Лекция 1

Вычислимые функции

В этой главе мы обсудим фундаментальное понятие в Computer Science — понятие вычислимости. Мы изучим, какие функции можно вычислить с помощью алгоритмов, а какие нельзя.

В наши дни у всех уже есть интуитивное представление об алгоритмах. Всем нам приходится выполнять последовательность действий по данному набору правил (умножение в столбик, кулинарные рецепты, переход дороги и т.д.). Все мы сталкиваемся с компьютерными программами на самых разных языках программирования, а эти программы как раз реализуют разные алгоритмы.

Тем не менее, чтобы говорить о вычислимости формально, нам нужно четко сказать, что же такое алгоритм. С другой стороны, если начать это делать, то оказывается, что это довольно долго и муторно. Естественный способ: попробовать сказать, что алгоритм — это программа на каком-то языке программирования. Но тогда мы должны зафиксировать конкретную версию конкретного языка. Также полное и подробное описание какого-то языка программирования — дело большое. Дополнительно тяжело пришлось бы тем читателям, которые выбранный нами язык программирования не изучали. Наконец, такое описание понятия вычислимости могло бы неожиданно быстро устареть вместе с устареванием выбранного языка программирования.

В математике и теоретической информатике эта проблема обычно решается введением специального и довольно простого языка и модели для реализации алгоритмов — машин Тьюринга. Но даже и это дело не быстрое, к машинам Тьюринга нужно немного привыкнуть, прежде чем начать делать с ними что-нибудь интересное. Ну и даже такая простая вычислительная модель, как машина Тьюринга, порождает довольно много технических деталей, за обсуждением которых не всегда легко разглядеть суть.

Мы будем выходить из этой ситуации следующем образом. Сначала мы будем понимать алгоритмы *неформально* и интуитивно. Мы лишь зафиксируем несколько базовых свойств, которым должны удовлетворять алгоритмы. И окажется, что этих свойств будет вполне достаточно, чтобы рассказать всю содержательную часть, которую хочется рассказать о вычислимости, при этом не погружаясь в излишние

технические детали. А уже после того, как мы извлечем из базовых свойств всю нужную нам теорию, мы формализуем понятие алгоритма, определив машины Тьюринга. Мы проверим, что для машин Тьюринга все наши базовые свойства алгоритмов выполняются, и таким образом формализуем то, что мы обсудим в этой главе с нашим неформальным понятием алгоритма. Таким образом, мы сначала обсудим вычислимость по существу, а технические детали обсудим потом, когда уже будет понятно, зачем эти детали нужны.

Итак, в этой главе мы будем понимать алгоритмы неформально. Под алгоритмом мы можем понимать просто четкую последовательность инструкций на русском (или каком-то другом) языке, следуя которой, любой человек может не думая совершить требуемые действия и прийти к желаемому результату. Также под алгоритмом мы можем понимать программу на нашем любимом языке программирования. Оказывается, что любое из этих пониманий дает одинаковый результат, в том смысле, что можно выбрать любое из этих пониманий и все понятия в этой главе будут одни и те же (это почти так, все-таки нужно требовать, чтобы язык программирования был "разумным", но любой популярный язык программирования является разумным, и чтобы придумать неразумный язык, нужно сильно постараться).

Так что далее можно выбрать любое из этих пониманий и читать эту главу, имея в голове соответствующее понятие алгоритма. Полезно только сделать оговорку, что здесь мы никак не будем учитывать ограничения на ресурсы. Нам будет не важно, сколько времени потребуется работать алгоритму и какой объем памяти ему потребуется. Даже если требуемое время для реализации алгоритма на современном компьютере окажется больше продолжительности жизни человека (или дальше больше времени существования вселенной), а объём требуемой памяти (измеряемой, например, в битах) будет превосходить число атомов в солнечной системе, всё равно такой алгоритм нас вполне устраивает в нашей математической теории. Это может показаться не очень близким к реальности, но с другой стороны, мы увидим, что даже при таких вольностях и такой большой свободе алгоритмы окажутся принципиально неспособны справиться с некоторыми задачами. Поскольку главные результаты этой главы будут отрицательными, не страшно, что мы позволяем алгоритмам слишком много. Если уж такие алгоритмы не способны справиться с какой-то задачей, то обычные алгоритмы из реальной жизни не справятся и подавно.

Хоть мы и позволяем себе иметь неформальное понятие алгоритма и более того, позволяем себе выбрать удобное нам понятие, нам всё же нужно зафиксировать основные свойства алгоритмов, на которые мы сможем опираться. Давайте перечислим эти свойства.

Свойство 0 Алгоритм имеет конечное описание.

Действительно, это довольно естественное свойство, которое выполняется для обычных алгоритмов в жизни. Когда мы выполняем последовательность инструкций, то у нас есть полный список этих инструкций. Когда мы имеем дело с программой на каком-то компьютерном языке, то она имеет какое-то конечное число строк (программа может дополнительно вызывать какие-то библиотеки, но и в них

тоже есть какое-то конечное число строк). Заметим, что мы не накладываем тут никаких ограничений на размер описания. Важно лишь, что оно конечно. Так что любая сколь угодно сложная программа подпадает под это свойство.

Свойство 1 Алгоритм выполняется по шагам.

Это важное для нас свойство, и оно действительно выполняется для алгоритмов в жизни. Когда мы выполняем цепочку инструкций, то мы последовательно переходим от одной инструкции к другой. Когда мы пишем программу, то у нас есть возможность запустить программу по шагам.

Свойство 2 На каждом шаге алгоритма должно быть четко и однозначно определено, что нужно делать на следующем шаге.

Это принципиальное свойство. Все действия в алгоритме должны быть однозначно определены, не должно быть никакой неопределенности и недосказанности.

Вот, собственно, и все основные свойства алгоритмов, за которыми нам потом нужно будет проследить, чтобы они выполнялись при формализации понятия алгоритма. В действительности чуть позже мы добавим еще некоторые свойства, но для начала нам хватит этих.

Прежде чем перейти к обсуждению вычислимости, нам также нужно обсудить, в каком виде алгоритм получает входные данные и в каком виде он выдает результат работы. Если посмотреть на пример с компьютерами, то можно заметить, что в реальности компьютер работает с битами, каждый из которых может быть нулем или единицей. Так что и входом, и выходом алгоритма является последовательность таких битов. С другой стороны, со стороны пользователя входом или выходом программы может быть число, массив чисел, строка символов или, например, текстовый файл. Таким образом, компьютер работая с последовательностями битов, кодирует с помощью них другие форматы входных и выходных данных. Если мы говорим о выполнении человеком инструкций, то тут тоже входными и выходными данными могут быть самые разные объекты: числа, наборы чисел, текст и т.д. Но всегда можно считать, что входом и выходом алгоритма являются последовательности символов в некотором конечном алфавите. В частности, и числа мы обычно задаем в десятичной записи, то есть как последовательность цифр. При этом мы всегда можем ограничиться двумя символами 0 и 1 — компьютеры так и кодируют входы программ.

Нам при этом на самом деле будет удобнее работать с алгоритмами, которые получают на вход и выдают на выход натуральные числа. В действительности это то же самое, что работать с конечными последовательностями нулей и единиц. Действительно, каждой последовательности $a \in \{0,1\}^n$ можно поставить в соответствие число с двоичной записью $1a$, что задает взаимно однозначное соответствие между последовательностями нулей и единиц и положительными натуральными числами.

Задача 1.1. Понятно ли, зачем нужно рассматривать число с двоичной записью $1a$, а не просто a? Подсказка: какие числа задаются двоичными записями 1 и 01?

Задача 1.2. Понятно ли, как получить взаимно однозначное соответствие между последовательностями нулей и единиц и натуральными числами, начиная с нуля? Подсказка: нужно слегка поправить конструкцию выше.

Таким образом, когда мы будем говорить об алгоритмах, мы будем в основном говорить об алгоритмах, получающих на вход и выдающих на выход натуральные числа.

Определение 1.1. Функция $f\colon \mathbb{N} \to \mathbb{N}$ называется вычислимой, если существует алгоритм P, который на любом входе $n \in \mathbb{N}$ выдает $f(n)$.

У алгоритмов есть такая особенность, что они в принципе не обязаны выдавать какой-то ответ. На некоторых входах вполне может получиться так, что цепочка инструкций или программа приводит либо к завершению работы без ответа, либо и вовсе к зацикливанию, то есть продолжению выполнения алгоритма до бесконечности без выдачи ответа. Было бы хорошо отразить эту возможность в понятии вычислимой функции.

Для этого в этой главе под записью $f\colon A \to B$ мы будем понимать *частичные* функции. То есть на некоторых аргументах $n \in A$ функция может быть не определена, и в этом случае $f(n)$ может быть не определено. Если функция f определена на всех элементах $n \in A$, мы говорим, что функция f *всюду определенная* или *тотальная*.

Наше определение вычислимой функции напрямую распространяется на не всюду определенные функции. Для тех аргументов $n \in \mathbb{N}$, для которых $f(n)$ не определено, алгоритм P при этом не выдает никакого значения на выход.

Пример 1.3. Все функции на натуральных числах, с которыми мы сталкиваемся на регулярной основе, являются вычислимыми. Функция $f(n) = n$ вычислима: достаточно, получив n на вход, сразу подать его на выход. Функция $f(n) = n^2$ также вычислима: получив n на вход, нужно возвести его в квадрат и передать результат на выход. Операция возведения в квадрат реализована в языках программирования. Человек же может реализовать ее "руками" с помощью умножения в столбик. Функция $f(n) = 2^n$ также вычислима. Достаточно умножить число 2 само на себя n раз. Функция $f(n) = \lceil \log n \rceil$ (не определенная для $n = 0$) вычислима: взятие логарифма и целой части реализовано в языках программирования. Если же мы хотим вычислить эту функцию "элементарными средствами", то можно перебирать все m от 1 до n и последовательно для каждого вычислять 2^m. На выход тогда нужно будет выдать последнее m, для которого $2^m \leqslant n$ (для $n = 0$ такого m не будет, так что и никакого результата мы не выдадим). Ничего, что предложенный нами алгоритм — не самый эффективный с точки зрения числа шагов работы, — как мы писали выше, для нас это и не важно.

Лемма 1.1. *Если две функции $f, g\colon \mathbb{N} \to \mathbb{N}$ вычислимы, то и их композиция $g \circ f$ вычислима.*

Стоит уточнить, что если для какого-то $n \in \mathbb{N}$ функция f не определена, то не определена будет и композиция $g \circ f$. Если $f(n)$ определено, но функция g не определена на $f(n)$, то вновь композиция $g \circ f$ не определена на n. Во всех остальных случаях композиция определена (и равна $g(f(n))$).

Доказательство леммы 1.1. Действительно, для вычисления композиции мы можем применить следующий алгоритм. Получив на вход n, мы сначала запускаем алгоритм для вычисления f. Если он что-то выдал, то запускаем на результате алгоритм для вычисления g. Если он в свою очередь что-то выдает, то мы подаем это на выход. Таким образом, наш алгоритм выдаст что-либо только в тех случаях, когда композиция определена, и при этом выдаст правильное значение. □

Пример 1.4. Теперь легко заметить, что, например, функции $2^{n^2}, 2^{2^n}, \lceil \log n \rceil^n$ вычислимы как композиции вычислимых функций.

В целом, повторимся, все функции на натуральных числах, с которыми мы обычно сталкиваемся на практике, вычислимы. Чтобы предъявить невычислимую функцию, нужно постараться. Этим мы и займемся в этой главе.

Чтобы лучше понять суть понятия вычислимости, полезно обсудить следующее **неправильное утверждение**. Чтобы его сформулировать, нам нужно понятие продолжения функции. Функция $g\colon \mathbb{N} \to \mathbb{N}$ называется продолжением функции $f\colon \mathbb{N} \to \mathbb{N}$, если для всякого $n \in \mathbb{N}$, при котором определено $f(n)$, определено и $g(n)$, и при этом $f(n) = g(n)$. Таким образом, g совпадает с f на области определения последней, но может также быть определена и для других значений n.

Итак, давайте попробуем доказать следующее утверждение (которое, подчеркнем еще раз, на самом деле неверно): *для всякой вычислимой функции* $f\colon \mathbb{N} \to \mathbb{N}$ *существует всюду определенное вычислимое продолжение* $g\colon \mathbb{N} \to \mathbb{N}$. Кажется, что это утверждение вполне разумно и доказать его можно так. Построим алгоритм для g. Получив на вход n, запустим алгоритм для вычисления f. Если алгоритм выдал какой-то результат, то подаем его на выход. Если нет, то выдаем произвольное значение (скажем, 0). Может показаться, что действительно это рассуждение показывает, что всякую вычислимую функцию можно вычислимо продолжить на все натуральные числа. Но на самом деле в этом рассуждении есть ошибка. Прежде чем читать дальше, полезно попробовать найти ошибку самостоятельно.

* * *

Ошибка в этом рассуждении состоит в том, что алгоритм для вычисления f может не выдать ответа довольно неприятным для нас образом: алгоритм может продолжать работать, и так и не выдавать ответа. И тогда в рассуждении выше не ясно, когда же следует решить, что алгоритм для f не выдал ответа и прервать его: то ли алгоритм уже зациклился и не остановится, то ли он еще делает содержательную работу и через какое-то время выдаст результат.

Глядя на это, хочется немного поправить рассуждение выше и как-то решить эту проблему: может быть, надо просто дать алгоритму поработать достаточно долго,

и тогда станет ясно, выдаст он ответ или нет. В действительности все эти попытки обречены на неудачу: не только наше доказательство содержит ошибку, но и утверждение, которое мы пытаемся доказать, не верно! А именно, существует вычислимая функция, у которой нет всюду определенного вычислимого продолжения. Мы докажем это немного позже.

Но важный вывод, который мы должны сделать из разобранного ошибочного доказательства, состоит в том, что алгоритм, который не выдает ответ, может делать это, продолжая работать, так что по нему может быть не ясно, планирует он выдать ответ в будущем или нет. Так что конструкции вида "запустим некоторый алгоритм, и если он ничего не выдаст, то сделаем то-то и то-то" опасны. Если мы хотим воспользоваться такой конструкцией, то нам нужно дополнительно как-то гарантировать, что запускаемый алгоритм остановится (пусть и не выдавая ответа).

Хоть, как мы уже упоминали, привести пример невычислимой функции не так легко, тем не менее, совсем не трудно понять, что невычисилмые функции существуют.

Лемма 1.2. *Существует невычислимая функция* $f\colon \mathbb{N} \to \mathbb{N}$.

Доказательство. В главе ? мы обсуждали, что множество функций из натуральных чисел в натуральные числа континуально. С другой стороны, как мы договорились выше, каждый алгоритм имеет конечное описание, то есть является конечной последовательностью символов в фиксированном конечном алфавите. Множество конечных последовательностей из конечного множества символов счетно.

Таким образом, множество функций континуально, а множество алгоритмов счетно. При этом каждый алгоритм вычисляет не более одной функции. Так что алгоритмов просто не хватит на все функции, а значит, существует невычислимая функция. □

В этом доказательстве мы вновь применили мощностной аргумент. Он позволил нам доказать существование невычислимой функции, но не позволяет предъявить никакую конкретную невычислимую функцию. Для этого нам потребуется другое рассуждение.

Вторым базовым объектом, кроме функций, для нас являются множества.

Определение 1.2. Множество $A \subseteq \mathbb{N}$ называется разрешимым, если существует алгоритм, который для всякого входа $n \in \mathbb{N}$ выдает 1, если $n \in A$, и выдает 0, если $n \notin A$.

Заметим, что алгоритм, разрешающий какое-либо множество, всегда выдает что-то в качестве ответа.

Разрешимость множеств можно сформулировать в терминах вычислимости функций. Для этого введем такое определение.

Определение 1.3. Характеристической функцией множества $A \subseteq \mathbb{N}$ называется функция $\chi_A \colon \mathbb{N} \to \{0, 1\}$, для которой

$$\chi_A(n) = \begin{cases} 1 & n \in A, \\ 0 & n \notin A. \end{cases}$$

Лемма 1.3. *Множество $A \subseteq \mathbb{N}$ разрешимо тогда и только тогда, когда функция χ_A вычислима.*

Доказательство. Действительно, алгоритм, разрешающий множество A, по определению есть в точности алгоритм, вычисляющий функцию χ_A. □

Свойство множества "быть разрешимым" замкнуто относительно основных теоретико-множественных операций.

Лемма 1.4. *Если множества $A, B \subseteq \mathbb{N}$ разрешимы, то разрешимы и множества $A \cap B, A \cup B, \mathbb{N} \setminus A$.*

Доказательство. Предъявим алгоритм, разрешающий $A \cap B$. Получив на вход n, запустим сначала алгоритм, разрешающий A. Когда он выдаст ответ (а он сделает это по определению разрешимости), запустим алгоритм, разрешающий B, на том же входе. Когда и второй алгоритм выдаст ответ, то мы выдадим ответ 1, если оба алгоритма выдали ответ 1, и выдадим ответ 0 иначе.

Алгоритм, разрешающий множество $A \cup B$, устроен почти так же, нужно лишь поменять правило, по которому выдается выход. Если хотя бы один из промежуточных алгоритмов выдает 1, то и мы выдаем 1, а если оба выдают 0, то мы также выдаем 0.

Наконец, алгоритм, разрешающий множество $\mathbb{N} \setminus A$, устроен совсем просто. На входе n мы запускаем алгоритм для разрешения множества A и выдаем 1, если он выдал 0, и выдаем 0, если он выдал 1. □

Замечание 1.1. Альтернативно, можно доказывать разрешимость нужных нам множеств через вычислимость характеристических функций. Для этого достаточно заметить, что $\chi_{A\cap B}(n) = \chi_A(n) \cdot \chi_B(n)$, $\chi_{A\cup B}(n) = 1 - (1 - \chi_A(n)) \cdot (1 - \chi_B(n))$, $\chi_{\mathbb{N}\setminus A}(n) = 1 - \chi_A(n)$. Теперь вычислимость этих функций легко следует из вычислимости функций χ_A и χ_B.

Задача 1.5. Докажите, что если множества $A, B \subseteq \mathbb{N}$ разрешимы, то разрешимо и множество $A \triangle B$.

1.1 Перечислимые множества

Наряду с разрешимыми множествами оказывается важным также понятие перечислимого множества.

Определение 1.4. Множество $A \subseteq \mathbb{N}$ называется перечислимым, если существует алгоритм P, который при запуске на пустом входе (то есть не получая на вход ничего) выдает все элементы множества A. Другими словами, по мере своей работы алгоритм P постепенно выдает последовательность $p_0, p_1, p_2, \ldots$ (которая может быть как конечной, так и бесконечной), причём все элементы последовательности лежат в A и, наоборот, для всякого $ainA$ существует k, для которого $p_k = a$. В частности, алгоритм P может не останавливаться. Важно лишь, чтобы всякий элемент A был выдан через конечное число шагов работы алгоритма.

Замечание 1.2. Заметим, что мы не требуем, что элементы множества A не повторялись в выдаче алгоритма P. Необходимо лишь, чтобы все элементы A в последовательности встретились, а ничего, кроме элементов A, не встретилось.

Задача 1.6. Докажите, что если множество $A \subseteq \mathbb{N}$ перечислимо, то есть алгоритм, перечисляющий элементы множества A без повторений.

Понятие перечислимости оказывается более широким, нежели понятие разрешимости.

Лемма 1.5. *Если множество $A \subseteq \mathbb{N}$ разрешимо, то оно перечислимо.*

Доказательство. Пусть P — алгоритм, разрешающий A. Построим алгоритм, перечисляющий A. Ничего не получая на вход, этот алгоритм перебирает последовательно натуральные числа $0, 1, 2, \ldots$ и для каждого из них запускает алгоритм P для проверки, лежит ли это число в A. Когда алгоритм P выдает результат (а он это обязательно делает по определению разрешимости), то мы выдаем наше число на выход, если P выдал 1, и не выдаем в противном случае. После этого мы переходим к следующему числу. Результирующий алгоритм печатает в точности все элементы множества A (причем в порядке возрастания). □

Алгоритм в доказательстве предыдущей леммы никогда не остановится. Но это и не требуется от алгоритма, перечисляющего множество.

Как и разрешимость, свойство перечислимости сохраняется при применении к множествам некоторых базовых операций.

Лемма 1.6. *Если множества $A, B \subseteq \mathbb{N}$ перечислимы, то перечислимы и множества $A \cap B$ и $A \cup B$.*

Доказательство этой леммы несколько сложнее, чем доказательство аналогичной леммы для разрешимых множеств. По сути, здесь мы первый раз используем одно из базовых свойств алгоритмов — пошаговую работу.

Доказательство. Сначала построим алгоритм, перечисляющий $A \cup B$. Для этого рассмотрим алгоритмы P_A и P_B, перечисляющие A и B соответственно.

Естественной идеей было бы попробовать сделать то же самое, что и в случае разрешимых множеств: сначала запустить алгоритм P_A, а затем запустить алгоритм P_B. Но в текущей ситуации у нас есть существенное отличие от случая разрешимых

множеств: там у нас была гарантия, что алгоритмы завершат свою работу. Здесь же у нас такой гарантии нет, и если алгоритм P_A свою работу не заканчивает, то мы никогда не перейдем к выполнению алгоритма P_B.

Вместо этого наш алгоритм будет работать следующим образом. Сначала мы запустим алгоритм P_A на один шаг его работы. После этого мы временно прерываем работу алгоритма P_A и запускаем алгоритм P_B на один шаг. После этого мы прерываем работу алгоритма P_B и запускаем P_A на второй шаг его работы. После этого мы делаем второй шаг работы алгоритма B, и так далее.

Таким образом работа наша алгоритма состоит из следующих итераций: для каждого $k = 1, 2, 3, \ldots$ на итерации с номером k мы запускаем сначала алгоритм P_A на еще один шаг, а затем алгоритм P_B на еще один шаг. После k-ой итерации в обоих алгоритмах P_A и P_B у нас будет сделано k шагов. По сути, мы запускаем наши алгоритмы параллельно по шагам.

Если какой-то из алгоритмов заканчивает свою работу после некоторого числа ходов, то ничего страшного, на всех последующих итерациях шаг в нем делаться уже не будет.

Осталось понять, зачем же мы делаем все эти итерации. На каждой итерации, если на очередном шаге какой-то из алгоритмов P_A и P_B выдает на выход какое-то число, то и мы тоже выдаем это число на выход.

Ясно, что построенный таким образом алгоритм будет печатать только те числа, которые лежат в $A \cup B$. С другой стороны, все элементы этого множества будут напечатаны. Действительно, если $x \in A \cup B$, то $x \in A$ или $x \in B$. Значит, один из алгоритмов P_A и P_B выдаст на выход число x на каком-то шаге k. Тогда наш алгоритм выдаст x на своей k-ой итерации.

Таким образом, построенный алгоритм действительно перечисляет $A \cup B$.

Алгоритм, перечисляющий $A \cap B$ строится похожим образом. Только нам не следует сразу выдавать на выход все, что выдают алгоритмы P_A и P_B. Вместо этого мы отдельно храним списки элементов, уже выданных P_A и уже выданных P_B (храним в массиве, если под алгоритмами понимаем программы, и записываем на отдельных листочках, если алгоритм выполняется “вручную”). Как только один из алгоритмов P_A и P_B на очередном шаге подает что-то на выход, мы проверяем, не было ли это же число раньше выдано другим алгоритмом (просто сравниваем его со всеми числами, ранее выданными другим алгоритмом). Если оно уже было выдано другим алгоритмом, то мы подаем его на выход, иначе нет.

Тогда в выходе нашего алгоритма будут встречаться только те числа, которые лежат в обоих множествах A и B. С другой стороны, всякое такое число будет выдано нашим алгоритмом. Действительно, пусть $x \in A \cap B$. Тогда P_A и P_B выдают x на выход на каких-то шагах, пусть это шаги k и n соответственно. Тогда наш алгоритм выдаст x на итерации $\max(k, n)$. □

Замечание 1.3. В связи с конструкцией в нашем доказательством может возникнуть такой вопрос. Почему мы решили запускать алгоритмы параллельно по шагам? Казалось бы гораздо естественнее запустить алгоритм P_A, подождать пока он выдаст какое-нибудь число, после этого запустить алгоритм P_B, подождать пока он выдаст

свое число, затем снова выполнять алгоритм P_A до появления следующего числа и так далее. Иными словами, почему бы не запускать алгоритмы параллельно, но прерывать их после выдачи очередного числа?

Так действительно можно сделать, если известно, что множества A и B оба являются бесконечными. Если же этого заранее не известно, то может так оказаться, что одно из множеств конечно, но алгоритм, перечисляющий его, работает бесконечно: сначала перечисляет все элементы множества, а потом бесконечно работает и ничего не печатает. Такое поведение допускается определением перечислимости. И тогда у нас может возникнуть проблема, что на очередной итерации мы запустим этот алгоритм, ожидая от него следующее число, алгоритм будет продолжать работать, и мы никогда не перейдем к следующим итерациям. А на следующих итерациях другой алгоритм мог напечатать что-то новое, что в нашей конструкции будет утеряно.

Есть несколько способов преодолеть эту трудность. Одним из них мы воспользовались в нашем доказательстве: если запускать алгоритмы параллельно по шагам, то такой проблемы не возникает, для всякого k каждый алгоритм дойдет до шага k независимо от того, как ведет себя другой алгоритм. Другой способ состоит в том, чтобы аккуратнее обработать случай конечных множеств. Мы сделаем это позже.

Можно заметить, что когда мы обсуждали устойчивость свойства разрешимости при применении операций над множествами, мы рассмотрели операцию взятия дополнения. Когда же мы доказывали аналогичные утверждения для перечислимых множеств, мы о дополнении множеств умолчали. И действительно, если вспомнить доказательство для разрешимых множеств, то не ясно, как его переносить на случай перечислимых множеств.

Оказывается, что и само утверждение неверно, существует перечислимое множество $A \subseteq \mathbb{N}$, такое что $\mathbb{N} \setminus A$ не является перечислимым. Мы докажем это позже.

Когда мы говорили о вычислимых функциях и разрешимых множествах, все было довольно естественно. Алгоритм вычислял функцию и разрешал множество в понятном и естественном смысле. Но затем мы начали обсуждать перечислимые множества, и здесь уже определение у нас довольно странное и непривычное. Зачем же мы стали изучать такой странный объект? Оказывается, что перечислимые множества очень тесно связаны с вычислимыми функциями. И если мы хотим изучать вычислимые функции, то нам в любом случае придется изучать и перечислимые множества тоже.

Мы показываем связь между вычислимыми функциями и перечислимыми множествами в следующей теореме.

Теорема 1.7. *Пусть $A \subseteq \mathbb{N}$ — некоторое множество натуральных чисел. Следующие утверждения эквивалентны:*

1. *A перечислимо;*
2. *A является областью значений некоторой вычислимой функции;*
3. *A является областью определения некоторой вычислимой функции;*

4. полухарактеристическая функция A

$$\chi_A^*(n) = \begin{cases} 0, & \text{если } n \in A, \\ \text{не определена}, & \text{если } n \notin A. \end{cases}$$

вычислима.

Доказательство. Начнем с простых соотношений между этими утверждениями.

Совсем нетрудно увидеть, что из 4 следует 3. Действительно, областью определения функции $\chi_A^*(n)$ как раз и является множество A. Раз эта функция вычислима, то A является областью определения вычислимой функции.

Докажем, что из 1 следует 4. Действительно, пусть множество A перечисляется алгоритмом P_A. Предъявим алгоритм, вычисляющий функцию χ_A^*. Получив на вход n, этот алгоритм запускает алгоритм P_A. Когда P_A выдает что-то на выход, мы сравниваем это число с n. Если оно совпадает с n, то мы останавливаемся и выдаем 0. Если очередное выданное P_A число не совпадает с n, то мы продолжаем работу алгоритма P_A. Если $n \in A$, то на каком-то шаге алгоритм P_A выдаст это число, и наш алгоритм выдаст 0. Если же $n \notin A$, то P_A никогда не выдаст n и наш алгоритм не выдаст никакого результата. Таким образом, наш алгоритм вычисляет в точности функцию χ_A^*.

Покажем теперь, что из 3 следует 2. Пусть A является областью определения вычислимой функции f и пусть P — алгоритм, вычисляющий f. Рассмотрим следующий алгоритм. Получив на вход n, мы запускаем алгоритм P на n. Если P выдает какой-то выход, то мы подаем на выход n. Заметим, что на входе $n \in A$ наш алгоритм выдаст n, поскольку n входит в область определения f. Если же $n \notin A$, то наш алгоритм не выдаст никакого выхода на входе n, поскольку выхода не выдает и P. Таким образом, наш алгоритм вычисляет некоторую функцию, и область ее значений совпадает с A.

Нам осталось доказать, что из 2 следует 1. Тогда мы установим эквивалентность всех четырех утверждений. Эта часть доказательства самая сложная, здесь нам вновь потребуется запускать несколько алгоритмов параллельно, но в несколько более сложной ситуации.

Пусть A является областью значений функции f, вычисляемой алгоритмом P. Построим алгоритм, перечисляющий A. Наш алгоритм вновь будет последовательно выполнять итерации. Сначала мы запускаем P на входе 0 на один шаг. Затем мы запускаем P на входе 1 на один шаг, после чего делаем второй шаг P на входе 0. Затем мы запускаем P на входе 2 на один шаг, затем делаем второй шаг P на входе 1, затем третий шаг P на входе 0. В общем случае, на итерации $k = 1, 2, 3, \ldots$ мы последовательно запускаем P на входе $k-1$ на один шаг, на входе $k-2$ на второй шаг, и так далее, на входе 0 на k-ый шаг.

Как только на каком-то входе на очередном шаге алгоритм P выдает какой-то выход, мы передаем его на выход нашего алгоритма (и продолжаем работу).

Ясно, что среди выходов нашего алгоритма встречаются только числа из области значений f (то есть из множества A). С другой стороны, всякий элемент области

значений f встретится среди выходов нашего алгоритма. Действительно, пусть $a = f(n)$. Тогда P на входе n выдает a на некотором шаге k. Тогда на итерации $n + k$ мы (среди прочего) запустим P на входе n на k шагов и выдадим a на выход. Таким образом, наш алгоритм перечисляет в точности множество A. □

Задача 1.7. Докажите напрямую, что область определения вычислимой функции перечислима.

Таким образом, оказывается, что перечислимые множества — это в точности области определения и области значений вычислимых функций. С другой стороны, мы получили и удобное эквивалентное определение перечислимых множеств: множество перечислимо тогда и только тогда, когда его полухарактеристическая функция вычислима.

Выше мы показали, что разрешимые множества являются перечислимыми. Но может быть, верно и обратное? Может быть мы зря мучаем себя сложным определением и от него можно избавиться? Оказывается, что это не так: существуют перечислимые неразрешимые множества. Мы докажем это позже.

Однако к свойству перечислимости можно добавить еще одно естественное свойство, чтобы получить условие, эквивалентное разрешимости.

Теорема 1.8 (теорема Поста)**.** *Множество $A \subseteq \mathbb{N}$ разрешимо тогда и только тогда, когда оба множества A и $\mathbb{N} \setminus A$ перечислимы.*

Доказательство. В одну сторону доказать утверждение совсем не сложно. Пусть A разрешимо. Тогда по уже доказанной лемме 1.5 множество A перечислимо. С другой стороны, по лемме 1.4 множество $\mathbb{N} \setminus A$ также разрешимо. И вновь по лемме 1.5 множество $\mathbb{N} \setminus A$ перечислимо.

Для доказательства в другую сторону мы используем прием, который у нас уже встречался. Пусть множества A и $\mathbb{N} \setminus A$ перечислимы, и пусть P_A и $P_{\mathbb{N}\setminus A}$ — перечисляющие их алгоритмы. Построим алгоритм, разрешающий A. Получив на вход число n, наш алгоритм запускает алгоритмы P_A и $P_{\mathbb{N}\setminus A}$ параллельно по шагам. На k-й итерации (для $k = 0, 1, 2, \ldots$) алгоритм выполняет k-ый шаг алгоритма P_A, а затем k-ый шаг алгоритма $P_{\mathbb{N}\setminus A}$. Если какой-то из алгоритмов P_A и $P_{\mathbb{N}\setminus A}$ после очередного шага выдает какое-то число на выход, то наш алгоритм сравнивает его с n. Если выданное число совпадает с n, то мы выдаем на выход 1, если число было выдано алгоритмом P_A, и выдаем 0, если число было выдано алгоритмом $P_{\mathbb{N}\setminus A}$. Если же выданное число не совпадает с n мы продолжаем работу.

Почему наш алгоритм работает правильно? Заметим, что всякое число $n \in \mathbb{N}$ лежит в ровно одном из двух множеств A и $\mathbb{N} \setminus A$. Так что для всякого n на какой-то итерации один из алгоритмов выдаст n, причем это будет P_A, если $n \in A$, и это будет $P_{\mathbb{N}\setminus A}$, если $n \notin A$. Так что описанный алгоритм действительно является разрешающим алгоритмом для множества A. □

1.2 Вычислимость и конечные объекты

Прежде чем двигаться дальше. стоит остановиться на одной особенности, которая существенно отличает нашу модель вычислимости от того, к чему мы привыкли в реальной жизни.

Лемма 1.9. *Если множество $A \subseteq \mathbb{N}$ конечно, то оно разрешимо.*

Это не вполне согласуется со здравым смыслом: если мы хотим на практике решать задачу о принадлежности числа какому-то множеству, то нам обычно вполне достаточно ограничиться числами, у которых число цифр не превышает числа атомов в солнечной системе. Тогда мы имеем дело с конечным множеством, и наша лемма говорит о том, что множество теперь разрешимо, хотя как его разрешать, попрежнему непонятно. Такое положение дел — следствие двух вещей. Во-первых, мы объявили, что нас не интересует время работы разрешающего алгоритма и длина его описания. Важно лишь, чтобы он был конечным и заканчивал работу за конечное время. Во-вторых, если посмотреть внимательно на определение разрешимости, то в нем требуется доказать, что разрешающий алгоритм *существует*. При этом от нас не требуется его предъявлять.

Доказательство. Пусть множество конечно: $A = \{a_1, \ldots, a_k\}$. Рассмотрим следующий алгоритм. Получив на вход n, алгоритм сравнивает его с числом a_1, затем сравнивает n с числом a_2, затем с a_3 и так далее до a_k. Если n совпадает с каким-то из этих чисел, то мы выдаем 1, иначе выдаем 0.

Наш алгоритм хранит числа $a_1, \ldots, a_k$ как константы и просто сравнивает с ними данный ему вход. Таким образом, мы показали, что алгоритм, разрешающий множество A, существует: достаточно взять алгоритм, который сравнивает вход с правильным набором констант. При этом остается совершенно не ясным, как по построить такой алгоритм. Ведь для него нужен список всех элементов множества A, а откуда его взять? Но от нас и не требовалось объяснять, как построить алгоритм для A. По определению разрешимости достаточно доказать, что такой алгоритм есть, — а это мы сделали. □

Аналогичное утверждение можно доказать и для функций на конечном множестве.

Лемма 1.10. *Пусть функция $f\colon \mathbb{N} \to \mathbb{N}$ имеет конечную область определения. Тогда f вычислима.*

Доказательство. Пусть областью определения f является множество $\{a_1, \ldots, a_k\}$. Обозначим $b_i = f(a_i)$ для всех $i = 1, \ldots, k$. Рассмотрим следующий алгоритм. Получив на вход число n, алгоритм сравнивает его с числом a_1, затем сравнивает n с числом a_2, затем с a_3, и так далее до a_k. Если n совпадает с каким-то числом a_i, то мы выдаем b_i на выход. Если нет — не выдаем ничего.

Здесь алгоритм хранит в качестве констант числа $a_1, \ldots, a_k$, а также числа $b_1, \ldots, b_k$. Видно, что предъявленный алгоритм вычисляет функцию f. □

Задача 1.8. Докажите, что множество таких натуральных чисел n, что в десятичной записи числа π встречается n идущих подряд девяток, разрешимо.

1.3 Универсальная вычислимая функция

Для того, чтобы доказать разные утверждения, обещанные ранее в этой главе, нам потребуется зафиксировать еще одно фундаментальное свойство алгоритмов. Для этого нам нужно следующее определение.

Определение 1.5. Функция $U\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ называется универсальной вычислимой функцией для класса вычислимых функций от одной переменной, если

1. U вычислима;
2. для всякой вычислимой функции $f\colon \mathbb{N} \to \mathbb{N}$ существует такое n, что для всякого x верно $f(x) = U(n, x)$.

Иными словами, универсальная вычислимая функция U должна быть вычислима сама; кроме того, если фиксировать ее первый аргумент n всеми возможными способами ($n = 0, 1, 2, \ldots$), то среди получающихся функций от второго аргумента должны встречаться все возможные вычислимые функции (заметим, что тут не возникает трудности из-за мощностей: множество возможных значений n счетно, но счетно и множество вычислимых функций).

Для $n \in \mathbb{N}$ удобно ввести обозначение $U_n\colon \mathbb{N} \to \mathbb{N}$. По определению полагаем $U_n(x) = U(n, x)$ для всякого $x \in \mathbb{N}$.

	0	$\cdots$	x	$\cdots$
0	$U(0,0)$	$\cdots$	$U(0,x)$	$\cdots$
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\ddots$
n	$U(n,0)$	$\cdots$	$U(n,x)$	$\cdots$
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\ddots$

Таблица 1.1: универсальная вычислимая функция

Универсальную функцию удобно изображать в виде таблицы (см таблицу 1.1). Строки таблицы нумеруются значениями первого аргумента функции, а столбцы — значениями второго аргумента. Тогда в ячейке на пересечении строки n и столбца

x записывается значение $U(n, x)$ (и пишется "не определено", если это значение не определено). Если посмотреть на отдельную строку с номером n в этой таблице, то в ней будут записаны значения функции U_n. Тогда второе условие в определении универсальной функции означает, что среди строк таблицы встречаются все вычислимые функции от одной переменной.

Замечание 1.4. Заметим, что мы определяли понятие вычислимости для функций натурального аргумента, а здесь хотим говорим о вычислимости функции, определённой на парах натуральных чисел. Но, как мы отмечали в самом начале, по существу это ничего не меняет. Все наши определения напрямую обобщаются на функции, получающие на вход конечные наборы чисел, а также на множества, состоящие из таких наборов. В дальнейшем мы будем говорить о вычислимости функций на $\mathbb{N}^k$ и о разрешимости подмножеств $\mathbb{N}^k$, не останавливаясь на этом дополнительно.

Теперь мы готовы сформулировать еще одно важное свойство алгоритмов.

Свойство 3 Существует универсальная вычислимая функция U для класса вычислимых функций одной переменной.

Неформальное доказательство. Мы должны будем дать строгое доказательство этого утверждения, когда формализуем понятие алгоритма. Но сейчас стоит пояснить, почему это свойство естественно.

Полезно представлять себе первый аргумент n универсальной функции как программу (или текст алгоритма на русском языке), а второй аргумент x как обычное натуральное число. Тогда универсальная функция будет делать следующее: для данного текста программы n и данного числа x она запускает программу n на входе x и выдает результат ее работы (если программа n ничего не выдает, то и универсальная функция ничего не подает на выход).

Почему вообще можно считать, что первый аргумент U — это программа, хотя на самом деле это число? На самом деле каждой программе на каком-либо языке программирования можно поставить в соответствие число и наоборот. Например, можно сделать это так. Предположим, что наш язык программирования (или наше описание алгоритмов на русском языке) использует k различных символов. Мы можем сопоставить каждому символу цифру в k-ичной системе счисления. И тогда, поскольку программа — это последовательность символов, её можно считать записью некоторого числа в k-ичной системе счисления. Именно это число можно и поставить в соответствие нашей программе. Понятно, что такое соответствие сопоставляет с любой содержательной программой какое-то огромное число. Но для нас это не важно, поскольку мы не интересуемся временем работы программ. Можно заметить ещё, что большинству чисел соответствуют бессмысленные программы, которые не будут работать хотя бы потому, что не выполняют синтаксические правила языка программирования. Но и это не страшно, такие программы будут вычислять нигде не определенные функции. Что для нас принципиально — это что каждой программе соответствует некоторое уникальное число.

Теперь легко объяснить принцип работы алгоритма, вычисляющего универсальную вычислимую функцию U. По данным числам n и x алгоритм сначала выписывает программу, соответствующую числу n, а затем по шагам выполняет эту программу на входе x.

Полезно заметить также, что ровно это делает компьютер, когда компилирует и выполняет написанную кем-то программу. В итоге по данному тексту программы и данному входу выполняется заданная программа на данном входе. То же самое делает человек, когда по данному ему набору инструкций выполняет последовательность действий с заданными входными данными. □

Имея в своем распоряжении универсальную функцию, мы можем доказывать отрицательные результаты. Начнем с результата, противоположного неверному утверждению, которое мы «доказывали» в начале главы.

Теорема 1.11. *Существует вычислимая функция $f\colon \mathbb{N} \to \mathbb{N}$, не имеющая всюду определенного вычислимого продолжения.*

Доказательство. Пусть $U\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ — универсальная вычислимая функция для класса вычислимых функций одной переменной.

Рассмотрим ее «диагональную» функцию $d(n) = U(n, n)$. Ее значения записаны на диагонали таблицы 1.1. Функция d вычислима, поскольку вычислима функция U.

Положим $f(n) = d(n) + 1$. Эта функция также вычислима как композиция двух вычислимых функций (функции d и прибавления единицы).

Мы хотим доказать, что у f нет всюду определенного вычислимого продолжения. Предположим противное. Пусть $g\colon \mathbb{N} \to \mathbb{N}$ является всюду определенным вычислимым продолжением f.

Поскольку g вычислима, то она встречается среди строк таблицы 1.1, то есть существует k, для которого $g = U_k$. Посмотрим на пересечение строки k с диагональю, то есть на клетку (k, k) таблицы. С одной стороны, там написано значение $d(k)$ (поскольку значения d написаны на диагонали). С другой стороны, там написано значение $g(k)$ (поскольку значения g записаны в k-ой строке). Значит $d(k) = g(k)$.

Рассмотрим два случая. Пусть $d(k)$ не определено. Тогда мы сразу получаем противоречие, ведь g — всюду определенная функция, а значит, $g(k)$ определено. Пусть теперь $d(k)$ определено. Тогда $f(k) = d(k)+1$ и также определено. Поскольку g является продолжением f, то это означает, что $g(k) = f(k)$. В итоге получаем, что $g(k) = f(k) = d(k) + 1$, что также противоречит тому, что $g(k) = d(k)$.

В обоих случаях мы пришли к противоречию, а значит, функция f не имеет всюду определенного вычислимого продолжения. □

Доказательство этой теоремы очень похоже на доказательство несчетности множества бесконечных последовательностей из нулей и единиц. Точно так же, как раньше мы располагали последовательности, теперь мы располагаем вычислимые функции в виде таблицы. Затем мы рассматриваем диагональ этой таблицы (функция d). Затем мы изменяем последовательность на диагонали во всех точках. Мы

сделали это в два действия: сначала поменяли значения там, где d определена (получилась функция f), затем, пользуясь предположением, поменяли значения во всех точках, где d не определена (получилась функция g). Затем рассмотрели строку, в которой располагается получившаяся последовательность значений функции и пришли к противоречию на пересечении этой строки и диагонали.

Теперь мы можем доказать, что существует перечислимое неразрешимое множество.

Теорема 1.12. *Существует перечислимое неразрешимое множество $K \subseteq \mathbb{N}$.*

Доказательство. Рассмотрим функцию d из доказательства теоремы 1.11 и возьмем в качестве K ее область определения. (Это же множество является областью определения функции f из доказательства той же теоремы.) Множество K перечислимо как область определения вычислимой функции.

Предположим, что множество K разрешимо. Рассмотрим функцию

$$g(n) = \begin{cases} f(n), & \text{если } n \in K, \\ 0, & \text{если } n \notin K. \end{cases}$$

Функция g является всюду определенным продолжением f. С другой стороны, легко понять, что функция g является вычислимой. Действительно, g вычисляется следующим алгоритмом. По входу n сначала (с помощью алгоритма, разрешающего K) проверяем, лежит ли n в K. Если не лежит, то сразу выдаем 0. Если лежит, то поскольку K является областью определения f, то $f(n)$ определено. Запустим алгоритм для вычисления f на входе n и выдадим результат.

Таким образом, мы построили всюду определенное вычислимое продолжение f, которого не существует по теореме 1.11. Противоречие, а значит, K неразрешимо. □

Также теперь мы можем доказать неразрешимость «проблемы остановки».

Определение 1.6. Рассмотрим множество $\mathsf{Halt} \subseteq \mathbb{N}\times\mathbb{N}$, состоящее из таких пар (n, x), что $U(n, x)$ определено. Проблема остановкм состоит в выяснении того, принадлежит ли данная пара множеству Halt.

Неформально, в проблеме остановки требуется по данной программе и данному входу определить, выдаст ли программа результат на этом входе, то есть закончит ли она успешно свою работу[1].

Теорема 1.13. *Множество* Halt *неразрешимо.*

Прежде чем переходить к доказательству, отметим, что множество Halt перечислимо, как область определения вычислимой функции.

[1] Обычно в проблеме остановки спрашивается, остановится ли программа. Но легко понять, что эта постановка эквивалентна нашей.

Доказательство. Предположим, что множество Halt разрешимо, то есть существует алгоритм P, который на данной паре (n, x) выдает 1, если $(n, x) \in \mathsf{Halt}$, и выдает 0 иначе. Но тогда множество K из доказательства теоремы 1.12 также разрешимо. Действительно, его можно было бы разрешать следующим алгоритмом. Получив на вход n запускаем алгоритм P на входе (n, n) и выдаем результат его работы на выход. Этот алгоритм разрешает множество K, поскольку Halt является областью определения функции $U(n, x)$, а K является областью определения функции $d(n) = U(n, n)$. Но по теореме 1.12 множество K неразрешимо, противоречие. □

1.4 Вычислимая биекция между $\mathbb{N}$ и $\mathbb{N} \times \mathbb{N}$

Мы уже обсуждали, что (в разумных пределах) не так важно, что подается на вход алгоритму, потому что все кодируется с помощью конечных двоичных последовательностей. Для дальнейшего нам будет удобно зафиксировать конкретное соответствие между двумя типами входов алгоритма: $\mathbb{N}$ и $\mathbb{N} \times \mathbb{N}$. Более точно, мы хотели бы зафиксировать биекцию между этими множествами, вычислимую в обе стороны.

Сама биекция нам уже знакома, это та же самая биекция, которую мы строили, когда доказывали равномощность этих множеств. Наша задача состоит в том, чтобы перенумеровать пары натуральных чисел, для этого достаточно выписать их в последовательность. Сначала выпишем пару $(0, 0)$. Затем выпишем пары $(1, 0), (0, 1)$. Затем выпишем пары $(2, 0), (1, 1), (0, 2)$. На k-ой итерации (для $k = 0, 1, 2, \ldots$) мы выписываем пары $(k, 0), (k - 1, 1), \ldots, (0, k)$, то есть пары с суммой чисел, равной k. Таким образом мы выпишем все пары натуральных чисел. Теперь каждой паре поставим в соответствие ее номер в последовательности (нумерация начинается с нуля). Описанный процесс по сути является алгоритмом. Так что по данной нам паре натуральных чисел мы можем определять ее номер: выписываем последовательно все пары, пока не встретим нашу, и выдаем порядковый номер нашей пары в последовательности. Таким образом мы построили вычислимую биекцию из $\mathbb{N} \times \mathbb{N}$ в $\mathbb{N}$. Мы будем обозначать результат применения этой биекции к паре (k, n) через $[k, n]$ (таким образом, $[k, n]$ — это натуральное число).

Задача 1.9. На самом деле построенная биекция задается простой формулой. Найдите эту формулу.

Не так сложно вычислить и обратную биекцию. По данному числу m нам нужно найти пару, которой соответствует это число. Будем выписывать все пары, как описано выше, пока не выпишем m-ю пару. Подадим m-ю пару на выход. Нам будет удобно обозначать первую координату пары, которую мы подаем на выход через $l(m)$, а вторую координату через $r(m)$. В частности, мы получаем, что для всех k, n верно $l([k, n]) = k$, $r([k, n]) = n$, а для всех m верно $[l(m), r(m)] = m$.

Задача 1.10. На самом деле утверждение о вычислимости обратной биекции верно в общем случае. Докажите, что для всякой вычислимой биекции $f\colon \mathbb{N} \to \mathbb{N}$ (для простоты формулировки рассмотрим биекции на натуральных числах) обратная биекция также вычислима.

1.5 Главная универсальная функция

На самом деле мы пока все еще не выделили все свойства алгоритмов, которые есть у алгоритмов в реальной жизни.

Следующее важное определение позволяет добавить еще одно ключевое свойство.

Определение 1.7. Универсальная вычислимая функция $U\colon \mathbb{N}\times\mathbb{N}\to\mathbb{N}$ для класса вычислимых функций от одной переменной называется главной (или геделевой), если для любой вычислимой функции $V\colon \mathbb{N}\times\mathbb{N}\to\mathbb{N}$ существует такая всюду определенная вычислимая функция $s\colon \mathbb{N}\to\mathbb{N}$, что для всякого $n\in\mathbb{N}$ и для всякого $x\in\mathbb{N}$ верно $U(s(n),x)=V(n,x)$, или, другими словами, для всякого $n\in\mathbb{N}$ верно $U_{s(n)}=V_n$.

Интуитивно смысл понятия главности можно понимать так. Будем смотреть на U как на язык программирования: по данной программе n и входу x функция U запускает программу n на входе x. Разным универсальным функциям U и U' соответствуют разные языки программирования: одно и то же число n соответствует разным программам U_n и U'_n в разных универсальных функциях. Представим себе, что V в определении выше — это тоже некий язык программирования. Он может быть даже не универсальным, то есть не способным вычислить все вычислимые функции. Но будем смотреть на его первый аргумент n как на программу, которую V запускает на своем втором аргументе x. Тогда свойство главности U говорит, что для всякого другого языка программирования V существует вычислимый *интерпретатор* s этого языка в U. Этот интерпретатор получает на вход программу n в языке V и выдает в результате своей работы программу $s(n)$ в языке U, которая работает так же, как n в языке V, то есть $U_{s(n)}=V_n$.

Определение главности довольно трудное, но у него есть простое и понятное применение. Если смотреть на U как на язык программирования, то свойство главности означает, что мы можем по программе n в этом языке вычислять номера несколько модифицированных программ. Другими словами, к программам в языке U легко применять небольшие изменения. Разберем несколько простых примеров.

Пример 1.11. Пусть $U\colon \mathbb{N}\times\mathbb{N}\to\mathbb{N}$ — главная универсальная функция. Тогда существует всюду определенная вычислимая функция $s\colon \mathbb{N}\to\mathbb{N}$, которая по всякой программе n строит программу $s(n)$, выдающую ответ на 1 больше, чем программа n. Другими словами, $U_{s(n)}(x)=U_n(x)+1$. Действительно, положим $V(n,x)=U(n,x)+1$. Тогда функция V вычислима, и в силу того, что U главная, существует такая всюду определенная вычислимая функция $s\colon \mathbb{N}\to\mathbb{N}$, что $U_{s(n)}(x)=V_n(x)=U_n(x)+1$ для всех n и x.

Понятно, что точно так же мы можем применять и другие преобразования к выходу программы (умножать на 2, возводить в квадрат и т.д.). Достаточно лишь немного иначе определить V.

Также мы можем применять изменения и ко входу программы. Например, можно доказать, что существует всюду определенная вычислимая функция $s\colon \mathbb{N}\to\mathbb{N}$,

которая по всякой программе n строит программу $s(n)$, выдающую на входе x тот же ответ, что n выдает на входе $x+1$, то есть, $U_{s(n)}(x) = U_n(x+1)$. Для этого обозначим $V(n,x) = U(n,x+1)$. Вновь функция V вычислима; поскольку U главная, существует такая всюду определенная вычислимая функция $s\colon \mathbb{N} \to \mathbb{N}$, что $U_{s(n)}(x) = V_n(x) = U_n(x+1)$ для всех n и всех x.

Разберем более сложный пример.

Пример 1.12. Пусть $U\colon \mathbb{N}\times\mathbb{N} \to \mathbb{N}$ — главная универсальная функция. Тогда существует всюду определенная вычислимая функция $c\colon \mathbb{N}\times\mathbb{N} \to \mathbb{N}$, которая по всякой паре программ p и q строит программу $c(p,q)$, вычисляющую композицию программ p и q. Другими словами, $U(c(p,q),x) = U(q,U(p,x))$. Действительно, рассмотрим функцию[2] $V([p,q],x) = U(q,U(p,x))$. Это вычислимая функция (как композиция вычислимых), а значит, в силу главности U существует такая всюду определенная вычислимая s, что $U(s(n),x) = V(n,x)$. В итоге получаем $U(q,U(p,x)) = V([p,q],x) = U(s([p,q]),x)$. Определяя $c(p,q) = s[p,q]$, мы получаем требуемое равенство.

Мы посмотрели, как можно использовать свойство главности, но нам еще нужно разобраться в том, существуют ли главные универсальные функции.

Теорема 1.14. *Существует главная универсальная функция* $U\colon \mathbb{N}\times\mathbb{N} \to \mathbb{N}$.

Сначала мы дадим неформальное пояснение, почему существует главная универсальная функция. Но оказывается, что в отличие от существования универсальной функции, здесь мы уже можем дать формальное доказательство: существование главной универсальной функции можно вывести из существования универсальной функции.

Неформальное доказательство. Достаточно вспомнить наше объяснение существования универсальной вычислимой функции и убедиться, что оно дает главную универсальную функцию. Напомним, что мы можем рассматривать U как язык программирования, первый аргумент функции $U(n,x)$ как программу, а второй аргумент как вход для этой программы. И тогда алгоритм для U запускает данную ему программу на данном ему входе.

Теперь нам дана функция V от двух аргументов. Поскольку она вычислима, то для нее есть программа в нашем языке программирования. Нам требуется построить функцию s, которая по данному n будет выдавать программу, вычисляющую функцию V_n. И теперь описать такую s совсем просто: s получает на вход n, берет программу для V (как для функции двух аргументов) и подставляет в тексте этой программы вместо запроса первого входа конкретное число n, полученное на

[2]Мы здесь используем не вполне аккуратную запись. Раз уж мы определяем функцию V, следовало бы написать $V(n,x) = U(r(n),U(l(n),x))$. Но поскольку $[\cdot,\cdot]$ — биекция, то по существу выбор записи — это вопрос обозначений (по сути мы ввели обозначения $p = l(n)$ и $q = r(n)$ и сделали замену переменных). Мы будем использовать такие обозначения, в которых лучше видно, что происходит.

вход. В результате получается программа, входом для которой является только x. Эту программу s и выдает на выход. Напомним еще раз, что здесь мы довольно свободно переходит от программ к числам и обратно, пользуясь тем, что мы уже обсуждали, как устанавливается взаимно однозначное соответствие между этими объектами. □

Заметим, что в объяснении выше нам впервые потребовалось в наших неформальных объяснениях как-то видоизменять текст программы в нашем языке программирования. Это как раз отражает то, что свойство главности соответствует возможности (вычислимо) проделывать простые преобразования с текстами программ.

Доказательство теоремы 1.14. Доказательство будет состоять из двух частей. Сначала мы докажем, что существует универсальная вычислимая функция $T\colon \mathbb{N}\times\mathbb{N}\times\mathbb{N}\to\mathbb{N}$ для класса вычислимых функций от двух переменных. Это значит, что T сама вычислима, и для всякой функции $V\colon \mathbb{N}\times\mathbb{N}\to\mathbb{N}$ существует такое $n\in\mathbb{N}$, что $T(n,x,y)=V(x,y)$ для всех x,y.

Для этого рассмотрим универсальную функцию $U\colon \mathbb{N}\times\mathbb{N}\to\mathbb{N}$ и определим $T(n,x,y)=U(n,[x,y])$. Функция T вычислима как композиция вычислимых функций. Пусть теперь $V\colon \mathbb{N}\times\mathbb{N}\to\mathbb{N}$ — произвольная вычислимая функция. Рассмотрим функцию $f([x,y])=V(x,y)$ (или, что то же самое, $f(z)=V(l(z),r(z))$). Функция f является вычислимой функцией одной переменной. Поскольку U является универсальной, существует такое n, что $U(n,z)=f(z)$ для всякого z. Суммарно получаем

$$V(x,y)=f([x,y])=U(n,[x,y])=T(n,x,y),$$

что и требовалось. Следовательно, T действительно является универсальной вычислимой функций для класса вычислимых функций двух переменных.

Теперь мы готовы построить главную универсальную вычислимую функцию. Положим $U'([n,x],y)=T(n,x,y)$ (или что то же самое, $U'(z,y)=T(l(z),r(z),y)$). Функция U' вычислима как композиция вычислимых. Покажем, что для U' выполняется свойство главности. Пусть $V\colon \mathbb{N}\times\mathbb{N}\to\mathbb{N}$ — произвольная вычислимая функция. В силу универсальности T существует такое k, что $T(k,n,x)=V(n,x)$. В итоге получаем $V(n,x)=T(k,n,x)=U'([k,n],x)$. Полагая $s(n)=[k,n]$, получаем то, что и требовалось.

Осталось лишь заметить, что свойство универсальности легко вытекает из свойства главности. Действительно, для произвольной вычислимой функции $f\colon \mathbb{N}\to\mathbb{N}$ рассмотрим вычислимую функцию $V(n,x)=f(x)$. В силу главности U' получаем, что существует всюду определенная вычислимая s, такая что $U(s(n),x)=V(n,x)=f(x)$. В частности, $U'_{s(0)}=f$. □

1.6 Теорема Райса – Успенского

Теперь мы можем воспользоваться главными универсальными функциями для получения новых отрицательных результатов.

Теорема 1.15. *Пусть $U\colon \mathbb{N}\times\mathbb{N}\to\mathbb{N}$ — главная универсальная функция. Тогда множество*

$$N_\varnothing = \{n \mid U_n \text{ является нигде не определенной функцией}\}$$

неразрешимо.

На языке нашей программистской интуиции, эта теорема означает неразрешимость следующей задачи: по тексту программы определить, выдает ли эта программа выход хоть на каком-нибудь входе.

Доказательство. Пусть $K\subseteq\mathbb{N}$ — перечислимое неразрешимое множество. Такое множество существует по теореме 1.12. Рассмотрим функцию

$$V(n,x) = \begin{cases} 0, & \text{если } n\in K,\\ \text{не определена}, & \text{если } n\notin K.\end{cases}$$

Функция V вычислима. Действительно, получив на вход n и x, можно запустить алгоритм, перечисляющий K. Когда этот алгоритм выдает какое-то число на выход, мы сравниваем его с n. Если числа совпадают, выдаем 0 и заканчиваем работу. Иначе продолжаем перечисление K. Если n лежит в K, то на каком-то шаге оно будет выдано алгоритмом, перечисляющим K, и мы выдадим 0. Если же n не лежит в K, то мы не выдадим никакого ответа, что и требуется.

Поскольку V вычислима, а U — главная, то существует такая вычислимая всюду определённая функция $s\colon\mathbb{N}\to\mathbb{N}$, что $U_{s(n)} = V_n$.

Заметим теперь, что функция V_n (а значит, и $U_{s(n)}$) тождественно равна 0, если $n\in K$, и является нигде не определенной, если $n\notin K$. Таким образом, $s(n)\notin N_\varnothing$ тогда и только тогда, когда $n\in K$.

Если бы множество $N_\varnothing$ было разрешимо, то мы бы могли построить алгоритм, разрешающий K: по данному n сначала вычисляем $s(n)$ (вычисление s всегда заканчивается, поскольку это всюду определенная вычислимая функция), а затем запускаем алгоритм, проверяющий, принадлежит ли $s(n)$ множеству $N_\varnothing$. Если алгоритм выдает 0, то мы выдаем 1, и наоборот.

Но поскольку множество K неразрешимо, то мы пришли к противоречию, а значит неразрешимо и множество $N_\varnothing$. □

Доказанное только что утверждение можно существенно обобщить. Обозначим через $\mathcal{F}$ класс всех вычислимых функций на натуральных числах. *Свойством вычислимых функций* будем называть произвольное подмножество $\mathcal{A}\subseteq\mathcal{F}$. Свойство вычислимых функций $\mathcal{A}$ называется нетривиальным, если $\mathcal{A}\neq\varnothing$ и $\mathcal{A}\neq\mathcal{F}$.

Теорема 1.16 (Теорема Райса – Успенского)**.** *Пусть $U\colon\mathbb{N}\times\mathbb{N}\to\mathbb{N}$ — главная универсальная функция. Пусть $\mathcal{A}$ — нетривиальное свойство вычислимых функций. Тогда множество*

$$N = \{n \mid U_n\in\mathcal{A}\}$$

не разрешимо.

В нашей программистской интуиции эта теорема означает, что по данному тексту программы на каком-либо языке программирования нельзя алгоритмически распознать никакое нетривиальное свойство функции, вычисляемой данной программой. Заметим, что когда мы ограничились нетривиальными свойствами, мы исключили действительно только самые тривиальные случаи: когда все функции обладают свойством, и когда ни одна функция не обладает этим свойством.

Доказательство. Рассмотрим нигде не определенную функцию $f_\varnothing$ на натуральных числах. Эта функция либо лежит в $\mathcal{A}$, либо лежит в $\mathcal{F}\setminus\mathcal{A}$. Пусть для определенности она лежит в $\mathcal{A}$. Поскольку $\mathcal{A}$ — нетривиальное свойство, существует функция $g \in \mathcal{F} \setminus \mathcal{A}$.

Пусть $K \subseteq \mathbb{N}$ перечислимое неразрешимое множество. Такое множество существует по теореме 1.12. Рассмотрим функцию

$$V(n,x) = \begin{cases} g(x), & \text{если } n \in K, \\ \text{не определена}, & \text{если } n \notin K. \end{cases}$$

Докажем, что функция V вычислима. Действительно, получив на вход n и x, можно запустить алгоритм, перечисляющий K. Когда этот алгоритм выдает какое-то число на выход, мы сравниваем это число с n. Если числа совпадают, мы запускаем алгоритм для вычисления g на x, и если он выдает какой-то результат, подаем его на выход. Если же числа не совпадают, продолжаем перечисление K. Если n лежит в K, то на каком-то шаге оно будет выдано алгоритмом, перечисляющим K, и мы дальше будем вычислять функцию g на входе x. Если же n не лежит в K, то мы не выдадим никакого ответа, что и требуется.

Поскольку V вычислима, а U главная, то существует такая вычислимая всюду определённая функция $s\colon \mathbb{N} \to \mathbb{N}$, что $U_{s(n)} = V_n$.

Заметим теперь, что функция V_n (а значит, и $U_{s(n)}$) совпадает с g, если $n \in K$, и совпадает с $f_\varnothing$, если $n \notin K$. Таким образом, $s(n) \notin N$ тогда и только тогда, когда $n \in K$.

Если бы множество N было разрешимо, то мы бы могли построить алгоритм, разрешающий K: по данному n сначала вычисляем $s(n)$, а затем запускаем алгоритм, проверяющий, принадлежит ли $s(n)$ множеству N. Если алгоритм выдает 0, то мы выдаем 1 и наоборот.

Но поскольку множество K неразрешимо, то мы пришли к противоречию, а значит, неразрешимо и множество N. □

Замечание 1.5. Важно не запутаться в формулировке теоремы Райса – Успенского. Стоит обратить внимание на то, что в формулировке теоремы обсуждаются два разных объекта. С одной стороны, мы рассматриваем программы в каком-то языке программирования. С другой стороны, мы рассматриваем функции, которые этими программами вычисляются. И это не одно и то же. Например, несколько разных программ вполне могут вычислять одну и ту же функцию. И вся соль теоремы состоит в том, что по программе трудно понять, что за функцию она вычисляет.

Теорема утверждает, что никакое свойство *функции* алгоритмически не распознается по вычисляющей ее *программе*.

Чуть раньше мы определили понятие главной универсальной функции, объяснили, что это естественное свойство и даже доказали, что главные универсальные функции существуют. Но, может, быть ситуация обратная и все универсальные функции на самом деле главные? Бывают ли вообще не главные универсальные функции? Оказывается бывают, и мы можем доказать это с помощью теоремы Райса – Успенского.

Теорема 1.17. *Существует неглавная универсальная функция для класса вычислимых функций одной переменной.*

Доказательство. Рассмотрим какую-нибудь универсальную функцию $U\colon \mathbb{N}\times\mathbb{N}\to\mathbb{N}$.

Введем обозначение

$$D = \{n \mid U_n \text{ определена хотя бы при одном значении аргумента}\}.$$

По теореме Райса-Успенского множество D не является разрешимым. Но при этом оно перечислимо. Действительно, покажем, что полухарактеристическая функция χ_D^* вычислима. При данном на вход n запустим вычисление функции U_n на всех возможных входах параллельно по шагам. Это можно сделать так же, как в доказательстве теоремы 1.7: вычисление проходит итерацией по $k = 1, 2, 3, \ldots$. На итерации k мы вычисляем U_n на входе $k-1$ на один шаг, на входе $k-2$ на два шага, и так далее, на входе 0 на k шагов. Если на какой-то итерации хотя бы на одном входе алгоритм, вычисляющий U_n, выдал что-то на выход, то наш алгоритм выдает на выход 0 и заканчивает работу. Если функция U_n определена хотя бы на одном значении аргумента, то на какой-то итерации мы закончим вычисление на этом входе и выдадим 0. Если же U_n является нигде не определенной функцией, то мы никогда не выдадим ничего на выход.

Таким образом, множество D перечислимо. Зафиксируем некоторый алгоритм, перечисляющий D, и пусть $d_1, d_2, d_3 \ldots$ — последовательность элементов D, которую выдает этот алгоритм.

Определим функцию

$$U'(n,x) = \begin{cases} \text{не определено}, & \text{если } n = 0, \\ U(d_n, x), & \text{если } n > 0. \end{cases}$$

Во-первых, U' вычислима. Действительно, алгоритм, вычисляющий U', получает на вход n и x, и если $n = 0$, сразу зацикливается и не выдает ничего на выход. Если же $n > 0$, то наш алгоритм сначала запускает алгоритм, перечисляющий множество D до тех пор, пока не будет выдан n-ый элемент d_n в выходе этого алгоритма. Затем наш алгоритм запускает алгоритм для вычисления U на входе d_n и x.

Далее, U' является универсальной. Действительно, пусть $f\colon \mathbb{N}\to\mathbb{N}$ — произвольная вычислимая функция. Если f является нигде не определенной, то $f = U'_0$.

Если же f определена хотя бы в одной точке, то рассмотрим такое k, что $f = U_k$ (такое k существует в силу универсальности U). Заметим, что k входит в D, так что существует n, для которого $k = d_n$. В итоге получаем, что $f = U_k = U_{d_n} = U'_n$. Таким образом, в любом случае для f существует n, при котором $f = U'_n$.

Наконец, осталось показать, что функция U' не является главной. Для этого рассмотрим множество

$$N'_\varnothing = \{n \mid U'_n \text{ является нигде не определенной функцией}\}.$$

По построению функции U' мы знаем, что $N'_\varnothing = \{0\}$. Это множество конечно, а значит по лемме 1.9 оно разрешимо. Но по теореме 1.15 для главных универсальных функций это множество неразрешимо. Следовательно, построенная универсальная функция не является главной. □

В этой теореме мы построили универсальную функцию, которая не удовлетворяет свойству главности, про которое мы объясняли, что оно выполняется для "разумных" языков программирования. Таким образом, мы построили "неразумный" язык программирования. И действительно, получившийся язык программирования довольно странный. Например, почти любая программа в нем будет осмысленной. Если набрать случайный набор символов и запустить его как программу, то на каком-то входе даже будет выдан какой-то результат. Пользоваться таким языком, однако, неудобно.

1.7 Теорема о неподвижной точке

Теперь с помощью свойства главности универсальных функций мы докажем второй важный результат. Из него мы легко получим решение в общем виде такой классической задачи по программированию: написать программу, печатающую свой собственный текст.

Теорема 1.18. *Пусть $U\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ — главная универсальная функция. Тогда для всякой всюду определенной вычислимой функции $h\colon \mathbb{N} \to \mathbb{N}$ существует $n \in \mathbb{N}$, при котором $U_n = U_{h(n)}$.*

В нашей программистской интуиции эта теорема утверждает, что для любого вычислимого преобразования программ в каком-либо языке программирования существует программа, которая до и после преобразования работает одинаково.

Доказательство. В доказательстве этой теоремы мы используем по существу ту же конструкцию, что и в доказательстве теоремы 1.11 о том, что существует вычислимая функция без всюду определенного вычислимого продолжения. Но нам потребуется немного обобщить эту конструкцию.

Сформулируем обобщенный вариант конструкции в виде леммы.

Лемма 1.19. *Пусть $\equiv$ — некоторое отношение эквивалентности на $\mathbb{N}$. Тогда следующие два утверждения не могут выполняться одновременно*:

1. *Для любой вычислимой $f\colon \mathbb{N} \to \mathbb{N}$ существует всюду определенное вычислимое $\equiv$-продолжение (то есть такая функция $g\colon \mathbb{N} \to \mathbb{N}$, что для всякого x из области определения f верно $f(x) \equiv g(x)$).*

2. *Существует всюду определенная вычислимая функция $h\colon \mathbb{N} \to \mathbb{N}$, не имеющая $\equiv$-неподвижной точки (это значит, что $h(n) \not\equiv n$ для всех n).*

Доказательство леммы. Доказательство по существу повторяет доказательство теоремы 1.11 (и здесь нам не требуется главность U). Предположим, что для некоторого отношения эквивалентности $\equiv$ оба утверждения верны. Рассмотрим функцию $d(n) = U(n, n)$. По утверждению 1 у функции d есть всюду определенное вычислимое $\equiv$-продолжение f, то есть $d(n) \equiv f(n)$ для всех n из области определения d. Рассмотрим функцию $h(f(n))$. Это всюду определенная вычислимая функция. Поскольку U является универсальной, то существует такое k, что $h(f(n)) = U(k, n)$ для всякого n. Рассмотрим теперь $n = k$. Тогда $h(f(k)) = U(k, k) = d(k)$. Если $d(k)$ не определено, то мы сразу получаем противоречие, поскольку $h(f(k))$ определено (композиция двух всюду определенных функций). Если же $d(k)$ определено, то $d(k) \equiv f(k) \not\equiv h(f(k))$, что также противоречит предыдущему равенству. □

Если в этой лемме в качестве отношения эквивалентности взять обычное равенство, то второе утверждение будет верно: достаточно взять функцию $h(n) = n + 1$. Значит первое утверждение неверно, и мы передоказали теорему 1.11.

Рассмотрим теперь следующее отношение эквивалентности: $k \equiv n$ тогда и только тогда, когда $U_k = U_n$. Легко убедиться, что это отношение действительно является отношением эквивалентности.

Для этого отношения, напротив, верно первое утверждение. Действительно, пусть $f\colon \mathbb{N} \to \mathbb{N}$ — вычислимая функция. Рассмотрим функцию $V(n, x) = U(f(n), x)$. Функция V вычислима как композиция двух вычислимых функций. Следовательно, в силу главности U существует такая всюду определенная вычислимая функция $s\colon \mathbb{N} \to \mathbb{N}$, что $U_{s(n)} = V_n$. Поскольку для всякого n из области определения f верно $V_n = U_{f(n)}$, мы получаем, что s является всюду определенным вычислимым $\equiv$-продолжением f.

Раз первое утверждение верно для нашего отношения эквивалентности, то второе должно быть ложно. Следовательно, для всякой всюду определенной $h\colon \mathbb{N} \to \mathbb{N}$ существует n, такое что $U_{h(n)} = U_n$. А это как раз то, что требовалось доказать. □

Полезно проследить, как выглядит неподвижная точка, существование которой мы только что доказали. Для этого нужно развернуть доказательство теоремы в другом порядке. Пусть нам дана всюду определенная вычислимая функция $h\colon \mathbb{N} \to \mathbb{N}$. Мы начинаем с диагональной функции d и рассматриваем функцию $V(n, x) = U(d(n), x) = U(U(n, n), x)$ (все действия у нас проходят внутри первого аргумента универсальной функции, так уж у нас определено наше отношение эквивалентности, и такая уж неподвижная точка нас интересует). В силу главности U существует $s\colon \mathbb{N} \to \mathbb{N}$, для которой

$$U(s(n), x) = V(n, x) = U(d(n), x) = U(U(n, n), x).$$

Затем мы применяем к s данную нам функцию h и, таким образом, рассматриваем функцию $U(h(s(n)), x)$. Дальше мы пользуемся универсальностью U и говорим, что существует k, для которого $U(k, n) = h(s(n))$ при всех n. Таким образом,

$$U(h(s(n)), x) = U(U(k, n), x)$$

при этом k и при любых n и x. Осталось лишь положить $n = k$ и объединить две цепочки равенств. Получаем

$$U(h(s(k)), x) = U(U(k, k), x) = U(d(k), x) = V(k, x) = U(s(k), x).$$

Сравнивая левую и правую часть цепочки, получаем $U_{h(s(k))} = U_{s(k)}$. И $s(k)$ является неподвижной точкой функции h. Полезно также проследить, что из себя представляют s и k. Оказывается, что s вовсе не зависит от h. Это функция, внутри которой спрятана диагональ нашей универсальной функции. А вот k уже связана с h — это номер программы, вычисляющей функцию $h \circ s$.

У теоремы о неподвижной точке есть альтернативная форма, пользоваться которой иногда бывает удобнее.

Следствие 1.20. *Пусть $U\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ — главная универсальная функция. Тогда для всякой вычислимой функции $V\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ существует n, при котором $U_n = V_n$.*

В нашей программистской интерпретации это следствие утверждает, что если U обладает свойством главности, то для любого другого языка программирования V (не обязательно даже универсального) есть программа, которая в обоих языках работает одинаково.

Доказательство. В силу главности U существует всюду определенная такая вычислимая функция $s\colon \mathbb{N} \to \mathbb{N}$, что $V_n = U_{s(n)}$ для любого n. По теореме о неподвижной точке существует n, при котором $U_{s(n)} = U_n$. В итоге получаем $V_n = U_{s(n)} = U_n$, что и требовалось. □

Наконец, докажем, что существует программа, печатающая свой собственный текст.

Теорема 1.21. *Пусть $U\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ — главная универсальная функция. Тогда существует такое $n \in \mathbb{N}$, что $U(n, x) = n$ для всякого x.*

Доказательство. Рассмотрим вычислимую функцию $V(n, x) = n$. По предыдущему следствию существует такое n, что $U(n, x) = V(n, x) = n$ для всех x, что и требовалось доказать. □