Основы теории алгоритмов

Коротков М.А.
Степанов Е.О.

14 сентября 2003 г.

Оглавление

1 Основы теории алгоритмов

Ранее мы занимались изучением языков, предназначенных для описания *рассуждений*. Теперь мы "поменяем угол зрения" и будем рассматривать языки, предназначенные для записи конкретных *действий*. Языки логики предикатов относятся к классу *декларативных* (описательных): смысл элемента такого языка (формулы) заключается в *описании* чего-либо. Те же языки, к изучению которых мы приступаем, можно назвать *языками инструкций*, так как их элементы, называемые часто инструкциями, являются предписаниями исполнителю (человеку, компьютеру, другому вычислительному устройству) совершить определенный набор действий. Поскольку набор точных инструкций для выполнения конкретных действий принято называть алгоритмом или программой, такие языки инструкций часто называются *алгоритмическими* языками или *языками программирования*. Последние два термина часто употребляются в слегка разных контекстах: под языком программирования обычно понимают язык инструкций для реально существующего вычислительного устройства – компьютера (**Fortran, Basic, Pascal, C++** и т.п.), а термин *алгоритмический язык* употребляется в более широком смысле и включает в себя как языки программирования, так и другие языки инструкций, в частности псевдоязыки, предназначенные для описания алгоритмов.

Смещение нашего внимания с логических языков на языки инструкций обусловлено желанием дать ответы на те вопросы, котроые остались открытыми в нашем исследовании возможностей логических языков, а именно, на вопросы о том, каким образом проверять истинность различного рода семантических утверждений о формулах языков логики предикатов (например, как проверить, является ли заданная формула языка логики первого или второго порядка тавтологией). Мы уже отметили, что непосредственная проверка семантических свойств формул логических языков при помощи определений практически никогда не представляется возможной, и поэтому специально рассмотрели механизмы, предназначенные для конструктивной проверки семантических свойств (формальные системы). Мы также доказали, что среди таких механизмов есть "хорошие", то есть одновременно корректные и полные, по крайней мере для языков логики первого порядка. Пользуясь такой формальной системой, можно заменить проверку того, является ли данная формула тавтологией, на поиск вывода данной формулы при помощи формальной системы из пустого множества посылок (иначе говоря, поиску *доказательства* данной формулы из пустого множества посылок). Однако, остается неясным, каким образом искать такой вывод. Можно ли, например, предъявить конкретный алгоритм вывода любой тавтологии? Интересен и другой вопрос: можно ли предъявить такой алгоритм, который по заданной формуле логического языка даст однозначный ответ, выводима ли данная формула из пустого множества посылок, то есть является ли тавтологией, или нет?

Мы пока подобного рода вопросами не интересовались. Кстати, на практике многие люди и не интересуются ими. Например, математики ищут доказательства математических утверждений, как правило не задумываясь о том, можно ли предъявить универсальный алгоритм их поиска. Подобного рода вопросы можно, конечно, задавать не

только о логических языках. Например, игроков в шахматы, возможно, интересет вопрос о том, может ли партия в шахматы всегда быть выиграна белыми (естественно, при соблюдении правил игры). Ответ в данном случае положительный: описание алгоритма решения этой задачи можно найти у Дональда Э. Кнута ("Искусство программирования", том 1, стр 314). Правда, проблема в том, что для выполнения этого алгоритма требуется невероятно большое время даже при использовании самых современных и быстродействующих вычислительных устройств, причем нет никакой надежды, что в обозримом будущем появятся компьютры, способные реализовать этот алгоритм за приемлемое время.

В этом разделе мы покажем, что использование языков инструкций помогает ответить на ряд вопросов о логических языках, подобных только что рассмотренным. В качестве побочного эффекта исследования мы сможем доказать и целый ряд нетривиальных и практически важных утверждений о языках инструкций, в частности, о невозможности решения некоторых важных алгоритмических задач.

Привыкший к математической строгости изложения читатель наверняка уже обратил внимание на то, что мы неоднократно и весьма вольно пользовались понятием алгоритма, не давая его строго определения. Дело в том, что нам бы не хотелось посвящать слишком много времени формализации этого понятия. Во-первых, потому, что читатель, как мы надеемся, обладает хотя бы элементарными основами алгоритмической культуры, под которой понимается не обязательно умение писать программы для компьютера, но хотя бы умение записать кулинарный рецепт так, чтобы его могла применить любая домохозяйка: написание такого рецепта в математических терминах как раз и есть составление алгоритма приготовления блюда. Во-вторы, вопросу о том, что такое алгоритм посвящены многочисленные пухлые тома специальной литературы, чтение которых, как показвывает практика, ни в коей мере не способствуют практической работе с алгоритмами: так, чтобы грамотно записать рецепт приготовления котлет, не нужно знать формального опредления алгоритма. Так что для наших целей достаточно будет лишь общего понимания алгоритма, как *эффективного описания* некоторых действий. Данный термин был впервые введен в 1936 году А. Черчем, который предложил считать действие M эффективным (или эффективно описанным), если

- M описано конечным числом точных инструкций, каждая из которых выражена конечных числом символов;
- в случае, когда M проводится правильно, оно обязательно дает желаемый результат за конечное время (конечное число шагов);
- M может быть в принципе проведено человеком вооруженным "пером и бумагой";
- проведение M не требует "ни проницательности ни изобретательности" от исполнителя. "Человек, вооруженный карандашом, бумагой и системой знаний о предмете представляет собой пример эффективного вычислителя" (Тьюринг, 1948).

Под алгоритмом мы будем понимать эффективное описание действия, при этом позволяя себе достаточно вольно обращаться с терминами алгоритм, программа, эффективное описание, употребляя их *почти* как синонимы. Единственное отличие, на которое стоит обратить внимание, состоит в том, что программа в нашем понимании может выполняться "бесконечное" время (например, программа, печатающая в бесконечном цикле одно и то же слово). Термин "бесконечное" не зря взят в кавычки, ведь бесконечное время является абстракцией, которую в реальной жизни никто не видел. Опять-таки, любители математического формализма могут считать бесконечно выполняющуюся программу набором бесконечного числа *конечных* алгоритмов. Предоставляем желающим возможность самим сформулировать соответствующие "строгое" определение, если они видят в этом необходимость.

Примером алгоритма из школьной арифметики является алгоритм Евклида проверки того, является ли заданное число простым. Рецепт из поваренной книги может тоже, хотя и с некоторыми оговорками, рассматриваться как алгоритм. Рецепт всегда конечен (иначе исполнитель умрет с голода), но вот инструкции в рецепте не всегда точны (вряд ли кто-то станет утверждать, что предписания "возьмите небольшую кастрюлю", "добавьте сахар по вкусу" и подобные им могут считаться действительно точными), поэтому и желаемый результат (вкусная пища) не всегда достигается. На самом деле для достижения результата как раз требуется "изобретательность и проницательность" исполнителя (в данном случае мастерство повара), что противоречит понятию эффективного действия. Тем не менее, если пренебречь некоторой неточностью, можно считать рецепт примером алгоритма.

Нас будут в дальнейшем интересовать два типа алгоритмов: алгоритмы распознавания и алгоритмы вычисления. Начнем с алгоритмов распознавания. Рассмотрим алфавит $\mathcal{A}$. Предположим, задано по некоторым правилам множество $L \subset \mathcal{A}^*$. Нас интересует следующий вопрос: можно ли предъявить такой алгоритм, который будет распознавать элементы множества L? Иными словами, можно ли написать такую программу, которая при подаче на вход слова из $\mathcal{A}^*$ через конечное время (т.е. за конечное число шагов) выдавала бы ответ, принадлежит ли данное слово множеству L или нет. Введем следующее определение.

Определение 1.1 *Множество $L \subset \mathcal{A}^*$ называется разрешимым, если существует такой алгоритм (программа), при подаче на вход которого произвольного слова из A^* за конечное число шагов алгоритм завершается выдачей ответа, принадлежит ли данное слово множеству L или нет. Соответствующий алгоритм (программа) называется разрешающим (разрещающей) для данного множества.*

Нас будет интересовать и другой вопрос: можно ли написать программу, которая генерировала бы на выходе все элементы данного множества, и только элементы этого множества (естественно, эта программа должна выполняться бесконечно, если множество L бесконечно). Введем соответствующее определение.

Определение 1.2 *Множество $L \subset \mathcal{A}^*$ называется перечислимым, если существует такая программа, которая выводит на выход все слова множества L, и только их, то есть в результате работы программы на выходе рано или поздно появится каждое слово из L, и никаких других слов не появится. Соответствующая программа называется перечисляющей программой.*

Подчеркнем, что перечисляющая программа может печатать элементы множества L в любом порядке и с любым количеством повторений. Определение 1.2 формулирует лишь два требования:

- любое слово из L рано или поздно будет выведено;
- никаких слов из $A^* \setminus L$ выведено не будет.

Прежде чем перейти к свойствам разрешимых и перечислимых множеств, приведем некоторые простейшие примеры. Во-первых, если множество L состоит из конечного числа элементов, то оно очевидно разрешимо и перечислимо. Вот чуть менее тривиальный пример.

Пример 1.1 *Рассмотрим алфавит английского языка. Пусть L – это множество палиндромов (слов, которые одинаково читаются в обе стороны)* [1].

Это множество перечислимо: мы можем написать программу, генерирующую последовательно палиндром за палиндромом. Она никогда не остановится, но каждый палиндром рано или поздно появиться на выходе программы. Приведем перечисляющую программу (пример написан на псевдокоде, а не конкретом языке программирования)

```
alphabeth_letters = {set_of_english_letters}

function generate_next_palindrome(previous_palindrome, needed_length)
{
  if (length(previous_palindrome) == needed_length)
    print(previous_palindrome);
  else foreach (alphabeth_letters as letter)
    generate_next_palindrome(letter + previous_palindrome + letter, needed_length);
}

start_length = 1;
while (TRUE) do
{
  generate_next_palindrome("", start_length);
  foreach (alphabeth_letters as letter)
    generate_next_palindrome(letter, start_length);
  start_length++;
}
```

[1] Для любопытных: на `http://www.palindromelist.com` можно найти палиндром, длиной почти 25000 символов (правда, в данном палиндроме не учитываются пробелы и знаки препинания).

Это же множество к тому же и разрешимо, потому что можно написать программу, которая брала бы на вход любую строчку из $\mathcal{A}^*$ и указывала бы, палиндром это или нет, за конечное время. Предлагаем читателю написать такую программу на любом доступном ему языке программирования.

Упражнение 1.1 *Пусть алфавит $\mathcal{A}$ является множеством десятичных цифр ($\mathcal{A} := \{0, 1, \ldots, 9\}$). При этом множество $\mathcal{A}^*$ будем отождествлять с множеством натуральных чисел. Пусть L – множество тех слов из $\mathcal{A}^*$, которые соответствуют простым числам. Является ли это множество разрешимиым? перечислимым?*

Возникает вопрос, в каком соотношении находятся перечислимые множества и разрешимые множества. Ответ на него дает следующее предложение.

Предложение 1.1 *Всякое разрешимое множество перечислимо.*

Для доказательства нам потребуется определение лексикографического порядка в $\mathcal{A}^*$.

Определение 1.3 *Порядок $(\mathcal{A}^*, <)$ называется лексикографическим, если $p < q$, когда либо длина слова $p \in \mathcal{A}^*$ меньше длины слова $q \in \mathcal{A}^*$, либо, при равной длине слов p и q, слово p предшествует q в алфавитном порядке.*

Теперь можно доказать предложение 1.1

Доказательство: Следующий алгоритм перечисляет элементы разрешимого множества $L \subset \mathcal{A}^*$: в цикле генерируем в лексикографическом порядке строчки из $\mathcal{A}^*$ (сначала в алфавитном порядке слова длины 1, потом в алфавитном порядке слова длины 2 и т.п.), и подаем каждое сгенерированное слово на вход разрешающей программе. Последняя за конечное число шагов завершится, выдав ответ, либо о том, что данное слово принадлежит L, либо, что оно не принадлежит L. В первом случае выдаем сгенерированное слово на выход, во втором случае игнорируем его и переходим к генерации следующего слова из $\mathcal{A}^*$. □

Справедливы следующие простые утверждения.

Предложение 1.2 *Если разрешимо множество $L \subset \mathcal{A}^*$, то разрешимо, а значит и перечислимо, его дополнение $\mathcal{A}^* \setminus L$.*

Доказательство: Если поменять в разрешающем алгоритме для множества L выдачу ответа "слово принадлежит L" на "слово не принадлежит $\mathcal{A}^* \setminus L$" и наоборот, то получится разрешающий алгоритм для $\mathcal{A}^* \setminus L$. □

Предложение 1.3 *Множество $L \subset \mathcal{A}^*$ разрешимо, если и только если перечислимы одновременно оно само и и его дополнение $\mathcal{A}^* \setminus L$.*

Доказательство: Из предложений 1.1 и 1.2 следует, что разрешимость множества L влечет перечислимость его самого и его дополнения $\mathcal{A}^* \setminus L$. Предположим теперь, что L и $\mathcal{A}^* \setminus L$ перечислимы, и докажем, что в этом случае L должно быть разрешимым. Разрешающий алгоритм для L следующий: после ввода слова из $\mathcal{A}^*$ запускаются параллельно перечисляющие программы для L и для $\mathcal{A}^* \setminus L$. Рано или поздно принятое на вход слово появится на выходе либо первой либо второй программы в силу определения перечислимости множества. В первом случае следует выдать ответ о принадлежности проверяемого слова множеству L, во втором – о непринадлежности, после чего следует завершить работу программы. "Параллельность" запуска сразу двух перечисляющих программ достигается чередованием передачи управления на каждую из этих программ, скажем, через регулярные промежутки времени. Иначе говоря, сначала запускается первая программа, через определенный интервал времени она приостанавливается и управление передается второй программе, а затем через регулярные интервалы времени эти программы "меняются местами" (работавшая программа приостанавливается, а управление передается на ранее приостановленную программу). □

Теперь мы можем дать ответ на гораздо менее тривиальный вопрос о том, существуют ли множества, которые не являются не только разрешимыми, но даже и перечислимыми. Интуитиция возможно подсказывает, что таких множеств нет, ибо в повседневной жизни мы имеем дело либо с конечными множествами, либо с множествами, построенными каким-либо эффективным способом. Однако в данном случае доверять интуиции нельзя, в силу следующего утверждения.

Предложение 1.4 *Существуют неперечислимые множества.*

Доказательство: Рассмотрим произвольный конечный алфавит $\mathcal{A}$. Очевидно, что мощность множества перечислимых подмножеств $\mathcal{A}^*$ не превышает мощности множества всех перечисляющих программ. Последние же имеют мощность не превышающую мощность всех вообще программ, работающих над алфавитом $\mathcal{A}$. Так как все такие программы записываются в виде конечных текстов на некотором языке с конечным алфавитом, то множество всех таких программ не более чем счетно. Следовательно, не более чем счетно и множестов всех перечислимых подмножеств $\mathcal{A}^*$. С другой стороны, множество $\mathcal{A}^*$ счетно, а значит, множество всех подмножеств $\mathcal{A}^*$ более чем счетно. Таким образом среди подмножеств $\mathcal{A}^*$ найдутся и неперичислимые. □

Нам понадобятся в дальнейшем также понятия вычислимых и полувычислимых функций. Сформулируем соответствующее определение.

Определение 1.4 *Пусть $\mathcal{A}$ – заданный алфавит. Функция $f : Domf \subset \mathcal{A}^* \to \mathcal{A}^*$ называется вычислимой, если существует программа, которая, при подаче на вход слова $a \in \mathcal{A}^*$, за конечное время (за конечное число шагов) выдаст $f(a)$, если $a \in Domf$, либо, в противном случае, заранее оговоренный признак того, что $a \notin Domf$. Соответствующая программа называется вычисляющей. Обычно приходится иметь дело с функциями, не принимающими значение $\square$ (пустое слово), так что в этом случае*

разумно резервировать $\square$ *в качестве признака того, что* $a \notin Domf$*. В дальнейшем мы будем всегда считать, что имеем дело именно с такой ситуацией.*

Фунция $f : Domf \subset \mathcal{A}^* \to \mathcal{A}^*$ *называется полувычислимой, если существует программа, которая, при подаче на вход слова* $Domf$*, за конечное время (за конечное число шагов) выдаст* $f(a)$*. Соответствующая программа называется полувычисляющей. Работа полувычисляющей программы при подаче на вход слова из* $\mathcal{A}^* \setminus Domf$ *не определена (в этом случае программа может даже работать бесконечно).*

Очевидно, что всякая вычислимая функция является и полувычислимый, так всякая вычисляющая программа является одновременно полувычисляющей. Также очевидно, что полувычислимая всюду определенная функция является вычислимой. Более точно связь между вычислимостью и полувычислимостью характеризуется следующим утверждением.

Предложение 1.5 *Функция* $f : Domf \subset \mathcal{A}^* \to \mathcal{A}^*$*, где* $\mathcal{A}$ *– заданный алфавит, вычислима если и только если полувычислимы одновременно* f *и характеристическая функция множества* $Domf$ *(последняя определяется как функция, принимающая значение* $\square$ *на* $\mathcal{A}^* \setminus Domf$ *и любое наперед заданное непустое значение из* $\mathcal{A}^*$ *на* $Domf$*).*

Доказательство: Если характеристическая функция множества $Domf$ полувычислима, то она и вычислима, так как она определена всюду на $\mathcal{A}^*$, при этом соответствующая полувычисляющая программа является очевидно и вычисляющей. Если к тому же f полувычислима, то вычисляющую программу для f можно построить следующим образом: при подаче на вход слова $a \in \mathcal{A}^*$ сначала запускается вычисляющая программа для характеристической функции множества $Domf$; если последняя завершается выдачей ответа о том, что $a \in Domf$, то запускается полувычисляющуая программа для функции f, в противном случае выводится $\square$. $\square$

Теперь по аналогии с вопросом о существовании неперечислимых множеств можно задаться вопросом о существовании функций не являющихся полувычислимыми (такие функции принято называть абсолютно невычислимыми). Ответ на него дает следующее утверждение.

Предложение 1.6 *Существуют абсолютно невычислимые функции.*

Доказательство: Полная аналогия с доказательством предложения 1.4. Упражняйтесь. $\square$

2 Тезис Черча. Регистровые машины

Читатель, приверженный формальной математической строгости изложения, наверняка удивился, читая предыдущий параграф, поскольку последний весь состоял из не вполне понятных слов, так что все приведенные там утвержедения на самом деле нельзя

считать не только доказанными, но даже, строго говоря, корректно сформулированными. Стоило бы описать инструкцию, алгоритм и исполнителя, причем так, чтобы описание соответствовало нашему интуитивному пониманию. Есть множество различных построений: машина Тьюринга, рекурсивные и частично рекурсивные функции Черча, алгорифмы Маркова, машина Поста, регистровая машина. При этом как бы вы ни формализовали понятие алгоритма, набор алгоритмов вы полчите, в некотором смысле, "совпадающий" с алгоритмами для других вычислителей. Имеет мето следующий опытный факт: все вычислительные устройства эквивалентны в смысле алгоритмов, которые могут на них выполняться. Данное утверждение называют тезисом Черча.

Тезис 2.1 *Если существует эффективный метод вычисления значений функции, то они могут быть вычислены и с помощью машины Тьюринга (или другого эффективного вычислителя)*

Впоследствии утверждение тезиса было расширено до следующего: "любое эффективно описанное действие реализуется вычислительной машиной". Тезис Черча (иногда его называют тезисом Черча-Тьюринга, поскольку похожее утверждение было сформулировано Тьюрингом примерно в то же время) – утверждение эмпирического, а не математического характера, он связывает формализованные понятия (описание конкретного вычислителя) с не формализованными понятиями алгоритма, эффективного описания.

Если мы конкретизируем таким образом понятия алгоритма, программы, вычислительного устойства, то мы получим уже формальные определения перечислимости и разрешимости множеств. В зависимости от конкретизации будут получатся разные определения.

Конкретизация, которую будем рассматривать мы, называятся регистровой машиной. Исполнителем будет являтся иедальное вычислительное устройство – регистровая машина. Для регистровых машин введем понятия: регистровая разрешимость, регистровая перечислимость, регистровая вычислимость функций, регистровая полувычислимость функций. Все эти определения получаются заменой понятия "алгоритм" на "алгоритм для регистровых машин."

Тезис Черча для нас будет выглядеть следующим образом: разрешимые и перечислимые множества совпадают с разрешимыми и перечислимыми множествами в смысле регистровых машин.

Определение 2.1 *Регистровая машина определяется как набор элементов:*

- *Внешнего алфавита $\mathcal{A}$, с которым она работает;*
- *Пустого символа $\square \in \mathcal{A}$ (напечатать пустой символ – это значит ничего не напечатать, но задействовать при этом устройство печати);*
- *Набора регистров: $R_0, \ldots, R_m$;*

- *Набора операторов для описания алгоритмов для регистровой машины следующего вида:*

  1. $\alpha\ LET\ R_i = R_i + a_i$
  2. $\alpha\ LET\ R_i = R_i - a_i$;
  3. $\alpha\ IF\ R_i = \square\ THEN\ \alpha'\ ELSE\ \alpha_0\ OR\ \alpha_1\ OR\ \ldots\ \alpha_n$;
  4. $\alpha\ PRINT$
  5. $\alpha\ halt$

  Где $a_i \in \mathcal{A}$, α *– номер, метка.*

Алфавит представляет собой слова: $\mathcal{A} = \{a_0, a_1, \ldots, a_n\}$ Рассмотрим приведенные операторы:

- $\alpha\ LET\ R_i = R_i + a_i$
  в содержимое регистра (ленты) R_i справа дописать букву a_i;

- $\alpha\ LET\ R_i = R_i - a_i$;
  если в регистре R_i слово заканчивается справа на букву a_i, то эта буква стирается;

- $\alpha\ IF\ R_i = \square\ THEN\ \alpha'\ ELSE\alpha_0\ OR\alpha_1\ OR\ \alpha_2\ \ldots\ OR\alpha_n$;
  условный переход: если R_i пусто, то тогда сделать переход на метку α', в противном случае, если R_i заканчивается на a_i, то перейти на метку α_i (если заканчивается на a_1 – перейти на α_1, на a_2 – перейти на α_2, и т.д.)

- $\alpha\ PRINT$
  напечатать содержимое регистра R_0 (R_0 – регистр ввода-вывода);

- $\alpha\ halt$
  оператор завершения программы.

К оператору IF необходим некоторый комментарий: так как регистровые машины являются теоритическим построением, то мы вправе требовать, чтобы в операторе были перечислены *все* слова алфавита (их конечное число).

Определение 2.2 *Программой для регистровой машины или регистровой программой будем называть конечный набор инструкций-операторов, определенных выше. Каждая инструкция* β_i *имеет метку* i*, то есть они нумеруются последовательно. Во всех инструкциях вида*

$$\beta_i\ IF\ R_i = \square\ THEN\ \beta'\ ELSE\ \beta_0\ OR\ \beta_1\ OR\ \ldots\ \beta_n$$

метки,

$$\beta_1,\ \beta_2,\ \ldots,\ \beta_n$$

не превосходят метки β_k (последней метки в программе). Иными словами, переход осуществляется только на реально существующие метки. Кроме того, инструкция β_k имеет такой вид:

$$\beta_k = k\ halt,$$

и инструкции halt более в программе не встречается.

Идеальность вычислителя заключается в следующем:

- во-первых, мы предполагаем, что регистр вмещает в себя *любое* количество информации, то есть представляет собой бесконечную ленту.

- во-вторых, мы не ставим ограничений на длину программы (если бы регистровая машина была реализована на компьютере с 256Mb оперативной памяти, то программа максимальной длины, работающая над алфавитом ASCII, занимала бы "всего" порядка 15000 машинописных страниц, то есть 435000 километров текста)

Определение 2.3 *Будем говорить, что регистровая программа P, при подаче слова $\xi \in A^*$ на вход, рано или поздно останавливается ($\xi \in A^* \to halt$), если программа P начинается со слова ξ в регистре R_0 и пустого слова во всех остальных регистрах. Далее выполняются последовательно все инструкции программы в соответствии с вышеизложенными правилами, и рано или поздно происходит остановка. В противном случае будем говорить, что программа P при подаче на вход слова ξ зацикливается.*

Определение 2.4 *Будем говорить, что программа P при подаче на вход слова ξ выдает на выходе слово $\eta \in \mathcal{A}^*$, если после запуска программы рано или поздно произойдет остановка, и кроме того к моменту остановки на выходном устройстве будет напечатано слово η и никаких других слов.*

Напечатать пустое слово – значит не напечатать ничего, но задействовать при этом выходное устройство. Предположим, что, например, на выходном устройстве есть еще лампочка, показывающая, было оно задействовано или нет.

Пример 2.1 *Алфавит $\mathcal{A}$ такой: $\mathcal{A} = \{\square, |\}$. В этом алфавите интерпретируем $\square$ как число 0, $|$ как число 1, $||$ – 2, ... и так и далее. Написать программу, которая бы определяла, какое на входе число, четное или нечетное. Если число нечетное, то она бы печатала $\square$ (останавливалась бы, ничего не печатая), в противном случае печатала бы $|$.*

Приведем простой вариант этой программы.

$$
\begin{array}{l}
1\ LET\ R_0 = R_0 - | \\
2\ IF\ R_0 = \square\ THEN\ 5\ ELSE\ 3 \\
3\ LET\ R_0 = R_0 - | \\
4\ IF\ R_0 = \square\ THEN\ 7\ ELSE\ 1 \\
5\ PRINT\ R_0 \\
6\ IF\ R_0 = \square\ THEN\ 9\ ELSE\ 9 \\
7\ LET\ R_0 = R_0 + | \\
8\ PRINT\ R_0 \\
9\ halt
\end{array}
$$

Программу для регистровой машины, как и программу на любом другом языке программирования обычно можно упростить.

$$
\begin{array}{l}
1\ LET\ R_0 = R_0 - | \\
2\ IF\ R_0 = \square\ THEN\ 6\ ELSE\ 3 \\
3\ LET\ R_0 = R_0 - | \\
4\ IF\ R_0 = \square\ THEN\ 5\ ELSE\ 1 \\
5\ LET\ R_0 = R_0 + | \\
6\ PRINT\ R_0 \\
7\ halt
\end{array}
$$

Теперь можно дать окончательные определения разрешимых и перечислимых множеств.

Определение 2.5 *Множество L называется регистрово разрешимым, если существует такая программа для регистровой машины, которая, получив на вход слово, за конечное число шагов выдает ответ: либо "да", данное слово принадлежит множеству, либо "нет", данное слово не принадлежит множеству.*

Множество $L \in \mathcal{A}^*$ называется регистрово перечислимым, если существует такая программа для регистровой машины, которая выводит все слова множества L, и только их.

Упражнение 2.1 *Алфавит $\mathcal{A}$ такой: $\mathcal{A} = \{a, b\}$. Напишите разрешающую и перечисляющую регистровые программы для множества палиндромов.*

Приведенная конструкция (регистровая машина) конечно не предсталяет собой удобное для программирования устройство, но, в силу тезиса Черча, является эквивалентом любого другого устройства. А так как достаточно формализованные вычислительные устройства удобнее при теоритических построениях, то мы будем рассматривать именно ее.

3 Некоторые алгоритмические массовые проблемы

Когда мы сформулировали, что такое алгоритм, мы можем получит несколько конструктивных результатов. Первый вопрос, который мы поставим, это вопрос о том, можно ли разрешить множество всех самоприменимых регистровых программ, иначе говоря, существует ли алгоритм, определяющий корректно ли поданная на вход программа работает с собственным кодом? И второй вопрос: а можно ли разрешить множество всех программ корректно работающих с заданными начальными данными (например, пустым словом)?

Данные вопросы представляют собой приметы массовых проблем, а именно, мы поставили цель найти алгоритм рашающий данную задачу для *всех* программ.

Возьмем программу вычисляющую значение некоторой функции. Правильность программы обычно проверяется путем тестирования, то есть проверки программы на конечном (пусть и большом) множестве входных данных. Множество же всех возможных входных данных может оказаться бесконечным. Анализ кода дает принципиально иной результат – мы можем гарантировать, что, при соблюдении некоторых условий, программа вычислит *верное* значение функции, а не сообщать, что программа протестирована на 2 (5, 100, 1000) тестах. Возникает вопрос: а всегда ли осуществима такая проверка? Такие проверки хотелось бы проводить с помощью другой программы–верификатора. Вопрос же заключается в следующем: а существует ли такой верификатор (и если да, то как его построить, и имеет ли смысл его применять).

На самом деле, такой проверяющий алгоритм построить *нельзя*, отчасти по причине его универсальности. Однако это не означает невозможность решения более узкой задачи. В таком случае говорят: *массовая проблема* такого рода неразрешима. То есть, хотя и не существует общего алгоритма проверки программы, но для некоторых (довольно узких) классов программ его можно построить.

Программа является достаточно неудобным объектом для операций над ней, значительно проще оперировать с числами, поэтому сейчас мы установим соответствие между числами и программами. *Внешний алфавит* программы $\mathcal{A}$, $\mathcal{A} = \{|\}$, множество слов над ним—$\mathcal{A}^*$; (здесь мы неявно подразумеваем также наличие пустого символа, то есть, $\mathcal{A} = \{\square, |\}$). Расширим этот алфавит в соответствии с языком, на котором пишутся программы: $\mathcal{B} := \mathcal{A} \cup \{a \dots z,\ 0 \dots 1, =, -, +, \square, ;\}$ ($\square$ здесь—не "пустой символ", а "символ, обозначающий пустой символ"), $\mathcal{B}$—*алфавит регистровых программ*; тогда $\mathcal{B}^*$—множество слов над алфавитом $\mathcal{B}$.

Определение 3.1 Эффективная нумерация программ Геделя *ставит в соответствие каждой регистровой программе натуральное число (называемое номером Геделя программы) по следующему правилу: слова из множества $\mathcal{B}^*$ перебираются в лексикографическом порядке; в случае, если данное слово является программой, ему присваивается порядковый номер.*

Замечание 3.1 *То, что очередное слово является программой, устанавливается с помощью синтаксического анализатора (грубо говоря, слово является программой, если*

отсутствуют ошибки компиляции). В нашем случае можно взять слово, и поочередно сравнивать его со всеми словами (генерируемыми программой, которая считает номера Геделя) как только слова совпадут, так сразу мы и найдем номер Геделя нашей программы. Причем так как слова мы генерируем по росту их длины, то мы также сможем узнать что слово не является программой. Таким образом у нас есть конструктивный алгоритм, устанавливающий соответствие между номерами программ (числами Геделя) и самими программами.

Номера Геделя программ определим следующим образом: элементу $a_0 \in \mathcal{A}^*$ сопоставим номер $n := \left\{\underbrace{a_0, \ldots, a_0}_{n \text{ раз}}\right\}$. Теперь можно формализовать заявленное ранее действие "подадим некому алгоритму на вход текст программы". Алгоритм будет получать элементы множества $\mathcal{A}^*$, то есть, соответствующее число Геделя (в то время как сам текст программы принадлежит $\mathcal{B}^*$).

Лемма 3.1 *Множество $\{\xi : \xi = \xi_P, P$ — программа над $\mathcal{A}^*\}$ разрешимо.*

3.1 Самоприменимые программы

Теперь перейдем к вопросу о проверке того, принадлежит ли программа некоторому классу.

Определение 3.2 *Множество самоприменимых программ – $\Pi'_{halt} := \{\xi_p : P : \xi_p \to halt\}$ – множество программ, корректно работающих с собственным кодом, то есть, которые когда-либо остановятся, имея свой текст (свое число Геделя) в качестве входных данных.*

Вопрос 1 *Является ли множество Π'_{halt} разрешимым?*

Теорема 3.1 *Множество Π'_{halt} программ корректно работающих с собственным кодом неразрешимо.*

Доказательство: Предположим обратное, то есть, пусть существует разрешающая его программа

$$P_0 : \begin{cases} \xi_P \to \square, & \xi_P \in \Pi'_{\text{halt}} \quad (P : \xi_P \to \text{halt}), \\ \xi_P \to \eta \neq \square, & \xi_P \notin \Pi'_{\text{halt}} \quad (P : \xi_P \to \infty); \end{cases}$$

которая ничего не выдает на печать в случае, если поданное ей на вход число Геделя соответствует программе, корректно работающей со своим кодом, и наоборот, выводит на печать непустое слово в противном случае.

Модифицируем эту программу P_0. Ее исходный текст выглядит следующим образом:

$$\begin{array}{ll} 0. & \ldots; \\ \ldots & \ldots \\ \alpha. & \text{PRINT;} \\ \ldots & \ldots \\ k. & \text{halt;} \end{array}$$

Вставим на k-ую строку оператор "k. IF $R_0 = \square$ THEN k ELSE k+1 OR k+1 OR ... OR k+1;" сдвинув вниз "(k+1). halt;" и заменим все строки вида "α. PRINT;" на "α. GOTO k;", где "GOTO k" осуществляет безусловный переход на k-ый оператор, то есть является сокращенной записью для инструкции "IF $R_0 = \square$ THEN k ELSE k OR k OR ... OR k". Назовем полученную программу P_1.

Программа P_1 ведет себя следующим образом: получая на вход число Геделя некой программы из Π'_{halt}, программа успешно доходит до строки k, где зацикливается (действительно, программа P_0 в этом случае ничего не выводила на печать, следовательно, инструкции вывода на печать не задействовались). В случае же, если входные данные соответствуют программе, не входящей в Π'_{halt}, мы достигнем строки k с непустым нулевым регистром (R_0) и будет совершен переход на к оператору $k+1$, то есть, успешное завершение P_1.

При такой схеме поведения программы мы получим противоречие, подав ей на вход ее собственное число Геделя. Действительно, подадим ей на вход собственное число Геделя, если оно из Π'_{halt}, то программа зациклится, а значит не является программой корректно работающей с собственным кодом. А если программа не является корректно работающей с собственным кодом, то и ее номер не принадлежит Π'_{halt}. Мы пришли к противоречию, следовательно, разрешающего алгоритма не существует. $\square$.

Отрицательный ответ на поставленный вопрос означает неразрешимость *массовой проблемы*, что не значит, что для некоторого, быть может, очень узкого класса программ нельзя построить проверяющий алгоритм.

3.2 Проблема останова

Рассмотрим следующую проблему. Положим, что у нас есть программа, которая "якобы вычисляет" значения функции $y = x^2$, и мы хотим это проверить. Проблема соответствия программ поставленным задачам – проблема верификации. Но мы можем упростить задачу, остановившись на проблеме более простого характера. Например, завершит ли программа работу при подаче на вход числа 2. И, даже, ещё упростить – завершится ли программа при подаче ей на вход пустого слова.

Определение 3.3 $\Pi_{halt} := \{\xi_p : P : \square \to halt\}$ – *множество программ (номеров Геделя), которые когда-нибудь остановятся (при пустых входных данных);*

Вопрос 2 *Является ли множество* Π_{halt} *разрешимым?*

Важность этого вопроса состоит в том, что если ответ на него положителен (и если реализующая найденный алгоритм программа работает обозримое время), то проверка программы на правильность работы при некоторых входных данных сводится к запуску *универсальной* проверяющей программы.

Теорема 3.2 *Множество* Π_{halt} *программ корректно работающих с пустым входным словом неразрешимо.*

Доказательство: Предположим существование аналогичной программы P_0 для нашего случая:

$$P_0 : \begin{cases} \xi_P \to \square, & \xi_P \in \Pi_{\text{halt}} \quad (P : \square \to \text{halt}), \\ \xi_P \to \eta \neq \square, & \xi_P \notin \Pi_{\text{halt}} \quad (P : \square \to \infty); \end{cases}$$

Теперь объясним как с ее помощью получить разрешающую программу для Π'_{halt}: мы хотим проверить, корректно ли программа ξ_1 работает с собственным номером Геделя. Рассмотрим программу ξ_2, полученную из ξ_1 сдвигом всех операторов исходной программы на n_ξ (ее номер Геделя) вниз, записав на освободившиеся n_ξ строк:

$$\begin{array}{l} 0.\ \text{LET } R_0 = R_0 + a_0; \\ 1.\ \text{LET } R_0 = R_0 + a_0; \\ 2.\ \text{LET } R_0 = R_0 + a_0; \\ \ldots; \\ (n_\xi - 1).\ \text{LET } R_0 = R_0 + a_0; \end{array}$$

Кроме того преобразуем все операторы $\alpha\ IF\ R_i = \square\ THEN\ \alpha'\ ELSE\ \alpha_0$ в $(\alpha + n_\xi)\ IF\ R_i = \square\ THEN\ (\alpha' + n_\xi)\ ELSE\ (\alpha_0 + n_\xi)$ Эта программа, получив на вход пустое слово будет содержать номер Геделя программы ξ_1 в регистре R_0, и далее будет следовать код программы ξ_1 (точнее, эквивалентный ему). В силу нашего предположения, множество Π_{halt} разрешимо, то есть мы можем сказать остановится программа ξ_2 или нет. Но эта программа остановится при подаче ей на вход пустого слова если и только если, программа ξ_1 остановится при работе с собственным номером Геделя. Таким образом, мы пришли к тому, что множество Π'_{halt} – разрешимо, что противоречит предыдущей теореме. □

И отрицательный ответ, как и в предыдущем случае, означает неразрешимость массовой проблемы. Верификация же отдельных программ является достаточно сложной, но паринципиально разрешимой задачей. В критических задачах (например управления ядерным реактором или космическим спутником) где цена ошибки высока, можно провести теоретическое доказательство соответствия поставленым условиям, что и является верификацией программы.

Упражнение 3.1 *Доказать:* Π_{halt} *и* Π'_{halt} *– перечислимые множества.*

Указание: *Поскольку перечислимость множества означает существование программы, которая при некоторых входных данных будет выдавать только элементы данного множества, причем рано или поздно выдаст их все, можно переформулировать*

условие данного упражнения: "предъявить перечисляющий алгоритм для данных множеств".

Общая идея построения алгоритмов как для Π_{halt} *так и для* Π'_{halt} *такова: сначала программы нужно модифицировать (вместо печати программа должна переходить к следующей инструкции, а вместо инструкции halt выводить свой номер Геделя) после этого запускаем программы параллельно, на каждой итерации выполняем по одной инструкции из всех, и первую инструкцию программы с номером Геделя равным номеру шага.*

3.3 Теорема Райса

Мы доказали, что существуют неперечислимые множества. Теперь же возникает вопрос, а что мы можем узнать про вычислимые и полувычислимые функции? Какие свойства вычислимых функций можно алгоритмически распознать по их программам? Оказывается, что никакие, кроме тривиальных. Доказательство этого факта содержится в теореме Райса.

Теорема 3.3 *Райса Пусть* C *– некое множество полувычислимых функций, причем* $C \neq \emptyset$ *и* $C \neq \{$*множество полувычислимых функций*$\}$*, тогда множество номеров Геделя программ, соответствующим этим функциям не является разрешимым.*

Доказательство: Занумеруем все все программы, соответствующие полувычислимым функциям. Получим некую нумерацию Геделя. Пусть M – множество номеров Геделя полувычислимых функций из C:

$$M := \{x : f_x \in C\}$$

Теперь рассмотрим характеристическую функцию:

$$1_M(x) := \begin{cases} 1, \in M (f_x \in C) \\ 0, \ x \notin M (f_x \notin C) \end{cases}$$

Эта характеристическая функция "говорит", является ли программа с данным номером полувычисляющей для одной из функций данного множества. Теперь рссмотрим некоторую эффективную полувычислимую функцию f_0: $dom(f_0) = \emptyset$. И возьмем произвольную $f_a \in C$. Рассмотрим такую функцию:

$$F(x,y) := \begin{cases} f_a, \ x \in dom(f_x(y)), \\ f_0, \text{ в противном случае} \end{cases}$$

Теперь фиксируем соответственно x. Получаем здесь функцию от y: $F(x,y)$. У этой функции от y будет некоторый номер Геделя. Этот номер Геделя мы обозначим $g(x)$. Для каждого фиксированного x он будет разный.

Ясно, что $g(x)$ как функция от x будет полувычислимой. Теперь рассматриваем $f_{g(x)}(y)$, то есть функцию, номер которой $g(x)$. В силу определения функции F будем иметь:

$$f_{g(x)}(y) = \begin{cases} f_a(y), & x \in dom(f_x), \\ f_0(y), & x \notin dom(f_x) \end{cases}$$

Таким образом, $f_{g(x)} \in C$, если и только если $f_x(x)$ определена.

Рассмотрим теперь такую функцию:

$$\chi_m(g(x)) = \begin{cases} 1, & x \in dom(f_x), \\ 0, & x \notin dom(f_x) \end{cases}$$

Получается, что если предполагаем, что множество M разрешимо, то множество $\{x\}$, таких, что $f_x(x)$ определена, является разрешимым. То есть тем самым оказывается разрешимой и задаяа самоприменимости для регистровых машин. Что противоречит теореме о неразрешимости задачи самоприменимости для регистровых машин. Следовательно никакое нетривиальное множество полувычислимых функций разрешимым не является. □

Теорему Райса иногда формулируют так: никакое нетривиальное свойство (подмножество) множества полувычислимых функций не является разрешимым.

Определение 3.4 *Пусть Q некоторое свойство полувычислимых функций. Свойство Q назовем нетривиальным, если существуют как функции, обладающие свойством Q, так и функции не обладающие этим свойством.*

Все обычно рассматриваемые свойства являются нетривиальными. Примерами нетривиальных свойств служат следующие:

- функция тождественно равна нулю;
- функция нигде не определена;
- функция всюду определена;
- функция взаимно однозначна;
- функция определена при x = 0.

Из теоремы Райса следует несколько утверждений о программах, вычисляющих значния функций. Под функцией f_x, мы будем понимать функцию, которая соответствует вычисляющей (полувычисляющей) программе с номером Геделя x.

Предложение 3.1

- *нельзя с помощью универсального алгоритма проверить всюду ли определена функция f_x;*
- *не существует алгоритма проверки того, будет ли программа вычислять нулевую функцию;*
- *вопрос о том, вычисляют ли две программы одну и ту же функцию, массово неразрешим;*
- *не существует алгоритма, определяющего по тексту программы, будет ли эта программа вычислять некоторую конкретную вычислимую функцию.*

Из теоремы Райса (и следствий из нее) очевидна неразрешимость многих задач, связанных с программированием. Например, если имеется некоторая программа, то по ней, вообще говоря, ничего нельзя сказать о функции, реализуемой программой. По двум программам нельзя установить, реализуют ли они одну и ту же функцию, а это приводит к неразрешимости многих задач, связанных с эквивалентными преобразованиями и минимизацией программ. В любом алгоритмическом языке, какие бы правила синтаксиса там ни применялись, всегда будут иметься "бессмысленные"программы, задающие функции не определенные ни в одной из точек (“эти программы нельзя обнаружить, потому что они никогда не работают, не являясь при этом ошибочными”). Теорема Райса доказывает алгоритмическую неразрешимость многих задач, связанных с вычислениями на компьютерах.

4 Разрешимость и перечислимость множества тавтологий

Мы уже упоминали о том, что эффективные вычисления реализуются как вычислительными машинами, так и человеком. Только что мы установили некоторые факты о языках программирования, теперь мы попробуем применить их к языкам логики.

Возьмем язык логики первого порядка L^σ с сигнатурой σ, множество формул $\Gamma \in L^\sigma$ и конкретную формулу $P \in L^\sigma$. Требуется проверить, является ли формула P семантическим следствием из Γ. Очевидно, что мы не можем напрямую проверить принадлежность формулы P всем возможным моделям множества Γ. Следовательно, нужно попробовать синтаксически вывести эту формулу, то есть, доказать ее истинность, и (по теореме о корректности) получить также и семантическую справедливость.

Здесь возникает ряд вопросов: можно ли построить такое доказательство? Можно ли написать алгоритм, который будет последовательно выдавать следствия из (пусть конечного) множества Γ, то есть, являются ли следствия из него перечислимым множеством? Существует ли алгоритм, определяющий, является ли P следствием из Γ, то есть, разрешимо ли это множество? (При отсутствии данного алгоритма процесс доказательства некой формулы является “шаманством”.) Можно также решать более простую

задачу, то есть определять является ли формула тавтологией (проверять свойства $\models \mathcal{P}$ или $\vdash_{ND} \mathcal{P}$.

Можно рассматривать "самый богатый" язык (далее мы поймем, какие требования не принципиальны). "Самый богатый" язык содержит счетное число констант, функциональных и предикатных симмволов, а также символ равенства. Назовем его $L^{\sigma\infty}$. В качестве Γ рассмотрим пустое множество формул и будем работать с множеством тавтологий $\mathcal{P} \in L^{\sigma}$, справедливых в любой модели. Является ли хотя бы оно разрешимым или по крайней мере перечислимым?

Теорема 4.1 *Множество тавтологий "достаточно богатого" языка логики первого порядка $L^{\sigma\infty}$ не является разрешимым.*

Замечание 4.1 *Уточним утверждение теоремы. Множество тавтологий языка логики первого порядка является разрешимым только для весьма бедных языков, не сильно отличающихся от логики высказываний (для которых допускается наличие только унарных предикатных символов). Для логики высказываний проверка истинности осуществляется путем построения таблицы истинности.*

Для дальнейших рассуждений нам существенно понадобится теория регистровых машин. Мы попытаемся "связать" регистровые машины и логические формулы между собой.

Определение 4.1 *Вектор из чисел $(L, m_0, m_1, \ldots, m_n)$ является конфигурацией регистровой машины, исполняющей программу P, после l шагов, если*

- *программа оканчивается меткой k и $L \leq k$;*
- *начавшись с пустого слова, после l шагов имеет в регистре R_i число элементов, равное m_i для всех допустимых значений i;*
- *следующая инструкция имеет метку L.*

С учетом этого определения условие конечности программы P с меткой k у команды halt можно переформулировать: найдутся такие числа $L, m_0, m_1, \ldots, m_n$, что после l шагов регистровая машина будет иметь конфигурацию $(k, m_0, m_1, \ldots, m_n)$.

Доказательство: Опишем на языке логики первого порядка работу регистровой машины (именно для этой цели и была выбрана такая большая мощность языка).

Определим круг рассматриваемых программ. Их алфавит состоит из элемента $|$, слова интерпретируются как числа (по количеству элементов в слове). Требуется построить по программе формулу φ_P, которая является тавтологией, если и только если данная программа корректно работает с пустым начальным словом ($\models \varphi_P$, IFF $\xi_P \in \Pi_{\text{halt}}$). Пусть в регистровой машине, в которой работает эта программа, задействован $n+1$ регистр $R_0, R_1, R_2, \ldots, R_n$.

Для построения формулы φ_P следует выбрать необходимые предикатные символы. Пусть программа рано или поздно остановится ($P : \square \to \text{halt}$), причем s_P—количество шагов до остановки (для бесконечных программ будем считать $s_P = \infty$). Из σ_∞ выберем: символ унарной функции s, бинарный предикатный символ '$<$', $(n+3)$-арный символ предиката R, символ константы '0' и предикат равенства '$=$'.

Построим специальную интерпретацию для удобной работы с формулами такого вида.

Случай 1. P некорректно работает с пустым начальным словом, $P : \square \to \infty$. В качестве универсума возьмем множество $A_P := \mathbf{N} \cup \{0\}$, символ ' $<$ ' будет соответствовать обычному отношению порядка на $\mathbf{N}$, $0^{\mathcal{A}_P} := 0$, ' $=$ ' интерпретируется как равенство, $s^{\mathcal{A}_P}(x) := x + 1$, $R^{\mathcal{A}_P}(l, L, m_0, m_1, \ldots, m_n)$ (где все аргументы— числа)— характеристическая функция некоторого отношения:

$$\begin{cases} 1, & \text{если после } l \text{ шагов регистровая машина} \\ & \text{имеет конфигурацию } (L, m_o, \ldots, m_r); \\ 0, & \text{в противном случае .} \end{cases}$$

Случай 2. P корректно работает с пустым начальным словом, $P : \square \to \text{halt}$. Универсумом будет являться некоторый конечный отрезок ряда целых чисел $A_P := \{0, 1, 2, \ldots, e\}$, где $e := \max(k, s_P)$—максимум от длины программы и количества шагов до останова. Функцию последователя определим следующим образом:

$$s^{A_P}(x) := \begin{cases} x + 1, & x \neq e, \\ e, & x = e; \end{cases}$$

остальные символы—аналогично первому случаю.

Обозначим $\underline{1} := s(0)$, $\underline{n} := s(\underline{n-1})$. Подчеркивание означает принадлежность данных символов к термам языка, а не числам.

Построим формулу ψ_P, которая описывает работу программы P, соблюдая при этом два свойства: $\mathcal{A}_P \models \psi_P$ (она истинна в интерпретации $\mathcal{A}_P$) и если ψ_P верно в некоторой другой модели $\mathcal{A}$ и $(L, m_0, \ldots, m_n)$—конфигурация регистровой машины после l шагов, то элементы 0^A, $\underline{1}^A$, $\ldots$, $\underline{l}^A$ попарно различны и в A истинна формула $R\left(\underline{l}, \underline{L}, \underline{m_0}, \ldots, \underline{m_n}\right)$. Данные свойства проверяются в процессе построения формулы ψ_P (первое—подстановкой, второе—индукцией по l).

Запишем формулу для ψ_P следующим образом: $\psi_P := \psi_0 \wedge R\left(\underbrace{0, 0, \ldots, 0}_{n+3}\right) \wedge \psi_{\alpha_0} \wedge \psi_{\alpha_1} \wedge \ldots \wedge \psi_{\alpha_{k-1}}$, где ψ_0—некоторая формула, диктующая правила обработки символов '$<$', $s\,()$ и т. д. в данной модели (

$$\psi_p = \psi_{\alpha_0} \wedge \psi_{\alpha_1} \wedge \ldots \wedge \psi_{\alpha_{k-1}} \wedge \forall x \forall y \forall z (x < y \wedge y < z \to x < z) \wedge$$

$$R(0, 0, \ldots, 0) \wedge \forall x (0 < x \vee 0 = x) \wedge \forall x (x < S(x) \vee x = S(x)) \wedge$$

$$\forall x (\exists y (x < y) \to (x < S(x) \wedge \forall z (x < z \to (S(x) < z \vee S(x) = z))))$$

или, говоря неформальным языком

$$\psi_0 := \underline{'<\text{—порядок}'} \wedge \forall x\,(0 < x \vee 0 = x) \wedge \forall x \left(\underline{x \leq s\,(x)}\right) \wedge$$
$$\underline{'s\,(x)\text{—единственный следующий за } x, \text{ если } x\text{—не последний}'}$$

);

$R\,(0, 0, \ldots, 0)$— состояние машины перед выполнением программы; формулы ψ_{α_i} отвечают состоянию, в которое переходит машина при выполнении соответствующей инструкции α_i.

Определим ψ_{α_i} в зависимости от типа инструкции α_i.

1. α_i. LET $R_j = R_j + |$;

$$\psi_{\alpha_i} := \forall x \forall y_0 \ldots \forall y_n$$
$$(R\,(x, \underline{L}, y_0, \ldots, y_n) \rightarrow$$
$$(x < s\,(x) \wedge R\,(s\,(x)\,, \underline{L+1}, y_0, \ldots, y_{j-1}, y_j + 1, y_{j+1}, \ldots, y_n)))$$

2. α_i. LET $R_j = R_j - |$;

$$\psi_{\alpha_i} := \forall x \forall y_0 \ldots \forall y_n$$
$$((R(x, \underline{L}, y_0, \ldots, y_n) \rightarrow (x < s(x) \wedge$$
$$(\neg y_j = 0 \wedge R(s(x), \underline{L+1}, y_0, \ldots, y_{j-1}, y_j - 1, y_{j+1}, \ldots, y_n) \vee$$
$$(y_j = 0 \wedge R(s(x), \underline{L+1}, y_0, \ldots, y_{j-1}, 0, y_{j+1}, \ldots, y_n))))$$

3. α_i. PRINT;

$$\psi_{\alpha_i} := \forall x \forall y_0 \ldots \forall y_n$$
$$(R\,(x, \underline{L}, y_0, \ldots, y_n) \rightarrow$$
$$(x < s\,(x) \wedge R\,(s\,(x)\,, \underline{L+1}, y_0, \ldots, y_n)))$$

4. α_i. IF $R_j = \square$ THEN L' ELSE L_0

$$\psi_{\alpha_i} := \forall x \forall y_0 \ldots \forall y_n$$
$$((R(x, \underline{L}, y_0, \ldots, y_n) \rightarrow$$
$$(x < s(x) \wedge (y_j = 0 \wedge R(s(x), \underline{L_0}, y_0, \ldots, y_{j-1}, 0, y_{j+1}, \ldots, y_n) \vee$$
$$(\neg y_j = 0 \wedge R(s(x), \underline{L_0}, y_0, \ldots, y_n))))$$

Теперь построим формулу φ_P, являющуюся тавтологией в случае, если P корректно работает с пустым начальным словом:

$$\varphi_P := \psi_P \rightarrow \exists x \exists y_0 \exists y_1 \ldots \exists y_n\,(R\,(x, \underline{k}, y_0, y_1, \ldots, y_n))$$

Докажем, что, действительно, $\models \varphi_P \leftrightarrow P \in \Pi_{\text{halt}}$.

- ' $\rightarrow$ '. φ_P—тавтология ($\models \varphi_P$), следовательно, она истинна в любой интерпретации, в том числе и в нашей, где истинна ψ_P. Из одновременной истинности φ_P и ψ_P получаем, что истинна и $\exists x \exists y_0 \exists y_1 \ldots \exists y_n \left(R\left(x, \underline{k}, y_0, y_1, \ldots, y_n\right)\right)$, то есть, программа P остановится после k шагов и таким образом корректно работает с пустым начальным словом.
- ' $\leftarrow$ '. $\models \psi_P \rightarrow A \models \varphi_P$ (по определению, из лжи выводится что угодно, например, истина). И, тогда $A \models \varphi_P \rightarrow \mathcal{A} \models R\left(\underline{l}, \underline{L}, \underline{m_0}, \underline{m_1}, \ldots, \underline{m_n}\right) \rightarrow \mathcal{A} \models \varphi_P$

Мы правильно описали работу любой регистровой машины и построили формулу, которая является тавтологией, если и только если регистровая машина, работу которой описывает эта формула, корректно работает с пустым словом. Пусть множество тавтологий разрешимо, следовательно, существует разрешающий алгоритм, который при этом после некоторого преобразования решает проблему останова (например, алгоритм, который получает на вход число Геделя, строит по нему формулу, описывающую работу машины и проверяет ее принадлежность к множеству тавтологий, после чего дает ответ о принадлежности программы множеству Π_{halt}). Но, в соответствии с ранее доказанным, существование такого алгоритма невозможно, следовательно, множество тавтологий неразрешимо. $\square$

Упражнение 4.1 *Множество тавтологий языка логики первого порядка перечислимо.*

Указание: *Мы уже умеем отождествлять тавтологии с программами, корректно работающими с пустым словом. Перечеслимость же множества таких программ уже обсуждалась.*

Замечание 4.2 *Если на языке можно записать аналог арифметики Пеано, для него справедливы все вышеприведенные выкладки для языка с бесконечной сигнатурой.*

Теорема 4.2 *Множество тавтологий языка логики второго порядка неперечислимо.*

Этот факт гораздо более сильный чем то, что нельзя построить одновременно корректную и полную формальную систему. Этот результат очевиден и из нашего результата (а именно,если бы существовала корректная и полная формальная система, то доказательство перечислимости множества тавтологий языков логики первого порядка переносилось бы и на языки логики второго порядка). Более того, верно не только наше утверждение о том, что нельзя построить корректную и полную формальную систему, но нельзя построить формальную систему корректную и полную лишь в слабом смысле (напомним, что ситема является корректной и полной в слабом смысле, если $\vdash \mathcal{P}$ тогда и только тогда, когда $\models_\Phi \mathcal{P}$, где $P \in L^\Sigma$).

Мы показали, что для достаточно богатых языков никакие семантические вопросы о формулах их не допускают эффективного решения (нет алгоритма, который давал бы ответ на эти вопросы). Это верно даже для языков логики первого порядка, не говоря уже о языках логики второго порядка. Тем не менее если язык оказывается достоточно беден, то для него можно проверить некоторые семантические факты.

5 Формальные теории

Мы уже говорили, что если на языке можно записать анаалог арифметики Пеано, то для него справедливо всё, что утверждалось о достаточно богатых языках. Теперь мы зададимся более интересным вопросом – вопросом создания системы аксиом некоторой теории. Кроме того мы можем, например интересоваться вопросом, можно ли перечислить все формулы данной теории (что автоматически означает, что формулы этой теории доказываются автоматически – путем их перечисления). Также любопытно, а можно ли по данной формуле сказать, истинна ли она в данной теории.

Данные вопросы отнюдь не являются праздными. К началу XX в. в теории множеств были обнаружены парадоксы – самые настоящие противоречия. К этому времени теория множеств уже успела показать себя как естественная основа и плодотворнейшее орудие математики. Для спасения ее немецкий математик Дэвид Гильберт предложил в 1904 г. свою программу перестройки оснований математики, которая состояла из двух частей:

- Представить существующую математику (включая очищенный от парадоксов вариант теории множеств) в виде формальной теории.
- Доказать непротиворечивость полученной теории (т.е. доказать, что в этой теории никакое утверждение не может быть доказано вместе со своим отрицанием).

Для дальнейшего обсуждения данной программы нам понадобится определение теории.

Определение 5.1 *Множество $T \subset L^\sigma$ называется теорией, если оно выполнимо и замкнуто относительно семантического следствия (то есть если $T \models \mathcal{P}$, то $\mathcal{P} \in T$).*

Пример 5.1 *В качестве примера формальной теории можно рассматривать игру в шахматы - назовем это теорией C. Утверждениями в C будем считать позиции (всевозможные расположения фигур на доске вместе с указанием "ход белых" или "ход черных", а также дополнительной информацией о том, делали ли стороны ходы королем и каждой из ладей). Тогда аксиомой теории C естественно считать начальную позицию, а правилами вывода - правила игры, которые определяют, какие ходы допустимы в каждой позиции. Правила позволяют получать из одних утверждений другие. В частности, отправляясь от нашей единственной аксиомы, мы можем получать теоремы C. Общая характеристика теорем C состоит, очевидно, в том, что это - всевозможные позиции, которые могут получиться, если передвигать фигуры, соблюдая правила.*

Если нам предложена теорема A в теории C, вместе с ее доказательством, то мы можем проверить истинность его (просто проверив, все ли переходы соответствуют правилам)

Более того, множество всех теорем теории C оказывается конечным. Значит, принципиально, можно проверить является ли данная теорема доказуемой в C (то

есть достижима ли данная позиция). Можно также построить цепочку доказательства любой теоремы (последовательность позиций, приводящую к позиции A), поскольку количество всех цепочек также конечно.

Напомним, что мы говорим лишь о принципиальной разрешимости данных задач, не касаясь проблемы трудоемкости их решения.

Отвлекаясь от нашего, не вполне серьезного, примера, заметим, что существует два способа построения теорий:

1. Если $\mathcal{M}$ – алгебраическая система, то $Th(\mathcal{M}) := \{\mathcal{P} \in L^\sigma : \ \mathcal{M} \models \mathcal{P}\}$.
2. Задание теории при помощи системы аксиом. Пусть $\Gamma \in L^\sigma$. Тогда $\Gamma \models T$, где T – теория.

В первом случае мы имеем некую интерпретацию, универсум и рассматриваем теорию, как все предложения заранее выбранного языка L^σ, верные в этом универсуме, то есть все теоремы, верные для данной интерпретации. Возникает вопрос, можно ли каким-то образом конструктивно сгенерировать все такие теоремы. Поэтому обычно идут другим путем. Существует такой хороший механизм, а именно одновременно корректная и полная формальная система. Их на самом деле много, но мы рассмотрели одну – это естественная дедукция. Далее создают некую систему аксиом Γ, обычно не произвольную, а конечную систему аксиом или "обозримую" систему аксиом. Иными словами систему аксиом, состоящую из отдельных аксиом и нескольких схем аксиом. Система аксиом получается бесконечной, но обозримой в том смысле, что она регистрово разрешима (есть такой алгоритм, который для любой формулы может сказать, является ли она аксиомой или не является). Разрешимость здесь как бы исключает произвольность этого множества, оно должно быть в каком-то смысле конструктивным.

Далее берется такое множество и строятся все семантические следствия. А так как естественная дедукция является корректной и полной, можно утверждать, что все семантические следствия – это и формальные следствия. Далее запускается аппарат, который позволяет выводить из формул Γ с использованием правил вывода другие теоремы и знаем гарантированно, что все то, что мы выводим – это теоремы нашей теории.

Остается неразрешенным вопрос, а в каком соотношении находится теория, построенная на множестве аксиом Γ и теории исходной модели. Если Γ подобрали удачно, в том смысле, что все формулы верны в модели $\mathcal{M}$, то ясно, что теория, построенная на Γ будет подмножеством теории модели. А будет ли совпадение? Чуть позже выясним этот вопрос конкретно для арифметики.

Рассмотрим некоторую теорию. В нашем примере про шахматы все формулы теории получались из одной аксиомы. Вообще, если можно предъявить множество аксиом, из которых выводятся все формулы тории, то теория называется *эффективно аксиоматизируемой*. Или, говоря более формальным языком:

Определение 5.2 *Теория T называется эффективно аксиоматизируемой, если существует такое разрешимое множество* $\Gamma : \quad T = \Gamma^{\models}$.

Такое множество Γ, не всегда конечно, если же оно все же оказалось конечным, то такая теория называется *конечно аксиоматизируемой.*

Определение 5.3 *Теория T называется конечно аксиоматизируемой, если существует такое конечное* $\Gamma: \quad T = \Gamma^{\models}$.

Еще нам бы хотелось, чтобы теория не содержала формул, о которых "ничего нельзя сказать", то есть, чтобы была верна либо формула, либо ее отрицание.

Определение 5.4 *Теория T называется полной, если для любого $\mathcal{P} \in L^\sigma$ выполняется либо $\mathcal{P} \in T$ либо $\neg\mathcal{P} \in T$.*

Определение 5.5 *Класс K алгебраических систем называется аксиоматизируемым, если существует сигнатура σ и такое множество предложений Z сигнатуры σ, что для любой системы ξ*

$$\xi \in K \;\; \textit{Если и только если} \;\; (\; \textit{сигнатура } \xi \textit{ равна } \sigma \textit{ и } \xi \vdash \Phi \textit{ для всех } \Phi \in Z)$$

Замечание 5.1 *Любая теория модели полна.*

Другими словами, в теории модели, для любой формулы верно, что либо она сама принадлежит этой теории, либо ее отрицание принадлежит этой теории.

Теперь, в качестве упражднений, вы можете доказать еще ряд интересных и полезных утверждений.

Замечание 5.2 *Если эффективно аксиоматизируемая теория T полна, то она разрешима.*

Упражнение 5.1 *Если теория T суть перечислимое множество и она полна, то она разрешима.*

Упражнение 5.2 *Если $T = \Gamma^{\models}$, где Γ перечислимо, то сама теория эффективно аксиоматизируема.*

Лемма 5.1 *Для любой программы P на регистровой машине с регистрами $R_0, \ldots, R_n$ найдется фраза $\psi_p(\upsilon_1, \ldots, \upsilon_{2n+3}) \in L^\sigma$ (L^σ – достаточно богатый язык), такая что для любой конфигурации $k_0, \ldots, k_n, L, m_0, \ldots, m_n)$ $\mathbf{N} \vdash \psi(\underline{k_0}, \ldots, \underline{k_n}, \underline{L}, \underline{m_0}, \ldots, \underline{m_n})$ если и только если программа начав с $(0, k_0, \ldots, k_n)$ приходит за конечное время к $(L, m_0, \ldots, m_n)$*

Теорема 5.1 *Th($\mathbf{N}$) – регистрово неразрешима.*

Доказательство: Докажем что Th(**N**) не является регистрово разрешимой.

Воспользуемся регистровой неразрешимостью Π_{halt}. Пусть задана программа P. По лемма найдем для нее φ_p.

$$\varphi_p : = \exists \upsilon_{n+3}, \ldots, \exists \upsilon_{2n+3} \psi_p(\underbrace{\underline{0}, \ldots, \underline{0}}_{n+1}, \underline{k}, \underline{\upsilon}_{n+3}, \ldots, \underline{\upsilon}_{2n+3})$$

где $\underline{k}$ – метка halt. Получаем соответствие: $P \to \varphi_p$, где $\mathbf{N} \models \varphi_p$ или, что тоже, $P : \square \to halt$.

Если Th(**N**) – регистрово разрешимое множество то и Π_{halt} регистрово разрешимо, а это не так. $\square$

Следствие 5.1 *Th(***N***) не является перечислимой и не является эффективно аксиоматизируемой.*

Доказательство: Th(**N**) – полна. Значит, с одной стороны, если она перечислима, то она и разрешима. С другой стороны, если она эффективно аксиоматизируема, то также является разрешиомой. $\square$

Можно даже сказать, что Th(**N**) неразрешима, и добавляя в неё формулу (формулы) мы не добьемся её полноты. Кроме того существует теорема теориии чисел о которой ничего нельзя сказать пользуясь PA, вследствие её неполноты.

То есть наша теория (Th(**N**)) оказалась несовершенной. А несовершенную теорию следует усовершенствовать. Может быть, мы "забыли" какие-то важные аксиомы? Следует найти их, присоединить к аксиомам Th(**N**), и в результате мы получим совершенную систему? К сожалению наш результат распространяется и на любые расширения теории. Если мы найдем эту "недостающую" теорему и добавим её в Th(**N**) в качестве аксиомы, то Th(**N**) останется неполна. Таким образом $PA^{\vDash}$ не полна, а $Th(\mathbf{N})$ неразрешима, и даже неперечислима. То есть, хотя мы можем перечислить все следствия из аксиом, но мы не можем даже перечислить все теоремы теории чисел.

Заметим, что мы хотя и доказали неполноту $PA^{\vDash}$, но мы не предъявили ту формулу, для которой ни она, ни её отрицание не выводятся из системы аксиом. Мы собираемся сформулировать теорему Геделя в форме независимой от арифметического языка, и, кроме того, предъявим конструкивный алгоритм построения данной формулы. Для того, чтобы перейти к этому нам понадобятся некоторые определения.

Определение 5.6 *Будем говорить, что $Q \subset \mathbf{N}^r$, является выразимым в $\Phi \subset L^{\sigma_{ar}}$, если существует формула $\varphi_Q(\upsilon_0, \upsilon_1, \ldots, \upsilon_{r-1})$, такая что*

1. *если $(n_0, n_1, \ldots, n_{r-1}) \in Q$, то $\Phi \models \varphi_Q(\underline{(n_0)}, \underline{(n_1)}, \ldots, \underline{(n_{r-1})})$*
2. *если $(n_0, n_1, \ldots, n_{r-1}) \notin Q$, то $\Phi \models \neg\varphi_Q(\underline{(n_0)}, \underline{(n_1)}, \ldots, \underline{(n_{r-1})})$*

Если Φ несовместно, то в нем выразимо любое $Q \subset \mathbf{N}$.

Определение 5.7 *Функция* $F : \mathbf{N}^r \to \mathbf{N}$ *называется выразимой в* $\Phi \subset L^{\sigma_{ar}}$, *если существует формула* $\varphi_Q(\upsilon_0, \upsilon_1, \ldots, \upsilon_{r-1})$, *такая что*

1. *если* $F(n_0, n_1, \ldots, n_{r-1}) = n_r$, *то* $\Phi \models \varphi_F(n_0, n_1, \ldots, n_r)$
2. *если* $F(n_0, n_1, \ldots, n_{r-1}) \neq n_r$, *то* $\Phi \models \neg\varphi_F(n_0, n_1, \ldots, n_r)$
3. $\Phi \models \upsilon_r(\varphi_F(n_0, n_1, \ldots, n_{r-1}, \upsilon_r))$

Функции выразимые в Φ выразимы в $\Phi' \subset \Phi$. Если $\Phi \vdash_{ND} \bot$ то всё выразимо в Φ.

Лемма 5.2 *Если* Φ *– все теоремы* $Th(\mathbf{N})$, *то в нем выразимы все разрешимые множества и вычислимые функции.*

Лемма 5.3 *Предыдущая лемма верна и в* $PA^{\vdash}$.

Запишем все формулы арифметики. Каждой формуле будем соотносить некоторый номер – число Геделя формулы (например номер формулы при записи в лексикографическом порядке). n_φ – число Геделя для $\varphi \in L^{\sigma_{ar}}$.

Замечание 5.3 *Если* $\Phi \nvdash_{ND} \bot$, *и, кроме того* Φ *разрешимо, то любое* Q *выразимое в* Φ *является разрешимым и любая* F *выразимая в* Phi *– вычислима.*

Обозначим универсально выражающие множества как $Repr\Phi$.

Теорема 5.2 $ReprTh(\mathbf{N})$ *и* $ReprPA^{\vdash}$.

Доказательство: Приведем лишь идею доказательства. Нужно по любому Q построить разрешающую формулу. Разрешающую программу нужно описать формулой. Тогда у нас будет формула, разрешающая множество (выразительную силу языка считаем достаточной)

5.1 Теорема о неподвижной точке

Теорема 5.3 *О неподвижной точке*

Пусть задано $\Phi \subset L^{\sigma_{ar}}$, *в котором выразимо любое разрешимое множество и любая выполнимая функция. Тогда для любой* $\psi \in L^{\sigma_{ar}}$, $\#free(\psi) = 1$ *существует* φ, *такое что* $\Phi \models \varphi \leftrightarrow \psi(\underline{n}_\varphi)$.

Доказательство: Определим функцию $F : \mathbf{N}x\mathbf{N} \to \mathbf{N}$.

Если $n = n_\chi$, где χ – формула с одной свободной переменнрой, то $F(n, m) := n_{\chi(\underline{(m)})}$. В противном случае $F(n, m) := 0$.

F является вычислимой, а следовательно и выразимой, тогда существует выражающая ее формула $\alpha(\upsilon_0, \upsilon_1, \upsilon_3)$.

Для заданой ψ с одной свободной переменной построим формулы $\beta := \forall z(\alpha(x, x, z) \to \psi(z))$ и $\varphi := (\alpha(\underline{n}_\beta, \underline{n}_\beta, z) \to \psi(z))$, т. е. $\varphi = \beta[\underline{n}_\beta / x]$.

Докажем, что $\Phi \models \varphi \leftrightarrow \psi(\underline{n}_\varphi)$, или $\Phi \vdash \varphi \leftrightarrow \psi(\underline{n}_\varphi)$, или $\Phi \vdash \varphi \to \psi(\underline{n}_\varphi)$, или $\Phi \vdash \varphi \to \varphi$.

1. $\varphi \vdash \alpha(n_\beta, n_\beta, n_\varphi) \to \psi(n_\varphi)$.

   Если $F(n_\beta, n_\beta) = n_\varphi$, то $\Phi \vdash \alpha(\underline{n}_\beta, \underline{n}_\beta, \underline{n}_\varphi)$. Верно, что $\varphi \vdash \alpha(\underline{n}_\beta, \underline{n}_\beta, \underline{n}_\varphi) \to \psi(\underline{n}_\varphi)$ и $\Phi \vdash \alpha((\underline{n}_\beta, \underline{n}_\beta, \underline{n}_\varphi)$.

   Отсюда $\Phi, \varphi \vdash \psi(n_\varphi)$ или $\Phi \vdash \varphi \to \psi(n_\varphi)$.

2. $F(n_\beta, n_\beta) = n_{\beta(\underline{n}_\beta)} = n_\varphi$

$$\begin{cases} \Phi \vdash \exists! z \alpha(\underline{n}_\beta, \underline{n}_\beta, z) \\ \Phi \vdash \alpha(\underline{n}_\beta, \underline{n}_\beta, \underline{n}_\varphi) \end{cases}$$

   $Phi \vdash \forall z(\alpha(n_\beta, n_\beta, z) \to z = n_\varphi)$

   $Phi \vdash \psi(n_\varphi) \to \forall z(\alpha(\underline{n}_\beta, \underline{n}_\beta, z) \to z = \underline{n}_\varphi)$

   $Phi \vdash \psi(\underline{n}_\varphi) \to \underbrace{\forall z(\alpha(n_\beta, n_\beta, z) \to \psi(z))}_{\varphi}$

Таким образом $\Phi \models \psi(n_\varphi) \to \varphi$ $\square$

Сформулируем данную теорему для программ.

Теорема 5.4 *Для $f : \mathbf{N} \to \mathbf{N}$ – отображения между номерами Геделя программ найдется программа α такая что $n_\alpha = n_{f(n_\alpha)}$.*

Данная теорема позволяет, в частности, утверждать, что для любого языка программирования существует программа, печатающая свой текст.

5.2 Теорема Тарского

Определение 5.8 *Пусть Φ – система аксиом $L^{\sigma_{ar}}$. Тогда $\Phi(\Phi^\vdash)$ – множество следствий из Φ. Будем говорить, что оно выразимо в Φ, если множество соответствующих номеров Геделя выразимо в Φ.*

Теорема 5.5 *Пусть Φ совместное и в нем выразимы все разрешимые множества векторов и выполнимые функции на $\mathbf{N}$. Если $\Phi^\models$ выразимо в Φ, то существует $\varphi \in L^{\sigma_{ar}}$, такая что $\varphi \notin \Phi^\models$, $\neg\varphi \notin \Phi^\models$.*

Доказательство: $\Phi^\models$ выразимо в Φ, следовательно выразимо множество номеров Геделя, т. е. натуральных чисел. Тогда $\chi(r_0)$ выражает свойство быть исчислимым, т. е. для $\alpha \in L^{\sigma_{ar}}$ $\Phi \models \alpha$ и $\Phi \vdash \alpha$ равносильно $\Phi \models \chi(n_\alpha)$ и $\Phi \vdash \chi(n_\alpha)$ соответственно.

Рассмотрим $\psi := \neg\chi$. По теореме о неподвижной точке можем построить φ, такую что $\Phi \vdash \varphi \leftrightarrow \psi(\underline{n}_\varphi)$, т. е. $\Phi \vdash \varphi \leftrightarrow \neg\chi(\underline{n}_\varphi)$.

Докажем, что $\Phi \not\vdash \varphi$ ($\varphi \in \Phi^\vdash$). Пусть $\Phi \vdash \varphi$, тогда $\Phi \vdash \neg\chi(\underline{n}_\varphi)$. Отсюда (по определению χ) $\Phi \vdash \neg\varphi$, но это противоречит совместности Φ.

Докажем, что $\Phi \not\vdash \neg\varphi$ ($\neg\varphi \in \Phi^\vdash$). Пусть $\Phi \vdash \neg\varphi$, тогда $\Phi \vdash \chi(\underline{n}_\varphi$. Отсюда (по определению χ) $\Phi \vdash \varphi$, а это противоречит совместности Φ.

Таким образом $\varphi \notin \Phi^\vdash$ и $\neg\varphi \in \Phi^\vdash$. $\square$

Теорема 5.6 *Тарского о невыразимости истины.*

Пусть в $L^{\sigma_{ar}} \subset L^{\sigma}$ Φ *– достаточно богатое множество формул (выразимы все разрешимые множества векторов над* $\mathbf{N}$ *и все вычислимые формулы на* $\mathbf{N}$*) на языке содержащем язык арифметики* Φ *– совместно,* $\Phi^{\models}$ *– выразимо в* Φ *и* $\Phi^{\models}$ *не является полной.*

Доказательство: Докажем от противного. Если теория модели выразима, достаточно богата, совместна, тогда она бы не была полной, но теория модели всегда полна. □

Тогда теория стандартной модели арифметики не выразима в себе самой.

Следствие 5.1 *Существует абсолютно невычислимая функция.*

Доказательство:

Определим $f : \mathbf{N} \to \{0, 1\}$ следующим образом:

$f(n) := 0$, если n – номер Геделя теоремы теории чисел и $f(n) = 1$, если n – не номер Геделя.

$$f(n) = \begin{cases} 0, \ n = n_{\xi}, & \forall \xi \in Th(\mathbf{N}) \\ 1, \ n = n_{\xi}, & \forall \xi \notin Th(\mathbf{N}) \end{cases}$$

Приведенный пример доказывает существование абсолютно невычислимой функции. □

Заметим, что в достаточно богатых языках логики первого порядка можно записывать утверждения о совмесности теорий формулами языка, но эти утверждения "почти" никогда не выводятся из теории. Почти, означает, что данный язык должен содержать арифметику.

Теорема 5.7 *(Геделя о неполноте). Пусть* $L\sigma$ *содержит арифметику Пеано. Пусть* $\Phi \subset L\sigma$ *совместное, разрешимое, достаточно богатое множество формул. Тогда можно конкретно выписать предложение этого языка* φ*, для которого* $\Phi \nvdash \varphi$, $\Phi \nvdash \neg\varphi$.

Доказательство: Пусть это не так. Тогда $Th(\mathbf{N})$ полна так как она регистрово аксиоматезируема (потому что Φ – регистрово разрешима), а так как все регистрово-разрешимые множества выразимы в Φ), то и $\Phi^{\models}$ выразима в Φ. Следовательно теория выразима. □

5.3 Вторая теорема Геделя

В этой части положим $\sigma_{ar} \subset \sigma$, если не оговорено обратное.

Фиксируем теорию в достаточно богатом языке $\sigma_{ar} \subset \sigma$, $\Phi \subset L^{\sigma}$ – система аксиом. Φ – регистрово разрешима, тогда $Th(\Phi)$ – регистрово аксиоматезируема.

Можно занумеровать все выводы из $\Phi(\Phi^{\vdash})$, проверяя каждый раз принадлежат ли листья Φ. В зависимости от этого выводим на печать.

$H \subset \mathbf{N}x\mathbf{N}$ $(m, n) \in H$, если m-й вывод в этой нумерации приводит к формуле φ с номером n. Так как Φ – регистрово-разрешимо, то и H – регистрово разрешимо. Выражение $\Phi \vdash \varphi$ равносильно выражению $\exists m \in \mathbf{N}$ $(m, n_{\varphi}) \in H$

$\alpha_H(\upsilon_0, \upsilon_1) \in L^\sigma$ выражает H – конкретная формула

Рассмотрим формулу $Der_\Phi(x) \in L^\sigma$, такую что $Der_\Phi(x) = \exists y \alpha_H(y, x)$ (формулу x можно вывести из Φ).

Тогда существует φ, такая что она не выводима из Φ ($\exists \varphi : \Phi \vdash \varphi \leftrightarrow \neg Deg_\varphi(n_\varphi)$).

Лемма 5.4 *Пусть* Φ *– совместно* ($\models \Phi$). *Тогда* $\Phi \vdash \varphi$.

Доказательство: Пусть $\Phi \vdash \varphi$, тогда $\exists m \in \mathbf{N}$ $(m, n_\varphi) \in H$, откуда $\Phi \vdash \alpha_H(\underline{m}, \underline{n_\varphi})$ (т.к. α_n выражает H). А следовательно $\Phi \vdash \underbrace{\exists y\ \alpha_H(y, n_\varphi)}_{Der_\Phi(\underline{n_\varphi})}$, но $\varphi \leftrightarrow \neg Der_\Phi(n_\varphi)$, занчит $\Phi \vdash \neg\varphi$. Значит Φ несовместно(невыразимо), сто противоречит условию. □

Запишем утверждение о совместности Φ: $Con_\Phi \in L^\sigma$. $Con\Phi := \neg Der_\Phi \quad (n_0 = s(0))$ – Φ совместно если из него не выводится, что $0 = s(0)$.

Если $PA^\models \subset \Phi$, то $\Phi \vdash Con_\Phi \leftrightarrow \neg Der_\Phi(\underline{n_\varphi})$, где φ та самая неподвижная точка.

(доказательство аналогично теореме о неподвижной точке).

Теорема 5.8 *Вторая теорема Геделя о неполноте.*

Пусть Φ*– регистрово-разрешимое совместное множество формул из* L^σ*, достаточно богатое,* $PA^\models \subset \Phi$. *Тогда* $\Phi \nvdash Con_\Phi$.

Доказательство: Докажем от противного. Пусть $\Phi \vdash Con_\Phi$. Тогда $Phi \vdash \neg Der_\Phi(\underline{n_\varphi})$, но тогда $\Phi \vdash \varphi$. Получили противоречие с только что доказаным утверждением. □.

Возьмем в качестве L^σ язык теории множеств. Из аксиом **ZF** выводится арифметика. Пользуясь только аксиомами **ZF**, мы не можем доказать ее непротиворечивость. Средствами математики нельзя доказать непротиворечивость математики. Кроме того, никакое нетривиальное семантическое свойство программ не является разрешимым (тривиальные свойства – те которым удовлетворяют все программы, или которым не удовлетворяют программы вообще).

В некоторых случаях одна теория способна доказать непротиворечивость другой ("более слабой"). Так, в теории множеств **ZF** можно доказать непротиворечивость Th(**N**). Формула Con(Th(**N**)) недоказуема в Th(**N**) (если эта теория непротиворечива), однако перевод ее в язык теории множеств можно доказать с помощью аксиом **ZF**. Формула Con(Th(**N**)) - замкнутая формула теории Th(**N**), т.е. вполне определенное утверждение о свойствах натуральных чисел. Это утверждение в Th(**N**) недоказуемо, но его можно доказать в теории множеств. Эта несколько стложная для интуитивного понимания фраза означает в точности следующее: некоторые утверждения о свойствах натуральных чисел, требующие для своей формулировки только понятие натурального числа (и, казалось бы, касающиеся только этих чисел), требуют для своего доказательства сложные понятия, не укладывающиеся в рамки Th(**N**).

6 Язык Пролог

Пусть есть некая система аксиом Γ и некоторая теорема $\mathcal{P}$. Нужно проверить, является ли она семантическим следствием Γ или не является. Иными словами выполняется ли такое:

$$\Gamma \models \mathcal{P}.$$

Семантическое следствие заменим на выводимость формальную, а для формальной выводимости мы построим соответствующую формальную систему. Формальная система будет называться формальной системой опровержения. Будем обознаяать выводимость методом опровержений таким образом:

$$\Gamma \vdash_R P.$$

Понятно, что сама по себе задача является алгоритмически неразрешимой. Максимум, на что можно надеятся – это на "половину" решения: если действительно $\mathcal{P}$ является семантическим следствием, то тогда рано или поздно такое доказательство мы найдем, но если $\mathcal{P}$ не является семантическим следствием, то не бедет никакой возможности выяснить, появится ли доказательство через годы или оно не появиться никогда. На большее расситывать даже в принципе не следует.

Теперь будем строить эту систему.

Определение 6.1 *Подстановкой называется запись вида:*

$$\sigma = \left\{ t_1/x_1, t_2/x_2 \dots t_n/x_n \right\},$$

где $x_1, x_2 \dots x_n$ – символы предметных переменных, а $t_1, t_2 \dots t_n$ – это термы.

Если E – либо терм, либо атомная формула, либо отрицание атомной формулы, то тогда результат подстановки σ в E будем обозначать как E_σ.

Определение 6.2 *Если E – это либо терм, либо атомная формула, либо отрицание отомной формы, то E будем называть выражением.*

Определение 6.3 *Атомную форму или ее отрицание будем еще называть литералом.*

Пример 6.1 *Если E – литерал: $E = A(x, f(y), z)$, а*

$$\sigma = \left\{ c/x, g(b)/y, a/z \right\},$$

то тогда

$$E_\sigma = A(c, f(g(b)), a).$$

Определение 6.4 *Если σ и Θ – две подстановки:*

$$\delta = \left\{ t_1/x_1, t_2/x_2 \dots t_n/x_n \right\},$$

$$\Theta = \left\{ v_1/y_1, v_2/y_2 \dots v_n/y_k \right\}$$

Определим композицию подстановок σ и Θ следующим образом:

$$\sigma \circ \Theta = \left\{ t_1\Theta/x_1, t_2\Theta/x_2 \dots t_n\Theta/x_n, v_1/y_1, v_2/y_2 \dots v_k/y_k \right\}$$

Вычеркнем все замены вида $[x/x]$; если среди $y_1, y_2, \dots, y_k$ есть переменные из множества $\{x_1, x_2, \dots, x_n\}$, соответствующая подстановка также удаляется. Результирующая подстановка и будет являться композицией подстановок σ и θ: $\sigma \circ \theta$.

Пример 6.2

$$\sigma = \left\{ f(c)/x, b/y, y/z \right\}, \theta = \left\{ c/u, a/y, b/z \right\} \Rightarrow$$
$$\sigma \circ \theta = \left\{ f(c)/x, b/y, a/z, c/u, \underline{a/y}, \underline{b/z} \right\} = \left\{ f(c)/x, b/y, a/z, c/u \right\}$$

Упражнение 6.1 *Рассмотрим пустую подстановку $\varepsilon := \{\}$. Доказать:*

1. $\forall \sigma \ (\sigma \circ \varepsilon = \varepsilon \circ \sigma = \sigma)$;
2. *Для любого выражения E верно:* $(E\sigma_1)\,\sigma_2 = E\,(\sigma_1 \circ \sigma_2)$;
3. *Порядок применения композиций несущественен:* $(\sigma_1 \circ \sigma_2) \circ \sigma_3 = \sigma_1 \circ (\sigma_2 \circ \sigma_3)$.

Определение 6.5 *Подстановка σ называется унификатором (unifier) множества выражений $\{E_1, E_2, \dots, E_n\}$, если $E_1\sigma = E_2\sigma = \dots = E_n\sigma$.*

Пример 6.3 $E_1 = A(x, c)$, $E_2 = A(y, c)$, $\sigma = \left\{ z/x, z/y \right\}$; *$\sigma$—унификатор, так как* $E_1\sigma = E_2\sigma = A(z, c)$.

Замечание 6.1 *Теория унификаторов была разработана в 60-х годах 20 в. Джулией Робинсон (J. Robinson).*

Определение 6.6 *Унификатор σ множества выражений $\{E_1, E_2, \dots, E_n\}$ называется самым общим унификатором (most general unifier) этого множества, если для любого другого унификатора θ этого множества верно $\theta = \sigma \circ \rho$, причем $\rho \neq \varepsilon$.*

Определение 6.7 *Множество выражений называется унифицируемым, если у него есть хотя бы один унификатор.*

Теорема 6.1 *Любое унифицируемое множество выражений имеет самый общий унификатор, причем единственный с точностью до переобозначения переменных.*

Доказательство: Докажем существование самого общего унификатора. Представим алгоритм его нахождения и рассмотрим работу этого алгоритма на примере.

1. Запишем выражения из множества $E = \{E_1, E_2, \ldots, E_n\}$ в столбик, используя символы с постоянной шириной, то есть, чтобы i-ые символы разных строк были записаны один под другим. Например:

$$\begin{aligned} \mathrm{E_1} &= \mathrm{A\,(x,f\,(u))} \\ \mathrm{E_2} &= \mathrm{A\,(g\,(y)\,,f\,(h\,(z)))} \end{aligned}$$

2. Составим *disagreement set—множество несовпадений* D следующим образом: будем «сканировать» записанные выражения вертикальными линиями. Пропустив k столбцов, состоящих из одинаковых символов, внесем в множество D все термы, попадающие в $k+1$-ый столбец. В нашем случае $D\,(E) = \{x, g\,(y)\}$;

3. Построим унификатор σ_1 для множества D ($\sigma_1 = \left\{ g\,(y)/_x \right\}$) и применим его к выражениям: $E_1\sigma_1 = A\,(g\,(y)\,,f\,(u))\,, E_2\sigma_1 = A\,(g\,(y)\,,f\,(h\,(z)))$; в дальнейшем будем работать уже с этими выражениями.

4. Если множество E еще не унифицировано, перейдем к шагу 1, работая уже с множеством $E = \{E_1\sigma_1, E_2\sigma_1\}$

Очевидно, что, в соответствии с алгоритмом, рано или поздно множество E будет унифицировано (в нашем случае—на втором шаге, $\sigma_2 = \left\{ h\,(z)/_u \right\}$), причем полученный унификатор $\sigma = \sigma_1 \circ \sigma_2 \circ \ldots \circ \sigma_n$ будет самым общим. В случае, если множество неунифицируемо (например, разные выражения содержат различные константы), алгоритм будет выполняться бесконечно, однако такая ситуация невозможна по условию теоремы (множество выражений унифицируемо). $\square$

Трудоемкость данного алгоритма является не почти линейной, как может показаться вначале, а экспоненциальной вследствие действия под названием *occur check* , то есть "проверка вхождения": $\exists\,(x \in \mathbf{Var}, t \in \mathbf{Ter}) : x \notin t$. Она предотвращает зацикливание алгоритма, гарантируя, что disagreement set содержит только те термы, которым не принадлежит некоторая входящая в него переменная, то есть, позволяет избежать циклических замен вида $[f(x)/x]$ для множества $D = \{x, f\,(x)\}$.

6.1 Формальная Система Опровержения

Нормальная форма Скулема. Любая формула языка логики первого порядка может быть приведена к нормальной форме Скулема (Skolem)—к виду

$$\forall x_1 \forall x_2 \forall x_3 \ldots \forall x_n\,(C_1 \wedge C_2 \wedge C_3 \wedge \ldots \wedge C_n)$$

где C_i—фразы, конечные дизъюнкции литералов. Все кванторы всеобщности ($\forall$) выносятся вперед, кванторы существования ($\exists$) недопустимы. В скобках находится выражение в конъюнктной нормальной форме (КНФ)— конъюнкции элементарных дизъюнкций.

Замечание 6.2 *Литерал—атомная формула или ее отрицание.*

Процесс приведения выражения к нормальной форме Скулема состоит из следующих этапов: сначала с помощью семантических эквивалентностей осуществляется переход к КНФ, после чего все кванторы выносятся за скобки и исключаются кванторы существования.

Проиллюстрируем исключение квантора существования на примерах. Для исключения '$\exists x$' из $\exists x \forall y \left(A\left(x, y\right)\right)$ внесем в сигнатуру языка новую константу c и запишем: $\forall y \left(A\left(c, y\right)\right)$. В случае другого расположения кванторов, например, $\forall y \exists x \left(A\left(x, y\right)\right)$, необходимо ввести в сигнатуру уже новую унарную функцию f: $\forall y \left(A\left(f\left(y\right), y\right)\right)$; приведение $\forall y \forall z \exists x \left(A\left(x, y, z\right)\right)$ требует введения бинарной функции: $\forall y \forall z \left(A\left(g\left(y, z\right), y, z\right)\right)$.

В дальнейшем мы будем работать только с формулами в нормальной форме Скулема. Это не сужает круг рассматриваемых формул, так как любая формула языка логики первого порядка может быть записана в таком виде. Будем писать для краткости: $\mathcal{P} = \{C_1, C_2, C_3, \ldots, C_n\}$ вместо $\mathcal{P} = \forall x_1 \forall x_2 \forall x_3 \ldots \forall x_n \left(C_1 \wedge C_2 \wedge C_3 \wedge \ldots \wedge C_n\right)$. Информация о наличии кванторов всеобщности не теряется, так как здесь мы имеем дело только с замкнутыми формулами и для переменных всеобщность следует автоматически.

Введем правила вывода. Пусть C_1, C_2, Q—некоторые фразы. Будем говорить, что C_1 и C_2 выводят Q в формальной системе опровержения R и записывать $C_1, C_2 \models_R Q$, если:

- Существуют такие подстановки σ_1, σ_2, что $C_1\sigma_1$ и $C_2\sigma_2$ не имеют общих имен переменных;
- Существуют такие множества литералов $\{L_1, L_2, L_3, \ldots, L_n\} \in C_1\sigma_1$ и $\{L'_1, L'_2, L'_3, \ldots, L'_n\} \in C_2\sigma_2$, где литералы второго множества являются отрицаниями литералов первого, что объединенное множество со снятыми отрицаниями $\{L_1, L_2, L_3, \ldots, L_n, L'_1, L'_2, L'_3, \ldots, L'_n\}$ унифицируемо, его самый общий унификатор—σ;
- Формула Q имеет вид:

$$\left(\left(C_1\sigma_1 \setminus \{L_1, L_2, L_3, \ldots, L_n\}\right) \cup \left(C_2\sigma_2 \setminus \{L'_1, L'_2, L'_3, \ldots, L'_n\}\right)\right)\sigma$$

Пример 6.4 $C_1 = \{A\left(z\right) \vee \neg B\left(y\right) \vee A\left(g\left(x\right)\right)\}$, $C_2 = \{\neg A\left(x\right) \vee \neg C\left(x\right)\}$. *Переименуем переменные в соответствии с правилом 1:* $\sigma_1 = \varepsilon$ *(пустая замена),* $\sigma_2 = \{{}^{u}/_{x}\}$. *Далее, рассматривая множества литералов, получим* $\sigma = \left\{{}^{g\left(x\right)}/_{z}, {}^{x}/_{u}\right\}$. *Здесь наличие*

множеств $\{L_{\vec{i}}\}$ *и* $\left\{L'_{\vec{j}}\right\}$ *позволяет избавиться от не унифицируемых литералов, входящих в одну формулу в положительном, а в другую—в отрицательном виде. Замена переменных позволяет избежать неунифицируемых множеств вида* $\{x, f(x)\}$. *Например,* $\{\neg x, f(x)\} \models_R \bot$, *так как в процессе редукции (удаления литералов, ведущих к лжи) остается пустое множество, которое семантически эквивалентно лжи.*

Теорема 6.2 *Пусть множество фраз* S *невыполнимо, тогда* S *выводит пустую фразу в этой формальной системе опровержения (*$S \models_R \square$*).*

Эту теорему можно назвать теоремой о "полноте наоборот": тогда как полнота в обычном понимании означает, что, если некое выражение семантически истинно, то оно истинно и синтаксически, эта теорема утверждает о том, что семантическая ложь выводит пустой литерал.

Рассмотрим **алгоритм проверки выполнимости** множества фраз:

$$\text{REPEAT } \{S := \text{Res}(S)\} \text{ UNTIL } \{\square \in S\} : \text{PRINT } '\!S \text{ невыполнимо}'$$

Здесь $\text{Res}(S)$—множество резольвентов, то есть всех возможных выводов из пар фраз, принадлежащих S. Если множество выполнимо, данный алгоритм будет работать бесконечно.

Пример 6.5 *Запишем формулировку Парадокса Брадобрея:*

$$\forall x \forall y (B(x) \wedge \neg S(y,y) \rightarrow S(x,y)) \wedge \forall x \forall y (B(x) \wedge S(y,y) \rightarrow \neg S(x,y))$$

Мы хотим доказать, что $\neg \exists x B(x)$, *или, что то же самое, что* $\exists B(x) \models \bot$, *то есть, существование семантического противоречия.*

Представим данную формулу (а также цель) в виде множества фраз в нормальной форме Скулема: $\{\neg B(x), S(y,y), \neg S(x,y)\}$, $\{\neg B(x), \neg S(y,y), \neg S(x,y)\}$, $\{B(c)\}$. *Выведем из двух первых фраз новую:* $\{\neg B(x)\}$, *и получим новую систему:* $\{\neg B(x)\}$, $\{B(c)\}$, *из которой выводится пустое множество* $\square$.

Алгоритм представляет каждую формулу в виде набора фраз и, совмещая каждую с каждой, пытается сократить результат с учетом унификации до получения пустой фразы на выходе.

Можно оптимизировать алгоритм путем ввода правил, ограничивающих полный перебор. Таким образом, получаем два класса стратегий, которых придерживаются алгоритмы: *полные стратегии*, которые для противоречивого набора входных фраз всегда рано или поздно получают на выходе ложь, и *неполные*, позволяющие оптимально обрабатывать некоторый класс задач, но допускающие невозможность найти ответ для других (грубо говоря, алгоритм зацикливается при некоторых корректных входных данных).

Рассмотрим **структуру PROLOG-программы**. Автоматический доказатель языка PROLOG основан на схожих принципах и использует *неполную стратегию*, которая заключается в следующем: рассматриваются первые две фразы; затем к выводам из них добавляется третья и рассматриваются выводы уже из этого набора и так далее. Неполноту этой стратегии можно продемонстрировать на примере: из $(A \vee B) \wedge (A \vee \neg B) \wedge (\neg A \vee B) \wedge (\neg A \vee \neg B)$ никогда не будет получена ложь. Однако, при некоторых ограничениях на входные фразы данная стратегия является полной, а именно, если программа состоит только из *фраз Хорна* (Hoarn).

Определение 6.8 *Фразы Хорна – это фразы, содержащие не более одной положительной атомной формулы.*

Например, $A \vee \neg B$, $\neg A \vee B$ и $\neg A \vee \neg B$ – фразы Хорна, а $A \vee B$—нет; фразы Хорна также записываются следующим образом: $A_0 \vee \neg A_1 \vee \neg A_2 \vee \neg A_3 = A_0 \leftarrow A_1, A_2, A_3$, на языке PROLOG—$\mathtt{A0} : - - \mathtt{A1}, \mathtt{A2}, \mathtt{A3}.$; фраза Хорна, не содержащая положительных литералов, называется целью: $\leftarrow A_1, A_2, A_3 = : - - \mathtt{A_1}, \mathtt{A_2}, \mathtt{A_3}.$.

Таким образом, PROLOG-программа состоит из набора фраз Хорна и цели.

Пример 6.6
```
Dog(Rex).
Cat(Felix).
Animal(X) :- Cat(X).
Animal(X) :- Dog(X).
:- Animal(Rex).
```
Программа говорит о том, что Rex—это Dog, Felix—это Cat, любой Cat—это Animal и любой Dog—также Animal; доказать: Rex—Animal. Автоматический доказатель выдаст ответ 'Yes'. Можно также задать цель :– Animal(X), и ответ будет 'Rex' и 'Felix'.

Скажем в заключение, что автор не претендует на полноту изложения в данной книге тем в ней затронутых. Что касается языков логического программирования, то читатель заинтерсовавшийся, может продолжить свое знакомство с ними.