k, k \geqslant 1\}$. Тогда существует вычислимая (в обе стороны) биекция $h : A \to B$.*

Мы докажем это утверждение только для нескольких частных случаев, а остальные оставим читателю в качестве упражнений. Далее во всей главе, дабы каждый раз не оговариваться, говоря о вычислимой биекции h, мы будем подразумевать, что обратная биекция h^{-1} также вычислима.

Доказательство для $A = \mathbb{N}$ и $B = \{0,1\}^*$. Естественно было бы в качестве $h(n)$ взять функцию $\mathrm{bin}(n)$, ставящую в соответствие числу n его двоичную запись. Однако в этом случае останутся непокрытыми строки, начинающиеся с нуля – исправим это недоразумение. Определим $h(n)$ как двоичную запись числа $n+1$ без первой единицы слева (и ведущих нулей), а $h^{-1}(w) = n - 1$, где n – это число, двоичная запись которого получается из строки w приписыванием единицы слева.

Функция h действительно является биекцией в силу инъективности h и h^{-1}, которая вытекает из определения. Вычислимость функции h не вызывает сомнений – мы определили её через алгоритм вычисления. Мы будем использовать построенную биекцию $h(n)$ далее – обозначим эту функцию как $\underline{\mathrm{bin}}(n)$. □

Замечание 19.1. Среди множества всех строк $\{0,1\}^*$ есть и строка нулевой длины – пустая строка λ. Заметим, что

$$\underline{\mathrm{bin}}(0) = \lambda, \qquad \underline{\mathrm{bin}}(1) = 0, \qquad \underline{\mathrm{bin}}(2) = 1, \qquad \underline{\mathrm{bin}}(3) = 00.$$

Доказательство для $A = \mathbb{N} \times \mathbb{N}$ и $B = \mathbb{N}$. Мы используем уже хорошо известную читателю биекцию – нумерацию квадрата $\mathbb{N}\times\mathbb{N}$ по диагоналям. Эта биекция очевидно является вычислимой; приведём в качестве иллюстрации псевдокод реализующей её программы – листинг 19.1.

Докажем корректность приведённого алгоритма. Обращаем внимание читателя, что псевдокод или иное описание алгоритма не является доказательством его корректности! Докажем, что в приведённом цикле пара переменных (i, j) пробегает координаты всех клеток квадрата размера $m + 1 \times m + 1$ ровно по одному разу. Согласно строчке 3, координата i пробегает все значения от 0 до m по одному разу, а из строки 4 видно, что j при увеличении i пробегает все значения от 0 до i. Таким образом, строчки 3-4 реализуют обход квадрата размера $m + 1 \times m + 1$ по диагоналям.

Во время обхода, при выходе из каждой клетки, значение индекса n увеличивается на единицу. Алгоритм возвращает n при попадании в клетку (x, y), а значит

```
Вход  : (x, y) ∈ ℕ × ℕ
Выход: n ∈ ℕ
m ← max(x, y)
n ← 0
for i ← 0 to m do
    for j ← 0 to i do
        if (i, j) = (x, y) then
            return n
        end
        n ← n + 1
    end
end
```

Листинг 19.1: биекция $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$

реализует биекцию. Обозначим построенную вычислимую биекцию

$$c : \mathbb{N} \times \mathbb{N} \to \mathbb{N}.$$

Доказав вычислимость биекции c, доказать вычислимость обратной биекции $c^{-1}(n)$ уже не представляет труда – см. листинг 19.2.

```
Вход  : n ∈ ℕ
Выход: (x, y) ∈ ℕ × ℕ
for x ← 0 to n do
    for y ← 0 to i do
        if c(x, y) = n then
            return (x, y)
        end
    end
end
```

Листинг 19.2: биекция $\mathbb{N} \to \mathbb{N} \times \mathbb{N}$

Доказательство корректности данного алгоритма очевидно из построения. Клетка под номером n очевидно содержится в квадрате $n+1 \times n+1$; значит, обойдя его клетки, мы найдём её и вернём её координаты. □

Заметим, что мы привели довольно общий способ доказательства вычислимости обратной биекции, если доказано вычисление прямой. Другой вариант доказательства вычислимости биекции c состоит в том, чтобы доказать её выразимость простой алгебраической формулой.

Задача 19.1. Докажите, что

$$c(x, y) = \binom{x + y + 1}{2} + y.$$

Доказательство для $A = \{0,1\}^*$, $B = \mathbb{N}^*$. Мы оставляем читателю возможность обобщить решение для $A = \mathbb{N}$, $B = \mathbb{N}^k$, мы же укажем здесь явную кодировку. Функция $h(w)$ получает на вход слово w, которое имеет вид $0^{x_1}10^{x_2}1\ldots10^{x_k}$, где

$$0^{x_i} = \underbrace{00\ldots0}_{x_i},$$

и возвращает набор чисел $(x_1, x_2, \ldots, x_k)$. Функция h очевидно является биекцией и вычислима, поскольку задана алгоритмом. Вычислимость h^{-1} очевидна из построения h. □

Задача 19.2. Контрольный вопрос: $h(\lambda) = ?$

Мы описали несколько разрешённых для использования типов данных и установили биекцию между ними. Как понять, какие типы данных допустимо использовать? Перейдём к ответу на этот вопрос. Для этого определим формально понятие типа. Будем считать, что определение типа T равносильно описанию его объектов двоичными словами, то есть $T \subseteq \{0,1\}^*$. Например, в качестве описания квадратных матриц с элементами $0, 1$ можно использовать двоичные строки длины n^2 – каждой такой строке соответствует матрица размеров $n \times n$, первые n битов строки задают первую строку матрицы, вторые n – вторую и т.д.

Если множество T конечно, то мы будем называть тип константным. Интерес в большей степени представляют неконстантные типы. Любой константный тип мы будем считать допустимым. В качестве условия допустимости для неконстантного типа мы требуем, чтобы корректность описания объектов типа была алгоритмически проверяема. То есть тип T является допустимым, если существует вычислимая функция, которая, получив на вход двоичное слово w, возвращает 1, если $w \in T$, и 0, если $w \notin T$. Заметим, что существование этого условия равносильно вычислимости характеристической функции χ_T множества T.

Искушённый в программировании читатель (особенно освоивший C++) может быть справедливо недоволен нашим определением допустимости – как мы позже убедимся, ему удовлетворяют и объекты слабо похожие на типы. Однако для наших скромных целей данного определения более чем достаточно – можно было и вовсе ограничиться введением конечного набора типов. Предлагаем читателю убедиться, что для допустимых типов справедлив следующий факт.

Задача 19.3. Докажите, что для любого допустимого неконстантного типа существует вычислимая биекция в множество двоичных строк.

Мы приведём решение этой задачи в одном из следующих разделов. Также мы покажем, что не все типы данных являются допустимыми.

19.1.2 Множество вычислимых функций счётно

Наряду с остальными свойствами это свойство является постулатом; однако мы приводим для него обоснование в терминах языков программирования. Если функция

вычислима на YFPL, то её реализует некоторая программа, а любая программа состоит из конечного набора символов. Без ограничения общности можно считать, что любая программа на YFPL – это конечный набор битов, а множество всех конечных наборов битов счётно. Осталось объяснить, что поскольку существует бесконечная последовательность различных вычислимых функций, например, $f_i(x) = i$, то множество вычислимых функций бесконечно, а посему счётно.

Ниже, в разделе о главных нумерациях, мы придадим связи между программами и вычислимыми функциями формальный смысл.

Итак, опираясь на счётность множества вычислимых функций, мы можем доказать, что не всякая функция на множестве двоичных слов вычислима.

Утверждение 19.2. *Существует невычислимая функция* $f : \{0,1\}^* \to \{0,1\}^*$.

Доказательство. Действительно, поскольку множество $\{0,1\}^*$ счётно, множество функций $\{0,1\}^* \to \{0,1\}^*$ имеет мощность континуум. В то же время, множество вычислимых функций счётно, а значит среди всех функций есть невычислимые. □

Опираясь на существование невычислимых функций, мы докажем существование недопустимых типов. Как мы уже объяснили выше, для этого достаточно доказать следующее утверждение.

Утверждение 19.3. *Существуют недопустимые типы.*

Доказательство. Тип T является допустимым, если вычислима его характеристическая функция. Заметим, что определение множества равносильно заданию его характеристической функции. Типов $T \subseteq \{0,1\}^*$ континуально много, а значит среди континуума характеристических функций есть невычислимые. □

Перейдём к следующему свойству вычислимых функций.

19.1.3 Вычислимые функции замкнуты относительно композиции

Обоснование этого свойства очень простое, если есть программы P_f и P_g, реализующие функции f и g, то можно запустить P_g, сохранить результат вычисления в переменную и запустить программу P_f, на вход которой подан результат P_g. Описанная программа реализует функцию $f \circ g$.

Помимо замкнутости относительно композиции, мы пользуемся тем, что YFPL может использовать любую написанную на нём программу как подпрограмму. На самом деле мы уже пользовались этим свойством без объявления в доказательстве утверждения 19.1, когда строили вычислимую биекцию между $\mathbb{N}$ и $\mathbb{N} \times \mathbb{N}$, поэтому просим прощения у бдительного читателя.

Из замкнутости относительно композиции сразу следует, что неважно, какой класс вычислимых функций рассматривать: из $\{0,1\}^*$ в $\{0,1\}^*$, из $\mathbb{N}$ в $\mathbb{N}$ или же класс вычислимых функций между другими двумя допустимыми, зафиксированными в YFPL типами (все они неконстантные). Действительно, для любой вычислимой

функции $f : \{0,1\}^* \to \{0,1\}^*$ можно предъявить вычислимую функцию $g : \mathbb{N} \to \mathbb{N}$, такую что

$$g(n) = m \xleftarrow[\text{(перекодировка в число)}]{\underline{\text{bin}}^{-1}} y \xleftarrow[\text{(вычисление)}]{f} x \xleftarrow[\text{(перекодировка в строку)}]{\underline{\text{bin}}} n,$$

то есть $g = \underline{\text{bin}}^{-1} \circ f \circ \underline{\text{bin}}$; но тогда $f = \underline{\text{bin}} \circ g \circ \underline{\text{bin}}^{-1}$. Таким образом, между классами вычислимых функций $\mathbb{N} \to \mathbb{N}$ и $\{0,1\}^* \to \{0,1\}^*$ есть биекция. Кроме того, любую функцию из одного класса можно получить, подобрав подходящую функцию из другого класса и применив к ней композицию с биекцией между $\mathbb{N}$ и $\{0,1\}^*$ и обратной к ней в нужном порядке.

Для удобства дальнейшего изложения мы будем в основном использовать вычислимые функции $\mathbb{N} \to \mathbb{N}$. Также нам потребуются вычислимые функции двух и трёх аргументов: $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ и $\mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

Заметим также, что элементы типа $\mathbb{N}^*$ по сути являются массивами. Используя вычислимую биекцию $\underline{\text{bin}} : \mathbb{N} \to \{0,1\}^*$ получаем возможность использования в YFPL массивов строк. Предоставим читателю возможность решить следующую задачу, опередив тем самым ход изложения.

Задача 19.4. Пусть T – допустимый тип. Докажите, что массивы из элементов типа T также являются допустимым типом в YFPL.

19.1.4 Запуск по шагам и отладочная функция. Частичная определённость.

Нам потребуется возможность запускать программы на YFPL по шагам. В качестве примера запуска по шагам программы P приведём следующую программу, см. листинг 19.3. Будем считать, что P реализует функцию $f_P : \mathbb{N} \to \mathbb{N}$.

```
   Вход  : (x, t) ∈ ℕ × ℕ
   Выход: (stoped, result) ∈ {0,1} × ℕ
1  for  i ← 0 to t do
2  |  Сделать шаг исполнения программы P на входе x
3  end
4  if P закончила работу then
5  |  return  (1, f_P(x)) /* f_P(x) - результат работы P на входе x        */
6  else
7  |  return  (0, 0)
8  end
```

Листинг 19.3: запуск по шагам; отладчик.

Строка 2 использует объявленное нами свойство. Описанная псевдокодом программа получает на вход пару чисел (x, t) и возвращает пару чисел (stoped, result). Первое из них – либо 0, либо 1, и равно единице, только если программа P остановилась на входе x за t шагов; второе число в этом случае равно результату работы

P. Если же P не остановилась на x за t шагов, то программа возвращает пару $(0,0)$. Назовём эту программу *отладчиком* (программы P).

Прежде чем перенести определение отладчика на вычислимые функции, напомним, что вычислимые функции являются частично определёнными. Мы считаем, что функция не определена тогда и только тогда, когда соответствующая ей программа не останавливается. Тут можно было бы договориться и иначе, допустив случаи ошибки исполнения: например, такое возможно, если переменная типа x была равна нулю и программа собирается выполнить команду $x \leftarrow x - 1$. Однако мы для простоты изложения договоримся, что все ошибки исполнения переводят программу в вечный цикл. Мы введём специальные обозначения $f(x) = \downarrow$ и $f(x) = \uparrow$; первое означает, что f определена на x, а второе – что не определена.

Перенесём теперь определение отладчика на вычислимые функции.

Определение 19.1. Назовём вычислимую функцию $\mathcal{D}f : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \times \mathbb{N}$ *отладочной функцией* (вычислимой) функции $f : \mathbb{N} \to \mathbb{N}$ такую функцию, что

- $\mathcal{D}f(x,t) = (0,0)$ для каждого t, если $f(x) = \uparrow$;
- $\mathcal{D}f(x,t') = (1, f(x))$, для некоторого t', если $f(x) = \downarrow$, при этом $\forall t \geqslant t' : \mathcal{D}f(x,t) = (1, f(x))$.

Итак, очередное свойство, которое мы постулируем, – существование отладочной функции для каждой вычислимой функции. Заметим, что отладчик, описанный в листинге 19.3, реализует отладочную функцию, но формально отладочная функция к отладчику не привязана.

Приведём пример функции, для доказательства вычислимости которой существенно исполнение программы по шагам. Определим для вычислимых функций $f, g : \mathbb{N} \to \{0,1\}$ функцию $f(x) \vee g(x)$. Хорошо знакомое читателю определение дизъюнкции тут, увы, не годится – что делать, когда одна из функций не определена? Поэтому мы доопределим дизъюнкцию следующим образом:

$f(x)$	$g(x)$	$f(x) \vee g(x)$
1	$\uparrow$	1
$\uparrow$	1	1
0	$\uparrow$	$\uparrow$
$\uparrow$	0	$\uparrow$
$\uparrow$	$\uparrow$	$\uparrow$

Утверждение 19.4. *Пусть $f, g : \mathbb{N} \to \{0,1\}$ вычислимые функции. Тогда функция $f(x) \vee g(x)$, определённая в указанном выше смысле, вычислима.*

Доказательство. Приведём здесь доказательство через запуск программ по шагам. Обозначим через P_f и P_g, некоторые программы, реализующие функции f и g. Алгоритм вычисления $f(x) \vee g(x)$ приведён в листинге 19.4.

```
   Вход  : x ∈ {0,1}
   Выход: y ∈ {0,1}
 1 for  i ← 0 to ∞ do
 2 |   Сделать шаг исполнения программы P_f(x)
 3 |   Сделать шаг исполнения программы P_g(x)
 4 |   if  P_f или P_g вернули 1 then
 5 |   |   return 1
 6 |   else
 7 |   |   if P_f и P_g вернули 0 then
 8 |   |   |   return 0
 9 |   |   end
10 |   end
11 end
```

Листинг 19.4: реализация $f(x) \vee g(x)$.

Заметим, что алгоритм возвращает 1, если хотя бы одна из двух функций вернула единицу, и 0 только если обе функции вернули нули; в других случаях алгоритм не останавливается – в этих случаях $f(x) \vee g(x) = \uparrow$. Оставим читателю проверить соответствие указанного поведения таблице для функции $f(x) \vee g(x)$. □

Отладочная функция позволяет упростить некоторые рассуждения с параллельным запуском программ, что позволяет не приводить в простых доказательствах псевдокод и детальное описание распараллеливания. Для того чтобы лучше познакомить читателя с идей параллельного запуска программ, мы будем первое время описывать этот процесс псевдокодом и не использовать отладочную функцию. Как мы покажем в одном из следующих разделов, неосторожный подход к параллельному запуску может привести к описанию алгоритма, отличного от ожидаемого. Напутствуем читателя не попасться в нашу ловушку.

19.2 О строгости описания программ

При доказательстве вычислимости функции, мы, как правило, используем один из двух подходов. Первый – выводим вычислимость функции из описанных выше свойств; второй – описываем программу, которая её реализует, и доказываем её корректность.

При втором подходе обычно возникает вопрос о степени детализации описания программы и, как следствие, строгости доказательства. Порой мы описываем программу, предъявляя псевдокод, а порой описываем её неформально. Первый способ

безусловно более строгий, однако второй – ничуть не хуже, однако нужно чувствовать грань. В случае неформального описания программы, мы стараемся описывать её на таком уровне, чтобы её код можно было легко написать на YFPL.

19.3 Разрешимые и перечислимые множества

При исследование функций невольно возникает необходимость исследовать связанные с ними множества. Начинать, как правило, приходится с области определений и множества значений – поэтому мы начали нашу главу с обсуждения допустимых типов данных. С вычислимыми функциями тесно связаны разрешимые и перечислимые множества. Читатель скоро убедится, что допустимые типы – это в точности разрешимые множества. Однако связь на этом только начинается, и сами эти классы множеств играют важную роль в теории алгоритмов.

В предыдущем разделе мы показали, что несущественно, для каких множеств A, B рассматривать класс вычислимых функций $A \to B$, при условии, что A и B допустимые неконстантные типы. Начиная с этого раздела, мы сосредоточимся на вычислимых функциях $\mathbb{N} \to \mathbb{N}$ для удобства изложения. Соответсвенно, при исследовании множеств мы сосредоточимся на подмножествах натуральных чисел.

19.3.1 Задачи разрешения и разрешимые множества

Чаще всего компьютерные программы встречаются в нашей жизни, когда нужно решить какую-то задачу. Что же такое задача? Есть задачи, которые требуют сложить два числа, есть задачи, которые требуют найти максимум функции или путь между двумя вершинами в графе. В общем случае, задаче соответствует вычислимая функция, а её решением является написание программы, которая её реализует. Более того, обычно эта вычислимая функция всюду определена.

Занятно, что под алгоритмами в компьютерных науках чаще всего понимают не просто набор инструкций, а такие инструкции, выполнение которых на каждом входе займёт лишь конечное число действий – таким алгоритмам соответствуют всюду определённые вычислимые функции. Исключением является как раз теория алгоритмов, которую мы с вами и изучаем.

Ограничением среди всех задач являются задачи разрешения (decision problems), которые допускают только два ответа: «да» или «нет». То есть среди всех всюду определённых вычислимых функций мы ограничиваемся только функциями вида $f : \mathbb{N} \to \{0, 1\}$. Мы зафиксировали в качестве множества возможных входов множество натуральных чисел – выше мы уже обсуждали, что это ограничение не является существенным. К таким задачам относятся проверка числа на чётность, графа на связность, формулы на выполнимость. Эти задачи состоят в проверке элемента на входе на некоторое свойство: все элементы, удовлетворяющие этому свойству образуют множество. Итак, каждой всюду определённой функции $f : \mathbb{N} \to \{0, 1\}$ соответствует задача разрешения, а ей в свою очередь соответствует множество

$$A_f = \{x \mid f(x) = 1\}.$$

Множество A_f называют *разрешимым*, если функция f вычислима. Заметим, что функция f является характеристической функцией множества A_f.

Определение 19.2. Множество A является разрешимым, если его характеристическая функция вычислима.

Исследование задач разрешения само по себе является естественным вопросом, однако к ним сводятся задачи о вычислении произвольной всюду определённой функции. Мы опишем эту сводимость после изучения более простых вещей о разрешимых множествах, а пока предлагаем читателю поломать голову над тем как такое возможно.

Для доказательства разрешимости множества S достаточно предъявить алгоритм вычисления его характеристической функции. Такой алгоритм называют *алгоритмом разрешения* S.

Утверждение 19.5. *Любое конечное множество разрешимо.*

Доказательство. Алгоритм разрешения конечного множества S аналогичен программе, реализующей вычислимую функцию с конечной областью определения, см. утверждение 19.5. Алгоритм содержит таблицу элементов множества S, вход сравнивается по очереди со всеми элементами таблицы; в случае совпадения выдаётся 1, если ни одного совпадения не обнаружено, выдаётся 0. □

Задача 19.5. Докажите, что если A, B — разрешимые множества, то и множества $A \cup B$, $A \cap B$, $\bar{A}$ разрешимы.

На самом деле мы успели познакомиться с разрешимыми множествами ещё в начале главы.

Замечание 19.2. Тип T является допустимым тогда и только тогда, когда T – разрешимое множество.

19.3.2 Перечислимые множества

Множество S называют *перечислимым*, если существует алгоритм, последовательно печатающий его элементы. Этот алгоритм называют алгоритмом перечисления S. Повторяются ли в процессе перечисления S его элементы, не очень важно.

Задача 19.6. Докажите, что если существует алгоритм перечисления элементов некоторого множества, то существует также и алгоритм, который перечисляет элементы множества без повторений.

Пока из нашего определения неясно как перечислимые множества связаны с вычислимыми функциями: в случае бесконечного множества, алгоритм перечисления печатает бесконечно много элементов, а бесконечная последовательность не годится в качестве значения вычислимой функции. Зафиксируем некоторый алгоритм перечисления P множества S. Поставим в соответствие P вычислимую функцию

$f_P(t)$, которая возвращает t-ый элемент из вывода P, а если такового нет, то она не определена. Из определения f_P получаем

$$S = \{x \mid \exists t : x = f_P(t)\}.$$

В случае, когда алгоритм перечисления может быть произвольным, мы будем обозначать через f_S произвольную функцию среди всевозможных функций вида f_P (где P – алгоритма перечисления).

Однако связь между вычислимыми функциями и перечислимыми множествами куда более обширна. Мы поговорим о ней в конце раздела.

Мы изучаем перечислимые множества наряду с разрешимыми. Как же они соотносятся? Очевидно, что каждое разрешимое множество является перечислимым.

Утверждение 19.6. *Если множество S разрешимо, то оно перечислимо.*

Доказательство. Алгоритм перечисления множества S использует алгоритм разрешения множества S. Он перебирает все числа, начиная с 0; для каждого числа n вычисляет характеристическую функцию $\chi_S(n)$ и печатает число n, если полученное значение равно 1.

Корректность такого алгоритма очевидна из определений. □

Чтобы лучше почувствовать вычислимость, полезно строить явные алгоритмы перечисления даже для множеств, перечислимость которых доказана.

Задача 19.7. Приведите явный алгоритм перечисления множества $\mathbb{N} \times \mathbb{N}$.

Решение. Листинг 19.6 описывает естественный алгоритм перечисления. Доказательство его корректности оставим читателю в качестве упражнения. Приведём второе решение, использующее вычислимую биекцию c, определённую в задаче 19.1. Алгоритм перечисления перебирает n от 0 до ∞ и для каждого n выводит $c^{-1}(n)$. Все наборы чисел из $\mathbb{N} \times \mathbb{N}$ будут перечислены в силу биективности c. □

```
1 for  i ← 0 to ∞ do
2 |   for  j ← 0 to i do
3 |   |   print((i, j))
4 |   end
5 end
```

Листинг 19.5: алгоритм перечисления $\mathbb{N} \times \mathbb{N}$

Задача 19.8. Приведите явный алгоритм перечисления множества $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$.

Задача 19.9. Докажите, что множество всех конечных наборов чисел $\mathbb{N}^*$ перечислимо. Приведите явный алгоритм перечисления.

Заметим наперёд, что в отличие от счётных множеств, объединение перечислимых множеств не обязательно перечислимо. Использование такого «аргумента» для решения предыдущей задачи ошибочно. Соответствующую задачу мы предложим читателю в следующем разделе.

Обратим внимание, что мы фактически решили перечисленные выше задачи, когда говорили о типах.

Замечание 19.3. Из разрешимости бесконечного типа T следует перечислимость T, а опираясь на задачу 19.8, получаем, что элементы T перечислимы без повторений. Таким образом, из разрешимости T следует существование вычислимой биекции между $\mathbb{N}$ и T, а значит и между $\{0,1\}^*$ и T. Мы же строили в разделе 19.1.1 эти биекции явно.

В следующей задаче мы изучим типичный приём построения перечислимых множеств.

Задача 19.10. Докажите, что множество, состоящее из таких наборов $(v,w,x,y,z) \in \mathbb{N}^5$, что $v^5+w^5+x^5+y^5=z^5$ перечислимо. Предъявите явный алгоритм перечисления.

COUNTEREXAMPLE TO EULER'S CONJECTURE ON SUMS OF LIKE POWERS

BY L. J. LANDER AND T. R. PARKIN

Communicated by J. D. Swift, June 27, 1966

A direct search on the CDC 6600 yielded

$$27^5 + 84^5 + 110^5 + 133^5 = 144^5$$

as the smallest instance in which four fifth powers sum to a fifth power. This is a counterexample to a conjecture by Euler [1] that at least n nth powers are required to sum to an nth power, $n>2$.

REFERENCE

1. L. E. Dickson, *History of the theory of numbers*, Vol. 2, Chelsea, New York, 1952, p. 648.

Рис. 19.6: статья с контрпримером к гипотезе Эйлера.

Дабы взгляд читателя не упал преждевременно на следующий абзац с решением, заметим, что эта задача напоминает великую теорему Ферма. Множество $\{(x,y,z) \mid x^5+y^5=z^5\}$ очевидно перечислимо (загадка: почему?). Однако задача, которую мы перед вами поставили, относится к гипотезе Эйлера. Гипотеза утверждает, что если для некоторого набора ненулевых целых чисел $x_1, x_2, \ldots, x_k$ выполнено $\sum\limits_{i=1}^{k-1} x_i^n = x_k^n$, то $k \geqslant n$. Контрпример к этой гипотезе для $n=5$ был опубликован в одной из самых коротких математических статей в истории (см. рис. 19.7). Надеемся, к концу абзаца читатель решил поставленную задачу.

Решение. Из приведённой выше статьи мы знаем, что описанное множество непусто, а следовательно тривиальный (ничего не выводящий) алгоритм перечисления не годится. Воспользуемся перечислимостью $\mathbb{N}^5$. Наш алгоритм перечисления перебирает все наборы $(v, w, x, y, z) \in \mathbb{N}^5$ и выводит набор, только если $v^5 + w^5 + x^5 + y^5 = z^5$. $\square$

Заметим, что при построении алгоритма перечисления множества из предыдущей задачи вовсе не обязательно описывать пять вложенных циклов. Вполне можно использовать факт перечислимости известного множества, в нашем случае $\mathbb{N}^5$, и использовать алгоритм его перечисления как подпрограмму – что мы и сделали.

Замечание 19.4. Решив эту задачу, мы доказали, что если вычислимая функция $f : A \to \{0, 1\}$ всюду определена на множестве A и множество A перечислимо, то и множество $\{a \mid a \in A \wedge f(a) = 1\}$ также перечислимо.

Вернёмся к связи разрешимых и перечислимых множеств. Обратите внимание, что алгоритм перечисления разрешимого множества, описанный в утверждении 19.9, перечисляет его элементы в возрастающем порядке. Верно и обратное.

Задача 19.11. Докажите, что если существует алгоритм перечисления элементов множества S в возрастающем порядке, то это множество разрешимо.

Итак, разрешимые множества содержатся среди перечислимых. Эти два класса не совпадают. Однако доказать их несовпадение мощностными соображениями не получится: оба класса содержат счётное количество множеств в силу счётности общего числа программ. Пример перечислимого неразрешимого множества будет построен ниже диагональным методом.

Эти два класса имеют схожие свойства замкнутости относительно операций объединения и пересечения.

Задача 19.12. Докажите, что если A, B — перечислимые множества, то и множества $A \cup B$, $A \cap B$ перечислимы.

Обратим внимание, что, в отличие от задачи 19.7, в задаче 19.14 отсутсвует дополнение. Это ключевое различие между разрешимыми и перечислимыми множествами отражено в следующей теореме.

19.4 Решения задач

Решение задачи 19.1. Рассмотрим таблицу 19.8, построенную по функции $c(x, y)$. Заполнив несколько начальных клеток, замечаем, что клетки занумерованы подряд по диагоналям. Для того чтобы доказать это, отметим два следующих факта.

Во-первых, в этой таблице значения на диагоналях идут последовательно: сумма $x + y$ на диагонали постоянна, а значит $c(x, y) = c(x, 0) + y$. Во-вторых, число $c(x, 0)$ равно количеству клеток в треугольнике таблицы над диагональю, начинающейся с клетки $(x, 0)$. Действительно, в первой строке такого треугольника x клеток, во второй $x - 1$ и так далее до 1. Получаем, что количество клеток в треугольнике

равно сумме арифметической прогрессии, которая равна $\binom{x+1}{2} = c(x,0)$. Итак, мы получили, что в таблице 19.8 клетки занумерованы по диагоналям, а значит c – это действительно биекция.

$x \setminus y$	**0**	**1**	**2**	**3**	$\cdots$
0	0	2	5	$c(3,0)+3$	$\cdots$
1	1	4	$c(3,0)+2$	$c(1,3)$	$\cdots$
2	3	$c(3,0)+1$	$c(2,2)$	18	$\cdots$
3	$c(3,0)$	$c(3,1)$	17	24	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Таблица 19.7: вычислимая биекция $c : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

□

Решение задачи 19.7. Характеристические функции для множеств $A \cup B$, $A \cap B$ и $\bar{A}$ выражаются через характеристические функции для A и B следующим образом:

$$\chi_{A\cup B}(x) = \chi_A(x) \vee \chi_B(x), \qquad \chi_{A\cap B}(x) = \chi_A(x) \wedge \chi_B(x), \qquad \chi_A(x) = \neg\chi_A(x),$$

а потому все упомянутые множества разрешимы.

Опишем также решение через построение алгоритма разрешения на примере множества $A \cap B$. Алгоритм проверяет принадлежность элемента x на входе множествам A и B, запуская последовательно алгоритмы их разрешения. Получив результаты, алгоритм возвращает единицу, только если оба результата положительны. □

Решение задачи 19.8. Пусть P – программа, реализующая алгоритм перечисления. Построим программу Q, реализующую перечисление без повторений. Программа Q исполняет программу P, но когда P выводит некоторый элемент x, Q кладёт его в массив и проверяет, встречался ли x в массиве ранее. Если x добавлен в массив впервые, Q выводит x.

Очевидно, что Q выводит только элементы из множества, перечисляемого P, поскольку она выводит только элементы, которые выводит P. При этом Q, очевидно, выводит все элементы без повторений: если P выводит какой-то элемент x несколько раз, то Q выводит x только при первом его выводе программой P. □

Решение задачи??. Воспользуемся вычислимой биекцией c, определённой в задаче 19.1. Алгоритм перечисления перебирает n от 0 до ∞ для каждого n вычисляет $(m, z) = c^{-1}(n)$, после чего вычисляет $(x, y) = c^{-1}(m)$ и выводит набор (x, y, z).

Докажем, что все наборы из $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$ будут перечислены. Алгоритм перечислит все наборы $(m, z) \in \mathbb{N} \times \mathbb{N}$, а также все наборы $(x, y) \in \mathbb{N} \times \mathbb{N}$ в силу биективности c; это значит, что будут перечислены все наборы $(x, y, z) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N}$. □

Решение задачи 19.11. Явно следует из построенной в разделе 19.1.1 вычислимой биекции между $\mathbb{N}$ и $\mathbb{N}^*$ □

Решение задачи 19.13. Если S – конечное множество, то оно разрешимо по утверждению 19.8. Предположим теперь, что S бесконечно, а P – алгоритм его перечисления в возрастающем порядке. Тогда построим алгоритм Q разрешения S. На вход Q подаётся число x. Программа Q запускает программу P и ждёт, пока та не напечатает либо x, либо число, большее x (не напечатав x ранее). В первом случае Q выводит 1, поскольку x был перечислен P, а значит $x \in S$. Во втором случае Q выводит 0: раз было напечатано число, большее x, то x уже никогда не будет перечислено P, поскольку P перечисляет S в порядке возрастания, а значит $x \notin S$. □

Решение задачи 19.14. Пусть P_A и P_B – алгоритмы перечисления A и B. Построим алгоритм Q перечисления $A \cap B$ – алгоритм для $A \cup B$ аналогичен и даже несколько проще.

Программа Q запускает пошагово программы P_A и P_B и кладёт результаты их вывода в массивы a и b соответсвенно. Как только Q добавляет в a элемент x, она проходит массив b и проверят, содержит ли b элемент x. В случае положительного результата проверки Q выводит x. При добавлении нового элемента в массив b, Q действует симметрично: проверяет массив a и т.д.

Программа Q очевидно является алгоритмом перечисления множества $A \cap B$, поскольку Q выводит элемент x, если, и только если, его вывели и P_A и P_B. □