Программа Гильберта, $\mathbb{NP} \neq \mathbb{P}$, Рефлексия

Д. Л. Гуринович

12 марта 2017 года

УДК 510.52

Содержание

Аннотация

В данной работе сделана попытка исследования теории алгоритмов с точки зрения программы Гильберта, которая в своё время привела к принципиальным теоремам о неразрешимости и неполноте в логике.

В частности, рассмотрен вопрос о возможности получить ответ (позитивный или негативный) о наличии доказательства, не превышающего заданного размера S, для произвольного утверждения. Притом вопрос не просто о наличии такого доказательства, но о наличии алгоритма, который мог бы давать такие ответы за полиномиальное – от заданного размера S – количество шагов своей работы.

Была использована методика Гёделя по построению такого рода тестовой задачи, которая оказывается заведомо неразрешимой для заданного (произвольного) алгоритма-решения. При этом данная тестовая задача принадлежит классу $\mathbb{NP}$, а искомый алгоритм-решение

представляет из себя метод свести данную задачу к задаче из класса $\mathbb{P}$. Негативный результат поиска такого метода означает, в частности, доказательство известной гипотезы теории алгоритмов $\mathbb{NP} \neq \mathbb{P}$.

В процессе построения тестовой задачи выявились существенные дополнительные требованиях к формализму для представления алгоритмов и доказательств в теории алгоритмов в отличии от логики. В теории алгоритмов математические инструменты должны соответствовать скорости работы и размеру рассматриваемых алгоритмов, что не имело никакого значения при рассмотрении принципиальных вопросов логики о (не) существовании тех или иных результатов в принципе – ценой любого размера конечных доказательств и сколь угодно длительных, но конечных по своей работе алгоритмов.

А затем выяснилось, что тестовая задача хоть и принадлежит классу $\mathbb{NP}$, но отличается от «классических» задач из класса $\mathbb{NP}$ хотя бы тем, что теорема Кука к этой задаче не применима («авторизуемая задача»). И оказалось, что в «классических задачах» из класса $\mathbb{NP}$ не учитывается, что задача может содержать формализованную информацию для разных алгоритмов-решений о отличных между собой возможностях этих алгоритмов-решений по получению доказательств.

И обнаружилась исключительно интересная возможность (необходимая для дальнейшей работы) формализации рефлексии – когда алгоритм-решение получает от задачи ту информацию, которая относится именно к нему. Вот формализацию рефлексии я считаю самым интересным и принципиальным результатом данной работы, хотя удалось, вроде бы, доказать и неравенство $\mathbb{NP} \neq \mathbb{P}$.

Не уверен, что за время исследования в одиночку я не утратил в нём связь с реальностью, перестав замечать какие-то принципиальные ошибки. Но все построения в данной работе конструктивные, опираются на обычные методы работы с теориями первого порядка, строю даже пример действующей программы в подтверждение «Теоремы о построении алгоритма, применяемого к себе». И задача из $\mathbb{NP}$, которую строю – вполне конкретная и может быть реализована хоть на уровне языка программирования (что было бы очень громоздкой работой, конечно). Всё это, надеюсь, поможет при чтении данной работы проще её понять и обнаружить не замеченные мной ошибки.

1 Задачи из классов $\mathbb{NP}$ и $\mathbb{P}$, $\mathbb{NP}^+$ и $\mathbb{P}^+$

Как соотносятся между собой тяготы (затраты времени, сил и т.п.) между поиском ответа на вопрос и проверкой полученного ответа на вопрос? Из истории науки мы знаем, насколько трудными и долгими могли быть поиски ответов на возникшие вопросы, и насколько стремительно люди убеждались в правильности найденного ответа – после того, как он был найден.

Из недавней эпохи фундаментальных открытий в качестве примеров можно взять теоремы Гёделя в математике и уравнения для теории относительности и квантовой механики в физике. Одна теорема или формула позволяла не только решить отдельный принципиальный вопрос, но и обнаружить массу сопутствующих ответов, увязав множество фактов в целостную картину.

А может, раз проверка найденного ответа оказывается относительно простым делом, то и поиски ответа тоже можно сделать занятием относительно лёгким? Тогда осталось «только» найти такой способ поиска ответов на возникающие вопросы, который был бы примерно таким же лёгким, как проверка ответов. Разумеется, речь идёт о поиске правильного ответа на вопрос. Неправильный ответ найти – проще простого.

Вопрос соотношения тягот между поиском и проверкой ответов поставлен в математике в настоящее время. Это одна из принципиальных задач теории алгоритмов – вопрос о равенстве классов $\mathbb{P}$ и $\mathbb{NP}$.

А именно, задача из класса $\mathbb{NP}$ – это:

1.1. Проверка при помощи некоторого алгоритма $L(x, y)$ того, что «теорема» x истинна из-за «доказательства» y.

1.2. Время (количество «шагов») работы $L(x, y)$ полиномиально от размеров $|x|$, $|y|$. Считаем, что время $n \leq p_1(|x|, |y|)$, где $p_1(|x|, |y|)$ – некоторый полином.

Замечание: Явное представление полинома $p_1(|x|, |y|)$ – не является частью задачи. Просто «существует полином $p_1(|x|, |y|)\ldots$»

1.3. Если в результате проверки выясняется, что «y доказывает x», то $L(x, y)$ возвращает 1, в противном случае $L(x, y)$ возвращает 0.

1.4. Размер $|y|$ имеет полиномиальное ограничение относительно размера $|x|$ - и при превышении этого ограничения $L(x, y)$ вернет 0:

$|y| \leq p_2(|x|)$, где $p_2(|x|)$ - некоторый полином, иначе верно: $L(x, y) = 0$:

Замечание: Если размер $|y|$ превышает заданное через $|x|$ ограничение, то можно считать, что алгоритм $L(x, y)$ вернёт 0 так быстро, словно размер $|y|$ не превышает максимально допустимого при данном x размера $|y|$. Поясню:

Полиномиальное от размера $|x|$ ограничение на размер $|y|$ является частью определения задачи из класса $\mathbb{NP}$. Но, в отличие от замечания пункта 1.2, не понятно (я не нашёл в учебниках прямого ответа), представлен данный ограничивающий алгоритм в явном виде как часть задачи или просто «существует...». Однако по контексту изложения в учебниках теории алгоритмов – полиномиальное ограничение на размер $|y|$ задаётся явным образом в задачах из класса $\mathbb{NP}$, так как всегда известен тот круг «доказательств», внутри которого только и может находиться подходящее «доказательство» для «теоремы» x.

Значит, при превышении данного явного ограничения на размер «доказательства» сразу возникает информация о непригодности «доказательства», даже не рассматривая его целиком. Это – вычисление, значит, оно может быть представлено посредством алгоритма в соответствии с тезисом Чёрча . Пусть для «слишком больших» доказательств y это тоже будет алгоритм $L(x, y)$ – возвращающий 0 и делающий это за полиномиальное от размера $|x|$ (но не от размера $|y|$, когда $|y|$ слишком велик) количество шагов. Поэтому далее будем рассматривать только такие задачи из класса $\mathbb{NP}$, которые соответствуют данному замечанию.

Как найти этот полином $p_2(|x|)$ по алгоритму $L(x, y)$ - не принципиально для данной работы, потому что в данной работе будет использован отдельный аргумент, задающий максимальный размер корректного доказательства – когда потребуется ограничение на размер доказательства.

Кстати, если бы мы не могли свести задачу к варианту из замечания (с известным явным ограничением на допустимый размер $|y|$), то у нас просто не было бы разрешимой разницы между работой алгоритма проверки $L(x, y)$ для «нормальных» и «запредельно больших» доказательств y. И – вообще говоря – тогда может всегда оставаться возможность, что для какого-то гигантского по размеру доказательства y алгоритм $L(x, y)$ вернёт 1 при заданной конкретной теореме x.

Неразрешимость этой проблемы (распознать, что исчезла возможность найти для данного фиксированного x корректное доказательство с ростом размера доказательства) вытекала бы из

неразрешимости проблемы остановки (можно легко свести). И тогда неразрешимость вопроса о сведении класса $\mathbb{NP}$ к $\mathbb{P}$ доказывалась бы элементарно (с помощью нюанса из пункта 5 ниже). Это не то же самое, что неравенство $\mathbb{NP} \neq \mathbb{P}$, но близко к тому. Поэтому явное ограничение полиномом от $|x|$ на допустимый размер $|y|$ будем считать частью задачи из класса $\mathbb{NP}$.

1.5. Композиция $p_0(|x|) = p_1(|x|, p_2(|x|))$ задаёт ограничение на время работы $L(x, y)$ относительно размеров $|x|$. Это свойство соответствует замечанию, сделанному в предыдущем пункте. В то же время, данное ограничение в явном виде (как готовый алгоритм) не является частью задачи из класса $\mathbb{NP}$, так как один из алгоритмов в композиции – который был определён в пункте 1.2 – не задан явно для задачи.

1.6. Далее будем считать, что задача из класса $\mathbb{NP}$ полностью задаётся алгоритмом проверки $L(x, y)$, который соответствует замечанию из пункта 1.4 и ограничивающий алгоритм пункта 1.4 можно «вычислить» из программного текста алгоритма $L(x, y)$.

2.1. Данная задача будет из класса $\mathbb{P}$, если есть алгоритм $M(x)$, который в пределах полиномиального от $|x|$ времени (количество шагов работы алгоритма $M(x)$ – в пределах $p(|x|)$) возвращает:

y, если имеется y, такой что $L(x, y) = 1$,

0, если такого y нет.

2.2. Алгоритмы вроде $p_0(|x|)$, которые ограничивают допустимый размер доказательств, допустимое время работы и т.п. будем называть ограничивающими алгоритмами.

2.3. В дальнейшем мы дополним аргумент «теорема» несколькими аргументами с ограничивающими параметрами. А ограничивающий алгоритм окажется зависимым не только от размера «теоремы с параметрами» – вроде $p_0(|d|, |t|, |S|)$, но и от самих аргументов - вроде $p_0^{d,t}(|S|)$. То есть, при фиксировании части аргументов мы получаем некую новую (под)задачу со специфическим ограничивающим алгоритмом. При этом остаётся в силе и «общий» ограничивающий алгоритм $p_0(|d|, |t|, |S|)$. Мы ещё вернёмся к этому нюансу в пункте 2.4 раздела 6 «Задачи Кука и авторизуемые задачи из класса $\mathbb{NP}$».

3.1. Но надо учитывать и условность полиномиального соотношения между размерами $|x|$ и $|y|$. Дело в том, что нет алгоритма в общем случае, который устанавливал бы соотношение между размером теоремы и размером её доказательства – когда доказательство есть. В противном случае у нас был бы негативный тест на доказуемость, что, как известно из логики, не имеет места.

Да и в самой теории алгоритмов данное соотношение $p_2(|x|)$ (в цитате ниже названное $q(|x|)$) между размером теоремы («слова») и доказательством («свидетелем») потребовалось лишь для ограничения количества рассматриваемых «свидетелей» для заданного «слова»:

«Полиномиальное ограничение $q(|x|)$ на длину свидетелей ограничивает перебор потенциальных свидетелей» - Раздел 16.4 «Класс $\mathbb{NP}$» в учебном пособии для бакалавров по направлению «Информатика и вычислительная техника» и «Информационные системы» В.Н. Крупский, В.Е. Плеско «Математическая логика и теория алгоритмов». То есть, полиномиальность соотношения $q(|x|)$ – весьма условна.

С учётом этого будем подразумевать и такой вариант пункта 1.4:

3.2 или 1.4^+. Из аргумента x за полиномиальное количество шагов от размера $|x|$ можно выделить (определение!) «параметр предельного размера доказательства» S – который задаёт максимальный размер для $|y|$, при превышении которого $L(x, y)$ вернет 0. Записывать в таком

случае $L(x, y)$ можно и в виде $L(t, S, y)$ – сразу разбивая x на содержательную часть t и параметр предельного размера доказательства S. Считаем $|y| \leq S(x)$, где $S(x)$ работает полиномиальное время от размеров $|x|$, либо S имеется в готовом виде – когда алгоритм проверки имеет вид $L(t, S, y)$.

При этом число S представлено в позиционном виде – как десятичное число, например. Разумеется, если x содержит внутри себя ещё и S, то x будет уже не просто «теорема», а «теорема + наибольшая длина доказательства». И выделить число S из аргумента x тогда можно быстро и просто.

Композиция $q_0(x) = p_1(|x|, S(x))$ задаёт ограничение на время работы $L(x, y)$ относительно x, и это время конечно и полиномиально относительно размера $|x|$ и размера $|y|$.

А ограничение на размер $|y|$ вычисляется из аргумента x алгоритмом $S(x)$, который полиномиальный по времени своей работы относительно размеров $|x|$.

А если алгоритм проверки имеет вид $L(t, S, y)$, то $q_0(x) = p_1(|t|, S)$. Последний вариант более естественный для поиска доказательств, предельный размер которых задан аргументом S – который может быть самым произвольным и записанным в десятичном, например, виде.

3.3. Если вместо пункта 1.4 используется пункт 3.2 (1.4^+), то для таких задач определим класс $\mathbb{NP}^+$. Очевидно, что пункт 1.4^+ превращается в п. 1.4 в случае, когда $S(x) = p_2(|x|)$.

Класс $\mathbb{NP}^+$ более удобный, чем $\mathbb{NP}$ для исследования вопроса о поиске доказательств для логических утверждений. Как известно, нет алгоритма, который в общем случае связывал бы размер утверждения и размер его доказательства, а нас интересует возможность найти доказательство – если оно есть – за полиномиальное время относительно каких-то задаваемых нами пределов на размер доказательства. Или же за полиномиальное от заданного размера доказательства время узнать, что нужного доказательства в таких пределах нет.

4. или 2.1^+ Данная задача будет из класса $\mathbb{P}^+$, если есть алгоритм $M(x)$, который в пределах полиномиального от $|x|$ и $S(x)$ времени (т.е. количество шагов работы алгоритма $M(x)$ – в пределах p(|x|, S(x)) где $S(x)$ работает полиномиальное время от размера $|x|$) возвращает:

y, если имеется y, такой что $L(x, y) = 1$ и в т.ч. $|y| \leq S(x)$,

0, если такого y нет.

Разумеется, если алгоритм проверки имеет вид $L(t, S, y)$, то соответствующий ему алгоритм из $\mathbb{P}^+$ будет иметь вид $M(t, S)$.

И пункт 4 (2.1^+) превращается в п.2.1, если $S(x) = p_2(|x|)$. А задача из класса $\mathbb{P}^+$ оказывается тогда задачей из класса $\mathbb{P}$.

5. (Не)полиномиальность времени работы алгоритма никак не зависит от того, сколько времени данный алгоритм работает на данной единичной задаче – если время его работы конечно.

Действительно, любая единичная задача, которая решается за конечное время, может считаться решенной за полиномиальное время. Ведь полиномиальность времени решения относится лишь к массовой задаче, и если единственная задача (или конечное их число) из данной массовой задачи решается за огромное, но конечное время, то к полиному просто добавляется какая-то константа. Поэтому корректный алгоритм-решение $M(x)$ может работать сколь угодно долго над решением единичной задачи $L(x, y)$ (при заданном x). Но если он всё же остановится с корректным результатом – то его долгая работа никак не уменьшит возможности найти полиномиальное решение для массовой задачи данного $L(x, y)$ (при произвольном аргументе x).

2 Программа Гильберта и теория алгоритмов

Гильберт формализовал процесс доказательства, сделав его проверку доступным даже алгоритмам. Доказательство теперь не только приводило к выводам о некоторых формальных объектах, но и само стало одним из таких формальных объектов, доступных логическому анализу.

Целью такой формализации было применение методов формальной математики к ней самой с целью доказательства непротиворечивости.

И когда Гёдель реализовал логический анализ таких объектов логики, то выяснилась невозможность при помощи доказательств теории получить данные о некоторых важных логических формулах про эту же теорию. Просто сложность подобной логической формулы о теории не уступала сложности самой теории и поэтому формула не могла быть «просчитана» никаким доказательством теории. Таким образом, были открыты знаменитые теоремы Гёделя о неполноте. В частности, непротиворечивость теории не может быть доказана в самой теории, если теория непротиворечива.

Итак, в области логической полноты и доказательства непротиворечивости мы получили ответы – пусть и отрицательные. Теперь не решённым и принципиальным стал вопрос о поиске тех доказательств, которые есть. И вопрос это такой:

Вопрос о поиске доказательств

Найдётся ли универсальный алгоритм-решение, который сумеет находить в предоставленной ему теории (произвольной) ответ на вопрос, если этот ответ в теории есть? Найти, например, доказательство интересующей теоремы, если эта теорема имеет доказательство в теории и это доказательство имеет какие-то разумные размеры? А если подходящего доказательства нет, то чтоб алгоритм сообщил и об этом. И чтобы выдавал свои результаты в разумные сроки.

В предыдущем разделе были сформулировали задачи, пригодные для проверки доказательств – задачи из класса $\mathbb{NP}^+$. А вопрос о возможности быстро их решать (находить доказательства) – это вопрос о их сводимости к задачам из класса $\mathbb{P}^+$.

Естественно предположить, что нам надо формализовать логические утверждения, доказательства, алгоритмы и их работу до такой степени, чтобы полученные формальные объекты сами могли обрабатываться алгоритмами.

Казалось бы, необходимая формализация проведена математиками ещё в эпоху Гильберта, но нет – для наших целей она слишком «грубая». Та формализация была рассчитана на применение к теориям, без всяких ограничений по размеру доказательств теории и тем более по «времени работы». Последнее («время работы») к доказательству вообще не применимо, но зато является важной характеристикой для работы алгоритма (количество шагов его работы, если он останавливается).

1.1. Для наших нынешних целей время работы является критически важным параметром и формальное представление алгоритма должны быть «быстрым» и «кратким», если сам формализуемый алгоритм «быстрый» и «краткий». Иначе время работы с формальным представлением алгоритма будет принципиально отличаться от времени работы самого алгоритма и его размера.

1.2. Между корректными теориями и алгоритмами, от которых мы ждём ответа, есть важное отличие в том, что для них означает «отсутствие результата». В теории, например, когда нет

доказательства для интересующего нас утверждения, то мы считаем, что «теория не знает». А корректный алгоритм-решение должен выдать ответ, что нужного доказательства нет – что он его не нашёл. То есть – корректный алгоритм-решение должен остановиться с негативным результатом как минимум. Это – принципиальное отличие, которое позволяет корректному алгоритму «понять» свою неспособность к чему-либо, как мы увидим далее.

1.3. Не вдаваясь в детали, напомню, что Гёдель представлял запись логических формул и доказательств как произведение простых чисел в разных степенях. То есть, если в математической формуле/доказательстве некий математический символ стоит на некотором месте, то этот символ и это место задают соответствующее простое число и его степень. И из таких сомножителей получалось число, которое соответствовало данной формуле/доказательству.

Так вот, такое представление формул/доказательств не годится для решения вопросов о полиномиальности времени работы алгоритмов в силу того, что разложение числа на простые сомножители относится к задачам с не решенной сложностью и, видимо, является неполиномиально сложной задачей. На трудности разложения числа на простые сомножители строятся некоторые системы шифрования с открытым ключом. Вычислить произведение простых чисел легко, а вот узнать по заданному числу из каких простых чисел оно состоит – неполиномиально трудно (видимо). И вопрос о (не)равенстве классов $\mathbb{P}$ и $\mathbb{NP}$ – всего лишь попытка первого приближения к доказательству/опровержению стойкости таких методов шифрования.

1.4. Кроме того, в эпоху Гильберта математики разбирали «стандартную интерпретацию» математических теорий о натуральных числах. А эта интерпретация – о «счётных палочках». Если математик говорил тогда о 10, то это означало 10 каких-то единиц. Тогда число 11234 будет соответствовать не строке из 5 цифр, а более чем 11000 единицам и такая интерпретация – заведомо неполиномиальная относительно привычного для нас десятичного представления чисел. И даже аргументы для алгоритмов на ленте машины Тьюринга понимались в такой же «стандартной интерпретации» как блоки единиц. Будем называть модель стандартной интерпретации «интерпретация S».

Разумеется, когда мы решаем вопрос о полиномиальности работы алгоритмов, то мы не можем считать запись аргумента алгоритма в виде блока единиц приемлемой. Тогда 11234 будет записано блоком единиц длинной более 11000 штук, а обычные компьютерные алгоритмы запишут тот же аргумент в паре байт. И как мы в интерпретации S сможем тогда получать правильные выводы о полиномиальных компьютерных алгоритмах, если в нашей интерпретации они изначально неполиномиальны? Никак. Поэтому нам надо пользоваться моделью интерпретации C:

2.1. Говоря «число» мы обычно подразумеваем десятичную запись числа, а для компьютера –2^8-ричную или 2^{16}-ричную. И со всеми нашими правилами «сложения столбиком», «умножения столбиком» и т.п. все эти объекты полностью соответствуют аксиомам математики о числах. Назовём «обычную» модель интерпретации «интерпретация C» (компьютерная интерпретация), а стандартную модель интерпретации – «интерпретация S».

Разумеется, между этими моделями интерпретации есть связь. Когда я в бухгалтерской программе считаю приходы и расходы единиц товара на складе по документам, то я (и компьютер) использую модель интерпретации C. Но когда я иду на склад и провожу там инвентаризацию, то считаю я – каждую единицу. И вот в процессе подсчёта единиц товара я опираюсь на интерпретацию S, хотя записи в «чистовик» инвентаризационной ведомости я вношу в интерпретации

C.

И представлять логические формулы и программные тексты для алгоритмов мы будем в рамках интерпретации C. То есть так, как это делают компьютеры – где для разных символов (цифр, букв, знаков препинания, математических символов и др.) есть свои числа, записанные либо байтом (кодировка ASCII) или парой байт (кодировка UTF-8) или аналогичным образом. А из этих символов составляются числа и тексты – путем записи символов в определенном порядке в последовательности «друг за другом».

Разумеется, все теоремы, доказанные в эпоху Гильберта, остаются в силе и для интерпретации C, потому что она полностью соответствует тем аксиомам Пеано (например), которые применялись для интерпретации S.

Вспоминается известная шутка математиков того времени, что в геометрии вместо понятия «прямые» можно использовать и пивные кружки. Главное – чтоб они соответствовали аксиомам геометрии.

Таким образом, можно считать, что как формулы и числа записаны в данной работе – так же они представлены и в нашей интерпретации – где каждый символ формулы/программы/доказательства представлен соответствующим кодом на соответствующем месте строки. И этими строками будут оперировать наши логические формулы и алгоритмы.

Замечу, что я не «открываю Америку» и аналог «строкового представления» для формул/программ/доказательств вместо нумерации Гёделя имеется и в книге «Вычислимость и логика» Дж. Булоса и Р. Джеффри. С момента написания той книги произошел огромный скачок в компьютерных технологиях, во многих языках программирования оперируют как текстами программ – генерируя их, так и логическими текстами – и вовсе не в формате «гёделевых номеров».

Тем не менее, в математической логике возник не просто застой, но существенный откат от достигнутого уровня развития – а в теории алгоритмов даже не сформировались полиномиальные – в смысле распознавания – стандарты в представлении логических формул и алгоритмов. У меня есть мнение, почему сложилась подобная ситуация, и я его выскажу (см. конец подраздела 8.3 «Рефлексия» раздела 8 «Приложения») когда доберёмся до тех важных аспектов реального мира, которые не были формализованы в математике.

2.2. И ещё одно существенное усложнение в разбираемых тут вопросах: мы различаем алгоритмы между собой по их внутреннему устройству (исключая незначимые символы в программном коде – пробелы, комментарии и т.п.). Даже если алгоритмы выдают одни и те же результаты за одинаковое время, но «программный код» у них разный – то это разные алгоритмы и разными будут соответствующие им задачи.

2.3. Машина Тьюринга – также не является адекватным отражением методов вычисления, производимых в реальном мире. Машина Тьюринга придумывалась для анализа логических проблем, а мы разбираем вопросы теории алгоритмов – где вопросы времени и размеров являются принципиальными – в отличие от логики.

В частности, аргументы в реальном мире часто передаются «по ссылке» - то есть не всё значение аргумента передаётся байт за байтом – передаётся только адрес, начиная с которого располагаются данные, соответствующие данному аргументу.

Это значит, что аргументы в оптимально построенном алгоритме расположены «на соседних лентах» рядом. И для доступа к одному аргументу нет нужды «проехать на тележке» всю

длину другого аргумента. А другой аргумент может быть неполиномиально велик в сравнении с нужным нам аргументом. Например – слишком большое доказательство для малозначимой теоремы, которое должно быть отвергнуто в силу его размера (есть, например, менее громоздкие альтернативные решения). И это очень важный момент – возможность избегать перебора/передачи ненужных данных при доступе к нужным данным. И этой возможностью мы ещё воспользуемся, так как она имеется в реальном мире – пусть её и нет в модели «машина Тьюринга».

Это замечание делается к тому, что привязанность к созданным человеком формализмам может превращать мышление в бессмысленное занятие, когда эти формализмы перестают отвечать реальности. Какой смысл рассуждать о медленных алгоритмах, если они медленные только в неадекватном формализме? И если мы видим, как обстоят дела в реальном мире, то и формализм свой мы должны приводить в соответствие с ним. Назову условно этот довод - «довод божьего творения» или «довод реальности» - кому как нравится. Мы его часто используем – пусть и неявно. А пренебрежение им в итоге заводит в тупик бессмысленности.

3 Сложность вопроса о $\mathbb{NP} \neq \mathbb{P}$

Разберем следующую задачу, которую можно взять за основу для построения задачи из класса $\mathbb{NP}$:

1. У нас есть теория Th первого порядка (один из вариантов – теория Пеано, например) со списком аксиом

2. У нас есть некоторое утверждение x, истинность которого в рамках данной теории надо проверить

3. У нас есть текст доказательства y

Наша задача – проверить при помощи алгоритма $L(x, y)$ является ли доказательство y доказательством для утверждения x в рамках теории Th. То есть, алгоритм $L(x, y)$ у нас разный для разных Th, ну или он просто зависит от ещё одного аргумента – списка аксиом Th. Однако про зависимость $L(x, y)$ от Th будем помнить, но не будем пока загромождать обозначение лишними для наших рассуждений значками.

Благодаря Гильберту и его современникам мы знаем, что такие алгоритмы можно построить и они довольно простые. На базе данного алгоритма можно построить и задачу из класса $\mathbb{NP}$ – проверяя предварительно размер $|y|$ на соответствие некоторым полиномиальным ограничениям $p_2(|x|)$, но это ничего не меняет в следующих рассуждениях, потому что в логике всегда можно найти утверждение любого размера, эквивалентное заданному. Ведь эквивалентное теореме утверждение любого размера всегда можно построить тривиальным добавлением конъюнктивных членов $(1 = 1)$, например.

Разбираясь с задачей из $\mathbb{NP}$ на базе проверки доказательств, мы столкнемся с вопросом о непротиворечивости. И не решив вопрос непротиворечивости – не сможем доказать, что наша задача из $\mathbb{NP}$ не сводится к задаче из класса $\mathbb{P}$. А дело вот в чём:

Если теория противоречива, то начиная с доказательства о противоречии, мы моментально доказываем всё что угодно из тавтологии:

$\neg A \vee A \vee X$, где X произвольно, а переписать данную тавтологию можно в виде: $A \Rightarrow (\neg A \Rightarrow X)$

Тогда, если у нас есть противоречие – доказаны $\neg A$ и A, то при помощи правила вывода MP мы можем отбросить первую посылку из $A \Rightarrow (\neg A \Rightarrow X)$ т.к. «истинно» A, а затем и вторую из $\neg A \Rightarrow X$, т.к. истинно $\neg A$. И докажем любой X.

Получается, что для противоречивой теории вопрос о поиске доказательства решается за полиномиальное время от размера теоремы: достаточно найти доказательство противоречия (хоть перебором), а затем время составления всех остальных доказательств будет зависеть от размера доказываемого (любого!) утверждения X – линейно.

А это значит, что уровень сложности вопроса о невозможности свести разбираемую задачу на основе теории Пеано из класса $\mathbb{NP}$ к задаче из класса $\mathbb{P}$ не уступает уровню сложности доказательства непротиворечивости теории Пеано. Доказательство непротиворечивости теории Пеано имеется – и не одно, но их сложность превосходит теоремы Гёделя о неполноте, например. Ну, или доказательство, что данная задача на базе теории Пеано из $\mathbb{NP}$ не сводиться к задаче из $\mathbb{P}$ должно явно опираться на факт непротиворечивости теории Пеано, например. Тогда сложная часть доказательства – обоснование непротиворечивости – будет «импортирована» из другого доказательства как готовое знание.

Поэтому разбираемая нами задача на базе теории Пеано (или её расширения) может оказаться сложнее, чем вопрос о неравенстве $\mathbb{NP} \neq \mathbb{P}$: Возможно, найдётся такая задача в классе $\mathbb{NP}$, про которую удастся доказать, что её нельзя свести к задаче из класса $\mathbb{P}$, но при этом сложность задачи будет не настолько большой, чтобы вставал вопрос о доказательстве непротиворечивости.

Однако, по теореме Кука любая задача из класса $\mathbb{NP}$ может быть сведена к КНФ (конъюнктивной нормальной форме) из логики высказываний. А в логике высказываний нет никаких проблем с непротиворечивостью. Поэтому возникает сомнение в правильности теоремы Кука – в отношении хотя бы задач из $\mathbb{NP}$ на базе теории Пеано. И это сомнение – как будет показано в разделе 6 «Задачи Кука и авторизуемые задачи из класса $\mathbb{NP}$» – оправдано.

Пока же отметим, что если теорема Кука ошибочна, то при её использовании могут неявно возникать противоречия, а это может приводить к противоречиво построенной задаче, а противоречивая задача, в случае, если она строится на базе проверки доказательств, оказывается из класса $\mathbb{P}$. Но это будет – если речь о «$\mathbb{NP}$-полной» задаче из теоремы Кука – информация о классе $\mathbb{NP}$ в целом. И в силу этого использование теоремы Кука потенциально может создавать «доказательства» для равенства $\mathbb{NP} = \mathbb{P}$.

4 Предварительное замечание о программе Гильберта для теории алгоритмов

То, что алгоритм-решение $M(x)$ (см. раздел 1 «Задачи из классов $\mathbb{NP}$ и $\mathbb{P}$, $\mathbb{NP}^+$ и $\mathbb{P}^+$») ищет «доказательство» y для «теоремы» x на основе полученной информации об x и о массовой задаче $L(x, y)$ – тоже ведь должно быть формализовано при реализации программы Гильберта и рассмотрено с точки зрения логики – то есть, в некоторой теории.

На самом деле, алгоритм-решение должен получать текст программы алгоритма проверки и быть вида:

$M(\overline{L(x, y)}, x)$

Где $\overline{L(x,y)}$ обозначает текст программы алгоритма проверки (на каком-то языке программирования), а не сам алгоритм. Именно такое обозначение для текста программ мы будем использовать и в дальнейшем.

Но немного отложим уточнение, что имеется аргумент, содержащий текст программы алгоритма проверки. Будем пока считать, что алгоритм проверки зафиксирован и не меняется, поэтому не представлен как аргумент у $M(x)$.

В качестве аналога возьмём эпоху логических открытий, когда стоял вопрос о полноте теории Пеано и о возможности найти в ней доказательство для любого утверждения или его отрицания про натуральные числа. И тогда само доказательство стало объектом рассмотрения – и уже теоремы о доказательствах выявили неполноту теорий.

Теперь же мы рассматриваем возможность находить полиномиально быстрые (относительно размера своих аргументов) алгоритмы-решения для задач $L(x,y)$ из класса $\mathbb{NP}$. А искомый нами алгоритм $M(x)$ за конечное время определяет, имеется ли такое y, что $L(x,y)=1$.

Метод Гильберта состоял в применении математического формализма теории к теории. Метод данного исследования состоит в применении алгоритма-проверки к алгоритму-решению. А алгоритм проверки построен на базе теории, способной представлять утверждения про алгоритмы - включая заданный алгоритм-решение. Притом «специальный вопрос» (и соответствующая задача) «сопротивляется» заданному алгоритму-решению в его поиске ответа. Этот приём «изучаемый объект поступает вопреки прогнозам наблюдателя» удачно использовал Гёдель, и мы тоже воспользуемся им.

И, применяя выбранный алгоритм-решение к подобной теории (точнее, к соответствующему алгоритму-проверке), мы выясним, что алгоритм-решение не может найти «специальный» ответ про неё. И эта невозможность известна сразу, то есть - полиномиально быстро.

И у нас есть одна из таких задач, способных представлять алгоритмы и логические утверждения о них – это проверка доказательств в теории Пеано и в расширенной теории Пеано. Ведь теория Пеано может выразить любой алгоритм-решение. Так нет ли тут возможности поставить перед алгоритмом-решением (любым) непреодолимые для него задачи? И показать невозможность для алгоритма-решения $M(x)$ (любого) всегда давать правильные ответы про задачу из класса $\mathbb{NP}$.

Невозможность дать правильный ответ проявится вот как: Некое утверждение x имеет соответствующее доказательство y при правильном своём ответе $M(x)=z$ (именно z, который не обязательно равен y) и при данных x , y будет $L(x,y)=1$, но при этом корректный алгоритм-решение $M(x)$ не может найти данный y, а возвращает отличный от него результат z, при котором $L(x,z)=0$. Сочетание «корректности» работы $M(x)=z$ с тем, что $L(x,z)=0$, когда имеется y, при котором $L(x,y)=1$ кажется невозможным, но мы увидим обратное – алгоритм вернёт результат «не могу найти доказательство» и он действительно не может – поэтому ответ корректный. Но и подходящее для нашей задачи доказательство будет существовать – поэтому ответ алгоритма – неполный (алгоритм не сможет найти существующий ответ).

Приведённые в предыдущем абзаце «чудеса» станут возможны после разделения «теоремы» x на несколько частей – с появлением дополнительных параметров для рассмотрения. И тогда появится возможность учесть одну хорошо известную нам в обычной жизни зависимость, которая никак не была формализована в математике ранее. К этому вопросу мы вскоре вернемся – в разделе 6 «Задачи Кука и авторизуемые задачи из класса $\mathbb{NP}$».

5 Диагонализация

5.1 Лемма о диагонализации

Диагонализация. Это понятие возникло заметно позже Гильберта, но позволяет более лаконично излагать доказательства многих теорем о неполноте и неразрешимости. В частности – теорем Гёделя.

Очевидно, что всегда можно сделать алгоритм $AA()$, который сначала выписывает текст алгоритма $A(x)$, а затем запускает алгоритм $A(x)$ с этим текстом в качестве аргумента. Так вот, у этого алгоритма $AA()$ тоже есть текст его «программы», и этот текст может быть получен из текста алгоритма $A(x)$ посредством обработки этого текста неким алгоритмом $\mathrm{diag}(\overline{A(x)})$. Это довольно очевидно и такая возможность берется за исходную посылку. То есть – нам нужны достаточно выразительные теории, которые способны представлять алгоритмы и тогда алгоритм diag(...) – будет среди них. Теория Пеано – относится к таким теориям.

В теории алгоритмов для аналогичных целей используется теорема о неподвижной точке, но там нумерация алгоритмов сплошная и неполиномиальная поэтому, а нам нужна нумерация в духе «тексты алгоритмов». Да, не все тексты будут текстами алгоритмов, но этот случай нас не пугает – тогда такие «алгоритмы» ничего не возвращают и это можно понять.

Поэтому за основу возьмем Лемму о диагонализации из книги «Вычислимость и логика» Дж. Булос, Р. Джеффри. Только там она доказана для предиката, а мы докажем ее для алгоритма. В остальном я практически переписываю доказательство. И еще один момент – там доказательство на основе «представлений», а я оперирую с алгоритмами – логика та же, просто через представления возникает дополнительный «этаж» манипуляций, но мы просто можем считать, что наша теория чуть более выразительна в обозначениях, чем «чистая» теория Пеано и можно оперировать прямо со свойствами алгоритмов. Такая выразительность легко достижима и теоретически обоснована. Смотри:

Э. Мендельсон, «Введение в математическую логику», Наука, М. 1984. Глава 2, раздел 9 «Введение новых функциональных букв и предметных констант». Введение новой функциональной буквы, когда функция однозначно представлена в теории – ничего не меняет в логическом отношении.

Лемма. О диагонализации

Для произвольного алгоритма $B(x)$ найдется алгоритм $G()$, такой что $G() = B(\overline{G()})$.

Доказательство.

Назовем алгоритмом $G()$ тот алгоритм, который выполняет расчет как

$$B(\mathrm{diag}(\overline{B(\mathrm{diag}(x))}))$$

.

Собственно, $G()$ отличается от $B(\mathrm{diag}(\overline{B(\mathrm{diag}(x))}))$ только тем, что алгоритм $G()$ сначала выписывает аргумент $\overline{B(\mathrm{diag}(x))}$, а затем запускает композицию функций $B(\mathrm{diag}(...))$ с этим аргументом. Если у нас прописан порядок подстановки аргумента в алгоритм с одним аргументом и получения на этой основе алгоритма без аргументов, то алгоритмы $G()$ и

$$B(\mathrm{diag}(\overline{B(\mathrm{diag}(x))}))$$

– разные обозначения для одного и того же. Но нюансы разных вариантов подстановки аргументов в алгоритм будут рассмотрены в пункте 2.3 раздела 6 «Задачи Кука и авторизуемые задачи из класса $\mathbb{NP}$». Текст алгоритма $G()$ получается вот так:

$$\mathrm{diag}(\overline{B(\mathrm{diag}(x))})$$

.

Ведь алгоритм diag() построен именно так, чтобы возвращать текст подобных алгоритмов.

Теперь перейдём к доказательству Леммы.

1. По свойствам diag(): $\overline{G()} = \mathrm{diag}(\overline{B(\mathrm{diag}(x))})$
2. По построению $G()$: $G() = B(\mathrm{diag}(\overline{B(\mathrm{diag}(x))}))$
3. Из п. 1 подстановкой частей равенства в $B()$:

$$B(\overline{G()}) = B(\mathrm{diag}(\overline{B(\mathrm{diag}(x))}))$$

4. Из пп. 2 и 3: $B(\overline{G()}) = G()$

Лемма доказана

Как запомнить эту формулу: $B(\mathrm{diag}(\overline{B(\mathrm{diag}(x))}))$?

Изначально Гёдель решал вопрос о применении предиката к самому себе и формула $B(\mathrm{diag}(...))$ была как раз способом применить предикат / алгоритм к тексту такого алгоритма, который работает со своим собственным текстом. Но в качестве такового алгоритма подставлялся текст ... самого алгоритма $B(\mathrm{diag}(x))$ и поэтому получается выражение $B(\mathrm{diag}(\overline{B(\mathrm{diag}(x))}))$.

Смысл этого выражения в том, что предикат / алгоритм $B(...)$ применяется к самому себе. А вложенность $\mathrm{diag}(x)$ двойная, потому что сначала надо применять $B(...)$ к произвольному предикату / алгоритму с аргументом x, в котором передаётся собственный текст предиката / алгоритма, а затем на место «произвольного x» в $B(\mathrm{diag}(x))$ ставится сам $\overline{B(\mathrm{diag}(x))}$ - и в нём уже есть $\mathrm{diag}(x)$. Два этапа подстановки – для произвольного и конкретизация произвольного в данный предикат / алгоритм – порождают композицию с двумя алгоритмами диагонализации.

Впрочем, сам Гёдель термин «диагонализация» не использовал, насколько известно, но смысл его действий был именно такой.

С помощью леммы о диагонализации легко доказать, что множество теорем теории «неопределимо». То есть, что нет такого предиката $B(...)$, что при подстановке в качестве аргумента текста теоремы t будет доказано $B(t)$, если у теоремы есть доказательство в теории, или будет доказано $\neg B(t)$ в противном случае. Простое доказательство «неопределимости» можно посмотреть в упомянутой «Вычислимость и логика» Дж. Булос, Р. Джеффри. Мы же разберём аналогичный по природе пример:

Нет такого алгоритма $C(\dots)$, который при подстановке в качестве аргумента программного текста некоторого алгоритма a выдавал бы 1, когда алгоритм с текстом a выдаёт 1, а в противном случае выдавал бы 0.

Для доказательства построим

$G() = \mathrm{Anti}(C(\overline{G()}))$ – из Леммы о диагонализации

Где диагонализацию проводим по алгоритму $\mathrm{Anti}(C(\dots))$

Что мы, фактически делаем? Мы запускаем алгоритм $C(...)$, передав ему в качестве аргумента текст алгоритма $G()$, а когда $C(...)$ возвращает результат, то алгоритм Anti(...) меняет

его на «противоположный» и возвращает в качестве своего результата, опровергая результат $C(\overline{G()})$. Алгоритм Anti(...) построен так:

Anti(1) = 0; А если $X \neq 1$, то Anti(X) = 1

Построенный процедурой диагонализации алгоритм $G()$ действительно исполняет Anti(...) по результатам работы $C(...)$ – так как это часть программы $G()$, код которой и был передан $C(...)$. Это я пытаюсь наглядно пояснить смысл равенства $G() = \text{Anti}(C(\overline{G()}))$.

Гипотезами нашего доказательства будут:

1. остановка $C(\overline{G()})$ с результатом и 2. Соответствие результата $C(\overline{G()})$ результату работы $G()$. При этом в соответствии с гипотезой 1 результат работы $G()$ – будет (он остановится), так как он определён как $G() = \text{Anti}(C(\overline{G()}))$, справа композиция, в которой $C(\overline{G()})$ останавливается в соответствии с гипотезой 1, а за этим остановится и Anti(...).

Перепишем гипотезы в виде формул:

1. $C(\overline{G()}) = x$, где $x = 0$ или $x = 1$ и 2. $C(\overline{G()}) = \text{Anti}(\text{Anti}(G()))$

Здесь конструкция Anti(Anti($G()$)) сводит работу $G()$ к 1 и 0, но без «наоборот», потому что тут композиция двух Anti(...). И гипотеза $C(\overline{G()}) = \text{Anti}(\text{Anti}(G()))$ как раз выражает, что алгоритм $C(...)$ способен сообщить о работе произвольного алгоритма, но нас интересует только $G()$.

Теперь выведем противоречие из приведенных гипотез:

$G() = \text{Anti}(C(\overline{G()}))$ – из Леммы о диагонализации

$\text{Anti}(C(\overline{G()})) = \text{Anti}(\text{Anti}(\text{Anti}(G())))$ – из 2ой гипотезы, применив Anti(...) к обеим частям равенства

$G() = \text{Anti}(\text{Anti}(\text{Anti}(G())))$ – из 2ух предыдущих и транзитивности равенства

$\text{Anti}(\text{Anti}(G())) = \text{Anti}(\text{Anti}(\text{Anti}(\text{Anti}(\text{Anti}(G())))))$ – из предыдущего, применив Anti(Anti(...)) к обеим частям

$C(\overline{G()}) = \text{Anti}(\text{Anti}(\text{Anti}(C(\overline{G()}))))$ – из предыдущего, заменяя Anti(Anti($G()$)) в обеих частях по 2ой гипотезе

$C(\overline{G()}) = \text{Anti}(C(\overline{G()}))$ из предыдущего и свойств Anti(...)

$C(\overline{G()}) \neq \text{Anti}(C(\overline{G()}))$ из свойств Anti(...) и $C(\overline{G()})$ – когда есть остановка и результат (0 или 1) у $C(\overline{G()})$ в соответствии с 1ой гипотезой

$C(\overline{G()}) \neq C(\overline{G()})$ – из 2ух предыдущих и транзитивности равенства. Противоречит одной из аксиом равенство (рефлексивности).

Значит, гипотезы не верны, и теорема доказана.

Приведенное доказательство раскрывает природу трудностей (непреодолимых) при построении «универсального решения»: На любое «универсальное решение» есть более сложный алгоритм, который использует это «универсальное решение» против него самого. И причина неопределимости множества теорем теории – точно такая же.

5.2 Теорема о построении алгоритма, применяемого к себе

Диагонализации – давний инструмент логики, ещё до появления её названия, но мы получим гораздо более наглядное и удобное для наших целей следствие из доказательства этой леммы.

В процессе доказательства мы использовали произвольный алгоритм с одним аргументом B(X). Из него мы построили композицию B(diag(X)). И после того, как передали в качестве X

программный код данного алгоритма в него самого в качестве X, мы получили алгоритм $G()$. Интересно то, что после такой подстановки в аргумент X, результат выражения diag(X) будет в точности равен программному коду алгоритма $G()$.

Нам в данной работе далее будет интересно совсем не то, что алгоритм B(X) получает в X текст такого же алгоритма $G()$, которому он равен. Нам будет интересно следующее следствие леммы о диагонализации:

Теорема о построении алгоритма, применяемого к себе.

На базе произвольного алгоритма с одним аргументом $B(X)$ всегда можно построить такой алгоритм $G()$, что этот алгоритм $G()$ получает в переменную X свой собственный программный код и работает с ним так же, как алгоритм $B(X)$ работает с аргументом X.

Метод построения алгоритма $G()$ на базе алгоритма (X) чрезвычайно прост:

$B(\mathrm{diag}(\overline{B(\mathrm{diag}(x))}))$

А теперь разберём практический пример – напишем программу, которая возвращает свой собственный программный код.

Самый доступный способ программирования для офисных работников – воспользоваться языком VBA в MS Words.

За основу берем оператор, который вставляет в документ значение из переменной x и завершает работу программы:

Selection.InsertAfter x: End Sub

Это та часть, в которой происходит работа со значением переменной x. Для простоты будем считать передачей аргумента операцию присваивания переменной соответствующего значения.

Теперь нам надо исходить из предположения, что в аргументе «внешний» x у нас программный код, который что-то делает со значением переменной «внутренний» x («своей» переменой x, а не той, в которой мы его получили). И нам надо обработать это значение «внешней» переменной x так (реализация алгоритма $\mathrm{diag}(x)$), чтобы получить программный код, в котором исходный программный код подставляется во «внутренний» x и затем обрабатывается прежним образом исходным программным кодом из «внешней» x:

x = "Sub dolly(): x ="+ Chr(34) + `#Исходный_программный_код_в_виде_строки` + Chr(34) + ": "+ `#Исходный_программный_код`:

Всё просто – мы присваиваем «внешнему» x исходный программный код, но добавляем к этому тексту (перед ним) программный код присваивания «внутреннему» x того, что получили изначально во «внешнюю» переменную x (исходный программный код). Помимо операции присваивания нам надо поставить и инструкцию начало программы – "Sub dolly(): а операция присваивания требует заключать символы текста в кавычки. Всё это и проделано – пока схематически.

Теперь разберём `#Исходный_программный_код_в_виде_строки`.

Нам нужно, чтобы получился корректный программный код для присваивания с кавычками по бокам, а не внутри строки, которая стоит справа от знака присваивания. Допустимы такие конструкции:

x = "Текст1 без кавычек"+ "Текст2 без кавычек"+ ... + Chr(34) + ...

Где Chr(34) – как раз добавляют кавычки в текст. То есть, нам надо преобразовать исходное значение «внешнего» x в подобное выражение. И там, где в исходном значении стоят кавычки между Текст1 и Текст2, нам надо разрывать текст, добавляя кавычки программным образом:

x = "Текст1 без кавычек"+ Chr(34) + "Текст2 без кавычек"+ ...

Поэтому `#Исходный_программный_код_в_виде_строки` можно получить так из исходной «внешней» переменной x:

Replace(x, Chr(34), Chr(34) + "+ Chr(34) + "+ Chr(34))

Этот кусочек программы заменяет в переменной x знак кавычек

```
<до кавычек>"<после кавычек>
```

на следующее:

```
<до кавычек>" + Chr(34) + "<после кавычек>
```

У знака кавычек есть смысл в программном коде как ограничитель строк, поэтому в значение переменной они не попадают. А вот Chr(34) генерирует кавычки и они становятся частью значения.

После этого перепишем абзац:

x = "Sub dolly(): x ="+ Chr(34) + `#Исходный_программный_код_в_виде_строки` + Chr(34) + ": "+ `#Исходный_программный_код`:"

Таким образом:

x = "Sub dolly(): x ="+ Chr(34) + Replace(x, Chr(34), Chr(34) + "+ Chr(34) + "+ Chr(34)) + Chr(34) + ": "+ x:

А вместе с программным кодом помимо диагонализации имеем:

x = "Sub dolly(): x ="+ Chr(34) + Replace(x, Chr(34), Chr(34) + "+ Chr(34) + "+ Chr(34)) + Chr(34) + ": "+ x: Selection.InsertAfter x: End Sub

Теперь в соответствии с порядком действий из леммы о диагонализации нам надо присвоить этот программный код переменной x и выполнить его – применительно к изменённой переменной x.

Но для операции присваивания надо сделать ту же модификацию с данным программным кодом, что мы делали ранее с переменной x. Для того, чтобы получить код присваивания переменной x воспользуемся программой `string_let_code`. Это программа, которая по значению переменной (например, x) формирует тот программный код, который должен стоять справа в операторе присваивания (например, x), чтобы переменной присвоилось именно данное значение.

```
Sub string_let_code()
Dim x
x = InputBox("Введите значение для переменной "+ _
"и в текст будет выведен код, который надо будет написать"+ _
"справа от знака присваивания")
x = Replace(x, Chr(34), Chr(34) + "+ Chr(34) + "+ Chr(34))
x = Chr(34) + x + Chr(34)
Selection.InsertAfter x
End Sub
```

Предварительно скопируйте в «карман» предыдущий результат, запустите написанную только что программу, введите по её запросу текст из «кармана» и программа выдаст в текущий документ в текущее положение каретки программный код для присваивания. Вот он:

"x = "+ Chr(34) + "Sub dolly(): x ="+ Chr(34) + "+ Chr(34) + Replace(x, Chr(34), Chr(34) + "+ Chr(34) + "+ Chr(34) + "+ Chr(34) + "+ Chr(34)) + Chr(34) + "+ Chr(34) + ": "+ Chr(34) + "+ x: Selection.InsertAfter x: End Sub"

Теперь делаем на базе полученного результата операцию присваивания (с инструкцией начала программы) и добавляем ранее полученный код (тот, который мы подставляли в строку запроса при запуске программы `string_let_code`):

Sub dolly(): x = "x = "+ Chr(34) + "Sub dolly(): x ="+ Chr(34) + "+ Chr(34) + Replace(x, Chr(34), Chr(34) + "+ Chr(34) + "+ Chr(34) + "+ Chr(34) + "+ Chr(34)) + Chr(34) + "+ Chr(34) + ": "+ Chr(34) + "+ x: Selection.InsertAfter x: End Sub": x = "Sub dolly(): x ="+ Chr(34) + Replace(x, Chr(34), Chr(34) + "+ Chr(34) + "+ Chr(34)) + Chr(34) + ": "+ x: Selection.InsertAfter x: End Sub

Запустите построенную программу, и она выдаст свой собственный программный код.

6 Задачи Кука и авторизуемые задачи из класса $\mathbb{NP}$

1.1. В силу тезиса Чёрча любой искомый ответ получается от какого-то алгоритма. Значит, среди параметров (аргументов) задачи всегда можно задать тот, который отвечает за алгоритм, дающий ответ. Нужно лишь, чтобы задача была достаточно выразительной и была способна представлять произвольные алгоритмы.

Тогда вместо задачи $L(x, y)$ мы получим задачу $L(d, t, y)$, где аргумент x разделён на 2 аргумента:

t – теорема, для которой надо найти доказательство y;

d – аргумент долга – формальное представление того алгоритма, который ищет доказательство y для теоремы t. Самое простое формальное представление – программный код алгоритма на некотором языке программирования. Если речь о поиске доказательства без учёта ограниченных возможностей каких-либо алгоритмов-решений, то в качестве значения у аргумента долга d будем задавать прочерк $\overline{}$ и называть такое значение алгоритма долга «объективно».

1.2. В логике хорошо известны примеры вопросов, которые неразрешимы, так как для каждого алгоритма найдётся соответствующий вариант вопроса, на который алгоритм не может дать ответ. На этой базе мы и попытаемся – после серьёзной подготовки – построить полиномиально неразрешимый набор задач в классе $\mathbb{NP}$ и даже доказать $\mathbb{NP} \neq \mathbb{P}$. И да, когда корректный алгоритм даёт ответ ноль («не могу найти доказательство») – то этот ответ может быть и неполным – когда другой алгоритм доказательство найти может.

Возьмём хотя бы лемму о диагонализации (лемма и доказательство были приведены выше в подразделе 5.1 «Лемма о диагонализации») – за этой леммой был приведён пример для алгоритма-решения, который «предсказывает» поведение алгоритмов-задач (получая их текст в качестве аргумента). И там был построен «антиалгоритм»: Этот «антиалгоритм» запускает внутри себя алгоритм-решение, передавая ему в качестве аргумента свой собственный программный текст, и поступает иначе, чем «спрогнозирует» алгоритм-решение – если алгоритм-решение остановится. Но, заранее можно сказать, что алгоритм-решение не может предсказать

то, какой результат выдаст его «антиалгоритм».

Из этого видно, что в задаче $L(d,t,y)$ может быть задана такая «персональная подсказка» для некого (произвольного) алгоритма-решения $M(d,t)$, что при правильном понимании алгоритмом $M(d,t)$ данной задачи он выдаст ответ 0 («не могу найти доказательство про результат работы $AntiMt()$») за конечное время. Все нюансы мы подробно рассмотрим в своё время, а сейчас просто намечаем путь исследования.

1.3. Подход с тезисом Чёрча и обязательным наличием какого-то детерминированного алгоритма-решения (верного или не верного), дающего значение для искомого доказательства – верен в детерминированном мире, разумеется, но такова модель наших теорий. И в этом смысле для любой достаточно общей задачи, ответ для которой мы ищем, можно подразумевать не только наличие «исполнителя», дающего (или пытающегося дать) ответ, но и разные (!) правильные ответы для разных исполнителей – вообще говоря.

И тут мы выходим на рефлексию: Такой «дополнительный параметр» (аргумент долга) в произвольной задаче, который алгоритм-решение должен выбрать самостоятельно, чтобы понять, какую персональную информацию для него имеет данная задача – и этот дополнительный параметр должен соответствовать «я» (фигурально выражаясь) алгоритма. И наличие подобного «аргумента долга» оказывается не аномалией, а правилом в нашем мире – если в нём верен тезис Чёрча.

Замечу, что подобная задача «Сфинкс» $\mathrm{Sph}(d,t,y)$ с «персональной подсказкой» будет полностью соответствовать реальным фактам – если она корректно сформулирована и заявляет, что алгоритм «Эдип» $\mathrm{Oed}(\overline{\mathrm{Sph}(...,...,...)},\overline{=},t)$ не может найти доказательство y для теоремы t (при аргументе долга $\overline{=}$ - «объективно»), то так оно и будет. И, выдавать своё предсказание, корректно сформулированная задача «Сфинкс» будет полиномиально быстро – относительно размера «Эдипа» и теоремы t – независимо от того, как долго будет работать алгоритм-решение «Эдип». Даже если «Эдип» вообще не остановится.

1.4. Работу алгоритма проверки при аргументе долга «объективно» будем называть «базовым алгоритмом проверки». Но за счет дополнительной информации о некотором (или некоторых) «избранном» алгоритме-решении внутри задачи из $\mathbb{NP}$ можно будет заранее знать о его специфических возможностях по решению заданной задачи.

Принцип независимости «базового» алгоритма проверки от алгоритмов-решений назовём «принцип независимости алгоритма проверки». И «персональная информация» не должна вступать в противоречие с «базовой проверкой». Например:

Если «персональная информация» соответствует невозможности алгоритма-решения «Эдип» найти доказательство при аргументе долга «объективно» для теоремы t, то необходимо, чтобы «Эдип» действительно был не способен сделать это.

То есть, базовая проверка не зависит от алгоритмов-решений – она как законы природы – общая для всех объектов в нашем мире. Но сами объекты – разные, и у них разные возможности проявить себя при данных законах природы. И мы вполне можем заранее знать о некоторых алгоритмах, что они не могут что-то сделать в плане решения данной задачи. И это «заранее знать» об избранном алгоритме может быть гораздо более очевидным и быстрым по получению информации результатом, чем информация о наличии доказательств для не-избранных алгоритмов.

2.1. «Персональную подсказку» для алгоритма-решения «Эдип» алгоритм «Сфинкс» будет

выдавать при следующих аргументах, (пояснения в следующих двух подпунктах):

$\overline{\text{Sph}(\text{Oed}(\overline{\text{Sph}(...,...,...)},\overline{-},...),t,y)} = 0$

Где алгоритм долга у «Эдипа» равен «объективно» (символ прочерка), у «Сфинкса» он равен «Эдипу» с аргументом долга «объективно», теорема t фиксирована, а результат 0 получается при любом доказательстве y за время, полиномиальное относительно размеров |t| и $\overline{|\text{Oed}(\overline{\text{Sph}(...,...,...)},\overline{-},t)|}$.

И тогда корректный «Эдип» должен понять «персональную подсказку» для себя от «Сфинкса» и вернуть 0 (Ноль):

$\text{Oed}(\overline{\text{Sph}(...,...,...)},\overline{-},t) = 0$

2.2. Пояснение к пункту 2.1 о разных местах подстановки аргументов:

Аргумент-теорема t в $\text{Oed}(\overline{\text{Sph}(...,...,...)},\overline{-},...)$ не указывается, раз он указан в «Сфинксе»: $\overline{\text{Sph}(\text{Oed}(\overline{\text{Sph}(...,...,...)},\overline{-},...),t,y)}$. Не может же «Эдип» искать одно, а «Сфинкс» проверять другое. А вот аргумент долга «Эдипа» должен быть задан подстановкой в алгоритме «Эдип» - который содержится в аргументе долга «Сфинкса». Дело вот в чём:

У «Сфинкса» аргумент долга имеет большую вложенность, чем соответствующий ему аргумент долга у «Эдипа» (или другого алгоритма-решения), если мы говорим про аргумент долга «Сфинкса», которые отличается от «объективно».

И мы не будем разбивать аргумент долга на составляющие – «кто рассказывает» («Эдип»), «с точки зрения кого рассказывает» («Объективно» - в разбираемом случае), потому что тогда нам и того «с точки зрения кого» придётся разбивать на «кто» и «с точки зрения кого» и т.д. Вложенность может быть любой и нам не хватит тогда никаких аргументов.

2.3. Пояснение к пункту 2.1 о способе подстановки аргументов в один алгоритм с получением другого алгоритма:

Алгоритм по превращению одного текста алгоритма в другой – с подставленным первым аргументом – можно изображать так: $Arg1(\overline{MyArg},\overline{A(...,...)})$. Мы использовали выше и продолжим использовать далее обозначения типа $B(\overline{A(...,...)},x)$, $B(\overline{A(...,...)},...)$ и т.п. Подразумевается, что таким образом мы используем какой-то стандартный известный способ по превращению алгоритма, зависящего от нескольких аргументов в некие другие алгоритмы – в которых часть аргументов (или даже все) исходного алгоритма уже подставлена. А можем передавать тексты алгоритмов и без подставленных аргументов, как, например $\overline{A(...,...)}$.

Тут есть важное различие между двумя способами использования 1-го (без потери общности пусть будет 1-й) аргумента в алгоритме $A_1(x1,x2,...)$:

2.3.1. Фиксацией некоторого значения $\overline{text1}$ в аргументе $x1$ алгоритма $A_1(x1,x2,...)$ и превращения его в новый алгоритм $A_2(x2,x3,...)$;

2.3.2. Использование алгоритма $A_1(x1,x2,...)$ и его аргументов обычным образом: сначала заполняется (вне рамок инструкций и работы алгоритма $A_1(x1,x2,...)$) информация о значениях аргументов там, где алгоритм $A_1(x1,x2,...)$ ожидает найти эту информацию, а затем запускается алгоритм $A_1(x1,x2,...)$.

Важное отличие между 2.3.1 и 2.3.2 в том, что в случае 2.3.1 время на присваивание переменной $x1$ значения $\overline{text1}$ является частью времени работы алгоритма $A_2(x2,x3,...)$, а размер этой операции – включая размер текста $\overline{text1}$ - является частью размера алгоритма $A_2(x2,x3,...)$. Поэтому в случае 2.3.1 мы не можем рассматривать значение переменной $x1$ как полученное

по ссылке (когда передаётся адрес места, где находятся данные, а не сами данные – байт за байтом).

При использовании алгоритмов в духе варианта 2.3.1 будем использовать обозначение $A_1(\overline{text1}, x2, ...$ что по виду не отличается от варианта 2.3.2 при аргументе $x1$ со значением $\overline{text1}$. Но если по контексту будет не ясно, какой из вариантов имеется в виду, то будем это специально оговаривать.

Например, на место первого аргумента внутри формулы:

$\mathrm{Sph}(\mathrm{Oed}(\overline{\mathrm{Sph}(...,...,...)}, \overline{=}, ...), t, y)$

подставлен программный текст $\mathrm{Oed}(\overline{\mathrm{Sph}(...,...,...)}, \overline{=}, ...)$. И в этом программном тексте подстановка в первый и второй аргументы значений $\overline{\mathrm{Sph}(...,...,...)}$ и $\overline{=}$ выполнены по варианту 2.3.1. И всегда для аргумента, который относится к подстановке внутри текста программы или текста логической формулы мы будем использовать вариант 2.3.1.

А если аргумент не является частью текста программы/логической формулы – может быть и вариант 2.3.1 и вариант 2.3.2. Например, в разбираемом сейчас алгоритме

$\mathrm{Sph}(\mathrm{Oed}(\overline{\mathrm{Sph}(...,...,...)}, \overline{=}, ...), t, y)$

аргумент y – подставляется по варианту 2.3.2. Ведь время работы данного алгоритма не зависит от размера слишком больших аргументов y и поэтому аргумент y – именно аргумент, размер которого и время, расходуемое на заполнение аргумента значением, не являются ни частью размера разбираемого алгоритма, ни частью времени его работы.

Более того, аргумент y передаётся по ссылке, чтобы не передавать байт за байтом ненужную (при слишком большом размере значения аргумента y) информацию при использовании «Сфинкса».

Разницу между вариантом 2.3.1 и 2.3.2 можно практически наблюдать на примере программы из подраздела 5.2 «Теорема о построении алгоритма, применяемого к себе». Там видно, что изначально нет операции присваивания «аргументу» x (хотя там аргумент представлен «готовой» переменной). Программа просто использует готовое значение из x и только с этого момента отсчитывается её размер и время её работы. А в процессе исполнения процедуры диагонализации присваивание становится частью «тела» и процесса работы итоговой программы.

И в подразделе 5.1 «Лемма о диагонализации» формула $B(\mathrm{diag}(\overline{B(\mathrm{diag}(x))}))$, используемая в доказательстве Леммы, понимается в смысле варианта 2.3.1.

2.4. Притом «персональную подсказку» для алгоритма-решения «Эдип» алгоритм «Сфинкс» должен выдать за полиномиальное количество шагов относительно не всех аргументов задачи, а относительно какого-то параметра (или параметров), которые позволяют выделить нужную зависимость остальных аргументов так, чтобы получались только «подходящие» аргументы для «персональной подсказки».

Например, пусть среди аргументов будут ещё аргументы S, s. Аргумент S будет обеспечивать все нужные зависимости, а аргумент s будет лишь иметь соответствующий размер, допускающий нужный размер аргумента y (у нас же задача из $\mathbb{NP}$ и наибольший размер доказательства y должен опираться через полином на размер каких-то других аргументов). Тогда перепишем формулу «персональной подсказки» для «Эдипа» так:

$\mathrm{Sph}(\mathrm{Oed}(\overline{\mathrm{Sph}(...,...,...,...,...)}, \overline{=}, ...,...,...), t(S), S, s, y) = 0$

А теперь можно представить данный алгоритм проверки в сокращённом виде – превратив

общую исходную задачу проверки «Сфинкс» в задачу только для разбираемой персональной подсказки:

$\mathrm{Sph}_S(S, s, y) = 0$

При этом известно, что никакое доказательство не подходит и ответ получается без рассмотрения доказательства для данной частной подзадачи. То есть тут – время работы алгоритма $\mathrm{Sph}_S(S, s, y)$ не зависит от y (и от размера $|y|$). И еще – между y и s есть соотношение – полиномиальности. Поэтому, когда «отмирает» y – то «отмирает» и s за ненадобностью.

Вся эта «абстрактная» подготовка в данном пункте – является более развёрнутым изложением пункта 2.3 раздела 1 «Задачи из классов $\mathbb{NP}$ и $\mathbb{P}$, $\mathbb{NP}^+$ и $\mathbb{P}^+$». А соответствующий алгоритм «Сфинкс» будет реализован в следующем разделе.

От «Эдипа» требуется решать полученную «подзадачу» на тех же принципах, что и любую задачу из класса $\mathbb{NP}$ (раз мы ищем полный алгоритм-решение, способный сводить произвольную задачу из класса $\mathbb{NP}$ к задаче из класса $\mathbb{P}$). И тогда время ответа «Эдипа» для частной задачи из «персональной подсказки» с её ограничивающим полиномом на время работы $p_{\mathrm{Sph}_S}(|S|)$ должно будет уложиться в его («Эдипа») ограничивающий алгоритм от тех же значимых аргументов – $p_{\mathrm{Oed}_S}(|S|)$. А такой ограничивающий алгоритм для времени работа «Эдипа» при ответе на «специальный вопрос» может быть неполиномиально меньше, чем у алгоритма-проверки для общей задачи – $p_0(|S|, |s|, ...)$.

При таком поведении «Эдипа» будет соблюдён и принцип решения задач из класса $\mathbb{NP}$: время работы алгоритма-решения не должно превышать некоторого полинома от тех значений, которые составляют ограничивающий полином для времени работы алгоритма проверки. И будет соблюдён принцип идентичности требований к решению задач – составленных подстановкой нужных зависимостей в некоторые параметры из более общей задачи – с одной стороны, а с другой стороны – те же задачи, но в которых эти же зависимости изначально считаются фиксированными (частью программы).

Обобщим полученный вывод:

Если для «персональной подсказки» алгоритма «Сфинкс» о возможностях алгоритма «Эдип» имеется ограничивающий полином $p_0(|x1|, |x2|, ...)$ на время получения «подсказки», то корректный алгоритм «Эдип», способный сводить задачи из класса $\mathbb{NP}$ к задачам из класса $\mathbb{P}$, найдёт правильный ответ (если он есть) или выдаст 0 (если «персональная подсказка» не содержит правильного ответа) за время, ограниченное некоторым полиномом от тех же аргументов $p(|x1|, |x2|, ...)$.

2.5. Для тех, кто помнит пункт 5 из раздела 1 «Задачи из классов $\mathbb{NP}$ и $\mathbb{P}$, $\mathbb{NP}^+$ и $\mathbb{P}^+$» поясню, что полиномиальность времени работы действительно является атрибутом массовой, а не единичной задачи – то есть, не все параметры (аргументы), относящиеся к «персональной подсказке» в ней могут быть зафиксированы. Уже и по этой причине нам потребовались дополнительные аргументы S, s.

3. Возникает вопрос об уместности требования от корректного «Эдипа» выявлять «подзадачи», которые пусть реально существуют внутри общей задачи, но не в «готовом виде». Но в таком нашем требовании к корректному «Эдипу» мы не нарушаем сам принцип требования к «Эдипу» – «понимать условности». Даже «готовый вид» задачи из класса $\mathbb{NP}$ – это некая условность. Мы даже информацию об алгоритме проверки «Сфинкс» передаём на каком-то языке. А это может быть Pascal, C, Prolog или даже на русском – из программы бухучёта 1С.

В сфере IT вопрос взаимодействия между программами является одним из важнейших. Написаны объёмные технологические стандарты типа com+, Corba, программы на .Net содержат в себе «манифесты» для других программ, в Интернете работает множество протоколов взаимодействия между клиентом и сервером и т.д. и т.п. И это всё – не принципиальные вопросы, а вопросы принятых условностей. Мы просто рассматриваем те алгоритмы-решения, которые «понимают» нужный нам «протокол». Если наша программа ориентирована на русский язык, то «английские» алгоритмы-решения её не поймут, но важен принцип возможности решить (или невозможности решить) со стороны той программы, которая «понимает» не принципиальные условности.

Даже «персональная подсказка» выдаётся по определённому протоколу. Как «Эдипу» выделить нужную «подзадачу»? Такой вопрос стоял бы, если бы у нас было множество «специальных задач», тогда и «Сфинкс» был бы написан иначе – и намного сложнее (в соответствии с каким-нибудь технологическим стандартом, например). Но мы будем рассматривать лишь одну «специальную задачу» – «АнтиЭдип» (для произвольного «Эдипа»). Наш протокол, который должен понимать «Эдип», можно упростить до такого: «Ищи персональную подсказку у «Сфинкса» для соответствующего тебе «АнтиЭдипа».

И ещё – по поводу того, что в общем случае для решения задачи из класса $\mathbb{NP}$ алгоритм-решение должен найти и использовать в данной задаче «персональную подсказку» для себя:

Нас ведь не удивляет, что в реальной жизни задачи, которые мы решаем – обычно поставлены персонально перед нами? Или перед нашим коллективом? Иногда их ставят люди, иногда «жизнь» или «воля божья» - кому как нравится считать. И то, что возможность решения задачи и результат решения зависят от «исполнителя» – как правило – факт. А то, что подобные факты до сих пор не были формализованы в математике – огромное упущение, которое будет исправлено в данной работе. И это гораздо более принципиальная и интересная часть работы, чем доказательство $\mathbb{NP} \neq \mathbb{P}$ – на мой взгляд.

Ещё кое-что о дополнительной «сложности» – понять, что речь «о тебе» – будет сказано в данном разделе ниже в пункте 5.1.

4.1. Заметим, что «персональная подсказка» для «Эдипа» про его возможность доказать теорему t – это вовсе не ответ 0 на все доказательства, которые не может дать «Эдип». «Эдип» даёт в лучшем случае одно доказательство для заданной ему для доказывания теоремы, а может и вовсе не остановиться, и не дать ничего. «Персональные подсказки» - это обычный (при аргументе долга «объективно») результат алгоритма проверки, в котором исключены из числа корректных те доказательства, которые алгоритм-решение не может выдать по какой-то причине. Например:

Все доказательства, доказывающие данную теорему про «АнтиЭдип», кроме тех доказательств, которые «Эдип» не может доказать в силу того, что он не может предсказать поведение «Анти-Эдипа».

Персональная подсказка – это дополнительный фильтр, не пропускающий кое-что из того, что по каким-то причинам не может быть сгенерировано алгоритмом-решением.

Это как при расширении исходной теории Пеано новыми аксиомами арифметики – каждая новая аксиома – дополнительное ограничение, которое не допускает (под угрозой превращения теории в противоречивую) добавления в теорию тех утверждений, которые будут противоречить новой аксиоме. Но при этом отсутствие каких-то аксиом не означает противоречия –

аксиом всегда «не хватает» (теории неполны) в теориях первого порядка, которые эффективно аксиоматизируемы. При этом из двух отрицающих друг друга утверждений лишь одно истинно в полной арифметике, но выразить этот факт доступные нам для понимания теории не могут. Зато это такие теории, с которыми можно работать на базе аксиоматического подхода.

Также и «персональная подсказка» для «Эдипа» - оставляет без «запрета» какие-то варианты ответов, которые для «Эдипа» исключены. Зато «персональная подсказка» может быть получена полиномиально быстро.

И для всех «обычных» (не «Эдипов» в отношении теоремы t) алгоритмов-решений тогда будет верно равенство:

$\overline{\mathrm{Sph}(\mathrm{NotOed}(\overline{\mathrm{Sph}(...,...,...)},\overline{-},t),t,y)} = \mathrm{Sph}(\overline{-},t,y)$

4.2. Таким образом, задача из класса $\mathbb{NP}$ в общем случае подразумевает наличие у алгоритма-проверки ответов, зависящих от алгоритмов-решений (ответы, зависящие от аргумента долга с соответствующим значением). Выделение аргумента долга в отдельный аргумент – операция условная. Со времён «канторовой пары», известно, что в одно значение (в один аргумент) можно полиномиально быстро «упаковать» и «распаковать» сколько угодно много значений. Но может оказаться и так, что в задаче из $\mathbb{NP}$ не существует никаких «персональных подсказок». Тогда мы получаем «классическую» задачу из $\mathbb{NP}$ или «задачу Кука». Дадим определение:

Классические задачи из $\mathbb{NP}$ (или задачи Кука) - те, которые нельзя свести к задачам со значимым аргументом долга. Классическая задача – это задача из $\mathbb{NP}$ с незначимым аргументом долга (когда для всех алгоритмов-решений информация одинаковая).

Чтобы как-то отличить «не классические» задачи из $\mathbb{NP}$ от задач Кука, назовём неклассические задачи – «авторизуемые задачи из класса $\mathbb{NP}$». «Авторизация» - термин из сферы IT. Такое название отражает то обстоятельство, что для решения данных задач от алгоритма-решения требуется способность отличать «себя» от «не себя», требуется уметь использовать соответствующую «тебе» персональную информацию, которую может выдать при соответствующем запросе алгоритм проверки.

4.3. И задачу Кука действительно можно свести к КНФ (конъюнктивной нормальной форме), как это делается в теореме Кука. Отмечу только, что посылка теоремы Кука о наличии готовой информации об ограничивающем алгоритме (для времени работы алгоритма проверки) ошибочна. Но можно, видимо, исправить данную ошибку перебором «кандидатов» в ограничивающие алгоритмы в виде $2^n * x^{2^n}$. Тогда для какого-то n будет достигнут (если это действительно задача Кука) ограничивающий алгоритм (с «перелётом» степени, скорее всего, но всё равно полиномиальный), а предыдущих $i < n$ шагов будет лишь логарифмическое количество от найденного n, что не повлияет на полиномиальность.

Разумеется, предлагаемое исправление ошибки в теореме Кука не позволяет, вообще говоря, отличить задачи из $\mathbb{NP}$ от тех, для которых ограничивающего алгоритма нет. Это и закономерно –нет отрицательного теста на остановку алгоритма и, следовательно, нет отрицательного теста на принадлежность алгоритма к классу $\mathbb{NP}$ (с наличием ограничивающего алгоритма). Что очевидно с учётом пункта 5 из раздела 1 «Задачи из классов $\mathbb{NP}$ и $\mathbb{P}$, $\mathbb{NP}^+$ и $\mathbb{P}^+$».

Если же задача является задачей Кука, то предложенный перебор n будет конечным, так как решение в каждом испытываемом случае допускает проверку при помощи алгоритма проверки – одинаковом для всех алгоритмов-решений. Однако для авторизуемых задач из $\mathbb{NP}$ дело обстоит иначе (и не только в отношении проверки).

4.4. Ясно, что превращение в КНФ – это разбор программного текста алгоритма проверки и приведение его к специальному виду – с целью некого стандартного анализа, если таковой имеется.

И разбор программного кода в общем случае необходим, потому что просто запускать алгоритм проверки при разных аргументах и таким путём найти нужное доказательство – не получится быстро: это неполиномиально долгий перебор. Поэтому выход – в анализе текста программы, а уж как делать этот анализ – через приведение к КНФ или иначе – вопрос второй.

4.5. Интересно, что в основе задачи Кука может быть и теория Пеано, которая, вроде бы, не сводится к логике высказываний. Но сама по себе проверка доказательств – действие исключительно простое – поэтому на него и опирается весь формализм – и в логике высказываний, и в сложных теориях первого порядка. Подобную проверку можно сводить к логике высказываний. Но! – когда речь идёт о свойствах алгоритмов и их возможностях по выдаче результатов – то тогда иное дело.

И алгоритм проверки «Сфинкс» с «персональной подсказкой» содержит в себе не просто обычный алгоритм проверки доказательств, но и выражает зависимость результата проверки от некого алгоритма («Эдипа»). А это уже задействует более богатые возможности теории Пеано, чем те, которые мы используем при проверке доказательств. И вот здесь к логике высказываний задача уже – не сводится.

4.6. И не сводится потому, что получая алгоритм проверки – алгоритм-решение должен «понять», нет ли «персональной информации» для него. А это уже выбор – на основе терма. А к логике высказывания индивидные переменные и символы не сводятся.

Интересные подробности о необходимости выхода за рамки логики высказываний для математики можно почитать в «Основаниях математики»:

Без индивидных переменных и индивидных символов логика высказываний «... обходит стороной один очень существенный логический момент, а именно – отношение сказуемого к подлежащему, т.е. связь между субъектом и предикатом» - «Основания математики» Д. Гильберт, П. Бернайс, Том 1, Глава IV «Формализация процесса вывода II: Исчисление предикатов», Параграф 1. «Введение индивидных переменных ...». И далее – использование функций с индивидными символами и переменными, любое корректное сочетания которых представляет из себя «терм» - «Основания математики» Д. Гильберт, П. Бернайс, Том 1, Глава V «Исчисление предикатов с равенством ...», Параграф 1. «Расширенный формализм», п. 5 «Добавление функциональных знаков: понятие терма ...».

5.1. Здесь перед алгоритмом-решением возникает дополнительная сложность – понять, что речь «о тебе». Но это одна из решаемых сложностей и чем она принципиально отличается от любых других сложностей при поиске ответа на вопрос? Сложности бывают самые невероятные и у многих сложностей – нет пока что способа решения (а у некоторых неразрешимых вопросов вообще никогда не будет решения в общем случае) – в отличие от «распознания себя». Для такого распознания подходит, например, метод диагонализации.

Есть полиномиально быстрая информация о возможностях данного алгоритма? Да. Является ли эта задача из класса $\mathbb{NP}$? Да. Можно поставить вопрос о возможности данного алгоритма-решения выдать соответствующий ответ за время, полиномиальное от времени на получение данной «персональной подсказки»? Да. Будет ли этот ответ из класса $\mathbb{P}$ для такой подзадачи?

Да.

Но при этом выяснилось, что задача имеет разные правильные решения для разных алгоритмов-решений. И это та особенность, которая не сводится к одной КНФ на все алгоритмы-решения.

5.2. И, разумеется, у нас нет интереса к «неклассическим» задачам из $\mathbb{P}$ – мы доказываем неполноту именно относительно сведения $\mathbb{NP}$ к «обычным» задачам из $\mathbb{P}$ (или $\mathbb{NP}^+$ к $\mathbb{P}^+$) и другими здесь не интересуемся.

7 $\mathbb{NP} \neq \mathbb{P}$

Чтобы построить авторизуемую задачу из класса $\mathbb{NP}$, которую не может решить (свести к задаче из класса $\mathbb{P}$) полным образом ни один алгоритм-решение, мы поступим следующим образом:

1.1. Для произвольного алгоритма-решения «Эдип» мы построим алгоритм «АнтиЭдип», про результат работы которого «Эдип» заведомо не может доказать соответствующее утверждение.

1.2. И в нашей авторизуемой задаче «Сфинкс» из класса $\mathbb{NP}$ будет «персональная подсказка» для «Эдипа», в соответствии с которой корректный алгоритм-решение «Эдип» должен вернуть ноль – «не могу найти доказательство утверждения про результат работы алгоритма АнтиЭдип». Притом ограничивающий алгоритм данной «персональной подсказки» у алгоритма «Сфинкс» будет неполиномиально меньшим («Сфинкс» по «персональной подсказке» будет работать неполиномиально меньшее количество шагов), чем ограничивающий алгоритм базового алгоритма проверки («общий» ограничивающий алгоритм) задачи «Сфинкс».

1.3. И вернуть результат корректный «Эдип» должен будет полиномиально быстро по меркам времени получения подсказки, что должно позволить неким другим алгоритмам найти в рамках допустимого размера доказательство о результате работы «АнтиЭдипа», которое не нашёл (и не мог найти) «Эдип».

2.1. За основу алгоритма проверки возьмём теорию на базе Пеано, способную представлять произвольные алгоритмы. Базовый алгоритм проверки при допустимом размере доказательства проверяет стандартным алгоритмом проверки соответствие теоремы и доказательства. А вот какой размер доказательства будет допустимым в базовой проверке – это мы зададим специальными аргументами размера. Тогда «Сфинкс» для интересующих нас аргументов примет такой вид:

$\mathrm{Sph}(d, t, S, s, y)$

Где S – число, задающее максимальную длину доказательства. Число S записано в десятичном (например) виде, а s – соответствующая (иначе алгоритм проверки выдаст 0) данному числу S строка из символов «1» длиной S символов, например. Но длина строки s может быть равна, скажем, и $S^{1/10}$. То есть – корень десятой степени от S. Можно поставить корень любой фиксированной степени – всё равно размер $|s|$ будет неполиномиально больше размера числа S, записанного в стандартном позиционном (десятичном, к примру) виде.

Для чего нужна строка s? Для того, чтобы размер проверяемого доказательства имел какое-то полиномиальное соответствие с размером аргументов. В случае аналогичной задачи из класса $\mathbb{NP}^+$ все рассуждения оказываются значительно проще, а аргумент s исчезает из рассмотрения. Но будем разбираться сразу с задачей из класса $\mathbb{NP}$, а задача из класса $\mathbb{NP}^+$ будет

разобрана при этом автоматически – методом сокращения рассуждений, связанных с аргументом s.

Аргументы S, s будем называть «аргументы размера». Аргумент S будем называть «число из аргументов размера», а аргумент s – «строка из аргументов размера».

2.2. Условимся об обозначении – правильный аргумент s получается тогда, когда $s = s1(S)$. То есть, алгоритм $s1(S)$ генерирует строку из символов «1» нужного для корректного s размера. Если строка s отличается от результата $s1(S)$, то базовый алгоритм проверки задачи «Сфинкс» возвращает 0 (Ноль). Для базовой проверки значение аргумента долга будет – «объективно», то есть, равным символу $\overline{-}$.

3. Начнем реализацию плана, намеченного в пункте 1.1.

Рассмотри алгоритм «Эдип» с теоремой о результате работы алгоритма «АнтиЭдип», аргументом долга «объективно» (значение аргумента $\overline{-}$) и числом из аргументов размера S:

$\mathrm{Oed}(\overline{\mathrm{Sph}(...,...,...,...,...)}, \overline{-}, \overline{\mathrm{AntiOed}_S() = 0}, S, s)$

Для написанного алгоритма «Эдип» с приведёнными аргументами нам нужен такой алгоритм «АнтиЭдип», на основе которого будет сформирована теорема $\mathrm{AntiOed}_S() = 0$. Разумеется, для разных S будут разные «АнтиЭдипы». Алгоритм $\mathrm{AntiOed}_S()$ работает следующим образом:

3.1. Формирует текст теоремы

$\overline{\mathrm{AntiOed}_S() = 0}$

3.2. Запускает «Эдипа» со следующими аргументами:

$\mathrm{Oed}(\overline{\mathrm{Sph}(...,...,...,...,...)}, \overline{-}, \overline{\mathrm{AntiOed}_S() = 0}, S, s1(S))$

Притом в программном коде алгоритма «АнтиЭдип» строка из аргументов размера «Эдипа» генерируется из сжатого представления: Соответствующей переменной присваивается результат работы $s1(S)$, а уже значение переменной будет использоваться в качестве строки из аргументов размера при запуске «Эдипа». Таким образом, размеры программного кода алгоритма «АнтиЭдип» и теоремы $\mathrm{AntiOed}_S() = 0$ будут неполиномиально малы относительно строки s из аргументов размера «Эдипа» с нужными нам аргументами.

3.3. Ждет результата работы «Эдипа» и действует в зависимости от того, что выдаст «Эдип»:

3.4. Если Эдип возвращает 0 – «не могу найти доказательство», то «АнтиЭдип» возвращает в качестве результата 0 и останавливается, доказав делом теорему $\mathrm{AntiOed}_S() = 0$

3.5. Если «Эдип» возвращает какую-то строку в качестве доказательства, то «АнтиЭдип» пытается выделить результат доказательства – утверждение $\mathrm{AntiOed}_S() = 0$ в качестве последнего шага доказательства. И действует в зависимости от успеха такого выделения:

3.6. Если выделение не удалось, то теорема не доказана и исполняется пункт 3.4.

3.7. Если выделение удалось, то $\mathrm{AntiOed}_S()$ возвращает 1 и останавливается. Доказав делом, что теорему $\mathrm{AntiOed}_S() = 0$ нельзя доказать.

Замечание. В построении нашей задачи и обосновании её свойств мы сейчас неявно использовали непротиворечивость той теории, которая лежит в её основе (теория Пеано или подобная ей). Именно об этом и говорилось в разделе 3 «Сложность вопроса о $\mathbb{NP} \neq \mathbb{P}$»:

В противоречивой теории было бы доказано всё – в том числе доказательство чего угодно мог бы найти и «Эдип» – независимо от того, как бы работал «АнтиЭдип». И, кстати, принцип независимости алгоритма проверки был бы нарушен.

4.1 Обоснуем, что всегда можно построить вышеописанный алгоритм «АнтиЭдип» - $\mathrm{AntiOed}_S()$ для произвольного алгориттма «Эдип» и аргументов вида:

$\mathrm{Oed}(\overline{\mathrm{Sph}(...,...,...,...,...)}, \overline{-}, t, S, s)$

4.2. Все описанные в пункте 3 действия легко реализуются при помощи некого алгоритма-генератора в отношении такого алгоритма $\mathrm{AntiOed}_S()$, программный код которого был бы передан для обработки в аргументе X. Тогда легко сгенерировать текст утверждение $\overline{\mathrm{AntiOed}_S() = 0}$, другие соответствующие аргументы для запуска «Эдипа» и запустить сам «Эдип», а затем выдать свой результат после того, как «Эдип» выдаст свой (если «Эдип» остановится). Можно схематически (а можно и подробно – но это бездна рутины) написать программу для соответствующего алгоритма-генератора.

4.3. В соответствии с подразделом 5.2 «Теорема о построении алгоритма, применяемого к себе» мы можем из алгоритма-генератора предыдущего пункта сделать алгоритм, который все действия, которые алгоритм-генератор исполняет в отношении программного кода, получаемого в аргументе X – будет исполнять в отношении собственного программного кода. И, таким образом, мы описали метод (алгоритм) построения алгоритма $\mathrm{AntiOed}_S()$.

4.4. Если кто-то реально захочет строить $\mathrm{AntiOed}_S()$ в соответствии с методикой диагонализации, то пусть обратит внимание, что сначала надо построить алгоритм-генератор с одним аргуменом X. В этом алгоритме-генераторе потребуется задать в соответствующих переменных нужное S, программный код нужного «Эдипа» и программный код «Сфинкса». А уже в отношении этого алгоритма-генератора провести процедуру диагонализации относительно аргумента X. И нужный $\mathrm{AntiOed}_S()$ будет построен. Собственно, в названии алгоритма $\mathrm{AntiOed}_S()$ отражено и то, что он построен в отношении алгоритма «Эдип» и то, что работа данного «Эдипа» рассматривается тогда, когда числовая часть аргумента размера у него равна S.

4.5. Достаточно очевидно, что алгоритм «Эдип» с аргументами размера S, s не может доказать $\mathrm{AntiOed}_S() = 0$? Если s корректный ($s = s1(S)$), то $\mathrm{AntiOed}_S()$ построен так, чтобы опровергнуть доказательство «Эдипа». Если же s не корректный, то базовый алгоритм проверки у задачи «Сфинкс» по построению должен возвращать 0 (Ноль) – о некорректности доказательства при подобном некорректном аргументе.

5. Перейдём к реализации плана, намеченного в пункте 1.2.

Нам надо, чтобы алгоритм проверки «Сфинкс» сообщал о произвольном алгоритме-решении «Эдип» то, что этот алгоритм «Эдип» (с аргументами размера S и $s = s1(S)$) не может доказать теорему о результате работы алгоритма «АтниЭдип» (построенным для заданного S). Разумеется, сообщать об алгоритме «Эдип» алгоритм «Сфинкс» будет в том случае, если переданный программный код – это программный код алгоритма, а не какая-то абракодабра.

5.1. Воспользуемся формулой из пункта 2.1 подраздела 6 «Задачи Кука и авторизуемые задачи из класса $\mathbb{NP}$», добавив аргументы размера:

$\mathrm{Sph}(\overline{\mathrm{Oed}(\overline{\mathrm{Sph}(...,...,...,...,...)}, \overline{-}, ...,..., ...)}, t, S, s, y)$

Как видим, алгоритм «Сфинкс» получает в первом аргументе программный код алгоритма «Эдип» (произвольного, вообще говоря).

5.2. Если же в аргументе долга вместо корректного алгоритма с подставленным аргументом долга «объективно» передана какая-то абракодабра, то работа Сфинкса сводится к работе базового алгоритма проверки (с аргументом долга «объективно»). И именно результат базовой проверки тогда «Сфинкс» и вернёт в качестве своего результата.

5.3. Если же значение аргумента долга является подходящим под шаблон формулы пункта 5.1, то «Сфинкс» генерирует текст теоремы про результат работы алгоритма «АнтиЭдип», разобранной в подпунктах пункта 3 данного раздела. Алгоритм генерации схематически изложен в пунктах 4 текущего раздела. При этом потребуется использовать значение аргумента S (но не аргумента s – что существенно).

5.4. Сгенерированный текст теоремы сравнивается с тем значением, которое передано в аргументе t. Если тексты не совпадают, то работа алгоритма «Сфинкс» снова сводится к базовому алгоритму проверки – как и в пункте 5.2.

Если же тексты совпадают, то «Сфинкс» возвращает 0 – независимо ни от значения аргумента s, ни от значения аргумента y.

Тут имеется занятный момент отличия от модели машины Тьюринга – в отношении переменной s: она практически никак не используется в работе по «персональной подсказке». И не должна тормозить работу по выдаче «персональной подсказки», а она и не тормозит даже при обращении к «Сфинксу» из «Эдипа», если аргумент s передаётся «по ссылке». В реальном мире программа именно так и будет построена. И, кстати, этот нюанс учтён в способе обращения к «Сфинксу»:

$\mathrm{Sph}(\overline{\mathrm{Oed}(\overline{\mathrm{Sph}(...,...,...,...,...)},\overline{=},...,...,...)},t,S,s,y)$

Как видим, в значение аргумента долга переменная s не «встроена» и поэтому не становится частью существенной для анализа информации. В некотором смысле аргумент s становится таким же, как и аргумент y здесь – ни на что не влияющим. Впрочем, даже в классических задачах из $\mathbb{NP}$ аргумент y не влияет на время работы алгоритма проверки – как только размер аргумента $|y|$ превосходит некий полином от размера аргумента $|x|$.

Такой ответ «Сфинкса» будет совершенно правильным, потому что если значение аргумента s не соответствует значению аргумента S (если $s \neq s1(S)$), то такое сочетание аргументов отвергается и нет никакого доказательства для y, которое могло бы это исправить. Если же аргументы S, s соответствуют друг другу, то «Эдип» не в состоянии найти никакого корректного доказательства для теоремы:

$\mathrm{AntiOed}_S() = 0$

5.5. Разберём теперь время ответа «Сфинкса» при таком обращении к нему:

$\mathrm{Sph}(\overline{\mathrm{Oed}(\overline{\mathrm{Sph}(...,...,...,...,...)},\overline{=},...,...,...)},\overline{\mathrm{AntiOed}_S() = 0},S,s,y)$ – при этом результат работы известен – 0 (Ноль).

Как видно из подпунктов пункта 4 данного раздела, генерация текста теоремы $\overline{\mathrm{AntiOed}_S() = 0}$ и выдача результата 0 потребует от «Сфинкса» полиномиального количества шагов от вот чего: От размера аргумента долга $|\overline{\mathrm{Oed}(\overline{\mathrm{Sph}(\dots,\dots,\dots,\dots,\dots)},\overline{=},\dots,\dots,\dots)}|$, от размера аргумента $|S|$ (от количества десятичных цифр в десятичной записи S) и больше ни от чего. Размер теоремы не имеет значения – она же всё равно генерируется и сравнивается затем с тем, что передано в аргументе t.

Размер аргумента долга в данном случае полиномиально зависит от размеров программных текстов $\mathrm{Sph}(\dots,\dots,\dots,\dots,\dots)$ и $\mathrm{Oed}(\dots,\dots,\dots,\dots,\dots)$. Но они фиксированы для разбираемых нами подзадач (в отношении заданных «Сфинкса» и «Эдипа») общей задачи. И, таким образом, время работы «Сфинкса» в интересующем нас случае будет ограничено полиномом от размера $|S|$. То есть, время работы «Сфинкса» в интересующем нас случае не будет превышать

некоторого $v * |S|^u$, где v, u – фиксированные числа.

5.6. Таким образом, мы выполнили план из пункта 1.2: «Сфинкс» выдаёт «персональную подсказку» про алгоритм «Эдип». «Персональная подсказка» - 0 (Ноль), то есть, корректный «Эдип» должен вернуть 0 (Ноль) – «Не могу найти доказательство для утверждения $\mathrm{AntiOed}_S() = 0$».

Выдаётся «персональная подсказка» алгоритмом «Сфинкс» за время, полиномиальное относительно $|S|$ (если зафиксировать размеры «Сфинкса» и «Эдипа»). В то же время «общий» ограничивающий алгоритм – полином относительно размера $|s|$.

Соотношение между $|S|$ и $|s|$ - как между линейной зависимостью и экспонентой. И, значит, любые полиномы на их базе будут неполиномиально сильно отличаться между собой – полином на базе $|S|$ будет неполиномиально меньше полинома на базе $|s|$.

6. Теперь начнем реализацию заключительного этапа нашего плана – пункта 1.3. Хотя уже сейчас из пункта 5.6 довольно очевидно, что корректный «Эдип» должен будет дать очень быстрый ответ по меркам «общего» ограничивающего алгоритма. А выдав такой ответ быстро – он сделает и ответ «АнтиЭдипа» очевидным, в силу чего доказательство о результате работы «АнтиЭдипа» окажется достаточно небольшим для того, чтобы не превысить величину S (или размер $|s|$ - что то же самое). И тогда будет и время на ответ «Эдипа» - «не могу найти доказательство», и само доказательство допустимого размера, которое «Эдип» не смог найти. То есть – будет доказана неполнота корректного «Эдипа». А так как «Эдип» - произволен, то будет доказано и $\mathbb{NP} \neq \mathbb{P}$ – неполнота разбираемой задачи из класса $\mathbb{NP}$ с точки зрения её сводимости к задаче из класса $\mathbb{P}$ силами произвольного алгоритма.

6.1. Допустим, что алгоритм «Эдип» в интересующем нас случае остановился и вернул результат 0. Любой другой результат был бы ошибочным в случае остановки, как мы видели. А не-остановка тоже была бы ошибкой, так как информация о невозможности решить «Эдипу» интересующую нас задачу имеется – достаточно запросить «Сфинкса» об этом.

Ещё один момент – в интересующем нас случае $s = s1(S)$, так как в противном случае правильный результат – 0 (Ноль) и мы не ищем в этом случае неполноты «Эдипа».

Вопрос: Какой размер доказательства будет после такой остановки у интересующей нас теоремы:

$\mathrm{AntiOed}_S() = 0$

6.2. Понятно, что алгоритм $\mathrm{AntiOed}_S()$ тоже остановится в случае остановки «Эдипа» - так построен алгоритм $\mathrm{AntiOed}_S()$, что он завершает свою работу после остановки «Эдипа» с соответствующими параметрами.

Доказательство того, что если остановится «Эдип» в интересующем нас случае, то остановится и «АнтиЭдип» будет полиномиального размера относительно размера «Эдипа», даже независимо от размера $|S|$ - пока вместо конкретного значения S рассматривается некая переменная:

$\mathrm{Oed}(\overline{\mathrm{Sph}(...,...,...,...,...)}, \overline{-}, \overline{\mathrm{AntiOed}_S() = 0}, S, s1(S)) = 0 \Rightarrow \mathrm{AntiOed}_S() = 0$

Но при подстановке S для конкретного значения зависимость появится, при этом исполнять $s1(S)$ не придётся. Например, для формулы:

$\mathrm{Oed}(\overline{\mathrm{Sph}(...,...,...,...,...)}, \overline{-}, \overline{\mathrm{AntiOed}_{100000}() = 0}, 100000, s1(100000)) = 0 \Rightarrow \mathrm{AntiOed}_{100000}() = 0$

Размер доказательства данной формулы будет полиномиального размера относительно размера «Сфинкса» и 6 (шести - количества цифр в 100000).

6.3. Итак – какого размера будет доказательство следующего выражения, в которое подставлено конкретное S, но не исполнено ещё $s1(S)$:

$\mathrm{Oed}(\overline{\mathrm{Sph}(...,...,...,...,...)}, \overline{-}, \overline{\mathrm{AntiOed}_S() = 0}, S, s1(S)) = 0$

Размер доказательства данного выражения надо оценить относительно его размера (включая подставленное конкретное значение S) и количества шагов работы алгоритма «Эдип» из данного выражения, если «Эдип» завершает свою работу через n шагов и в это число n не входит время работы $s1(S)$.

Разумеется, мы можем «перевести» каждый шаг работы программы в несколько шагов доказательства, полиномиально зависящих от размера программы и размера занятой переменными и аргументами памяти. Но работу конкретно алгоритма $s1(S)$ таким образом «переводить» в доказательство нет никакой нужды. Так как результат работы $s1(S)$ заранее известен.

6.3.1. Для каждой ячейки памяти аргумента s, которая получает своё значение после исполнения $s1(S)$, нам известно как её значение (символ «1»), так и количество соответствующих ячеек (их количество - S). И в тексте доказательства для разбираемого выражения, каждый шаг работы алгоритма «Эдип», в которой он делает операцию с одной из ячеек памяти аргумента s, будет «переведён» в полиномиальное количество шагов доказательства. Притом «полиномиальное количество» - это относительно вот чего:

размера $|\,\mathrm{Oed}(\overline{\mathrm{Sph}(...,...,...,...,...)}, \overline{-}, \overline{\mathrm{AntiOed}_S() = 0}, S, s1(S))|$, величины n.

Надо пояснить, что после того, как $s1(S)$ исполнен, состояние программы пришло в «исходное положение», аналогичное запуску алгоритма с готовыми аргументами (так мы строим подстановку аргументов):

$\mathrm{Oed}(\overline{\mathrm{Sph}(...,...,...,...,...)}, \overline{-}, \overline{\mathrm{AntiOed}_S() = 0}, S, s)$

И переход из этого состояния в последующие будет полностью задаваться номером очередного шага $i \leq n$, от которого и будет полиномиально зависеть размер шагов доказательства, соответствующих этому шагу работы. Размер самого аргумента $|s|$ при этом может не иметь никакого значения для размера шагов доказательства о данном шаге работы – ведь программа всё равно не может обратиться к числу ячеек переменной s, превышающему само n и имеет дело с той частью аргумента s, ячейки которого доступны через «индекс» $i \leq n$.

Сделанное замечание остаётся в силе и в том случае, если учитывать ограниченность скорости света и отчасти использовать модель «машины Тьюринга» - тогда добраться до конкретной ячейки памяти надо за несколько шагов работы программы, но тогда тем более количество использованных в процессе работы ячеек окажется меньше n.

И каждый шаг доказательства в отношении работы с ячейкой памяти s по своему размеру (количеству символов) сам будет полиномиальным от данных величин. Значит, размер всех таких шагов доказательства (количество символов в них) будет полиномиальным относительно тех же размеров. Ведь таких шагов – меньше n. «Меньше» - потому что не все шаги работы программы будут операциями с аргументом s.

6.3.2. Таким образом, времени работы алгоритма $s1(S)$ в нашем доказательстве не соответствует ничего, сопоставимого по своим размерам со временем его работы. Но какое-то небольшое количество предварительных шагов доказательства, соответствующее заполнению аргумента s алгоритмом $s1(S)$, в доказательстве будет сделано для дальнейшего применения в пункте 6.3.1. Будем считать – тоже полиномиальное количество символов у этой «предварительной» части доказательства (хоть это и с запасом – n лишнее, видимо) от

размера $|\mathrm{Oed}(\overline{\mathrm{Sph}(...,...,...,...,...)},\overline{-},\overline{\mathrm{AntiOed}_S()=0},S,s1(S))|$, величины n.

6.3.3. Остальные шаги работы алгоритма

$\mathrm{Oed}(\overline{\mathrm{Sph}(...,...,...,...,...)},\overline{-},\overline{\mathrm{AntiOed}_S()=0},S,s1(S))$

обычным образом «переводятся» в шаги доказательства о работе данного алгоритма – вплоть до его остановки. Размер (количество символов) каждого такого шага доказательства полиномиально относительно:

размера $|\mathrm{Oed}(\overline{\mathrm{Sph}(...,...,...,...,...)},\overline{-},\overline{\mathrm{AntiOed}_S()=0},S,s1(S))|$, величины n.

Количество шагов доказательства на каждый шаг работы алгоритма – полиномиальное относительно того же, а само количество шагов программы – кроме тех, что разбирались в пунктах 6.3.1 и 6.3.2 – не больше n (меньше n, если была хоть одна операция «Эдипа» с аргументом s). Таким образом, размер всего доказательства (количество символов) помимо того, что рассмотрено в пунктах 6.3.1 и 6.3.2 – полиномиально относительно:

размера $|\mathrm{Oed}(\overline{\mathrm{Sph}(...,...,...,...,...)},\overline{-},\overline{\mathrm{AntiOed}_S()=0},S,s1(S))|$, величины n.

6.3.4. Просуммировав пункты 6.3.1, 6.3.2 и 6.3.3 выясняем, что размер доказательства выражения

$\mathrm{Oed}(\overline{\mathrm{Sph}(...,...,...,...,...)},\overline{-},\overline{\mathrm{AntiOed}_S()=0},S,s1(S))=0$

в случае остановки алгоритма из данного выражения с результатом 0, будет иметь размер полиномиальный относительно:

размера $|\mathrm{Oed}(\overline{\mathrm{Sph}(...,...,...,...,...)},\overline{-},\overline{\mathrm{AntiOed}_S()=0},S,s1(S))|$, величины n.

Где n – количества шагов работы алгоритма «Эдип» из разбираемого выражения и в это число n не входит время работы алгоритма $s1(S)$.

6.4. С учётом пункта 6.2, предыдущего пункта, правила вывода MP мы имеем такие формулы и результат:

$\mathrm{Oed}(\overline{\mathrm{Sph}(...,...,...,...,...)},\overline{-},\overline{\mathrm{AntiOed}_S()=0},S,s1(S))=0\Rightarrow\mathrm{AntiOed}_S()=0$

$\mathrm{Oed}(\overline{\mathrm{Sph}(...,...,...,...,...)},\overline{-},\overline{\mathrm{AntiOed}_S()=0},S,s1(S))=0$

$\mathrm{AntiOed}_S()=0$

Теперь можем дать такой первоначальный ответ на вопрос пункта 6.1 о размере доказательства для теоремы:

$\mathrm{AntiOed}_S()=0$

Для данной теоремы в случае остановки алгоритма

$\mathrm{Oed}(\overline{\mathrm{Sph}(...,...,...,...,...)},\overline{-},\overline{\mathrm{AntiOed}_S()=0},S,s1(S))$ с результатом 0,

доказательство будет иметь размер полиномиальный относительно:

размера $|\mathrm{Oed}(\overline{\mathrm{Sph}(...,...,...,...,...)},\overline{-},\overline{\mathrm{AntiOed}_S()=0},S,s1(S))|$, величины n.

Где n – количество шагов работы алгоритма «Эдип» – $\mathrm{Oed}(\overline{\mathrm{Sph}(...,...,...,...,...)},\overline{-},\overline{\mathrm{AntiOed}_S()=0},S,s$ – и в это число n не входит время работы алгоритма $s1(S)$.

Разумеется, мы лишь оцениваем размер доказательства – мы всегда можем написать доказательство «не больше по размеру чем. . . ». Вполне могут быть и гораздо меньшие доказательства, но нам хватит и полученной сейчас оценки.

6.5. Осталось оценить величину n относительно $|S|$ и $|s|$, чтобы понять, помещается ли доказательство разбираемой теоремы в пределы допустимого размера доказательства S при корректной работе алгоритма «Эдип».

Вспомним последний абзац из пункта 2.4 разделе 6 «Задачи Кука и авторизуемые задачи из класса $\mathbb{NP}$»:

Если для «персональной подсказки» алгоритма «Сфинкс» о возможностях алгоритма «Эдип» имеется ограничивающий полином $p_0(|x1|, |x2|, ...)$ на время получения подсказки, то «Эдип», способный сводить задачи из класса $\mathbb{NP}$ к задачам из класса $\mathbb{P}$, найдёт правильный ответ (если он есть) или выдаст 0 (если «персональная подсказка» не содержит правильного ответа) за время, ограниченное некоторым полиномом $p(|x1|, |x2|, \ldots)$.

Мы рассматриваем именно такие «Эдипы», о которых говорится в процитированном абзаце. Но из пункта 5.5 известно, что время работы «Сфинкса» для интересующих нас случаев ограничено полиномом $v * |S|^u$, где v, u – фиксированные числа. Значит, корректный «Эдип», который способен сводить задачи из класса $\mathbb{NP}$ к задачам из класса $\mathbb{P}$ должен выдать свой ответ за время, ограниченное неким полиномом $p(|S|)$. А тогда найдётся и ограничивающий полином для такого времени вида $w * |S|^r$, где w, r – фиксированные числа.

6.6. Итак, для корректного алгоритма-решения:

$\mathrm{Oed}(\overline{\mathrm{Sph}(..., ..., ..., ..., ...)}, \overline{-}, \overline{\mathrm{AntiOed}_S() = 0}, S, s)$, где $s = s1(S)$,

ограничивающий полином для времени n работы такого алгоритма будет равен $w * |S|^r$, где w, r – фиксированные числа. Разумеется, это один из бесконечного количества ограничивающих полиномов для данного времени работы.

После того, как мы определились с ограничением на величину n, мы можем переписать пункт 6.4 таким образом:

Для теоремы: $\mathrm{AntiOed}_S() = 0$ составленной для корректного алгоритма «Эдип», который способен сводить задачи из класса $\mathbb{NP}$ к задачам из класса $\mathbb{P}$ доказательство будет иметь размер, не превышающий полином $k * |S|^q$, где k, q – фиксированные числа, если фиксированы программы «Сфинкс» и «Эдип».

Каким образом, размер доказательства, ограниченный полиномом $k * |S|^q$, соотносится с предельным размером доказательства, заданным аргументом S? Соотношение тут – неполиномиальное. При росте S полином $k * |S|^q$ обязательно станет меньше S (сравнивается число и полином от количества десятичных цифр в нём), а это и будет означать неполноту «Эдипа», вернувшего 0 (Ноль) в отношении той теоремы, для которой имеется доказательство в пределах «общего» ограничивающего алгоритма.

Но и не возвращать ответ для «Эдипа» было бы ошибочным, потому что задача построена именно так, чтобы найти доказательство он не мог.

7. Поскольку неполнота «Эдипа» доказана для произвольного алгоритма-решения задачи «Сфинкс», то нет ни одного алгоритма, который мог бы свести построенную нами авторизуемую задачу «Сфинкс» из класса $\mathbb{NP}$ к какой-то задаче из класса $\mathbb{P}$. Поэтому неравенство $\mathbb{NP} \neq \mathbb{P}$ доказано.

8 Приложения

8.1 Первая теорема Гёделя о неполноте

Смысл первой теоремы Гёделя следующий: «Это утверждение нельзя доказать в рамках данной теории». Утверждается, что внутри непротиворечивой теории доказать нельзя не только данное утверждение, но и его отрицание. Впрочем – насчёт «но» мы ещё разберём один нюанс.

Гёдель рассматривал предикат доказуемости. За основу он брал (обозначения тут не бук-

вально как у Гёделя) предикат $P_0(x, y)$, который имел простое доказательство, если некое утверждение с текстом в аргументе доказано текстом из аргумента y. А если это не так, тогда имеется простое доказательство для отрицания данного выражения, то есть – доказано $\neg P_0(x, y)$.

Я изложу чуть иначе – $P_0(x, y)$ будет алгоритмом, который возвращает 1, если y доказывает x, и возвращает 0 в противном случае. Смысл от этого не меняется, а наглядность увеличивается – и это ближе к нашей теме из теории алгоритмов. А главное – в следующем подразделе это сильно упростит нам доказательство 2-й т. Гёделя о неполноте.

Тогда выражение $\exists y(P_0(x, y) = 1)$ будет обозначать доказуемость утверждения x. Сокращенное обозначение такое: $P(x)$. А у недоказуемости обозначение будет таким: $\neg P(x)$. И это уже – предикаты, а не алгоритмы.

И нам надо применить формулу $\neg P(x)$ к самой себе – как мы это рассматривали в Лемме о диагонализации (хоть она у нас об алгоритмах, но для предикатов доказывается то же самое практически, и изложено в учебниках):

$\neg P(diag(\overline{\neg P(diag(x))}))$, это и есть формула из 1-й теорема Гёделя о неполноте. Из данной формулы мы выделим «это утверждение» G, текст которого получается так: $diag(\overline{\neg P(diag(x))})$. Вот теперь утверждение $\neg P(\overline{G})$ и обозначает, что «нельзя доказать» ($\neg P(...)$) «это» (G) утверждение. В соответствии с леммой о диагонализации получим (вместо равенства – будет эквивалентность для предикатов):

$\neg P(\overline{G}) \Leftrightarrow G$

Если предположить, что утверждение G доказано, то в силу предыдущей эквивалентности доказано и $\neg P(\overline{G})$. И – доказано $P(\overline{G})$. Последнее доказано потому, что у нас же есть текст для доказательства G – пусть k, значит $P_0(\overline{G}, k) = 1$ и по свойствам квантора существования будет $\exists y(P_0(\overline{G}, y) = 1)$. А последнее и есть $P(\overline{G})$. А доказанные утверждения $\neg P(\overline{G})$ и $P(\overline{G})$ образуют противоречие. Поэтому в непротиворечивой теории доказать G нельзя.

Доказывать вторую часть (про недоказуемость $\neg G$) 1ой т. Гёделя теоремы – не обязательно, потому что для наших целей нужна 2-я т. Гёделя, а чтоб доказать 2-ю теорему – нам хватит и первой части 1ой теоремы. Однако ради целостности исторической картины хотя бы популярно разберём и вторую часть 1ой т. Гёделя о неполноте.

Допустим, у нас есть доказательство для $\neg G$ с текстом k. Тогда в непротиворечивой теории у нас нет доказательств для G, нет, соответственно, никаких таких q, что $P_0(\overline{G}, q) = 1$. И – вот здесь серьёзный нюанс! – нет доказательства для $\exists y(P_0(\overline{G}, y) = 1)$.

Вот переход от того, что в теории нет никаких значений q, при которых доказано выражение $B(q)$ к тому, что тогда и выражение $\exists y B(y)$ не доказано – это свойство называется «омега-непротиворечивость». Оно кажется очевидным, но оно есть не у всех непротиворечивых теорий.

Если есть омега-непротиворечивость – то есть и просто непротиворечивость, потому что при омега-непротиворечивости что-то недоказуемо, а в противоречивой теории доказано всё. Но вот обратное – что непротиворечивая теория ещё и омега-непротиворечивая – не всегда верно.

Так вот, если омега-непротиворечивость есть, то нет доказательства для $\exists y(P_0(\overline{G}, y) = 1)$, то есть – нет доказательства для $P(\overline{G})$. Но у нас же доказано $\neg G$, у нас же есть эквивалентность $\neg P(\overline{G}) \Leftrightarrow G$, и, следовательно, $P(\overline{G}) \Leftrightarrow \neg G$. А из последней эквивалентности и $\neg G$ вытекает $P(\overline{G})$. Но это входит в противоречие с тем, что у нас нет доказательства для $P(\overline{G})$.

Первая теорема Гёделя доказана.

Хоть 1-я теорема Гёделя и требовала омега-непротиворечивости, но математическое сообще-

ство признало, что мечты о полноте провалились. Программа Гильберта тогда показала свою эффективность и принесла важнейшие результаты.

К тому же, прикладных математиков не интересуют теории омега-противорчивые – там даже предикат доказуемости может неправильно работать. И теория Пеано является (как и прочие прикладные теории) омега-непротиворечивой. Интересные рассуждения (и отсылки к главам с доказательствами) о внешней непротиворечивости можно почитать у Д. Гильберта, П. Бернайса «Основания математики», Том 2, Глава 5 «Выход за рамки теории доказательств», Раздел 1 «Границы изобразимости и выводимости», Подраздел б) «Первая теорема Гёделя о неполноте» - в конце подраздела. А ранее середины подраздела сформулировано и понятие омега-непротиворечивости. Тема довольно принципиальная, кстати, хоть и подзабытая – думаю, временно.

Омега-непротиворечивость – это про то, что если некое утверждение недоказуемо в частном (при подстановке вместо x любого конкретного числа), то недоказуемо и в общем (в смысле «существования»). А внешняя непротиворечивость – это про то, что если доказуемо в частном, то потенциально доказуемо и в общем (для «любого»). И оба случая «особой» непротиворечивости включают в себя и «обычную» непротиворечивость.

Притом «потенциально доказуемо» означает, что расширив непротиворечивую теорию аксиомой-обобщением частных случаев, мы получим опять непротиворечивую теорию.

Что же касается омега- и внешней непротиворечивости, то в учебниках даётся доказательство «обычной» непротиворечивости, а доказательство внешней непротиворечивости приходится додумывать из доказательств непротиворечивости самостоятельно.

Омега-непротиворечивость не очень интересна, так как чуть ниже будет приведена первая теорема Гёделя в форме Россера, где удаётся обойтись обычной непротиворечивостью, а вот для одного дополнительного доказательства после 2ой теоремы Гёделя нам потребуется внешняя непротиворечивость.

Поэтому дам пояснения:

Если почитать доказательство непротиворечивости теории Пеано в Э.Мендельсон «Введение в математическую логику» - то там сразу получается и доказательство внешней непротиворечивости. Дело в том, что теория-инструмент S_∞ включает в себя правило бесконечной индукции: Когда предикат $A(n)$ истинный для каждого числа n, то выводится $\forall x A(x)$.

Для S_∞ выводятся все те замкнутые формулы , которые выводятся для теории S (теории Пеано). То есть – если бы в S выводилось существование чего-то, то – выводилось бы и в S_∞. Но существование $\exists x \neg A(x)$ противоречиво в S_∞ для того $A(n)$, который верен для каждого n: Так как в силу бесконечной индукции будет выведено «для любого» $\forall x A(x)$, что является отрицанием (противоречит) для $\exists x \neg A(x)$. Поэтому, если теория S_∞ является просто непротиворечивой (а она является), то она является и внешне непротиворечивой. А раз в S_∞ не доказываются какие-то абсурдные утверждения с квантором существования, то тем более не доказаны они и в теории Пеано – а значит и там имеет место внешняя непротиворечивость.

Интересно, что квантор существования – это отрицание квантора «для любого» и напрямую из бесконечной индукции не выводится. Но дело в том, что вывод из S в «Введение в математическую логику» опирается только на правила вывода MP (из импликации и посылки выводится следствие) и Gen (из истинного $A(x)$ выводится $\forall x A(x)$). А квантор существования определяется через квантор «для любого». Теория получается эквивалентной тому, что

и в «Основаниях математики» – хотя в «Основаниях математики» есть свои правила вывода для кванторов «существует» и «для любого». В этом смысле подход Э.Мендельсона удобен для использования S_∞ – разбираться приходится только с одним квантором, а второй (квантор существования) не имеет самостоятельного значения.

И вот при правилах вывода MP и Gen выводы опираются лишь на то, что в состоянии сделать и S_∞. Поэтому ничего сверх того, что может делать S_∞, от неё не требуется для того, чтобы повторить доступное для вывода в S. Отсюда, кстати видно, что для внешней противоречивости потребовалось бы некая аксиома с квантором существования или нечто эквивалентное. А то, что имеется в Пеано, не содержит ничего подобного аксиоме о существовании небывалого.

Позже Б. Россер так построил формулировку 1-й теоремы Гёделя о неполноте, что она не требовала омега-непроиворечивости при доказательстве. Не углубляясь в формулы, 1-я теорема Гёделя в форме Россера такая:

«Если это утверждение доказано в данной теории, то в данной теории доказано его отрицание и длина доказательства отрицания не превышает доказательство данного утверждения».

Очевидно, что само утверждение Россера без противоречий нельзя доказать – противоречие (отрицание) будет найдено в заранее известном ограниченном наборе доказательств. А если и не будет найдено – то это тоже будет противоречием.

Отрицание Россера будет конъюнкцией 2х членов (отрицание импликации):

1. Доказанное утверждение Россера (опять оно!)

2. Отсутствие доказательства отрицания Россера в пределах известной длины.

Но само утверждение Россера (член 1) включает в себя требование наличия доказательства своего отрицания в пределах известной длины. А раз утверждение доказано (член 1), то и его требование истинно. И это истинное требование входит в противоречие с членом 2.

В строгом доказательстве нужна аккуратность, чтобы проделать всё это под кванторами, но кванторы охватывают ограниченный (а не бесконечный) круг объектов и доказательство всё равно получается небольшое. Подробности, например, в Э. Мендельсон, «Введение в математическую логику», Глава 3 «Формальная арифметика», раздел 5 «Теорема Гёделя для теории S». Или в указанном чуть выше подразделе «Оснований математики», но там «в нагрузку» решается много дополнительных вопросов.

8.2 Вторая теорема Гёделя о неполноте

За доказательством 2-й т. Гёделя о неполноте много лет принято отсылать к двухтомнику Д. Гильберта, П. Бернайса «Основания математики». Или приводить свойства для предикатов доказуемости и уже на их базе доказывать теорему. Но с предикатами доказуемости без названного двухтомника не понятно, почему в одном свойстве имеет место выводимость, а в другом – импликация.

«Основания математики» – отличная книга, но там, в основном, даны сведения для профессионалов, которые не являются обязательными для принципиального понимания доказательства теорем Гёделя. А теоремы Гёделя стали уже чуть ли не частью культурного кода широкого слоя непрофессионалов. Поэтому пора дать компактное доказательство всех принципиальных моментов 2-й теоремы Гёделя.

Так вот, идея 2-й т. Гёделя в том, что утверждение G из 1-й т. Гёделя в случае своего

доказательства позволяет доказать и противоречие. В качестве «любимого» у математиков обычно выступает противоречие $1 = 0$, и мы не будем нарушать традицию.

То есть, из доказуемости G следует доказуемость $1 = 0$. Это импликация (вот тут принципиальный момент, который надо будет раскрыть в доказательстве) $P(\overline{G}) \Rightarrow P(\overline{1 = 0})$. Из импликации по принципу контрапозиции получим $\neg P(\overline{1 = 0}) \Rightarrow \neg P(\overline{G})$. Из последнего выражения вместе с свойством для G:

$\neg P(\overline{G}) \Leftrightarrow G$, получим:

$\neg P(\overline{1 = 0}) \Rightarrow G$

Таким образом, если бы в теории была доказана непротиворечивость, то было бы доказано и утверждение Гёделя из 1-й теоремы Гёделя, а это означает противоречивость. Как и было обещано выше в 1-й т. Гёделя – нам потребовалась только 1-я часть 1-й т. Гёделя для доказательства 2-й. Итак:

Вторая теорема Гёделя о неполноте.

Если в рамках теории имеется доказательство её непротиворечивости, то теория противоречива.

Доказательство.

Начиная с выражения $P(\overline{G}) \Rightarrow P(\overline{1 = 0})$ у нас имеется вывод до конца доказательства – см. выше. Значит, нам осталось доказать только $P(\overline{G}) \Rightarrow P(\overline{1 = 0})$.

Тут мы и воспользуемся ради упрощения тем, что $P_0(x, y)$ у нас – алгоритм, а не предикат. И за счет этого вместо импликаций будем иметь дело на начальном этапе с равенствами, что понятно в наше время очень многим (программистам, например). А затем по теореме Дедукции перейдём к импликации.

Если у нас есть доказательство k для G, то мы можем достроить вывод до противоречия и получить в результате доказательства $1 = 0$. Это значит, что есть алгоритм $f(k)$, который вернет по k доказательство для $1 = 0$. И тогда $P_0(\overline{1 = 0}, f(k))$ вернет 1. Запишем это в виде равенства:

$P_0(\overline{1 = 0}, f(y)) = if[P_0(\overline{G}, y), 1, ...]$

Где алгоритм $if[x1, x2, x3]$ – вычисляет результат работы алгоритма $x1$, и если он вернет 1, то $if[x1, x2, x3]$ возвращает результат работы алгоритма $x2$, а иначе возвращает результат работы $x3$. На месте многоточия (место $x3$) стоит $P_0(\overline{1 = 0}, f(y))$ – но не будем его писать ради краткости. Равенство правильное, потому что результат работы правой сторона равенства отличается от результата работы левой стороны $P_0(\overline{1 = 0}, f(y))$ только если в переменой y – доказательство для G, но в этом случае левая часть равенства $P_0(\overline{1 = 0}, f(y))$ тоже вернет 1 – как и правая часть равенства.

Начнем процедуру из теоремы Дедукции. В качестве гипотезы берём $P_0(\overline{G}, y) = 1$. Теперь можем считать, что это равенство истинное, можем пользоваться этим, но в конце процедуры должны будем поставить эту гипотезу в качестве посылки импликации, а заключением будет результат процедуры.

Из $P_0(\overline{1 = 0}, f(y)) = if[P_0(\overline{G}, y), 1, ...]$ и Гипотезы $P_0(\overline{G}, y) = 1$:

$P_0(\overline{1 = 0}, f(y)) = if[1, 1, ...]$. Так как по свойствам if[x1, x2, x3] имеем if[1, 1, ...] = 1, то:

$P_0(\overline{1 = 0}, f(y)) = 1$

Завершаем процедуру из теоремы Дедукции и получаем истинную импликацию:

$(P_0(\overline{G}, y) = 1) \Rightarrow (P_0(\overline{1 = 0}, f(y)) = 1)$. Поменяем переменную, чтобы использовать переменную y с кванторами:

$(P_0(\overline{G}, k) = 1) \Rightarrow (P_0(\overline{1 = 0}, f(k)) = 1)$.

Воспользуемся аксиомой для квантора существования (Д. Гильберт, П. Бернайс «Основания математики» Том 1, Глава 4 «Исчисление предикатов», Параграф 2 «Связанные переменные и правила для кванторов», Раздел 4 «Окончательная формулировка правил исчисления предикатов») и запишем аксиому для частного случая:

$(P_0(\overline{1 = 0}, f(k)) = 1) \Rightarrow (\exists y P_0(\overline{1 = 0}, y) = 1)$. Из 2х последних формул перейдем по правилу силлогизма к:

$(P_0(\overline{G}, k) = 1) \Rightarrow (\exists y P_0(\overline{1 = 0}, y) = 1)$, Используя правило вывода для квантора существования (см. вышеуказанный раздел в Д. Гильберт, П. Бернайс «Основания математики») получим:

$(\exists y P_0(\overline{G}, y) = 1) \Rightarrow (\exists y P_0(\overline{1 = 0}, y) = 1)$, теперь перепишем это, используя принятое сокращение $P(x)$ для $(\exists y P_0(x, y) = 1)$:

$P(\overline{G}) \Rightarrow P(\overline{1 = 0})$

Последняя формула и есть то, что нам оставалось доказать для завершения доказательства Второй теоремы Гёделя о неполноте.

Теорема доказана.

Теперь докажем, что недоказуема и $P(\overline{1 = 0})$. То есть – что внутри непротиворечивой теории нет доказательства для отрицания утверждения о недоказуемости противоречия в данной непротиворечивой теории. Тогда $\neg P(\overline{1 = 0})$ будет недоказуемо вместе со своим отрицанием и Вторая теорема Гёделя окажется вариантом Первой теоремы Гёделя о неполноте. Для доказательства нам потребуется «внешняя непротиворечивость», которую мы рассмотрели и сделали обзор о её доказательстве в предыдущем подразделе.

Если задуматься, то было бы странно, если бы в непротиворечивой теории было доказано утверждение о существовании в ней доказательства противоречия. Очевидно, что у нас есть недоказуемость противоречия для каждого конкретного доказательства k:

$P_0(\overline{1 = 0}, k) = 0$, откуда:

$\neg(P_0(\overline{1 = 0}, k) = 1)$

В силу внешней непротиворечивости у нас «потенциально доказано» и

$\forall x \neg(P_0(\overline{1 = 0}, x) = 1)$, откуда «потенциально доказано»:

$\neg\exists x(P_0(\overline{1 = 0}, x) = 1)$, откуда «потенциально доказано»:

$\neg P(\overline{1 = 0})$

Это всё означает, что теория Пеано (да и любая достаточно выразительная внешне непротиворечивая теория) может быть расширена аксиомой о непротиворечивости исходной (до расширения) теории. И никаких противоречий не возникнет.

А это значит, что и в исходной (до расширения) теории не было доказано $P(\overline{1 = 0})$, что, собственно, мы и доказывали. Значит, во внешне непротиворечивой теории недоказуемо ни утверждение о её (теории) непротиворечивости, ни отрицание этого утверждения.

8.3 Рефлексия. То, что все хотели знать о дилемме заключённого, но боялись спросить

В проблеме из теории игр «дилемма заключенного» ставится вопрос о выгодах и невыгодах того или иного поведения заключенного в отношениях с тюремным начальством и сокамерниками. Задачу надо решить при неясном уровне злопамятности/доброты тех и других.

Но в математике не рассматривался пока самый главный вопрос в этой проблеме: Как заключенный отличает себя от сокамерников и тюремного начальства?

Как отличие «себя» от другого выразить математически? Заметим, что задачи с «персонализацией» - самые распространенные в нашей жизни. Мы планируем свои поступки, сообщаем о своей жизни другим людям, нам дают поручения, нам сообщают о причитающихся нам поощрениях или наказаниях и мы не путаем себя с другими. По крайней мере, обычно не путаем, хотя иногда кто-то и путает себя – с Наполеоном, например.

И понимание «себя» - стандартное негласное требование к нам при решении задач. Например, мне дают поручение снести направленным взрывом три дома. Задача кажется независимой от исполнителя: Закладываю взрывчатку в каждый дом, кручу взрыватель для сноса первого, затем второго и третьего. Где тут «я»? «Я» тут имеется – потому что надо учесть – а сам-то я не нахожусь в одном из этих домов? Заложил взрывчатку в 1й, затем в 3й, затем во 2й и – стоя во втором, сношу 1й, затем 2й... упссс – 3й я уже не снесу. И планы будут сорваны.

И вот, когда мы разбирали работу «Сфинкса» и «Эдипа» в предыдущих разделах – мы как раз имели дело со случаем, когда результат работы «субъекта» влияет на последствия для него самого. И за счёт этого «Эдип» не смог «пересчитать» «Сфинкса», а класс $\mathbb{NP}$ оказался несводимым к классу $\mathbb{P}$.

То есть, теперь у нас есть формализм, который позволяет учитывать влияние действий алгоритма на него самого и ставить задачи так, чтобы они отражали зависимость правильности своего решения от того, кто (какой алгоритм, человек, робот и т.д.) это решение реализует.

И появившаяся принципиальная возможность рассматривать «я», рефлексию – даёт доступ к анализу и таких вопросов, как жизнь и смерть разумного субъекта. Если, конечно, считать, что тезис Чёрча верен, и к нам можно подходить с теми же мерками, которые применяются к алгоритмам. Исследование подобных вопросов математическими методами назову «рефлексивной логикой». И это будет отчасти естественной наукой, потому что в обычной жизни мы совершаем многое, опираясь на рефлексию, и эти действия происходят независимо от того, рассмотрено это математически или нет.

Опять же, в нашем примере со «Сфинксом» и «Эдипом»: «Эдип» в принципе не может найти результат соответствующего ему «АнтиЭдипа», но вернув результат «не знаю» он может исполнить «требование» «Сфинкса» – если остановился с правильным для него («Эдипа») результатом «не знаю». Тут мы видим пример «поведения», основанного на исполнении некого требования. То есть – для «Эдипа» информация от «Сфинкса» носила нормативный, а не дескриптивный характер. Теперь у нас есть формализм, объясняющий связь между нашей «свободной» волей и подчинением законам природы.

Собственно, алгоритмы разные и они «не обязаны» подчинятся тем требованием, которые «предъявляют» к ним разные «Сфинксы». Но, дело в том, что среди этих алгоритмов есть и такие, которые «понимают» и исполняют требования. Выборка, которая делается среди разных

алгоритмов, для отбора только тех из них, которые «слушаются» и «исполняют» требования «Сфинксов» объясняет связь между дескриптивным и нормативным. Потому что в этой выборке остаются лишь те алгоритмы, которые подчиняются нормативным (требованиям «долга») законам. А дескриптивным (описательным) законам подчиняются все, про кого это написано – это как раз то, что мы называем «законы природы» и чем, преимущественно, занимается наука.

Даже выделение себя из мира – зависит от языка и условностей, которые ты признаёшь «своими» - если ты вообще что-то признаешь и способен соответствовать нормативным законам. Есть разные общества и «я» человека зависит от того, как позиционируется человек в данном обществе с нормативной стороны. Поэтому «я» - это в значительной степени понятие общественное. И тут мы намечаем путь разрешения отличий между «я» и обществом. И долг – это вполне может быть «хорошо» не для человека индивидуально, но для общества или – шире – Бога. И попытка свести поведение к индивидуализму оказывается глубоко ущербным ограничением с точки зрения логической полноты, а уж человек – это, прежде всего, социальное творение и, уходя в индивидуализм, он отказывается от самого главного и в себе самом, и в том, частью чего он является или хотя бы должен являться. Отказывается от того, для чего он и создан.

И, кроме того, при «правильном поведении» «Эдипа» задача стала решаемой и для других алгоритмов – хоть их решение и отличалось от ответа «Эдипа» («Эдип» не нашёл результат «АнтиЭдипа», а некоторые алгоритмы нашли, притом такую возможность создал «Эдип» своим ответом). То есть, мы тут видим, что выделение «Эдипа» из «команды» несколько условно – каждый алгоритм выполняет свой этап и не может этого сделать без другого. «Эдип», например, не может найти доказательство результата «АнтиЭдипа», но в состоянии выполнить требование задачи в рамках своей «компетенции», выступая «частью» решения задачи.

Разумеется, в жизни мы не проводим процедуру диагонализации – в приведенном практическом примере (в подразделе 5.2 «Теорема о построении алгоритма, применяемого к себе») мы видели, что эта процедура довольно громоздкая. Но всегда можно создать такой «язык программирования» где заданный алгоритм будет записан каким угодно текстом. Это как с именем – можно договориться, что «Дима» будет обозначаться как «ьооаывлдмтраиьбт», а строка «ьооаывлдмтраиьбт» будет обозначаться как «Дима». И наше требование к алгоритму решения («Эдипу») чтобы он мог понимать принятые условности.

Можно выбрать и такой язык, видимо, в котором имя субъекта будет заменять процедуру диагонализации для данного субъекта. Видимо, в какой-то мере подобное соглашение действует и в обычной жизни. А с учётом того, что аргумент долга зачастую используется для формирования требований к субъекту, которые он сам придумать не в состоянии – это даёт некоторую власть над ним. Тут можно вспомнить некоторые древние поверья о влиянии имени на человека.

Разумеется, всё это лишь предварительные наблюдения, для систематизации которых в математике даже формализма не было. Нужно ещё систематизировать факты, связанные с рефлексией, прежде чем делать какие-то обобщающие выводы. Хотя факты, связанные с рефлексией - есть и без обобщения. В этом смысле их изучение относится, скорее, к естественным, чем к точным наукам.

На примере авторизуемых задач из класса $\mathbb{NP}^+$ становится понятна и наша иллюзия «сво-

боды воли» - субъект просто не может «опередить» некоторые задачи. Просто он не может быть быстрее себя самого и собственные действия/мысли в значительной мере «приходят сами собой» по его восприятию – без предсказания, ведь нельзя во всем опередить себя самого. Это и даёт, видимо, «ощущение жизни», своей самостоятельности. И тут не в квантовой неопределенности дело – точно такая же иллюзия «свободы воли» имеет место и в детерминированном варианте.

И, кстати, относясь к разуму, как к возможности «опередить» мы опровергаем и возможности «демона Лапласа» - который охватил все положения и скорости атомов в детерминированном мире и узнал неотвратимое будущее. Решение: чтобы знать X – надо отличаться о X. Потребовалась бы другая Вселенная, чтобы понять нашу. И, говоря не формально, система не может знать больше половины информации о себе самой. А если «демона Лапласа» посадить за терминал изучать свой мозг – то материальные процессы в мозгу по запоминанию состояния мозга будут изменять его сильнее, чем он был понят в процессе изучения.

Если я задался целью опровергнуть свой собственный прогноз, то я его опровергну. Хоть со стороны все мои «опровержения» и будут вполне предсказуемыми – если «со стороны» за мной наблюдает кто-то более умный и быстрый чем я, конечно, и если он не сообщает мне своих прогнозов. А вот в борьбе (в шахматной партии, например) примерно равных соперников невозможно, вообще говоря, предвидеть стратегию и результат борьбы со своим визави.

Занятный вопрос, если всё предопределено, то зачем же напрягаться и прилагать усилия в попытках достижения целей? Но такое чувство как раз и означает негативный прогноз из-за перспективы бездействия. Наиболее известное решение данного парадокса – в цитате Жанны д'Арк: «Чтобы Бог даровал победу, солдаты должны сражаться».

Вопрос вины, если «всё предопределено» (точнее – подчинено законам природы, учитывая квантовую вероятность) тоже довольно легко решается на простейшем уровне – поведение судей тоже «предопределено» и они не виновны в вынесенных ими приговорах. Если же говорить более объективно, то земная жизнь отдельного человека не была в приоритете большую часть Истории и нынешний «гуманизм» - скорее всего, ложное отклонение. Общество не обязано вникать в проблемы преступника и проявлять снисхождение, если он из-за своих действий поставил себя вне общества. Кто вредит – тот исключается.

А как быть с той же Второй теоремой Гёделя, что я не могу быть уверенным в своей правоте? Ведь есть же случаи упрощения меня, когда я вполне могу предвидеть будущий примитивный результат? Но проблема в том, что опора на диагонализацию вовсе не гарантирует корректность данной диагонализации. Ты зависишь от неё, а не наоборот. И диагонализация, которой пользуется алгоритм для получения «своего» программного кода может дать ему код совсем другого алгоритма. Это как при полёте в пропасть – я отлично могу прогнозировать ту мокрую лепёшку, которой я стану на дне пропасти через короткое время, но... за мгновение до удара я – просыпаюсь. Довольно обычная ситуация, кстати.

В общем, когда Бог создавал мир, он здорово придумал с рефлексией (диагонализацией). Это то, что доступно для разумного субъекта и позволяет ему узнавать, какие задачи стоят именно перед ним. При этом сама возможность узнать свои цели подразумевает твою вовлеченность в некую общность – условностей, языка и распределения компетенций. Не только индивидуализм, но и общность, как это ни парадоксально.

Если задуматься, то 19-20 века с атеизмом, дарвинизмом, индивидуализмом, коммунизмом,

либерализмом и т.п. бездоказательными теоретически и непроверенными практически даже веками идеологиями далеко отклонили разум и мораль людей от истины. Человек греховен и разрушение государств и морали было легко осуществлять при взрывном развитии технологий, когда возможность предвидеть и пресекать деструктивные тенденции очень затруднена. А лишившись ограничений – человеческая греховность привела к рекам крови и моральной деградации.

Отсюда и тренд – к отрицанию того, что может быть важнее отдельного человека, отдельного разума. Тренд, под атаку которого попала и математика – с поисками «полноты», «разрешимости» и т.п. желаемых свойств для отдельных алгоритмов, теорий и т.п. косвенного «обоснования» индивидуализма.

Но в математике (редкий случай в ту эпоху Революций) данный «индивидуалистический тренд» был сокрушен Гильбертом, Гёделем, Тьюрингом, Тарским, Ричи и другими математиками, показавшими, с божьей помощью, и ограниченность возможностей у любой «вменяемой» теории, и отсутствие оснований для мечтаний о всемогуществе отдельного разума и заведомая ошибочность любых «доказательств» для утверждений типа «в целом я прав». Выводы такого рода сам субъект делать не может.

Ведь та же теорема Гёделя – о том, что «теория, доказавшая свою правоту – противоречива». А можно толковать теорему Гёделя и как условность «своих» границ для теории - «теория не может доказать, что в ней что-то не доказывается». Такая логическая ложность «индивидуализма», если угодно.

Вот и ответ на вопрос, сформулированный в Разделе 2. «Программа Гильберта и теория алгоритмов», почему наблюдался откат от достигнутого уровня развития логики: Косвенно выводы математики о неполноте и неразрешимости идут против нынешнего разрушительного тренда общества о неприятии ограничений над индивидуальностью – со всей её греховностью.

Опровергнуть достижения логики нельзя, но можно просто не думать о них – раз математика показала, что её выводы скорее погубят доминирующий (пока) греховный тренд, чем «обоснуют» его. Поэтому и развитие в направлении, намеченном Гильбертом и другими выдающимися логиками той эпохи затормозилось. Но я считаю, что на греховном тренде индивидуализма сказываются удары его противников и необходимо его добивать. И для этого тоже надо продолжать работать в вопросах неполноты, неразрешимости и рефлексии.

Не будем далее углублять в «рефлексивную логику» – вопросов и задач для одной заметки в ней слишком много, поэтому ограничимся тут тем, что данный путь обнаружен.

Список литературы

[1] *Дж. Булос, Р. Джеффри.* Вычислимость и логика. Мир, М. 1994.

[2] *Д. Гильберт, П. Бернайс.* Основания математики. Логические. исчисления и формализация арифметики Наука, М. 1979.

[3] *Э. Мендельсон.* Введение в математическую логику Наука, М. 1984