Глава 1

Машины Тьюринга.

1.1 Неформальное введение.

Неформальное понятие *модели вычислений* предполагает задание *языка программирования* вместе с его операционной *семантикой*, т.е. объяснением того, каким образом программе сопоставляется *процесс вычислений*.

Процесс вычислений, развиваясь во времени (которое дискретно), потребляет различные (разнородные) *ресурсы*. Предполагается, что величина потребленного к *данному* моменту ресурса измеряется натуральными числами и не может быть бесконечной, т.е. бесконечный ресурс требуется только для никогда не заканчивающегося процесса. Удобно (хотя не столь существенно) также считать, что каждый никогда не заканчивающийся процесс потребляет бесконечное количество каждого ресурса, т.е.

$$\text{Ресурс} < \infty \Leftrightarrow \text{процесс вычисления заканчивается.}$$

Обычно модель вычислений задает способ подсчета использованных ресурсов в каждом конечном вычислении. Т.е.

- функция $Compl : (program, input) \mapsto recourse$ – вычислима,
- отношение $Compl(p, i) < n$ – разрешимо.

Основные (но не единственные) ресурсы – время и память. *Но для каждой модели вычислений они свои !*

Как используют модель вычислений для оценки трудоемкости алгоритмов: хорошо реализуют алгоритм в виде программы p на языке

V. Krupski
COMPLEXITY
Lecture Notes, draft

программирования данной модели, выбирают ресурс r (из числа определенных в модели) и ищут верхние оценки $f(n)$ на величину затрат. Например, в худшем случае:

$$\max_{size(i)=n} Compl_r(p,i) < f(n).$$

Основное противоречие: точные оценки такого рода интересны для моделей вычислений с реалистичными языками программирования, для которых получение каких-нибудь теоретических оценок очень трудно. Выходы:

- (машинно зависимый, зависит от "железа", а потому результаты быстро устаревают) Численный эксперимент.
- (не вполне честный) Огрубления в определении ресурса. Например, в качестве времени для C – подсчет только количества выполняемых арифметических операций (или только сравнений).
- Построение честных оценок для более простой "игрушечной"модели и учет замедления при компиляции в реальный язык.

Пример. Пусть "игрушечная" модель вычисления M допускает компилятор программ в C

$$compiler : p_M \mapsto p_C,$$

про который *доказано* (одна теорема для пары моделей), что

$$Time_C(p_C,i) < (Time_M(p_M,i))^5 + 1000.$$

Если мы установим, что временная сложность некоторой задачи в M есть $O(n^2)$, то получим верхнюю оценку $O(n^{10})$ в C. Это загрубленная, но честная оценка.

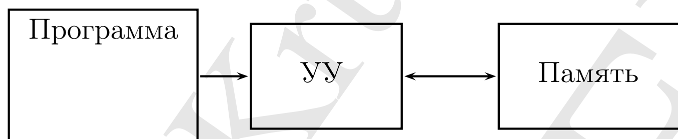
Как относится к таким оценкам. Конечно, показатель степени весьма неточен, но то, что время есть степенная функция, а не экспонента – абсолютно точно! Значит, использовать этот подход разумно тогда, когда важна разница между многочленом и экспонентой, а не между n^2 и n^3. Тогда и теорему про компилятор можно доказывать не в самой сильной форме (например, не уточняя показатель степени), когда она в большинстве случаев оказывается очевидной.

V. Krupski COMPLEXITY Lecture notes, draft

Наблюдение. Для реальных универсальных языков программирования и большинства универсальных "игрушечных"(не специально испорченных) языков компиляция одного в другой замедляет по времени не более, чем полиномиально, и увеличивает потребность к памяти не более, чем линейно. Именно на это положение дел следует рассчитывать в приложениях теории сложности и внутри самой теории.

1.2 Модели Тьюринга.

Это семейство моделей вычислений наиболее честно отражает время вычислений. Возможных вариантов определения много. Машина Тьюринга состоит из управляющего устройства (УУ) и потенциально бесконечной внешней памяти, структура которой не меняется со временем. Она снабжена программой, задающей правила ее функционирования.

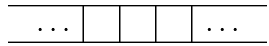


Память разбита на одинаковые ячейки, предназначенные для хранения букв фиксированного конечного алфавита (одна буква в ячейке). Имеется одна или несколько (конечное число) читающе-пишущих головок. В каждый момент времени головка обозревает одну ячейку памяти. За один такт работы головка может изменить содержимое этой ячейки и переместиться к одной из соседних (или остаться на месте). Функционированием головок управляет УУ. В каждый момент времени УУ находится в одном из фиксированного конечного множества внутрнних состояний. Один такт работы состоит в следующем: машина читает содержимое обозреваемых ячеек и внутреннее состояние, выбирает из программы инструкцию, однозначно определяемую этими данными, и исполняет ее. В инструкции сказано, каким должно стать новое внутреннее состояние, какие буквы должны быть записаны в обозреваемые ячейки и куда должны переместиться головки.

Варианты определения различаются геометрией памяти (это существенно, т.к. за один шаг головкам разрешается переместиться только в соседние ячейки). В простейших моделях память представляет собой ленту с одной головкой, бесконечную в одну

V. Krupski COMPLEXITY Lecture Notes, draft

или обе стороны



В теории сложности обычно используют многоленточные машины Тьюринга – конечный набор лент с одной головкой на каждой (более детальное описание см. ниже). Рассматривались также машины Тьюринга с "многомерной"памятью (напр., в виде клетчатой плоскости) и различные "многоглавые"варианты (несколько головок на одной связной компоненте памяти). Можно показать, что все эти варианты моделируют друг друга с полиномиальным (квадратичным) замедлением по времени и линейным увеличением памяти.

Вычислительные возможности машин Тьюринга достаточно хорошо изучены. Их качественная характеризация состоит в описании класса всех функций, вычислимых на машинах Тьюринга. Речь идет о частичных словарных функциях $f : \Sigma^* \to \Sigma^*$, где Σ^* – множество всех слов в конечном алфавите Σ, а термин "частичная функция" означает, что значение функции $f(v)$ для некоторых слов $v \in \Sigma^*$ может оказаться неопределенным. Результат сравнительного изучения запасов вычислимых словарных функций для различных (не только "игрушечных") моделей вычислений может быть сформулирован в виде следующего неформального утверждения:

> **Тезис Тьюринга (неформальный):** Каждую вычислимую (программой какого угодно языка программирования) частичную функцию типа $\Sigma^* \to \Sigma^*$ можно вычислить на подходящей машине Тьюринга.

Неформальность Тезиса Тьюринга состоит в том, что он не может быть полностью обоснован математическими средствами (доказан как математическая теорема). В то же время все многочисленные попытки его опровергнуть, т.е. предложить язык программирования с большими вычислительными возможностями, оказались безуспешными (в результате чего в настоящее время подобные проекты считаются абсолютно бесперспективными). Причина невозможности полного математического обоснования кроется в словах "какого угодно языка программирования" – они не определяют ни самого семейства языков, о которых идет речь, ни какого-либо свойства этого семейства. Если их заменить на достаточно информативное описание семейства языков программирования (которое необходимо окажется менее общим), то соответствующий частный случай Тезиса станет обычным, "поддающимся доказательству" математическим утверждением.

V. Kritskii COMPLEXITY Lecture notes, draft

Пример. Пусть семейство состоит из языков, C и PASCAL, заданных полным описанием их синтаксиса и операционной семантики. Оба языка обладают компиляторами в ASSEMBLER, поэтому для обоснования соответствующего частного случая Тезиса Тьюринга достаточно построить компилятор, преобразующий ассемблерный код в программу для машины Тьюринга. Последнее является весьма трудоемкой, но абсолютно реалистичной задачей по программированию. Для аккуратного математического доказательства потребуется еще верифицировать все три компилятора, т.е. доказать, что они работают правильно.

1.3 Многоленточные машины Тьюринга.

Ленты бесконечны в обе стороны. Пустые ячейки несут символ #. Есть одна или несколько входных лент (только для чтения), одна или несколько рабочих лент (можно читать и писать), некоторые из которых можно объявить выходными. Входной алфавит Σ_0 содержится в ленточном Σ и не содержит #. Результат (на выходной ленте) есть то слово в алфавите Σ_0, на котором или непосредственно перед которым остановилась головка. Удобно также требовать, чтобы на входных лентах головка не удалялась от границ входного слова наружу: например, можно ввести ограничительные буквы b и e для обозначения границ входного слова, за которые головке вылезать запрещается. Начальная конфигурация лент выглядит так:

Входная лента с исходными данными:

$\ldots\#$	b	1	0	1	1	e	$\#\ldots$
	$\triangle$						

Рабочие ленты:

$\ldots\#$	#	#	#	#	#	$\#\ldots$
	$\triangle$					
$\ldots\#$	#	#	#	#	#	$\#\ldots$
	$\triangle$					
$\ldots\#$	#	#	#	#	#	$\#\ldots$
	$\triangle$					

Работа машины описывается тремя конечными функциями (k – количество лент, Q – фиксированное заранее конечное множество состояний

COMPLEXITY V. Krupski Lecture Notes, draft

УУ):

$$
\begin{array}{ll}
\alpha : Q \times (\Sigma^*)^k \to Q & \text{– новое состояние} \\
\beta : Q \times (\Sigma^*)^k \to (\Sigma^*)^k & \text{– новое содержимое ячеек} \\
\gamma : Q \times (\Sigma^*)^k \to \{N, L, R\}^k & \text{– движения головок}
\end{array}
$$

Их удобно записывать в виде программы – набора непротиворечивых команд вида:

$$q\bar{a} \mapsto \alpha(q,\bar{a})\beta(q,\bar{a})\gamma(q,\bar{a})$$

Непротиворечивость набора означает, что в нем нет двух команд с одинаковой левой частью $q\bar{a}$. Для сокращения письма предполагают также, что если команда с левой частью $q\bar{a}$ не выписана явно, то она все-таки есть в программе, но имеет специфический вид $q\bar{a} \mapsto q\bar{a}\, N \dots N$. Такие команды тривиальны, т.к. не вызывают никаких действий.

Пример. На Рис.1.1 приведена программа машины Тьюринга с одной входной и одной рабочей лентой, которая переписывает исходное двоичное слово в обратном порядке.

Полная спецификация многоленточной машины Тьюринга есть набор

$$< Q, \Sigma, \alpha, \beta, \gamma, q_., q_. > .$$

Если к ней добавлен входной алфавит $\Sigma_0 \subset \Sigma \setminus \{\#, b, e\}$, выделены входные ($k_0$ штук) и выходные (l_0 штук) ленты и α в самом деле не меняет содержимое входных лент, то получаем вычислитель (частичной) функции $f : (\Sigma_0^*)^{k_0} \to (\Sigma_0^*)^{l_0}$. Для вычисления значения функции – вектора $(w_1, \dots, w_{l_0}) = f(v_1, \dots, v_{k_0})$ надо записать на входных лентах слова bv_ie, запустить машину и дождаться остановки, после чего считать с j-той выходной ленты то слово в алфавите Σ_0, на которое указывает головка. Это w_j. Если головка остановилась не на букве алфавита Σ_0, то результат – пустое слово. Если машина работает бесконечно долго, то значение $f(v_1, \dots, v_{k_0})$ не определено.

Замечание. Конечное множество состояний на самом деле может представлять любую ограниченную (для данной машины) структуру данных. Это имитирует статически распределенную память с возможностью за один шаг переприсваивать ее всю целиком. Единственное ограничение – новое состояния определяется (заранее) однозначно по предыдущему состоянию и содержимому обозреваемых ячеек ленты.

COMPLEXITY
N. Krupski
Lecture Notes, draft

Рис. 1.1: Пример машины Тьюринга.

V. Krupski COMPLEXITY Lecture Notes, draft

V. Krupski
COMPLEXITY
Lecture Notes, draft

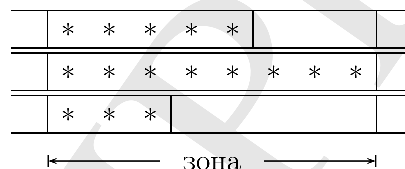
Глава 2

Время и память.

2.1 Время и зона машины Тьюринга.

Время $T_M(input)$ – это число шагов в вычислении машины Тьюринга M на входе *input*. Шагом считается исполнение одной команды, т.е. соответствующие изменения состояния и содержимого обозреваемых ячеек памяти, а также перемещения головок, происходят за 1 времени.

Зона, память $S_M(input)$ определяется как максимум по всем рабочим лентам количества использованных ячеек к моменту остановки машины M на входе *input*. Если машина никогда не заканчивает работу, то полагаем зону бесконечной. Ячейка считается использованной, если в ней побывала головка.



Выбор максимума в определениии зоны вместо более естественной меры – суммарного количества использованных ячеек – значительно упрощает расчеты, но практически не сказывается на получаемых оценках. В то же время оказывается существенным исключение входных лент из подсчета. Оно позволяет адекватно оценивать малые затраты памяти, когда вычисление требует рабочих ячеек меньше, чем длина входного слова.

Проблема с определением зоны: из определения непонятно, как алгоритмически проверять условие "$S_M(input) < s$" (это было одним из

COMPLEXITY V. Krupski Lecture Notes, draft

желанных свойств определения меры сложности). На самом деле такой алгоритм существует. Он основан на следующей оценке:

Теорема 2.1 *По описанию машины Тьюринга M можно вычислить константу C такую, что если процесс вычисления M на некотором входе заканчивается за t шагов с зоной s, то*

$$s \leq t \leq n^{k_0} C^s,$$

где n – максимум длин входных слов, k_0 – число входных лент.

Доказательство. Левое неравенство: за один шаг на рабочей ленте расходуется не более одной ячейки.

Докажем правое неравенство. Для данного вычисления определим понятие конфигурации в данный момент времени: это состояние, содержимое всех лент и положение всех головок (относительно левой границы блока использованных на данной ленте ячеек). Легко видеть, что если в два разных момента времени мгновенные конфикурации совпадают, то вычисление зациклилось и никогда не закончится. Значит, для нашего (заканчивающегося) вычисления все конфигурации различны. Если зона вычисления есть s, вход фиксирован, то различных возможных конфигураций не более, чем

$$|Q| \cdot |\Sigma|^{(k-k_0)s} \cdot (n+2)^{k_0} \cdot s^{k-k_0} \leq n^{k_0} C^s.$$

Здесь $|Q|$ – количество состояний, $|\Sigma|$ – количество букв в ленточном алфавите, k – общее количество лент. Сомножитель $|\Sigma|^{(k-k_0)s}$ оценивает сверху число различных вариантов заполнения лент буквами алфавита $|\Sigma|$, а произведение $(n+2)^{k_0} \cdot s^{k-k_0}$ – количество возможных расположений головок.

Число шагов t не больше числа различных конфигураций, т.е. оценивается сверху этой же величиной $n^{k_0} C^s$.

■

Алгоритм проверки условия “$S_M(input) < s$”:

1. По $M, input, s$ вычисляем $T = n^{k_0} C^s$ и моделируем вычисление $M(input)$ на T шагов.

2. Если оно закончилось и фактическая зона оказалась меньше s, то возвращаем $true$; иначе – $false$.

COMPLEXITY V. Krupski Lecture Notes, draft

2.2 Цена сокращения алфавита.

Теорема 2.2 *Для вычисления функции типа*

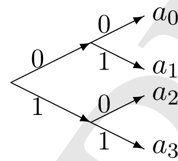
$$(\{0,1\}^*)^{k_0} \to (\{0,1\}^*)^{l_0}$$

достаточно ленточного алфавита $\{0,1,\#,b,e\}$*. При переходе к такому алфавиту зона и время изменяются линейно. (Буквы* b *и* e *используются только как ограничители на входных лентах.)*

Замечание. При прямой конструкции добавится l_0 рабочих лент (выходных). Если не добавлять, то время может возрасти не более, чем квадратично.

Доказательство. Для простоты изложения рассмотрим случай одной рабочей ленты. Пусть исходная машина уже модифицирована так, что на рабочей ленте она не оставляет "пустыми"(т.е. $\#$) ячейки, в которых бывала. (Это можно сделать с помощью введения новой буквы $*$ – двойника для $\#$.)

Фиксируем подходящее число $m \approx \log_2 |\Sigma|$ так, чтобы каждую букву ленточного алфавита Σ можно было однозначно закодировать блоком из m бит. Фиксируем такое кодирование. Его удобно представлять в виде бинарного дерева D глубины m: пути из корня к листьям – коды; на листе – соответствующая буква алфавита Σ.



Каждой ленточной (без состояния) конфигурации исходной машины сопоставим ленточную конфигурацию новой машины, заменив на рабочей ленте буквы алфавита Σ кроме $\#$ на их коды (головки в началах соответствующих блоков). Зона возрастет в m раз.

...	a_5	...	$\longmapsto$	...	10010	...
	$\triangle$				$\triangle$	

Каждый шаг исходной машины будет моделироваться $2m$ шагами (изменение буквы a_i на a_j) и еще m шагов потребуется для перемещения головки к соседнему блоку (если необходимо). Основное состояние моделирующей машины состоит из

V. Krupski
COMPLEXITY
Lecture Notes, draft

1. состояния исходной машины,

2. переменной для хранения одного символа $\in \{N, L, R\}$,

3. переменной-счетчика для хранения одного из чисел $0, \ldots, m-1$,

4. пометки "read", "write"или "go",

5. бинарного дерева D с выделеной вершиной. В начальный момент моделирования шага выделен корень, а пометка есть "read".

Заметим, что число состояний конечно и не зависит от входа.

Этап "read". Первые m шагов моделирования состоят в том, что головка сканирует код буквы слева направо, а в состоянии меняется только выделенная вершина дерева D – она перемещается по пути, соответствующему считываемому коду. Когда она достигнет листа, на нем будет написана изменяемая буква a_i. Тогда пометка "read"меняется на "write"и выделенной вершиной становится лист с буквой a_j. В этот момент меняем хранимое состояние исходной машины и (если необходимо) производим перемещение головок на входных лентах так, как это делала моделируемая машина. Также определяем и запоминаем требуемое направление движения головки на рабочей ленте.

Этап "write". Следующие m шагов головка движется по блоку в обратном направлении (налево), а выделенная вершина – к корню дерева D. При этом путь, проходимый выделенной вершиной, определяет биты кода буквы a_j (справа налево) и они пишутся в соответствующие ячейки блока. Когда выделенная вершина достигает корня, пометка "write"меняется на "go".

Этап "go". Если запомнен символ движения L или R, то перемещаем головку на m позиций в соответствующем направлении, отсчитывая их с помошью счетчика. В конце меняем пометку "go"на "read"и переходим к моделированию следующего шага. Если запомнен символ N, то просто меняем пометку.

Заметим, что очередное состояние всегда есть функция от предыдущего и очередной считанной буквы, как того и требует формализм.

Отдельные усилия требуются для моделирования поведения машины на границе рабочей зоны – когда головка моделирующей машины в начале шага находится в ячейке с буквой #. Тогда текущий и ближайшие вправо $m-1$ символов # (они обязательно будут такими!) надо заменить на код буквы # (понадобятся дополнительные состояния), а только потом выполнять основной шаг.

COMPLEXITY N. K. Vereshchagin Lecture Notes, draft

Так построенная моделирующая машина делает то же, что и исходная, но выдает ответ в виде последовательности кодов букв 0 и 1. Для декодирования добавляем одну дополнительную выходную ленту и в конце переписываем на нее результат, декодируя в процессе переписывания. Время на это – $m \cdot$ (длина результата), что не превосходит

$$m \cdot (\text{время исходного вычисления}).$$

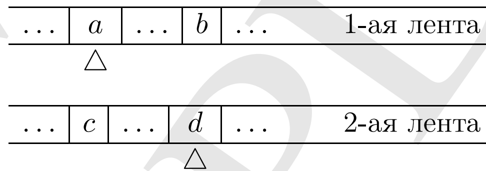
■

2.3 Цена сокращения количества лент.

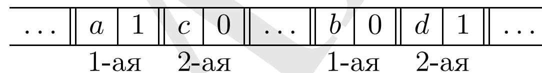
Замечание. Одна входная и одна выходная ленты (рабочих – много) моделирует общий случай за счет введения буквы-разделителя. Цена – время на переписывание входа с одной ленты на много и аналогичного переписывания результата.

Теорема 2.3 *Много рабочих лент моделируются одной с квадратичным замедлением по времени. Зона меняется линейно. Если ленточный алфавит был не однобуквенным, то его удается сохранить.*

Доказательство. (Набросок.) Пара рабочих лент



моделируется одной так:



Одной ячейке на исходной ленте соответствуют две соседние ячейки на моделирующей. Левая содержит ту же букву, а правая (0 или 1) – признак того, что одна из головок исходной машины обозревает моделируемую ячейку. Такие пары чередуются, т.е. соседним ячейкам на одной из исходных лент соответствуют пары, расположенные на расстоянии 2. Легко видеть, что зона возрастает в два раза.

V. Krutski COMPLEXITY Lecture Notes, draft

Один шаг исходной машины моделируется двойным проходом по всей рабочей зоне моделирующей ленты. При движении справа налево определяются буквы, обозреваемые исходной машиной, а при движении в обратную сторону имитируются необходимые изменения (замена букв и перемещение головок). Моделирующее состояние поддерживает счетчик для хранения остатка от деления на 4 величины текущего смещения головки, что позволяет определить, к какой из моделируемых лент относится обозреваемая ячейка. При этом в каждой ячейке рабочей зоны головка моделирующей машины бывает не более фиксированного числа (c) раз, поэтому время работы моделирующей машины не превосходит $c \cdot 2S \cdot T$, где T – время, а S - зона исходного вычисления. Т.к. $S \leq T$, то это моделирование приводит к квадратичному замедлению.

■

Замечание. Что изменится, если надо избавляться не только от многих рабочих лент, но и от входной ленты, т.е. моделировать вычисления функции типа $\{0,1\}^* \to \{0,1\}^*$ на одноленточной машине? Пусть моделируемая машина уже двухленточная, одна из которых – входная. Надо добавить 2 модуля: один будет переписывать входное слово "разряженно",

#	c_1	c_1	...
△			

в

	#	1	#	1	c_1	0	#	0	c_2	0	#	0	...
	1-ая		2-ая		1-ая		2-ая		1-ая		2-ая		

,

а второй – выполнять обратное преобразование с результатом. Все остальное моделирование сохраняется. Добавление требует полиномиального времени и линейной зоны (от длины входа и результата).

Итог: Каждую вычислимую на многоленточной машине функцию типа $(\{0,1\}^*)^{k_0} \to \{0,1\}^*$ можно вычислить на машине с единственной рабочей лентой и ленточным алфавитом $\{0,1\}$ *с полиномиальным замедлением по времени и линейным увеличением зоны.* Для функций типа $\{0,1\}^* \to \{0,1\}^*$ моделирующая машина может быть выбрана одноленточной (т.е. и без входных лент тоже) с теми же оценками времени и памяти, если исходное вычисление по крайней мере читает все входное слово (для того, чтобы вклад дополнителных модулей не превышал остального, достаточно, чтобы время и зона исходного вычисления оценивались снизу линейной функцией от длины входа).

Конечно, аналогичные факты справедливы и для произвольного алфавита мощности > 1, а также для вектор-функций с размерностью

COMPLEXITY
V. Krupski
Lecture notes, draft

значений $l_0 \geq 1$, если разделить функции рабочей и выходных лент (например, разрешить на выходной ленте только писать с одновременным сдвигом головки вправо).

V. Krupski
COMPLEXITY
Lecture Notes, draft

COMPLEXITY
V. Krupski
Lecture Notes, draft

Глава 3

Универсальные машины Тьюринга.

3.1 Машина Тьюринга, универсальная для класса $\mathcal{C}$.

Пусть фиксирован ленточный алфавит $\Sigma \supset \{0, 1\}$, наборы входных, рабочих и выходных лент. Устроим кодирование класса $\mathcal{C}$ всех таких машин Тьюринга словами в алфавите $\{0, 1\}$:

1. Состояния условимся записывать двоичными записями их номеров, взятыми в кавычках: '101' есть q_5. Пусть начальное состояние всегда имеет номер 1, а заключительное – 0.

2. Машину Тьюринга сначала запишем в виде программы, состоящей из команд вида
$$q\bar{a} \mapsto q'\bar{a}'D;$$
Программа оказывается словом в фиксированном алфавите программ.

3. Фиксируем однозначное кодирование букв алфавита программ двоичными словами фиксированной длины и применим это кодирование к программе машины Тьюринга $M \in \mathcal{C}$. Так определяется *код машины* $Code(M)$.

Определение 3.1 Универсальной для класса $\mathcal{C}$ называется машина Тьюринга U с дополнительной входной лентой такая, что

$$U(Code(M), input) \simeq M(input) \tag{3.1}$$

COMPLEXITY
V. Krupski
Lecture Notes, draft

выполняется для всех $M \in \mathcal{C}$ и всех входов $input \in (\Sigma^*)^{k_0}$.

Замечание. Условие (3.1) означает, что оба вычисления $U(Code(M), input)$ и $M(input)$ либо зацикливаются, либо дают одинаковый результат. Использованный в определении символ $\simeq$ есть так называемое условное равенство. Это традиционный заменитель обычного отношения равенства, когда приходится сравнивать выражения, которые могут оказаться неопределенными. Утверждение $a \simeq b$ считается истинным в двух случаях: когда оба выражения a и b определены и равны между собой, либо когда они оба не определены. Во всех остальных случаях утверждение считается ложным. Отличие от обычного равенства состоит в том, что когда хотя бы одно из выражений a или b не определено, выражение $a = b$ не является корректным утверждением и не может быть ни истинным, ни ложным. Напротив, $a \simeq b$ всегда есть корректное утверждение.

3.2 Конструкция универсальной машины.

Факт существования универсальной машины немедленно следует из тезиса Тьюринга. По коду машины можно алгоритмически восстановить саму машину и применить ее к данному входу, т.е. функция, которую должна вычислять U, вычислима. Значит, ее можно вычислить на машине Тьюринга.

Нетрудно явно построить универсальную машину U в том же алфавите *с одной дополнительной рабочей лентой*, для которой время вычисления $U(code(M), input)$ лишь в константу раз больше времени вычисления $M(input)$ (сама константа зависит от M; зона может возрасти не более, чем на аддитивную константу).

На дополнительной рабочей ленте U хранит двоичный код текущего состояния моделируемой машины M, а остальные ленты (кроме дополни-

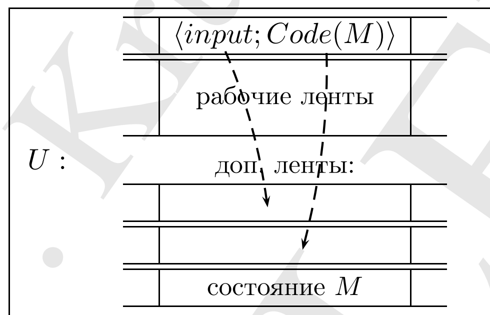
COMPLEXITY Lecture Notes, draft

тельной входной) она использует так же, как M. Имитация одного шага M: машина U читает текущее состояние, ищет в Code(M) ту команду, которую надо выполнить и выполняет (измененное состояние записывается на место предыдущего). На все это уходит не более $(|Code(M)| + |Q|) \cdot c$ шагов.

Избавиться от дополнительной входной ленты (для $Code(M)$) позволяет следующее кодирование пар двоичных слов:

$$\langle \overline{a_1 \ldots a_n}\ ;\ \overline{b_1 \ldots b_m} \rangle := \overline{a_1 \ldots a_n 01 b_1 b_1 \ldots b_m b_m}.$$

Код машины M можно подавать на вход, добавляя его к содержимому первой из входных лент в удвоенном виде (т.е. на ленту пишется $\langle input; Code(M) \rangle$). Тогда в начале работы универсальная машина переписывает на две дополнительные рабочие ленты *input* и $Code(M)$ по отдельности, а затем работает как указано выше.



Избавиться от дополнительных рабочих лент можно с помощью теоремы 2.3. Все это приведет к тому, что универсальная машина для класса $\mathcal{C}$ будет *присутствовать в самом классе*. Для класса с одной входной лентой определяющее ее равенство будет выглядеть

$$U(\langle input\ ;\ Code(M) \rangle) \simeq M(input), \quad M \in \mathcal{C}, \tag{3.2}$$

но время моделирования согласно теореме 2.3 возрастет квадратично, а зона – линейно. Тем самым установлена следующая теорема.

Теорема 3.2 *В классе $\mathcal{C}$ существует универсальная в смысле (3.2) машина U для этого класса. Для нее*

$$\begin{aligned} T_U(\langle input\ ;\ Code(M) \rangle) &= O(\ (T_M(input))^2\ ), \\ S_U(\langle input\ ;\ Code(M) \rangle) &= O(\ S_M(input)\ ). \end{aligned}$$

V. Krupski COMPLEXITY Lecture Notes, draft

Замечание. Время моделирования можно несколько улучшить – до $O(T \log T)$ за счет того, что длина содержимого устраняемых рабочих лент в процессе моделирования машины M остается ограниченной и оставшейся головке удается таскать все эти данные по ленте с собой.

3.3 Теоремы об иерархии по времени и по зоне.

Вопрос. Когда добавочный ресурс в самом деле увеличивает вычислительные возможности?

Фиксируем класс $\mathcal{C}$. Ограничимся случаем одной входной и одной выходной ленты. Пусть f – тотальная неубывающая неограниченная вычислимая функция из N в N. Определим классы $TIME(f)$ и $SPACE(f)$ как семейства всех функций типа $\Sigma^* \to \{0,1\}$, вычислимых на машинах из класса $\mathcal{C}$ с условием

$$T_M(v) < f(|v|) \text{ для достаточно больших } |v|$$

или

$$S_M(v) < f(|v|) \text{ для достаточно больших } |v|$$

соответственно.

Вопрос уточняется следующим образом: *найти условия на порядки роста функций f, g, достаточные для выполнения строгого включения*

$$\begin{gathered} TIME(f) \underset{\neq}{\subset} TIME(g) \\ (\text{соотв., } SPACE(f) \underset{\neq}{\subset} SPACE(g)\ ). \end{gathered} \tag{3.3}$$

Результаты, устанавливающие такие оценки, называются *теоремами об иерархии.*

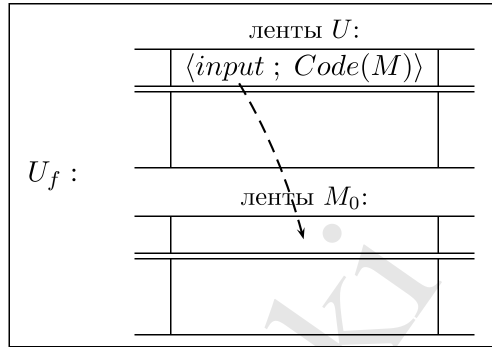
Способ получения теорем об иерархии.

Функция f называется конструируемой по времени (по зоне), если существует машина Тьюринга M, для которой

$$T_M(v) = f(|v|) \quad (\text{ соотв., } S_M(v) = f(|v|)\ ).$$

Пусть f конструируема по времени (или по зоне) машиной M_0. Рассмотрим следующую модификацию U_f универсальной машины U. К собственным лентам U в качестве дополнительных рабочих добавлены ленты машины M_0. Сначала вход копируется на бывшую входную ленту M_0, после чего обе машины M_0 и U (в составе одной U_f) запускаются параллельно.

V. Krupski
COMPLEXITY
Lecture Notes, draft



Машина M_0 служит таймером – как только она останавливается, прерываем работу U тоже. При этом синхронизируем работу машин U и M_0 таким образом, чтобы 1 шаг машины M_0 соответствовал 1 шагу моделируемой машины M, т.е. U делает все шаги, моделирующие 1 шаг M, после чего M_0 делает 1 шаг, и т.д. Результат читаем с выходной ленты U и меняем на противоположный: 0 на 1, а все остальное – на 0.

Легко видеть, что функция

$$h(v) = U_f(\langle v; v \rangle)$$

тотальна, вычислима, имеет тип $\Sigma^* \to \{0,1\}$, но не лежит в $TIME(f)$ (соответственно, в $SPACE(f)$). В самом деле, если бы $h \in TIME(f)$, то добавлением в программу вычисляющей ее машины "лишних"команд можно добиться выполнения условия $T_M(v) < f(|v|)$ (или аналогичного для зоны) уже при $v = Code(M)$. Но тогда

$$\begin{aligned} h(Code(M)) = M(Code(M)) &= U(\langle Code(M); Code(M) \rangle) \\ &\neq U_f(\langle Code(M); Code(M) \rangle) \end{aligned}$$

по построению. Противоречие.

Остается найти верхнюю оценку времени (соотв., зоны), достаточную для вычисления функции h. Рассмотренное ранее довольно грубое моделирование приводит к оценкам вида $(f(n))^C$ для времени и $C \cdot f(n)$ для памяти (константа C извлекается из конструкции). Это означает, что строгие включения (3.3) выполняются для всех функций g, растущих быстрее этих оценок. Более точные оценки следующие:

Теорема 3.3 *Для выполнения (3.3) достаточно, чтобы функция g удовлетворяла условию*

$$\frac{f(n) \log_2 f(n)}{g(n)} = o(1) \text{ (для времени)}$$

V. Krupski COMPLEXITY Lecture Notes, draft

или

$$\frac{f(n)}{g(n)} = o(1) \text{ (для памяти)},$$

причем функция $f(n) \geq n$ *должна быть конструируемой.*

Непосредственным программированием можно показать, что задаваемые простыми формулами арифметические функции – конструируемы, поэтому

$$\begin{aligned} &TIME(n) \underset{\neq}{\subset} TIME(n^2) \underset{\neq}{\subset} TIME(n^3) \underset{\neq}{\subset} \ldots \\ &\quad \underset{\neq}{\subset} TIME(2^n) \underset{\neq}{\subset} TIME(3^n) \underset{\neq}{\subset} \ldots, \end{aligned}$$

$$\begin{aligned} &SPACE(n) \underset{\neq}{\subset} SPACE(2 \cdot n) \underset{\neq}{\subset} \ldots \\ &\underset{\neq}{\subset} SPACE(n^2) \underset{\neq}{\subset} \ldots \underset{\neq}{\subset} SPACE(2^n) \underset{\neq}{\subset} \ldots. \end{aligned}$$

V. Krupski COMPLEXITY Lecture Notes, draft

Глава 4

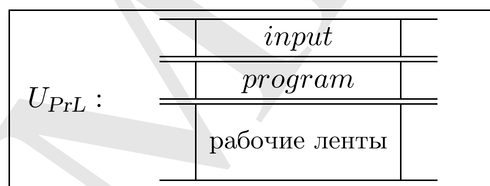
Моделирование других языков программирования.

4.1 Схема моделирование других языков программирования машинами Тьюринга.

Пусть для некоторого языка программирования PrL программы и возможные входные/выходные данные удается представить словами некоторого конечного (м.б. достаточно большого) алфавита. Естественно также предположить, что отображение

$$program, input \mapsto output$$

окажется вычислимой словарной функцией. Тогда по тезису Тьюринга найдется машина Тьюринга U_{PrL}, вычисляющая это отображение. Она будет универсальной для языка PrL, т.е. $U_{PrL}(input, program) \simeq program(input)$.



Построение универсальной машины, расширенной блоком записи на ее "программную"входную ленту исходной PrL-программы, фактически есть компиляция PrL в язык машин Тьюринга.

COMPLEXITY
V. Krupski
Lecture Notes, draft

4.2 Моделирование RAM.

В качестве примера рассмотрим простейший вариант языка BASIC (модели вычислений такого рода называются RAM, Random Access Mashines, или машинами с неограниченными регистрами).

- Программа оперирует с конечным набором переменных V0, V1, ... типа int (двоичное представление, разрядность заранее не ограничена). Вход и выход записывается в несколько первых переменных.
- Команды программы пронумерованы и выполняются в порядке нумерации. В языке два вида команд – присваивание (напр., 33:V5<-V3+V1) и условный переход (напр., 125:if V5<0 goto 7).

Для удобства обработки текста программы машиной Тьюринга выберем следующий "тэговый" вариант синтаксиса:

ПРИСВАИВАНИЕ:

```
<LET уникальный_номер> <V номер> выражение </LET>
        выражение::= константа
                     |<V номер>
                     |<V номер> op <V номер>
               op::= + | - | *
```

УСЛОВНЫЙ ПЕРЕХОД:

```
<IF уникальный_номер> <V номер> номер </IF>
```

команда:	ее запись:
33:V5<-V3+V1	<LET33><V5><V3>+<V1></LET>
125:if V5<0 goto 7	<IF125><V5>7</IF>

Соответствующая универсальная машина Тьюринга в качестве своей части имеет "математический процессор" – набор лент и компонент состояний для реализации операций +,-,* , рабочую ленту для хранения текущих значений переменных в формате <V номер> значение </V>, а также отдельную рабочую ленту для хранения уникального номера исполняемой команды. Исполнение очередной команды сводится к поиску ее текста на входной "программной"ленте, произведения мат. вычисления, записи его результата в соответствующую переменную и нахождения номера следующей команды.

V. Krupski COMPLEXITY Lecture notes, draft

Нетрудно заметить, что время моделирования одного шага оценивается полиномом от текущей зоны, которая, в свою очередь оценивается линейной функцией от s – максимума длин двоичных записей значений переменных (максимум – по всему вычислению). Таким образом, для моделирующей машины

$$TIME = O(t \cdot poly(s)), \quad SPACE = O(s), \tag{4.1}$$

где t – количество шагов, а s – максимум длин двоичных записей текущих значений переменных моделируемого вычисления.

Замечание. При добавлении в язык других *представляемых словами* типов данных и дополнительных встроенных операций (при условии их вычислимости на машинах Тьюринга за полиномиальное время на линейной зоне) полученные оценки не изменятся. Нетрудно также реализовать этими средствами управляющие конструкции:

IF vi>=0 THEN ... ELSE ... FI Цена – дополнительных 2 шага.

FOR vi:= vj DOWNTO 0 DO ... DONE Цена: +2 шага к каждой итерации, но само количество итераций есть vj +1 и оценка (4.1) сохранится только в том случае, когда текущее значение переменной vj есть $poly(s)$ (т.е. его длина – $O(\log s)$).

REPEAT ... UNTIL vi>= 0 Цена: +1 шаг к каждой итерации, но для получения оценок времени необходим честный подсчет числа итераций.

WHILE vi >= 0 DO ... DONE Цена: +2 шага к каждой итерации но для получения оценок времени необходим честный подсчет числа итераций.

Замечание. Некоторое неудобство для программирования представляет ограничение, состоящее в фиксированности (для данной программы) числа используемых переменных. Его легко преодолеть добавлением в синтаксис ссылок: <REF номер > означает содержимое переменой <V номер'>, где номер' есть содержимое переменной-указателя <V номер>. Разрешим использовать ссылки всюду, где могли стоять переменные. Например, если в переменной V5 хранилось число 31, то команда

```
25: REF5 <- 13
```

V. Krutski COMPLEXITY Lecture Notes, draft

приводит к присваиванию `V31 <- 13` (если же значение `V5` не было определено заранее, то происходит ошибка). Нетрудно заметить, что в случае такого расширения оценки (4.1) сохранятся, если взять $s = m \cdot l$, где m – количество использованных программой переменных (прямо и косвенно), а l – максимум длин двоичных записей их текущих значений.

Эти оценки обычно используют для получения грубых верхних оценок времени, достаточного для решения той или иной задачи на машинах Тьюринга. Выбирают подходящий язык PrL и программируют соответствующий алгоритм на нем, оценивая порядки величин t, s. Для получения более точных оценок стараются улучшить оценку (4.1) в случае моделирования специфического алгоритма.

4.3 Моделирование булевых схем.

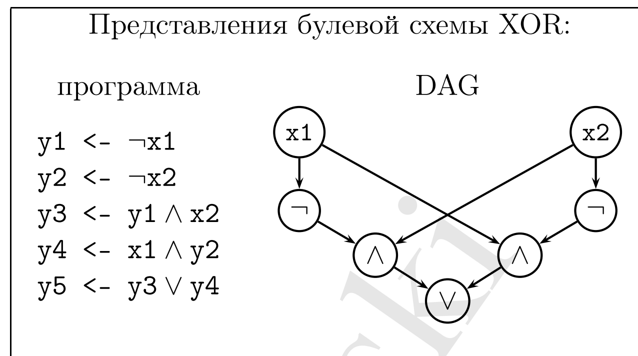
Булевы схемы – это более простой (не универсальный) язык программирования того же сорта. Основное отличие состоит в том, что возможные значения переменных здесь булевы, т.е. 0 или 1. Отсутствует нумерация команд и условные переходы (нет возможности оперировать с адресами команд и переменных). Программа состоит из последовательности операторов присваивания

```
y <-  <op> (x1,...,xk),
```

исполняемых последовательно. При этом фиксирована конечная полная система булевых функций F (например, $\{\neg, \vee, \wedge\}$) и `<op>`$\in F$. Все переменные, встречающиеся только в правых частях операторов присваивания, считаются входными. Некоторые из переменных схемы объявляются выходными. Таким образом, каждая схема вычисляет булеву вектор-функцию $\{0,1\}^n \to \{0,1\}^m$.

Другое представление булевых схем – помеченный ориентированный граф без циклов (DAG). Вершины-источники (без входящих ребер) помечены входными перемеными, а все остальные вершины – символами `<op>`$\in F$ При этом в вершину, помеченную n-местной опрерацией, должно входить ровно n ребер. Некоторые из вершин отмечены как стоки (выходные переменные). Это представление соответствует программам, в которых в левых частях операторов присваивания переменные не повторяются.

COMPLEXITY V. Krupski Lecture Notes, draft



В качестве временной сложности булевой схемы S используют размер схемы $size(S)$ – длину программы (т.е. количество операторов) в первом случае и количество помеченных символами операций вершин – во втором. Легко заметить, что по отношению к временной сложности эти два представления "эквивалентны"(в программе можно переименовать переменные так, чтобы имена переменных в левых частях операторов не повторялись, а программа вычисляла ту же функцию с той же временной сложностью).

Схемной сложностью функции $f : \{0,1\}^n \to \{0,1\}^m$ называется

$$c(f) = \min\{size(S) \mid S \text{ вычисляет } f\}.$$

Эта характеристика зависит от выбора полной системы F, т.е. $c(f) = c_F(f)$, но зависимость не очень существенна: при переходе к другой полной системе схемная сложность функций меняется на множитель O(1). В дальнейшем полную систему F будем считать фиксированной.

Универсальная для языка булевых схем машина Тьюринга стоится полностью аналогично случаю RAM, но с соответствующими упрощениями (нет необходимости отдельно хранить номер текущей команды, не надо моделировать `IF`). Оценка времени работы моделирующей машины Тьюринга в этом случае будет

$$TIME = O(\ (size(S))^2 \cdot \log size(S)\ ). \tag{4.2}$$

Она получается аналогично (4.1). Т.к. количество переменных ограничено сверху размером схемы, а на хранение одной переменной достаточно

$$\log size(S) + const$$

клеток ленты (не более $\log size(S)$ на номер переменной и $const$ на булево значение и разделители), то для хранения текущих значений переменных уходит $s = size(S) \cdot \log size(S)$ клеток ленты.

V. Krupski
COMPLEXITY
Lecture Notes, draft

Моделирование исполнения одного оператора сводится к поиску значений операндов ($O(s)$ шагов), вычисления значения правой части оператора (булева операция вычисляется за $O(1)$ шагов) и записи результата в переменную, стоящую в правой части оператора (еще $O(s)$ шагов). Перемещение головки на "программной"ленте к следующему оператору дает незначительный вклад $O(\log size(S))$ шагов. Итого, на один оператор моделируемой программы уходит $O(s)$ шагов, а всего операторов $size(S)$, что и дает требуемую оценку.

Замечание. На самом деле логарифмический сомножитель в (4.2) можно убрать, организовав хранение значений всех переменных без имен (номеров). Поиск значения переменной номер i в этом случае осуществляется выбором i-того значения от начала. Этот же метод может быть применен в случае BASIC, но не проходит при добавлении ссылок.

V. Krupski COMPLEXITY Lecture Notes, draft