Лекция 17

Примеры неразрешимых задач

17.1 Почему важна проблема остановки?

Мы доказали, что проблема остановки неразрешима — и что, собственно? Кого вообще она так уж волнует (кроме каких-нибудь разработчиков компиляторов, которым полезно было бы знать, какие циклы завершаются, а какие нет)? Допустим, она была бы разрешима — ну и что тогда? Или наоборот: мы теперь знаем, что она неразрешима — и что из этого следует? В этой лекции мы попытаемся ответить на эти вопросы.

Начнём с простого: что было бы хорошего, если бы проблема остановки была разрешима, то есть был бы алгоритм, который по данной программе проверяет, останавливается она или нет. В этом случае можно было бы его использовать, чтобы отвечать на разные математические вопросы. Скажем, мы хотим узнать, верна «великая теорема Ферма» или нет (сейчас мы на самом деле знаем, что верна, но временно забудем про это). Эта теорема утверждает, что уравнение $x^n + y^n = z^n$ не имеет решений в целых числах с $x, y, z > 0$ и $n > 2$. Напишем несложную программу поиска этих решений перебором, которая останавливается, найдя такое решение. Эта программа, очевидно, останавливается тогда и только тогда, когда теорема Ферма неверна. Соответственно, умея решать проблему остановки, мы могли бы это узнать.

Можно взять в качестве примера другую известную проблему, «гипотезу Гольдбаха»: она утверждает, что каждое чётное число можно представить в виде суммы двух простых чисел.[1] С ней всё то же самое: напишем программу, которая ищет чётное число, не разлагающееся в сумму двух простых (для данного числа это можно проверить конечным числом проб), и останавливается, его найдя.

Ещё одна нерешённая проблема (Collatz conjecture, она известна также как $3n+1$-гипотеза, и под многими другими именами), состоит в следующем. Возьмём какое-

[1] Отсюда следует, что каждое нечётное число, большее 7, можно представить в виде суммы трёх простых чисел; это более простое утверждение было доказано И. М. Виноградовым для всех достаточно больших нечётных чисел, но граница была астрономически велика — а недавно Х. Хельфгот опубликовал доказательство того же факта, но с меньшей границей, до которой это свойство уже проверено.

то целое число $n > 1$. Если оно чётно, разделим его на 2. Если оно нечётно, то заменим его на $3n + 1$. Будем повторять этот шаг многократно, пока не получится 1. Например, если мы начнём с 11, то получатся числа

$$11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1$$

($3 \cdot 11 + 1 = 34$, $34/2 = 17$, $3 \cdot 17 + 1 = 52$ и так далее). Для других чисел это может быть дольше (попробуйте 27), но пока что всегда рано или поздно приходили к единице. (Хотя в принципе могло бы и в бесконечность уйти, и зациклиться, не дойдя до единицы.)

Если бы проблема остановки была разрешима некоторым алгоритмом, то этот алгоритм можно было бы использовать для того, чтобы выяснить, верна ли гипотеза Коллатца. Правда, это уже сложнее, чем раньше: мы не можем искать контрпример к гипотезе перебором, потому что не знаем способа распознать контрпример. Допустим, мы подозреваем, что какое-то N является контрпримером: мы сделали много шагов и так и не пришли к единице — но вдруг надо сделать ещё несколько шагов, и единица появится? В отличие от случаев Ферма и Гольдбаха, уже проверка контрпримера здесь дело непростое. Но можно воспользоваться таким трюком: для каждого N можно написать программу, которая останавливается, когда из N получается единица, и применить к ней алгоритм (гипотетический), решающий проблему остановки. Вместе получится программа, которая по N проверяет, является ли N контрпримером. Используя её как подпрограмму, мы можем написать программу поиска контрпримера перебором, и уже к ней ещё раз применить алгоритм, решающий проблему остановки.

Задача 17.1. «Гипотеза близнецов» утверждает, что простых чисел-близнецов (два простых числа, отличающихся ровно на 2) бесконечно много. Как можно было бы использовать алгоритм, решающий проблему остановки, чтобы выяснить, так это или нет?

Последний пример. *Диофантовым уравнением* называется уравнение в целых числах, в котором используются лишь операции сложения и умножения. (Другими словами, мы ищем целочисленные точки, в которых многочлен от нескольких переменных с целыми коэффициентами обращается в нуль.) Например, уравнение $x^2 + y^2 = z^2$, решениями которого являются стороны прямоугольных треугольников с целыми сторонами, является диофантовым. (Для его решений есть общая формула: как доказал ещё Евклид, все такие тройки имеют вид $m^2 - n^2, 2mn, m^2 + n^2$ с целыми m и n, с точностью до перестановки m и n и умножения всех трёх чисел на одно и то же целое число.) Другой пример диофантова уравнения: $x^2 - 5y^2 = 1$. Перебором легко найти одно из его решений: $x = 9$, $y = 4$. На самом деле у него бесконечно много решений, и это умели доказывать уже в древней Индии, а Лагранж доказал, что можно заменить 5 на любое натуральное число, не являющееся точным квадратом, и всегда будет бесконечно много решений.

Задача 17.2. Уравнение $x^2 - 5y^2 = 1$ можно переписать в виде $(x - \sqrt{5}y)(x + \sqrt{5}y) = 1$. Покажите, что произведение двух чисел вида $(x - \sqrt{5}y)$ (и тем самым степень любого

такого числа) имеет тот же вид, и используйте этот факт, чтобы найти бесконечно много (ну или хотя бы несколько) решений уравнения $x^2 - 5y^2 = 1$.

Вообще для разных классов диофантовых уравнений имеется развитая теория — для каждого своя. В начале XX века Гильберт, перечисляя важные на его взгляд нерешённые проблемы в математике, предлагал найти общий метод, с помощью которого по любому диофантовому уравнению можно было бы узнать с помощью конечного числа действий (по нашему: с помощью алгоритма), имеет ли оно решение. Поскольку поиск такого решения можно оформить в виде программы, эта задача сводится к проблеме остановки: алгоритм, решающий проблему остановки, дал бы решение этой проблемы Гильберта (в его списке она была под номером 10, так что её называют «десятой проблемой Гильберта»). Но оказалось, что проблема эта неразрешима: алгоритма, о котором мечтал Гильберт, не существует. Последний шаг в доказательстве этой неразрешимости сделал Ю. Матиясевич (продолжая работы Х. Патнэма и Дж. Робинсон) в 1970 году. И доказательство это использует проблему остановки: доказано, что эта проблема сводится к диофантовым уравнениям в том смысле, что имея программу, мы можем написать диофантово уравнение, которое имеет решение тогда и только тогда, когда программа останавливается. То есть сводимость тут в обе стороны, и если бы Гильберт построил алгоритм, о котором мечтал, то его можно было бы использовать для решения проблемы остановки (а так не бывает).

Доказательство Матиясевича сложное, и мы его приводить не будем. Но мы попытаемся проиллюстрировать, как подобные вещи доказываются, на более простом примере. Этот пример придумал Конвей (известный широкой публике по игре «Жизнь»), и назвал его «FRACTRAN» (намекая на популярный тогда язык программирования FORTRAN и имея в виду английское слово fractions, означающее дроби).

17.2 FRACTRAN

Роль программы в FRACTRANe играет последовательность дробей. Пусть на доске записаны (слева направо) некоторые дроби (=положительные рациональные числа)

$$r_1, r_2, \ldots, r_N.$$

Пусть отдельно от них в другом месте доски написано некоторое целое положительное число m. Мы занимаемся таким странным делом: стараемся умножить m на одну из этих дробей, при этом хотим, чтобы оно осталось целым. Это может быть невозможным (все N произведений не целые), тогда на этом всё кончается. Это может быть возможным, и даже несколькими способами — тогда мы берём первый (слева) подходящий множитель. С полученным числом (произведением) мы делаем то же самое: ищем множитель r_i, на который его можно умножить, чтобы оно осталось целым, и так далее.

Например, если написаны две дроби $\frac{7}{5}$ и $\frac{5}{7}$, а начальное число равно 10, то будут чередоваться числа 10 и 14 (и даже не будет неоднозначности в выборе дроби-множителя). Если же дописать в начале, скажем, дробь $\frac{11}{20}$, то для числа 10 всё будет по-прежнему, а для числа 20 вместо цикла $20 \to 28 \to 20 \to 28 \to \ldots$, как раньше, мы сразу же получим 11 и на этом всё кончится, подходящих множителей уже не будет.

Имея последовательность $r_1, \ldots, r_N$ и начальное число m, можно спросить: остановится ли этот процесс или нет? сойдётся ли наш FRACTRAN-пасьянс[2] (дойдя до ситуации, когда больше хода сделать нельзя) или нет?

Теорема 17.1. *Этот вопрос алгоритмически неразрешим: не существует алгоритма, который по данным $r_1, \ldots, r_n, m$ отвечает на вопрос, остановится ли описанный процесс.*

Мы сейчас воспроизведём доказательство этой теоремы Конвея. Но до этого давайте обсудим, в чём её интерес. Её можно рассматривать как пример алгоритмически неразрешимой проблемы в математике: в отличие от проблемы остановки, где надо сначала рассказывать про язык программирования, её можно объяснить всякому, кто знает элементарную арифметику (целые числа, дроби, умножение) — таких людей много, и к тому же готов нас слушать (это уже сложнее).

Собственно говоря, этот пасьянс под названием FRAKTRAN не так далеко ушёл от игры в $3n + 1$: там мы тоже умножали на $1/2$, если это оставляло число целым, а если нет, то умножали его на 3 (и ещё прибавляли единицу, что теперь не предусмотрено).

Пожалуй, если говорить о технической сложности доказательства неразрешимости конкретного математического вопроса, пример Конвея — самый простой пример такого рода, поэтому мы его и разбираем.

17.3 Язык программирования

Для начала мы опишем некоторый примитивный язык программирования. Программы на этом языке используют лишь неотрицательные целые переменные, но это настоящие натуральные числа, а не до 2^{32} или 2^{64}, как обычно бывает — никаких ограничений на их величину нет.

Программа представляет собой последовательность команд, а команды бывают всего двух видов. Во-первых, команда может увеличивать значение какой-то из переменных на 1, то есть иметь вид

[2] В карточном пасьянсе обычно нужно перекладывать карты по определенному правилу, пока он не «сойдётся», то есть не получится ситуация определённого вида, когда уже больше перекладывать ничего не надо. Поэтому мы и называем FRACTRAN пасьянсом. Правда, типичные пасьянсы — это недетерминированный процесс, в котором правила игры допускают различные ходы, и игрок имеет право выбрать любой из них. У нас же в каждый момент можно сделать только одно действие, как в известной карточной игре в «пьяницу». Недетермированные программы (где в какие-то моменты есть выбор) в computer science тоже рассматриваются: именно в таких терминах было дано первое определение знаменитого класса NP.

```
a <- a+1
```

Следуя примеру программистов, мы будем иногда сокращать эту запись до `a++`. Эта команда выполнима всегда, поскольку ограничений на величину чисел у нас нет. Вторая команда, как можно догадаться, уменьшает число на 1. Тут уже может выйти, что называется, облом — или, как говорят программисты, exception (исключение). Это бывает, когда переменная равна нулю и уменьшать её некуда. В этом случае значение переменной не меняется и происходит «обработка исключения» — мы переходим к исполнению строки программы с номером X, который указан в команде. Короче говоря, разрешены команды вида

```
a <- a-1; exception: go to X
```

Исполняя такую команду, мы смотрим, можно ли уменьшить переменную `a` на единицу, то есть положительно ли `a`. Если да, то уменьшаем, если нет, то переходим к строке с номером `X`. (Можно также считать, что `X` — это не номер строки, а поставленная нами около неё метка, так сказать, перейдя от программирования в машинных кодах к ассемблеру.)

В начале исполнения программы значения всех переменных равны нулю.

Несмотря на убогость такого языка программирования, мы сейчас покажем, как в этом языке можно реализовать все традиционные программистские конструкции (и тем самым транслировать любую программу на этот язык — правда, с сильным замедлением работы программы).

- **Безусловные переходы**. Допустим, мы хотим использовать в программе команды безусловного перехода типа

  ```
  go to X
  ```

  Дойдя до такой команды, исполнитель переходит к строке с номером (или меткой) `X`. Как это сделать? Давайте выделим специальную переменную `null`, которую никогда не будем менять, она как была вначале нулём (по нашему соглашению), так и будет оставаться. Тогда можно написать

  ```
  null <- null-1; exception: go to X
  ```

- **Условные переходы**. Пусть мы хотим реализовать условный переход вида

  ```
  if a>0 then go to X else go to Y
  ```

  (если `a` положительно, переходим к строке `X`, а если равно нулю, то к строке `Y`). Можно попробовать написать так:

  ```
  a <- a-1; exception: go to Y
  go to X
  ```

(мы уже знаем, как реализовать безусловные переходы, и можем их использовать).

Это почти правильно, плохо только, что при положительном `a` мы не только переходим к строке `Y`, но и уменьшаем `a` на единицу. Понятно, как это исправить:

```
a <- a-1; exception: go to Y
a <- a+1
go to X
```

- **Циклы** типа «while» (пока $\langle\ldots\rangle$ повторять $\langle\ldots\rangle$) теперь легко реализовать стандартным образом: вместо

```
while a>0 do
     <...>
od
```

можно написать

```
A: if a>0 go to B else go to C
B:     <...>
   go to A
C:
```

(после метки `C` следует то, что шло за циклом). Собственно, примерно это и делает компилятор любого языка высокого уровня, содержащего циклы while.

- **Присваивания**. Мы говорили об управляющих конструкциях, забыв, что в языке не предусмотрены даже присваивания `a<-b` (после которого значение переменной `b` помещается в переменную `a`) или присваивания вида `a<-0`. Последнее, впрочем, легко реализовать в виде цикла

```
while a>0 do a--; od
```

(здесь `a--` сокращает `a<-a-1`; заботиться об исключении не надо, так как условие цикла гарантирует положительность `a`). Можно попробовать примерно так же реализовать и присваивание `a<-b`, сначала поместив в `a` нуль уже известным способом, а затем написав

```
while b>0 do a++; b--; od
```

После этого действительно значение переменной `b` помещается в `a`, но само `b` становится равным нулю. Получается такое «разрушающее» присваивание. Легко сообразить, как исправить положение: надо ввести вспомогательную переменную `tmp`, и написать

```
tmp<-0
while b>0 do a++; tmp++;  b--; od
b<=tmp
```

(последняя строка обозначает разрушающее присваивание, восстанавливающее `b` из `tmp`).

- **Арифметические операции и сравнения**. Теперь понятно, как реализовать операцию сложения `a<-b+c`:

```
a<-0
while b>0 do a++; b--; od
while c>0 do a++; c--; od
```

При этом, правда, разрушатся `b` и `c`, но мы уже умеем копировать без разрушения, так что их можно сохранить заранее. Аналогично можно запрограммировать «безопасное вычитание» , при котором $a-b$ считается равным нулю, когда $a<b$. А именно, безопасно вычесть единицу можно так:

```
if a>0 then a--;
```

а вычесть произвольное `b` можно с помощью цикла:

```
while b>0 do b--; [a--;] od
```

Здесь `[a--;]` обозначает безопасное вычитание единицы. Имея безопасное вычитание, мы очевидным образом программируем переходы вида `if a>=b`, и так далее. После этого можно реализовать умножение (как сложение в цикле), проверку делимости, проверку простоты. Видно, что все управляющие конструкции стандартного языка программирования теперь у нас есть, и дальше мы можем программировать как обычно.[3]

- **Массивы произвольного размера**. Единственное, чего нам ещё не хватает — это массивов произвольного размера (не фиксированного заранее в тексте программы), в которые можно записывать и из которых можно читать числа, указав номер ячейки. В самом деле, любая программа на нашем языке содержит конечное число переменных (их можно подсчитать, читая программу). Как же нам поступить, если надо хранить массив произвольного размера?

[3]Более сложно реализовать рекурсию — но это и не удивительно, первые компиляторы тоже этого не умели. Как это делается, описывается в учебниках по реализации языков программирования — для этого нужен стек произвольного размера, и о массивах произвольного размера речь пойдёт дальше. Поскольку бывают языки программирования, в которых нет рекурсии и которые тем не менее универсальны (можно запрограммировать всё, что можно запрограммировать на других языках), то рекурсия для доказательства теоремы Конвея не нужна.

Вспомним, что множество всех конечных последовательностей натуральных чисел счётно: скажем, последовательность $x_1, \ldots, x_n$ можно закодировать числом

$$2^{x_1}3^{x_2}5^{x_3}\ldots,$$

и разным последовательностям будут соответствовать разные числа в силу однозначности разложения на простые множители. Хотя нет, минуточку — тут мы погорячились: если к последовательности x_i в конце дописать несколько нулей, то ничего не изменится. Более точно надо сказать, что мы кодируем не конечные последовательности, а бесконечные последовательности натуральных чисел, в которых все члены, начиная с некоторого места равны нулю. (Такие бесконечные последовательности иногда называют *финитными*.) Используя это кодирование, можно считать, что одна переменная с натуральными значениями представляет собой бесконечный виртуальный массив натуральных чисел, в котором почти все члены равны нулю. (Нулевой массив, где все числа равны нулю, кодируется при этом как 1.) Остаётся реализовать операции чтения и записи в i-ю ячейку этого массива. Для начала мы пишем код, который по i находит i-е простое число p_i (обычным образом, при этом используется лишь конечное число переменных), затем можем увеличивать значение в i-ой ячейке виртуального массива, умножая на p_i, и уменьшать, деля на p_i, а после этого можем и копировать число из этой виртуальной ячейки с помощью уменьшения в цикле, как мы это делали выше. Мы не будем сейчас вдаваться в подробности — надеясь, что все, кто дочитали до этого места, легко могут это сделать сами (и эти подробности всё равно читать не будут).

17.4 Сведение проблемы остановки: от программ к пасьянсам

Теперь вспомним, для чего нам нужен был такой простой язык программирования, а именно, покажем, что по всякой программе π на этом языке можно написать последовательность дробей и начальное целое число таким образом, что FRACTRAN-пасьянс остановится тогда и только тогда, когда останавливается программа π. На самом деле там не только вопросы об остановке равносильны, а просто каждый шаг исполнения программы соответствует одному действию по правилам FRACTRAN.

Вот как эта последовательность пишется. Пусть в программе π у нас используется n переменных $x_1, \ldots, x_n$. Текущие значения этих переменных будут закодированы в текущем целом числе FRACTRAN-пасьянса. А именно, в его разложении на множители i-е простое число p_i будет входить в степени x_i. Тогда операция увеличения x_i на единицу соответствует в пасьянсе умножению на p_i, а уменьшение x_i на единицу соответствует делению на p_i. Заметим, что логика перехода по исключению как раз соответствует правилам пасьянса: если мы пытаемся уменьшить x_i на 1, а оно уже равно нулю, то число не делится на p_i, и соответствующая операция пасьянса неприменима, и надо искать дальше (следующую применимую).

Как знают конструкторы микропроцессоров, помимо собственно переменных, нужно где-то хранить program counter — информацию о том, какая строка про-

граммы сейчас выполняется. Для этого мы тоже будем использовать простые числа, но не те, которые зарезервированы для переменных, а другие. Пронумеруем строки программы простыми числами (не занятыми на обслуживание переменных) $q_1, \ldots, q_m$. Договоримся, что если текущая строка программы помечена числом q, то текущее число пасьянса содержит простой множитель q (в первой степени). Таким образом, в каждый момент ровно одно из чисел q_i входит в разложение текущего числа пасьянса (в первой степени).

Всё это пока описания, как всё хорошо будет — но надо объяснить, как мы этого добъёмся. Допустим, в программе есть строка

q : `x++`
q' : ...

Здесь q — простое число, выбранное номером этой строки, а q' — другое простое число, выбранное номером следующей строки. Эта строка выполняется в тот момент, когда текущей является строка номер q (когда число в пасьянсе делится на q), в результате переменная x увеличивается на 1 (число умножается на p_i, если x кодируется как показатель степени при p_i), а текущей строкой становится строка с номером q'. Все эти действия будут обеспечены, если в список дробей в пасьянсе включить дробь

$$\frac{p_i q'}{q},$$

и позаботиться о том, чтобы не сработали более ранние дроби.

Вычитание единицы чуть сложнее, поскольку надо заботиться об исключениях. Для строки

q : `x--; if exception, go to` q''
q' : ...

надо добавить в список дробей группу из двух дробей

$$\frac{q'}{qp_i}, \frac{q''}{q}$$

Они могут сработать, если число в пасьянсе делится на q (то есть если текущая строка имеет метку q). Первая сработает, если к тому же число делится на p_i, то есть если значение переменной x, которая кодируется как показатель степени при p_i, больше нуля. Если же это значение равно нулю, то первая дробь не сработает, и исключение подхватит вторая: в ней никакие переменные не изменяются, а текущей становится строка с меткой q''.

Группы дробей, соответствующие разным строкам программы, могут идти в любом порядке, поскольку всё равно в каждый момент текущее число пасьянса делится только на один номер строки (мы об этом уже говорили, и это свойство сохраняется при умножении на построенные дроби). В качестве начального числа пасьянса надо взять простое число, являющееся меткой первой строки программы. После

этого исполнение правил пасьянса будет в точности соответствовать исполнению программы, и остановка пасьянса равносильна остановке программы.

Следовательно, если бы у нас был алгоритм, проверяющий, остановится ли данный пасьянс, его можно было бы использовать для проверки остановки программ на нашем мини-языке программирования. А поскольку на этот язык можно алгоритмически транслировать программы с любого другого, то и вообще проблема остановки была бы разрешима.

Что и завершает доказательство теоремы Конвея.

Задача 17.3. Покажите, что существует последовательность дробей, для которой задача «определить по данному числу m, остановится ли игра FRACTRAN, начатая с числа m», алгоритмически неразрешима.

Разница с теоремой Конвея в том, что там требовался алгоритм, который по списку дробей и начальному числу даёт ответ (и его не было), а теперь мы спрашиваем: может быть, для каждого отдельного списка существует алгоритм, который по начальному числу даёт ответ. Это, вообще говоря, более слабое свойство — но в данном случае и такого алгоритма тоже нет.

17.5 Задача достижимости на графе подстановок слов

Как вам известно, задача достижимости (можно ли из одной вершины попасть в другую, двигаясь по рёбрам) на конечном ориентированном графе разрешима. Более того, есть весьма эффективные алгоритмы её решения.

Подходящее обобщение этой задачи на бесконечные графы даёт уже алгоритмическую неразрешимость. Начнём с описания такого обобщения.

Здесь мы будем рассматривать бесконечные ориентированные графы, причём такие, что выходная и входная степень каждой вершины конечна.

Нужно договориться о способе задания графа и его вершин: для бесконечного графа это можно делать существенно различными способами.

Мы выберем такой способ задания (ор)графа. Множество вершин — это множество слов в некотором алфавите Σ. А рёбра задаются *правилами подстановки*. Каждое правило имеет вид

$$L \to R,$$

где L, R — слова в алфавите Σ. Из слова x ведёт ребро в слово y по правилу подстановки $L \to R$, если $x = uLv$, $y = uRv$.

Пример 17.4. Рассмотрим слова (последовательности букв) в русском алфавите. Тогда из слова `входная` ведёт ребро в слово `выходная` по правилу подстановки `ход` $\to$ `ыход`.

Таким образом, одно правило подстановки даёт бесконечное количество рёбер.

Мы разрешим использовать несколько правил подстановки:

$$L_1 \to R_1,$$
$$L_2 \to R_2,$$
$$\dots$$
$$L_n \to R_n,$$

и множество рёбер графа будет состоять из всех рёбер, отвечающих какому-то из этих правил.

Будем также использовать обозначение $u \underset{\mathcal{R}}{\longrightarrow} v$ для пары вершин нашего графа (т.е. слов), связанных одним из правил подстановки из множества правил подстановки $\mathcal{R} = \{L_i \to R_i\}$. Если из слова u можно попасть в слово v, двигаясь по рёбрам орграфа, будем указывать это как $u \overset{*}{\underset{\mathcal{R}}{\longrightarrow}} v$ (другими словами, это обозначение для отношения достижимости).

Задача достижимости. Задан граф на множестве слов в алфавите Σ набором правил подстановки $\mathcal{R} = \{L_i \to R_i\}$ и два слова $u, v \in \Sigma^*$. Верно ли, что $u \overset{*}{\underset{\mathcal{R}}{\longrightarrow}} v$?

В некоторых случаях решить задачу достижимости легко.

Пример 17.5. Пусть есть ровно одно правило подстановки в алфавите $\{a, b\}$:

$$ab \to ba.$$

Тогда верно, что $aaaabbaab \overset{*}{\to} baabaabaa$. (Количество букв a и b одинаково, а правило подстановки позволяет поменять местами любую пару различных букв.)

Пример 17.6. Пусть правила замены слов в алфавите $\{a, b\}$ имеют вид:

$$aba \to baabbb,$$
$$bbba \to a,$$
$$abab \to bbbb.$$

Тогда неверно, что $ababbbaa \overset{*}{\to} bbbaaba$. (Подстановки не меняют чётности количества букв a в слове.)

Однако общего «рецепта» решения задачи достижимости, т.е. алгоритма, нет.

Чтобы говорить о существовании или несуществовании алгоритма решения задачи достижимости, нужно представить условия задачи словом в конечном алфавите. Будем использовать те же правила, что и при описании машин Тьюринга. Каждый символ алфавита $i \in \Sigma = \{0, 1, \dots, n\}$ кодируется двоичным словом

$$\langle i \rangle = \underbrace{00\dots0}_{i \text{ символов}}\,1\,\underbrace{00\dots00000}_{n-i \text{ символов}},$$

слово $u = u_1 \dots u_k$, $u_i \in \Sigma$, кодируется словом

$$\langle u \rangle = \#\langle u_1 \rangle \# \langle u_2 \rangle \# \dots \# \langle u_k \rangle \#$$

в алфавите $\{0, 1, \#\}$. Правило подстановки $L \to R$ записывается как коды слов L и R, разделённые символом $\to$ и окружённые разделителями $\triangleleft$, $\triangleright$:

$$\triangleleft \langle L \rangle \to \langle R \rangle \triangleright .$$

Код набора правил $\mathcal{R} = \{L_i \to R_i\}$ — это просто запись всех правил подстановки одного за другим.

17.6 Неразрешимость задачи достижимости для графа подстановок слов

Теорема 17.2. *Не существует алгоритма, который по слову $\langle \mathcal{R} \rangle \odot \langle u \rangle \odot \langle v \rangle$, где $\mathcal{R}$ — набор правил подстановки, u, v — слова, выдавал бы результат* 1*, если $u \xrightarrow[\mathcal{R}]{*} v$, и* 0 *в противном случае.*

Доказательство. Используем неразрешимость проблемы остановки машины Тьюринга. Общий план доказательства такой: предположим, что существует алгоритм для решения задачи достижимости. Тогда мы построим алгоритм решения задачи остановки МТ следующего вида: алгоритм преобразует вход $\langle M \rangle \odot \langle x \rangle$ в описание набора правил $\mathcal{R}$ и двух слов u, v; после этого вызывает алгоритм решения задачи достижимости и выдаёт в качестве результата результат работы этого алгоритма.

Чтобы описанный алгоритм был корректным, нужно выполнение такого условия: машина M останавливается на входе x тогда и только тогда, когда $u \xrightarrow[\mathcal{R}]{*} v$.

Мы сейчас опишем такое преобразование машины и её входа в набор правил подстановки и два слова, для которого это свойство выполняется.

Итак, пусть имеется МТ $M = (A, Q, \delta, q_0, \Lambda)$ и её входное слово x. Заметим, что за один такт работы конфигурация МТ изменяется незначительно — изменения затрагивают лишь небольшую окрестность символа из множества состояний Q, который обязательно присутствует в конфигурации.

Мы хотим описать это изменение как результат применения правил подстановки. При этом удобно применять правила подстановки к словам чуть более общего вида, чем конфигурации.

В качестве алфавита для графа подстановок слов возьмём множество

$$\Sigma = A \cup Q \cup \{\Lambda_0, f\}, \qquad \{\Lambda_0, f\} \cap (A \cup Q) = \varnothing.$$

Дополнительный символ Λ_0 будет использоваться для расширенного описания конфигураций. А именно, конфигурация uqv машины M будет представляться любым словом вида $\Lambda_0 \Lambda^k uqv \Lambda^s \Lambda_0$. Другими словами, мы разрешаем иметь слева и справа от символов конфигурации произвольное количество пустых символов, окаймлённое дополнительным символом Λ_0.

Дополнительный символ f будет играть вспомогательную роль, которая станет ясна позднее.

Достижимость будет проверяться для пары слов $u = \Lambda_0 q_0 x \Lambda_0$ и $v = \Lambda_0 f \Lambda_0$.

Осталось описать набор правил подстановки Δ. Правила будут двух типов.

Правила первого типа описывают такт работы МТ. Чтобы их задать, нужно аккуратно рассмотреть все возможные случаи положения и движения головки на ленте и выписать правила преобразования конфигурации.

Для удобства описания расширим таблицу переходов на множество $(A \cup \{\Lambda_0\}) \times Q$ по правилу $\delta(\Lambda_0, q) = \delta(\Lambda, q)$ для любого $q \in Q$, это формальная запись такого свойства «с точки зрения таблицы переходов символ Λ_0 не отличается от пустого символа Λ».

Пусть $\delta(a, q) = (b, q', 0)$. По такой строчке таблицы переходов добавим правила

$$qa \underset{\Delta}{\to} q'b, \qquad a \neq \Lambda_0, \tag{17.1}$$

$$q\Lambda_0 \underset{\Delta}{\to} q'b\Lambda_0. \tag{17.2}$$

Если $\delta(a, q) = (b, q', 1)$, то добавим правила

$$qa \underset{\Delta}{\to} bq', \qquad a \neq \Lambda_0, \tag{17.3}$$

$$q\Lambda_0 \underset{\Delta}{\to} bq'\Lambda_0. \tag{17.4}$$

Наконец, если $\delta(a, q) = (b, q', -1)$, то добавим правила

$$xqa \underset{\Delta}{\to} q'xb, \qquad \text{для всех } x \in A,\ a \neq \Lambda_0, \tag{17.5}$$

$$\Lambda_0 qa \underset{\Delta}{\to} \Lambda_0 q'\Lambda b, \qquad a \neq \Lambda_0, \tag{17.6}$$

$$xq\Lambda_0 \underset{\Delta}{\to} q'xb\Lambda_0, \qquad \text{для всех } x \in A, \tag{17.7}$$

$$\Lambda_0 q\Lambda_0 \underset{\Delta}{\to} \Lambda_0 q'\Lambda b\Lambda_0. \tag{17.8}$$

Правила второго типа задают преобразования слов после окончания работы МТ.

Для каждой пары (a, q), $a \in A \cup \{\Lambda_0\}$, на которой таблица переходов не определена, добавим правило

$$qa \underset{\Delta}{\to} fa. \tag{17.9}$$

Добавим также правила

$$af \underset{\Delta}{\to} f, \qquad \text{для всех } a \in A, \tag{17.10}$$

$$fa \underset{\Delta}{\to} f, \qquad \text{для всех } a \in A. \tag{17.11}$$

На этом построение набора правил закончено. Проверим, что построенные Δ, u, v удовлетворяют сформулированному выше требованию: машина M останавливается на x тогда и только тогда, когда $\Lambda_0 q_0 x \Lambda_0 \overset{*}{\underset{\Delta}{\to}} \Lambda_0 f \Lambda_0$.

Прежде всего заметим, что к слову вида

$$\Lambda_0\Lambda^k w\Lambda^s\Lambda_0,$$

где w — конфигурация МТ, применимо не более одного правила первого типа, и его применение даёт слово такого же вида

$$\Lambda_0\Lambda^{k'} w'\Lambda^{s'}\Lambda_0,$$

где w' — конфигурация после такта работы машины M на конфигурации w. (Для каждого правила (17.1–17.8) это утверждение проверяется непосредственным применением правила выполнения такта работы МТ и определения конфигурации.)

Значит, если МТ M не останавливается на входе x, то преобразованиями по данной системе правил будут получаться слова, в которые обязательно входит символ из Q, т.е. слово $\Lambda_0 f\Lambda_0$ получить из начального слова невозможно.

Пусть МТ M останавливается на входе x. Заметим, что возможна ровно одна подстановка для слова вида $\Lambda_0\Lambda^k w\Lambda^s\Lambda_0$, где w — конфигурация МТ. Поэтому возможно выполнение подстановок до тех пор, пока не будет достигнута финальная конфигурация. В финальной конфигурации применимо правило (17.9). После этого применением правил (17.10, 17.11) возможно убрать из слова все символы алфавита A. Останется в точности искомое слово $\Lambda_0 f\Lambda_0$.

Описание правил подстановки было вполне конструктивным. Поэтому легко поверить, что существует алгоритм, который по описанию МТ M и входа x строит описание Δ и слов u, v. Если говорить о (многоленточной) МТ, решающей такую задачу, нужно разбить её построение на несколько шагов:

- Определение размера алфавита Σ и построение МТ, которая преобразует код символа в описании M в код соответствующего символа в алфавите Σ.
- Преобразование описания x в описание слов $u = \Lambda_0 q_0 x\Lambda_0$ и $v = \Lambda_0 f\Lambda_0$.
- Запись описания правил первого типа по каждой строчке таблицы переходов.
- Перечисление всех пар (a, q), для которых таблица переходов не определена и запись правил второго типа, отвечающих таким парам.
- Запись остальных правил второго типа, для этого нужно перечислить все символы алфавита A.

□

Описанное преобразование МТ и её входа в набор правил подстановки обладает дополнительными интересными свойствами. Скажем, легко заметить, что правила зависят только от машины, а не от входа. Вспомнив о существовании универсальной машины, получаем такое следствие.

Следствие 17.3. *Существует такой граф подстановок, что задача достижимости для этого графа алгоритмически неразрешима.*

Неориентированные графы также можно задавать правилами подстановки. Но в этом случае подстановки двусторонние: можно не только заменять левую часть правила на правую, но и наоборот: правую на левую.

Задача 17.7. Докажите, что задача достижимости для неориентированных графов, заданных правилами подстановки, алгоритмически неразрешима.

Подсказка: используйте то же самое преобразование, что и в доказательстве теоремы, и докажите, что даже при двусторонних заменах слово v достижимо из слова u тогда и только тогда, когда машина M останавливается на входе x.