Лекции по теории алгоритмов

В. Н. Брагилевский

7 мая 2019 г.

Оглавление

1. Введение в теорию алгоритмов

1.1. Теория алгоритмов и понятие вычисления

Наш курс называется «Теория алгоритмов», хотя на самом деле его следовало бы назвать «Теория вычислений» (Computation theory), как традиционно эта область информатики называется на Западе. Впрочем, в нашей стране исторически сложилось именно такое название, поэтому было бы неправильно от него отказываться. Итак, центральным понятием этого курса является понятие вычисления, и мы в итоге должны разобраться, что же это такое. Давайте обсудим, почему понятие вычисления является более существенным, нежели понятие алгоритма, ведь они очевидно связаны между собой. Дело в том, что под алгоритмом обычно понимают некоторый способ описания вычисления — набор действий, которые необходимо осуществить для достижения искомого результата, результата вычислений. Таким образом вычисление существует само по себе, при этом оно может быть описано разными способами, с помощью разных алгоритмов.

В этом семестре вы изучаете сразу три курса, названия которых так или иначе связаны между собой: это курсы алгоритмов и структур данных, архитектуры вычислительных систем, теории алгоритмов или, что то же самое, теории вычислений. На курсе архитектуры вам будут рассказывать, как устроены машины, позволяющие нам проводить вычисления быстрее, чем если бы мы выполняли их вручную. На курсе алгоритмов вы будете изучать алгоритмы, позволяющие решать конкретные задачи из разных областей (скажем, эффективная обработка строк), и общие методы, позволяющие алгоритмы проектировать. Ещё в одном курсе, а именно в курсе математических основ защиты информации вы будете изучать некоторые конкретные алгоритмы теории чисел, такие как проверка чисел на простоту или вычисление наибольшего общего делителя двух чисел.

В дальнейшем вас ждут курсы по алгоритмам на графах и алгоритмам, используемым в компьютерной графике, за спиной курс вычислительной математики, посвящённый алгоритмам, решающим важные математические задачи, например, уравнения, для которых не существует явных формул.

Наша цель в этом курсе совсем другая, нас практически не интересуют методы решения конкретных задач, нас интересует сам феномен вычисления. Мы хотим формально, то есть математически строго, описать, что такое вычисление вообще.

1.2. Возникновение теории алгоритмов

Важно понять, как возникла эта дисциплина, зачем понадобилось формально определять, что такое вычисление. Один из самых известных алгоритмов — алгоритм Евклида — был описан на рубеже четвёртого и третьего веков до н.э. С тех пор человечество разработало огромное количество алгоритмов, но вплоть до первой трети

XX века никто не задумывался о необходимости формального определения. Чуть позже мы поговорим о том, что понимали под алгоритмами всё это время, но сейчас важнее другое. История теории алгоритмов берёт своё начало в 1900 году, когда крупнейший математик того времени Давид Гильберт на математическом конгрессе в Париже огласил список из 23 нерешённых математических проблем. В некоторых проблемах Гильберт просил найти «механическую процедуру» отыскания решения задачи, то есть фактически он просил найти алгоритм. К примеру, Десятая проблема Гильберта касалась отыскания общего метода решения диофантовых уравнений (уравнений в целых числах — с целыми коэффициентами и целыми же решениями). Эта проблема, кстати, была окончательно решена только в 1970 году советским российским математиком Ю. В. Матиясевичем. Гильберт не успокаивался: в 1922 году он начал заниматься проблемами основания математики, он хотел выстроить всю математику как формальную систему, начав с аксиом и правил вывода и получив в итоге все известные теоремы. Более того, в 1928 году он сформулировал ещё одну знаменитую проблему, так называемую «проблему разрешения» (с нем. *Entscheidungsproblem* или с англ. *decision problem*). Гильберт просил найти алгоритм (механическую процедуру), который принимал бы на вход описание некоторой формальной системы (набор аксиом и правил вывода) и конкретное утверждение в этой системе, а возвращал бы «да» или «нет» в зависимости от того, можно ли доказать это утверждение в данной формальной системе. Вкупе с формализацией всей математики решение этой проблемы означало бы получение механического способа выяснения истинности или ложности любого математического утверждения.

Гильберт всю свою жизнь был абсолютно уверен, что любая задача имеет решение, но некоторые математики довольно быстро стали подозревать, что иногда алгоритм придумать в принципе невозможно. Заметьте важное отличие: либо мы пока не придумали такой алгоритм (не хватило знаний, умений, математической интуиции), либо его вообще невозможно придумать.

Но как доказать, что чего-то не существует? Как минимум, нужно точно понимать, что такое этот алгоритм, дать ему формальное определение. Как это обычно бывает в математике: мы даём определение некоторому объекту, а потом доказываем, что если бы существовал такой объект с некоторыми свойствами, то это противоречило бы чему-то, в чём мы уже уверены. Только отсюда может вытекать несуществование этого объекта с этими свойствами. Подобные объекты с другими свойствами при этом вполне могут существовать, этому ничего не препятствует. Скажем, не существует треугольника, сумма длин двух сторон которого меньше третьей стороны. Мы точно знаем, что такое треугольник вообще, и можем доказать, что такого треугольника не существует.

Так вот, первые утверждения о неразрешимости (то есть о несуществовании алгоритма решения некоторой задачи) были получены как раз для проблемы разрешения Гильберта и сделано это было независимо Алонсо Чёрчем и Аланом Тьюрингом в 1935–36 годах. Первый разработал для этой цели аппарат λ-исчисления, а второй придумал то, что мы теперь называем машиной Тьюринга. И то, и другое может служить формальным определением алгоритма и обычно обозначается сейчас термином *модель вычислений*. Десятая проблема Гильберта о диофантовых уравнениях была, кстати, решена в том же отрицательном смысле, было доказано, что общего метода их решения не существует, а на одном из этапов доказательства используются именно машины Тьюринга. Оказалось, что обе эти модели вычислений эквивалент-

ны друг другу в том смысле, что любая задача, которую можно решить в одной из них, может быть также решена и в другой. Как следствие, если задача неразрешима средствами одной модели, то она неразрешима и средствами другой.

1.3. Структура курса

Итак, формальное определение понятия вычисления, или алгоритма, потребовалось для того, чтобы научиться доказывать, что в некоторых конкретных ситуациях алгоритма не существует, то есть что задача является алгоритмически неразрешимой. В первом разделе нашего курса мы будем изучать разные эквивалентные друг другу модели вычислений, а к его концу познакомимся с тезисом Чёрча–Тьюринга. Затем мы перейдём к проблеме вычислимости, то есть будем изучать, что вычислить можно, а что нельзя, будем рассматривать различные неразрешимые задачи. К концу апреля мы перейдём к третьему разделу курса, к теории сложности. В его рамках мы будем обсуждать те задачи, которые решить всё-таки можно, мы будем интересоваться тем, какие ресурсы (по времени или по памяти) для этого требуются, как задачи классифицируются по этим принципам. Можно ли считать практически вычислимой задачу, для решения которой с использованием всех имеющихся на планете вычислительных ресурсов требуется время, сравнимое с временем существования Вселенной? То есть получить решение вообще говоря можно, но будет ли к тому времени существовать заказчик вычислений?

1.4. Вычислительные парадигмы и задачи

Мы можем говорить о задаче вычисления чего бы то ни было как о числовой функции. Скажем, даже рисование в Paint в конечном итоге приводит к преобразованию действий пользователя (последовательность сигналов от мыши и клавиатуры, легко кодируемых числами) в файл с изображением, также кодируемый с помощью чисел. Любое другое вычисление также можно представить подобным образом, задав некоторым специальным образом числовое кодирование сущностей, участвующих в вычислении. Поэтому мы всегда можем говорить о вычислении как о числовой функции, возможно, с несколькими аргументами, исходными данными.

Работать с числами не всегда удобно. Заметьте, что говоря о числах, я не уточнял, какие именно числа имеются в виду, натуральные, целые, действительные, комплексные или, быть может, вообще p-адические. Разнообразие числовых множеств может усложнять работу с ними. Во многих ситуациях проще манипулировать символами. Вычисление при таком подходе может выглядеть как преобразование символьных строк, называемых также словами из некоторого алфавита.

Мы будем называть вычислительной парадигмой способ представления вычислительной задачи, выбирая между вычислением как числовой функцией или же как манипулированием символами.

Часто выделяют специальный важный вид вычислительных задач, а именно, проблемы разрешения (decision problem). Так называют задачи, ответом на которые является «да» или «нет». Скажем, является ли заданное число n простым, да или нет? Или: является ли некоторая формула доказуемой в заданной формальной системе?

Сформулируем некоторые определения, которые облегчат рассмотрение этой и других задач в форме манипулирования символами. *Алфавит* — это конечное множество знаков, называемых *буквами*. Например, алфавитом является множество $\Sigma = \{a, b\}$, это двухбуквенный алфавит. Конечная цепочка следующих друг за другом букв называется *словом*. Множество всех слов алфавита Σ обозначается символом Σ^*. Среди всех возможных слов выделим *пустое* слово, не содержащее никаких букв, и обозначим его символом ε ($\varepsilon \in \Sigma^*$). Слово из Σ^* называется также словом в алфавите Σ. *Языком* в алфавите Σ называется любое подмножество множества Σ^*.

Применительно к языкам задача разрешения называется *задачей распознавания*, при этом вопрос формулируется в следующей форме: принадлежит ли заданное слово некоторому языку. Говорят также, что задача распознавания *является экземпляром* задачи разрешения.

Приведём два примера задач распознавания. Рассмотрим алфавит $\Sigma = \{|\}$ и язык $P = \{||, |||, |||||, |||||||, \dots\}$, содержащий все слова в алфавите Σ, длина которых (то есть количество символов) является простым числом. Тогда вопрос о простоте натурального числа (в теоретико-числовой форме) совпадает с задачей распознавания языка P: принадлежность слова языку соответствует простоте натурального числа, равного длине этого слова, и наоборот, если слово языку не принадлежит, то соответствующее число не является простым. Если нам удастся подобрать алфавит для записи формул логики первого порядка (с предикатами и кванторами), а потом рассмотреть язык, содержащий все истинные формулы, то проверку истинности формулы также можно представить в виде задачи распознавания.

Помимо задач распознавания при работе с символами мы можем рассматривать задачи преобразования, например: продублировать каждый символ входного слова, инвертировать его символы, удалить каждый второй символ. Задачи преобразования больше похожи на числовые функции: здесь можно говорить об области определения и области значения. Задачи распознавания могут рассматриваться как частный случай задач преобразования, когда область значений состоит из двух возможных символов, например 1 и 0, соответствующих ответам «да» и «нет».

Любопытно, что задачу преобразования всегда можно свести к, казалось бы, гораздо более простой, задаче распознавания, достаточно лишь подходящим образом подобрать распознаваемый язык. Предположим, что перед нами стоит задача вычисления k-го простого числа: в терминах числовых функций это функция $f(k)$, возвращающая k-е по порядку простое число; в терминах задач преобразования это построение слова в односимвольном алфавите, длина которого равна k-му простому числу, по заданному входному слову длины k. Приведём теперь постановку соответствующей задачи распознавания. Пусть алфавит содержит два символа: $\Sigma = \{|, \#\}$, а язык

$$P' = \{|\#||, ||\#|||, |||\#|||||, ||||\#|||||||, \dots\}$$

(слева от символа # стоит порядковый номер, а справа — соответствующее простое число, оба числа представлены в унарной системе счисления). Для отыскания k-го простого числа необходимо последовательно проверять принадлежность языку P' слов вида

$$\underbrace{|||\dots|||}_{k \text{ символов}}\#||,\ \underbrace{|||\dots|||}_{k \text{ символов}}\#|||,\ \underbrace{|||\dots|||}_{k \text{ символов}}\#||||,\ \underbrace{|||\dots|||}_{k \text{ символов}}\#|||||,\ \underbrace{|||\dots|||}_{k \text{ символов}}\#||||||,\quad \dots$$

Как только в ответ на задачу распознавания будет получено «да», справа от символа # будет находиться искомое простое число, записанное в унарной системе счисления.

Здесь мы воспользовались кодированием натуральных чисел символами. Можно выполнять и обратное кодирование, то есть представлять символы числами. Для этого, к примеру, можно воспользоваться кодовыми таблицами ASCII или Unicode. Вообще, кодирование позволяет свободно переходить от преобразования символов к числовым функциям и наоборот, выбирая то представление, которое в данный момент удобнее.

Возможность переходить от решения задач одного типа к задачам другого типа и менять одновременно представление данных позволяет ограничиться теоретическим рассмотрением задач распознавания. Именно их анализ будет составлять существенную часть нашего курса.

2. Модели вычислений

2.1. Нормальные алгорифмы Маркова

2.1.1. Алфавит, слова, конкатенация слов, подслова и вхождения

Алфавит — это конечное множество знаков, называемых *буквами*. Например, алфавитом является множество $\Sigma = \{a, b\}$, это двухбуквенный алфавит. Конечная цепочка следующих друг за другом букв называется *словом*. Множество всех слов алфавита Σ обозначается символом Σ^*. Среди всех возможных слов выделим *пустое* слово, не содержащее никаких букв, и обозначим его символом ε ($\varepsilon \in \Sigma^*$). Слово из Σ^* называется также словом в алфавите Σ. Будем далее обозначать слова прописными латинскими буквами.

Конкатенацией слов P и Q называется слово, обозначаемое PQ и содержащее сначала цепочку букв слова P, а затем цепочку букв слова Q. Например, конкатенацией слов $abab$ и $aaabbb$ в алфавите $\Sigma = \{a, b\}$ является слово $ababaaabbb$.

Слово P называется *подсловом* слова Q, если Q является конкатенацией слов X, P и Y (то есть, если $Q = XPY$), где X и Y (называемые, соответственно, префиксом и суффиксом), возможно, пустые. Например, слово ab является подсловом слова $aabb$, причём $X = a$, $Y = b$. Другой пример: слово 111 является подсловом слова 11111, причём здесь пара X и Y может определяться различными способами. Наконец, само слово «слово» является подсловом слова «подслово», которое в свою очередь иногда называется *надсловом* слова «слово». Вообще, всякое слово является как своим подсловом, так и надсловом. Пустое слово является подсловом любого слова.

Подслово может *входить* в слово неоднократно. Например, имеется три *вхождения* слова ab в слово $ababab$. Каждое такое вхождение однозначно определяется парой, состоящей из префикса и суффикса (X, Y). Так, в приведённом примере три вхождения определяются парами: 1) $(\varepsilon, abab)$; 2) (ab, ab); 3) $(abab, \varepsilon)$. Вхождение непустого слова называется первым, если соответствующее слово не является подсловом префикса X. Например, первым вхождением слова ab в слово $ababab$ является вхождение, определяемое парой $X = \varepsilon$, $Y = abab$. Вхождения пустого слова в слово P определяются всеми возможными разбиениями слова P вида $P = XY$. Первым вхождением пустого слова считается вхождение с пустым префиксом (то есть вхождение в начало слова).

2.1.2. Марковские подстановки и их применение

Пара слов, обозначаемая $P \longrightarrow Q$, называется *марковской подстановкой*. Говорят, что подстановка *применима* к слову M, если левая часть подстановки (P) является подсловом слова M. Если подстановка $P \longrightarrow Q$ применима к слову M, то её *применением* называется *замена первого вхождения слова P в слово M на слово Q*.

В качестве примера рассмотрим марковскую подстановку в алфавите $\Sigma = \{a, b\}$:

$$ab \longrightarrow a$$

Эта подстановка применима к словам ab, $abab$, $aabb$, $aabaaabb$ и не применима к словам aa, ε, ba, $bbbbb$. В результате применения подстановки к словам из первого списка получим следующее (обратите внимание на использование двойной стрелки для обозначения применения подстановки к слову, первое вхождение подчёркнуто):

$$\underline{ab} \Longrightarrow a$$
$$\underline{ab}ab \Longrightarrow aab$$
$$a\underline{ab}b \Longrightarrow aab$$
$$a\underline{ab}aaabb \Longrightarrow aaaaabb$$

В левой и правой частях подстановки может находиться пустое слово. В первом случае её смысл заключается во вставке правой части подстановки в начало слова, а во втором — удаление первого вхождения левой части подстановки. Следует заметить, что эти действия соответствуют нашим определениям: первое вхождение пустого слова является вхождением в начало слова, поэтому речь в этом случае идёт о вставке в начало, а замена на пустое слово является в точности удалением. Например, применение подстановки $\varepsilon \longrightarrow a$ к слову ba приводит к преобразованию $ba \Longrightarrow aba$, а применение подстановки $a \Longrightarrow \varepsilon$ к тому же слову — к преобразованию $ba \Longrightarrow b$. Подстановка вида $\varepsilon \longrightarrow P$ применима к любому слову, в том числе пустому, например, $\varepsilon \Longrightarrow P$. Здесь следует обратить внимание на различие между подстановкой (одинарная стрелка) и её действием на слово (двойная стрелка).

Далее нам потребуется особый вид подстановок, называемых *заключительными* или *финальными*. Будем обозначать их одинарной стрелкой с точкой:

$$P \longrightarrow\!. \; Q$$

Смысл этого названия станет понятен позднее.

2.1.3. Схемы нормальных алгорифмов

Схемой нормального алгорифма называется конечный упорядоченный набор марковских подстановок, некоторые из которых могут быть заключительными:

$$\begin{cases} P_1 & \longrightarrow (.)\ Q_1 \\ P_2 & \longrightarrow (.)\ Q_2 \\ & \dots \\ P_n & \longrightarrow (.)\ Q_n \end{cases}$$

Однократным применением схемы алгорифма к заданному слову называется применение первой (по порядку следования в схеме) применимой к нему марковской подстановки. Если ни одна из подстановок схемы к слову не применима, то и схема в целом является неприменимой к данному слову.

Нормальным алгорифмом называется последовательность однократных применений схемы к заданному слову, завершаемая в одном из двух случаев:

1) схема оказывается неприменимой к слову;
2) применена заключительная подстановка.

Во всех последующих примерах будем считать, что схемы определены для алфавита $\Sigma = \{a, b\}$. Рассмотрим пример применения схемы

$$\begin{cases} ab \longrightarrow \varepsilon \\ a \longrightarrow b \end{cases}$$

к слову $aababab$ (подчёркнуто первое вхождение левой части применимой подстановки):

$$a\underline{ab}abab \Longrightarrow a\underline{ab}ab \Longrightarrow a\underline{ab} \Longrightarrow \underline{a} \Longrightarrow b$$

Первые три шага нормального алгорифма состоят из применения первой подстановки, а на четвёртом шаге применяется вторая подстановка (первая уже неприменима). Четвёртый шаг оказывается последним, поскольку после него ни одна из подстановок неприменима (первый случай завершения нормального алгорифма).

Схема может состоять из единственной подстановки:

$$\begin{cases} a \longrightarrow b \end{cases}$$

Ясно, что соответствующий нормальный алгорифм заменяет все вхождения символа a на символ b, после чего завершается.

Наличие в схеме заключительной подстановки приводит к завершению нормального алгорифма сразу после её применения, например, применение схемы

$$\begin{cases} b \longrightarrow . \ a \\ a \longrightarrow b \end{cases}$$

к слову aaa приводит к следующим шагам нормального алгорифма:

$$\underline{a}aa \Longrightarrow \underline{b}aa \Longrightarrow aaa$$

На первом шаге первая подстановка неприменима (в слове отсутствует буква b), поэтому применяется вторая. В результате её действия буква b появляется, поэтому на втором шаге алгорифма применяется первая подстановка. Поскольку она является заключительной, действие алгорифма после её применения завершается.

Нормальный алгорифм может вообще не завершаться. Так, применение схемы

$$\begin{cases} a \longrightarrow b \\ b \longrightarrow a \end{cases}$$

к любому непустому слову приводит к следующему: сначала все буквы a заменяются на b, после чего начинается бесконечное «мигание» первой буквы слова, она заменяется то на a, то на b.

Слово в процессе преобразования нормальным алгорифмом может даже неограниченно увеличиваться, таково, к примеру, действие схемы

$$\begin{cases} \varepsilon \longrightarrow a \end{cases}$$

Соответствующий нормальный алгорифм будет бесконечно долго вставлять букву a в начало слова (эта ситуация будет иметь место, даже если начать с пустого слова).

2.1.4. Примеры нормальных алгорифмов

Рассмотрим некоторые простые примеры нормальных алгорифмов. Следующая схема предназначена для удаления всех символов входного слова в алфавите $\Sigma = \{a, b\}$:

$$\begin{cases} a \longrightarrow \varepsilon \\ b \longrightarrow \varepsilon \end{cases}$$

Сначала удаляются все символы a, затем все символы b. Для удаления одного любого символа следует обе подстановки сделать заключительными:

$$\begin{cases} a \longrightarrow. \varepsilon \\ b \longrightarrow. \varepsilon \end{cases}$$

Такой алгорифм оказывается «предвзятым»: если в слове есть символ a, то именно он и удаляется. Символ b удалится, только если символов a в слове нет. Удаляются первые вхождения соответствующих символов.

Рассмотрим односимвольный алфавит $\Sigma = \{|\}$, посредством которого представим натуральное число в унарной записи (пустое слово соответствует нулю). Схема нормального алгорифма, увеличивающего заданное число на единицу, выглядит следующим образом:

$$\begin{cases} \varepsilon \longrightarrow. | \end{cases}$$

Например:

$$|| \Longrightarrow |||$$

Вычитание единицы (считаем, что $0 - 1 = 0$):

$$\begin{cases} | \longrightarrow. \varepsilon \end{cases}$$

Более сложная задача: проверим, делится ли число, записанное в унарной системе счисления, на три. Для этого будем стирать по три цифры за один шаг алгорифма, после чего проверим, что осталось. Результатом преобразования будет символ $|$, если число делится на 3 и пустое слово в противном случае. Схема алгорифма имеет следующий вид:

$$\begin{cases} ||| \longrightarrow \varepsilon \\ || \longrightarrow. \varepsilon \\ | \longrightarrow. \varepsilon \\ \varepsilon \longrightarrow. | \end{cases}$$

Примеры: $||||||| \Longrightarrow |||| \Longrightarrow | \Longrightarrow \varepsilon$ (7 не делится на 3), $||||||||| \Longrightarrow |||||| \Longrightarrow ||| \Longrightarrow \varepsilon \Longrightarrow |$ (9 делится на 3).

2.1.5. Расширение алфавита и алгорифмы над алфавитом

Нормальный алгорифм задаётся схемой, которая в свою очередь состоит из подстановок, левая и правая части которых содержат буквы некоторого алфавита. Тот же алфавит используется для записи слов, к которым нормальные алгорифмы применяются, и, соответственно, для результатов применения. Иногда символов, уже

имеющихся в алфавите, недостаточно, то есть в процессе преобразования слова оказываются необходимыми другие (вспомогательные) символы.

Алфавит Σ' называется *расширением* алфавита Σ, если $\Sigma' \supset \Sigma$. Говорят, что нормальный алгорифм действует *над* алфавитом Σ, если он задан в некотором расширении алфавита Σ (то есть в схеме алгорифма могут встречаться символы из расширения Σ). При этом важно, что и входное слово, и окончательный результат преобразования (то есть слово, получаемое после завершения алгорифма), содержат символы только из основного алфавита Σ, но не из его расширения. В ситуации, когда расширение алфавита не требуется, используют предлог *в*, то есть говорят «алгорифм действует в алфавите Σ».

Возможность расширить алфавит вспомогательными символами иногда позволяет существенно упростить решение задачи, а иногда делает невозможное без этого решение возможным. В качестве примера разработаем схему алгорифма, который удаляет первую букву непустого входного слова в алфавите $\Sigma = \{a, b\}$, какой бы она не была. Для этого вставим в начало слова вспомогательный символ $*$, а затем удалим его вместе со следующей за ним буквой заключительной подстановкой (таких подстановок в схеме должно быть две по числу букв в алфавите). При разработке схемы следует внимательно относиться к порядку следования подстановок, учитывая, что применяется всегда первая из применимых к слову подстановок. Используя эти соображения, получаем следующую схему:

$$\begin{cases} *a \longrightarrow . \ \varepsilon \\ *b \longrightarrow . \ \varepsilon \\ \varepsilon \longrightarrow * \end{cases}$$

Вспомогательный символ фиксирует во входном слове начальную позицию. Без его использования отличить первую букву от всех последующих невозможно. Заметим, что на первом шаге нормального алгорифма всегда будет применяться последняя подстановка схемы, а на втором — первая или вторая (заключительная):

$$abab \Longrightarrow *abab \Longrightarrow bab$$
$$baba \Longrightarrow *baba \Longrightarrow aba$$

Если применить эту схему к пустому слову, то мы получим зацикливание (первые две подстановки останутся после первого шага неприменимыми, а третья не предусматривает остановки):

$$\varepsilon \Longrightarrow * \Longrightarrow ** \Longrightarrow *** \Longrightarrow \ldots$$

Устранить такое поведение можно, добавив третью заключительную подстановку, удаляющую одинокий символ $*$:

$$\begin{cases} *a \longrightarrow . \ \varepsilon \\ *b \longrightarrow . \ \varepsilon \\ * \longrightarrow . \ \varepsilon \\ \varepsilon \longrightarrow * \end{cases}$$

Теперь поведение на пустом слове следующее: $\varepsilon \Longrightarrow * \Longrightarrow \varepsilon$. По-прежнему применение заключительных подстановок становится возможным только после применения последней подстановки, вставляющей вспомогательный символ $*$ в начало слова.

Полезно рассмотреть ещё один пример нормального алгорифма, но уже над алфавитом $\Sigma = \{0, 1\}$. Предположим, что входное слово задаёт натуральное число в двоичной форме, и требуется увеличить его на единицу. Действие алгорифма будет состоять из двух этапов: сначала найдём конец слова (для этого вставленный в начало слова вспомогательный символ будет последовательно перемещаться в его конец), а затем выполним прибавление единицы с переносом единичного разряда (перенос будет реализован движением в обратном направлении), если это окажется необходимым. Схема этого алгорифма имеет следующий вид (для удобства подстановки здесь помечены):

$$\begin{cases} 0b \longrightarrow .\ 1 & \text{(а)} \\ 1b \longrightarrow b0 & \text{(б)} \\ b \longrightarrow .\ 1 & \text{(в)} \\ a0 \longrightarrow 0a & \text{(г)} \\ a1 \longrightarrow 1a & \text{(д)} \\ 0a \longrightarrow 0b & \text{(е)} \\ 1a \longrightarrow 1b & \text{(ё)} \\ \varepsilon \longrightarrow a & \text{(ж)} \end{cases}$$

Первый этап реализуется применением подстановки (ж) и следующей за ней последовательностью применений подстановок (г) и (д). В результате первого этапа вспомогательный символ a оказывается в конце слова. Затем начинается второй этап: подстановки (е) или (ё) заменяют a на b и либо срабатывают заключительные подстановки (а) или (в), либо запускается перенос единичного разряда подстановкой (б). Рассмотрим действие алгорифма на примерах:

$$\varepsilon \overset{\text{(ж)}}{\Longrightarrow} a \overset{\text{(ж)}}{\Longrightarrow} aa \overset{\text{(ж)}}{\Longrightarrow} aaa \overset{\text{(ж)}}{\Longrightarrow} \ldots$$

$$0 \overset{\text{(ж)}}{\Longrightarrow} a0 \overset{\text{(г)}}{\Longrightarrow} 0a \overset{\text{(е)}}{\Longrightarrow} 0b \overset{\text{(а)}}{\Longrightarrow} 1$$

$$1 \overset{\text{(ж)}}{\Longrightarrow} a1 \overset{\text{(д)}}{\Longrightarrow} 1a \overset{\text{(ё)}}{\Longrightarrow} 1b \overset{\text{(б)}}{\Longrightarrow} b0 \overset{\text{(в)}}{\Longrightarrow} 10$$

$$11 \overset{\text{(ж)}}{\Longrightarrow} a11 \overset{\text{(д)}}{\Longrightarrow} 1a1 \overset{\text{(д)}}{\Longrightarrow} 11a \overset{\text{(ё)}}{\Longrightarrow} 11b \overset{\text{(б)}}{\Longrightarrow} 1b0 \overset{\text{(б)}}{\Longrightarrow} b00 \overset{\text{(в)}}{\Longrightarrow} 100$$

После достижения первым вспомогательным символом конца слова действуют следующие соображения. Если число заканчивается на цифру 0, то она заменяется на 1 и вычисление завершается. Если число заканчивается на цифру 1, то она заменяется на 0, а затем запускается перенос разряда. Разработанный алгорифм к пустому слову неприменим. Эту схему нетрудно обобщить на десятичную систему счисления: для этого достаточно добавить подстановки, дублирующие подстановки (а), (г)-(ё) для остальных цифр 2–9 и учесть, что роль 1, как разряда, для которого возникает перенос, теперь будет играть цифра 9.

2.1.6. Нормально вычислимые функции

Нормальные алгорифмы в описанной выше форме не производят собственно вычислений, они преобразуют слова, то есть манипулируют символами. Однако результаты таких преобразований вполне можно интерпретировать как вычисления, а

значит, нормальными алгорифмами можно воспользоваться для определения вычислимых функций.

Итак, назовём функцию f, заданную на некотором множестве слов алфавита Σ, *нормально вычислимой*, если найдётся такой нормальный алгорифм над этим алфавитом, что каждое слово $P(\in \Sigma^*)$ из области определения f преобразуется этим алгорифмом в слово $f(P)$.

Заметим, что определение нормально вычислимой функции допускает использование нормальным алгорифмом вспомогательных символов из расширения алфавита Σ. Если нормальный алгорифм на некотором слове не завершается, то считается, что функция на этом слове не определена.

2.2. λ-исчисление Чёрча

Модели вычисления начали возникать задолго до появления первых языков программирования, однако впоследствии открылось неожиданное их применение. Стало возможным использовать теоретические результаты, полученные для различных моделей вычисления, для исследования собственно языков программирования. Интересно, что далеко не всегда это было ясно создателям языков программирования. Изучаемые в этом разделе понятие λ-определимых функций и λ-исчисление лежат в основе так называемого функционального программирования, идеи и подходы которого применяются сейчас практически во всех достаточно распространённых современных языках программирования.

Одной из важнейших характеристик функционального программирования является использование функций как значений первого класса («first class value»). Это означает, что функции наряду с данными других типов — числами, логическими значениями, списками — могут быть параметрами или возвращаемыми значениями других функций, а кроме того с ними можно выполнять различные операции, например, композицию (суперпозицию). Такое широкое применение функций требует введения удобной нотации (способа обозначения).

Попробуем определить смысл выражения $x - y$. Во-первых, его можно трактовать как функцию переменной x:

$$f(x) = x - y,$$

что иначе можно записать как

$$f : x \mapsto x - y.$$

С другой стороны то же самое выражение можно понимать как функцию переменной y:

$$g(y) = x - y$$

или

$$g : y \mapsto x - y.$$

Поскольку в этих примерах смысл выражения с точки зрения функции меняется кардинальным образом, необходим способ, позволяющий явно указать, что именно является аргументом функции, но при этом желательно избежать её именования. Действительно, ведь в арифметике для указания того факта, что $2 \times 2 = 4$ не требуется вводить имена для чисел 2 и 4.

Соответствующая нотация была изобретена в 30-е годы XX века американским логиком и математиком Алонсо Чёрчем (1903—1995). В этой нотации в качестве вспомогательного символа используется греческая буква λ. Предыдущие примеры в λ-нотации Чёрча выглядят так:

$$f = \lambda x\,.\,x - y,\ g = \lambda y\,.\,x - y.$$

Вообще говоря, в именах f и g теперь необходимости нет, в выражениях их можно опускать:

$$\begin{aligned}(\lambda x\,.\,x - y)(42) &= 42 - y,\\ (\lambda y\,.\,x - y)(42) &= x - 42.\end{aligned}$$

Если необходимо использовать функции многих переменных, λ-нотацию нетрудно расширить, записав, например:

$$h(x, y) = x - y = \lambda xy\,.\,x - y.$$

Однако этого расширения можно избежать, заменив функции многих переменных на функции одной переменной, значениями которых являются другие функции. Определим функцию h^* следующим образом:

$$h^* = \lambda x\,.\,(\lambda y\,.\,x - y).$$

Значением функции h^* в точке a будет уже другая функция:

$$h^*(a) = (\lambda x\,.\,(\lambda y\,.\,x - y))(a) = \lambda y\,.\,a - y.$$

Вычислим теперь $(h^*(a))(b)$:

$$(h^*(a))(b) = (\lambda y\,.\,a - y)(b) = a - b = h(a, b).$$

Таким образом, можно считать, что функция h^* *представляет* ту же функцию, что и h, но является функцией одной переменной. Такой способ записи функций многих переменных называется *каррированием* по имени другого американского логика Хаскелла Карри (1900—1982). Известно, впрочем, что до Карри этим приёмом пользовались еще Моисей Исаевич Шёнфинкель (1889—1942) и Готлоб Фреге (1848—1925). Имя Хаскелла Карри дало также название популярному в академических кругах языку программирования Haskell. Каррирование полезно еще и по той причине, что оно явно вводит в рассмотрение *функции высшего порядка* (*higher-order functions*), т.е. функции, аргументами или значениями которых среди прочего могут являться функции.

Введённая Чёрчем λ-нотация важна не сама по себе, а как инструмент построения формальной системы λ-исчисления, которое является одной из формализаций понятия алгоритма и одновременно теоретической основой функционального программирования.

2.2.1. Определение λ-термов

Дадим теперь *индуктивное определение* множества λ-термов. Такого сорта определения встречаются в дискретной математике довольно часто: мы перечисляем некие

простейшие элементы (термы) множества (базис индукции), а потом указываем правила для построения из них всех остальных (индуктивный переход).

Пусть задано счётное множество переменных x_1, x_2, x_3, ... Множество выражений, называемых λ-термами, определяется с помощью следующих трёх условий:

- Все переменные являются λ-термами.
- Если M и N — произвольные λ-термы, то (MN) — λ-терм, называемый применением (application).
- Если x — переменная, а M — произвольный λ-терм, то $(\lambda x\,.\,M)$ — λ-терм, называемый абстракцией (abstraction).

Введённым понятиям можно придать следующую неформальную интерпретацию. Так, можно считать, что *λ-абстракция* обозначает функцию одной переменной, причём переменная, указанная после символа λ, является её *формальным* параметром. При таком подходе *применение* MN означает вызов функции M с передачей ей в качестве *фактического* параметра выражения N. То, что при этом используется бесскобочная запись (в отличие от более традиционного $f(x)$), может напоминать, например, о выражениях с тригонометрическими функциями:

$$\operatorname{tg} x = \frac{\sin x}{\cos x}.$$

В качестве переменных всюду далее будем использовать буквы x, y, z, s, t, u, v или w с индексами и без, а прописными латинскими буквами M, N, P и Q будем обозначать произвольные λ-термы.

Рассмотрим примеры λ-термов, построенных в соответствии с данным выше определением:

1. x
2. (xy)
3. $(\lambda x\,.\,x)$
4. $((\lambda x\,.\,y)\,t)$
5. $(\lambda x\,.\,y)$
6. $(\lambda x\,.\,(\lambda y\,.\,(xy)))$
7. $((\lambda x\,.\,(\lambda x\,.\,(xu)))(\lambda x\,.\,v))$

Здесь терм (1) это просто переменная, терм (2) представляет собой применение переменной x к переменной y (заметьте, что в определении не требуется, чтобы слева в применении стояла функция), терм (3) можно трактовать как тождественную функцию (тело «функции» совпадает с формальным параметром), а терм (5) — как константную (её значение всегда равно y независимо от значения аргумента x). Терм (7) — это применение одной абстракции к другой; здесь можно обратить внимание на многократное использование переменной x в λ-абстракциях, наше определение этому не препятствует, хотя при понимании таких выражений возможны определённые трудности.

Заметим, что сам символ λ и выражение $\lambda x\,.$ термами не являются, они используются только как фрагменты λ-абстракции.

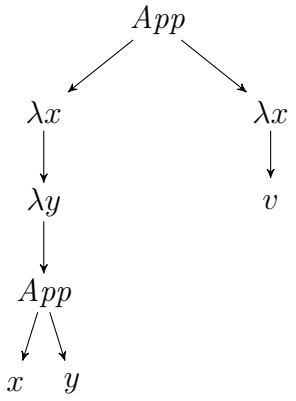
Упрощая запись термов, примем соглашение о том, что в выражениях с λ-термами можно опускать все внешние скобки, а также некоторые внутренние, считая, что:

- применение является левоассоциативным, то есть выражение MNP понимается как $((MN)P)$;
- абстракция является правоассоциативной, причём символ λ захватывает максимально большое выражение, то есть терм $\lambda x\,.\,MN$ соответствует $(\lambda x\,.\,(MN))$, а терм $\lambda x\,.\,\lambda y\,.\,\lambda z\,.\,M$ означает $(\lambda x\,.\,(\lambda y\,.\,(\lambda z\,.\,M)))$.

Теперь приведённые выше примеры можно переписать следующим образом:

x	x
(xy)	xy
$(\lambda x\,.\,x)$	$\lambda x\,.\,x$
$((\lambda x\,.\,y)\,t)$	$(\lambda x\,.\,y)\,t$
$(\lambda x\,.\,y)$	$\lambda x\,.\,y$
$(\lambda x\,.\,(\lambda y\,.\,(xy)))$	$\lambda x\,.\,\lambda y\,.\,xy$
$((\lambda x\,.\,(\lambda x\,.\,(xu)))(\lambda x\,.\,v))$	$(\lambda x\,.\,\lambda x\,.\,xu)\,(\lambda x\,.\,v)$

Скобки отражают структуру терма, которая на самом деле определяется последовательностью использования конкретных правил из определения при его построении. Например, терм $(\lambda x\,.\,\lambda y\,.\,xy)\,(\lambda x\,.\,v)$ — это применение из двух абстракций, тело первой из которых есть снова абстракция с телом — применением одной переменной к другой, а тело второй есть переменная. Это крайне неуклюжее описание гораздо удобнее представить графически, нарисовав *дерево разбора* терма:



Узлы дерева вида App обозначают применение, узлы λx и λy — абстракции по соответствующим переменным, всё остальное — узлы переменных.

2.2.2. Свободные и связанные переменные

Дадим теперь несколько определений, характеризующих роль переменных в λ-термах. *Областью действия* (scope) λ-абстракции λx в терме $(\lambda x\,.\,M)$ называется терм M. *Вхождением* переменной в терм называется любое её использование внутри этого терма (в том числе и сразу после символа λ).

Вхождение переменной x в терм M называется

- *связанным* (bound), если оно находится в области действия некоторой абстракции λx в терме M или является следующей за символом λ буквой x;
- *свободным* (free) во всех остальных случаях.

Заметим, что в этом определении идёт речь о связанных и свободных *вхождениях* переменной x, а не о самой переменной x. Переменная, вообще говоря, может входить в терм несколько раз, причём некоторые из её вхождений могут оказаться свободными, а другие — связанными, например в терме $x\,\lambda x\,.\,x$ имеется три вхождения переменной x: первое вхождение является свободным, второе и третье — связанными.

Множеством свободных переменных терма P (обозначается $FV(P)$ от англ. free variables), будем называть множество переменных, имеющих в терме P хотя бы одно свободное вхождение:

$$\begin{array}{lcl} FV(x) & = & \{x\} \\ FV(\lambda x.M) & = & FV(M)\backslash\{x\} \\ FV(MN) & = & FV(M) \cup FV(N) \end{array}$$

Следует обратить внимание на то, что структура определения повторяет структуру определения множества λ-термов.

Множеством связанных переменных терма P (обозначается $BV(P)$ от англ. bound variables), будем называть множество переменных, имеющих в терме P хотя бы одно связанное вхождение:

$$\begin{array}{lcl} BV(x) & = & \varnothing \\ BV(\lambda x.M) & = & \{x\} \cup BV(M) \\ BV(MN) & = & BV(M) \cup BV(N) \end{array}$$

В соответствии с этими определениями множества свободных и связанных переменных терма $x\,\lambda x\,.\,x$ будут совпадать:

$$\begin{array}{ll} FV(x\,\lambda x\,.\,x) & = FV(x) \cup FV(\lambda x\,.\,x) = \{x\} \cup (FV(x)\backslash\{x\}) = \\ & = \{x\} \cup (\{x\}\backslash\{x\}) = \{x\} \cup \varnothing = \{x\}; \\ BV(x\,\lambda x\,.\,x) & = BV(x) \cup BV(\lambda x\,.\,x) = \varnothing \cup (\{x\} \cup BV(x)) = \\ & = \varnothing \cup (\{x\} \cup \varnothing) = \{x\}. \end{array}$$

2.2.3. Подстановка

Перейдём теперь к понятию *подстановки*, которое используется при определении вычисления в терминах λ-исчисления. В вводных курсах по программированию (независимо от используемого языка программирования) обычно используют следующее несколько упрощённое объяснение смысла формальных и фактических параметров: при вызове подпрограммы предварительно вычисленные значения фактических параметров подставляются в теле подпрограммы вместо её формальных параметров, после чего начинает исполняться собственно подпрограмма. Это соображение можно использовать и в λ-исчислении, однако необходимо точно определить, что именно понимается под подстановкой. В том, что это не так просто, как может показаться на первый взгляд, можно убедиться, дав следующее *неправильное* определение.

Неправильное определение. *Операцией подстановки терма N (фактический параметр) в терм P (подпрограмма) вместо переменной x (формальный параметр), обозначаемой*[1] *как $[N/x]P$, называется операция преобразования терма, выполняемая по правилам:*

$$\begin{array}{lcl} [N/x]x & = & (N); \\ [N/x]y & = & y, \quad \textit{если } x \neq y; \\ [N/x](PQ) & = & (([N/x]P)([N/x]Q)); \\ [N/x](\lambda x\,.\,P) & = & (\lambda x\,.\,([N/x]P)); \\ [N/x](\lambda y\,.\,P) & = & (\lambda y\,.\,([N/x]P)), \quad \textit{если } x \neq y. \end{array}$$

Проблемы этого определения скрываются в двух последних правилах. Понятно, что при выполнении подстановки не должен меняться смысл терма (точнее, наше интуитивное представление о его интерпретации), однако посмотрим, что произойдёт с термом $\lambda x\,.\,x$ при проведении подстановки $[t/x]$ по сформулированным выше правилам:

$$[t/x](\lambda x\,.\,x) = \lambda x\,.\,([t/x]x) = \lambda x\,.\,t.$$

Получается, что тождественная функция превратилась в константную, а связанная в исходном терме переменная x исчезла, превратившись в свободную переменную t.

Теперь выполним подстановку $[y/x]$ в терме $\lambda y\,.\,x$:

$$[y/x](\lambda y\,.\,x) = \lambda y\,.\,([y/x]x) = \lambda y\,.\,y.$$

Произошло нечто противоположное: константная функция превратилась в тождественную. В таких случаях обычно говорят, что произошёл *захват* свободной переменной y, имея в виду, что в подставляемом терме y выступала в качестве свободной переменной, а после подстановки она же оказалась связанной.

Таким образом, корректное определение подстановки должно учитывать связанные и свободные переменные, препятствуя исчезновению одних и захвату других. Проблему с исчезновением связанной переменной можно решить с помощью запрета на распространение подстановки $[N/x]$ внутрь абстракции $\lambda x\,.\,P$, причём это вполне согласуется с пониманием смысла формального параметра подпрограммы: он никак не зависит от действий во внешних по отношению к данной подпрограмме частях программы. Проблема с захватом свободных переменных может быть решена с помощью *переименования* связывающих переменных в соответствующих λ-абстракциях. Это решение не противоречит нашей интерпретации, поскольку переименование формального параметра подпрограммы с одновременной заменой имени параметра в её теле не приводит к изменению смысла ни данной подпрограммы, ни программы в целом.

Следующее определение полностью отражает предпринятые решения.

Операцией подстановки терма N в терм P вместо переменной x, обозначаемой как

[1] В разных источниках используются абсолютно разные обозначения операции подстановки, поэтому при их чтении следует убедиться, что смысл обозначения понят корректно.

$[N/x]P$, называется операция преобразования терма, выполняемая по правилам:

$$\begin{array}{lcll}
[N/x]x & = & (N) & \\
[N/x]y & = & y, & \text{если } x \neq y \\
[N/x](PQ) & = & (([N/x]P)([N/x]Q)) & \\
[N/x](\lambda x\,.\,P) & = & (\lambda x\,.\,P) & \\
[N/x](\lambda y\,.\,P) & = & (\lambda y\,.\,P), & \text{если } x \notin FV(P) \\
[N/x](\lambda y\,.\,P) & = & (\lambda y\,.\,([N/x]P)), & \text{если } x \in FV(P) \text{ и } y \notin FV(N) \\
[N/x](\lambda y\,.\,P) & = & (\lambda z\,.\,([N/x]([z/y]P))), & \text{если } x \in FV(P) \text{ и } y \in FV(N), \\
 & & & \text{причём } z \notin FV(NP)
\end{array}$$

Вместо двух последних правил неправильного определения в правильном появились четыре новых правила, первое из которых решает проблему исчезновения связанных переменных, а три последних — проблему захвата свободных. Протестируем новое определение на старых примерах:

$$[t/x](\lambda x\,.\,x) = \lambda x\,.\,x,$$
$$[y/x](\lambda y\,.\,x) = \lambda z\,.\,([y/x]([z/y]x)) = \lambda z\,.\,([y/x]x) = \lambda z\,.\,y.$$

Действительно, теперь тождественная функция остаётся тождественной, а константная — константной, правда, с точностью до имени связывающей переменной.

Замена терма $\lambda x\,.\,M$ на терм $\lambda y\,.\,[y/x]M$ (то есть переименование связанной переменной) называется *α-конверсией*. Два терма P и Q, один из которых можно получить из другого с помощью конечного числа α-конверсий, называются *α-эквивалентными*, что обозначается как $P \equiv_\alpha Q$.

Например, следующие термы *α-эквивалентны*:

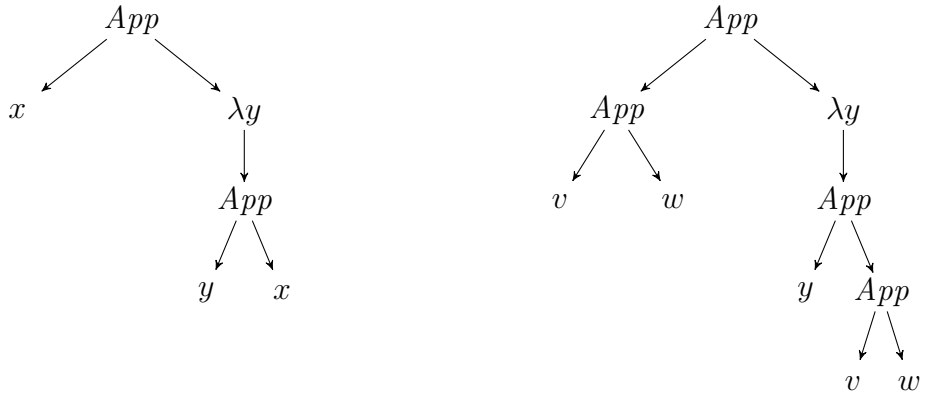
$$\begin{array}{lcl}
\lambda x\,.\,x & \equiv_\alpha & \lambda y\,.\,y; \\
y\,\lambda y\,.\,y & \equiv_\alpha & y\,\lambda z\,.\,z; \\
\lambda x\,.\,(\lambda y\,.\,x) & \equiv_\alpha & \lambda y\,.\,(\lambda x\,.\,y).
\end{array}$$

Убедиться в *α-эквивалентности* последней пары термов можно, переименовав последовательно x в t, y в x и t в y.

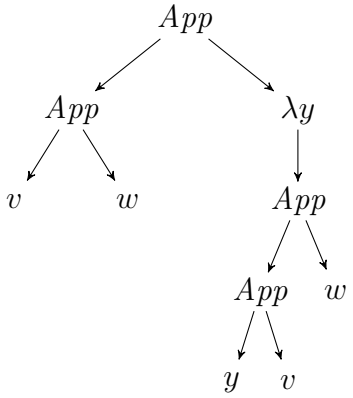
Рассмотрим пример выполнения подстановки, обращая внимание на некоторые трудности, которые при этом могут возникать:

$$[vw/x](x\lambda y\,.\,yx) = vw\lambda y\,.\,y(vw)$$

Заметьте, что в результирующем выражении вокруг второго вхождения vw появились скобки. Они не требовались, когда в соответствующей позиции находилась переменная x, но стали нужны теперь, когда её заменили на применение. Подстановка не должна нарушать структуру терма: поддерево дерева разбора, соответствующее заменяемой переменной, изменяется, но всё, что находится вне его, никак изменяться не должно (на рисунке слева дерево разбора до подстановки, справа — после):



Стоит сравнить полученное в этом примере в результате подстановки дерево разбора с деревом, соответствующим λ-терму $vw\lambda y\,.\,yvw$, который мог бы *ошибочно* получиться, если забыть скобки:

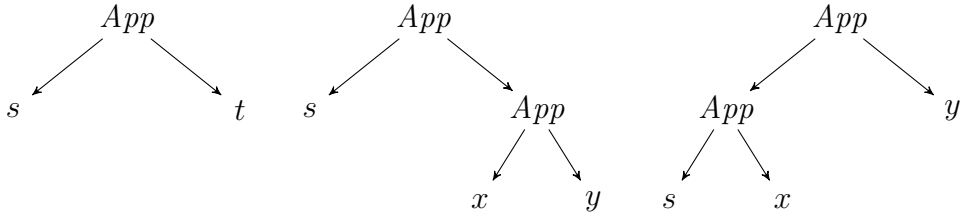


Это дерево соответствует терму $vw\lambda y\,.\,(yv)w$, который не мог получиться в результате подстановки.

Ещё один потенциально опасный пример на ту же тему:

$$[xy/t]st = s(xy) \neq sxy = (sx)y$$

и три дерева разбора — исходное, после подстановки и в результате некорректно выполненной подстановки с забытыми скобками:



2.2.4. Редукция термов

Термы вида $(\lambda x\,.\,M)N$ называются *редексами* или *редуцируемыми выражениями* (reducible expression).

Если считать λ-абстракцию функцией, а применение — вызовом функции, то редекс представляет собой вызов функции, явно заданной абстракцией. Выполнив такой вызов, то есть подставив фактический параметр на место формального, можно получить результат вызова:

$$[N/x]M.$$

Отношение, которое ставит в соответствие редексу $(\lambda x\,.\,M)N$ терм $[N/x]M$ называется *β-редукцией*. Если терм P содержит в качестве своего фрагмента редекс $(\lambda x\,.\,M)N$, который сводится (редуцируется) к терму $[N/x]M$, причём после замены редекса на этот терм получается терм P', то будем писать, что

$$P \to_\beta P'.$$

Если терм P' можно получить из терма P с помощью конечного числа шагов β-редукции, будем обозначать этот факт как

$$P \to_\beta^* P'.$$

Рассмотрим несколько примеров:

1. $(\lambda x\,.\,x)\,y \to_\beta y$;
2. $(\lambda x\,.\,y)\,t \to_\beta y$;
3. $(\lambda x\,.\,x(xy))N \to_\beta N(Ny)$.

Говорят, что λ-терм находится в *нормальной форме*, если он не содержит ни одного редекса. Для приведения терма в нормальную форму, вообще говоря, может потребоваться несколько шагов редукции:

$$\begin{array}{ll}(\lambda x\,.\,\lambda y\,.\,yx)\,z\,v & \to_\beta ([z/x](\lambda y\,.\,yx))\,v = (\lambda y\,.\,yz)\,v \\ & \to_\beta [v/y](yz) = vz; \\ (\lambda x\,.\,\lambda y\,.\,x)(\lambda z\,.\,z)\,t & \to_\beta ([\lambda z\,.\,z/x](\lambda y\,.\,x))\,t = (\lambda y\,.\,\lambda z\,.\,z)\,t \\ & \to_\beta [t/y](\lambda z\,.\,z) = \lambda z\,.\,z.\end{array}$$

Иногда в терме одновременно присутствуют несколько редексов. В этой ситуации можно выбирать, какой из них редуцировать. Рассмотрим такой пример (вычисляемый первым редекс подчёркнут):

$$\begin{array}{ll}(\lambda x\,.\,\underline{(\lambda y\,.\,yx)\,z})\,v & \to_\beta (\lambda x\,.\,[z/y](yx))\,v = (\lambda x\,.\,zx)\,v \\ & \to_\beta [v/x](zx) = zv; \\ \underline{(\lambda x\,.\,(\lambda y\,.\,yx)\,z)\,v} & \to_\beta [v/x]((\lambda y\,.\,yx)\,z) = (\lambda y\,.\,yv)\,z \\ & \to_\beta [z/y](yv) = zv.\end{array}$$

Заметим, что в обоих случаях результат получился одинаковым. Следующий пример показывает, что не все термы имеют нормальную форму:

$$\begin{array}{ll}(\lambda x\,.\,xx)(\lambda x\,.\,xx) & \to_\beta [\lambda x\,.\,xx/x](xx) = (\lambda x\,.\,xx)(\lambda x\,.\,xx) \\ & \to_\beta [\lambda x\,.\,xx/x](xx) = (\lambda x\,.\,xx)(\lambda x\,.\,xx) \\ & \to_\beta \ldots\end{array}$$

Более того, некоторые термы после «редуцирования» только увеличиваются, например:

$$\begin{aligned}(\lambda x\,.\,xxy)(\lambda x\,.\,xxy) &\to_\beta [\lambda x\,.\,xxy/x](xxy) = (\lambda x\,.\,xxy)(\lambda x\,.\,xxy)y\\ &\to_\beta ([\lambda x\,.\,xxy/x](xxy))y = (\lambda x\,.\,xxy)(\lambda x\,.\,xxy)yy\\ &\to_\beta \dots\end{aligned}$$

С помощью этого «расширяющегося» терма можно заметить интересный эффект. Рассмотрим терм вида:

$$(\lambda y\,.\,v)((\lambda x\,.\,xxy)(\lambda x\,.\,xxy))$$

Ясно, что с одной стороны (вычисляемый в данный момент редекс подчёркнут):

$$\underline{(\lambda y\,.\,v)((\lambda x\,.\,xxy)(\lambda x\,.\,xxy))} \to_\beta v,$$

а с другой

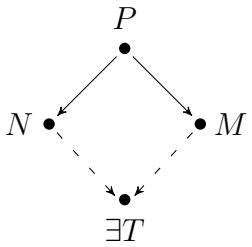
$$\begin{aligned}(\lambda y\,.\,v)(\underline{(\lambda x\,.\,xxy)(\lambda x\,.\,xxy)}) &\to_\beta (\lambda y\,.\,v)(\underline{(\lambda x\,.\,xxy)(\lambda x\,.\,xxy)}y)\\ &\to_\beta (\lambda y\,.\,v)(\underline{(\lambda x\,.\,xxy)(\lambda x\,.\,xxy)}yy)\\ &\to_\beta \dots\end{aligned}$$

Таким образом, приход к нормальной форме в результате редукции зависит от порядка шагов редукции: «неправильная» последовательность редукции может никогда не привести к нормальной форме, даже если она существует.

Возникает вопрос: всегда ли в конечном итоге будет получаться одна и та же нормальная форма (при условии, что она вообще будет получаться), то есть единственна ли она. Ответ на этот вопрос даёт теорема Чёрча—Россера, одна из важнейших теорем λ-исчисления.

Теорема (Чёрча—Россера)**.** *Если $P \to_\beta^* N$ и $P \to_\beta^* M$, то существует такой терм T, что $N \to_\beta^* T$ и $M \to_\beta^* T$.*

Доказательство теоремы Чёрча—Россера выходит за пределы нашего курса. Сформулированное в теореме свойство β-редукции иногда называется свойством Чёрча—Россера или свойством ромба, поскольку графически его можно изобразить так:



Пользуясь этой теоремой, можно, наконец, ответить на вопрос о единственности нормальной формы λ-термов.

Следствие. *Если терм имеет хотя бы одну нормальную форму, то все его нормальные формы совпадают с точностью до α-эквивалентности.*

2.2.5. Стратегии редукции

Так как существование нормальной формы не гарантируется, а к тому же при некоторых порядках редукции нормальная форма может вообще не возникнуть, даже если она существует, необходимо сформулировать правила, позволяющие редуцировать выражения с наибольшей вероятностью успеха. Порядок выбора вычисляемого редекса обычно называют *стратегией редукции.*

Стратегия редукции, при которой на каждом шаге выбирается самый левый, самый внешний редекс, называется *нормальным порядком* редукции. Левизна редекса определяется по символу λ составляющей его абстракции, а выражение «самый внешний» означает, что редекс не является частью никакого другого редекса.

Теорема. *Нормальный порядок вычислений всегда приводит к нормальной форме, при условии, что она существует.*

На практике могут также использоваться две следующие модификации нормального порядка редукции. Стратегией редукции с *вызовом по имени* называется порядок, при котором на каждом шаге выбирается самый левый, самый внешний редекс, но редукция внутри абстракций не производится. Можно также определить стратегию с *вызовом по необходимости.* В её рамках при первом использовании аргумента вычисляется его значение (нередуцируемая далее форма), а все остальные его вхождения заменяются на это значение.

Стратегией редукции с *вызовом по значению* называется порядок редукции, при котором на каждом шаге выбираются только внешние редексы, причём их правые части предварительно вычисляются. Стратегию с вызовом по значению иногда называют *аппликативной.* Говорят, что стратегия с вызовом по значению является *строгой* в том смысле, что значения аргументов всегда вычисляются независимо от того, используются они в теле функции или нет. При *нестрогих* (или *ленивых*) вычислениях — с использованием вызова по имени или по необходимости — вычисляются только те аргументы, которые действительно используются.

Следует заметить, что все упомянутые стратегии редукции, кроме нормального порядка, не производят редукцию внутри абстракций. Они приводят терм к так называемой *слабой (головной) нормальной форме*, т.е. форме, не содержащей внешних редексов.

2.2.6. Представление данных в λ-исчислении

λ-исчисление можно рассматривать как простой язык программирования. Далее нашей задачей будет кодирование базовых элементов любого языка программирования в терминах λ-исчисления. В числе таких элементов будут логические значения и операции, натуральные числа, пары значений, условные операции и рекурсивные вызовы.

Для начала введём логические значения $true$ и $false$:

$$true = \lambda x\,.\,\lambda y\,.\,x; \quad false = \lambda x\,.\,\lambda y\,.\,y.$$

Смысл знака $=$ в этих выражениях заключается в следующем: мы *договариваемся* обозначать термы, стоящие в правой части равенства, символами из левой части. Вообще говоря, назвать истиной и ложью можно любые термы, но такой выбор

обозначений оказался наиболее удобным. Истинное значение можно трактовать как функцию двух аргументов, возвращающую первый из них, а ложное значение — как возвращающую второй.

Используя эти определения, можно ввести *условное выражение*:

$$\text{if } C \text{ then } E_1 \text{ else } E_2 = CE_1E_2.$$

Действительно, вычислим:

$$\text{if } true \text{ then } E_1 \text{ else } E_2 = true\ E_1E_2 = (\lambda x\,.\,\lambda y\,.\,x)E_1E_2 = E_1.$$

С другой стороны:

$$\text{if } false \text{ then } E_1 \text{ else } E_2 = false\ E_1E_2 = (\lambda x\,.\,\lambda y\,.\,y)E_1E_2 = E_2.$$

Пользуясь условной операцией, нетрудно определить отрицание, конъюнкцию и дизъюнкцию:

$$\begin{array}{lcll} \text{not } P & = & \text{if } P \text{ then } false \text{ else } true & = & P\ false\ true; \\ P \text{ and } Q & = & \text{if } P \text{ then } Q \text{ else } false & = & P\ Q\ false; \\ P \text{ or } Q & = & \text{if } P \text{ then } true \text{ else } Q & = & P\ true\ Q. \end{array}$$

Данные могут объединяться в структуры данных, простейшей такой структурой является пара:

$$(E_1, E_2) = \lambda z\,.\,z\ E_1\ E_2.$$

Для работы с парами необходимы функции доступа к отдельным компонентам:

$$\begin{array}{lcl} \text{fst } P & = & P\ true; \\ \text{snd } P & = & P\ false. \end{array}$$

Убедимся в работоспособности этих функций:

$$\begin{array}{lcl} \text{fst } (E_1, E_2) & = & (\lambda z\,.\,z\ E_1\ E_2)\,true = true\ E_1\ E_2 = E_1; \\ \text{snd } (E_1, E_2) & = & (\lambda z\,.\,z\ E_1\ E_2)\,false = false\ E_1\ E_2 = E_2. \end{array}$$

Наконец, натуральные числа можно ввести так:

$$\begin{array}{rcl} 0 & = & \lambda x\,.\,\lambda y\,.\,y \\ 1 & = & \lambda x\,.\,\lambda y\,.\,xy \\ 2 & = & \lambda x\,.\,\lambda y\,.\,x(xy) \\ 3 & = & \lambda x\,.\,\lambda y\,.\,x(x(xy)) \\ & \ldots & \end{array}$$

Натуральные числа, представленные таким образом, называют *нумералами Чёрча*. Нумерал Чёрча можно интерпретировать как функцию двух переменных, которая применяет свой первый аргумент к второму соответствующее число раз, например:

$$\begin{array}{rcl} 2\ f\ t & = & (\lambda x\,.\,\lambda y\,.\,x(xy))\,f\ t \\ & = & (\lambda y\,.\,f(fy))\,t \\ & = & f(ft). \end{array}$$

Можно ввести *функцию следования*, которая ставит в соответствие каждому натуральному числу следующее за ним ($n \mapsto n+1$):

$$\text{succ} = \lambda n \,.\, \lambda x \,.\, \lambda y \,.\, x(nxy).$$

Вычислим число, следующее за числом 3:

$$\begin{aligned} \text{succ } 3 &= (\lambda n \,.\, \lambda x \,.\, \lambda y \,.\, x(nxy))(\lambda x \,.\, \lambda y \,.\, x(x(xy))) \\ &= \lambda x \,.\, \lambda y \,.\, x\,((\lambda x \,.\, \lambda y \,.\, x(x(xy)))xy) \\ &= \lambda x \,.\, \lambda y \,.\, x\,((\lambda y \,.\, x(x(xy)))y) \\ &= \lambda x \,.\, \lambda y \,.\, x\,(x(x(xy))) \\ &= 4. \end{aligned}$$

Заметим, что такое определение соответствует нашей интерпретации: терм, находящийся в скобках (nxy) означает n-кратное применение x к y, после этого следует еще одно применение x к полученному ранее результату. В итоге получается, что x применяется к y в точности $n+1$ раз. В целом, процесс вычислений можно трактовать как $(n+1)$-кратное прибавление единицы к нулю. Пользуясь той же интуицией, можно сформулировать и другое определение функции следования, прибавляя n раз единицу к единице:

$$\text{succ}' = \lambda n \,.\, \lambda x \,.\, \lambda y \,.\, nx(xy).$$

Связь между натуральными числами и логическими значениями можно организовать, определив функцию, проверяющую, является ли заданное число нулём:

$$\text{is\_zero?} = \lambda n \,.\, n\,(\lambda x \,.\, false)\,true.$$

Действительно:

$$\begin{aligned} \text{is\_zero? } 0 &= (\lambda n \,.\, n\,(\lambda x \,.\, false)\,true)\,(\lambda x \,.\, \lambda y \,.\, y) \\ &= (\lambda x \,.\, \lambda y \,.\, y)(\lambda x \,.\, false)\,true \\ &= (\lambda y \,.\, y)\,true \\ &= true; \\ \text{is\_zero? } 1 &= (\lambda n \,.\, n\,(\lambda x \,.\, false)\,true)\,(\lambda x \,.\, \lambda y \,.\, xy) \\ &= (\lambda x \,.\, \lambda y \,.\, xy)(\lambda x \,.\, false)\,true \\ &= (\lambda y \,.\, (\lambda x \,.\, false)\,y)\,true \\ &= (\lambda x \,.\, false)\,true \\ &= false. \end{aligned}$$

Аналогично, для всех ненулевых n функция $is\_zero?$ возвращает $false$.

Функции сложения и умножения чисел можно определить следующим образом:

$$\begin{aligned} m+n &= \lambda x \,.\, \lambda y \,.\, mx(nxy); \\ m*n &= \lambda x \,.\, \lambda y \,.\, m(nx)y. \end{aligned}$$

Возможность проверить справедливость этих определений на примерах предоставляется читателю, но ясно, что интуитивная интерпретация сохраняется: чтобы сложить m и n, нужно n раз прибавить единицу к нулю (применить x к y), а затем прибавить к полученному результату единицу еще m раз; чтобы умножить m на n, достаточно m раз прибавить n единиц к нулю.

Интересно, что *функция предшествования* pred и операция вычитания определяются гораздо более сложным образом. Идея этого определения принадлежит ученику Чёрча Стивену Клини (1909—1994)[2]. Суть его идеи заключается в использовании вспомогательной функции, которая любой паре вида (m, n), где m и n — нумералы Чёрча, ставит в соответствие пару $(n, n+1)$. Определить такую функцию нетрудно:

$$\text{step} = \lambda p \,.\, (\text{snd}\ p, \text{succ}(\text{snd}\ p)).$$

Здесь используется введённое ранее обозначение пары, функции доступа к второму элементу пары и функция следования, однако понятно, что определение можно записать и непосредственно в виде λ-терма.

Теперь, для того чтобы найти число, предшествующее заданному m, достаточно m раз применить функцию step к паре $(0, 0)$, после этого первый компонент получившейся пары будет равен в точности $m - 1$:

$$\text{pred} = \lambda m \,.\, \text{fst}(m\ \text{step}\ (0, 0)).$$

Будучи применённой к нулю, функция pred даёт ноль. Пользуясь этой функцией, можно определить вычитание:

$$m - n = n\ \text{pred}\ m.$$

2.2.7. Комбинаторы неподвижной точки и рекурсия

Очередной шаг — введение такого важного элемента языка программирования, как рекурсия. Однако здесь есть некоторая сложность: обычно для вызова функции внутри её тела используется её же имя, но λ-абстракции имени не имеют. Тем не менее, организовать рекурсивные вызовы возможно. Это обеспечивается существованием *комбинаторов неподвижной точки*.

Терм называется *замкнутым* (или *комбинатором*), если он не содержит ни одной свободной переменной.

Комбинатором неподвижной точки называется такой замкнутый терм Y, для которого при любом f выполняется равенство:

$$f(Yf) = Yf.$$

Название комбинатора объясняется тем, что он находит такую точку из области определения функции f, которая переводится этой функцией сама в себя, то есть является неподвижной.

Существует достаточно большое количество комбинаторов неподвижной точки, одним из них является комбинатор Тьюринга:

$$Y_T = AA, \text{ где } A = \lambda x \,.\, \lambda y \,.\, y(xxy).$$

Проверим, что комбинатор Тьюринга действительно является комбинатором неподвижной точки:

$$\begin{aligned} Y_T f &= AAf \\ &= (\lambda x \,.\, \lambda y \,.\, y(xxy))Af \\ &= (\lambda y \,.\, y(AAy))f \\ &= f(AAf) \\ &= f(Y_T f). \end{aligned}$$

[2]Определение функции предшествования занимало мысли Чёрча в течение нескольких месяцев, над этой же задачей работал и его ученик.

Теперь, следуя давней традиции, запрограммируем рекурсивную функцию вычисления факториала числа. Функцию fact хотелось бы определить следующим образом:

$$\text{fact } n = \text{if is\_zero? } n \text{ then } 1 \text{ else } n * \text{fact}(\text{pred } n)$$

Или, что то же самое, как

$$\text{fact} = \lambda n \,.\, \text{if is\_zero? } n \text{ then } 1 \text{ else } n * \text{fact}(\text{pred } n)$$

Очевидная проблема этого определения — присутствие fact в теле функции. Чтобы его избежать, определим факториал как функцию от функции:

$$\text{fact} = (\lambda f \,.\, \lambda n \,.\, \text{if is\_zero? } n \text{ then } 1 \text{ else } n * f(\text{pred } n))\text{fact}$$

Обозначим выражение в скобках через H:

$$H = \lambda f \,.\, \lambda n \,.\, \text{if is\_zero? } n \text{ then } 1 \text{ else } n * f(\text{pred } n)$$

Ясно, что H — это вполне законный λ-терм, а именно, λ-абстракция. Теперь видно, что

$$\text{fact} = H \text{ fact}$$

Последнее равенство означает, что функция fact является неподвижной точкой абстракции H, а значит, её можно найти с помощью комбинатора неподвижной точки[3]:

$$\text{fact} = Y_T H.$$

Попробуем теперь для проверки вычислить fact 3, несколько раз воспользовавшись тем фактом, что $Y_T H = H(Y_T H)$:

$$\begin{aligned}
\text{fact } 3 &= (Y_T H)\ 3 = H(Y_T H)\ 3 \\
&= (\lambda f \,.\, \lambda n \,.\, \text{if is_zero? } n \text{ then } 1 \text{ else } n * f(\text{pred } n))(Y_T H)\ 3 \\
&= \text{if is_zero? } 3 \text{ then } 1 \text{ else } 3 * ((Y_T H)(\text{pred } 3)) \\
&= 3 * ((Y_T H)\ 2) = 3 * (H(Y_T H)\ 2) \\
&= 3 * ((\lambda f \,.\, \lambda n \,.\, \text{if is_zero? } n \text{ then } 1 \text{ else } n * f(\text{pred } n))(Y_T H)\ 2) \\
&= 3 * (\text{if is_zero? } 2 \text{ then } 1 \text{ else } 2 * ((Y_T H)(\text{pred } 2))) \\
&= 3 * (2 * ((Y_T H)\ 1)) = 3 * (2 * (H(Y_T H)\ 1)) \\
&= 3 * (2 * ((\lambda f \,.\, \lambda n \,.\, \text{if is_zero? } n \text{ then } 1 \text{ else } n * f(\text{pred } n))(Y_T H)\ 1)) \\
&= 3 * (2 * (\text{if is_zero? } 1 \text{ then } 1 \text{ else } 1 * ((Y_T H)(\text{pred } 1)))) \\
&= 3 * (2 * (1 * ((Y_T H)\ 0))) = 3 * (2 * (1 * (H(Y_T H)\ 0))) \\
&= 3 * (2 * (1 * ((\lambda f \,.\, \lambda n \,.\, \text{if is_zero? } n \text{ then } 1 \text{ else } n * f(\text{pred } n))(Y_T H)\ 0))) \\
&= 3 * (2 * (1 * ((\text{if is_zero? } 0 \text{ then } 1 \text{ else } 0 * ((Y_T H)(\text{pred } 0)))))) \\
&= 3 * (2 * (1 * 1)) = 3 * (2 * 1) = 3 * 2 = 6.
\end{aligned}$$

[3]Здесь мы неявно пользуемся тем фактом, что неподвижная точка является единственной, и комбинатор Тьюринга вычисляет именно её. Вообще говоря, соответствующий факт требует доказательства, он выводится в рамках теории порядка и составляет содержание известной теоремы Клини о неподвижной точке.

2.2.8. Локальные объявления

В принципе, язык программирования на основе λ-исчисления построен. Любая программа на этом языке — это λ-абстракция. Запуск программы означает, что этой абстракции передаётся в качестве параметра исходное значение. Результат редукции — результат выполнения программы. Таким результатом оказывается некоторое выражение в слабой нормальной форме. Чтобы программирование на этом языке стало еще более удобным, добавим возможность создания локальных объявлений, то есть способ именования вспомогательных конструкций.

Введём в рассмотрение следующее обозначение:

$$(\text{let } x = s \text{ in } t) = (\lambda x \,.\, t)\, s$$

и приведём простой пример его использования:

$$\begin{array}{rcl} (\text{let } z = 2 + 3 \text{ in } z * z) & = & (\lambda z \,.\, z * z)\,(2 + 3) \\ & = & (2 + 3) * (2 + 3) \end{array}$$

Конструкцию let/in можно использовать последовательно:

$$\text{let } x = 1 \text{ in } (\text{let } x = 2 \text{ in } (\text{let } y = x \text{ in } x + y)) = 4$$

Можно также определить параллельное объявление нескольких имён:

$$\begin{array}{l} \text{let} \\ \quad x_1 = s_1 \\ \quad x_2 = s_2 \\ \quad \ldots \\ \quad x_n = s_n \\ \text{in } t = (\lambda x_1 \,.\, \lambda x_2 \,.\, \ldots \lambda x_n \,.\, t)\, s_1 s_2 \ldots s_n \end{array}$$

Приведём пример его использования:

$$\begin{array}{l} \text{let } x = 1 \text{ in} \\ \qquad\qquad \begin{array}{l} \text{let} \\ \quad x = 2 \\ \quad y = x \\ \text{in } x + y = 3 \end{array} \end{array}$$

2.2.9. Комбинаторная логика

Рассматривавшиеся ранее λ-термы строились из трёх типов компонентов — переменных, применений и абстракций. Оказывается, от абстракций (и, соответственно, связанных переменных) можно отказаться, ограничившись небольшим набором базовых преобразований. Соответствующая система называется *комбина́торной логикой*, а упомянутые базовые преобразования — комбинаторами. С комбинаторами связаны правила редукции, это необходимо, поскольку в отсутствие абстракций старые определения работать перестают.

Введём в рассмотрение базовые комбинаторы **I**, **K** и **S**, определяемые следующими правилами редукции:

- $\mathbf{I}x = x$;
- $\mathbf{K}xy = x$;
- $\mathbf{S}xyz = xz(yz)$.

Комбинатор $\mathbf{I}$ называют тождественным, комбинатор $\mathbf{K}$ — константным (он всегда возвращает свой первый параметр, игнорируя второй), комбинатор $\mathbf{S}$ — распределяющим. Тождественный комбинатор $\mathbf{I}$, вообще говоря, является лишним, поскольку по своему действию он эквивалентен комбинатору $\mathbf{SKK}$. Действительно,

$$(\mathbf{SKK})x = \mathbf{SKK}x = \mathbf{K}x(\mathbf{K}x) = x.$$

Существует относительно простая процедура, позволяющая преобразовать произвольный λ-терм в форму без абстракций, содержащую только комбинаторы $\mathbf{S}$ и $\mathbf{K}$. Обозначим такое преобразование символом $T[\cdot]$ и опишем его с помощью шести правил:

1. $T[x] \Rightarrow x$;
2. $T[(MN)] \Rightarrow (T[M]\ T[N])$;
3. $T[\lambda x\,.\,M] \Rightarrow (\mathbf{K}T[M])$ (если x не входит свободно в M);
4. $T[\lambda x\,.\,x] \Rightarrow \mathbf{I}$;
5. $T[\lambda x\,.\,\lambda y\,.\,M] \Rightarrow T[\lambda x\,.\,T[\lambda y\,.\,M]]$ (если x входит свободно в M);
6. $T[\lambda x\,.\,(MN)] \Rightarrow (\mathbf{S}T[\lambda x\,.\,M]T[\lambda x\,.\,N])$ (если x входит свободно в M или N).

Воспользуемся этими правилами для преобразования в комбинаторную форму терма $\lambda x\,.\,\lambda y\,.\,yx$:

$$\begin{aligned}
T[\lambda x\,.\,\lambda y\,.\,yx] &= \\
&= T[\lambda x\,.\,T[\lambda y\,.\,(yx)]] \quad \text{(правило 5)} \\
&= T[\lambda x\,.\,(\mathbf{S}T[\lambda y\,.\,y]T[\lambda y\,.\,x])] \quad \text{(правило 6)} \\
&= T[\lambda x\,.\,(\mathbf{SI}T[\lambda y\,.\,x])] \quad \text{(правило 4)} \\
&= T[\lambda x\,.\,(\mathbf{SI}(\mathbf{K}x))] \quad \text{(правила 3 и 1)} \\
&= (\mathbf{S}T[\lambda x\,.\,(\mathbf{SI})]T[\lambda x\,.\,(\mathbf{K}x)]) \quad \text{(правило 6)} \\
&= (\mathbf{S}(\mathbf{K}(\mathbf{SI}))T[\lambda x\,.\,(\mathbf{K}x)]) \quad \text{(правило 3)} \\
&= (\mathbf{S}(\mathbf{K}(\mathbf{SI}))(\mathbf{S}T[\lambda x\,.\,\mathbf{K}]T[\lambda x\,.\,x])) \quad \text{(правило 6)} \\
&= (\mathbf{S}(\mathbf{K}(\mathbf{SI}))(\mathbf{S}(\mathbf{KK})T[\lambda x\,.\,x])) \quad \text{(правило 3)} \\
&= (\mathbf{S}(\mathbf{K}(\mathbf{SI}))(\mathbf{S}(\mathbf{KK})\mathbf{I})) \quad \text{(правило 4)}
\end{aligned}$$

Проверим, действительно ли полученный терм ведёт себя в точности так же, что и исходная абстракция:

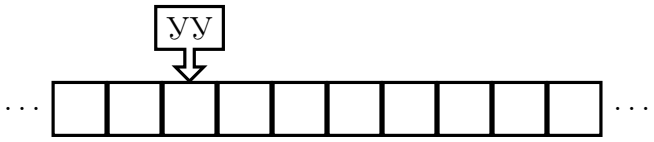
$$\begin{aligned}
(\mathbf{S}(\mathbf{K}(\mathbf{SI}))(\mathbf{S}(\mathbf{KK})\mathbf{I})xy) &= (\mathbf{K}(\mathbf{SI})x(\mathbf{S}(\mathbf{KK})\mathbf{I}x)y) \\
&= (\mathbf{SI}(\mathbf{S}(\mathbf{KK})\mathbf{I}x)y) = (\mathbf{I}y(\mathbf{S}(\mathbf{KK})\mathbf{I}xy)) \\
&= (y(\mathbf{S}(\mathbf{KK})\mathbf{I}xy)) = (y(\mathbf{KK}x(\mathbf{I}x)y)) \\
&= (y(\mathbf{K}(\mathbf{I}x)y)) = (y(\mathbf{I}x)) = (yx)
\end{aligned}$$

Преобразованный терм оказался значительно больше, чем исходный, но это и понятно: мы имеем дело с более примитивными конструкциями, а значит, для выражения действия оригинального терма этих конструкций требуется больше. Мы не будем углубляться далее в комбинаторную логику, нам достаточно того факта, что в форму с комбинаторами можно преобразовать любой λ-терм, то есть соответствующая модель вычислений эквивалентна λ-исчислению, а значит, для описания любого вычисления нам достаточно переменных, применений и всего лишь двух базовых комбинаторов **K** и **S** (мы помним, что комбинатор **I** через них выражается).

2.3. Машины Тьюринга

Машина Тьюринга — это мысленное устройство, состоящее из двух компонентов:

- бесконечная лента, разделённая на клетки, в каждой из которых записана одна буква некоторого алфавита;
- устройство управления, снабжённое считывающей головкой, позволяющей читать и записывать содержимое одной (текущей) клетки ленты.



Устройство управления в каждый момент времени находится в одном из конечного множества *состояний*. Считывающая головка может перемещаться вдоль ленты влево или вправо.

Один шаг работы машины Тьюринга (*переход*) состоит в чтении текущего символа ленты, записи текущего символа, изменении состояния и перемещении на одну позицию влево или вправо (оставаться на месте нельзя). Последние три действия выполняются в зависимости от прочитанного символа и текущего состояния. Эта зависимость задаётся *функцией переходов*.

В начале работы на ленте записывается *входное слово* (конечной длины), все остальные клетки заполнены *пробельными символами*. Некоторые состояния считаются *финальными* или *допускающими*, при их достижении машина Тьюринга останавливается. Если функция переходов не определена на прочитанном символе при текущем состоянии, то машина Тьюринга также останавливается.

Формально машина Тьюринга может быть представлена семёркой

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F),$$

компоненты которой имеют следующий смысл:

- $Q = \{q_0, q_1 \dots q_n\}$ — конечное множество состояний устройства управления;
- Σ — входной алфавит, символами которого записывается входное слово;
- Γ — ленточный алфавит, включающий в себя входной алфавит, пробельный символ и, возможно, другие символы, требуемые машине во время работы;

- $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$ — функция переходов, определяющая по текущему состоянию и символу на ленте новое состояние, записываемый на ленту символ и сдвиг (влево или вправо);
- $q_0 (\in Q)$ — начальное состояние;
- $B (\in \Gamma)$ — пробельный символ;
- $F (\subset Q)$ — множество финальных (допускающих) состояний, при достижении которых машина останавливается.

Наиболее существенным компонентом семёрки является функция переходов, задающая по сути программу для работы машины Тьюринга. Её можно определить просто перечнем значений на всех возможных аргументах, но удобнее использовать один из следующих способов: задание таблицей и диаграммой.

Рассмотрим в качестве примера машину Тьюринга

$$M = (\{q_0, q_1\}, \{0, 1\}, \{0, 1, B\}, \delta, q_0, B, \{q_1\}),$$

функция переходов которой задана таблицей:

δ	B	1	0
q_0	—	$(q_0, 1, R)$	$(q_1, 1, R)$
q_1	—	—	—

При встрече пробельного символа машина всегда останавливается (соответствующие значения функции переходов не определены), при встрече единицы в состоянии q_0 считывающая головка перемещается вправо. При встрече нуля в состоянии q_0 ноль заменяется на единицу, считывающая головка перемещается вправо, устройство управления переходит в допускающее состояние q_1 и работа машины завершается.

Функция переходов этой машины Тьюринга может быть также представлена следующей диаграммой:

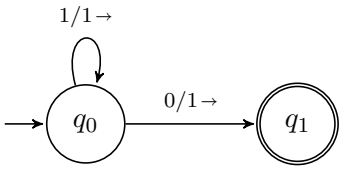


Диаграмма позволяет обозначить начальное состояние (отмечено входящей стрелкой без источника), допускающие состояния (двойная окружность) и переходы (дуга показывает изменение состояния, а в подписях дуг указываются тройки — прочитанный символ / записываемый символ / направление движения). Отсутствие дуги по некоторому символу означает, что соответствующее значение функции переходов не определено.

Машина Тьюринга может рассматриваться как *распознаватель* некоторого языка L, если она завершает работу в допускающем состоянии, будучи запущенной на тех и только на тех словах, которые принадлежат этому языку. Соответствующий язык называют языком данной машины Тьюринга и пишут $M = M(L)$. Говорят также, что машина допускает язык и что слово допускается машиной. Нетрудно увидеть,

что языком машины Тьюринга из примера является множество всех слов в алфавите $\{0, 1\}$, содержащих хотя бы один ноль (обоснование: встреча нуля есть единственный способ попасть в допускающее состояние q_1).

Если машина Тьюринга решает задачу распознавания языка, то неважно, что остаётся на ленте после её работы, важно лишь, в допускающем ли состоянии она завершает свою работу. Машина-преобразователь, наоборот, предназначена именно для преобразования входного слова, в этом случае допускающие состояния не используются.

Конфигурацией или мгновенным описанием машины Тьюринга называется описание машины во время работы, включающее в себя содержимое ленты, состояние устройства управления и положение считывающей головки. Например, если на ленте записано слово $X_1 \dots X_{i-1} X_i X_{i+1} \dots X_n$, окружённое пробельными символами, машина находится в состоянии q, обозревая при этом символ X_i, то конфигурация будет выглядеть следующим образом:

$$X_1 \dots X_{i-1} q X_i X_{i+1} \dots X_n.$$

Заметим, что пробельные символы слева и справа от слова на ленте не включаются в конфигурацию. Начальная конфигурация в таких обозначениях выглядит как $q_0 X_1 \dots X_n$ (считывающая головка находится слева от входного слова).

Переход от одной конфигурации к другой записывается с помощью знака $\vdash$ или $\vdash_M$, если необходимо явно указать, о какой машине идёт речь. Например, если функция переходов машины Тьюринга M включает правило $\delta(q, X_i) = (p, Y, R)$, то переход из приведённой выше конфигурации будет выглядеть так:

$$X_1 \dots X_{i-1} q X_i X_{i+1} \dots X_n \vdash_M X_1 \dots X_{i-1} Y p X_{i+1} \dots X_n.$$

Договоримся включать пробельный символ B в конфигурацию в ситуациях, когда он оказывается обозреваемым — это может произойти при движении за левый или правый края слова, например: $qBX_1 \dots X_n$ или $X_1 \dots X_n qB$.

Символами $\vdash^*$ и $\vdash_M^*$ будем обозначать несколько переходов, что позволяет дать формальное определение языка машины Тьюринга $M = M(Q, \Sigma, \Gamma, \delta, q_0, B, F)$:

$$L(M) = \{w \in \Sigma^* : \ q_0 w \vdash_M^* \alpha p \beta, \ p \in F, \ \alpha, \beta \in \Gamma^*\},$$

то есть множества всех слов, на которых машина Тьюринга через произвольное число переходов попадает в допускающее состояние, оставляя после себя на ленте любые последовательности символов ленточного алфавита.

Весь остальной материал прочитан очень близко к тексту книги Хопкрофта, Мотвани и Ульмана «Введение в теорию автоматов, языков и вычислений» (2-е издание), глава 8.

Пройденные темы (в скобках указаны номера разделов):

1. Язык машины Тьюринга, рекурсивно-перечислимые и рекурсивные языки, множества языков (8.2.5–6).

2. Вариации конструкции машин Тьюринга: память в состоянии (регистры), составные символы (многодорожечные ленты на примере распознавания языка L_{wcw}) и ограничение алфавита (достаточно ограничиться алфавитом $\Sigma = \{0, 1\}$, закодировав с его помощью все символы любого другого алфавита машины Тьюринга и исправив соответствующим образом состояния и функцию перехода) — 8.3.1–2.

3. Многоленточные машины Тьюринга и их эквивалентность одноленточным, время имитации (8.4.1–3).

4. Забывающие (oblivious) машины Тьюринга. Машина Тьюринга называется забывающей, если перемещение её считывающей головки не зависит от символов, составляющих входное слово, а зависит только от его длины. Возможность представления любой машины Тьюринга в виде забывающей вытекает из доказательства теоремы об эквивалентности многоленточных машин Тьюринга одноленточным: в нём приводится способ представления ленты, при котором считывающая головка двигается в соответствии с определённым паттерном, сначала слева направо, а затем справа налево.

5. Машины Тьюринга с полулентами и построение машин с ограничениями по записи пробелов и движению влево от начального положения (8.5.1).

6. Многоленточные машины со специализированными лентами: для удобства работы с машиной Тьюринга можно строить машины, содержащие полуленту только для чтения (для входного слова), несколько рабочих лент и/или полулент, ленту для вывода результата (только для записи); конкретный набор лент может определяться решаемой задачей.

7. Недетерминированные машины Тьюринга, их эквивалентность детерминированным, время имитации (8.4.4).

2.4. Машины с неограниченными регистрами

2.4.1. Определение, вычислимые функции

Модель машины с неограниченными регистрами (unbounded register machine, URM) была предложена в 1963 году математиками Шефердсоном и Стёрджисом. Целью этой модели является описание понятия вычисления в терминах, близких к современным языкам программирования, одновременно достаточно простых, чтобы относительно неё можно было применять формальные рассуждения.

Алфавитом регистровой машины является следующий набор символов:

$$:\ ,+,\dot{-},\leftarrow,X,0,1,2,3,4,5,6,7,8,9,\text{if},\text{else},\text{goto},\text{stop}$$

Переменные (или регистры) машины — это ячейки с именами $X1, X11, X111, \ldots$, каждая из которых может хранить любое натуральное число. Машина может использовать любое конечное количество таких регистров. Регистры называются неограниченными, поскольку размер хранимого числа ничем не ограничен. Для удобства мы будем обозначать переменные символами $x, y, z, s, t, u, v, w, \ldots$ с индексами и без.

Машина может выполнять инструкции. Каждая инструкция снабжена меткой (натуральное число), для неё определено правило вычисления и правило выбора следующей за ней инструкции. Имеются следующие инструкции (в скобках приводится их семантика и правило перехода):

- $L : x \leftarrow a$
  (запись в регистр x натурального значения a и переход на инструкцию с меткой $L+1$)
- $L : x \leftarrow x+1$
  (увеличение значения регистра x на единицу и переход на инструкцию с меткой $L+1$)
- $L : x \leftarrow x \dot{-} 1$
  (уменьшение значения регистра x на единицу и переход на инструкцию с меткой $L+1$; если в регистре уже содержался ноль, то значение регистра не изменяется);
- $L : \text{stop}$
  (остановка работы машины, можно считать, что правило перехода заключается в переходе на инструкцию с той же самой меткой L);
- $L : \text{if } x = 0 \text{ goto } M \text{ else goto } R$
  (если в регистре x хранится число 0, то перейти на инструкцию с меткой M, в противном случае перейти на инструкцию с меткой R).

Никаких других инструкций не предусмотрено. К примеру, нельзя скопировать значение из одного регистра в другой, или увеличить значение регистра на заданное натуральное число.

Программа для регистровой машины содержит упорядоченный набор инструкций, первая инструкция всегда имеет метку 1. Инструкция $L : \text{stop}$ должна присутствовать в программе в точности один раз.

Регистровая машина может вычислять теоретико-числовые функции вида

$$f : \mathbb{N}^n \longrightarrow \mathbb{N}.$$

Опишем, как это происходит. Пусть требуется вычислить значение $f(a_1, \ldots, a_n)$. Перед началом работы машины проходит инициализация: числа $a_1, \ldots, a_n$ копируются в выбранные регистры, например, $x_1, \ldots, x_n$, все остальные регистры (вспомогательные переменные, используемые во время работы) обнуляются. Предположим, что результат вычислений должен оказаться в регистре y, он также инициализируется

нулём. После инициализации управление передаётся инструкции с меткой 1, далее последовательность выполнения инструкций определяется соответствующими правилами перехода. Если в какой-то момент достигается инструкция stop, то вычисление считается завершённым (этот факт обозначают как $f(a_1, \ldots, a_n) \downarrow$), а вычисленным значением является значение регистра y.

Машину с зафиксированными входными и выходным регистром мы будем обозначать символом $M_y^{x_1,\ldots,x_n}$. Если машина $M_y^{x_1,\ldots,x_n}$ вычисляет значение функции f на любом наборе аргументов, то мы будем писать $f = M_y^{x_1,\ldots,x_n}$.

Функция $f : \mathbb{N}^n \longrightarrow \mathbb{N}$ от переменных $x_1, \ldots, x_n$ называется *частично вычислимой*, если существует такая регистровая машина M, которая вычисляет её значение, то есть $f = M_y^{x_1,\ldots,x_n}$. Множество всех частично вычислимых функций обозначается символом $\mathcal{P}$. Слово «частично» в этом определении означает, что функция вообще говоря может быть не определена на некоторых аргументах (соответствующее вычисление может зацикливаться). Множество всех всюду определённых вычислимых функций (мы будем называть их просто вычислимыми) является подмножеством $\mathcal{P}$ и обозначается символом $\mathcal{R}$.

Слово «вычислимый» мы будем использовать взаимозаменяемо со словом «рекурсивный», смысл синонимичности станет понятен позднее. Таким образом, «частично вычислимая функция» это тоже самое, что и «частично рекурсивная функция», а «вычислимая функция» это то же самое, что и «рекурсивная функция», причём последняя по определению является всюду определённой.

2.4.2. Примеры

Первым примером рассмотрим машину M_x^x, которая увеличивает значение во входном регистре на единицу:

$1 : x \leftarrow x + 1$

$2 : \text{stop}$

Мы можем использовать λ-нотацию, чтобы описать функцию, которая вычисляется этой машиной: $M_x^x = \lambda x.x + 1$. Соответствующая функция называется функцией следования.

Функция предшествования вычисляет предшествующее заданному натуральное число (нулю по определению предшествует ноль), соответствующая машина $M_x^x = \lambda x.x \dot{-} 1$ имеет следующий вид:

$1 : x \leftarrow x \dot{-} 1$

$2 : \text{stop}$

Ещё одна простая функция — *нуль-функция* — всегда возвращает ноль, а её машина $M_x^x = \lambda x.0$ определяется как

$1 : x \leftarrow 0$

$2 : \text{stop}$

Иногда возникает желание расширить набор инструкций, чтобы упростить часто встречающиеся вычисления. При этом необходимо точно определить, как расширение соответствует базовому определению. Например, можно добавить инструкцию

безусловного перехода L : goto L', поскольку она может быть выражена базовой инструкцией:

L : if $x = 0$ goto L' else goto L'

Рассмотрим следующий фрагмент машины:

$k-1 : x \leftarrow 0$
$k : x \leftarrow x + 1$
$k+1 : z \leftarrow z \dot{-} 1$
$k+2$: if $z = 0$ goto $k+3$ else goto k
$k+3 : \ldots$

Нетрудно видеть, что здесь выполняется вычисление, которое можно записать короче, введя новую инструкцию:

$L : x \leftarrow z$ (новая инструкция — копирование регистров!)
$L+1 : z \leftarrow 0$
$L+2 : \ldots$

Неудобством описанной реализации является обнуление регистра z после выполнения копирования. Вообще говоря, можно придумать реализацию, которая таким недостатком обладать не будет, соответствуя просто

$L : x \leftarrow z$

без всяких побочных эффектов (то есть без изменения значения других регистров, кроме x). Будем далее предполагать, что соответствующее расширение уже есть. Это позволит без ограничения общности там, где это удобно, считать, что входные регистры не изменяют своего значения во время работы. Действительно, если в некоторой машине входной регистр изменяется, то её можно модифицировать так, чтобы его значение сразу копировалось во вспомогательную переменную, и вся работа в дальнейшем шла именно с ней. Модифицированная машина будет вычислять ту же функцию, но будет обладать свойством неизменяемости входных регистров.

Последним примером определим набор машин, вычисляющих функции-проекторы, то есть функции, возвращающие один из своих аргументов. Для этого нам понадобится новое обозначение. Символом $\bar{x}$ будем обозначать набор произвольного количества аргументов, а символом $\bar{x}_n$ — набор из n аргументов. Тогда $\lambda \bar{x}_n . x_i$, где $1 \leqslant i \leqslant n$ — это функция n аргументов, возвращающая i-й. Те же обозначения мы будем использовать и для описания входных регистров машин.

Итак, определим машину $M_z^{\bar{w}_n} = \lambda \bar{x}_n . x_i$:

$1 : w_1 \leftarrow 0$
$\ldots$
$i : z \leftarrow w_i$
$\ldots$
$n : w_n \leftarrow 0$
$n+1$: stop

Мы намеренно упоминаем в тексте программы все входные переменные, чтобы явно указать на факт их использования. Заметим, что эта машина не удовлетворяет свойству неизменяемости входных регистров, но в данном примере это и не требуется. Результат вычислений, как и положено, оказывается в регистре z.

2.4.3. Операторы композиции, примитивной рекурсии и неограниченного поиска

Убедимся теперь, что регистровые машины можно комбинировать друг с другом для реализации более сложных, чем в предыдущих примерах, вычислений.

Предположим, что у нас имеется две частично вычислимые функции $\lambda x\bar{y}.f(x,\bar{y})$ и $\lambda\bar{z}.g(\bar{z})$, причём $f = F_u^{x,\bar{y}}$, а $g = G_x^{\bar{z}}$. Опишем вычисление вида $\lambda\bar{y}\bar{z}.f(g(\bar{z}),\bar{y})$.

Будем считать, что переменная x является единственной общей переменной машин F и G. Это позволит не беспокоиться, что изменения значения регистров одной машины повлияют на значения регистров, используемых другой (как входных и выходного, так и вспомогательных). Мы также потребуем, чтобы эти машины не изменяли значения входных регистров.

Понятно, что в требуемом вычислении необходимо сначала посчитать значение x машиной G, а затем воспользоваться им, запустив машину F, которая и вернёт окончательный результат. Поскольку мы хотим объединить две машины в одну, то потребуются некоторые их модификации. Так, машина G больше не должна останавливаться после завершения вычисления x, то есть из неё следует исключить инструкцию k : stop для некоторой метки k. Вместо этой инструкции нужно выполнять первую инструкцию машины F, а значит, необходимо соответствующим образом перенумеровать все инструкции последней, исправив одновременно метки в goto. Пусть модифицированные указанным образом машины обозначены G' и F', тогда наше вычисление будет выполняться новой машиной $(G'F')_u^{\bar{y},\bar{z}}$.

Свойство неизменности входных регистров гарантирует нам, что к моменту начала работы фрагмента F' построенной машины необходимый для её работы набор регистров $\bar{y}$ испорчен не будет, а x уже будет вычислен машиной G'.

Рассмотренное вычисление является примером *композиции* функций. Если повторить соответствующую процедуру несколько раз, по одному разу для каждого аргумента функции f, то мы получим доказательство следующего утверждения.

Теорема. *Если имеются частично вычислимые функции $\lambda\bar{y}_n.f(\bar{y}_n)$ и $\lambda\bar{z}.g_i(\bar{z})$, где $1 \leqslant i \leqslant n$, то их композиция, то есть функция $\lambda\bar{z}.f(g_1(\bar{z}),\ldots,g_n(\bar{z}))$, также является частично вычислимой.*

Такой способ построения новой функции из набора имеющихся функций, как в условиях теоремы, называют *оператором композиции*, поэтому это утверждение может быть переформулировано следующим образом: *множество $\mathcal{P}$ замкнуто относительно оператора композиции.*

Теперь обратимся к реализации посредством регистровых машин *оператора примитивной рекурсии.*

Пусть имеются две функции $\lambda\bar{y}.h(\bar{y})$ и $\lambda x\bar{y}z.g(x,\bar{y},z)$. Говорят, что функция $f = f(x,\bar{y})$ построена с помощью оператора примитивной рекурсии, если она определяет-

ся следующими двумя равенствами (называемыми *схемой примитивной рекурсии*):

$$f(0, \bar{y}) = h(\bar{y})$$
$$f(x+1, \bar{y}) = g(x, \bar{y}, f(x, \bar{y}))$$

Количество рекурсивных вызовов здесь определяется значением x. Мы будем использовать для указанного построения обозначение $f = \mathrm{prim}(h, g)$. Покажем, что применение оператора примитивной рекурсии может быть реализовано комбинацией регистровых машин $G_z^{i,\bar{y},z} = g$ и $H_z^{\bar{y}} = h$.

Предположим, что единственными общими переменными машин G и H являются переменные z и $\bar{y}$, причём переменные из набора $\bar{y}$ используются только для чтения, и G не меняет i. Пусть G' и H' — их модификации с удалёнными инструкциями stop и соответствующим образом перенумерованные. Идея построения $F_z^{x,\bar{y}}$ состоит в том, чтобы сначала вычислить в z результат H', то есть $f(0, \bar{y})$, а затем несколько раз вызывать G', пока не будет получено значение для заданного x.

Машина, реализующая эту идею, может выглядеть следующим образом:

$$\begin{aligned}
&H' \\
&r : i \leftarrow 0 \\
&r+1 : \text{if } x = 0 \text{ goto } k+m+2 \text{ else goto } r+2 \\
&r+2 : x \leftarrow x \dot{-} 1 \\
&G' \\
&k : i \leftarrow i+1 \\
&k+1 : w_1 \leftarrow 0 \\
&\dots \\
&k+m : w_m \leftarrow 0 \\
&k+m+1 : \text{goto } r+1 \\
&k+m+2 : \text{stop}
\end{aligned}$$

Инструкции с номерами $k+1, \dots, k+m$ предназначены для обнуления всех вспомогательных переменных, используемых в машине G', перед повторным её запуском. Напомним, что автоматически это происходит только в момент инициализации (то есть перед первым запуском).

Возможность приведённого построения доказывает следующую теорему:

Теорема. *Имеют место два утверждения:*

1. *Если функции g и h — частично вычислимые, то функция $f = \mathrm{prim}(g, h)$ также частично вычислима.*

2. *Множество $\mathcal{P}$ замкнуто относительно оператора примитивной рекурсии.*

Определим ещё один оператор, называемый *оператором неограниченного поиска*, μ-оператором или *оператором минимизации*.

Пусть имеется функция $\lambda x \bar{y}.g(x, \bar{y})$. Говорят, что функция $f(\bar{y})$ получена из неё с помощью оператора неограниченного поиска, если

$$f(\bar{a}) = \min\{x : g(x, \bar{a}) = 0, \forall y\,(y < x \longrightarrow g(y, \bar{a})\downarrow)\}.$$

Соответствующий факт коротко обозначается как $f(\bar{a}) = (\mu x)g(x, \bar{a})$. Обратите внимание, что значением построенной функции является наименьшее такое x, что g для него обращается в ноль, причём для всех меньших значений x функция g должна быть определена.

Пусть функция g вычисляется машиной $G_z^{x,\bar{y}} = g$, в которой все входные регистры используются только для чтения, а G' — её перенумерованная модификация без инструкции stop. Построим машину $F_x^{\bar{y}} = f$:

$$
\begin{aligned}
& 1 : x \leftarrow 0 \\
& \quad G' \\
& k : \text{if } z = 0 \text{ goto } k + m + 3 \text{ else goto } k + 1 \\
& k + 1 : w_1 \leftarrow 0 \\
& \quad \ldots \\
& k + m : w_m \leftarrow 0 \\
& k + m + 1 : x \leftarrow x + 1 \\
& k + m + 2 : \text{goto } 2 \\
& k + m + 3 : \text{stop}
\end{aligned}
$$

Мы начинаем поиск со значения $x = 0$. Если переменная z в результате вычисления G' оказывается равной 0, то поиск завершается. В противном случае, как и в предыдущем примере, приходится обнулить все вспомогательные переменные G', а затем увеличить x на единицу и повторить вычисление G'. Ясно, что это вычисление зациклится, если z никогда не обратится в ноль, что вообще говоря вполне возможно.

Приведённое построение доказывает следующую теорему.

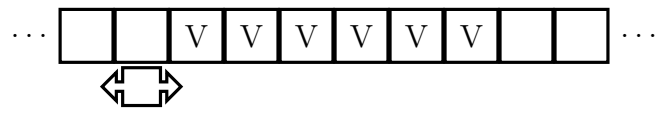
Теорема. *Множество частично вычислимых функций $\mathcal{P}$ замкнуто относительно оператора неограниченного поиска.*

В следующей части курса, посвящённой теории вычислимости, мы будем использовать регистровые машины и введённые здесь операторы для обоснования существования невычислимых проблем.

2.5. Другие модели вычислений

2.5.1. Машины Поста

Машина Поста — это модель вычислений, похожая на машины Тьюринга, но использующая более примитивные операции и имеющая более простую структуру. Машина Поста состоит из бесконечной ленты, каждая ячейка которой может быть помеченной, либо нет (понятие ленточного алфавита при этом не вводится). Одна из ячеек ленты считается в каждый момент времени текущей (её обозревает каретка считывающего устройства). На рисунке показано возможное состояние ленты (помеченные ячейки обозначены символом V, под лентой нарисована каретка считывающего устройства):



Программой для машины Поста является конечная пронумерованная последовательность команд:

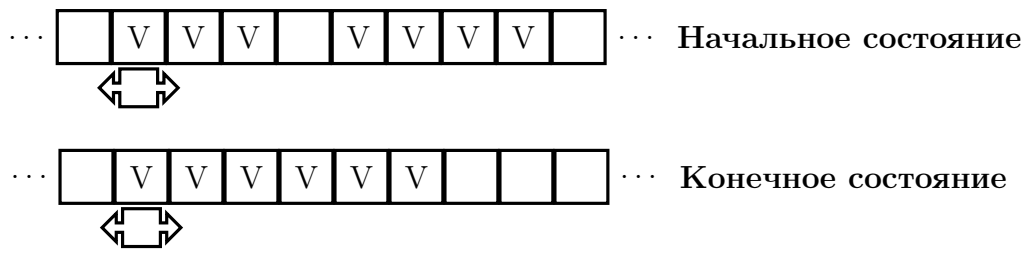
- `V j` — установить метку и перейти на команду с номером `j`;
- `X j` — стереть метку и перейти на команду с номером `j`;
- `L j` — сдвинуть каретку влево и перейти на команду с номером `j`;
- `R j` — сдвинуть каретку вправо и перейти на команду с номером `j`;
- `? j1,j2` — если текущая ячейка помечена, то перейти на команду с номером `j1`, иначе перейти на команду с номером `j2`;
- `!` — остановить работу машины.

Помимо команды останова машина Поста останавливается при выполнении недопустимой операции (установка метки в уже помеченную ячейку, снятие метки с непомеченной, переход на несуществующую команду). Ясно, что при работе машины Поста возможно зацикливание.

Продемонстрируем работу машины Поста на примере сложения двух натуральных чисел. Для представления числа n будем использовать последовательность из $n+1$ подряд идущей помеченной ячейки (одинокая помеченная ячейка соответствует числу 0), числа будут отделяться друг от друга одной непомеченной ячейкой. Начальная позиция каретки — первая помеченная ячейка первого числа, на ней же будем завершать работу машины. Для решения задачи нужно будет а) найти и установить метку в ячейку, разделяющую числа; б) найти и стереть две метки в конце второго числа; в) вернуться на первую ячейку результата:

```
1  R 2          8  L 9
2  ? 1,3        9  X 10
3  V 4          10 L 11
4  R 5          11 ? 10,12
5  ? 4,6        12 R 13
6  L 7          13 !
7  X 8
```

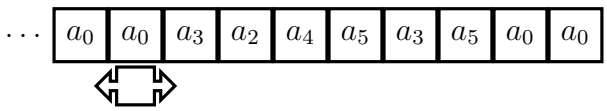
Первая подзадача выполняется командами 1–3, вторая — командами 4–9, третья — 10–13. То же самое можно сделать короче, хотя и не столь очевидно. Начальное и конечное состояния ленты при сложении чисел 2 и 3 показаны на рисунке:



2.5.2. Модель $\mathcal{P}''$

С точки зрения языков программирования машины Поста и Тьюринга, а также регистровые машины чересчур ориентированы на инструкцию `goto`. В период становления структурного программирования один из его пионеров, итальянский специалист Коррадо Бём решил сформулировать эквивалентную модель вычислений, ориентированную на структурное программирование. Результатом этой работы явилась модель $\mathcal{P}''$ (1964 год).

В этой модели используется полубесконечная влево лента, в каждой позиции которой находится один символ ленточного алфавита $\{a_0, a_1, \ldots, a_n\}$, где $n \geqslant 1$. Символ a_0 считается пробельным, так что в алфавите имеется как минимум один непробельный символ. По ленте как обычно движется каретка считывающего устройства:



Программа в модели $\mathcal{P}''$ — это слово в алфавите $\{R, \lambda, (,)\}$, строящееся по следующим правилам:

1) R и λ — слова;
2) если p и q — слова, то pq — слово;
3) если q — слово, то (q) — слово;
4) других слов нет.

Следующие программы являются синтаксически корректными:

- $RR\lambda(RR\lambda)$;
- $((\lambda\lambda)(RR))$;
- $R\lambda R\lambda(R)(\lambda)$.

Семантика, то есть порядок обработки этих символов при исполнении программы, следующая:

- R — сдвинуть каретку вправо на одну позицию (если это возможно);
- λ — заменить текущий символ a_i на $a_{(i+1) \bmod (n+1)}$ и сдвинуть каретку влево на одну позицию;
- (q) — пока текущий символ не является пробельным (a_0), повторять q.

Символ λ выполняет циклический инкремент текущего символа, то есть символ a_n заменяется на a_0, а все остальные символы заменяются на следующий символ ленточного алфавита. Круглые скобки являются аналогом условного цикла **while**, так что здесь действительно удалось отказаться от использования конструкции, так или иначе похожей на `goto`. Символы программы обрабатываются слева направо, выполнение программы завершается после обработки последнего символа.

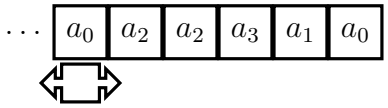
Можно показать, что введённых команд достаточно для реализации произвольной вычислимой функции, однако пользоваться ими не очень удобно. Следующий набор команд также достаточен, но гораздо более удобен (часть из них можно считать *синтаксическим сахаром*):

- $r \equiv \lambda R$ — инкремент символа без сдвига каретки;
- $r' \equiv r^n \equiv \underbrace{rr\ldots r}_{n \text{ раз}}$ — декремент символа (стоит проверить, почему это так);
- $L \equiv r'\lambda$ — сдвиг влево без изменения символа;
- R — сдвиг вправо.

Пользуясь этим набором команд, реализуем вычитание единицы из числа, заданного в n-ичной системе счисления:

- символ a_1 соответствует цифре 0, символ a_2 — цифре 1 и т.д.;
- младший разряд числа записывается справа;
- число ограничено пробельными символами слева и справа.

К примеру, на следующем рисунке изображена лента модели $\mathcal{P}''$ с записанным на ней числом 42 в троичной системе счисления (размер ленточного алфавита $n = 3$):

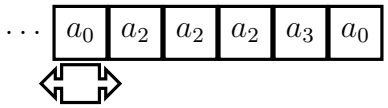


Будем считать, что перед началом работы каретка считывает пробельный символ, ограничивающий заданное число слева. Там же она должна находиться в конце работы.

Программа, решающая поставленную задачу, имеет следующий вид:

$$R(R)L(r'(L(L))r'L)Rr.$$

При этом содержимое ленты в конце работы окажется следующим:



Полезно проследить за выполнением команд программы вручную, чтобы разобраться с реализуемым здесь алгоритмом и убедиться в его корректности. Собственно, вычитание единицы здесь вполне традиционное, знакомое по алгоритму вычитания столбиком.

Модель $\mathcal{P}''$ имеет реальное воплощение в виде настоящего языка программирования, который называется Brainfuck. Его команды однозначно отображаются на введённые здесь команды r, r', L и R, и обозначаются, соответственно, `+`, `-`, `<` и `>`. Круглые скобки заменяются на квадратные. Программа вычитания единицы будет выглядеть в этих обозначениях следующим образом: `>[>]<[-[<[<]]-<]>+`. Помимо этих команд в языке Brainfuck имеются также команды `.` (вывод символа) и `,` (ввод символа). Таким образом, это полноценный язык с вводом-выводом, на котором можно запрограммировать любую вычислимую функцию (разумеется, при условии наличия сколь угодно большого объёма памяти).

2.6. Эквивалентность моделей и тезис Чёрча—Тьюринга

Все вычислительные модели, которые были введены в этом разделе, эквивалентны друг другу. Эквивалентность понимается в том смысле, что любое вычисление, которое можно выполнить с помощью одной модели вычислений, можно выполнить и средствами другой. В этом курсе мы не доказываем эквивалентность моделей вычислений. Любые такого рода доказательства крайне утомительны, поскольку полны множеством технических деталей. Они также не представляют особого интереса с теоретической точки зрения, давая лишь пример тщательной работы по использованию той или иной модели вычислений, а также по перекодированию разных способов представления данных. Тем не менее, такое доказательство необходимо при построении новой модели вычислений, поскольку только оно способно легитимизировать новую модель в качестве эквивалентной машине Тьюринга, то есть способной описать любое вычисление, которое может быть описано машиной Тьюринга.

Для любой появляющейся новой модели вычислений достаточно доказать её эквивалентность только одной из тех моделей, для которых эквивалентность остальным уже была установлена. Очень часто для этой цели используется машина Тьюринга, как наиболее убедительная модель, описывающая работу человека-вычислителя в чрезвычайно простых операциях. За прошедшие почти восемьдесят лет с момента первого описания машин Тьюринга так и не появилась модель вычисления, которая смогла бы превзойти их по вычислительным способностям. Все новые модели оказывались либо более слабыми, либо эквивалентными ей. Не является исключением и модель квантовых вычислений, её эквивалентность машинам Тьюринга уже доказана.

В настоящее время принято считать, что имеет место *тезис Чёрча—Тьюринга*: *любая вычислимая функция может быть вычислена с помощью соответствующей машины Тьюринга или любой другой эквивалентной ей модели.* Это утверждение не может быть доказано, поскольку в нём участвует неопределяемое понятие «вычислимой функции», под которым мы понимаем нечто, что может быть принципиально вычислено, какими бы методами мы при этом ни пользовались.

Тезис Чёрча—Тьюринга принимается на веру, он отражает наши представления о современной картине мира, его устройстве с точки зрения процесса вычислений. Он также позволяет нам утверждать, что если что-то не может быть вычислено в одной из разобранных ранее моделей вычислений, то это что-то не может быть вычислено в принципе. Таким образом, мы получаем способ решения проблем, поставленных во введении к этому курсу, а именно проблем о доказательстве несуществования алгоритмов решения определённых задач. Теперь достаточно доказать, что не существует, к примеру, машины Тьюринга, которая бы их решала. Поскольку это понятие формально определено, то доказательство несуществования оказывается возможным.

Непосредственное использование вычислительных моделей крайне затруднительно, они для этого слишком упрощены. Вместо этого на практике используются *полные по Тьюрингу языки программирования* (таковыми являются абсолютное большинство известных языков программирования). В некотором смысле они тоже являются моделями вычислений, но снабжены большим количеством *синтаксического са-*

хара, который облегчает их использование, но затрудняет формальные рассуждения относительно семантики программ. Известно, что любой язык программирования с возможностью ветвления и условным циклом (например, **while**) или неограниченной рекурсией, полон по Тьюрингу (то есть представляет собой эквивалентную машине Тьюринга модель вычислений). Интересно, что иногда используют и неполные по Тьюрингу языки программирования. К ним относятся языки, программы на которых гарантированно завершаются (например, Coq или Agda). В некоторых языках требование завершаемости может быть явно предъявлено к некоторой функции, в этом случае компилятор выполняет соответствующую проверку (правда, как мы скоро узнаем, в общем случае это неразрешимая задача). Примером таких языков является язык программирования Idris. Требование завершаемости необходимо в случаях, когда критически важна корректность работы программного обеспечения.

3. Теория вычислимости

В предыдущем разделе были определены два множества функций: множество частично рекурсивных функций $\mathcal{P}$ и множество рекурсивных (всюду определённых) функций $\mathcal{R}$. С их определениями не очень удобно работать, поскольку непонятно, когда существует регистровая машина, вычисляющая функцию, и всегда ли она останавливается. Первой целью этого раздела будет более точная характеризация функций из $\mathcal{P}$: мы обнаружим *нормальную форму* для частично рекурсивных функций и получим строгое индуктивное определение их множества. К сожалению, получить аналогичное определение для всюду определённых функций не удастся, а выяснение причин является второй важной целью этого раздела. Как мы увидим, определение завершаемости вычисления является примером *неразрешимой задачи*. Мы также рассмотрим примеры других неразрешимых задач.

Этот раздел будет заканчиваться достижением третьей цели — описанием неразрешимых задач, которые возникают за пределами теории алгоритмов.

3.1. Примитивная рекурсивность

3.1.1. Примитивно рекурсивные функции

Введём в рассмотрение так называемые *инициальные* функции:

- нуль-функция $Z = \lambda x\,.\,0$;
- функция следования $S = \lambda x\,.\,x + 1$;
- набор функций-проекторов $P_i^n = \lambda \bar{x}_n\,.\,x_i$, где $1 \leqslant i \leqslant n$ (возвращают один аргумент из набора n аргументов).

Инициальные функции являются в некотором смысле самыми простыми функциями, в пункте 2.4.2 для них уже были построены регистровые машины, поэтому функции принадлежат множествам $\mathcal{P}$ и, очевидно, $\mathcal{R}$.

Пользуясь этим множеством, дадим индуктивное определение множества *примитивно рекурсивных* функций $\mathcal{PR}$, содержащее

1. Инициальные функции.
2. Функции, построенные из примитивно рекурсивных функций с помощью оператора композиции: если $\lambda \bar{x}_n\,.\,f(\bar{x}_n) \in \mathcal{PR}$ и $\lambda \bar{z}\,.\,g_i(\bar{z}) \in \mathcal{PR}$ для $1 \leqslant i \leqslant n$, то $\lambda \bar{z}\,.\,f(g_1(\bar{z}), \ldots, g_n(\bar{z})) \in \mathcal{PR}$.
3. Функции, построенные из примитивно рекурсивных функций с помощью оператора примитивной рекурсии: если $\lambda \bar{y}\,.\,h(\bar{y}) \in \mathcal{PR}$ и $\lambda x \bar{y} z\,.\,g(x, \bar{y}, z) \in \mathcal{PR}$, то $\mathbf{prim}(h, g) \in \mathcal{PR}$.

Таким образом, множество $\mathcal{PR}$ *по определению* является замыканием множества инициальных функций относительно операторов композиции и примитивной рекурсии.

Определение оператора композиции является довольно жёстким, оно требует подставлять функции из набора g_i сразу во все аргументы, однако это требование может быть снято благодаря применению инициальных функций. Пусть, скажем, функции $\lambda xyw\,.\,f(x,y,w),\ \lambda z\,.\,g(z) \in \mathcal{PR}$. Покажем, что $\lambda xzw\,.\,f(x,g(z),w) \in \mathcal{PR}$ (здесь подстановка вызова функции выполнена только в один из аргументов). Действительно,

$$\lambda xzw\,.\,f(x,g(z),w) = \lambda xzw\,.\,f(P^3_1(x,z,w), g(P^3_2(x,z,w)), P^3_3(x,z,w)).$$

Во втором аргументе функции f участвует композиция g и проектор P^3_2, а в остальных участвуют только проекторы. Последнее выражение полностью соответствует форме определения оператора композиции (подстановка выполняется во все аргументы), поэтому построенная функция является примитивно рекурсивной.

Вместо переменной в вызов примитивно рекурсивной функции можно также подставлять константу, поскольку, к примеру:

$$\lambda yw\,.\,f(2,y,w) = \lambda yw\,.\,f(S(S(Z(P^2_1(y,w)))), P^2_1(y,w), P^2_2(y,w)).$$

Снова функция получена оператором композиции, а подставляемая вместо первого аргумента функция сама по себе является результатом многократного его применения, то есть результирующая функция также примитивно рекурсивна. При необходимости записывать константу мы будем далее пользоваться обозначением вида $SSZ(x) = 2$, избегая в нём лишних скобок.

Приведённые примеры обхода жёсткости оператора композиции можно обобщить, сформулировав *принцип подстановки Гржегорчика.*

Теорема. *Множество $\mathcal{PR}$ замкнуто относительно следующих операций:*

1) *подстановка вызова функции вместо переменной:*
   получение $\lambda\bar{x}\bar{w}\bar{z}\,.\,f(\bar{x},g(\bar{w}),\bar{z})$ из $\lambda\bar{x}y\bar{z}\,.\,f(\bar{x},y,\bar{z})$ и $\lambda\bar{w}\,.\,g(\bar{w})$;

2) *подстановка константы вместо переменной:*
   получение $\lambda\bar{x}\bar{z}\,.\,f(\bar{x},c,\bar{z})$, где c — произвольная константа, из $\lambda\bar{x}y\bar{z}\,.\,f(\bar{x},y,\bar{z})$;

3) *перестановка переменных:*
   получение $\lambda\bar{x}y\bar{z}w\bar{u}.f(\bar{x},w,\bar{z},y,\bar{u})$ из $\lambda\bar{x}y\bar{z}w\bar{u}.f(\bar{x},y,\bar{z},w,\bar{u})$;

4) *отождествление двух переменных:*
   получение $\lambda\bar{x}y\bar{z}\bar{u}.f(\bar{x},y,\bar{z},y,\bar{u})$ из $\lambda\bar{x}y\bar{z}w\bar{u}.f(\bar{x},y,\bar{z},w,\bar{u})$;

5) *введение неиспользуемых переменных:*
   получение $\lambda\bar{x}\bar{z}.f(\bar{x})$ из $\lambda\bar{x}.f(\bar{x})$.

Доказательство этого утверждения выполняется аналогично рассмотренным перед ним примерам.

Определение оператора примитивной рекурсии также довольно жёсткое: в частности, оно не позволяет определять функции с одним аргументом или вести рекурсию по не являющемуся первым аргументу. К счастью, инициальные функции вместе с

принципом подстановки Гржегорчика позволяют обойти и эти ограничения. Приведём несколько примеров.

Покажем, что функция вычитания единицы $p = \lambda x\,.\,x \dot{-} 1$ является примитивно рекурсивной, определив её посредством оператора примитивной рекурсии. Нетрудно видеть, что

$$\begin{aligned} p(0) &= 0 \\ p(x+1) &= x \end{aligned}$$

Однако это определение не соответствует схеме примитивной рекурсии, поэтому определим сначала двухаргументную версию $\widetilde{p} = \lambda xy\,.\,x \dot{-} 1$:

$$\begin{aligned} \widetilde{p}(0,y) &= 0 = Z(y) \\ \widetilde{p}(x+1,y) &= x = P_1^3(x,y,\widetilde{p}(x,y)) \end{aligned}$$

Таким образом, $\widetilde{p} = \mathbf{prim}(Z, P_1^3)$ и, значит, $\widetilde{p} \in \mathcal{PR}$, а для получение p из неё достаточно подставить константу на место второго аргумента: $p = \lambda x\,.\,\widetilde{p}(x,0) \in \mathcal{PR}$.

Установим теперь примитивную рекурсивность операции вычитания второго аргумента из первого: $\mathrm{sub} = \lambda xy\,.\,x \dot{-} y$. Очевидное определение требует рекурсии по второму аргументу:

$$\begin{aligned} \mathrm{sub}(x,0) &= x \\ \mathrm{sub}(x,y+1) &= p(\mathrm{sub}(x,y)) \end{aligned}$$

Определим вместо этого функцию $\widetilde{\mathrm{sub}} = \lambda xy\,.\,y \dot{-} x$:

$$\begin{aligned} \widetilde{\mathrm{sub}}(0,y) &= y = P_1^1(y) \\ \widetilde{\mathrm{sub}}(x+1,y) &= p(\widetilde{\mathrm{sub}}(x,y)) = p(P_3^3(x,y,\widetilde{\mathrm{sub}}(x,y))) \end{aligned}$$

Схема рекурсии теперь выглядит подобающим образом, в правых частях стоят примитивно рекурсивные функции (во второй строке справа их композиция), следовательно $\widetilde{\mathrm{sub}} \in \mathcal{PR}$. Исходная функция sub теперь может быть получена перестановкой переменных по принципу подстановки Гржегорчика, а значит также примитивно рекурсивна.

Пользуясь подобными приёмами нетрудно установить примитивную рекурсивность операций сложения, умножения и возведения в степень. В качестве последнего примера докажем примитивную рекурсивность операции выбора

$$\mathrm{sw} = \lambda xyz\,.\,\text{if } x = 0 \text{ then } y \text{ else } z$$

Воспользуемся для этого схемой примитивной рекурсии:

$$\begin{aligned} \mathrm{sw}(0,y,z) &= y = P_1^2(y,z) \\ \mathrm{sw}(x+1,y,z) &= z = P_3^4(x,y,z,\mathrm{sw}(x,y,z)) \end{aligned}$$

Таким образом, $\mathrm{sw} = \mathbf{prim}(P_1^2, P_3^4) \in \mathcal{PR}$.

Теорема. $\mathcal{PR} \subset \mathcal{R}$.

Доказательство. Поскольку для инициальных функций можно написать регистровые машины, их вычисляющие, то есть они являются частично рекурсивными, а множество $\mathcal{P}$ замкнуто относительно операторов композиции и примитивной рекурсии, то $\mathcal{PR} \subset \mathcal{P}$.

Легко видеть, что и оператор композиции и оператор примитивной рекурсии сохраняют всюду определённость, если используются для всюду определённых функций. Скажем, при вычислении примитивной рекурсии идёт уменьшение аргумента до нуля, то есть определённая примитивной рекурсией функция гарантированно вычислится. Инициальные функции очевидно всюду определены, поэтому $\mathcal{PR} \subset \mathcal{R}$. □

В качестве побочного результата этих рассуждений получаем:

Теорема. *Множество $\mathcal{R}$ замкнуто относительно операций композиции и примитивной рекурсии.*

3.1.2. Рекурсивные и примитивно рекурсивные отношения

Отношение $R(\bar{x})$ называется (примитивно) рекурсивным, если его характеристическая функция

$$\chi_R = \lambda\bar{x}\,.\begin{cases}0, \text{если } R(\bar{x})\\ 1, \text{если } \neg R(\bar{x})\end{cases}$$

(примитивно) рекурсивна. Множества примитивно рекурсивных и рекурсивных отношений будет обозначать символами $\mathcal{PR}_*$ и $\mathcal{R}_*$ соответственно. Мы также будем отождествлять понятия отношения, предиката и 0-1-значной функции, и свободно переходить между отношением и его характеристической функцией, пользуясь тождеством $R(\bar{x}) \equiv \chi_R(\bar{x}) = 0$.

Теорема. *Для того чтобы отношение $R(\bar{x})$ было примитивно рекурсивным, необходимо и достаточно, чтобы существовала такая функция $f \in \mathcal{PR}$, что для всех $\bar{x}$: $R(\bar{x}) \equiv f(\bar{x}) = 0$.*

Доказательство. Для доказательства необходимости можно заметить, что искомой функцией f является характеристическая функция отношения R.

Для доказательства достаточности возьмём в качестве характеристической функции отношения R функцию $\chi_R = \lambda\bar{x}\,.\,1 \dot{-} (1 \dot{-} f(\bar{x}))$. Ясно, что эта функция примитивно рекурсивна и обращается в ноль тогда и только тогда, когда $f(\bar{x}) = 0 \equiv R(\bar{x})$. Существование такой функции доказывает примитивную рекурсивность отношения R. □

Аналогичный факт с дословно переносимым доказательством имеет место и для отношений из $\mathcal{R}_*$. Поскольку $\mathcal{PR} \subset \mathcal{R}$, то из доказанной теоремы и упомянутого факта следует, что $\mathcal{PR}_* \subset \mathcal{R}_*$.

Теорема. *Множество $\mathcal{PR}_*$ замкнуто относительно логических операций.*

Доказательство. Поскольку одни логические операции выражаются через другие, для обоснования теоремы достаточно рассмотреть, скажем, отрицание и дизъюнкцию.

Ясно, что в качестве характеристической функции отношения $\neg R$ можно взять функцию $\chi_{\neg R} = \lambda\bar{x}\,.\,1 \dot{-} \chi_R(\bar{x})$. Она является примитивно рекурсивной и, очевидно, что $\neg R(\bar{x}) \equiv \chi_{\neg R}(\bar{x}) = 0$, поэтому отношение $\neg R$ также примитивно рекурсивно.

Для построения характеристической функции отношения $R(\bar{x}) \vee Q(\bar{y})$ удобно воспользоваться примитивно рекурсивной операцией выбора:

$$\chi_{R\vee Q}(\bar{x},\bar{y}) = \Big(\text{if } \chi_R(\bar{x}) = 0 \text{ then } 0 \text{ else } \chi_Q(\bar{y})\Big) = \mathrm{sw}(\chi_R(\bar{x}), 0, \chi_Q(\bar{y})).$$

Эта функция также является характеристической (принимает значения 0 или 1), примитивно рекурсивна (как композиция примитивно рекурсивных функций) и обращается в ноль если и только если $R(\bar{x}) \vee Q(\bar{y})$. Поэтому дизъюнкция двух примитивно рекурсивных отношений также оказывается примитивно рекурсивным отношением. □

Совершенно аналогично доказывается замкнутость относительно логических операций множества $\mathcal{R}_*$.

Покажем, что отношения $x \leqslant y$, $x < y$ и $x = y$ являются примитивно рекурсивными. Для доказательства примитивной рекурсивности отношения $x \leqslant y$ достаточно (в соответствии с доказанной выше теоремой) предъявить примитивно рекурсивную функцию $\lambda xy\,.\,x \dot{-} y$. Ясно, что $x \leqslant y \equiv x \dot{-} y = 0$. Примитивная рекурсивность двух оставшихся отношений вытекает из замкнутости множества $\mathcal{PR}_*$ относительно логических операций и того, что $x < y \equiv \neg(y \leqslant x)$ и $x = y \equiv (x \leqslant y) \wedge (y \leqslant x)$.

Сформулируем теперь полезную с технической точки зрения теорему.

Теорема. *Если* $R(\bar{x}, y, \bar{z}) \in \mathcal{PR}_*$ *и* $\lambda\bar{w}\,.\,f(\bar{w}) \in \mathcal{PR}$, *то* $R(\bar{x}, f(\bar{w}), \bar{z}) \in \mathcal{PR}_*$.

Эта теорема легко доказывается переходом к характеристическим функциям и ссылкой на подстановку Гржегорчика. Аналогичный факт, очевидно, имеет место для $\mathcal{R}_*$ и $\mathcal{R}$.

Назовём *графиком* функции $f(\bar{x})$ отношение $f(\bar{x}) = z$. Если взять примитивно рекурсивное отношение $y = z$ и функцию $f(\bar{x})$, то из предыдущей теоремы и подстановки $f(\bar{x})$ вместо y вытекает, что

Теорема. *График (примитивно) рекурсивной функции (примитивно) рекурсивен.*

Интересно, что в случае $\mathcal{R}_*$ верно и обратное, в отличие от $\mathcal{PR}_*$: из примитивной рекурсивности графика не следует примитивная рекурсивность функции. Соответствующий пример будет приведён позднее.

3.1.3. Ограниченная квантификация и ограниченный поиск

Введённый ранее оператор неограниченного поиска, как мы видели, может приводить к зацикливающимся вычислениям, однако поиск сам по себе удобен для решения некоторых задач, поэтому необходимо ввести его ограниченную версию, которая оставляла бы нас в пределах $\mathcal{R}$. Начнём с ограниченной квантификации — его разновидности для отношений.

Символом $(\forall y)_{<z} R(y, \bar{x})$ будем обозначать предикат $(\forall y)(y < z \longrightarrow R(y, \bar{x}))$, а символом $(\exists y)_{<z} R(y, \bar{x})$ — предикат $(\exists y)(y < z \wedge R(y, \bar{x}))$, введём также их версии с нестрогим неравенством.

Теорема. *Множество $\mathcal{PR}_*$ замкнуто относительно ограниченной квантификации.*

Доказательство. Замкнутость достаточно доказать относительно одного из двух кванторов, например, квантора существования, поскольку они выражаются один через другой с помощью логических операций, замкнутость относительно которых была установлена ранее:

$$(\forall y)_{<z} R(y, \bar{x}) \equiv \neg\Big((\exists y)_{<z} \neg R(y, \bar{x})\Big)$$

Пусть $R(y, \bar{x}) \in \mathcal{PR}_*$. Покажем, что $Q(z, \bar{x}) = (\exists y)_{<z} R(y, \bar{x}) \in \mathcal{PR}_*$. Ясно, что $Q(0, \bar{x})$ ложно (соответствующих y просто нет), а $Q(z+1, \bar{x}) \equiv Q(z, \bar{x}) \vee R(z, \bar{x})$ (существование имеет место либо на последнем значении z, либо где-то ранее, то есть $< z$). Эти соображения позволяют определить характеристическую функцию отношения Q посредством оператора примитивной рекурсии:

$$\begin{aligned} \chi_Q(0, \bar{x}) &= 1 \\ \chi_Q(z+1, \bar{x}) &= \chi_Q(z, \bar{x})\chi_R(z, \bar{x}) \end{aligned}$$

(умножение во второй строке соответствует именно дизъюнкции, поскольку в нашем определении характеристическая функция на элементах из отношения обращается в ноль). Из примитивной рекурсивности функции χ_Q следует примитивная рекурсивность Q и, соответственно, замкнутость $\mathcal{PR}_*$ относительно ограниченной квантификации. □

Замкнутость $\mathcal{R}_*$ относительно ограниченной квантификации доказывается аналогично.

Пусть $\lambda y \bar{x} \,.\, f(y, \bar{x})$ — всюду определённая функция. Оператором ограниченного поиска называется следующее преобразование этой функции:

$$g(z, \bar{x}) = (\mu y)_{<z} f(y, \bar{x}) = \begin{cases} \min\{y : y < z \wedge f(y, \bar{x}) = 0\}, \text{если } (\exists y)_{<z} f(y, \bar{x}) = 0 \\ z, \text{в противном случае} \end{cases}$$

Ограниченный поиск с нестрогим неравенством определяется как $(\mu y)_{\leqslant z} \equiv (\mu y)_{<z+1}$.

Теорема. *Множество $\mathcal{PR}$ замкнуто относительно оператора ограниченного поиска $(\mu y)_{<z}$.*

Доказательство. Необходимо доказать, что $g = \lambda z \bar{x} \,.\, (\mu y)_{<z} f(y, \bar{x}) \in \mathcal{PR}$ при условии, что $f \in \mathcal{PR}$. Действительно, построим для g схему примитивной рекурсии:

$$\begin{aligned} g(0, \bar{x}) &= 0 \\ g(z+1, \bar{x}) &= \text{if } g(z, \bar{x}) < z \text{ then } g(z, \bar{x}) \text{ else } \Big(\text{if } f(z, \bar{x}) = 0 \text{ then } z \text{ else } z+1\Big) \end{aligned}$$

Во второй строке справа стоит примитивно рекурсивная функция, поэтому $g \in \mathcal{PR}$. □

Следствием этой теоремы является замкнутость $\mathcal{PR}$ относительно $(\mu y)_{\leqslant z}$.

3.1.4. Примеры

Введём удобные обозначения: $(\mu y)_{<z}R(y,\bar{x}) \equiv (\mu y)_{<z}\chi_R(y,\bar{x})$ и $(\mu y)_{\leqslant z}R(y,\bar{x}) \equiv (\mu y)_{\leqslant z}\chi_R(y,\bar{x})$, то есть будем далее использовать ограниченный поиск с отношениями.

1. Вычисление частного от деления x на y — функция $\lambda xy\,.\,\lfloor x/y\rfloor$ — примитивно рекурсивная операция, поскольку $\lfloor x/y\rfloor = (\mu z)_{\leqslant x}((z+1)y > x)$. Вообще говоря, в математике делить на ноль запрещено, однако наше определение через оператор ограниченного поиска это позволяет. Поскольку влиять ни на что существенное это не будет, доопределим деление, считая, что $\lfloor x/0\rfloor = x+1$ (это согласуется с результатом ограниченного поиска), получив, таким образом, всюду определённую функцию.

2. Вычисление остатка от деления x на y — $\lambda xy\,.\,\mathrm{rem}(x,y)$ — также примитивно рекурсивно, поскольку $\mathrm{rem}(x,y) = x \dot{-} y\lfloor x/y\rfloor$.

3. Отношение делимости $\lambda xy\,.\,x|y$ (x «делит» y) примитивно рекурсивно, так как $x|y \equiv \mathrm{rem}(y,x) = 0$. Заметим, что из-за нашего определения частного получается, что $0|0$, но с этим можно жить.

4. Предикат $Pr(x)$ — «x — простое» — примитивно рекурсивен, поскольку $Pr(x) \equiv x > 1 \wedge (\forall y)_{\leqslant x}(y|x \longrightarrow y = 1 \vee y = x)$.

5. Функция $\pi(x)$, вычисляющая количество простых чисел, не превосходящих x, является примитивно рекурсивной, что вытекает из следующей схемы:

   $$\pi(0) = 0$$
   $$\pi(x+1) = \text{if } Pr(x+1) \text{ then } \pi(x)+1 \text{ else } \pi(x)$$

   Понятно, что для завершения обоснования эту схему необходимо привести в соответствие со схемой примитивной рекурсии.

6. Функция $\lambda n\,.\,p_n$, вычисляющая n-е простое число (нумерация с нуля), является примитивно рекурсивной. Этот факт можно установить за два шага. Во-первых, её график $y = p_n \equiv Pr(y) \wedge \pi(y) = n+1 \in \mathcal{PR}_*$. Во-вторых, по графику можно организовать ограниченный поиск, поскольку, как нетрудно установить индукцией по n, для всех n выполняется неравенство $p_n \leqslant 2^{2^n}$, а значит, $p_n = (\mu y)_{\leqslant 2^{2^n}}(y = p_n)$.

7. Отыскать показатель степени p_n в разложении числа x на простые множители также можно примитивно рекурсивной функцией $\lambda nx\,.\,\exp(n,x)$, поскольку $\exp(n,x) = (\mu y)_{\leqslant x}\neg(p_n^{y+1}|x)$.

8. В качестве последнего примера установим примитивную рекурсивность предиката $Seq(x)$ — «разложение x на простые множители содержит все без пропусков простые числа до некоторого предела»:

   $$Seq(x) \equiv x > 1 \wedge (\forall y)_{\leqslant x}(\forall z)_{\leqslant x}\Big(Pr(y) \wedge Pr(z) \wedge y < z \wedge z|x \longrightarrow y|x\Big)$$

   (если некоторое простое число делит x, то всякое меньшее простое число также его делит).

3.2. Арифметизация вычислений

3.2.1. Кодирование числовых последовательностей

Пусть $a_0, \ldots, a_n$, $n \geqslant 0$ — последовательность натуральных чисел. Обозначим символом $[a_0, \ldots, a_n]$ *число*

$$\prod_{i \leqslant n} p_i^{a_i+1}$$

где p_i — подряд идущие простые числа, то есть $2, 3, 5, \ldots$. В силу единственности разложения числа на простые множители исходная последовательность по заданному числу (*коду*) восстанавливается однозначно, если

$$[a_0, \ldots, a_n] = [b_0, \ldots, b_n], \text{то } a_i = b_i \text{ для всех } i.$$

Последовательность с кодом z может быть раскодирована с помощью двух операций:

1) $l(z) = (\mu y)_{\leqslant z} \neg(p_y | z)$ — длина последовательности (то есть номер, считая с нуля, первого простого числа, не входящего в разложение z на простые множители);

2) $(z)_i = \exp(i, z) \dot{-} 1$ — i-й элемент последовательности.

Таким образом, декодирование последовательности может быть выполнено примитивно рекурсивными функциями. Ясно, что обе функции работают корректно при условии, что число z действительно является кодом, то есть удовлетворяет предикату $Seq(z)$, причём

$$z = [(z)_0, \ldots, (z)_{l(z) \dot{-} 1}].$$

Предикат $Seq(x)$ вместе с функциями декодирования $\lambda z \,.\, l(z)$ и $\lambda i z \,.\, (z)_i$ образуют *схему кодирования* Гёделя, которую мы будем использовать в дальнейшем. Ещё раз подчеркнём, что и кодирование (произведение степеней простых чисел), и проверка кода на корректность (Seq), и декодирование (l и $(\cdot)_i$) являются примитивно рекурсивными операциями.

3.2.2. Кодирование регистровых машин

В исходном определении регистровой машины участвуют переменные — регистры с именами $X1, X11, X111, \ldots$, поэтому каждой переменной

$$X1^i = X\underbrace{1 \ldots 1}_{i \text{ раз}}$$

можно сопоставить число i. Метки инструкций уже являются числами, тип инструкции также можно задать числом. Отсюда получаем следующий способ кодирования инструкций:

1) инструкция $L : X1^i \leftarrow a$ кодируется числом $[1, L, i, a]$;

2) инструкция $L : X1^i \leftarrow X1^i + 1$ кодируется числом $[2, L, i]$;

3) инструкция $L : X1^i \leftarrow X1^i \dot{-} 1$ кодируется числом $[3, L, i]$;

4) инструкция L : if $X1^i = 0$ goto P else goto R кодируется числом $[4, L, i, P, R]$;

5) инструкция L : stop кодируется числом $[5, L]$.

Таким образом, если z — код инструкции, то $(z)_0$ — её тип (число от 1 до 5), $(z)_1$ — её метка и т. д. Для кодирования регистровой машины в целом достаточно сначала закодировать все её инструкции, а затем закодировать последовательность их кодов.

Теорема. *Предикат $URM(z)$ — «число z является кодом некоторой регистровой машины» — является примитивно рекурсивным.*

Доказательство. Корректность кода регистровой машины подразумевает, что

- число действительно является кодом;
- последний элемент соответствующей коду последовательности кодирует инструкцию stop;
- ни один элемент последовательности кроме последнего не кодирует инструкцию stop, и все они являются кодами инструкций одного из типов 1–5.

Все эти условия можно выразить формально следующим образом:

$$\begin{aligned}
URM(z) \equiv Seq(z) \wedge \Big((z)_{l(z)\dot{-}1} = [5, l(z)]\Big) \wedge \\
\wedge (\forall L)_{<l(z)} \Bigg(Seq((z)_L) \wedge \Big(L \neq l(z) \dot{-} 1 \longrightarrow ((z)_L)_0 \neq 5\Big) \wedge \\
\wedge \Big\{ (z)_L = [5, L+1] \vee \\
\vee (\exists i)_{\leqslant z} (\exists a)_{\leqslant z} \Big((z)_L = [1, L+1, i+1, a] \Big) \vee \\
\vee (\exists i)_{\leqslant z} \big\{ (z)_L = [2, L+1, i+1] \vee (z)_L = [3, L+1, i+1] \vee \\
\vee (\exists M)_{<l(z)} (\exists R)_{<l(z)} \big((z)_L = [4, L+1, i+1, M+1, R+1] \big) \big\} \Big\} \Bigg).
\end{aligned}$$

Поскольку ограниченные квантификация и поиск начинают с нуля, в номерах инструкций и меток мы всюду прибавляем единицу. Заметьте также, что метка последней инструкции есть $l(z)$, тогда как индекс кодирующего её элемента последовательности равен $l(z) \dot{-} 1$. □

Будем считать все используемые далее регистровые машины *нормализованными* по входу и выходу в том смысле, что выходным является регистр $X1$, а n входных регистров — это регистры $X11, \ldots, X1^{n+1}$. Все остальные регистры могут использоваться в качестве вспомогательных. Ясно, что любая регистровая машина легко может быть приведена к нормализованному виду. Поскольку любая машина должна как минимум записать в выходной регистр результирующее значение, а инструкция stop ни на какие переменные не ссылается, то можно считать, что любая машина содержит как минимум две инструкции.

3.2.3. Конфигурации, переходы и их кодирование

Конфигурацией (мгновенным описанием, снэпшотом) регистровой машины назовём набор чисел $L; a_1, \ldots, a_r$, содержащий метку инструкции, которая должна выполняться следующей, и текущие значения всех используемых машиной регистров. Понятно, что все конфигурации, возникающие во время работы машины, имеют одну и ту же длину. Способ кодирования конфигурации очевиден: $\mathrm{code}(I) = [L, a_1, \ldots, a_r]$.

Переход машины из конфигурации I_1 в конфигурацию I_2 будем обозначать как $I_1 \vdash I_2$.

Переход можно определить формально как правило, согласно которому изменяется конфигурация. Понятно, что это правило должно зависеть от инструкции, которая выполняется следующей.

Пусть $I_1 = L; a_1, \ldots, a_r$, а $I_2 = P; b_1, \ldots, b_r$. Будем говорить, что $I_1 \vdash I_2$, если имеет место одно из следующих условий:

1) метка L соответствует инструкции $X1^i \leftarrow c$, и конфигурации I_1 и I_2 совпадают за исключением того, что $b_i = c$, а $P = L + 1$;

2) метка L соответствует инструкции $X1^i \leftarrow X1^i + 1$, и конфигурации I_1 и I_2 совпадают за исключением того, что $b_i = a_i + 1$, а $P = L + 1$;

3) метка L соответствует инструкции $X1^i \leftarrow X1^i \dot{-} 1$, и конфигурации I_1 и I_2 совпадают за исключением того, что $b_i = a_i \dot{-} 1$, а $P = L + 1$;

4) метка L соответствует инструкции if $X1^i = 0$ goto R else goto Q, и конфигурации I_1 и I_2 совпадают за исключением того, что $P = R$ в случае, когда $a_i = 0$, и $P = Q$ в противном случае;

5) метка L соответствует инструкции stop, и конфигурации I_1 и I_2 совпадают.

Воспользуемся этим определением для доказательства теоремы, которая пригодится нам в дальнейшем.

Теорема. *Отношение $Step(z, u, v)$ — «машина с кодом z может выполнить переход из конфигурации с кодом u в конфигурацию с кодом v» — является примитивно рекурсивным.*

Доказательство. При построении этого отношения мы не будем проверять корректность кодов z, u и v, отложив эту проверку на более позднее время, проверим лишь, что переход выполнен в соответствии с одним из перечисленных выше условий 1)–5):

$$
\begin{aligned}
Step(z,u,v) \equiv (\exists k)_{\leqslant z}(\exists L)_{<l(z)}\Big(& L+1 = (u)_0 \wedge k > 0 \wedge \\
& \wedge \Big\{ (\exists a)_{\leqslant z}\big((z)_L = [1, L+1, k, a] \wedge v = 2p_k^{a+1}\lfloor u/p_k^{\exp(k,u)}\rfloor\big) \vee \\
& \quad \vee \big((z)_L = [2, L+1, k] \wedge v = 2p_k u\big) \vee \\
& \quad \vee \big((z)_L = [3, L+1, k] \wedge v = 2(\text{if } (u)_k = 0 \text{ then } u \text{ else } \lfloor u/p_k\rfloor)\big) \vee \\
& \quad \vee (\exists P)_{\leqslant l(z)}(\exists R)_{\leqslant l(z)}\big((z)_L = [4, L+1, k, P, R] \wedge P > 0 \wedge R > 0 \wedge \\
& \qquad \wedge v = 2^{(\text{if } (u)_k = 0 \text{ then } P \text{ else } R)+1}\lfloor u/2^{L+2}\rfloor\big) \vee \\
& \quad \vee \big((z)_L = [5, L+1] \wedge v = u\big) \Big\}\Big).
\end{aligned}
$$

Чтобы разобраться в приведённом построении, стоит иметь в виду следующие соображения:

- параметр k соответствует номеру регистра, значение которого меняется при выполнении найденной инструкции — отсюда условие $k > 0$ и ограничение поиска ($\leqslant z$), поскольку номера регистра входят в код инструкции в качестве показателя простого числа, а код инструкции входит в код машины, то есть если такой номер существует, то он гарантированно будет найден в указанных пределах;
- встречающееся несколько раз $(u)_k$ — это значение регистра с номером k, содержащееся в конфигурации;
- также несколько раз присутствующее умножение на 2 — это на самом деле изменение кода при переходе к следующей по порядку инструкции (от L к $L{+}1$): поскольку в конфигурации метка стоит на первом месте, то за соответствующее значение в коде отвечает показатель при первом простом числе — 2, таким образом, это просто переход к следующей степени двойки;
- умножения и деления на простые числа p_k и их степени — это изменения значения k-го регистра (благодаря нашей схеме кодирования деление уменьшает старое значение, а умножение его увеличивает, в сочетании же с функцией exp мы можем полностью заменить старое значение на новое), заметим также, что все деления гарантированно выполняются без остатка;
- совпадение нового и старого кодов $v = u$ — это совпадение конфигураций;
- стоит также помнить, что в схеме кодирования показатель простого множителя всегда увеличен на единицу по сравнению с кодируемым значением, это позволяет кодировать нули без риска получить пропуск в ряду простых чисел.

Все использованные в построении функции и отношения являются примитивно рекурсивными, поэтому отношение $Step(z, u, v) \in \mathcal{PR}_*$.

□

3.2.4. Завершающиеся вычисления и основная теорема

Последовательность конфигураций $I_1, \ldots, I_n$ называется *завершающимся вычислением* регистровой машины M, если I_0 — начальная конфигурация, для любого i машина M допускает переход $I_i \vdash I_{i+1}$, и существует такое $j \leqslant n$, что метка L в конфигурации I_j соответствует инструкции stop машины M.

Заметим, что начальная конфигурация для нормализованной регистровой машины с n входами должна иметь вид

$$1; 0, a_1, \ldots, a_n, \overbrace{0, \ldots, 0}^{r-n-1}.$$

(первой будет выполняться инструкция с меткой 1, выходной регистр имеет нулевое значение, регистры со второго по $n+1$-й содержат входные значения, все остальные (вспомогательные) регистры инициализированы нулями.

Чтобы построить код завершающегося вычисления, можно закодировать последовательность кодов конфигураций:

$$[\mathrm{code}(I_0), \ldots, \mathrm{code}(I_n)].$$

Теорема. *Для любого $n \geqslant 1$ отношение $Comp^{(n)}(z, y)$ — «нормализованная регистровая машина с n входами и кодом z может выполнить завершающееся вычисление с кодом y» — является примитивно рекурсивным.*

Доказательство. Как мы помним, в нормализованной регистровой машине как минимум две инструкции, поэтому количество конфигураций в завершающемся вычислении $l(y) \geqslant 2$ (в простейшем случае вторая конфигурация уже содержит метку инструкции stop, поэтому двигаться дальше не нужно).

Будем также считать, что длина всех конфигураций есть $z+1$, что заведомо больше числа реально используемых регистров, понятно, что добавление в конфигурацию неиспользуемых регистров ничего не испортит.

Зафиксируем произвольное $n \geqslant 1$. При построении отношения $Comp$ необходимо учесть, что z действительно кодирует некоторую регистровую машину, что y является кодом завершающегося вычисления, содержит правильную начальную конфигурацию и конфигурацию с меткой инструкции stop (то есть $l(z)$), а также, что все переходы обоснованы инструкциями машины.

$$
\begin{aligned}
Comp^{(n)}(z, y) \equiv\ & URM(z) \wedge \\
& Seq(y) \wedge l(y) > 1 \wedge (\forall i)_{<l(y)} \big(Seq((y)_i) \wedge l((y)_i) = z + 1 \big) \wedge \\
& ((y)_0)_0 = 1 \wedge ((y)_0)_1 = 0 \wedge (\forall i)_{\leqslant z} (i > n + 1 \longrightarrow ((y)_0)_i = 0) \wedge \\
& \wedge ((y)_{l(y) \dot{-} 1})_0 = l(z) \wedge \\
& \wedge (\forall j)_{<l(y) \dot{-} 1}\, Step(z, (y)_j, (y)_{j+1}).
\end{aligned}
$$

Все использованные в построении функции и отношения являются примитивно рекурсивными, поэтому отношение $Comp^{(n)}(z, y) \in \mathcal{PR}_*$. □

3.2.5. Следствия

Предикат Клини

Предикатом Клини называется отношение $T^{(n)}(z, \bar{x}_n, y)$ — «нормализованная регистровая машина M с n входами и кодом z на наборе значений $\bar{x}_n$ выполняет завершающееся вычисление с кодом y».

Теорема. *Для любого $n \geqslant 1$ предикат Клини $T^{(n)}(z, \bar{x}_n, y)$ является примитивно рекурсивным.*

Доказательство. Выражение для предиката Клини нетрудно построить, воспользовавшись отношением $Comp$ и явно указав значения входных регистров в начальной конфигурации. Зафиксируем n:

$$
T^{(n)}(z, \bar{x}_n, y) \equiv Comp^{(n)}(z, y) \wedge \big(((y)_0)_2 = x_1 \big) \wedge \cdots \wedge \big((((y)_0)_{n+1} = x_n \big).
$$

Ясно, что $T \in \mathcal{PR}_*$. □

Теорема Клини о нормальной форме

Теорема. *Имеют место следующие два утверждения:*

1) *Для того, чтобы нормализованная регистровая машина* $M_{X1}^{X11,\ldots,X1^{n+1}}$ *с кодом* z *и* n *входами была определена на наборе* $\bar{x}_n$ *необходимо и достаточно, чтобы* $(\exists y)T^{(n)}(z,\bar{x}_n,y)$.

2) *Существует такая примитивно рекурсивная функция* d*, что для всех частично рекурсивных функций* $\lambda\bar{x}_n\,.\,f(\bar{x}_n)\in\mathcal{P}$ *существует число* z *и для всех* $\bar{x}_n$

$$f(\bar{x}_n)=d\big((\mu y)T^{(n)}(z,\bar{x}_n,y)\big).$$

Доказательство. Определённость значения регистровой машины на некотором наборе аргументов как раз и означает существование у неё завершающегося вычисления, поэтому первое утверждение теоремы очевидно.

Для доказательства второго утверждения заметим, что у любой частично рекурсивной функции имеется вычисляющая её значения регистровая машина, а значит, и кодирующее её число z, существование завершающегося вычисления гарантирует, что его код будет найден неограниченным поиском, наконец, из найденного вычисления можно извлечь последнюю конфигурацию и содержащееся в ней значение выходного регистра — последним и должна заниматься искомая функция d, не зависящая, следовательно, ни от функции, ни от её аргументов. Поэтому для завершения доказательства остаётся положить

$$d(y)=((y)_{l(y)\dot{-}1})_1\in\mathcal{PR}.$$

Понятно, что если неограниченный поиск приводит к зацикливанию, то есть завершающегося вычисления не существует, то d никогда применена не будет, а функция на соответствующем наборе аргументов не определена. □

Эта теорема называется теоремой о нормальной форме, потому что она действительно указывает нормальную форму для произвольной частично рекурсивной функции: неограниченный поиск по предикату Клини для этой функции и заданного набора аргументов и применение функции d к его результату.

Новая характеризация множества $\mathcal{P}$

В определении множества частично рекурсивных функций $\mathcal{P}$ указывалось, что это любые функции, для вычисления которых можно построить регистровую машину. Теперь мы готовы дать более строгую индуктивную характеризацию этого множества.

Теорема. *Множество* $\mathcal{P}$ *является замыканием множества инициальных функций относительно операторов композиции, примитивной рекурсии и неограниченного поиска.*

Доказательство. Пусть $\widetilde{\mathcal{P}}$ — замыкание множества инициальных функций относительно операторов композиции, примитивной рекурсии и неограниченного поиска. Докажем, что $\widetilde{\mathcal{P}}=\mathcal{P}$.

Поскольку $\mathcal{P}$ замкнуто относительно всех упомянутых в формулировке теоремы операторов, а все инициальные функции частично рекурсивны, то $\widetilde{\mathcal{P}}\subset\mathcal{P}$.

Ясно также, что $\mathcal{PR} \subset \widetilde{\mathcal{P}}$ (второе множество является более широким замыканием). Возьмём произвольную $f \in \mathcal{P}$ и построим её нормальную форму:

$$f(\bar{x}_n) = d\big((\mu y)T^{(n)}(z, \bar{x}_n, y)\big)$$

Поскольку в нормальной форме f участвуют примитивно рекурсивные функции и один раз применён оператор неограниченного поиска, то $f \in \widetilde{\mathcal{P}}$, а значит $\mathcal{P} \subset \widetilde{\mathcal{P}}$. $\square$

Универсальная регистровая машина

Пользуясь удобствами λ-нотации, определим регистровую машину U:

$$U = \lambda z\bar{x}\,.\,d\big((\mu y)T(z, \bar{x}, y)\big),$$

которая принимает на вход код регистровой машины и набор её аргументов, а возвращает её результат. Такую машину можно назвать *универсальной* в том смысле, что она эмулирует любую другую машину, или, иначе, играет роль интерпретатора. Теорема Клини о нормальной форме позволяет утверждать, что

Теорема. *Универсальная регистровая машина существует.*

Каково поведение универсальной машины в ситуациях, когда переданное ей в качестве аргумента число z не кодирует никакую машину? Ясно, что предикат Клини никогда не становится истинным, поэтому машина зацикливается на неограниченном поиске, а значит, мы получаем нигде неопределённую функцию. Такова, к примеру, функция, соответствующая машине с кодом 0.

Перечисление частично рекурсивных функций и ϕ-индексы Роджерса

Ранее было установлено, что любой регистровой машине соответствует однозначно определяемое натуральное число (её код). Мы также знаем, что любой частично рекурсивной функции соответствует некоторая регистровая машина, вычисляющая её значения. Понятно, что одной функции может соответствовать любое число регистровых машин, поскольку нетрудно придумать бесконечно много модификаций машин, вычисляющих одну и ту же функцию: можно, к примеру, в каждой новой модификации присваивать некоторому неиспользуемому в других целях регистру новое натуральное число — код всякий раз будет новым.

С другой стороны, любому натуральному числу соответствует в точности одна регистровая машина, а значит некоторая функция, возможно, нигде не определённая. Это наблюдение позволяет перечислить все частично рекурсивные функции, поставив каждому числу в соответствие некоторую функцию. Соответствующее число называется ϕ-индексом Роджерса этой функции:

$$\forall z \in \mathbb{N} \quad \phi_z^{(n)} = \lambda\bar{x}_n\,.\,d\big((\mu y)T^{(n)}(z, \bar{x}, y)\big).$$

Выражение справа в соответствии с теоремой Клини о нормальной форме перечисляет все частично рекурсивные функции, а значит, все они оказываются перенумерованными ϕ-индексами Роджерса.

3.3. Полурекурсивные отношения и неразрешимые задачи

3.3.1. Полурекурсивные отношения

Обобщим понятия рекурсивных и примитивно рекурсивных отношений на случай всех частично рекурсивных функций.

Отношение $P(\bar{x}_n)$ называется полурекурсивным (полувычислимым), если существует такая функция $f \in \mathcal{P}$, что для всех $\bar{x}_n$ имеет место $P(\bar{x}_n) \equiv f(\bar{x}_n)\downarrow$. Множество всех полурекурсивных отношений по аналогии обозначим символом $\mathcal{P}_*$.

Для того, чтобы работать с полурекурсивными отношениями более удобным образом, сформулируем несколько условий, эквивалентных полурекурсивности.

Теорема. *Следующие условия эквивалентны:*

1) *отношение $P(\bar{x}_n)$ является полурекурсивным;*

2) *существует такое $a \in \mathbb{N}$, что для всех $\bar{x}_n$ $P(\bar{x}_n) \equiv (\exists z)T^{(n)}(a, \bar{x}_n, z)$ (**нормальная форма**);*

3) *существует такое $Q(\bar{x}_n, z) \in \mathcal{R}_*$, что для всех $\bar{x}_n$ $P(\bar{x}_n) \equiv (\exists z)Q(\bar{x}_n, z)$ (**проекция**).*

Доказательство. Докажем сначала направление 1) $\Rightarrow$ 2). Пусть $P(\bar{x}_n)$ — полурекурсивное отношение, а f — соответствующая ему частично рекурсивная функция, такая что для всех $\bar{x}_n$ $P(\bar{x}_n) \equiv f(\bar{x}_n)\downarrow$. По теореме о нормальной форме функция f обладает некоторым индексом Роджерса a, то есть $f = \phi_a^{(n)}$, причём её определённость $(f(\bar{x}_n)\downarrow)$ равносильна в точности $(\exists z)T^{(n)}(a, \bar{x}_n, z)$ для всех $\bar{x}_n$.

Для доказательства направления 2) $\Rightarrow$ 3) достаточно положить

$$Q(\bar{x}_n, z) \equiv \lambda \bar{x}_n z \,.\, T^{(n)}(a, \bar{x}_n, z).$$

Тогда

$$P(\bar{x}_n) \equiv (\exists z)T^{(n)}(a, \bar{x}_n, z) \equiv (\exists z)Q(\bar{x}_n, z).$$

Рекурсивность Q вытекает из рекурсивности предиката Клини T. Отношение Q называют проекцией отношения P на множество рекурсивных отношений.

Наконец, для доказательства 3) $\Rightarrow$ 1) при данном отношении Q положим

$$f = \lambda \bar{x}_n \,.\, (\mu z)Q(\bar{x}_n, z).$$

Функция f очевидно оказывается частично рекурсивной. Если для некоторого z и набора $\bar{x}_n$ отношение Q оказывается истинным, то оператор неограниченного поиска соответствующее z найдёт и функция окажется определённой. Если же требуемого z не существует, то поиск никогда не завершится, а значит функция f для данного $\bar{x}_n$ определена не будет, поэтому

$$P(\bar{x}_n) \equiv (\exists z)Q(\bar{x}_n, z) \equiv f(\bar{x}_n)\downarrow .$$

□

Из этой теоремы вытекает следствие о графике частично рекурсивной функции.

Теорема. *Функция* $\lambda\bar{x}_n\,.\,f(\bar{x}_n) \in \mathcal{P}$ *тогда и только тогда, когда её график* $y = f(\bar{x}_n)$ *— полурекурсивное отношение.*

Доказательство. Начнём с необходимости. Пусть $f = \phi_i^{(n)}$. Если для машины с кодом i имеется завершающееся вычисление z, то результат вычисления y, то есть значение функции $f(\bar{x}_n)$, можно извлечь функцией d из доказательства теоремы Клини о нормальной форме.

Тогда

$$y = f(\bar{x}_n) \equiv (\exists z)\big(\underbrace{T^{(n)}(i, \bar{x}_n, z) \wedge d(z) = y}_{Q(\bar{x}_n, y, z)}\big) \equiv (\exists z)Q(\bar{x}_n, y, z).$$

Отношение Q по построению является примитивно рекурсивным, а значит $Q(\bar{x}_n, y, z) \in \mathcal{R}_*$, следовательно по проекции отношение $y = f(\bar{x}_n)$ полурекурсивно.

В доказательстве достаточности предположим, что график полурекурсивен. Это означает, что существует такое $Q(\bar{x}_n, y, z)$, что $y = f(\bar{x}_n) \equiv (\exists z)Q(\bar{x}_n, y, z)$. Функция f может быть вычислена следующим образом: организуем перебор всех возможных пар вида $[y, z]$ и будем останавливаться на первой удовлетворяющей отношению Q, извлекая из пары значение функции y (первый её компонент).

Простейший способ перебора всех пар состоит в переборе всех натуральных чисел и раскодировании их как пар. Ясно, что такой перебор будет содержать много «мусора», но он также будет содержать все возможные пары, код которых в соответствии с выбранной ранее схемой кодирования имеет вид 2^i3^j.

Таким образом,

$$f(\bar{x}_n) = \big((\mu w)Q(\bar{x}_n, (w)_0, (w)_1)\big)_0.$$

Функция f является частично рекурсивной по построению.

□

3.3.2. Разрешимые и полуразрешимые задачи и их дополнения

Задачей будем называть вопрос вида $\bar{x} \in Q$ или, что то же самое, отношение (предикат) $Q(\bar{x})$. Задача называется разрешимой (рекурсивной), если соответствующее отношение является рекурсивным, то есть для него существует рекурсивная (то есть всегда завершающаяся) характеристическая функция, отвечающая на вопрос о принадлежности элемента множеству «да» или «нет». Неразрешимыми называются задачи, которым соответствуют нерекурсивные отношения.

Помимо «полной» разрешимости можно ввести понятие «полуразрешимости», когда отношения, соответствующие задачам, являются полурекурсивными, то есть имеют характеристическую функцию, завершающуюся при ответе «да» и не завершающуюся в противном случае. В таких задачах мы можем получить подтверждение положительного ответа, но не можем определить отрицательный, поскольку неизвестно, то ли функция через некоторое время всё-таки выдаст ответ, то ли она не вычислится никогда. Таким образом, в случае полуразрешимости мы имеем «полурешатель» или «проверятель», который либо подтверждает положительный ответ, либо зацикливается.

Дополнением задачи $Q(\bar{x}_n)$ называется задача $\neg Q(\bar{x}_n)$. Ясно, что если задача разрешима, то её дополнение также разрешимо: для получения её ответа достаточно «обратить» ответ исходной задачи. Более того, оказывается, что если задача и её дополнение являются полуразрешимыми, то они обе являются разрешимыми. Этот факт нетрудно обосновать алгоритмически. Пусть дан некоторый набор $\bar{x}_n$, для которого необходимо решить задачу, которая вместе со своим дополнением является полуразрешимой. Ясно, что истинно либо $Q(\bar{x}_n)$, либо $\neg Q(\bar{x}_n)$. Запустим для этого набора оба «полурешателя» — один из них гарантированно даст ответ. Если ответ получен от «полурешателя» Q, то ответ «да», в противном случае ответ «нет». Таким образом, мы построили решатель для этой задачи.

Формально эти рассуждения представлены в следующей теореме.

Теорема. *Если* $Q(\bar{x}_n) \in \mathcal{P}_*$ *и* $\neg Q(\bar{x}_n) \in \mathcal{P}_*$, *то* $Q(\bar{x}_n) \in \mathcal{R}_*$ *и* $\neg Q(\bar{x}_n) \in \mathcal{R}_*$.

Доказательство. В соответствии с нормальной формой полурекурсивных отношений $Q(\bar{x}_n) \equiv (\exists z)T^{(n)}(i, \bar{x}_n, z)$ и $\neg Q(\bar{x}_n) \equiv (\exists z)T^{(n)}(j, \bar{x}_n, z)$ для некоторых i и j.

Положим

$$g = \lambda \bar{x}_n \,.\, (\mu z)\big(T^{(n)}(i, \bar{x}_n, z) \vee T^{(n)}(j, \bar{x}_n, z)\big).$$

Ясно, что $g \in \mathcal{P}$, но, поскольку искомое z обязательно существует (либо $Q(\bar{x}_n)$, либо $\neg Q(\bar{x}_n)$ истинно), то $g \in \mathcal{R}$. Функция g является аналогом «решателя» из неформальных рассуждений перед этой теоремой.

Теперь заметим, что $Q(\bar{x}_n) \equiv T^{(n)}(i, \bar{x}_n, g(\bar{x}_n))$ — третий параметр предиката Клини уже является завершающимся вычислением, поэтому его истинность зависит от того, какой это ответ, «да» или «нет». В силу теоремы о подстановке в рекурсивное отношение рекурсивной функции получаем, что $Q \in \mathcal{R}_*$. Рекурсивность отношения $\neg Q$ вытекает теперь из замкнутости множества рекурсивных отношений относительно операции отрицания. □

В качестве следствия из доказанной теоремы покажем, что рекурсивные отношения являются полурекурсивными.

Теорема. $\mathcal{R}_* \subset \mathcal{P}_*$.

Доказательство. Пусть $Q(\bar{x}) \in \mathcal{R}_*$. Введём свежую переменную y. Ясно, что $Q(\bar{x}) \equiv (\exists y)Q(\bar{x})$, поэтому по проекции $Q(\bar{x}) \in \mathcal{P}_*$. □

Это утверждение также имеет простую алгоритмическую интерпретацию: оно соответствует замене отрицательного ответа решателя на бесконечный цикл, что даёт в результате «полурешателя».

3.3.3. Пример неразрешимой задачи: проблема останова

Как следует из теоремы Клини о нормальной форме, все частично рекурсивные функции можно перенумеровать с помощью их ϕ-индексов Роджерса, то есть

$$\mathcal{P} = \{\phi_i : i \in \mathbb{N}\}.$$

Таким образом, всего имеется счётное число частично рекурсивных функций. В то же время задач даже в пределах множества натуральных чисел столько, сколько имеется характеристических функций $f : \mathbb{N} \longrightarrow \{0, 1\}$, то есть столько, сколько существует

различных бесконечных последовательностей, состоящих из нулей и единиц. Число последних, как известно из процесса диагонализации Кантора, несчётно.

Получается, что из общего несчётного числа задач только счётное их число имеет частично рекурсивные характеристические функции, то есть даже полурешатели есть далеко не у всех. Следовательно, неразрешимые задачи существуют. Этот аргумент называют *аргументом кардинальности.*

Проблемой останова называется задача $K = \{x : \phi_x(x)\downarrow\}$ или, словами, завершается ли вычисление функции с заданным кодом на своём коде. Дополнением проблемы останова является задача $\overline{K} = \{x : \phi_x(x)\uparrow\}$.

Теорема. *Проблема останова а) полурекурсивна и б) неразрешима.*

Доказательство. Полурекурсивность проблемы останова вытекает из равносильности утверждения $\phi_x(x)\downarrow$ утверждению $(\exists y)T(x,x,y)$ (по первой части теоремы Клини о нормальной форме). Поскольку предикат Клини примитивно рекурсивен, а значит рекурсивен, то по проекции это отношение оказывается полурекурсивным.

Установим теперь её неразрешимость. Предположим обратное. Если K разрешима, то разрешима, и, следовательно, полурекурсивна задача $\overline{K}$. Это означает, что $x \in \overline{K} \equiv (\exists z)T(i,x,z)$ для некоторого i (по нормальной форме полурекурсивного отношения). Одновременно с этим по определению $x \in \overline{K} \equiv \phi_x(x)\uparrow \equiv \neg(\exists z)T(x,x,z)$.

Проверим теперь i на принадлежность $\overline{K}$ (подставим i вместо x) и получим, что $(\exists z)T(i,i,z) \equiv \neg(\exists z)T(i,i,z)$ — противоречие, доказывающее, что исходное предположение неверно, а значит, проблема останова неразрешима. □

Заметим, что из неразрешимости K вытекает неполурекурсивность $\overline{K}$. Действительно, в противном случае и K, и $\overline{K}$, будучи полурекурсивными, должны были бы оказаться одновременно разрешимыми.

Заметим также, что в силу неразрешимости множества K его характеристическая функция даёт пример невычислимой всюду определённой функции. Эта функция лежит за пределами множества частично вычислимых функций.

3.3.4. Свойства замкнутости $\mathcal{P}_*$

Теорема. *Множество $\mathcal{P}_*$ замкнуто относительно операций $\vee$, $\wedge$, $(\exists y)_{<z}$, $\exists y$ и $(\forall y)_{<z}$ и не замкнуто относительно $\neg$ и $\forall y$.*

Доказательство. Пусть имеются следующие полурекурсивные отношения и их нормальные формы: $P(\bar{x}_n) \equiv (\exists z)T^{(n)}(p,\bar{x}_n,z)$, $Q(\bar{y}_m) \equiv (\exists z)T^{(m)}(q,\bar{y}_m,z)$ и $R(y,\bar{u}_k) \equiv (\exists w)T^{(k+1)}(r,y,\bar{u}_k,w)$. Тогда для проверки замкнутости можно построить следующие проекции:

$$\begin{aligned} P(\bar{x}_n) \vee Q(\bar{y}_m) &\equiv (\exists z)T^{(n)}(p,\bar{x}_n,z) \vee (\exists z)T^{(m)}(q,\bar{y}_m,z) \\ &\equiv (\exists z)\big(T^{(n)}(p,\bar{x}_n,z) \vee T^{(m)}(q,\bar{y}_m,z)\big); \end{aligned}$$

$$\begin{aligned} P(\bar{x}_n) \wedge Q(\bar{y}_m) &\equiv (\exists z)T^{(n)}(p,\bar{x}_n,z) \wedge (\exists z)T^{(m)}(q,\bar{y}_m,z) \\ &\qquad (\textit{среди двух существующих } z \textit{ есть наибольшее}) \\ &\equiv (\exists w)\big((\exists z)_{<w}T^{(n)}(p,\bar{x}_n,z) \wedge (\exists z)_{<w}T^{(m)}(q,\bar{y}_m,z)\big); \end{aligned}$$

$$(\exists y)_{<z}R(y,\bar{u}_k) \equiv (\exists y)_{<z}(\exists w)T^{(k+1)}(r,y,\bar{u}_k,w)$$
$$\text{(одноимённые кванторы коммутируют)}$$
$$\equiv (\exists w)(\exists y)_{<z}T^{(k+1)}(r,y,\bar{u}_k,w);$$

$$(\exists y)R(y,\bar{u}_k) \equiv (\exists y)(\exists w)T^{(k+1)}(r,y,\bar{u}_k,w)$$
$$\text{(вновь существует наибольшее)}$$
$$\equiv (\exists z)(\exists y)_{<z}(\exists w)_{<z}T^{(k+1)}(r,y,\bar{u}_k,w);$$

$$(\forall y)_{<z}R(y,\bar{u}_k) \equiv (\forall y)_{<z}(\exists w)T^{(k+1)}(r,y,\bar{u}_k,w)$$
$$\text{(среди ограниченного числа } w \text{ существует наибольшее)}$$
$$\equiv (\exists v)(\forall y)_{<z}(\exists w)_{<v}T^{(k+1)}(r,y,\bar{u}_k,w).$$

Нетрудно видеть, что все проекции (выражения под кванторами существования) в силу замкнутости $\mathcal{R}_*$ относительно логических операций и ограниченной квантификации являются рекурсивными.

Контрпримером для замкнутости относительно отрицания служит задача $K \in \mathcal{P}_*$, дополнение которой $\overline{K}$ не полурекурсивно. Контрпримером для замкнутости относительно квантора всеобщности служит отношение $\neg T(x,x,y) \in \mathcal{P}_*$, поскольку $(\forall y)\neg T(x,x,y) \equiv \neg(\exists y)T(x,x,y) \equiv x \in \overline{K}$. □

В качестве следствия приведём теорему о подстановке, аналогичную соответствующим утверждениям для $\mathcal{PR}_*$ и $\mathcal{R}_*$.

Теорема. *Если* $\lambda\bar{x}_n\,.\,f(\bar{x}_n) \in \mathcal{P}$ *и* $Q(z,\bar{y}) \in \mathcal{P}_*$, *то* $Q(f(\bar{x}),\bar{y}) \in \mathcal{P}_*$.

Доказательство. $Q(f(\bar{x}),\bar{y}) \equiv (\exists z)\big(z = f(\bar{x}) \wedge Q(z,\bar{y})\big) \in \mathcal{P}_*$. □

3.3.5. Сводимость

В этом пункте мы рассмотрим ещё два примера неразрешимых задач, обращая внимание на метод, с помощью которого их неразрешимость устанавливается.

Рассмотрим задачу $K_0 = \{[x,y] : \phi_x(y)\downarrow\}$. Предположим, что она разрешима. Тогда характеристическая функция $\lambda xy\,.\,\chi_{K_0}(x,y) \in \mathcal{R}$. Выполним в ней отождествление переменных x и y и в соответствии с принципом подстановки Гржегорчика получим новую рекурсивную функцию $\lambda x\,.\,\chi_{K_0}(x,x) \in \mathcal{R}$, которая является характеристической функцией множества K. Следовательно, проблема останова разрешима, и мы приходим к противоречию. Значит, K_0 неразрешима.

Проследим последовательность рассуждений. Первым делом мы показываем, что задача A (в нашем случае K) *сводится* к задаче B (в нашем случае K_0):

1) имеется алгоритм, решающий задачу B (рекурсивная характеристическая функция χ_{K_0});
2) с его помощью строится алгоритм, решающий задачу A (рекурсивная характеристическая функция для K — строится посредством отождествления переменных).

Благодаря *сводимости* задачу A можно решать с помощью «решателя» задачи B. Поскольку ранее было доказано, что задача A неразрешима, то из сводимости следует, что задача B также неразрешима, ведь иначе задачу A можно было бы решить, воспользовавшись «решателем» B.

Рассмотрим теперь *проблему эквивалентности программ*: имеются две программы, требуется выяснить, одну ли функцию они вычисляют. Ограничимся для простоты программами, вычисляющими примитивно рекурсивные функции. Предположим, что эта проблема разрешима. Проверим посредством её «решателя» эквивалентность двух программ: $\lambda y\,.\,1$ и $\lambda y\,.\,\chi_T(x,x,y)$. Математически факт эквивалентности можно выразить следующим образом:

$$(\forall y)(1 = \chi_T(x,x,y)) \equiv (\forall y)(\neg T(x,x,y)) \equiv \neg(\exists y)T(x,x,y) \equiv x \in \overline{K}.$$

Таким образом, мы получили «решатель» для $\overline{K}$, то есть задача $\overline{K}$ сведена к проблеме эквивалентности программ. Следовательно, поскольку $\overline{K}$ даже не полурекурсивна, то и исходная проблема не является полурекурсивной.

3.3.6. Рекурсивная перечислимость и проблема проверки рекурсивности

Некоторые множества характерны тем, что их элементы можно *перечислить*, то есть поставить в соответствие натуральным числам. Ограничим себя подмножествами множества натуральных чисел и «простыми» функциями-перечислителями, скажем, примитивно рекурсивными. Оказывается, есть непосредственная связь между полурекурсивностью таких множеств и их *рекурсивной перечислимостью*.

Предположим, что элементы некоторого множества можно перечислить. Является ли оно полурекурсивным, то есть можно ли для него построить «проверятель»? Да, например, так: при решении проблемы принадлежности элемента x множеству запускаем перечисление и если элемент x в таком перечислении появляется, то даём положительный ответ, если элемент никогда не появляется, то «проверятель» не завершается, как и должно быть в случае полурекурсивности.

Если, теперь, у нас имеется полурекурсивное множество, можно ли его элементы перечислить? Действовать напрямую, проверяя последовательно все натуральные числа нельзя, поскольку при такой проверке мы можем сразу зациклиться и вообще ничего не перечислить. Идея, однако, состоит в том, чтобы перебирать пары чисел x и y и для каждой из них отвечать на вопрос «можно ли установить ответ для x за y шагов». Таким образом, вычисление для каждой пары будет гарантированно конечным, а в результате перебора мы рано или поздно получим все элементы множества, каждый, правда, бесконечно много раз (если ответ можно найти за y шагов, то его же можно найти и за $y+1$ шаг).

Формализуем эти рассуждения. Множество $S(\subset \mathbb{N})$ называется *рекурсивно перечислимым*, если оно либо пусто, либо является областью значений некоторой примитивно рекурсивной функции $f : f(\mathbb{N}) = S$. Заметим, что мы никак не ограничиваем количество повторений элемента во множестве значений. Если множество S конечно, то все элементы неминуемо будут повторяться бесконечное число раз.

Теорема. *Для того чтобы непустое множество $S(\subset \mathbb{N})$ было полурекурсивным необходимо и достаточно, чтобы оно было рекурсивно перечислимым.*

Доказательство. Докажем сначала достаточность. Пусть $f \in \mathcal{PR}$ — функция-перечислитель, $f(\mathbb{N}) = S$. Отношение $y \in S$ равносильно $(\exists x) f(x) = y$ и, так как график примитивно рекурсивной функции является примитивно рекурсивным отношением, по проекции получаем, что $S \in \mathcal{P}_*$.

Теперь докажем необходимость. Пусть S — полурекурсивно и

$$x \in S \equiv (\exists y) T(i, x, y) \qquad (*)$$

для некоторого i. Идея построения функции-перечислителя состоит в том, чтобы перебирать закодированные пары чисел (элемент, завершающееся вычисление). Число шагов из неформальных обсуждений в начале этого пункта напрямую переносится в перебор завершающихся вычислений, поскольку чем больше код вычисления, тем больше конфигураций (шагов) в него попадает. Определим, таким образом:

$$f(z) = \begin{cases} (z)_0, \text{если } T(i, (z)_0, (z)_1) \\ a, \text{где } a \text{ — фиксированный элемент } S, \text{ в противном случае.} \end{cases}$$

При переборе всех z мы перебираем все возможные пары $[(z)_0, (z)_1]$, а также массу некорректных кодов, на которых предикат Клини будет ложным.

Для завершения доказательства теоремы остаётся убедиться, что $f(\mathbb{N}) = S$. С одной стороны, $f(\mathbb{N})$ содержит только те значения, на которых регистровая машина с кодом i останавливается, поэтому из $(*)$ следует, что $f(\mathbb{N}) \subset S$. С другой стороны, из принадлежности $x \in S$ вытекает, что существует некоторое b такое, что $T(i, x, b)$. Поскольку перебираются все возможные пары, то пара $[x, b]$ рано или поздно попадётся, а значит, $x = f([x, b])$ окажется в образе $f(\mathbb{N})$. □

В качестве примера применения понятия рекурсивной перечислимости множества покажем, что задача $\mathscr{R} = \{x : \phi_x \in \mathcal{R}\}$ не является полурекурсивной. Предположим, что множество $\mathscr{R}$ рекурсивно перечислимо, то есть $\mathscr{R} = f(\mathbb{N})$ для некоторой примитивно рекурсивной f и, следовательно, $\{\phi_{f(x)} : x \in \mathbb{N}\} = \mathcal{R}$. Рассмотрим функцию $d = \lambda x \,.\, \phi_{f(x)}(x) + 1$ — эта функция, очевидно, рекурсивна, поэтому она также перечисляется функцией f. Пусть, к примеру, $d = \phi_{f(i)}$. Вычислим теперь $d(i)$: с одной стороны, $d(i) = \phi_{f(i)}(i)$, но с другой, по определению, $d(i) = \phi_{f(i)}(i) + 1$. Таким образом, $\phi_{f(i)}(i) = \phi_{f(i)}(i) + 1$ и $0 = 1$. Полученное противоречие показывает, что исходное предположение неверно, следовательно множество $\mathscr{R}$ не является рекурсивно перечислимым, а значит, не является и полурекурсивным, тем более разрешимым.

Сформулируем этот результат более просто: не существует алгоритма, который бы в общем случае определял, завершается ли заданная программа на всех входах. Более того, для таких программ невозможно даже получить подтверждение положительного ответа.

Неполурекурсивность множества ϕ-индексов всюду определённых частично рекурсивных функций является причиной отсутствия удобной характеризации множества $\mathcal{R}$, которые были прежде получены для $\mathcal{PR}$ и $\mathcal{P}$.

3.4. Теорема Райса

Целью этого раздела является доказательство теоремы Райса о неразрешимости практически всех задач, касающихся множества частично рекурсивных функций. На

этом пути нужно будет предварительно сформулировать и доказать пару важных технических результатов — S-m-n-теорему Клини и теорему о рекурсии.

3.4.1. S-m-n-теорема Клини

Предположим, что перед нами стоит следующая задача: необходимо зафиксировать второй параметр функции $\lambda xy\,.\,f(x,y)$, получив из неё одноаргументную функцию $\lambda x\,.\,f(x,a)$, где a — подставляемая константа.

Пусть M_z^{xy} — машина, вычисляющая двухаргументную функцию f. Чтобы преобразовать её требуемым образом, достаточно исключить y из списка входных параметров, добавить в начало машины инструкцию $y \leftarrow a$ и перенумеровать все следующие за ней инструкции вместе с метками в инструкциях типа if. В результате мы получим машину, которая по входному аргументу x вычисляет результат, используя y, изначально равный a. Эти рассуждения лежат в основе техники доказательства S-m-n-теоремы.

Начнём формальные построения с определения операции конкатенации кодов. Положим

$$x * y = x \cdot \prod_{i<l(y)} p_{i+l(x)}^{\exp(i,y)}.$$

Нетрудно видеть, что определённая таким образом операция удовлетворяет нашим ожиданиям относительно конкатенации кодов двух числовых последовательностей:

$$[a_1, \ldots, a_n] * [b_1, \ldots, b_m] = [a_1, \ldots, a_n, b_1, \ldots, b_m]$$

Пусть теперь M и N — регистровые машины с кодами m и n, соответственно. Конкатенацию MN двух машин, выполняемую над их кодами, обозначим символом $m \frown n$ и определим следующими правилами:

1) удаляется последняя инструкция машины M (stop);
2) список инструкций машины N присоединяется к списку инструкций машины M;
3) номера инструкций и метки в if для инструкций машины N перенумемеровываются соответствующим образом.

Введём функцию $\mathrm{adj}(n,k)$ (от англ. adjust — настроить), задача которой — исправить инструкции машины с кодом n так, чтобы ко всем меткам (номера и их упоминания в if) прибавилось число k.

Лемма. *Функция* $\lambda nk\,.\,\mathrm{adj}(n,k)$ *является примитивно рекурсивной.*

Доказательство. Решим сначала задачу попроще, реализовав функцию $f(z,k)$, которая исправляет требуемым образом одну инструкцию с кодом z:

$$f(z,k) = \begin{cases} 3^k z, \text{если } (\exists L,i,a)_{\leqslant z}(z = [1,L,i,a] \vee z = [2,L,i] \vee z = [3,L,i] \vee z = [5,L]); \\ 231^k z, \text{если } (\exists L,i,P,Q)_{\leqslant z} z = [4,L,i,P,Q]; \\ 0, \text{в противном случае.} \end{cases}$$

Для всех инструкций кроме if (первый случай) достаточно исправить номер, который хранится во второй позиции, а значит, соответствует показателю степени тройки.

В случае if (второй случай) необходимо дополнительно исправить показатели при 7 и 11 ($231 = 3 \cdot 7 \cdot 11$). Домножая на соответствующие степени, мы увеличиваем закодированное число на k. Если z не является корректным кодом, функция возвращает 0. Ясно, что функция f является примитивно рекурсивной.

Теперь можно определить функцию adj, которая должна применить f ко всем инструкциям данной машины:

$$\mathrm{adj}(n,k) = \begin{cases} \prod\limits_{i<l(n)} p_i^{f((n)_i,k)+1}, \text{если } \mathrm{URM}(n); \\ 0, \text{в противном случае.} \end{cases}$$

мы здесь извлекаем i-ю инструкцию, преобразуем её посредством f и возвращаем обратно в код, длина машины (количество инструкций) при этом никак не меняется. Ясно, что полученная таким образом функция является примитивно рекурсивной. □

Лемма. $\lambda mn\,.\,m \frown n \in \mathcal{PR}$.

Доказательство. Для построения $m \frown n$ теперь достаточно удалить инструкцию stop первой машины и сконкатенировать остаток с преобразованной функцией adj второй машиной:

$$m \frown n = \begin{cases} \left\lfloor \frac{m}{p_{l(m)\dot{-}1}^{\exp(l(m)\dot{-}1,m)}} \right\rfloor * \mathrm{adj}(n, l(m) \dot{-} 1), \text{если } \mathrm{URM}(m) \wedge \mathrm{URM}(n); \\ 0, \text{в противном случае.} \end{cases}$$

Ясно, что эта функция также примитивно рекурсивна. □

Теорема (S-m-n-теорема Клини)**.** *Для любых $m \geqslant 1$ и $n \geqslant 1$ существует такая функция $\lambda i \bar{y}_n\,.\,S_n^m(i, \bar{y}_n) \in \mathcal{PR}$, что для всех i, $\bar{x}_m$ и $\bar{y}_n$*

$$\phi_i^{(m+n)}(\bar{x}_m, \bar{y}_n) = \phi_{S_n^m(i,\bar{y}_n)}^{(m)}(\bar{x}_m).$$

Доказательство. Зафиксируем m и n. Пусть $\phi_i^{(m+n)} = M_{x1}^{x11,\ldots,x1^{m+n+1}}$ (то есть M — машина с кодом i). Для преобразования машины M необходимо удалить аргументы с $x1^{m+2}$ по $x1^{m+n+1}$ и вставить в её начало машину N следующего вида:

$$\begin{aligned} 1 &: x1^{m+2} \leftarrow y_1 \\ &\ldots \\ n &: x1^{m+n+1} \leftarrow y_n \\ n+1 &: \mathrm{stop} \end{aligned}$$

где $\bar{y}_n$ — подставляемые значения. Код машины N есть функция от $\bar{y}_n$:

$$\mathrm{init}(\bar{y}_n) = p_0^{[1,1,m+2,y_1]+1} \cdots p_{n-1}^{[1,n,m+n+1,y_n]+1} \cdot p_n^{[5,n+1]+1}.$$

Ясно, что $\mathrm{init} \in \mathcal{PR}$. Поскольку кодом машины M является i, то теперь достаточно выполнить конкатенацию машин:

$$S_n^m(i, \bar{y}_n) = \mathrm{init}(\bar{y}_n) \frown i.$$

Нетрудно видеть, что $S_n^m \in \mathcal{PR}$. □

S-m-n-теорема обычно применяется в следующем виде.

Следствие. *Если* $\lambda \bar{x}_k y \bar{z}_r \,.\, f(\bar{x}_k, y, \bar{z}_r) \in \mathcal{P}$*, то существует такая функция* $h \in \mathcal{PR}$*, что*

$$f(\bar{x}_k, y, \bar{z}_r) = \phi^{(k+r)}_{h(y)}(\bar{x}_k, \bar{z}_r).$$

Доказательство. Действительно, из частичной рекурсивность функции f следует существование такого i, что $f(\bar{x}_k, y, \bar{z}_r) = \phi^{(k+r+1)}_i(\bar{x}_k, \bar{z}_r, y)$. Здесь следует обратить внимание на то, что y стоит на последнем месте: ранее утверждалось, что для любой функции существует бесконечно много кодов, поскольку существует бесконечно много машин, её вычисляющих. Ясно, что среди этих машин будет и такая, которая принимает y последним аргументом.

Теперь остаётся применить S-m-n-теорему к $\phi^{(k+r+1)}_i$ и положить $h = \lambda y \,.\, S^{k+r}_1(i, y)$. □

3.4.2. Теорема о рекурсии

Теорема (о рекурсии)**.** *Если* $\lambda z \bar{x}_n \,.\, f(z, \bar{x}_n) \in \mathcal{P}$*, то для некоторого* e *для всех* $\bar{x}_n$

$$\phi^{(n)}_e(\bar{x}_n) = f(e, \bar{x}_n).$$

Доказательство. Введём в рассмотрение функцию $\lambda z \bar{x}_n \,.\, f(S^n_1(z, z), \bar{x}_n)$, пусть a — её индекс Роджерса, то есть

$$\phi^{(n+1)}_a = \lambda z \bar{x}_n \,.\, f(S^n_1(z, z), \bar{x}_n).$$

Вычислим теперь $f(S^n_1(a, a), \bar{x}_n) = \phi^{(n+1)}_a(a, \bar{x}_n) = \phi^{(n)}_{S^n_1(a,a)}(\bar{x}_n)$ (во втором равенстве примененена S-m-n-теорема) и положим $e = S^n_1(a, a)$. □

Попытаемся дать «программистскую» интерпретацию этой теоремы: любая программа на $n+1$ входе является интерпретатором некоторой программы с n входами, переданной ей в качестве первого аргумента.

Теорема о рекурсии может дать пример куайна (программы, которая возвращает свой текст, игнорируя входные данные). Пусть $\psi = \lambda xy \,.\, x$. Тогда существует e: $\phi_e(y) = \psi(e, y) = e$ для любого y, так что найденная ϕ_e и есть куайн.

Следствие (теорема о рекурсии в форме Роджерса)**.** *Если* $\lambda x \,.\, g(x) \in \mathcal{R}$*, то существует* e *такое, что* $\phi_{g(e)} = \phi_e$*.*

Доказательство. Пусть $\phi_{g(x)}(y) = f(x, y) \in \mathcal{P}$. Применим к f теорему о рекурсии: существует e: $f(e, y) = \phi_e(y)$. □

Это следствие можно интерпретировать следующим образом: каково бы ни было преобразование программ g (это функция на кодах программ, поэтому мы назвали его преобразованием программ), существует программа, которая при этом преобразовании останется на месте (окажется неподвижной точкой преобразования).

3.4.3. Теорема Райса

Теорема Райса является центральным результатом этого раздела.

Теорема (Райса)**.** *Пусть $\mathcal{C} \subset \mathcal{P}$ — некоторое множество частично рекурсивных функций. Множество $A = \{x : \phi_x \in \mathcal{C}\}$ индексов Роджерса функций из $\mathcal{C}$ является разрешимым (рекурсивным) тогда и только тогда, когда оно тривиально (является пустым либо совпадает со всем $\mathbb{N}$).*

Из этой теоремы следует, что любая попытка определить, обладает ли заданная частично рекурсивная функция некоторым свойством (принадлежит некоторому множеству функций), является неразрешимой задачей, если имеются функции, как обладающие этим свойством, так и не имеющие его. Таким образом, действительно, практически все задачи, связанные с частично рекурсивными функциями, неразрешимы. Важно понимать, что множество индексов из условия теоремы обязательно включает все индексы одной и той же функции.

Доказательство. Достаточность доказывается очень легко: у тривиальных множеств имеются очевидно рекурсивные (всюду определённые) характеристические функции $\chi_\varnothing = \lambda x\,.\,1$ и $\chi_{\mathbb{N}} = \lambda x\,.\,0$.

Для доказательства необходимости предположим, что $A \in \mathcal{R}_*$ (разрешимо) и, тем не менее, нетривиально, то есть существуют $a \in A$ и $b \notin A$. Определим функцию

$$f(x) = \begin{cases} b, \text{если } x \in A; \\ a, \text{если } x \notin A. \end{cases}$$

Из разрешимости A следует, что это функция из $\mathcal{R}$. Она обладает следующим свойством:

$$x \in A \Leftrightarrow f(x) \notin A \text{ для всех } x. \qquad (*)$$

По следствию из теоремы о рекурсии существует такое e, что $\phi_{f(e)} = \phi_e$, то есть e должно принадлежать множеству A одновременно с $f(e)$ (эти два кода соответствуют одной и той же функции), что противоречит свойству $(*)$. Обнаруженное противоречие завершает доказательство теоремы. □

3.5. Альтернативное построение теории вычислимости

Теория вычислимости может строиться на основе самых разных моделей вычисления. Здесь мы избрали путь на основе регистровых машин и частично рекурсивных функций. Можно было бы делать то же самое, пользуясь машинами Тьюринга и задачей распознавания языков. Введённые ранее понятия рекурсивных и рекурсивно перечислимых языков соответствуют понятиям рекурсивных и рекурсивно перечислимых множеств (рекурсивных и полурекурсивных отношений). Для машины Тьюринга также можно определить способ кодирования и построить универсальную машину, цель которой — отвечать на вопрос, распознаёт ли заданная машина заданное слово. Множество всех пар «машина — распознаваемое ею слово» составляют так называемый универсальный язык, который оказывается неразрешимым. Из его

неразрешимости можно вывести проблему останова для машин Тьюринга и другие важные результаты теории вычислимости, например, теорему Райса. В этом случае теорема Райса формулируется для «свойств рекурсивно перечислимых языков» (то есть подмножеств множества всех рекурсивно перечислимых языков), в ней утверждается, что всякое нетривиальное свойство неразрешимо.

Познакомиться с таким подходом к построению теории вычислимости можно, к примеру, по учебнику Хопкрофта, Мотвани и Ульмана «Введение в теорию автоматов, языков и вычислений» (глава 9).

3.6. Неразрешимые задачи за пределами теории алгоритмов

3.6.1. Задача разрешимости и теорема Гёделя о неполноте

Установленная в разделе 3.3.3 неразрешимость проблемы останова можно использовать для доказательства двух крайне важных результатов математической логики — неразрешимости исчисления предикатов первого порядка (гильбертовская *Entscheidungsproblem*) и первой теоремы Гёделя о неполноте. Мы не будем их доказывать, приведём лишь идею, лежащую в основе обоих доказательств. Дело в том, что утверждение $\phi_x(x) \downarrow$ можно представить в виде формулы в логике предикатов первого порядка, содержащей аксиомы формальной арифметики. Эта формула содержит переменные, кванторы, простые арифметические операции и неравенства.

Если бы логика предикатов первого порядка была разрешимой, то есть существовал бы алгоритм, позволяющий определить по заданной формуле, является ли она доказуемой, то можно было бы применить этот алгоритм к формуле, описывающей проблему останова. Её истинность означала бы, что вычисление останавливается, а ложность — что зацикливается. Тем самым была бы установлена и разрешимость проблемы останова, что на самом деле не имеет места. Таким образом, *Entscheidungsproblem* неразрешима.

Далее, первая теорема Гёделя о неполноте (в сильно упрощённой формулировке) утверждает, что формальная система, содержащая арифметику и допускающая перечисление (вообще говоря бесконечное) всех доказуемых утверждений, не может быть одновременно полной и непротиворечивой. Напомним, что полнота формальной системы означает, что для любого утверждения доказуемо либо оно, либо его отрицание, непротиворечивость же требует, чтобы доказуемым было только одно из них. Если, теперь, запустить перечисление всех доказуемых утверждений в данной формальной системе, то в таком перечислении рано или поздно появится утверждение об останове, либо его отрицание. Полнота гарантирует, что что-то такое появится, а непротиворечивость — что не появятся одновременно утверждение и его отрицание. Следовательно, мы снова получаем ответ на проблему останова. Отсюда делается вывод, что такая формальная система не может быть одновременно полной и непротиворечивой.

Разумеется, приведённое изложение, касающееся применения неразрешимости в математической логике, является крайне поверхностным, однако его полезно иметь в виду, оно даёт представление о важности применения результатов теории вычислимости в математике и современной науке в целом.

3.6.2. Примеры других неразрешимых задач

Проблема соответствий Поста

Пусть имеется конечный алфавит Σ и два набора слов этого алфавита

$$A = (a_1, a_2, \ldots, a_n) \text{ и } B = (b_1, b_2, \ldots, b_n).$$

Проблемой соответствий Поста называется задача определения, существует ли такая последовательность индексов $i_1, i_2, \ldots, i_k$, что

$$a_{i_1} a_{i_2} \ldots a_{i_k} = b_{i_1} b_{i_2} \ldots b_{i_k}.$$

Экземпляр проблемы соответствий Поста (ПСП) содержит конкретный алфавит и оба набора слов. Каждый конкретный экземпляр обычно можно проанализировать и либо найти решение, либо доказать, что его не существует. Относительно же общего случая можно доказать, что эта проблема неразрешима, то есть алгоритма, который бы по заданным A и B устанавливал наличие или отсутствие решения, не существует.

Десятая проблема Гильберта

Десятая проблема Гильберта состоит в нахождении универсального метода отыскания решения произвольного алгебраического диофантова уравнения (в целых числах). Эта задача оказалась также неразрешимой, что было окончательно установлено советским российским математиком Юрием Матиясевичем в 1970 году, использовавшим результаты работы других математиков, в первую очередь Мартина Дэвиса, Хилари Патнем и Джулии Робинсон.

Проблема смертности во множестве квадратных матриц

Пусть дано конечное множество квадратных матриц одинакового размера с целочисленными элементами. Требуется определить, можно ли построить такое их произведение, возможно, с повторениями, чтобы в результате получилась нулевая матрица.

Известно, к примеру, что для матриц размера 3×3 эта проблема неразрешима, если во множестве имеется шесть или более матриц, а для матриц 15×15 проблема неразрешима даже для множеств с двумя матрицами.

Другие примеры

Богатым источником неразрешимых задач является теория групп (например, задача об изоморфизме групп). Известны также задачи усердного бобра (busy beaver), замощения плоскости плитками Вана, вывода типов в λ-исчислении с полиморфизмом второго порядка, существования нормальной формы λ-терма, эквивалентности контекстно-свободных грамматик и пр.

4. Теория сложности

Теория сложности занимается изучением разрешимых задач, в её рамках исследуются ресурсы, необходимые для получения решения. В качестве основных ресурсов обычно рассматривают время (например, количество шагов, которые делает машина Тьюринга) и пространство (объём задействованной памяти, например, длина ленты машины Тьюринга). В соответствии с этим выделяют *временну́ю* и *пространственную* сложности. В некоторых случаях вычислительным моделям добавляются дополнительные возможности, например, получение случайных чисел или обращение к оракулам (устройство, умеющее давать ответ на некоторый конкретный вопрос). Задачи в зависимости от необходимых ресурсов классифицируют по *классам сложности*. Теория сложности изучает эти классы и соотношения между ними.

4.1. Временная сложность: классы P, NP и EXP

В качестве средства для анализа временной сложности задачи используется зависимость между длиной входа машины Тьюринга и количеством шагов, необходимым для получения ответа.

Пусть $T : \mathbb{N} \longrightarrow \mathbb{N}$ — некоторая функция. Говорят, что язык L принадлежит классу $\mathbf{DTIME}(T(n))$, если существует распознающая язык L машина Тьюринга, которая на входном слове длины n делает не более $cT(n)$ шагов, где $c(> 0)$ — некоторая константа.

Название класса **DTIME** означает «детерминированное время», то есть здесь речь идёт о решении задач детерминированной машиной Тьюринга.

4.1.1. Классы P и EXP

Обычно считается, что наиболее приемлемой оценкой для времени вычисления является полином. Если задача имеет полиномиальный алгоритм, то можно получить ответ за разумное время даже для очень больших экземпляров. Поэтому имеет смысл определить класс задач *полиномиальной сложности* (*класс* **P**):

$$\mathbf{P} = \bigcup_{c \geqslant 1} \mathbf{DTIME}(n^c).$$

Понятно, что каждый следующий класс в объединении содержит все предыдущие. Аналогично можно определить класс задач, для решения которых требуется *экспоненциальное* время:

$$\mathbf{EXP} = \bigcup_{c \geqslant 1} \mathbf{DTIME}(2^{n^c}).$$

Ясно, что $\mathbf{P} \subset \mathbf{EXP}$. Задачи, принадлежащие классу **EXP**, но не принадлежащие классу **P**, считаются *практически* неразрешимыми, поскольку время, необходимое

для их решения даже в случае очень небольших экземпляров, может быть крайне велико. Разумеется, в случае полинома очень большой степени дожидаться результата всё равно придётся долго, однако практика показывает, что алгоритмы со степенями 10, 20 или даже 100 обычно со временем оптимизируются до разумных степеней, например 3 или 5.

Понятие временной сложности было введено для машин Тьюринга, однако оно может быть распространено и на другие модели посредством тезиса Чёрча—Тьюринга в сильной форме: любая вычислимая функция может быть вычислена на машине Тьюринга с не более чем полиномиальным замедлением (то есть t шагов вычисления могут быть выполнены машиной Тьюринга за не более чем t^c шагов для некоторой константы c, зависящей от конкретного способа вычислений). Для всех рассмотренных ранее моделей вычислений соответствующее утверждение действительно справедливо.

Приведём пример задачи, принадлежащей классу **P**: «умножение целых чисел». Понятно, что в традиционной постановке эта задача не является задачей разрешения, поэтому её следует переформулировать. Введём в рассмотрение следующий язык:

$$\{(x, y, i) : i\text{-й бит произведения } xy \text{ равен } 1\}$$

Время вычисления должно рассчитываться от количества битов во входном слове. Понятно, что эта задача может быть решена за полиномиальное время с помощью почти любого алгоритма умножения.

4.1.2. Класс NP

Принято считать, что решить задачу гораздо сложнее, чем проверить правильность данного решения. До этого рассматривался класс задач, которые могут быть решены за полиномиальное время. Перейдём теперь к классу задач, решение которых может быть проверено за полиномиальное время. Для этого класса используется обозначение **NP**, смысл которого станет понятен позднее.

Говорят, что язык $L \subset \{0,1\}^*$ принадлежит классу **NP**, если существуют многочлен $p : \mathbb{N} \longrightarrow \mathbb{N}$ и работающая полиномиальное время машина Тьюринга M, такие, что для всех слов $x \in \{0,1\}^*$ длины n

$$x \in L \text{ тогда и только тогда, когда существует } u \in \{0,1\}^{p(n)} : M(x,u) = 1.$$

Машина M называется *проверяющей* машиной, ей на вход помимо входного слова подаётся *сертификат*, некоторое двоичное слово, позволяющее проверить принадлежность входного слова рассматриваемому языку за полиномиальное время. Сертификат обычно содержит некоторую информацию о решении, изучая которую нетрудно установить, что входное слово действительно принадлежит языку. Следует иметь в виду, что длина сертификата должна быть ограничена полиномом от длины входного слова.

Рассмотрим важные примеры задач из класса **NP**:

INDSET *Независимым множеством* графа называется множество тех вершин, никакие две из которых не связаны между собой дугами. В теории графов обычно рассматривают задачу определения размера максимального независимого множества, то есть множества, содержащего максимально возможное количество

вершин. В версии задачи разрешения постановка звучит так: необходимо проверить, существует ли в заданном графе G независимое множество размера k. Для решения задачи в общем случае достаточно запустить цикл от 1 до количества вершин в графе и на каждом шаге запускать задачу разрешения для счётчика цикла. Последний положительный ответ даст размер максимального множества.

Входом задачи является граф G и число k. Для проверки её принадлежности классу **NP** достаточно обосновать существование достаточно короткого сертификата, позволяющего удостовериться в том, что в графе действительно имеется независимое множество требуемого размера. В качестве такого сертификата можно использовать просто перечень вершин, составляющих такое множество. Понятно, что длина сертификата будет не более чем полиномиально зависеть от длины входа (в зависимости от способа представления графа). Нетрудно также придумать работающий полиномиальное время алгоритм, проверяющий, действительно ли вершины из сертификата составляют независимое множество: для этого достаточно, к примеру, организовать двойной цикл по вершинам множества и для каждой пары вершин проверять, связаны ли они дугой. Ясно, что сертификат такого вида существует тогда и только тогда, когда в графе имеется независимое множество требуемого размера.

TSP В *задаче коммивояжёра* требуется обнаружить максимально короткий цикл, проходящий по одному разу через все вершины неориентированного графа. Как задача разрешения она формулируется следующем образом: необходимо проверить, имеется ли в графе G цикл длины k. Понятно, что сертификатом может служить сам такой цикл, то есть перечень вершин, через которые он проходит. Для проверки сертификата достаточно перебрать вершины цикла, посчитать его длину и удостовериться, что каждая вершина графа обходится в точности один раз. Ясно, что такая проверка требует не более чем полиномиального времени.

GCONN Требуется по заданному графу G и двум вершинам u и v определить, имеется ли между ними какой-либо путь. Сертификатом может служить собственно путь, проверить который можно очень быстро.

GI Предположим, что имеется два графа G_1 и G_2, представленных матрицами смежности одинакового размера. В задаче об *изоморфизме графов* требуется проверить, можно ли перенумеровать вершины одного из графов таким образом, чтобы получился в точности второй граф. Сертификатом может случать выполняющая требуемое переименование перестановка: достаточно применить её к матрице смежности графа G_1 и проверить две матрицы на равенство.

SUBSETSUM Дано множество целых чисел $A = \{a_1, a_2, \ldots, a_n\}$ и целое число T. Необходимо выяснить, существует ли такое подмножество множества A, сумма чисел которого равна T. Сертификатом является само подмножество: ясно, что его длина короче входа, а посчитать сумму его элементов можно очень быстро.

COMPOSITE Дано целое число. Требуется выяснить, является ли оно составным. Сертификатом служит делитель, проверка решения — это просто деление исходного числа на сертификат и проверка равенства остатка нулю.

FACTORING Даны целые числа N, L и U. Требуется проверить, имеется ли у числа N простой делитель p в промежутке от L до U. Сертификатом является простой делитель p.

LP В задаче *линейного программирования* необходимо выяснить, имеется ли рациональное решение у системы m линейных неравенств с рациональными коэффициентами и n неизвестными:

$$\begin{cases} a_{11}u_1 + a_{12}u_2 + \cdots + a_{1n}u_n \leqslant b_1, \\ \dots \\ a_{m1}u_1 + a_{m2}u_2 + \cdots + a_{mn}u_n \leqslant b_m, \end{cases}$$

где $a_{ij}, b_i (\in \mathbb{Q})$ — некоторые константы, а $u_j (\in \mathbb{Q})$ — неизвестные ($1 \leqslant i \leqslant m$, $1 \leqslant j \leqslant n$). Сертификатом может служить подстановка, задающая неизвестным значения, удовлетворяющие системе.

0/1-IP Задача *0/1-целочисленного программирования* совпадает по постановке с задачей линейного программирования, но решения требуется искать во множестве векторов $\{0,1\}^m$. Сертификатом снова служит удовлетворяющая системе подстановка, назначающая каждой из переменных $u_1, u_2, \dots, u_n$ значение из множества $\{0,1\}$.

SAT Дана булева формула, содержащая набор переменных и операции отрицания, конъюнкции и дизъюнкции. Требуется проверить, является ли она *выполнимой*, то есть существует ли подстановка, после применения которой формула превращается в истинное утверждение. Сертификатом является подстановка, то есть набор значений переменных. Ясно, что подставить значения переменных и вычислить значение формулы можно за полиномиальное время.

Статус рассмотренных десяти задач из класса **NP** совершенно различен. Так, известно, что задачи GCONN, COMPOSITE и LP принадлежат классу **P**. Связность графа может быть проверена с помощью поиска в ширину с полиномиальной сложностью, проверка на простоту может быть выполнена за полиномиальное время алгоритмом Агравала—Каяла—Саксены (2004 год), а задача линейного программирования решается полиномиальным алгоритмом Хачияна.

Для всех остальных задач неизвестно полиномиальных алгоритмов, однако нет и доказательства, что они не могут быть решены за полиномиальное время. Относительно пяти задач (INDSET, TSP, SUBSETSUM, 0/1-IP и SAT) известно, что они являются в некотором смысле самыми трудными задачами в классе **NP**. Если будет найден алгоритм, решающий хотя бы одну из них за полиномиальное время, то также за полиномиальное время можно будет решить вообще любую задачу из класса **NP**. Такие задачи называются **NP**-*полными*.

Статус ещё двух задач (GI и FACTORING) пока[1] не выяснен: с одной стороны, полиномиальный алгоритм для них неизвестен, а с другой стороны неизвестно также, являются ли они **NP**-полными.

[1] По информации на 7 мая 2019 года.

4.1.3. Соотношение между классами P, NP и EXP

Имеет место следующее простое утверждение, описывающее соотношение между введёнными ранее классами **P**, **NP** и **EXP**:

Теорема. $\mathbf{P} \subset \mathbf{NP} \subset \mathbf{EXP}$.

Доказательство. Докажем первое вложение $\mathbf{P} \subset \mathbf{NP}$. Пусть язык $L \in \mathbf{P}$, тогда существует машина Тьюринга M, распознающая язык L за полиномиальное время. Возьмём в качестве сертификата u пустое слово ε (его длина задаётся нулевым многочленом $p(n) = 0$), а машину M будем использовать в качестве проверяющей машины: $x \in L \Leftrightarrow M(x, u) = 1$ (машина M допускает слово x и без сертификата, поэтому пустое слово вместо сертификата помешать ей не может). Следовательно, $L \in \mathbf{NP}$.

Для доказательства второго вложения ($\mathbf{NP} \subset \mathbf{EXP}$) достаточно заметить, что вычислительная мощность класса **EXP** позволяет перебрать все возможные сертификаты полиномиальной длины (их количество составляет $2^{p(n)}$), запуская для каждого из них существующую для любой задачи из класса **NP** работающую за полиномиальное время проверяющую машину Тьюринга. Входное слово будет допускаться, если найден сертификат, удостоверяющий его членство в рассматриваемом языке. □

Известно, что $\mathbf{P} \neq \mathbf{EXP}$, однако никакие другие соотношения между этими тремя классами в настоящее время неизвестны. Вопрос о равенстве или неравенстве классов **P** и **NP** является важнейшим *открытым* вопросом теории сложности, он также важен для других областей математики и науки вообще. Большинство специалистов считает, что $\mathbf{P} \neq \mathbf{NP}$, однако пока этот факт остаётся недоказанным.

4.1.4. Альтернативный подход к определению класса NP

Мы рассматриваем класс **NP** как класс задач, решение которых может быть проверено за полиномиальное время с помощью сертификата, однако исторически он был введён в рассмотрение другим путём. Его название означает *nondeterministic polynomial* и соответствует классу задач, которые могут быть решены за полиномиальное время недетерминированной машиной Тьюринга.

Пусть $T : \mathbb{N} \longrightarrow \mathbb{N}$ — некоторая функция. Говорят, что язык L принадлежит классу $\mathbf{NTIME}(T(n))$, если существует распознающая язык L недетерминированная машина Тьюринга, которая на входном слове длины n делает не более $cT(n)$ шагов, где $c(> 0)$ — некоторая константа.

Такой подход к определению класса **NP** эквивалентен принятому ранее, что доказывается следующей теоремой.

Теорема. $\mathbf{NP} = \bigcup\limits_{c>=1} \mathbf{NTIME}(n^c)$.

Доказательство. Идея доказательства этой теоремы заключается в двухэтапном рассмотрении работы недетерминированной машины Тьюринга: на первом этапе она на протяжении нескольких шагов делает недетерминированный выбор, что можно рассматривать как формирование сертификата, а на втором — занимается проверкой полученного сертификата, более к недетерминированному выбору не прибегая. Второй этап соответствует работе проверяющей машины Тьюринга из определения

класса **NP**. Поскольку общее время работы недетерминированной машины полиномиально, то таково же время работы каждого из этапов, а значит и длина сертификата, и время работы проверяющей машины оказываются полиномиальными. Проведём это доказательство более формально.

Для доказательства вложения $\mathbf{NP} \subset \bigcup\limits_{c>0} \mathbf{NTIME}(n^c)$ возьмём язык $L \in \mathbf{NP}$ и построим распознающую его за полиномиальное время недетерминированную машину Тьюринга. Пусть на вход ей подаётся слово x длины n. Предположим, что длина сертификата ограничена многочленом $p(n)$, а проверяющая машина работает за время $q(n)$. На первом этапе недетерминированная машина будет строить все возможные сертификаты, недетерминированно печатая, к примеру, на каждом шаге 0 или 1 (продолжительность этого этапа — $p(n)$), а на втором — запускать проверяющую машину (продолжительность этого этапа — $q(n)$). В случае допускания входного слова и сертификата проверяющей машиной строящаяся недетерминированная машина будет переходить в допускающее состояние. Ясно, что это может произойти тогда и только тогда, когда входное слово принадлежит языку L (найден сертификат, удостоверяющий этот факт). Поскольку существует константа c, такая что $p(n) + q(n) = O(n^c)$, то $L \in \mathbf{NTIME}(n^c)$ и, следовательно, $L \in \bigcup\limits_{c>0} \mathbf{NTIME}(n^c)$.

Докажем обратное вложение. Предположим, что существует недетерминированная машина Тьюринга, распознающая язык L за полиномиальное время. Необходимо предъявить проверяющую машину и сертификат, зависящий, разумеется, от входного слова. Запустим недетерминированную машину на слове x. Во время работы она сделает не более чем полиномиальное количество недетерминированных выборов, которые можно закодировать с помощью слова, состоящего из символов 0 и 1, длина которого также будет не более чем полиномиальной от длины входного слова. Нас интересует та последовательность выборов, на которой машина приходит в итоге в допускающее состояние (она существует, если $x \in L$), эту последовательность будем считать сертификатом. Проверяющей машине теперь достаточно просимулировать только одну ветвь работы недетерминированной машины, задаваемую сертификатом, проверив, что в конце концов достигается допускающее состояние. На это ей потребуется то же количество шагов, то есть полиномиальное по длине входного слова. Существование проверяющей машины и сертификата доказывает, что $L \in \mathbf{NP}$. □

В дальнейшем мы будем попеременно пользоваться то одним подходом к определению класса **NP**, то другим, в зависимости от того, что будет удобнее.

4.1.5. Что если P=NP?

Стоит иметь в виду те последствия, которые могут иметь место в случае, если окажется, что $\mathbf{P} = \mathbf{NP}$. Коротко их перечислим:

1. Целый ряд задач, связанных с поиском в экспоненциальном пространстве, получит эффективное полиномиальное решение. К таким задачам относятся многие задачи комбинаторной оптимизации, в том числе проектирование цифровых схем с оптимальным энергопотреблением, задачи составления расписаний, искусственного интеллекта, проверки корректности программного обеспечения, доказательства математических теорем и пр.

2. Можно будет доказать, что полиномиальные рандомизированные алгоритмы, использующие во время работы датчики случайных чисел, могут быть выполнены за полиномиальное же время без использования случайных чисел. То есть случайность оказывается ненужной.

3. Многие широко используемые в настоящее время криптографические алгоритмы и технологии (RSA, SSL, PGP, цифровые подписи и пр.) перестанут работать, так как любая зашифрованная с их помощью информация может быть эффективно расшифрована без знания секретных ключей.

Все эти последствия представляются сейчас крайне маловероятными, однако история науки знает немало примеров, когда совершенно невероятные вещи оказывались истинными. Чего стоят, к примеру, обнаружение факта вращения Земли вокруг Солнца и открытие геометрии Лобачевского или теории относительности Эйнштейна.

4.2. Сводимость по Карпу и понятие NP-полноты

Аппарат сведений, использовавшийся нами в теории вычислимости, работает также и в теории сложности, однако здесь добавляется необходимое условие: сведение (то есть преобразование экземпляра одной задачи к экземпляру другой с тем же ответом) должно выполняться за полиномиальное время. Такие сведения называются полиномиальными. Дадим соответствующее этому понятию формальное определение.

Говорят, что язык $L \subset \{0,1\}^*$ *полиномиально сводится по Карпу* к языку $L' \subset \{0,1\}^*$ (обозначается $L \leqslant_p L'$), если существует функция $f : \{0,1\}^* \longrightarrow \{0,1\}^*$, вычисляемая за полиномиальное время и такая, что $x \in L \Leftrightarrow f(x) \in L'$.

Сводимость по Карпу это только один из нескольких используемых в теории сложности видов сводимостей. С её помощью можно дать формальное определение **NP**-*трудных* (**NP**-*hard*) и **NP**-*полных* (**NP**-*complete*) задач.

Язык L называется **NP**-трудным, если для любого языка $L' \in \mathbf{NP}$ имеет место сводимость $L' \leqslant_p L$. Если сам L при этом принадлежит классу **NP**, то он называется **NP**-полным.

Сводимость, как и прежде, позволяет решать одни задачи, имея алгоритмы решения других. Например, если $L \leqslant_p L'$ и M' — машина, распознающая язык L', то для проверки принадлежности входного слова x языку L достаточно вычислить результат сведения $f(x)$, а затем запустить на нём машину M'. Следует обратить внимание на то, что если M' работает полиномиальное время, то полученное решение для x также требует только полиномиального времени (поскольку время сведения по определению полиномиально). Это наблюдение позволяет понять, почему языки, к которым сводятся любые языки из **NP**, называются **NP**-трудными: они как минимум столь же трудны, что и все языки из **NP**, возможность распознать их за полиномиальное время означает, что за полиномиальное время можно распознать любой язык из класса **NP**.

Теорема. *Имеют место следующие свойства отношения сводимости по Карпу:*

1. *(Транзитивность) Если $L \leqslant_p L'$ и $L' \leqslant_p L''$, то $L \leqslant_p L''$.*

2. *Если* L — **NP**-*полный,* $L \leqslant_p L'$ *и* $L' \in \mathbf{NP}$*, то* L' — *также* **NP**-*полный.*

3. *Если* L — **NP**-*трудный и* $L \in \mathbf{P}$*, то* $\mathbf{P} = \mathbf{NP}$.

4. *Если* L — **NP**-*полный, то* $L \in \mathbf{P}$ *тогда и только тогда, когда* $\mathbf{P} = \mathbf{NP}$.

Доказательство. Для доказательства этих утверждений достаточно воспользоваться наблюдением, обсуждавшимся перед этой теоремой, и тем фактом, что композиция многочленов является многочленом, хотя и большей степени. □

Итак, для доказательства равенства классов **P** и **NP** достаточно предъявить полиномиальный алгоритм, решающий некоторую **NP**-полную задачу. Наши дальнейшие усилия будут направлены на выявление наиболее важных из таких задач. Как только будет установлено существование одной **NP**-полной задачи, **NP**-полноту всех остальных можно будет доказывать с помощью второго свойства, сводя к ним задачи, **NP**-полнота которых уже установлена.

4.3. Теорема Кука—Левина

Теорема (Кука—Левина)**.** *Задача* SAT — **NP**-*полна.*

Напомним, что задачей SAT называется задача проверки выполнимости произвольной булевой формулы. Введём в рассмотрение её частный случай, а именно задачу CSAT, состоящую в проверке выполнимости булевой формулы, находящейся в конъюнктивной нормальной форме (КНФ). Поскольку ни для одной задачи факт **NP**-полноты пока не установлен, в первый раз для доказательства **NP**-полноты нам придётся пользоваться определением, то есть доказывать, что к некоторой задаче можно свести *любой* язык из $\{0,1\}^*$. В качестве такой задачи мы будем использовать задачу CSAT, но прежде сформулируем известное из курса дискретной математики полезное утверждение относительно булевых функций и формул в КНФ.

Утверждение. *Пусть имеется булева функция* $f : \{0,1\}^l \longrightarrow \{0,1\}$*. Тогда существует находящаяся в КНФ булева формула* ϕ *с* l *переменными, такая что* $\phi(u) = f(u)$ *для всех* $u \in \{0,1\}^l$*, причём её длина не превосходит* $l2^l$ *(длина формулы считается по количеству операций* $\vee$ *и* $\wedge$*).*

Для построения соответствующей формулы достаточно взять таблицу значений функции (в ней 2^l строк), построить по её нулям дизъюнкты (длиной $l-1$) и соединить их конъюнкциями (ещё до 2^l операций).

Лемма. *Задача* CSAT — **NP**-*полна.*

Доказательство. Необходимо доказать, что любой язык из класса **NP** сводится полиномиально по Карпу к задаче CSAT, то есть построить соответствующее сведение.

Пусть $L \in \mathbf{NP}$. Это означает, что существуют такие многочлен p и работающая полиномиальное время $T(n)$ машина M, что для всех $x \in \{0,1\}^*$ длины n

$$x \in L \Leftrightarrow \text{существует } u \in \{0,1\}^{p(n)}, \text{такое что } M(x,u) = 1.$$

Покажем, что существует полиномиальное преобразование, которое ставит в соответствие этому x формулу в КНФ ϕ_x, такую что

$$x \in L \Leftrightarrow \phi_x \in \text{CSAT } (\phi_x\text{— выполнима}).$$

Рассмотрим сначала тривиальное, но неправильное сведение. Можно было бы попытаться построить по x функцию

$$f_x : u \in \{0,1\}^{p(n)} \mapsto M(x,u),$$

действующую из $\{0,1\}^{p(n)}$ в $\{0,1\}$. Это отображение позволило бы вычислять результат работы проверяющей машины, соответствующей языку L, на фиксированном x. Если сертификат u существует, что имеет место при $x \in L$, то функция на нём обращается в единицу, в противном же случае она всегда возвращает 0. Согласно сформулированному выше утверждению по такой функции можно было бы далее построить формулу ϕ_x, находящуюся в КНФ и принимающую одинаковое с функцией значение. К сожалению, размер такой формулы может достигать $p(n)2^{p(n)}$, а значит, даже выписать её за полиномиальное время не удастся.

Проблема предложенного подхода состоит в том, что мы пытаемся описать работу машины одной функцией, которая должна за один шаг отобразить произвольный элемент экспоненциального пространства сертификатов на результат работы машины. Но ведь машина Тьюринга работает иначе, она делает всего лишь полиномиальное число шагов и, что самое главное, на каждом шаге работает с очень небольшим объёмом информации — основываясь на текущем состоянии и символе на ленте, она вычисляет новое состояние, заменяет символ и перемещает считывающую головку на одну позицию. Говорят, что вычисление на машине Тьюринга *локально*, то есть действие каждого шага локализовано в пределах состояния и нескольких ленточных символов. Именно локальность вычислений позволяет построить требуемую формулу за полиномиальное время.

Будем далее предполагать, что проверяющая машина M, соответствующая зафиксированному ранее языку L, удовлетворяет следующим ограничениям:

- она имеет неизменяемую полуленту для записи входных слов с ограничителем ▶ в её начале;
- у неё также имеется рабочая полулента с тем же ограничителем;
- машина является забывающей (то есть перемещение считывающей головки не зависит от содержание входного слова, а только от его длины).

Ранее мы доказывали, что эти ограничения не являются существенными в том смысле, что если машина M им не удовлетворяет, то можно построить полностью эквивалентную ей машину, которая будет им удовлетворять. Однако ими удобно пользоваться при построении требуемой формулы.

Работа машины на каждом шаге определяется так называемым *снэпшотом* — тройкой $[a,b,q] \in \Gamma \times \Gamma \times Q$ (Γ — это ленточный алфавит, Q — множество состояний, a и b — символы на каждой из лент, q — текущее состояние). Снэпшот можно закодировать двоичным словом длины c, где c зависит от размеров множеств Γ и Q, но не зависит от длины входного слова. Можно считать, что один шаг вычисления — это переход от одного снэпшота к другому, а вычисление в целом — это

последовательность снэпшотов, от первого до последнего. Если у нас есть последовательность снэпшотов, то мы можем убедиться, что она действительно описывает работу некоторой машины.

Зададимся вопросом: от чего зависит снэпшот на i-м шаге (обозначим его символом z_i)? Очевидный ответ — от всех предыдущих снэпшотов. Однако на самом деле зависимость проще. Чтобы проверить корректность снэпшота на шаге i, следует посмотреть на предыдущий снэпшот z_{i-1} (из него нас интересует только состояние), на символ входного слова, читаемый на i-м шаге (напомним, что благодаря тому, что машина является забывающей, позиция этого символа может быть вычислена заранее, а входная лента неизменяема), и на символ рабочей ленты, который может быть извлечён из снэпшота того шага, на котором машина находилась в той же позиции в последний раз (ведь если машина находилась в других позициях, то этот символ гарантированно не поменялся — здесь сказывается локальность вычислений).

Обозначим входное слово через y (это пара x и u). Теперь снэпшот z_i зависит от трёх значений — z_{i-1}, $y_{\mathrm{in}(i)}$ и $z_{\mathrm{prev}(i)}$, общая длина которых есть $2c+1$ (два снэпшота и символ входного слова). Функции in и prev могут быть вычислены заранее, они не зависят от входного слова (так как машина M забывающая), для этого достаточно предварительно запустить машину M на тривиальном слове $0^n 0^{p(n)}$ (n — длина входного слова, $p(n)$ — длина сертификата). В силу детерминированности машины Тьюринга, z_i определяется этими значениями однозначно, поэтому по функции переходов машины M можно построить функцию

$$F : \{0,1\}^{2c+1} \longrightarrow \{0,1\}^c,$$

которая вычисляет $z_i = F(z_{i-1}, y_{\mathrm{in}(i)}, z_{\mathrm{prev}(i)})$.

Теперь можно обратиться непосредственно к сведению, то есть к построению формулы в КНФ по заданному x. Вспомним, что

$$x \in L \Leftrightarrow \text{существует } u \in \{0,1\}^{p(n)}, \text{такое что } M(x,u) = 1.$$

Последнее имеет место тогда и только тогда, когда существуют $y \in \{0,1\}^{n+p(n)}$ и последовательность снэпшотов $z_1, \ldots, z_{T(n)} \in \{0,1\}^c$, такие что выполняются следующие четыре условия:

(1) первые n битов y совпадают с битами x;

(2) z_1 совпадает с двоичным кодом начального снэпшота $[\blacktriangleright, B, q_0]$;

(3) для каждого $i \in \{1, \ldots, T(n)\}$ снэпшот $z_i = F(z_{i-1}, y_{\mathrm{in}(i)}, z_{\mathrm{prev}(i)})$;

(4) $z_{T(n)}$ кодирует снэпшот с допускающим состоянием (машина возвращает 1).

Наша формула будет содержать переменные $y \in \{0,1\}^{n+p(n)}$ и $z \in \{0,1\}^{cT(n)}$ (здесь собраны все снэпшоты) и состоять из конъюнкции формул, соответствующих каждому из условий в отдельности:

$$\phi_x = (1) \wedge (2) \wedge (3) \wedge (4).$$

Если построить формулы полиномиальной в КНФ для условий (1)–(4), то ϕ_x окажется в КНФ, при этом $x \in L$ экивалентно истинности всех четырёх условий, то есть выполнимости ϕ_x.

Условие (1) реализуется с помощью побитовой эквивалентности

$$y_1 \sim x_1 \equiv (y_1 \longrightarrow x_1) \wedge (x_1 \longrightarrow y_1) \equiv (y_1 \vee \bar{x}_1) \wedge (x_1 \vee \bar{y}_1),$$

применённой n раз для каждого из n битов x, поэтому суммарная длина формулы оказывается равной $4n$ (по 3 операции в каждой экивалентности и ещё одна на их соединение). Условия (2) и (4) описываются 0/1-значными функциями, действующими на $\{0,1\}^c$ (условие на наборе переменных либо выполняется, либо нет, отсюда 0/1-значная функция), поэтому их можно представить формулами размером до $c2^c$, то есть константой, не зависящей от размера входного слова. Наконец, условие (3) есть конъюнкция $T(n)$ условий, соответствующих каждому из переходов. За один переход отвечают $3c+1$ переменная (три снэпшота и один бит входного слова), поэтому формула, проверяющая его корректность, будет иметь размер до $(3c+1)2^{3c+1}$. Общая длина формулы для условия (3) будет ограничиваться значением $T(n)(3c+1)2^{3c+1}$. Таким образом, длина построенной формулы ϕ_x ограничена $O(n+T(n))$, то есть полиномиальна относительно длины x,так что формула может быть построена за полиномиальное время по заданному x при фиксированном языке L (и, соответственно, машине M). □

Вернёмся теперь к доказательству теоремы Кука—Левина. Для этого (по второму свойству сводимости по Карпу) достаточно построить тривиальное сведение задачи CSAT к SAT: поскольку формула в КНФ является частным случаем булевой формулы произвольного вида, то в сведении ничего делать не требуется, к тому же формула одновременно выполнима сама с собой. Такое сведение полностью соответствуют определению и свидетельствует о том, что сведение вовсе не обязано быть сюръективным: нам достаточно научиться преобразовывать *все* экземпляры исходной задачи в некоторое подмножество экземпляров задачи, **NP**-полноту которой мы хотим доказать. Этим соображением мы будем неоднократно пользоваться в дальнейшем.

В книге Хопкрофта, Мотвани и Ульмана (раздел 10.2.3) приводится непосредственное доказательство теоремы Кука—Левина, не сводящееся к NP-полноте задачи CSAT. Там же (разделы 10.3.2–3) обосновывается сведение задачи SAT к задаче CSAT, необходимое для доказательства NP-полноты последней. Алгоритм этого сведения рассказывался на лекции.

4.4. Важные примеры NP-полных задач

4.4.1. Задача 3SAT

Говорят, что булева формула находится в 3-КНФ, если она находится в КНФ, причём в каждом дизъюнкте имеется в точности три различных литерала (переменная или её отрицание). Задачей 3SAT называется задача проверки выполнимости булевой формулы, находящейся в 3-КНФ.

Теорема. *Задача 3SAT* **NP***-полна.*

Доказательство. Для доказательства **NP**-полноты задачи 3SAT построим сведение к ней задачи CSAT, **NP**-полнота которой уже установлена.

Пусть ϕ — булева формула, находящаяся в КНФ. Необходимо построить находящуюся в 3-КНФ формулу ψ, выполнимую одновременно с ϕ, то есть $\phi \in \text{SAT} \Leftrightarrow \psi \in \text{3SAT}$.

Будем преобразовывать каждый дизъюнкт исходной формулы ϕ различными способами, в зависимости от его размера:

1) Предположим, что дизъюнкт содержит один литерал, скажем, для определённости, x. Тогда заменим его на следующую формулу в 3-КНФ с двумя добавленными переменными u и v:

$$(x \vee u \vee v) \wedge (x \vee \bar{u} \vee v) \wedge (x \vee u \vee \bar{v}) \wedge (x \vee \bar{u} \vee \bar{v}).$$

Ясно, что если дизъюнкт x выполним (то есть $x = 1$), то выполнимыми будут и все четыре построенных дизъюнкта, если же x невыполним ($x = 0$), то какими бы ни были значения добавленных переменных u и v, один из построенных дизъюнктов обязательно будет оказываться ложным, а значит, формула в целом будет невыполнимой.

2) Предположим, что дизъюнкт содержит два литерала, например, $\bar{x} \vee y$. В этом случае достаточно добавить только одну переменную и заменить дизъюнкт на два новых:

$$(\bar{x} \vee y \vee u) \wedge (\bar{x} \vee y \vee \bar{u}).$$

Выполнимость исходного дизъюнкта делает выполнимыми оба построенных, тогда как невыполнимость приводит к тому, что один из построенных дизъюнктов при любой подстановке будет оказываться ложным (если $u = 0$, то первый, иначе — второй), то есть построенная формула оказывается также невыполнимой.

3) В случае дизъюнкта, содержащего три литерала, преобразование не требуется.

4) Предположим, что дизъюнкт содержит четыре литерала, например, $\bar{x} \vee y \vee \bar{z} \vee u$. Введём новую переменную w и разобъём дизъюнкт на два:

$$(\bar{x} \vee y \vee w) \wedge (\bar{w} \vee \bar{z} \vee u).$$

Если исходный дизъюнкт является истинным, то это происходит за счёт как минимум одной из его переменных, та же переменная будет делать истинным один из двух построенных дизъюнктов. Второй же из построенных дизъюнктов также можно сделать истинным за счёт добавленной переменной w (истинное её значение делает истинным первый дизъюнкт, а ложное — второй). Если же исходный дизъюнкт невыполним, то независимо от значений w один из построенных дизъюнктов обязательно будет оказываться ложным. Таким образом, исходный дизъюнкт выполним одновременно с выполнимостью построеннной формулы из двух дизъюнктов.

5) Рассмотрим, наконец, общий случай дизъюнкта $C_k = x_1 \vee \cdots \vee x_k$, содержащего k литералов ($k > 4$). Его можно разбить аналогично предыдущему случаю на два дизъюнкта, C_{k-1} с $k-1$ литералом и дизъюнкт с тремя литералами, добавив новую переменную:

$$(x_1 \vee \cdots \vee x_{k-2} \vee w) \wedge (\bar{w} \vee x_{k-1} \vee x_k).$$

Если C_k выполним, то его истинность достигается за счёт значения как минимум одной из переменных $x_1, \ldots, x_k$, она же делает истинным один из построенных дизъюнктов, второй делается истинным за счёт переменной w. Если C_k невыполним, то переменная w в одиночку не может удовлетворить обоим построенным дизъюнктам, поэтому построенная формула также оказывается невыполнимой. Процедуру отделения дизъюнкта с тремя литералами нужно теперь повторять до тех пор, пока не будет получена формула в 3-КНФ.

Следует заметить, что все случаи, кроме последнего, приводят к увеличению длины формулы на различные постоянные значения. В последнем случае формула увеличивается линейно по числу входящих в неё литералов. Поэтому такое преобразование формулы в целом может быть выполнено за полиномиальное время. Ясно также, что выполнимость и невыполнмость исходной формулы в целом после замены каждого её дизъюнкта на формулы в 3-КНФ сохраняются.

Существование описанного сведения доказывает **NP**-полноту задачи 3SAT.

□

4.4.2. Задача 0/1-IP

Напомним, что в задаче 0/1-IP требуется выяснить, имеется ли решение у системы m линейных неравенств с целыми коэффициентами и n неизвестными:

$$\begin{cases} a_{11}u_1 + a_{12}u_2 + \cdots + a_{1n}u_n \leqslant b_1, \\ \ldots \\ a_{m1}u_1 + a_{m2}u_2 + \cdots + a_{mn}u_n \leqslant b_m, \end{cases}$$

где $a_{ij}, b_i (\in \mathbb{Z})$ — некоторые константы, а $u_j (\in \{0,1\})$ — неизвестные ($1 \leqslant i \leqslant m$, $1 \leqslant j \leqslant n$).

Теорема. *Задача 0/1-IP* **NP***-полна.*

Доказательство. Докажем эту теорему сведением **NP**-полной задачи CSAT к 0/1-IP. Для этого по заданной булевой формуле в КНФ необходимо построить систему линейных неравенств, имеющую решение одновременно с выполнимостью исходной формулы. Будем преобразовывать каждый дизъюнкт в одно линейное неравенство.

Будем заменять положительный литерал u на точно такую же переменную u, а отрицательный литерал $\bar{u}$ — на выражение $1 - u$. Продемонстрируем это преобразование на примере, построив для дизъюнкта $\bar{u}_1 \vee \bar{u}_3 \vee u_{15}$ неравенство:

$$(1 - u_1) + (1 - u_3) + u_{15} \geqslant 1,$$

эквивалентное

$$u_1 + u_3 - u_{15} \leqslant 1.$$

Нетрудно видеть, что наличие у дизъюнкта удовлетворяющей ему подстановки соответствует наличию решения у построенного линейного неравенства, и наоборот. Формула в целом теперь преобразуется к системе, имеющей решение одновременно с выполнимостью формулы. Ясно, что для этого преобразования достаточно полиномиального времени. Существование такого сведения доказывает **NP**-полноту рассматриваемой задачи. □

4.4.3. Задача о независимом множестве

Напомним, что *независимым* называется такое множество вершин графа, никакие две из которых не связаны между собой дугами. Постановка задачи INDSET звучит так: необходимо проверить, существует ли в заданном графе G независимое множество размера m.

Теорема. *Задача INDSET* **NP***-полна.*

Доказательство. Опишем сведение задачи 3SAT, **NP**-полнота которой была установлена ранее, к задаче INDSET. Для этого по заданной формуле, находящейся в 3-КНФ и имеющей m дизъюнктов, построим граф G, имеющий независимое множество размера m одновременно с выполнимостью исходной формулы:

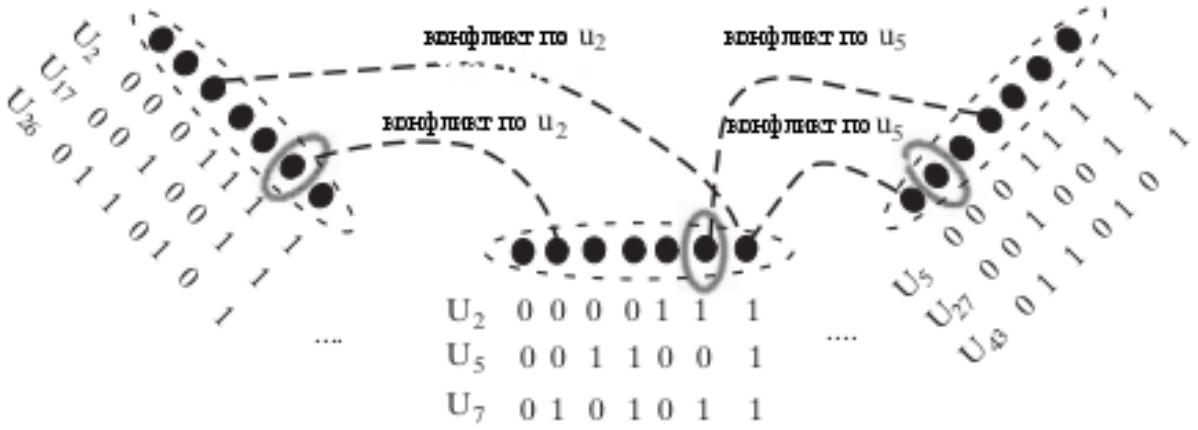
1) Для каждого дизъюнкта (с тремя литералами) исходной формулы добавим в граф по полносвязному кластеру с семью вершинами в каждом (полносвязность кластера означает, что каждая вершина кластера связана дугами со всеми вершинами того же кластера). Вершины одного кластера будут соответствовать различным подстановкам (содержащим по три переменные), *удовлетворяющим* дизъюнкту (проверьте, что таких подстановок всегда семь).

2) Свяжем дугами такие вершины различных кластеров, которые соответствуют *несогласованным* подстановкам, то есть таким, в которых одной переменной назначены разные значения.

В результате для формулы с m дизъюнктами мы получили граф G, содержащий $7m$ вершин. Пример построения для формулы

$$(u_2 \vee \bar{u}_{17} \vee u_{26}) \wedge \cdots \wedge (\bar{u}_2 \vee \bar{u}_5 \vee u_7) \wedge \cdots \wedge (u_5 \vee \bar{u}_{27} \vee u_{43})$$

приведён на рисунке:



Фрагмент независимого множества выделен серым.

Ясно, что такое построение может быть выполнено за полиномиальное время. Проверим теперь, что выполнимость формулы будет иметь место тогда и только тогда, когда в графе имеется независимое множество размера m.

Если формула выполнима, то существует удовлетворяющая ей подстановка. Рассмотрим её *сужения* на множества переменных, содержащихся в каждом из дизъюнктов. Такие сужения будут соответствовать некоторым удовлетворяющим дизъюнктам подстановкам, которым, в свою очередь, соответствуют некоторые вершины кластеров. Выберем в каждом кластере по одной такой вершине, всего m вершин — полученное множество будет независимым, поскольку различные сужения одной подстановки являются согласованными, а значит, дуг между выбранными вершинами быть не может.

Наоборот, если в графе имеется независимое множество размера m, то, во-первых, его вершины обязательно расположены в различных кластерах (поскольку кластеры полносвязны), и, во-вторых, соответствуют согласованным подстановкам (так как между ними нет дуг). Из таких согласованных подстановок (с тремя переменными), каждая из которых удовлетворяет одному дизъюнкту формулы, можно собрать подстановку, удовлетворяющую формуле в целом, то есть формула оказывается выполнимой.

□

4.4.4. Задача коммивояжёра

Задача коммивояжёра (TSP) в нашем случае формулируется следующем образом: необходимо проверить, имеется ли во взвешенном графе G цикл веса k, проходящий через все вершины графа.

Доказывать **NP**-полноту задачи коммивояжёра мы будем через последовательность сведений, в которых участвуют следующие вспомогательные задачи:

1) dHAMPATH — задача о существовании гамильтонова пути (пути, проходящего по одному разу через все вершины графа) в ориентированном графе;
2) HAMPATH — задача о существовании гамильтонова пути в неориентированном графе;
3) HAMCYCLE — задача о существовании гамильтонова цикла (путь, начинающийся и заканчивающийся в одной и той же точке) в неориентированном графе.

При этом цепочка сведений будет следующей:

$$\text{CSAT} \leqslant_p \text{dHAMPATH} \leqslant_p \text{HAMPATH} \leqslant_p \text{HAMCYCLE} \leqslant_p \text{TSP}.$$

Гамильтонов путь в ориентированном графе

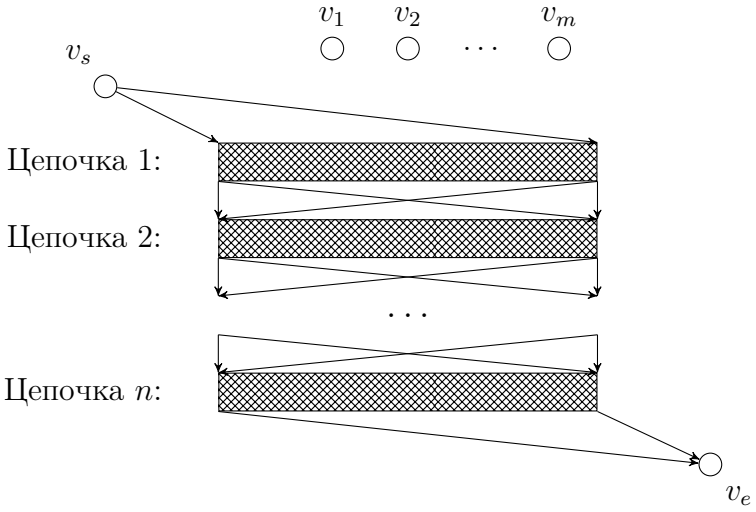
Теорема. *Задача* dHAMPATH **NP***-полна.*

Доказательство. Нетрудно видеть, что задача dHAMPATH принадлежит классу **NP**, поскольку сертификатом может служить список вершин, составляющих гамильтонов путь. Для доказательства её **NP**-полноты покажем, что существует сведение задачи CSAT к ней, то есть по любой формуле ϕ в КНФ можно построить ориентированный граф G такой, что гамильтонов путь в нём существует одновременно с выполнимостью формулы ϕ.

Предположим, что формула ϕ содержит m дизъюнктов $c_1, \ldots, c_m$ и n переменных $x_1, \ldots, x_n$.

Каждой переменной x_i поставим в соответствие «цепочку», содержащую $4m$ вершин, связанных следующим образом:

Всего получится n таких цепочек. Обозначим вершины, составляющие цепочки, символами $v_j^{(i)}$, где i — номер цепочки, $1 \leqslant i \leqslant n$, а j — номер вершины в цепочке при обходе слева направо, $1 \leqslant j \leqslant 4m$. По построению эти цепочки допускают обход как слева направо, так и справа налево. Теперь добавим в граф по одной вершине v_i, соответствующей каждому дизъюнкту c_i, и ещё две вершины, «стартовую» v_s и «конечную» v_e, связав последние с цепочками, то есть их первыми и последними вершинами:

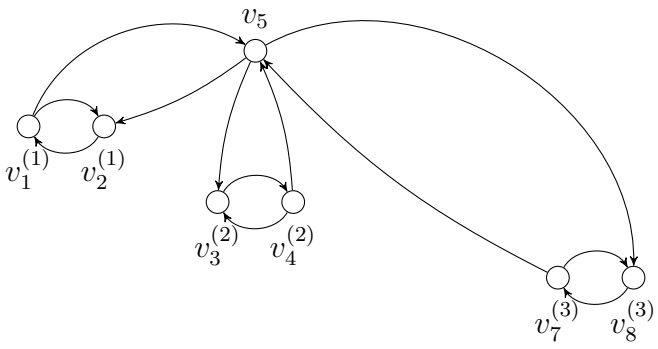


Таким образом, мы получили граф, содержащий $4mn+m+2$ вершины. Из вершины v_s выходят две дуги в вершины $v_1^{(1)}$ и $v_{4m}^{(1)}$, из каждой вершины $v_1^{(i)}$ и $v_{4m}^{(i)}$, $1 \leqslant i < n$, выходят дуги в каждую из $v_1^{(i+1)}$ и $v_{4m}^{(i+1)}$, из вершин $v_1^{(n)}$ и $v_{4m}^{(n)}$ выходят дуги в v_e. Для завершения сведения остаётся связать вершины v_i, соответствующие дизъюнктам, с вершинами цепочек. Сделаем это следующим образом:

- Если дизъюнкт c_i содержит литерал x_k, то выберем две соседние вершины k-й цепочки $v_j^{(k)}$ и $v_{j+1}^{(k)}$, добавим дугу из $v_j^{(k)}$ в v_i и из v_i в $v_{j+1}^{(k)}$ и организуем тем самым возможность «экскурсии» из цепочки в вершину дизъюнкта.

- Если дизъюнкт c_i содержит отрицание $\overline{x_k}$, то снова выберем две соседние вершины k-й цепочки $v_j^{(k)}$ и $v_{j+1}^{(k)}$, но дуги теперь будем вести в противоположном направлении, то есть из $v_{j+1}^{(k)}$ в v_i и из v_i в $v_j^{(k)}$.

При выборе вершин в цепочках мы не будем их переиспользовать, то есть если одна и та же переменная участвует в нескольких дизъюнктах, то вершины в соответствующей цепочке будут выбираться всякий раз новые. Поскольку всего их $4m$, то даже если некоторая переменная входит в каждый дизъюнкт сама по себе и с отрицанием, то вершин в соответствующей ей цепочке для организации «экскурсий» всё равно достаточно. Если, к примеру, пятый дизъюнкт $c_5 = x_1 \vee \overline{x_2} \vee x_3$, то вершина, ему соответствующая, может получить следующие связи:



Понятно, что построение графа по заданной формуле может быть выполнено за полиномиальное время.

Убедимся, что формула ϕ выполнима одновременно с наличием в построенном ориентированном графе G гамильтонова пути. Предположим, что формула ϕ выполнима, то есть имеет удовлетворяющую ей подстановку $x_1, \ldots, x_n$. Покажем, что существует путь, проходящий через все вершины построенного графа. Этот путь начинается в вершине v_s, затем обходит по порядку все цепочки и заканчивается в v_e. Если значение переменной x_j в удовлетворяющей подстановке равно 1, то соответствующая цепочка обходится слева направо, а если 0, то справа налево. Построенный таким образом путь обходит все вершины графа кроме вершин, соответствующих дизъюнктам, однако его нетрудно модифицировать так, чтобы он включал и их. Поскольку подстановка делает истинной каждый из дизъюнктов, и происходит это за счёт истинного либо ложного значения некоторой входящей в него переменной (достаточно одной), то для посещения соответствующей вершины имеется предусмотренная ранее «экскурсия», на которую и нужно заменить одну из связей построенного ранее обхода. Заметим, что если переменная входит с отрицанием, то соответствующая ей цепочка обходится справа налево и именно такое направление было выбрано для «экскурсии» в этом случае (второй пункт описания построения «экскурсий»). Всякая такая экскурсия возвращает нас в цепочку, поэтому после неё обход продолжается как обычно. Гамильтонов путь построен.

Предположим теперь, что в графе G имеется гамильтонов путь. Необходимо предъявить удовлетворяющую формуле ϕ подстановку. Заметим, что этот путь обязательно начинается в вершине v_s, поскольку у неё нет входящих дуг, и завершается в вершине v_e, поскольку из неё не выходит ни одна дуга. Теперь следует убедиться в том, что все цепочки обходятся одна за другой и внутри каждой цепочки строго сохраняется направление обхода либо слева направо, либо справа налево. Это было бы очевидным, если бы не возможность экскурсий в вершины, соответствующие дизъюнктам. Покажем, что если экскурсия выходит из некоторой вершины u, то она обязательно возвращает нас в соседнюю с ней вершину v. Действительно, если выход из экскурсии произошёл в другой цепочке, то при последующем посещении вершины v, которое всё равно должно произойти в силу гамильтоновости пути, обход остановится, так как все исходящие из неё дуги будут вести в посещённые ранее вершины (в u и вершину той же цепочки, из которой мы попали в v). Составим теперь удовлетворяющую формуле подстановку, беря значение переменной равным 1, если соответствующая цепочка обходится слева направо, и 0 в противном случае. Наличие в пути экскурсий во все вершины, соответствующие дизъюнктам, гарантирует, что каждый из дизъюнк-

тов будет удовлетворяться благодаря одной из переменных (например, если цепочка обходится слева направо и есть экскурсия, построенная по первому пункту, то значение переменной равно 1 и она входит в дизъюнкт без отрицания, поэтому дизъюнкт оказывается истинным; аналогично для случая отрицания). □

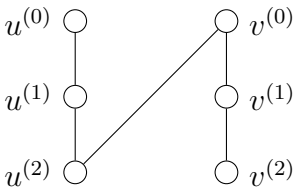
Гамильтонов путь в неориентированном графе

Теорема. *Задача* HAMPATH **NP**-*полна.*

Доказательство. Для доказательства этой теоремы достаточно построить сведение задачи dHAMPATH, **NP**-полнота которой уже установлена, к задаче HAMPATH.

Пусть имеется ориентированный граф G. Построим неориентированный граф G' следующим образом:

- Каждой вершине v графа G поставим в соответствие три вершины $v^{(0)}$, $v^{(1)}$ и $v^{(2)}$ в новом графе G'.
- Соединим рёбрами пары вершин $(v^{(0)}, v^{(1)})$ и $(v^{(1)}, v^{(2)})$.
- Если в исходном графе была дуга $u \longrightarrow v$, то добавим в новый граф ребро $(u^{(2)}, v^{(0)})$:



Таким образом, каждая вершина исходного графа заменяется на три вершины с рёбрами между ними, а каждая дуга исходного графа — на ребро между вершиной с индексом 2 и вершиной с индексом 0. Такое преобразование, очевидно, выполнимо за полиномиальное время.

Необходимо показать, что граф G имеет ориентированный путь тогда и только тогда, когда в графе G' есть неориентированный путь. Предположим, что в графе G имеется ориентированный путь $v_1, v_2, \ldots, v_n$. Тогда ясно, что в графе G' будет неориентированный путь $v_1^{(0)}, v_1^{(1)}, v_1^{(2)}, v_2^{(0)}, v_2^{(1)}, v_2^{(2)}, v_3^{(0)}, \ldots, v_{n-1}^{(2)}, v_n^{(0)}, v_n^{(1)}, v_n^{(2)}$.

Наоборот, любой неориентированный путь в графе G' по построению обязательно проходит через вершины с верхними индексами $0, 1, 2, 0, 1, 2, \ldots, 0, 1, 2$ или, в обратном направлении, $2, 1, 0, 2, 1, 0, \ldots, 2, 1, 0$. Причина этого в том, что вершина с индексом 1 должна входить в любой гамильтонов путь рядом с вершинами, проиндексированными 0 и 2, поскольку ни с какими другими вершинами графа она рёбрами не связана. Понятно также, что из пути с индексами вершин $0, 1, 2, 0, 1, 2, \ldots, 0, 1, 2$ можно извлечь путь в исходном графе, взяв дуги, соответствующие переходам $2, 0$. Они будут иметь правильное направление и составлять гамильтонов путь в G. □

Гамильтонов цикл в неориентированном графе

Теорема. *Задача* HAMCYCLE **NP**-*полна.*

Доказательство. Для доказательства NP-полноты задачи о существовании гамильтонова цикла в неориентированном графе строится сведение к ней задачи о существовании гамильтонова пути к неориентированном графе. Идея сведения: к заданному неориентированному графу G добавляется одна новая вершина и дуги между ней и каждой из вершин графа G. Для этого, очевидно, достаточно полиномиального времени. Далее показывается, что гамильтонов путь в исходном графе легко превращается в гамильтонов цикл в построенном (начало и конец пути соединяются дугами с добавленной вершиной, что даёт цикл), тогда как цикл в последнем легко урезается до пути в исходном (цикл проходит через добавленную вершину, инцидентные ей дуги, составляющие цикл, отбрасываются, и получается путь в исходном графе). Таким образом, путь и цикл одновременно либо существуют, либо нет, то есть сведение является корректным. □

Задача коммивояжёра

Теорема. *Задача* TSP **NP**-*полна.*

Доказательство. Сведём задачу HAMCYCLE к TSP. Такое сведение элементарно: для этого в исходном графе G назначим всем дугам вес 1 и получим взвешенный граф G'. Теперь существование гамильтонова цикла в графе G, содержащего k дуг, это в точности существование цикла веса k, проходящего через все вершины, во взвешенном графе G'. □

Разделы 4.5–4.7 не читались в 2016/2017 учебном году.

4.5. Соотношение задач разрешения и задач поиска

В рамках теории сложности обычно рассматривают задачи разрешения, но не задачи поиска. Мы можем считать, что ответить на вопрос «да или нет» гораздо проще, чем, к примеру, найти подстановку, удовлятворяющую булевой формуле. Тем не менее, если задачу разрешения для задачи SAT удалось бы решить за полиномиальное время, то за полиномиальное же время можно было бы и найти соответствующую подстановку. Имеет место следующее утверждение:

Теорема. *Если* $\mathbf{P} = \mathbf{NP}$*, то существует полиномиальный алгоритм, позволяющий отыскать подстановку, удовлетворяющую заданной выполнимой булевой формуле.*

Соответствующий алгоритм последовательно фиксирует значения переменных, решая задачу SAT для получающихся в результате фиксации всё более простых формул. При этом в искомой подстановке оказывается одно из значений, при фиксации которого выполнимость формулы сохраняется (такое значение обязательно есть в силу выполнимости исходной формулы).

4.6. Дополнения языков из NP

Нетрудно заметить, что если некоторый язык L принадлежит классу **P**, то его дополнение $\overline{L}$ также принадлежит этому классу. Если M — работающая полиномиальное время машина Тьюринга, распознающая язык L, то для построения машины, распознающей его дополнение, достаточно заменить прежнее множество допускающих состояний на новое, с одним новым состоянием, и организовать переход в него из всех остановов в недопускаемых ранее состояниях. Ясно, что такая машина будет допускать $\overline{L}$ и делать это за полиномиальное время. Таким образом, класс **P** замкнут относительно операции дополнения.

4.6.1. Класс coNP и coNP-полнота

Введём теперь в рассмотрение новый класс **coNP**, который содержит языки из $\{0,1\}^*$, дополнения которых принадлежат **NP**:

$$\mathbf{coNP} = \{L : \overline{L} \in \mathbf{NP}\}.$$

В силу замкнутости **P** относительно дополнения и вложения $\mathbf{P} \subset \mathbf{NP}$ класс **P** также вложен в **coNP**, то есть классы **NP**и **coNP** имеют общую часть, содержащую **P**.

Простым примером задачи из класса **coNP** служит задача

$$\overline{\text{SAT}} = \{\phi \text{ — невыполнимая булева формула}\},$$

дополнение которой принадлежит **NP**, поэтому $\overline{\text{SAT}} \in \mathbf{coNP}$ по определению. Как мы помним, ключевым свойством для задач из класса **NP** являлось существование сертификата, способного подтвердить положительный ответ для экземпляра задачи. Здесь же кажется невероятным наличие короткого сертификата для подтверждения невыполнимости формулы, такое подтверждение требует проверки всех возможных подстановок, количество которых экспоненциально. Тем не менее, вопрос о существовании сертификата пока остаётся открытым.

Классу **coNP** можно дать альтернативное определение, которое иногда может оказаться удобнее. Говорят, что язык $L \subset \{0,1\}^*$ принадлежит классу **coNP**, если существуют многочлен $p : \mathbb{N} \longrightarrow \mathbb{N}$ и работающая полиномиальное время машина Тьюринга M, такие, что для всех слов $x \in \{0,1\}^*$ длины n

$$x \in L \text{ тогда и только тогда, когда для всех } u \in \{0,1\}^{p(n)} : M(x,u) = 1.$$

Следует убедиться в том, что это определение эквивалентно исходному. Возьмём отрицание от обеих частей равносильности:

$$x \notin L \text{ тогда и только тогда, когда существует } u \in \{0,1\}^{p(n)} : M(x,u) = 0.$$

Построим теперь машину M', которая представляет собой обращённую версию машины M (она допускает вместо прежнего недопускания и наоборот). Машина M' будет по-прежнему работать полиномиальное время. Равносильность принимает следующий вид:

$$x \in \overline{L} \text{ тогда и только тогда, когда существует } u \in \{0,1\}^{p(n)} : M'(x,u) = 1.$$

Но это в точности определение принадлежности языка $\overline{L}$ классу **NP**, то есть, по первому определению, $L \in$ **coNP**.

Второе определение позволяет легко установить принадлежность задачи

$$\text{TAUTO} = \{\phi \text{ — тавтология (тождественно истинная формула)}\}$$

классу **coNP**. Действительно, любая подстановка (любой сертификат u) удовлетворяет тождественно истинной формуле.

Для задач из класса **coNP** можно также определить понятие **coNP**-полноты. Говорят, что язык L является **coNP**-полным, если

1) $L \in$ **coNP**;

2) любой язык $L' \in$ **coNP** полиномиально сводится по Карпу к L.

При работе с **coNP**-полнотой удобно пользоваться следующим очевидным свойством сводимости по Карпу:

Лемма. *Если* $L' \leqslant_p L''$, *то* $\overline{L'} \leqslant_p \overline{L''}$.

Доказательство. Действительно, если $L' \leqslant_p L''$, то существует полиномиальное преобразование f, такое что:

$$x \in L' \Leftrightarrow f(x) \in L'',$$

то есть

$$x \notin L' \Leftrightarrow f(x) \notin L''.$$

Следовательно, $x \in \overline{L'} \Leftrightarrow f(x) \in \overline{L''}$, а значит, $\overline{L'} \leqslant_p \overline{L''}$. □

Доказательство этого свойства показывает, что полиномиальное преобразование при переходе к дополнениям остаётся тем же самым.

Теорема. *Задачи* $\overline{SAT}$ *и* TAUTO *являются* **coNP**-*полными.*

Доказательство. Воспользуемся определением **coNP**-полноты. Пусть $L \in$ **coNP**. Покажем, что язык L полиномиально сводится по Карпу и к $\overline{\text{SAT}}$, и к TAUTO.

Так как $L \in$ **coNP**, то $\overline{L} \in$ **NP** и существует сведение $\overline{L}$ к **NP**-полной задаче SAT:

$$x \in \overline{L} \Leftrightarrow \phi_x \in \text{SAT}$$

или, после взятия отрицания:

$$x \in L \Leftrightarrow \phi_x \in \overline{\text{SAT}} \Leftrightarrow \neg\phi_x \in \text{TAUTO}.$$

Таким образом по x построены экземпляры задач $\overline{\text{SAT}}$ и TAUTO, имеющие тот же ответ, что и доказывает их **coNP**-полноту. □

Заметим, что дополнение $\overline{\text{SAT}}$ **NP**-полной задачи SAT оказалось **coNP**-полным.

4.6.2. Соотношения между классами P, NP и coNP

Рассмотрим несколько соотношений, связывающих классы **P**, **NP** и **coNP** и полные задачи в них.

Теорема. *Если* $\mathbf{P} = \mathbf{NP}$*, то* $\mathbf{NP} = \mathbf{coNP}$*.*

Доказательство этой теоремы очевидным образом следует из замкнутости класса **P** относительно дополнения. По контрапозиции получаем:

Следствие. *Если* $\mathbf{NP} \neq \mathbf{coNP}$*, то* $\mathbf{P} \neq \mathbf{NP}$*.*

Теорема. $\mathbf{NP} = \mathbf{coNP}$ *тогда, и только тогда, когда в классе* **coNP** *имеется хотя бы одна* **NP***-полная задача.*

Доказательство. Необходимость условия очевидна, докажем его достаточность. Пусть язык $L \in \mathbf{coNP}$, L — **NP**-полный. Докажем, что $\mathbf{NP} = \mathbf{coNP}$.

Начнём с вложения $\mathbf{NP} \subset \mathbf{coNP}$. Возьмём $L' \in \mathbf{NP}$. В силу **NP**-полноты L имеет место сводимость $L' \leqslant_p L$, следовательно, $\overline{L'} \leqslant_p \overline{L}$. Так как $L \in \mathbf{coNP}$, то $\overline{L} \in \mathbf{NP}$. Построим теперь недетерминированную машину, распознающую язык $\overline{L'}$ и работающую в два этапа:

1) полиномиальное сведение экземпляра $\overline{L'}$ к экземпляру $\overline{L}$;
2) существующая полиномиальная НМТ для $\overline{L}$ (так как $\overline{L} \in \mathbf{NP}$).

Полученная НМТ работает полиномиальное время и распознаёт $\overline{L'}$, поэтому $\overline{L'} \in \mathbf{NP}$, а значит, $L' \in \mathbf{coNP}$.

Для доказательства обратного вложения берём $L' \in \mathbf{coNP}$, строим его дополнение (из **NP**) и сведение к **NP**-полному L: $\overline{L'} \leqslant_p L$, обращаем сведение: $L' \leqslant_p \overline{L}$, и снова строим полиномиальную НМТ, распознающую теперь уже L'. В результате получаем, что $L' \in \mathbf{NP}$. □

4.6.3. Альтернирующая машина Тьюринга и полиномиальная иерархия

Как известно, класс **NP** может быть определён посредством недетерминированной машины Тьюринга, в определении которой «зашито» правило допускания: машина допускает, если существует хотя бы одна ветка, по которой происходит допускание входного слова. Существование такой ветки отражается и в определении класса **NP** через сертификаты (существование u!).

Можно было бы ввести аналог недетерминированной машины, соответствующий классу **coNP**, изменив лишь правило допускания: допускание должно происходить только при условии допускания по всем веткам (квантор всеобщности в определении!). Более того, можно разрешить недетерминированной машине во время работы чередовать правила допускания, требуя попеременно то допускания по всем веткам, то существования одной ветки с допусканием. Такие машины Тьюринга называются

альтернирующими, в соответствующих им определениях через сертификаты появляются последовательности чередующихся кванторов существования и всеобщности.

Понятно, что появление альтернирования не может привести к сужению класса решаемых задач, однако неизвестно, возникает ли при этом расширение. Класс **NP**, очевидно, вложен в класс задач, решаемых альтернирующей машиной, в которой сначала идёт существование, а затем всеобщность, а этот класс, в свою очередь, вложен в класс с существованием, всеобщностью и снова существованием. Поскольку машины по-прежнему делают полиномиальное число шагов, то бесконечное объединение таких классов даёт так называемую *полиномиальную иерархию* (класс **PH**). Такую же иерархию даёт объединение классов, начинающееся с **coNP**, только там чередование начинается с всеобщности.

Вся полиномиальная иерархия, очевидно, вложена в **EXP**, поскольку число кванторов, а значит и число проходов по экспоненциальному пространству сертификатов, в каждом конкретном случае постоянно. Если $\mathbf{P} = \mathbf{NP}$, то нетрудно увидеть, что полиномиальная иерархия схлопывается до **P**, однако схлопывание может произойти и на любом другом её уровне (могут оказаться равными два класса с одинаковым количеством чередований, но разными начальными правилами допускания). Если окажется, что $\mathbf{NP} = \mathbf{coNP}$, то это схлопывание иерархии на первом уровне. В настоящее время неизвестно, схлопывается ли полиномиальная иерархия вообще, и на каком уровне это происходит.

4.7. Классы EXP и NEXP

Напомним, что класс **EXP** определяется следующим образом:

$$\mathbf{EXP} = \bigcup_{c \geqslant 1} \mathbf{DTIME}(2^{n^c}).$$

Определим аналогичным образом класс **NEXP**:

$$\mathbf{NEXP} = \bigcup_{c \geqslant 1} \mathbf{NTIME}(2^{n^c}).$$

Имеет место очевидное вложение: $\mathbf{EXP} \subset \mathbf{NEXP}$, однако неизвестно, является ли вложение строгим. Тем не менее, можно утверждать следующее:

Теорема. *Если* $\mathbf{P} = \mathbf{NP}$*, то* $\mathbf{EXP} = \mathbf{NEXP}$*.*

Доказательство. Покажем, что в условиях теоремы $\mathbf{NEXP} \subset \mathbf{EXP}$. Пусть $L \in \mathbf{NEXP}$, то есть для некоторого c язык $L \in \mathbf{NTIME}(2^{n^c})$, и M — соответствующая недетерминированная машина Тьюринга.

Рассмотрим язык, в котором все слова L дополнены единицами (*padding*):

$$L_{pad} = \{<x, 1^{2^{|x|^c}}> : x \in L\}.$$

Для этого языка можно построить следующую распознающую его недетерминированную машину: сначала проверяем структуру входного слова y, действительно ли оно содержит корректное число единиц, и, если для некоторого z получается, что $y =< z, 1^{2^{|z|^c}} >$, то запускаем на этом z машину M (в противном случае y сразу не

допускается). Проверка структуры, очевидно, может быть выполнена за полиномиальное (даже линейное) время, а машина M делает $2^{|z|^c}$ шагов, то есть полиномиально по длине y (слово y специально было сделано настолько длинным!). Таким образом, мы получили, что $L_{pad} \in \mathbf{NP}$ и, следовательно, $L_{pad} \in \mathbf{P}$.

Детерминированная машина для распознавания языка L теперь может сначала дополнять x достаточным количеством единиц (экспоненциальное число шагов), а затем запускать детерминированную машину для распознавания L_{pad} (она существует, поскольку $L_{pad} \in \mathbf{P}$). Следовательно, $L \in \mathbf{EXP}$. □

По контрапозиции получаем:

Следствие. *Если* $\mathbf{EXP} \neq \mathbf{NEXP}$, *то* $\mathbf{P} \neq \mathbf{NP}$.

4.8. Пространственная сложность

4.8.1. Классы PSPACE и NPSPACE

Класс **PSPACE** определяется как класс языков, для распознавания которых существуют детерминированные машины Тьюринга, использующие не более чем полиномиальный по длине входного слова объём памяти (количество ячеек ленты). Аналогично определяется класс **NPSPACE**.

Ясно, что $\mathbf{P} \subset \mathbf{PSPACE}$ и $\mathbf{NP} \subset \mathbf{NPSPACE}$ (за полиномиальное время работы невозможно воспользоваться более чем полиномиальным объёмом памяти). Заметим, что в определение классов пространственной сложности не входит требование завершаемости вычислений (то есть рекурсивности соответствующих языков), тем не менее, можно установить следующее:

Теорема. $\mathbf{PSPACE} \subset \mathbf{EXP}$.

Доказательство. Пусть $L \in \mathbf{PSPACE}$, то есть существует распознающая L детерминированная машина Тьюринга M, использующая не более $p(n)$ ячеек ленты, где p — некоторый многочлен. Это означает, что максимальная длина конфигурации такой машины есть $p(n) + 1$ (содержимое ячеек и состояние). Посчитаем, сколько всего различных конфигураций может у неё быть. Пусть t — это число ленточных символов, а s — число состояний. Если машина использует $p(n)$ ячеек ленты, то число различных конфигураций можно вычислить как $sp(n)t^{p(n)}$ — выбираем любое положение для любого состояния и произвольным образом расставляем по $p(n)$ ячейкам ленточные символы. Если теперь положить c равным суммарному количеству ленточных символов и состояний $t + s$, то число состояний $sp(n)t^{p(n)} \leqslant c^{1+p(n)} = (t+s)^{1+p(n)} = t^{1+p(n)} + (1+p(n))st^{p(n)} + \ldots$, где второе слагаемое справа очевидно больше левой части неравенства.

Таким образом, число различных конфигураций ограничено сверху некоторым значением $c^{1+p(n)}$, то есть экспоненциально по длине входного слова. Ясно, что повторение конфигурации означает, что машина зациклилась. Соответственно, если входное слово допускается машиной, то это должно произойти до повторения конфигурации, если же зацикливание произошло, то входное слово не допускается. Это наблюдение позволяет построить машину M', которая будет делать не более чем экспоненциальное число шагов и распознавать при этом тот же язык L. Такая машина

будет полностью повторять работу M, но иметь при этом дополнительную ленту — счётчик шагов. Если использовать для этой ленты c-ичную систему счисления, то для счётчика будет достаточно иметь $1 + p(n)$ ячеек — достижение счётчиком максимального значения $c^{1+p(n)}$ будет означать, что исходная машина зациклилась (её конфигурация гарантированно повторилась), а значит входное слово не допускается и можно завершать работу с ответом «нет». Если исходная машина допускает слово или завершается до повторения конфигурации в недопускающем состоянии, то это будет обнаружено до обнуления счётчика и выдано в качестве ответа машины M'. Машина M' по построению принадлежит **EXP**, поскольку делает не более $c^{1+p(n)}$ шагов и распознает тот же язык, что и M. Многоленточную машину M' теперь можно преобразовать в одноленточную, число шагов при этом увеличится не более чем квадратично, то есть останется экспоненциальным, поэтому $\mathbf{PSPACE} \subset \mathbf{EXP}$. □

Аналогичным образом доказывается теорема для недетерминированных машин Тьюринга, поскольку соображение относительно повторения конфигураций относится к ним в той же мере.

Теорема. $\mathbf{NPSPACE} \subset \mathbf{NEXP}$.

Ещё одна (неожиданная) теорема, доказанная Уолтером Сэвитчем, утверждает, что недетерминированность в случае ограничения по памяти не даёт особых преимуществ. Причина этого состоит в том, что, в отличие от времени, память может быть использована повторно, а значит, возможность одновременного исследования нескольких конфигураций, значительно экономя время, не даёт преимуществ по памяти.

Теорема (Сэвитча)**.** $\mathbf{PSPACE} = \mathbf{NPSPACE}$.

Доказательство. Раздел 11.2.3 книги Хопкрофта, Мотвани, Ульмана «Введение в теорию автоматов, языков и вычислений» (с. 487–489). □

Доказанные теоремы завершают обоснование следующих известных на данный момент вложений введённых ранее классов сложности:

$$\mathbf{P} \subset \mathbf{NP}, \mathbf{coNP} \subset \mathbf{PSPACE} \subset \mathbf{EXP} \subset \mathbf{NEXP}.$$

Можно также доказать, что $\mathbf{PH} \subset \mathbf{PSPACE}$.

4.8.2. PSPACE-полнота и пример PSPACE-полной задачи

Раздел 11.3 книги Хопкрофта, Мотвани, Ульмана «Введение в теорию автоматов, языков и вычислений» (с. 489–498).

4.9. Обзор других разделов теории сложности

- *Задачи, разрешимые в логарифмическом пространстве.*
- *Сложность вычислений на логических схемах.*

- *Рандомизированные вычисления и вероятностные машины Тьюринга.*
- *Интерактивные доказательства.*
- *Вероятностно-проверяемые доказательства (PCP) и пределы аппроксимации NP-трудных задач.*