Алгоритмы генерации эффективного кода

Д.Ю.Булычев
db@tepkom.ru
Санкт-Петербургский государственный университет
198504, Университетский пр., 28,
Санкт-Петербург, Россия

Аннотация

Генерация эффективного кода — одна из важнейших задач в области реализации языков программирования и разработки трансляторов. Первые результаты в данном направлении были получены в середине 60-х — начале 70-х годов и касались, в основном, вопросов порождения оптимального кода для некоторых классов машинных архитектур. Тогда же были сформулированы главные результаты, касающиеся труднорешаемости рассматриваемой задачи. Позднее основной упор был сделан на разработку схем настраиваемой (*retargetable*) генерации кода, при которой эффективный кодогенератор получается автоматически из некоторого описания. В данной работе рассмотрены основные подходы в области генерации эффективного кода и дан обзор основанных на них современных средств.

Введение

Если через $[p]_L$ обозначить семантику программы p в языке L, то в самом общем виде задачу порождения кода можно сформулировать как реализацию сохраняющего семантику отображения

$$gen : H \to L,\ \forall p\ ([p]_H = [gen(p)]_L)$$

где H и L – соответственно исходный язык высокого уровня и машинный язык (система команд). Имея на множестве машинных программ некую численную функцию (стоимость) c можно говорить, что данный способ генерации кода gen оптимален относительно c тогда и только тогда, когда

$$\forall gen' : H \to L,\ \forall p\ (c(gen(p)) \leq c(gen'(p)))$$

В качестве стоимости целевой программы могут рассматриваться различные величины — например, число машинных инструкций, размер программы, время ее исполнения и т.д.

В приведенной формулировке задача нахождения оптимального кода оказывается неразрешимой — в противном случае, например, можно было

бы подобрать такую систему команд и такую функцию стоимости, чтобы произвольная всегда зацикливающаяся программа отображалась бы оптимальным алгоритмом в инструкцию

```
loop: jmp loop
```

что доставляло бы алгоритм распознавания зацикливаемости. Поэтому в реальной жизни ставятся гораздо более скромные задачи.

Прежде всего, практически всегда рассматриваются не произвольные алгоритмы генерации кода, а их специальный класс, при котором целевая программа имеет тот же поток управления, что и исходная. Иными словами это означает, что исходная программа сначала переводится в машинный код локально на уровне линейных участков, а затем уже собирается из этих фрагментов путем порождения команд перехода и согласования размещения данных. Поскольку на линейном участке программа представляет собой набор выражений, генерация эффективного кода для выражений становится основной задачей.

Что касается функции стоимости, то в настоящее время наиболее актуальной метрикой программ при их реализации является время исполнения. Однако это время вообще говоря зависит от значений входных данных, так что обоснование оптимальности алгоритма кодогенерации в этом случае оказывается затруднительным. Поэтому обычно в качестве стоимости целевой программы выступает некоторое эвристически вычисленное значение, а применимость эвристики обосновывается экспериментально. В силу такого положения вещей далее вместо термина “оптимальный” мы часто будем использовать термин “эффективный”.

Задача порождения кода для программы может быть логически разделена на следующие подзадачи:

Выбор инструкций (*instruction selection*). Современные системы команд предоставляют большое разнообразие операций, режимов адресации и способов хранения значений (собственно говоря, это является одной из причин сложности генерации эффективного кода). Соответственно, возникает задача выбора наиболее предпочтительного варианта машинной инструкции, что включает в себя определение размещения операндов и выбор способа адресации.

Распределение регистров (*register allocation*). С точки зрения системы команд как правило доступно два способа размещения значений — память и регистр. Оправданность этой модели следует из того факта, что в программа всегда исполняется локально, то есть число значений, одновременно необходимых в каждой ее точке, невелико. Кроме того, увеличение числа внутренних регистров процессора приведет к тому, что для их обозначения потребуется большее число бит и, таким образом, накладные расходы на хранение и дешифрацию регистровых команд сведут на нет выигрыш в производительности памяти. Стоимость самой памяти при

этом становится все менее определяющим обстоятельством. Значительная разница во временных характеристиках между регистровой и общей памятью, а также объективно ограниченное количество регистров делает задачу их распределения чрезвычайно важной. Разброс временных характеристик программ при различных вариантах распределения регистров может достигать нескольких раз. Распределение регистров задевает результаты выбора инструкций, поскольку может оказаться, что выбранное "из лучших побуждений" размещение значения недопустимо в силу отсутствия необходимого регистра (или ячейки памяти). Наконец, результаты распределения регистров оказывают существенное влияние на следующую стадию — планирование. Именно, чем компактнее размещены значения по регистрам, тем хуже будут результаты планирования (т.к. тем более зависимы по данным окажутся фрагменты машинного кода). Таким образом укладывать программу в минимальное число регистров не всегда оказывается правильным.

Планирование инструкций (*instruction scheduling*). Важным свойством современных архитектур является их конвейеризованность — реальные процессоры совмещают во времени разные стадии исполнения инструкций и даже исполнение целых инструкций целиком. В такой ситуации число инструкций программы перестает коррелировать со временем ее исполнения — более длинная программа может исполняться быстрее короткой (примером преобразования, наиболее ярко демонстрирующего такое несоответствие, является раскрутка циклов). Кроме того, поскольку параллельное исполнение инструкций возможно только в случае отсутствия зависимостей этих инструкций по данным, такая общепринятая метрика, как число используемых регистров, также перестает быть актуальной — программа с меньшим числом используемых регистров оказывается более медленной, чем программа с большим их числом.

Таким образом, все эти подзадачи являются взаимозависимыми. Однако выбор инструкций наиболее чувствителен к системе команд, поскольку ассортимент конкретных машинных операций и размещений сильно меняется от архитектуры к архитектуре. В связи с этим мы ограничим рассмотрение в основном алгоритмами выбора инструкций. Хорошим библиографическим источником для остальных задач может служить, например, [25].

Далее будут рассмотрены основные известные подходы к построению генераторов кода для широкого класса регистровых архитектур, то есть архитектур с такими системами команд, в которых для каждой инструкции ее операнды указаны явно.

Помимо таких систем отдельный класс составляют *стековые* архитектуры, в которых инструкции оперируют с неявно заданными операндами, находящимися на вершине аппаратного стека. В последнем случае порядок исполнения машинных команд оказывается жестко задан, что препятствует их конвейеризации. В силу такого положения вещей стековые архитектуры оказались в стороне от основного направления развития вычислительной техники, хотя справедливости ради надо заметить, что имен-

но стековые системы команд получили наибольшее распространение в области разработки *виртуальных машин*, приспособленных для исполнения языково-оринтированного байт-кода (таковы, например, Java-машина и целое семейство виртуальных машин для функциональных языков). Тем не менее на уровне реальных вычислительных систем преобладает именно регистровая парадигма, так что стековый байт-код фактически выполняет роль переносимого промежуточного языка.

1 Основные понятия

Как было замечено выше, задача порождения эффективного кода допускает большое число различных постановок, охватывающих широкий диапазон проблем. В следующих разделах строится подходящая формальная модель, для которой и будут излагаться все результаты.

1.1 Программы и дэги

Для того, чтобы сформулировать интересующие нас задачи и обосновать свойства их решений, прежде всего нам потребуется определить, что есть программа и что есть ее семантика.

Мы можем считать, что программа представлена ее текстом на некотором эталонном императивном строго типизированном языке программирования, предоставляющем основные необходимые для записи произвольной программы элементы: присваивание, ветвление, арифметические выражения, переменные и т.д. Мы не будем подробно описывать этот язык в силу того, что все опускаемые нами детали восстанавливаются элементарным образом. Заметим, кроме того, что выбор такого языка фактически не ограничивает общезначимость обоснованных с его применением результатов, поскольку все реально используемые вычислительные устройства являются интерпретаторами подобных языков.

Как мы отмечали выше, в силу объективной трудности задачи генерации кода для программы в целом мы вынуждены ограничиться случаем генерации кода для ее линейного участка (то есть такого последовательного набора конструкций, который не допускает внутри себя ветвлений потока управления). С точки зрения представления программы на эталонном языке таким участком является группа последовательных присваиваний, и без ограничения общности мы можем считать, что семантика этой группы может быть описана заданием правил вычисления значений переменных-получателей. В свою очередь эти правила очевидно задаются перечислением того, какие операции выполняются над какими операндами и в какой последовательности.

Мы будем рассматривать такую ситуацию, при которой существо выполняемых операций не имеет для генерации кода никакого значения — важно лишь, чтобы в исходной и целевой программе они были одни и те же. Это возможно тогда, когда набор базовых операций в эталонном язы-

ке совпадает с таковым в системе команд. Такое предположение очевидно не является ограничительным, поскольку, во-первых, практически всегда так дело и обстоит, а во-вторых, можно эквивалентно выразить отсутствующие на уровне системы команд операции исходного языка через другие машинные операции (если этого сделать нельзя, то система команд оказывается очевидно неполной). Таким образом, с точки зрения генерации кода семантика генерируемого фрагмента программы — это упорядоченный набор *символов формальных операций*. Опишем это понятие более строго.

Далее символ $\mathbb{N}$ будет обозначать множество целых неотрицательных чисел, $dom(f)$ — область определения отображения f, $f|_X$, $X \subseteq dom(f)$ — сужение отображения f на множество X.

Пусть $F = (\Theta, Arity)$, где Θ — не более чем счетное множество,

$$Arity : \Theta \to \mathbb{N}$$

функция, которую будем называть *функцией арности*. Множества

$$Oper(F) = \{\theta \in \Theta \mid Arity(\theta) > 0\}$$
$$Opnd(F) = \{\theta \in \Theta \mid Arity(\theta) = 0\}$$

назовем соответственно *множествами операндов* и *операторов*. В дальнейшем ограничим рассмотрение только такими системами F, в которых множества операторов конечно, а множество операндов непусто.

Множество операторов — это и есть тот элементарный операционный базис, в котором выражаются действия, выполняемые программой во время ее работы, а функция арности приписывает каждому оператору количество его операндов. Множество операндов служит для обозначения операндов, каковыми в эталонном языке программирования в простейшем случае являются переменные и константы. Для целей же генерации кода в качестве множества операндов рассматривают множества *размещений* (или режимов адресации) целевого процессора. Таким образом, переменным эталонного языка перед генерацией кода должны быть назначены машинные размещения.

С точки зрения фрагмента для кодогенерации все переменные, находящиеся в позиции получателя присваивания, логически могут быть разделены на две группы. В первую группу попадают те из них, значения которых не используются за пределами данного фрагмента. Эти переменные, таким образом, хранят *локальные* или *временные* значения. Во вторую группу попадают переменные, хранящие остальные, *глобальные* значения. С точки зрения описания семантики интерес представляют переменные именно этой группы, поскольку именно их значения и являются окончательным результатом выполнения рассматриваемого фрагмента.

Локальные значения допускают большую свободу в обращении — их можно разместить в регистре, на стеке, в памяти или вообще не размещать нигде (если они являются промежуточными при исполнений какой-либо машинной инструкции). В отличие от них о глобальных значениях необходима дополнительная забота — по крайней мере изменение их размещений

должно быть учтено в местах использования, находящихся за пределами данной группы выражений. Кроме того, тип размещения глобального значения может быть жестко задан соображениями, не имеющими отношения к генерации кода (например, соглашениями о связях). Все это свидетельствует о необходимости разделения значений переменных на глобальные и временные. При этом надо учесть, что разные значения одной и той же переменной могут в разное время быть как глобальными, так и временными.

Поскольку распределение глобальных значений в существенной мере не зависит от выбора инструкций и чувствительно главным образом к потоку управления, соглашениям о связях и т.д. мы будем считать, что уже распределенные глобальные значения являются аргументами и результатами фрагмента для кодогенерации, а алгоритм генерации кода должен синтезировать такую программу, размещение агрументов и результатов которой согласованно с этим распределением.

После предварительного распределения глобальных значений кодогенератор получает свободу обращения с локальными значениями и, следовательно, с переменными, которые эти значения хранят. Заметим, что разбиение на переменные, выражения и присваивания с точки зрения семантики программы является в значительной мере случайным, обусловленным многими не относящимися к собственно смыслу программы соображениями — наглядностью, удобством написания и сопровождения и т.д. Поэтому самым правильным действием по отношению к ним является удаление их из рассмотрения — вместо этого можно *явным образом* указать, где какое значение используется. Для более строгой формулировки этого преобразования нам потребуются дополнительные конструкции.

Ориентированным графом назовем четверку $G = (V, E, beg, end)$, где V и E — конечные множества соответственно вершин и дуг,

$$beg : E \to V$$

$$end : E \to V$$

— всюду определенные функции. Если G — граф, то через $G.V$ и $G.E$ обозначим соответственно множества его вершин и дуг. Если $G.V = \emptyset$, то такой граф обозначим через Λ и назовем пустым. Для графов $G = (V_1, E_1, beg_1, end_1)$ и $H = (V_2, E_2, beg_2, end_2)$ их объединением назовем граф

$$G \cup H = (V_1 \cup V_2, E_1 \cup E_2, beg', end')$$

такой, что

$$beg'|_{V_{1,2}} = beg_{1,2}$$

$$end'|_{V_{1,2}} = end_{1,2}$$

Для произвольной дуги $e \in E$ вершина $beg(e)$ называется ее началом, а вершина $end(e)$ — ее концом. Далее условимся обозначать вершины гра-

фа строчными греческими буквами. Для произвольной вершины графа α определим множества

$$in(\alpha) = \{e \in E \mid end(e) = \alpha\}$$
$$out(\alpha) = \{e \in E \mid beg(e) = \alpha\}$$
$$succ(\alpha) = \bigcup_{e \in out(\alpha)} \{end(e)\}$$

соответственно входящих и исходящих из нее дуг и вершин-наследников. Далее мы будем использовать традиционное изображение графов в виде диаграмм, на которых каждой вершине соответствует точка, а каждой дуге e — стрелка, ведущая от точки, соответствующей вершине $beg(e)$ к точке, соответствующей вершине $end(e)$.

Вершина α достижима из вершины β (обозначение $\beta \rightarrow^* \alpha$) тогда и только тогда, когда либо $\alpha = \beta$ либо $\alpha = end(e)$ при условии, что $beg(e)$ достижима из β.

Будем говорить, что в графе существует контур, если в нем найдется пара различных взаимно достижимых вершин или петля (т.е. дуга e, такая, что $beg(e) = end(e)$).

Ориентированным ациклическим графом или *дэгом*[1] назовем ориентированный граф, не содержащий контуров. В произвольном дэге D выделяются множества вершин

$$Roots(D) = \{\alpha \mid in(\alpha) = \emptyset\}$$
$$Leaves(D) = \{\alpha \mid out(\alpha) = \emptyset\}$$

которые называются соответственно множествами корней и листьев. Для непустого дэга эти множества непусты. Дэг D называется *простым* в том и только том случае, когда $|Roots(D)| = 1$. В этом случае мы будем использовать выражение $root(D)$ для обозначения его единственного корня.

Вершины α и β назовем связанными, если либо они достижимы из одного и того же корня, либо найдется связанная с β вершина, достижимая из того же корня, что и α.

Дэг D называется *связным* тогда и только тогда, когда все его вершины, за исключением корней, попарно связаны.

Вершину дэга v назовем *сложной*, если $|in(v)| > 1$. Простой дэг называется *деревом* тогда и только тогда, когда в нем отсутствуют сложные вершины. Далее для деревьев мы будем использовать общеизвестную скобочную линейную запись, при которой наследники вершины перечисляются в скобках справа от нее.

Мы будем рассматривать только такие дэги, вершины которых помечены символами системы F, то есть для которых задано отображение $\mu : V \rightarrow \Theta$, такое, что $\forall \alpha \in V \; |out(\alpha)| = Arity(\mu(\alpha))$. Кроме того, множества входящих

[1] От английского DAG — Directed Acyclic Graph

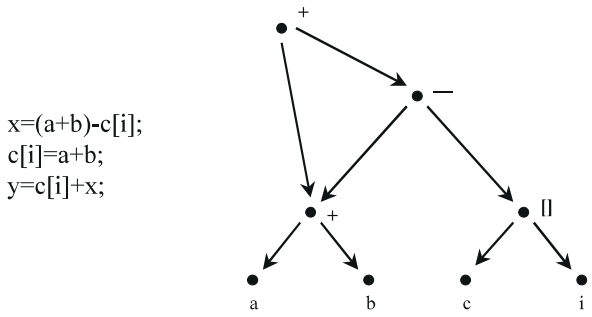

Рис. 1: Пример программы и ее дэга

и исходящих дуг для вершин мы будем считать упорядоченными — таким образом, деревья $x(\alpha, \beta)$ и $x(\beta, \alpha)$ считаются различными.

Далее множество дэгов, помеченных символами системы F, будем обозначать $\mathfrak{D}(F)$, множество деревьев — $\mathfrak{T}(F)$.

Неформально говоря, дэг есть способ представления потока данных для группы последовательных выражений. Листьями дэга являются аргументы этой группы (то есть константы или глобальные по отношению к этой группе переменные), а внутренними вершинами — операторы. Исполнение группы выражений в такой трактовке есть вычисление всех корней дэга. На рис. 1 приведен фрагмент программы и построенный для него дэг. Здесь переменные `y` и `c[i]` полагаются глобальными, переменная `x` — локальной. В качестве множества операторов выступают операции '`+`','`-`' и '`[]`', где '`[]`' символизирует доступ к элементу массива; множеством операндов являются распределенные по размещениям переменные. Присваивания глобальных переменных никак не отражены в собственно дэге; можно считать, что эти присваивания задают ограничения на размещение значений, вычисленных в некоторых его вершинах. Далее мы не будем обращать внимание на такие ограничения, поскольку их наличие не влияет на излагаемые результаты.

Заметим, что сложные вершины появляются в дэге из-за повторного использования некоторых величин, например, общих подвыражений и т.д. Таким образом, построение "хорошего" дэга для императивной программы вообще говоря может потребовать анализа потока данных. В данном случае видно, что при преобразовании в дэг была осуществлена экономия общего выражения `a+b` и удаление присваивания переменной `x`. Отметим, что представление в виде дэга удобно тем, что оно позволяет абстрагироваться от конкретных деталей реализации программы, поскольку сохраняет только содержательный поток данных вне зависимости от разбиения на присваивания, порядок вычисления выражений и т.д. Мы не будем подробно останавливаться на преобразовании исходной императивной программы в дэг; изложение этого и смежных вопросов может быть найдено, например,

в [5, 25, 24].

Теперь можно сформулировать суть нашего подхода к семантике фрагмента программы для кодогенерации: таковой семантикой является его дэг. Понятие эталонного языка программирования в этой ситуации оказывается излишним и постановка задачи генерации кода приобретает следующую простую форму: для данного дэга построить машинную программу, которая его вычисляет. Таким образом, теперь нам необходимо определить понятие машинной программы и ее значения.

1.2 Системы команд и программы

Инструкцией в системе F назовем выражение вида

$$`x \leftarrow t', \ x \in Opnd(F), \ t \in \mathfrak{T}(F)$$

при условии, что пометки всех листьев t попарно различны.

Для произвольной инструкции $I = `x \leftarrow t'$ определим объекты

$$Arg(I) = \{\mu(\alpha) \mid \alpha \in Leaves(t)\}$$
$$res(I) = x$$
$$body(I) = t$$

которые в дальнейшем будем называть соответственно *множеством аргументов*, *результатом* и *телом* инструкции I.

Произвольное множество инструкций

$$S = \{I_1, I_2, \ldots\}$$

назовем *системой команд*.

В используемой формализации каждая инструкция системы команд трактуется как присваивание некоторому операнду значения выражения, заданного деревом. С одной стороны это плохо согласуется с реальными системами команд, в которых практически все инструкции имеют побочные эффекты (установку флагов, изменение счетчика команд и т.д.). Однако эти побочные эффекты как правило влияют на те элементы архитектуры, прообраз которых отсутствует на уровне исходной программы, поэтому их рассмотрение в целях генерации кода имеет мало смысла.

Следует также заметить, что мы считаем различными инструкции, различающиеся хотя бы одним операндом. Это значит, например, что в системе команд с двумя регистрами `r1`, `r2` и командой `mov` регистровой пересылки между различными регистрами в нашей формализации появится *две различных* инструкции: `mov r1,r2` и `mov r2,r1`. Таким образом, выбор инструкций будет автоматически определять конкретные размещения всех операндов.

Командой системы S назовем дерево $c \in \mathfrak{T}(F)$, такое, что существует инструкция $I \in S$, что $body(I)$ с удаленными листьями совпадает с c. Неформально говоря, команда определяет инструкцию с точностью до размещений аргументов и результата. Далее для инструкции I через $command(I)$ будем обозначать соответствующую ей команду. Очевидно, что если тело инструкции состоит из единственного листа, то ее команда пуста. Все такие инструкции мы будем называть *пересылками*.

Вид системы команд имеет большое значение для алгоритма кодогенерации. Можно перечислить наиболее типичные варианты систем команд:

Однорегистровая система. Множество операндов состоит из символа регистра r и счетного множества символов ячеек памяти $m_1, m_2, \ldots$. Каждая инструкция имеет один из следующих видов:

- '$r \leftarrow m_i$' или
- '$m_i \leftarrow r$' или
- '$r \leftarrow t$', где один из листьев t помечен r

Система с равноправными регистрами. Множество операндов состоит из конечного набора символов регистров $r_1, \ldots, r_k$ и счетного множества символов ячеек памяти $m_1, m_2, \ldots$. Множество инструкций распадается на следующие классы:

- $\forall i, 1 \leq i \leq k$, '$r_i \leftarrow m_j$' или
- $\forall i, 1 \leq i \leq k$, '$m_j \leftarrow r_i$' или
- $\forall i, 1 \leq i \leq k$, '$r_i \leftarrow t$', где один из листьев t помечен r_i

Полная ортогональная система. Данная система команд удовлетворяет следующему ограничению: если инструкция '$x \leftarrow t$' принадлежит ей, то ей также принадлежат и все инструкции вида '$y \leftarrow t'$', где y — произвольный операнд, t' получено из t произвольной переразметкой листьев t произвольными символами операндов. Иными словами, в полной ортогональной системе команд машинные команды существуют для любых сочетаний операндов.

Заметим, что выше мы не требовали от системы команд возможности адресовать произвольную ячейку памяти. Далее мы неявно всегда будем предполагать наличие такого свойства. Формально это будет обозначать бесконечность числа команд, однако с точки зрения реального положения вещей это не более грубое допущение, чем допущение об универсальности современных вычислительных устройств, в то время как на самом деле они представляют собой конечные автоматы. Кроме того, мы будем считать, что количество собственно команд в любой системе всегда конечно.

Программой в системе команд S будем называть упорядоченную последовательность инструкций

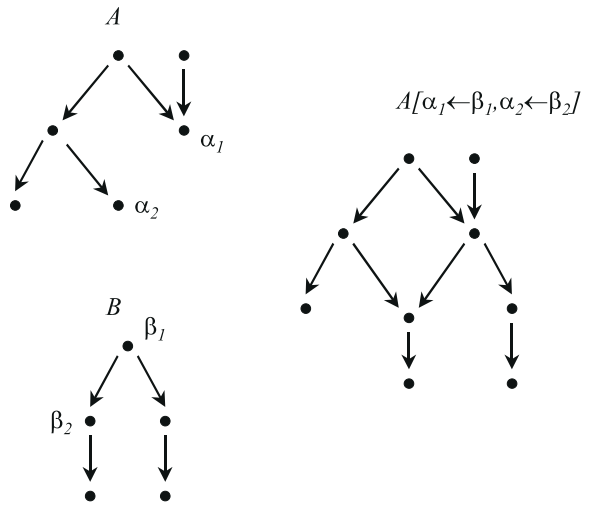

Рис. 2: Пример подстановки одного дэга в другой

$$P = I_1 I_2 \dots I_n$$

каждая из которых принадлежит S. Множество всех программ в системе команд S обозначим $\mathfrak{P}(S)$. Количество инструкций в программе P назовем ее длиной и обозначим $|P|$. Для произвольной инструкции I_i программы P определим множество

$$Use_P(I_i) = \{I_j \mid j > i,\ res(I_i) \in Arg(I_j),\ \forall i < k < j\ res(I_k) \neq res(I_i)\}$$

инструкций, использующих I_i.

Пусть I — инструкция программы P. Определим для её результата преобразование *согласованного переименования* следующим образом: возьмем некоторый операнд x и заменим I на инструкцию '$x \leftarrow body(I)$', а каждую инструкцию $J \in Use_P(I)$ — на инструкцию $res(J) \leftarrow t$, где t получено из $body(J)$ заменой листа, помеченного символом $res(I)$ на лист, помеченный символом x. Очевидно, что это преобразование определено для данного x и данной инструкции I только тогда, когда в системе команд существуют инструкции, которые получаются после описанной замены.

Пусть заданы дэг A, набор (не обязательно различных) дэгов $B_1, \dots, B_k$ и фиксированы вершины $\{\alpha_1, \dots, \alpha_k\} \subseteq Leaves(A)$ и $\beta_1 \in B_1, \dots, \beta_k \in B_k$. *Подстановкой* $B_1, \dots, B_k$ в A относительно вершин $\alpha_1, \dots, \alpha_k$ и $\beta_1, \dots, \beta_k$ назовем граф

$$A[\alpha_1 \leftarrow \beta_1, \dots, \alpha_k \leftarrow \beta_k]$$

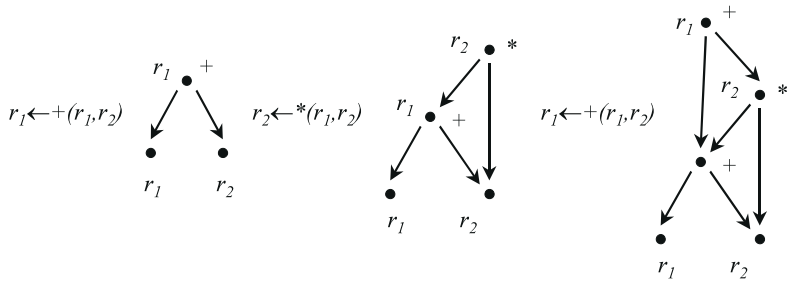

Рис. 3: Пример исполнения программы

полученный из A перенаправлением всех дуг, ведущих к вершинам α_i, на вершину β_i с последующим удалением изолированной теперь вершины α_i. Пример подстановки приведен на рис. 2.

Легко видеть, что если дэг A отличен от каждого из B_i, то полученный граф также является дэгом.

Назовем *состоянием* пару $\sigma = (D, v)$, где $D \in \mathfrak{D}(F)$ — произвольный дэг, $v : Opnd(F) \to D.V \cup \{\perp\}$ — отображение, сопоставляющее каждому операнду вершину D или специальный элемент $\perp$. Для инструкции I и состояния σ определим *эффект* инструкции как состояние $\sigma' = (D', v')$, определенное следующим образом:

$$D' = D \cup body(I)[l_1 \leftarrow v(\mu(l_1)), \ldots, l_k \leftarrow v(\mu(l_k))]$$

где $\{l_1, \ldots, l_k\} \subseteq Leaves(body(I))$, такие, что $v(\mu(l_i)) \neq \perp$,

$$v'(x) = v(x), \text{ если } x \neq res(I)$$

$$v'(res(I)) = root(body(I))$$

Очевидно, что это определение корректно, поскольку результат подстановки дэгов вместо листьев в дерево есть дэг. Далее обозначение $\sigma I \sigma'$ будем использовать для отражения того факта, что σ' есть эффект инструкции I в состоянии σ.

Пусть $P = I_1 I_2 \ldots I_k$ — произвольная программа. *Финальным состоянием* программы P назовем состояние σ_k, удовлетворяющее следующему условию:

$$(\sigma_0 = (\Lambda, v_0)) I_1 \sigma_1, \sigma_1 I_2 \sigma_2, \ldots, \sigma_{k-1} I_k \sigma_k$$

где $\forall x \in Opnd(F)\ v_0(x) = \perp$. Для финального состояния $\sigma_k = (D_k, v_k)$ *значением* программы P назовем дэг $val(P) = D_k$.

Неформально говоря, состояние приписывает каждому операнду его значение при исполнении программы. При этом символ $\perp$ используется для отражения того факта, что это значение вычислено за пределами данной программы (и, следовательно, может трактоваться как ее аргумент).

Каждая инструкция добавляет к состоянию свой результат, подставляя в свое тело значения аргументов. Пример исполнения программы приведен на рис. 3.

Теперь мы можем ввести следующее определение: программа P вычисляет дэг D в том и только том случае, когда $val(P) = D$.

Рассмотрим численную функцию стоимости c, заданную на множестве программ и произвольный дэг $D \in \mathfrak{D}(F)$. Будем говорить, что программа P является оптимальной программой, вычисляющей D, тогда и только тогда, когда выполнены оба нижеследующих условия:

1. P вычисляет D
2. $c(P) \leq c(P')$ для произвольной программы P', вычисляющей D

Инструкцию I программы P назовем *бесполезной*, если $val(P) = val(P')$, где P' получена из P удалением I. Легко можно показать, что если I — бесполезна, то либо $Use_P(I) = \emptyset$, либо $I = `x \leftarrow x'$(то есть I — бесполезная пересылка). Далее мы всегда будем рассматривать программы, не содержащие таких инструкций, поскольку при любой разумной функции стоимости программы, содержащие бесполезные инструкции, "не лучше", чем не содержащие.

Для данной системы F и системы команд S определим *алгоритм генерации кода* как отображение

$$gen : \mathfrak{D}(F) \to \mathfrak{P}(F)$$

удовлетворяющее условию

$$\forall D \in \mathfrak{D}(F)\ val(gen(D)) = D$$

Для данной функции стоимости c алгоритм генерации кода gen назовем оптимальным в том и только том случае, когда для произвольного дэга $D \in \mathfrak{D}(F)$ $gen(D)$ — оптимальная программа.

1.3 Покрытие и линеаризация

Программа, вычисляющая дэг, индуцирует в нем некоторую структуру, к рассмотрению которой мы и переходим.

Из определения значения программы, данного в предыдущем разделе, видно, что оно получается "склеиванием из кусочков", каждым из которых является тело инструкции. При этом листья тел одних инструкций склеиваются с корнями тел других. Определим эту конструкцию более формально.

Рассмотрим дэг D и некоторый набор деревьев $p = \{t_1, \ldots, t_k\}$. *Покрытием* D образцами из p назовем пару (c, ξ), где

$$c = (V, E, beg, end)$$

— дэг,

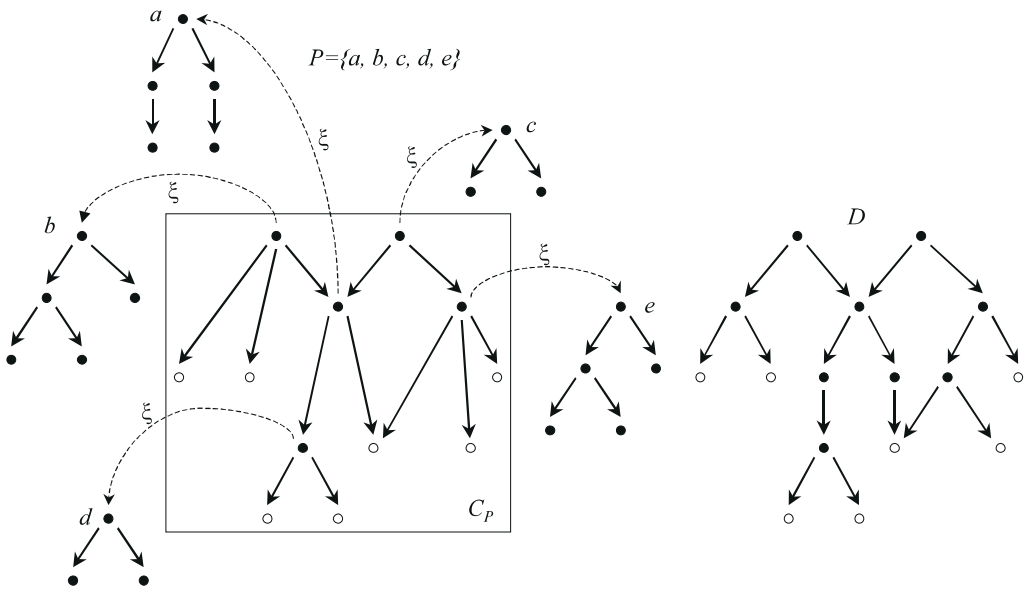

Рис. 4: Пример покрытия дэга образцами

$$\xi : V \setminus Leaves(c) \to p$$

— взаимно-однозначное отображение, имеющее следующие свойства:

1. $\forall \alpha \in V \; |Leaves(\xi(\alpha))| = |out(\alpha)|$
2. $\bigcup\limits_{\alpha \in Roots(c)} (\alpha[\xi]) = D$, где для $\forall \alpha \in V \; \alpha[\xi]$ определено следующим образом:

   - $\alpha[\xi] = \alpha$, где $\alpha \in Leaves(c)$
   - $\alpha[\xi] = \xi(\alpha)[\lambda_1 \leftarrow root(\tau_1[\xi]), \ldots, \lambda_k \leftarrow root(\tau_k[\xi])]$, где

$$\{\lambda_1, \ldots, \lambda_k\} = Leaves(\xi(x))$$

$$\{\tau_1, \ldots, \tau_k\} = succ(v)$$

Неформально говоря, покрытие есть факторизация исходного дэга по набору образцов-деревьев. Пример покрытия показан на рис. 4.

Для дальнейшего продвижения нам потребуется еще одно понятие. *Топологической сортировкой* называется такой линейный порядок '$\leq$' на вершинах графа, при котором для произвольных вершин α, β из $\beta \in succ(\alpha)$ следует $\alpha \leq \beta$. Иными словами, топологическая сортировка согласована с направлением дуг графа. Очевидно, что для дэга топологическая сортировка существует всегда. *Линеаризацией* назовем такой линейный порядок на

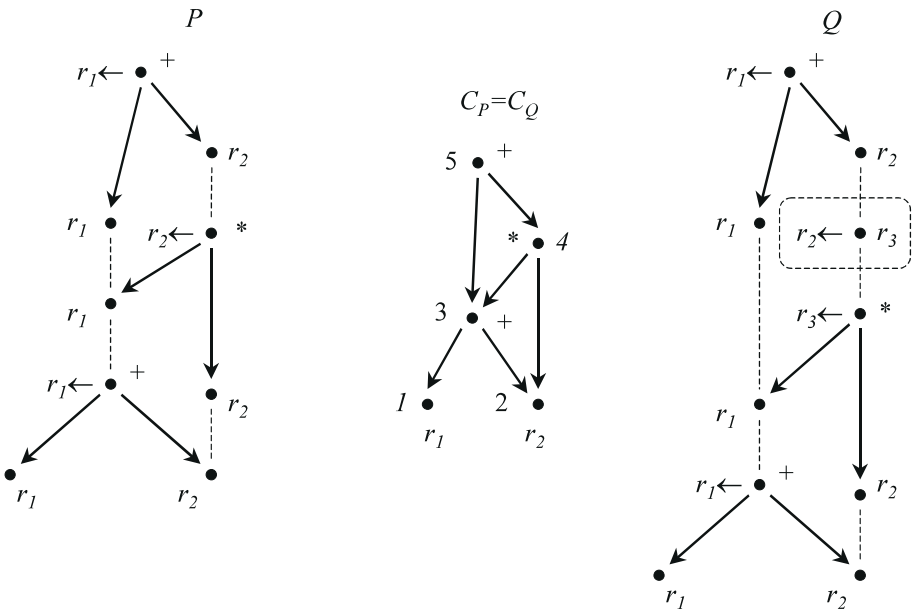

Рис. 5: Две программы для одного покрытия

вершинах дэга, обратный к которому является его топологической сортировкой. Очевидно, что при обходе дэга в порядке, задаваемом линеаризацией, каждая вершина посещается после того, как посещены все вершины, достижимые из нее.

Пусть есть программа $P = I_1 I_2 \dots I_k$, вычисляющая дэг D. Легко можно показать, что P определяет покрытие C_P дэга D, в котором в качестве образцов выступают непустые команды, соответствующие инструкциям $I_1, I_2, \dots, I_k$. При этом каждой инструкции P соответствует не более одной вершины C_P, а порядок следования инструкций в программе соответствует некоторой линеаризации C_P. Инструкциями, которым не соответствуют никакие вершины покрытия, являются *пересылки*, то есть такие инструкции, команды которых пусты. Отсюда следует, что покрытие дэга соответствует программе *с точностью до пересылок*.

В силу произвольности программы P можно сделать следующий вывод: линеаризация покрытия исходного дэга командами в данной системе команд с точности до пересылок определяет программу, вычисляющую этот дэг.

На рис. 5 показано покрытие дэга и две разные программы, соответствующие ему. Видно,что эти программы различаются друг от друга одной пересылкой.

Заметим, что вообще говоря не для любого покрытия существет программа, которая его определяет — наличие этого свойства зависит от струк-

туры системы команд. Мы далее будем предполагать, что из любого покрытия может быть построена хотя бы одна программа. В неформальном плане это означает, что система команд предоставляет достаточный ассортимент размещений, чтобы можно было, вставляя пересылки, осуществить корректную передачу данных между образцами покрытия. Далее, несмотря на то, что вообще говоря одно покрытие может определяеть бесконечное множество программ (поскольку пересылок можно вставить сколько угодно), разумно ограничиться только некоторым конечным их числом, поскольку существует конечное число способов самого короткого преобразования размещения данных.

От чего зависит стоимость программы, которая может быть восстановлена из покрытия? Эта стоимость, очевидно, зависит от количества образцов в покрытии, поскольку каждый образец определяет одну команду. Кроме того, она зависит от того, сколько пересылок потребуется вставить, чтобы сохранить корректную передачу данных. Количество этих пересылок, в свою очередь, зависит как от покрытия, так и от его линеаризации.

Пусть '$\leq$' — линеаризация дэга D. *Шириной* '$\leq$' назовем число

$$width(\leq) = \max_{\alpha \in D.V} |\{e \in D.E \setminus (out(\alpha) \cup in(\alpha)) \mid \alpha \leq beg(e) \& end(e) \leq \alpha\}|$$

Неформально говоря, ширина линеаризации описывает максимальное количество "транзитных" значений, которые переносятся дугами дэга через все его вершины. Таким образом, ширина определяет максимальное количество одновременно существующих значений при исполнении программы.

На рис. 6 приведен пример дэга и двух его линеаризаций. Ширина верхней линеаризации есть 1, поскольку для каждой ее вершины существует не более одной "транзитной" дуги, ширина нижней — 2, поскольку для вершины γ таких дуг две.

В общем случае, к сожалению, ни количество образцов покрытия, ни ширина линеаризации не позволяют оценить сложность программы независимо от функции стоимости и системы команд. При определенных обстоятельствах программы с большим числом образцов могут быть эффективнее программ с меньшим их числом; то же справедливо и для ширины. Таким образом, алгоритм генерации кода для каждой системы команд должен решить некоторую оптимизационную задачу в пространстве всевозможных пар покрытие-линеаризация. Нам, следовательно, необходим как минимум способ конструктивного описания этого пространства.

Поскольку перечисление линеаризаций является тривиальной задачей, мы сконцентрируемся на задаче перечисления покрытий. Заметим предварительно, что по определению в каждом покрытии дэга образцами сложным вершинам дэга могут соответствовать только корни или листья образцов. Следовательно, без ограничения общности можно предварительно "расклеить" исходный дэг по сложным вершинам на набор деревьев, затем найти покрытие для каждого дерева этого набора и затем восстановить из этих покрытий покрытие исходного дэга. Строго говоря, сужение любого покрытия

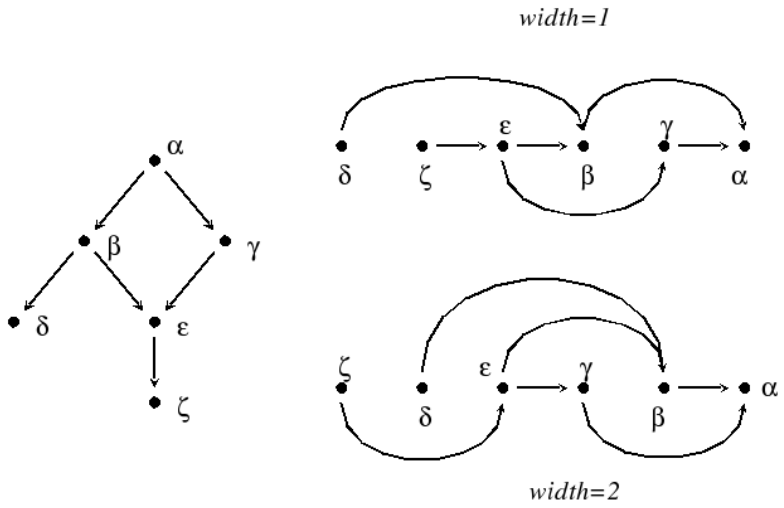

Рис. 6: Покрытие, его линеаризации и их ширины

дэга на дерево внутри него, ограниченное корнями, листьями или сложными вершинами, является покрытием этого дерева; аналогично, объединение покрытий всех таких деревьев есть покрытие дэга.

На рис. 7 приведен пример декомпозиции дэга на деревья. Видно, что покрытие того же дэга, показанное на рис. 4, получается объединением покрытий этих деревьев.

Таким образом, нахождение покрытия дэга сводится к его однозначной декомпозиции на деревья и нахождению покрытий для этих деревьев. К решению этой задачи мы и переходим.

2 Деревянные грамматики и покрытия

Рассмотрим некоторую систему $F = (Arity, \Theta)$ и выделим непустое множество $N \subseteq Opnd(F)$. Деревянной грамматикой (*tree grammar*) в системе F называется контекстно-свободная грамматика $G = (N, T = \Theta \setminus N, R, S)$, где N, T — непересекающиеся множества соответственно нетерминалов и терминалов, $S \in N$ — выделенный стартовый нетерминал, R — множества правил вида

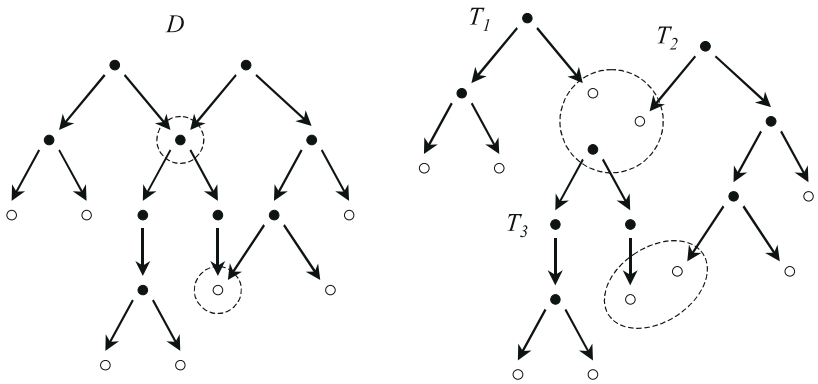

Рис. 7: Декомпозиция дэга на деревья для построения покрытия

$$K : t,\ K \in N,\ t \in \mathfrak{T}(F)$$

Таким образом, по сравнению с обычными контекстно-свободными грамматиками у деревянных грамматик допускаются правила, в правой части которых стоит *деревянный образец*. В качестве деревянного образца выступает дерево, у которого некоторые листья помечены нетерминалами, а все остальные вершины — терминалами. Частными случаями правил, таким образом, являются правила вида

$$K : x,\ K \in N,\ x \in T$$

и

$$K : M,\ \{K, M\} \subseteq N$$

Подобно обычным грамматикам деревянные грамматики также определяют некоторый язык, только в качестве слов этого языка выступают не последовательные цепочки терминалов, а помеченные терминалами деревья. Опишем эту конструкцию более подробно.

Пусть p — некоторый деревянный образец, $r = K : t \in R$ — правило грамматики. Будем говорить, что образец q получен из образца p применением правила r (обозначение $p \xrightarrow{r} q$), тогда и только тогда, когда существует лист $\alpha \in Leaves(p)$, помеченный нетерминалом K, такой, что

$$q = p[\alpha \leftarrow root(t)]$$

Иными словами, применение правила подставляет в образец вместо листа, помеченного нетерминалом, образец в правой части правила для того же нетерминала.

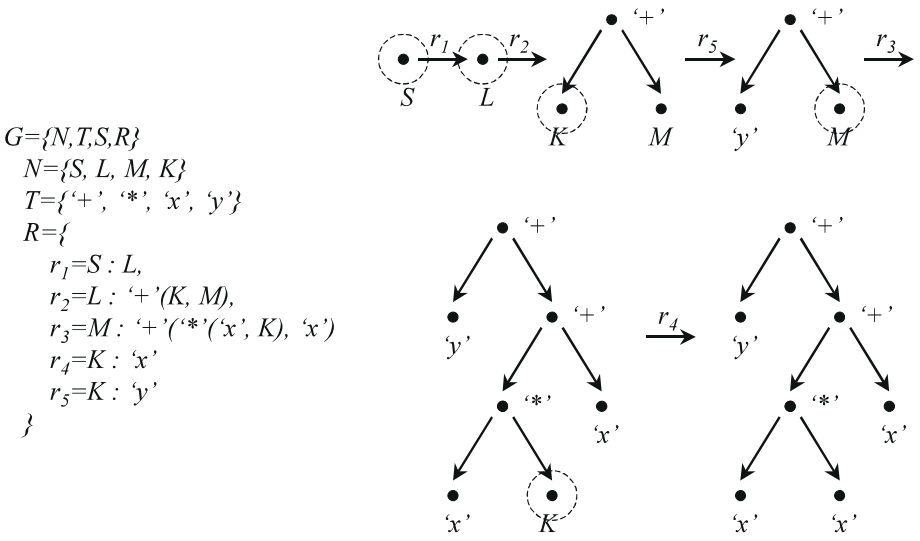

Рис. 8: Пример вывода дерева в грамматике

Пусть t — дерево, вершины которого помечены только терминалами. Будем говорить, что t *выводится* из нетерминала K в грамматике G, если для него существует вывод — последовательность применений правил $r_1, r_2, \ldots, r_k$, удовлетворяющая условию

$$K \xrightarrow{r_1} \ldots \xrightarrow{r_k} t$$

Неформально говоря, вывод определяет последовательность применений правил грамматики к образцам, начиная с одного исходного нетерминала, приводящую в конечном итоге к заданному дереву. Множество всех деревьев, выводимых из нетерминала K, обозначим через $\mathfrak{L}(K)$. *Языком*, порождаемым деревянной грамматикой G, назовем множество $\mathfrak{L}(G) = \mathfrak{L}(S)$.

Деревянные грамматики, с одной стороны, являются расширением обычных "линейных" грамматик, поскольку допускают в правой части правил не только линейное слово в объединенном алфавите терминалов и нетерминалов, но и дерево. С другой стороны, деревянные грамматики описывают только *автоматные* языки (поскольку в них нет аналогов правилам вида $K : MT$), хотя для их распознавания требуются специальные деревянные автоматы. Подробнее с теорией деревянных языков и автоматов можно познакомиться в [10].

На рис. 8 показан пример вывода дерева в деревянной грамматике. Заметим, что вывод для дерева может определяться неоднозначно, поскольку одни и те же правила могут применяться к листьям образцов в разном порядке. Далее мы будем считать, что очередное правило всегда применяется к самому "левому" листу текущего образца во избежание несущественных

неоднозначностей.

Очевидно, что вывод дерева в деревянной грамматике определяет его покрытие образцами, каждый из которых есть правая часть некоторого правила. Нашей задачей, следовательно, является построение для системы команд грамматики, такой, что для произвольного дерева каждому его покрытию командами данной системы соответствует некоторый вывод. Кроме того, нас будут интересовать только конечные грамматики, у которых множества нетерминалов и правил конечны.

Поскольку множество операндов системы команд может быть бесконечно, для построения грамматики разобъем его на множество *классов размещений* $C_1, C_2, \ldots$ по следующему принципу: операнды x и y попадают в один класс тогда и только тогда, когда

1. для любой инструкции $x \leftarrow t$ существует инструкция $y \leftarrow t$
2. для любой инструкции $z \leftarrow t$ существует инструкция $z \leftarrow t'$, где t' получается из t заменой листа, помеченного x, на лист, помеченный y

В силу конечности множества команд количество полученных таким разбиением классов также будет конечно. Например, для рассматривавшихся классов систем команд разбиение на классы размещений таково:

- однорегистровой системе команд соответствуют два класса — в первый попадает единственный представитель-регистр r, во второй — все ячейки памяти;
- системе с равноценными регистрами также соответствует два класса — все регистры, все ячейки памяти;
- в полной ортогональной системе все операнды составляют один класс.

Неформально говоря каждый класс состоит из множества взаимозаменяемых операндов — любой из операндов данного класса можно заменить в любой содержащей его команде на любой другой операнд этого же класса. Для операнда x через $C(x)$ обозначим класс, которому он принадлежит.

Теперь мы готовы описать нужную нам грамматику. В качестве множества терминалов возьмем объединенное множество операторов и операндов системы команд $\Theta = Oper(F) \cup Opnd(F)$, в качестве множества нетерминалов — множество классов размещений $C_1, C_2, \ldots, C_m$.

Множество правил грамматики состоит из четырех частей. Первая часть образована правилами вида $C_i : t$, соответствующими инструкциям $I = \text{`}x \leftarrow p\text{'}$, где

- $x \in C_i$
- $t = p[l_1 \leftarrow C(l_1), \ldots, l_k \leftarrow C(l_k)]$, где $l_i \in Leaves(p)$

Вторая часть образована цепными правилами $C(x) : C(y)$ для каждой пересылки $x \leftarrow y$.

Третья часть состоит из правил вида $C(x) : x$ для каждого операнда $x \in Opnd(F)$.

Наконец, во множество нетерминалов добавляется выделенный стартовый нетерминал S, а ко множеству правил добавляются все правила вида $S : C_i$.

Видно, что грамматика получается из системы команд факторизацией по разбиению, задаваемому классами размещений.

Строго говоря, построенная таким образом грамматика не обязана иметь конечное число правил, поскольку она подразумевает наличие для каждого операнда как минимум одного правила. Мы не будем обращать на это несоответствие внимания, поскольку в реальности число операндов программы всегда конечно.

Можно показать, что построенная грамматика действительно обладает тем свойством, что, во-первых, вывод в ней данного дерева определяет покрытие в системе команд, и, во-вторых, для любого покрытия в системе команд найдется определяющий его вывод.

Таким образом, перед нами встает задача нахождения всех выводов для данного дерева в данной грамматике.

Для решения этой задачи прежде всего приведем деревянную грамматику к более простому виду. Именно, будем говорить, что деревянная грамматика находится в *нормальной форме*, если все ее правила имеют один из двух видов:

- $K : x,\ K \in N,\ x \in T$
- $K : x(M_1, \ldots, M_k),\ \{K, M_1, \ldots, M_k\} \subseteq N,\ x \in T$

Произвольную грамматику всегда можно преобразовать в грамматику в нормальной форме, определяющую тот же язык, что исходная. Действительно, сначала необходимо стандартным образом удалить цепные правила, а затем каждое правило вида $K : x(\ldots, t_i, \ldots)$, где t_i не является нетерминалом, заменять на пару правил $K : x(\ldots, L, \ldots)$ и $L : t_i$, где L — свежий нетерминал, до тех пор, пока таких правила не останется.

Очевидно, что вывод в исходной грамматике можно однозначно восстановить из вывода в приведенной. Исходя из этих соображений, мы будем считать, что все грамматики находятся в нормальной форме.

Перейдем к описанию алгоритма построения вывода. Заметим, что если дерево имеет вывод в грамматике в нормальной форме, то этот вывод можно представить разметкой, сопоставляющей каждой вершине дерева правило грамматики, поскольку при применении любого правила в выводе высота дерева увеличивается не более чем на единицу. Таким образом все такие разметки доставляют все выводы данного дерева в данной грамматике, и интересующий нас алгоритм есть в сущности алгоритм нахождения такой разметки.

```
    boolean labels (N, v) :
01    for each r in label(v) do
02      if r==’N:p’ return true;
03    done
04    return false;

    labeling (v) :
01    for each w in succ(v) do labeling(w);
02    for each r in G.R do
03      if r=’K:x’ and terminal(v) == x and leaf(v)
04      then label(v) += {r}
05      else if
06        r=’K:t(L(1),...,L(k))’ and
07        terminal(v) == t       and
08        (for i=1 to k) labels(L(i), son(v, i))
09      then label(v) += {r}
```

Рис. 9: Построение выводов в деревянной грамматике: разметка

2.1 Построение выводов

Алгоритм нахождения всех разметок разметки приведен на рис. 9. Вспомогательная функция `labels` получает корень некоторого поддерева и определяет, выводится ли это поддерево из данного нетерминала — для этого среди пометок данной вершины должно найтись правило с этим нетерминалом в левой части.

Основная работа алгоритма сконцентрирована в функции `labeling`, которая получает параметром корень некоторого поддерева и строит для него разметку. Данная функция, прежде всего, применяет себя ко всем сыновьям текущей вершины (строка `01`), а затем добавляет в разметку те правила, которые можно применить к данной вершине. Эти правила могут быть двух видов. Первый случай рассмотрен в строках `03-04`: правило $K : x$ можно применить к вершине, если она является листом и ее терминальная пометка есть x. Второй случай охватывают строки `05-09`: если правило есть $K : t(L_1, \ldots, L_k)$, то его можно применить в том и только том случае, когда текущая вершина помечена терминалом t, а соответствующие сыновья помечены правилами, выводящими нетерминалы $L_1, \ldots, L_k$.

Можно показать, что данный алгоритм добавляет в разметку вершины α правило r тогда и только тогда, когда поддерево с корнем α принадлежит $\mathfrak{L}(K)$, где K — нетерминал из левой части r.

После построения разметки из нее могут быть извлечены все выводы. Действительно, пусть нам дана вершина α и некоторый предзаданный нетерминал K, происхождение которого мы объясним позже. Легко построить начальный фрагмент *всех* выводов поддерева с корнем α из нетерминала K: для этого достаточно перебрать все правила в разметке α, которые

```
    reduce (K, v) :
01    if leaf(v) then
02      begin
03        r=any_of label(v) where r='K:x';
04        append(v, r)
05      end
06    else begin
07        r=any_of label(v) where r='K:x(L(1),L(2),...,L(k))';
08        append(v, r);
09        for i=1 to k do reduce(L(i),son(v,i))
10    end
```

Рис. 10: Построение выводов в деревянной грамматике: свертка

имеют K в своей левой части. Каждое такое правило начинает самостоятельный вывод поддерева с корнем α из K. Далее, если правило имеет вид $K : x$, то вершина α — лист с пометкой x и здесь вывод заканчивается. В противном случае правило имеет вид $K : x(L_1, L_2, \ldots, L_k)$ и вывод можно продолжить, выбрав произвольный вывод i-го сына α из нетерминала L_i. Таким образом, выбор правила в текущей вершине определяет предзаданные нетерминалы для всех ее сыновей. Очевидно, что для получения всех выводов данного дерева в качестве предзаданного нетерминала для его корня следует выбрать S.

Функция определения некоторого вывода по разметке (*свертка*) приведена на рис. 10. Здесь примитив `any_of...where` позволяет выбрать произвольный элемент множества, удовлетворяющий некоторому условию, `append(v,r)` пополняет вывод применением правила r к вершинет α. Заметим, что поскольку мы условились применять в выводе правило к самому левому сыну образца, этой информации достаточно для полного восстановления вывода.

Видно, что для фиксированной грамматики алгоритм построения вывода имеет сложность $O(|V|)$, где V — множество вершин дерева.

Продемонстрируем применение описанного подхода на примере.

Рассмотрим систему F, в которой множество операторов есть

$$Oper(F) = \{'+', fetch, index\}$$

а множество операндов —

$$Opnd(F) = \{r_1, \ldots, r_k\} \cup \{m_1, m_2, \ldots\} \cup \{1, 2, 3 \ldots\}$$

Неформально говоря, множество операндов состоит из конечного числа регистров и счетного числа констант и ячеек памяти. Система команд состоит из пяти групп инструкций:

1. $r_i \leftarrow$ ' + '(r_i, r_j) — регистрового сложения;

2. $r_i \leftarrow fetch(m_j)$ — загрузки из памяти в регистр;

3. $r_i \leftarrow index(m_j, r_k)$ — загрузки с индексацией из памяти в регистр;

4. $r_i \leftarrow 1, 2, 3, \ldots$ — загрузки константы в регистр;

5. $m_i \leftarrow r_j$ — выгрузки из регистра в память.

Построим для данной системы команд деревянную грамматику. Все операнды разбиваются на три класса размещений — регистры, память и константы (непосредственные операнды). Соответствующие нетерминалы обозначим через Reg, Mem и Imm. Правила грамматики таковы:

$$p_1 = \text{`}Reg : r_i\text{'}$$
$$p_2 = \text{`}Mem : m_i\text{'}$$
$$p_3 = \text{`}Imm : c\text{'}$$

для каждого регистра r_i, ячейки памяти m_i и константы c;

$$p_4 = \text{`}Reg : \text{`} + \text{'}(Reg, Reg)\text{'}$$
$$p_5 = \text{`}Reg : fetch(Mem)\text{'}$$
$$p_6 = \text{`}Reg : index(Mem, Reg)\text{'}$$
$$p_7 = \text{`}Reg : Imm\text{'}$$
$$p_8 = \text{`}Mem : Reg\text{'}$$

— правила для собственно команд;

$$p_9 = \text{`}S : Reg\text{'}$$
$$p_{10} = \text{`}S : Mem\text{'}$$
$$p_{11} = \text{`}S : Imm\text{'}$$

— правила для стартового нетерминала.

Эта грамматика не находится в нормальной форме, поскольку в ней присутствуют цепные правила $p_7 - p_{11}$. После их удаления ко множеству правил добавляются правила следующего вида (для сокращения записи правила с одинаковой правой частью сгруппированы, а различиные выводимые нетерминалы показаны слева в фигурных скобках):

$$p_{12,13} = \text{`}\{Mem, S\} : r_i\text{'}$$
$$p_{14} = \text{`}S : m_i\text{'}$$
$$p_{15,16,17} = \text{`}\{S, Reg, Mem\} : c\text{'}$$
$$p_{18,19} = \text{`}\{Mem, S\} : \text{`} + \text{'}(Reg, Reg)\text{'}$$
$$p_{20,21} = \text{`}\{Mem, S\} : fetch(Mem)\text{'}$$

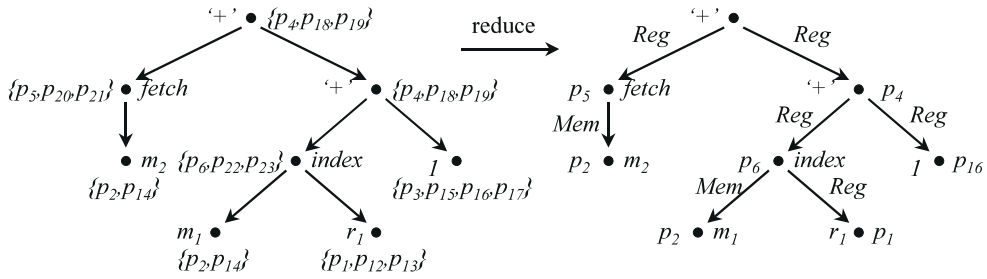

Рис. 11: Пример построения вывода

$$p_{22,23} = \text{`}\{Mem, S\} : index(Mem, Reg)\text{'}$$

Пример разметки, получаемой с помощью описанного выше алгоритма, приведен на рис. 11 слева. Справа показан результат построения вывода с помощью свертки. Каждая вершина помечена правилом, из которого в этом выводе выводится ее поддерево, а каждой дуге сопоставлен предзаданный нетерминал, определяемый правилом для вышележащей вершины.

Полученный вывод теперь необходимо преобразовать в вывод в исходной грамматике. Для этого достаточно заменить каждое вхождение правила, полученного нормализацией грамматики, на соответствующую последовательность правил исходной. В данном случае требуется заменить правило p_{19} на последовательность p_9, p_4 и правило p_{16} на последовательность p_7, p_3. Результат такой замены и полученное покрытие показаны на рис. 12. Линеаризация и распределение регистров приводит это покрытие к программе.

Заметим, что стартовый нетерминал грамматики потребовался нам только из формальных соображений, поэтому при реализации настоящих кодогенераторов без него можно обойтись; кроме того, описанная процедура приведения грамматики к нормальной форме увеличивает число правил. Что касается удаления правил, содержащих в своей правой части сложный образец, то это увеличивает размер грамматики не более чем вдвое (если считать размером сумму вершин во всех образцах всех правил) и существенно упрощает процедуру сопоставления с образцом. Удаление же цепных правил в худшем случае увеличивает количество правил квадратично, не меняя при этом числа нетерминалов. Из этого следует, что при конкретной реализации целесообразно отказаться от удаления цепных правил и модифицировать алгоритмы построения разметки, транзитивно замыкая по цепным правилам каждый вновь появляющийся в разметке нетерминал.

2.2 Деревянные автоматы

Выше для иллюстрации общности деревянных грамматик и автоматных грамматик обычных “линейных” языков нами уже упоминалось понятие деревянного автомата. Сейчас мы опишем этот объект подробнее.

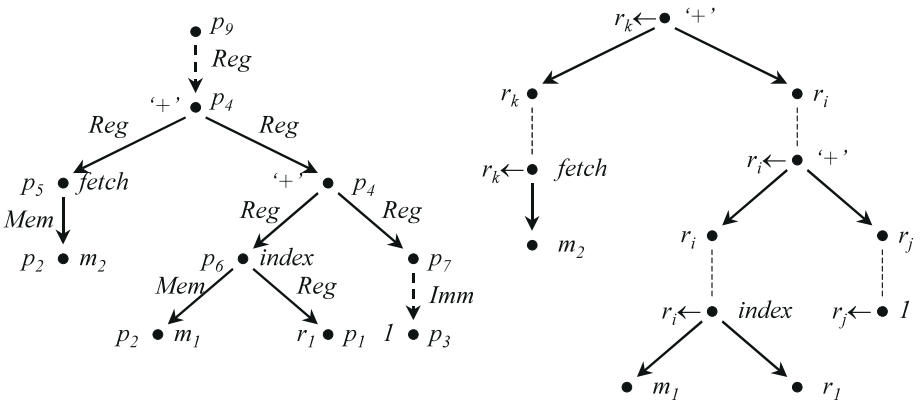

Рис. 12: Пример построения вывода (продолжение)

Использование автомата для проверки выводимости дерева в данной грамматике основывается на следующем простом наблюдении. Пусть у нас есть грамматика G. В этой грамматике для каждой терминальной пометки x, соответствующей листу выводимого дерева, существует конечное число правил, каждое из которых имеет вид '$K_i : x$', $K_i \in N$. Очевидно, что алгоритм построения вывода для каждого листа, помеченного x, построит разметку, состоящую только из нетерминалов K_i. Следовательно, можно сократить работу этого алгоритма, если ввести в грамматику "метаправило" вида '$\{K_i\} : x$', которое выводит для листа, помеченного терминалом x, сразу все множество нетерминалов $\{K_i\}$. Покажем, что данное соображение может быть применено и к вершинам, отличным от листьев.

Назовем *состоянием* произвольное множество нетерминалов. Далее множество всех состояний будем обозначать $State : Exp$ (таким образом, $State = 2^N$). Без ограничения общности будем считать, что все образцы деревянной грамматики имеют степень ноль или два. Тогда *деревянным автоматом* для грамматики $G = (N, T, R, S)$ назовем пару $A_G = (S, \tau)$, где $S \subseteq State$ — множество состояний автомата, τ — функция перехода вида

$$\tau : T \cup (T \times S \times S) \to S$$

При этом мы считаем, что для произвольных $s, s' \in S$, $x \in T$ $\tau(x)$, $\tau(x, s, s')$ соответственно определены только тогда, когда $Arity(x) = 0$, $Arity(x) = 2$.

Алгоритм работы деревянного автомата над данным деревом приведен на рисунке 13. Так же как и при обычном построении вывода, этот алгоритм осуществляет обход дерева снизу вверх; в каждой вершине выводится состояние, определяемое ее терминальной пометкой и состояниями ее сыновей. Если для каких-то значений аргументов функция `A.t` не определена, то это означает, что дерево не имеет вывода в данной грамматике. Видно,

```
    run_automaton (A, v) :
01    x = terminal(v);
02    if leaf(v) then label(v)=A.т(x)
03    else begin
04      for each s in sons(v) do run_automaton(A, s);
05      label(v)=A.т(x, label(son(v,1)), label(son(v,2)))
      end
```

Рис. 13: Алгоритм работы деревянного автомата

```
    build_leaf_states (G) :
01    states = empty;
02    for each x in T where arity(x)=0 do begin
03      state_x = empty;
04      for each r='K:x' in R do state_x += {K};
05      т(x) = state_x;
06      states += {state_x}
07    end;
08    return (states, т);
```

Рис. 14: Построение деревянного автомата: вычисление начальных состояний

что все нетерминалы в пометке вершины выводятся одновременно, минуя стадию перебора применимых правил. Данный алгоритм также линеен относительно числа вершин дерева.

Автомат A_G назовем эквивалентным для грамматики G в том и только том случае, если его работа над произвольным деревом t ассоциирует с его корнем такое состояние s, что $\forall K \in s \;\; K \to^*_G t$.

Далее мы покажем, как для произвольной деревянной грамматики построить эквивалентный ей деревянный автомат.

Очень легко построить начальную часть нужного автомата, которая выводит правильные состояния для листьев дерева (см. рис. 14): для этого необходимо объединить все нетерминалы, которые выводятся для каждого листа в данной грамматике. Результатом этого является начальное множество состояний `states` и функция перехода `т`, пока что определенная только на операндах.

Алгоритм построения остальной части автомата приведен на рисунке 15. Для каждого терминала `x`, не являющегося операндом, и для каждой пары `(left,right)` состояний из текущего множества состояний `states` (строки 03-04) этот алгоритм исследует всевозможные правила, в правую часть которых входит терминал `x` (строка 06). Правило `K:x(P,Q)` может быть применено к вершине с пометкой `x` тогда и только тогда, когда ее сыновья помечены соответственно нетерминалами `P` и `Q`, т.е., в терминах автомата,

```
    build_automaton (G) :
01    (states, т) = build_leaf_states (G);
02    do
03      for each x in T where arity(x)>0 do begin
04        for each (left, right) in (states x states) do begin
05          state_x = empty;
06          for each r='K:x(P,Q)' in R do
07            if P in left and Q in right then state_x += {K};
08          if state_x <> empty then begin
09            т(x,left,right)=state_x;
10            states += {state_x}
11          end
12        end
13      end
14    while changed(states);
15    return (states, т);
```

Рис. 15: Алгоритм построения деревянного автомата

тогда, когда `P` содержится в `left`, а `Q` — в `right` (строка `07`). Все такие нетерминалы `K` образуют новое состояние `state_x`, которое, в случае его непустоты, добавляется к множеству состояний автомата и для которого определяется соответствующее правило (строки `08-10`). Данный процесс повторяется до тех пор, пока множество состояний не перестанет расти.

Очевидно, что в силу конечности множества нетерминалов построение деревянного автомата всегда завершается. Эквивалентность полученного автомата деревянной грамматике может легко быть доказана индукцией по длине вывода.

В качестве примера рассмотрим деревянную грамматику, множества терминалов и нетерминалов которой есть соответственно

$$T = \{a, b, c\}$$
$$N = \{X, Y, Z\}$$

а множество правил —

$$p_1 = \text{`}X : a\text{'}$$
$$p_2 = \text{`}Z : a\text{'}$$
$$p_3 = \text{`}Y : b\text{'}$$
$$p_4 = \text{`}Z : c(X, Y)$$
$$p_5 = \text{`}X : c(Z, Y)\text{'}$$

Множества состояний и функция перехода автомата, построенного изложенным алгоритмом, есть

$$S = \{X, Y, XZ\},\ \tau(a) = XZ,\ \tau(b) = Y,\ \tau(c, XZ, Y) = XZ$$

Здесь для краткости опущены скобки и запятые в записи состояний.

В приведенной формулировке деревянный автомат, как нетрудно заметить, не может быть использован для построения множества выводов, поскольку его состояние кодирует множество нетерминалов, но не сохраняет правила, которыми эти нетерминалы выводятся. Для детерминированной грамматики, в которой для каждого нетерминала существует не более одного правила, это не является препятствием, однако грамматика вообще говоря может не быть детерминированной. Кроме того, число правил в автомате вообще говоря может быть экспоненциальным относительно числа правил грамматики. Тем не менее, как мы увидим ниже (см. раздел 5.2), при использовании динамического программирования для выбора оптимального вывода применение деревянных автоматов оказывается оправданным с практической точки зрения, поскольку дает возможность перенести этап динамического программирования на период компиляции компилятора. Размеры получаемых на практике автоматов после специальных оптимизаций оказываются приемлемыми для использования в реальных компиляторах.

3 Труднорешаемость задачи генерации оптимального кода для дэга

В данном разделе мы обоснуем факт труднорешаемости задачи генерации оптимального кода для дэгов для широкого класса практически значимых случаев.

Рассмотрим прежде всего ограниченный случай однорегистровой системы команд, в которой все инструкции имеют один из трех видов:

- $r \leftarrow m_i$ (загрузка);
- $m_i \leftarrow r$ (выгрузка);
- $r \leftarrow x(r, m_i)$ (операция), где $x \in Oper(F)$.

В качестве функции стоимости примем число инструкций в программе. Очевидно, что с помощью программ в данной системе команд можно вычислять только такие дэги, все вершины которых, за исключением листьев, имеют двух наследников. Это ограничение не повлияет на общезначимость результата.

Рассмотрим свойства программ в этой системе команд, вычисляющих заданные дэги. Без ограничения общности будем считать, что все листья этих дэгов помечены ячейками памяти и что все программы записывают вычисленные корни дэга также в ячейки памяти. Далее для краткости условимся первого наследника произвольной вершины считать *левым*, а второго — *правым*.

Пусть D — дэг, P — вычисляющая его программа. Очевидны следующие свойства:

1. каждой отличной от листа вершине дэга в покрытии соответствует одна инструкция, отличная от загрузки и выгрузки;
2. если у отличной от листа вершины существует непосредственный предшественник, для которого данная вершина — правый наследник, то такой вершине дополнительно соответствует выгрузка;
3. если отличная от листа вершина является сложной, то ей дополнительно соответствует выгрузка;
4. отличному от листа корню дэга дополнительно соответствует выгрузка;
5. если у листа существует непосредственный предшественник, для которого он — левый наследник, то такому листу соответствует загрузка.

Данные свойства позволяют оценить снизу размер программы, вычисляющей данный дэг. Оптимальной для данного дэга будет программа, содержащая минимальное число дополнительных инструкций по отношению к этой нижней границе. Для изучения ситуаций, в которых эти дополнительные инструкции могут появиться, рассмотрим дальнейшие свойства программ.

Утверждение 3.1. *Пусть P — оптимальная программа, вычисляющая дэг D. Тогда существует оптимальная программа P', такая, что для произвольной ячейки памяти m_i P содержит не более одной инструкции выгрузки S, такой, что $res(S) = m_i$.*

Доказательство. Если для P данное утверждение не выполнено, то в ней найдется хотя бы две инструкции выгрузки S_1 и S_2, таких, что $res(S_1) = res(S_2)$. Возьмем свежую ячейку памяти m, которая не используется в качестве аргумента или результата никакой инструкции P и согласованно переименуем $res(S_2)$ в m. Продолжая в том же духе, получим оптимальную программу P', для которой утверждение выполнено. □

Таким образом, мы можем ограничиться рассмотрением только таких оптимальных программ, которые каждую ячейку памяти записывает не более одного раза. Кроме того, очевидно можно рассматривать только такие программы, которые ничего не записывают в ячейки памяти, которыми помечены листья дэга.

Утверждение 3.2. *Пусть P — оптимальная программа. Тогда в ней не существует пары последовательных выгрузок.*

Доказательство. Пусть S_1S_2 — пара последовательных выгрузок в программе P. Так как $res(S_1) \neq res(S_2)$, согласованно переименуем $res(S_2)$ в

$res(S_1)$. Очевидно, что значение программы при этом не изменится. В любом случае теперь можно удалить инструкцию S_2 без изменения значения программы, что противоречит ее оптимальности. □

Утверждение 3.3. *Пусть оптимальная программа P имеет вид P_1LP_2, где $|P_1| > 0$, L — инструкция загрузки. Тогда последняя инструкция P_1 является инструкцией выгрузки.*

Доказательство. Обозначим через J последнюю инструкцию P_1. Если она является инструкцией выгрузки, то утверждение верно. В противном случае она является либо операцией, либо загрузкой. В первом случае она бесполезна (поскольку сразу же за ней исполняется инструкция L), во втором возможно два варианта: либо $Arg(J) = Arg(L)$, либо $Arg(J) \neq Arg(L)$. В любом случае удаление J из P приводит к эквивалентной и более короткой программе, что противоречит оптимальности P. □

Утверждение 3.4. *Пусть $P = P_1LIP_2$ — оптимальная программа, L — инструкция загрузки. Тогда I — операция.*

Доказательство. Если I — загрузка, то L — бесполезная инструкция. Если I — выгрузка, то согласованно переименуем $res(I)$ в x, где $\{x\} = Arg(L)$. В любом случае либо L, либо I можно удалить без изменения значения программы, что противоречит ее оптимальности. □

Утверждение 3.5. *Пусть D — дэг, α, β — его вершины, такие, что α — левый наследник вершины β и α не является сложной. Тогда произвольная оптимальная программа, вычисляющая D, имеет вид P_1IJP_2, где инструкция I соответствует α, инструкция J — β.*

Доказательство. Рассмотрим программу, вычисляющую D. Поскольку α — наследник β, то инструкция, соответствующая α (обозначим ее I) находится в P "левее" инструкции, соответствующей β (обозначим ее J). Таким образом, P имеет вид P_1IQJP_2. Если $|Q| = 0$, то утверждение верно. Рассмотрим ситуацию, при которой $|Q| > 0$.

Поскольку α — левый наследник β (и, следовательно, вычисленное в ней значение передается через единственный регистр), то I является операцией или загрузкой. Если I — загрузка, то по предыдущему утверждению следующая инструкция должна быть операцией. Но такой операцией может быть только J, поскольку α — непосредственный наследник β. Это противоречит предположению о том, что $|Q| > 0$. Следовательно, I — операция, а первая инструкция Q — выгрузка (загрузкой она быть не может, поскольку в этом случае I — бесполезна). Таким образом, $P = P_1ISQ'LJP_2$, где S — выгрузка, L — загрузка, $Arg(L) = \{res(S)\}$.

Если $|Q'| = 0$, то удаление инструкций S и L не меняет значения программы (так как из того, что α не является сложной следует, что $Use_P(S) = \{L\}$), что противоречит оптимальности. Следовательно, $|Q'| > 0$.

Поскольку I — операция, $|P_1| > 0$. Рассмотрим ближайшую к I инструкцию P_1, являющуюся выгрузкой или загрузкой и обозначим ее через Y. Таким образом, $P_1 = P_1' Y I_1 I_2 \dots I_k$, где все I_l — операции. Если Y — выгрузка, то рассмотрим программу $P_1' Y Q' L' I_1 I_2 \dots I_k I J P_2$, где инструкция L' имеет вид '$r \leftarrow res(Y)$'. Очевидно, что значение этой программы совпадает со значением P, а длина меньше на единицу, что противоречит оптимальности P. Если Y — загрузка, то рассмотрим программу $P_1' Q' Y I_1 I_2 \dots I_k I J P_2$. Значение этой программы также очевидно совпадает со значением P и она так же короче P. □

Последнее свойство существенно проясняет состояние дел: оказывается, в оптимальной программе вычисление вершины непосредственно следует за вычислением ее левого наследника, если только он не является сложной вершиной. Если же он сложен, то всегда после его вычисления следует инструкция выгрузки. Ячейка памяти, являющаяся результатом этой выгрузки, может быть непосредственно использована как аргумент в тех командах, которые соответствуют вершинам, для которых этот наследник является правым. Вычисления тех вершин, для которых он является левым, могут потребовать загрузки. Очевидно, что если данная вершина является левым наследником для n других вершин, то как минимум требуется $n-1$ дополнительная загрузка. Следовательно, единственным оставшимся вариантом оказывается ситуация, при которой некоторой сложной вершине, являющейся левым наследником для n других вершин, в оптимальной программе соответствует n загрузок. Если такая ситуация действительно имеет место, то эту n-ю загрузку будем называть *дополнительной инструкцией*. Таким образом, построение оптимальной программы для данной системы команд сводится к минимизации числа дополнительных инструкций. Покажем, что эта задача является труднорешаемой.

Как окажется ниже, для наших целей достаточно рассмотрения только таких дэгов, в которых, во-первых, каждая вершина является левым наследником не более чем одной вершины, и во-вторых, сложные вершины могут иметь среди своих непосредственных наследников только листья. Для таких дэгов *левой цепью* назовем максимальную последовательность вершин $\alpha_1, \alpha_2, \dots, \alpha_k$ таких, что α_i является левым наследником α_{i+1}. Очевидно, что любая левая цепь начинается в листе и заканчивается в той вершине, которая не является левым наследником ни для какого своего предшественника. Кроме того, каждой дополнительной инструкции можно поставить в соответствие единственную левую цепь.

Будем говорить, что левая цепь l_1 *непосредственно зависит* от отличной от нее левой цепи l_2 (обозначение $l_2 \rightsquigarrow l_1$) тогда и только тогда, когда найдутся вершины $\alpha \in l_1$ и $\beta \in l_2$, такие, что β — правый наследник α.

Построим ориентированный *граф зависимостей*, в котором каждой вершине соответствует левая цепь, а для произвольной дуги e выполнено $beg(e) \rightsquigarrow end(e)$. Очевидно, что если граф зависимостей является дэгом, то оптимальная программа не содержит дополнительных инструкций. Действительно, для построения оптимальной программы достаточно вычис-

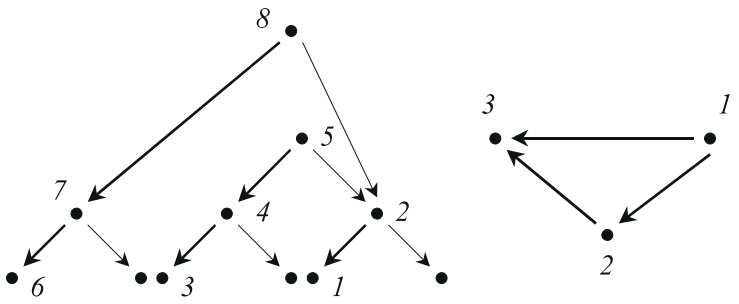

Рис. 16: Вычисление дэга без дополнительных инструкций

лить все левые цепи в порядке произвольной топологической сортировки графа зависимостей. При этом вершины каждой левой цепи будут вычисляться последовательно, поскольку все их правые наследники уже вычислены.

На рис. 16 приведен пример дэга (слева) и его графа зависимостей (справа), который также является дэгом. Левые цепи отмечены жирным начертанием, вершины дэга зависимостей пронумерованы в порядке топологической сортировки. Эта топологическая сортировка задает порядок вычисления левых цепей исходного дэга, что в свою очередь полностью определяет порядок вычисления всех его вершин. Нумерация вершин исходного дэга отражает этот порядок. Видно, что поскольку все вершины каждой левой цепи вычисляется снизу-вверх непосредственно друг за другом, оптимальная программа не содержит дополнительных инструкций.

Пусть G — произвольный ориентированный граф, не содержащий петель, $X = \{\alpha_1, \alpha_2, \ldots, \alpha_k\}$ — некоторое множество его вершин. Будем говорить, что X *разрезает все контуры* G тогда и только тогда, когда его удаление переводит G в дэг.

Утверждение 3.6. *Пусть P — оптимальная программа, вычисляющая дэг D и пусть $I_1, I_2, \ldots, I_k$ — все ее дополнительные инструкции. Тогда соответствующие этим инструкциям вершины графа зависимостей разрезают все его контуры.*

Доказательство. Возьмем произвольную оптимальную программу P, вычисляющую D, произвольный контур графа зависимостей и предположим, что в P подпрограмма вычисления всех левых цепей этого контура не содержит дополнительных инструкций. Это означает, что каждая левая цепь вычисляется непрерывно снизу-вверх. Поскольку вершины программы P вычисляются в порядке линеаризации D, получаем существование топологической сортировки замкнутого контура в графе зависимостей, что является противоречием. □

Таким образом, для порождения оптимальной программы, вычисляющей дэг, достаточно построить для этого дэга граф зависимостей и найти

в нем наименьшее по мощности множество вершин, разрезающее все контуры. Поскольку задача нахождения такого множества является хорошо известной NP-трудной задачей [1], возникает вопрос: возможен ли более простой способ генерации оптимального кода? Ответ на этот вопрос дает следующая теорема [3]:

Теорема 3.1. *(Ахо, Джонсон, Ульман, 1976) Пусть G — произвольный ориентированный граф, не содержащий петель. Тогда существует дэг D, такой, что G является его графом зависимостей.*

Доказательство. Покажем, как по данному ориентированному графу построить искомый дэг.

Изначально считаем дэг пустым.

Для каждой вершины $\alpha \in G.V$ построим в дэге левую цепь $\alpha_0, \alpha_1, \ldots, \alpha_k$, где $k = |out_G(\alpha)|$ — количество дуг, выходящих из α в G. К вершине α_0 дополнительно добавим в дэге два листа.

Пусть (α, β) — i-я дуга, выходящая из вершины α. Тогда добавим в дэг дугу (α_i, β_0).

G является графом зависимостей для построенного дэга D по определению. □

Таким образом, задача порождения оптимального кода для дэгов оказалась не труднее и не проще, чем известная NP-трудная задача, что свидетельствует о ее NP-трудности.

На рис. 17 приведен пример ориентированного графа (A) и построенного для него в соответствии с вышеприведенной теоремой дэга (B). Минимальными множествами вершин, разрезающими все контуры, являются множества $\{\alpha\}$, $\{\beta\}$ или $\{\gamma\}$. Эти множества определяют левые цепи дэга, которым в оптимальной программе могут соответствовать дополнительные инструкции. По вышеперечисленным соображениям эти инструкции могут появиться только непосредственно после сложных вершин — соответствующие места на рисунке обозначены стрелками. Выбор минимального множества, разрезающего контуры, однозначно определяет местоположение дополнительных инструкций и, следовательно, порядок вычисления дэга в оптимальной программе. Порядок вычисления дэга при выборе множества $\{\gamma\}$ показан на том же рисунке (C).

Факт труднорешаемости задачи генерации оптимального кода для дэга пока что установлен для системы команд весьма специального вида. Очевидно, однако, что класс систем команд, обладающих тем же свойством, может быть легко расширен.

Прежде всего, заметим, что ограничение на вид правой части операций несущественно, поскольку он влияет только на структуру покрытия, которая практически не использовалась. Учитывая, что фрагменты исходного дэга, ограниченные корнями, листьями и сложными вершинами являются деревьями, можно было бы везде рассматривать оптимальное покрытие деревьев, что сохранило бы все результаты. Требование того, чтобы регистровый операнд каждой операции был левым, также является несущественным

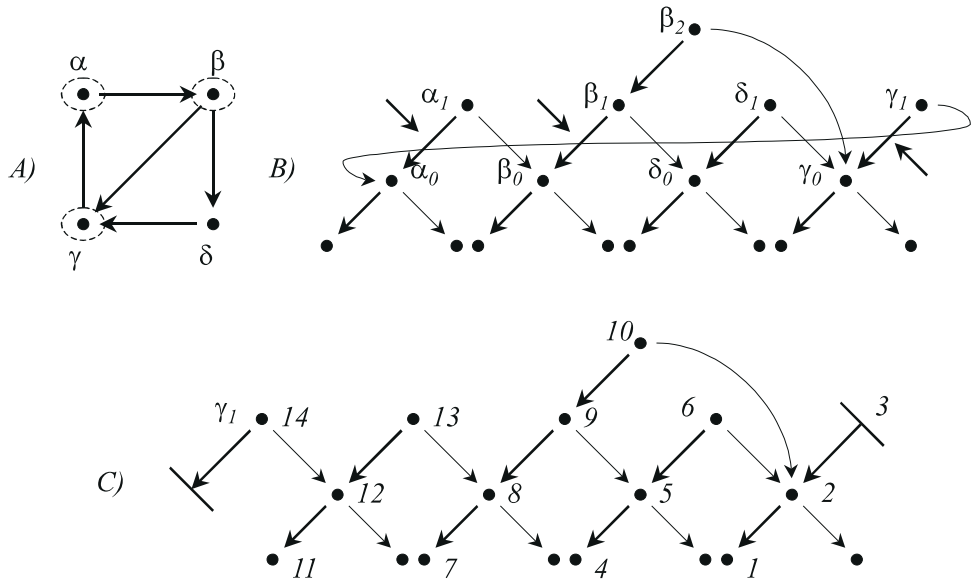

Рис. 17: Связь между множеством вершин, разрезающих контуры, и оптимальной программой

— можно рассматривать ситуацию, при которой он может занимать среди листьев тела операции произвольную позицию, а при доказательстве вместо понятия “левая цепь” использовать понятие “регистровая цепь”.

Более содержательным является вопрос о влиянии числа регистров на труднорешаемость рассматриваемой задачи. Оказывается, что наличие произвольного (но конечного) числа регистров не упрощает ее. Можно доказать, что в этой ситуации наличие простого контура в графе зависимостей ведет к необходимости использовать дополнительный регистр и, следовательно, количество дополнительных команд в оптимальной программе оказывается равно $|X|/K$, где X — минимальное разрезающее контуры графа зависимостей множество, K — число регистров. Обсуждение вопросов, связянных с труднорешаемостью генерации оптимального кода для дэгов и изложение некоторых эвристических алгоритмов содержится в [3].

Очевидно, что в качестве функции стоимости можно рассматривать не только число инструкций, но и вообще произвольную функцию вида

$$c(P) = \sum_{I \in P} c(I)$$

где $c(I) > 0$ — вес инструкции I.

Наконец, легко заметить, что построение минимальной по длине программы является тривиальной задачей для полной ортогональной системы команд, посколько в ней такую программу можно построить, обходя дэг

в порядке произвольной линеаризации и рассматривая только минимальные покрытия. В то же время для других функций стоимости это свойство перестает выполняться.

4 Генерация оптимального кода для деревьев

В силу того, что задача генерации оптимального кода для дэгов оказалась труднорешаемой даже в простейших случаях, целесообразно исследовать вопрос о существовании применимого алгоритма генерации кода для деревьев.

Как будет показано ниже, для широкого класса архитектур задача генерации оптимального кода для дерева оказывается решаемой за время, линейное относительно его размера. Кроме того, как мы увидим в дальнейшем, идея излагаемого далее алгоритма допускает обобщение, приводящее к наиболее используемой на сегодняшний день схеме построения настраиваемых генераторов кода.

Для описания алгоритма и его обоснования прежде всего требуется изучить некоторые свойства программ, вычисляющих деревья. В качестве системы команд в данном разделе будет использована система с равноценными регистрами. Кроме того, введенное ранее понятие ширины программы мы будем трактовать как регистровую ширину, то есть ширину на множестве инструкций, вырабатывающих регистровый результат.

4.1 Свойства программ и нормальные формы

Для данной программы $P = I_1 I_2 \dots I_q$ и произвольной инструкции I_t при $1 \leq t < q$ определим понятие *использования* I_t относительно P следующим образом:

$$U_P(I_t) = max\{m \mid I_m \in Use_P(I_t)\}$$

Иными словами, $U_P(I_t)$ — это номер самой "далекой" инструкции, которая использует результат I_t.

Областью данной инструкции I_t в программе P будем называть последовательность инструкций

$$Scope_P(I_t) = I_t I_{t+1} \dots I_{U_p(I_t)}$$

Далее мы будем рассматривать только такие программы P, которые удовлетворяют следующим двум ограничениям:

1. для произвольной инструкции I_t существует единственная инструкция $I_k \in Scope_P(I_t)$, такая, что $res(I_t) \in Arg(I_k)$;

2. для произвольной ячейки m_i памяти в программе P существует не более одной инструкции I, такой, что $m_i \in Arg(I)$.

Легко можно показать, что значениями программ, удовлетворяющих сформулированным ограничениям, будут являться деревья.

Утверждение 4.1. *Пусть P — программа, $width(P) = w$ и R — подмножество из w различных регистров. Тогда существует эквивалентная программа P', содержащая такое же количество инструкций и использующая только регистры из R.*

Доказательство. Будем получать программу P' потем согласованного переименования регистров. Заметим, что такое согласованное переименование задается переименованием результатов инструкций, имеющих либо вид '$r_i \leftarrow m_i$', либо вид '$r_i \leftarrow t$', где t не содержит листьев-регистров (это следует из того, что перед использованием регистра он должен быть предварительно вычислен).

Двигаясь от начала программы, будем последовательно назначать очередной регистр из R для каждой инструкции I_k, имеющей описанный вид. Это можно сделать только в том случае, если в R существует хоть один свободный регистр. Предположим, что такого регистра вдруг не оказалось. Это означает, что ширина I_k в программе P была равна $w + 1$. Действительно, поскольку в R нет свободных регистров, то w регистров были заняты непосредственно перед I_k (это были те регистры, которые переименовали в регистры из R). Сама же I_k не использует ни одного регистра и добавляет к своей ширине тот регистр, который вычисляет. Таким образом, ширина I_k оказалась больше ширины программы P, что является противоречием. □

Данное утверждение позволяет трактовать программы ширины w как программы, использующие w регистров.

Утверждение 4.2. *Пусть есть программа $P = I_1 I_2 \dots I_q$ ширины w, не содержащая инструкций выгрузки. Пусть инструкция I_q использует k регистров, чьи значения в точке $q - 1$ равны соответственно $A_1, A_2, \dots, A_k$. Тогда существует эквивалентная программа $Q = J_1 J_2 \dots J_q$ и перестановка π, такие, что*

1. *$width(Q) \le w$;*
2. *Q имеет вид $P_1 P_2 \dots P_k J_q$, где $val(P_i) = A_{\pi(i)}$, $1 \le i \le k$, и $width(P_i) \le w - i + 1$.*

Доказательство. По предыдущему утверждению можно считать, что P использует только регистры $R_1, \dots, R_w$. Возьмем инструкцию I_1 и рассмотрим ее значение. Поскольку мы рассматриваем только такие программы, которые не содержат бесполезных инструкций, значение $res(I_1)$ входит как поддерево в некоторое A_j. Полагаем $\pi(1) = j$ и конструируем P_1 путем выделения всех тех инструкций P, которые вычисляют поддеревья A_j (в том же порядке). Легко видеть, что $val(P_1) = A_j$ и $width(P_1) \le w$. Переименовывая регистры, используемые в P_1, добъемся того, что $val(P_1)$ вычисляется в регистре R_w.

Рассмотрим оставшиеся инструкции в P. Они вычисляют остальные A_i и затем завершаются I_q. Легко видеть, что ширина оставшейся программы не превосходит $w-1$, поскольку как минимум один регистр необходим для хранения значения поддерева A_j (здесь существенно используется то, что P не содержит инструкций выгрузки). Таким образом, регистры остаточной программы можно переименовать, после чего она будет использовать только регистры $R_1, \ldots, R_{w-1}$ и вычислять результат в R_{w-1}.

Возьмем теперь первую инструкцию P, не вошедшую в P_1 и повторим те же действия. Получим значение $\pi(2)$ и программу P_2, такую, что $val(P_2) = A_{\pi(2)}$. Кроме того, $width(P_2) \leq w-1$, так как она является подпрограммой программы, оставшейся после выделения P_1. Переименуем регистры, использующиеся в P_2 с тем, чтобы она использовала только $R_1, \ldots, R_{w-1}$ и вычисляла свое значение в R_{w-1}. Продолжая в том же духе, получим перестановку π и последовательность программ P_i, вычисляющих поддеревья $A_{\pi(i)}$. Переименование регистров, использующихся I_q, завершает доказательство. □

Программу без инструкций выгрузки назовем *строго непрерывной*, если она удовлетворяет пункту 2 предыдущего утверждения и все P_i в свою очередь строго непрерывны.

Пусть есть программа $P = I_1 I_2 ... I_q$. Будем говорить, что P находится в *нормальной форме*, если она имеет следующий вид:

$$P = P_1 J_1 P_2 J_2 \ldots P_{s-1} J_{s-1} P_s$$

где

1. все J_k — инструкции выгрузки и ни одна из программ P_k не содержит инструкций выгрузки;
2. ни один регистр не активен непосредственно после каждой J_k.

Утверждение 4.3. *Для произвольной программы существует такая перестановка ее инструкций, которая переводит ее в эквивалентную программу в нормальной форме.*

Доказательство. Если программа P не содержит инструкций выгрузки, то она уже находится в нормальной форме. В противном случае возьмем первую инструкцию выгрузки I_f в программе P. Можно определить, какие из инструкций $I_1 \ldots I_f$ не вычисляют никакую часть значения I_f и переместить их непосредственно после I_f, получив таким образом программу $P_1 I_f Q_1$. Очевидно, что ни один регистр не активен после I_f — в противном случае та инструкция, которая его вырабатывает, должна была быть переставлена после I_f (сама I_f не может его использовать, т.к. вся программа вычисляет дерево выражения). Кроме того, $val(Q_1) = val(P)$.

Действительно, пусть I_s — инструкция P, которая использовала результат I_f. В этом случае она, очевидно, находилась после I_f. Поскольку I_f

— первая инструкция выгрузки, после перестановки ее результат не будет "затерт" никакой другой инструкцией.

Применяя такое же преобразование к Q_1, получаем нормальную форму программы P. □

Определим, наконец, следующее понятие. Пусть программа $P = P_1 J_1 P_2 J_2 \dots P_{s-1} J_{s-1} P_s$ находится в нормальной форме. Будем говорить, что она находится в *строгой нормальной форме*, если все P_i строго непрерывны.

Очевидно, произвольная программа может быть преобразована к программе в строгой нормальной форме путем перестановок без увеличения ширины. В частности, таким образом может быть преобразована произвольная оптимальная программа.

Обозначим через $opt(t)$ длину оптимальной программы, вычисляющей дерево t. Имеет место следующий критерий оптимальности алгоритма генерации кода [2]:

Теорема 4.1. *(Ахо, Джонсон, 1979). gen — оптимальный алгоритм генерации кода тогда и только тогда, когда*

1. $|gen(t)| = opt(t)$ *для любого* $t \in \mathfrak{T}(F)$, *для которого существует оптимальная программа без выгрузок;*
2. $|gen(t)| \le |gen(s)| + |gen(t/s)| + 1$ *для любого t и любого его поддерева s, где через t/s обозначено дерево, полученное из t стягиванием поддерева s в лист.*

Доказательство. Достаточность. Пусть t — дерево, оптимальная программа вычисления которого содержит k инструкций выгрузки, gen — алгоритм кодогенерации. Индукцией по k докажем, что gen — оптимален.

База: $k = 0$. В этом случае по условию 1 получаем, что $|gen(t)| = opt(t)$.

Индукционный переход. Предположим, что 2 выполняется для произвольного дерева, оптимальная программа вычисления которого содержит не более $k-1$ инструкций выгрузки. Пусть t — дерево, для которого оптимальная программа P содержит k инструкций выгрузки. Будем считать, что P находится в строгой нормальной форме. Тогда $P = P_1 I P_2$, где I — первая инструкция выгрузки P. При этом P_1 вычисляет некоторое поддерево s дерева t, I сохраняет это значение в ячейке памяти, а P_2 вычисляет дерево t/s. Поскольку P — оптимальная программа, оптимальными же являются P_1 и P_2. Следовательно, $opt(t) = opt(s) + 1 + opt(t/s)$.

Рассмотрим теперь программы, порождаемые алгоритмом gen. Поскольку P_1 — программа без выгрузок, а P_2 — программа, к которой применимо индукционное предположение, получаем:

$$|gen(t)| \le |gen(s)| + |gen(t/s)| + 1 = opt(s) + opt(t/s) + 1 = opt(t)$$

Таким образом, $gen(t)$ — оптимальна.

Необходимость. Предположим, что gen — оптимальный алгоритм. Тогда пункт 1 выполняется сразу. Возьмем произвольную программу P_1, вычисляющую s, и произвольную программу P_2, вычисляющую t/s. Тогда $P_1 I P_2$ — программа, вычисляющая t, где I — подходящая инструкция выгрузки. В качестве P_1 теперь возьмем $gen(s)$, а в качестве P_2 — $gen(t/s)$. Очевидно, что $|gen(s)Igen(t/s)| = |gen(s)| + |gen(t/s)| + 1$. Поскольку gen — оптимальный алгоритм порождения кода,

$$opt(t) = |gen(t)| \leq |gen(s)| + |gen(t/s)| + 1$$

что и завершает доказательство. □

Мы видим, что в данной обстановке в качестве оптимальных программ можно рассматривать программы, устроенные следующим образом: генерируемое дерево разбивается на фрагменты, каждый из которых вычисляется программой без выгрузок. Каждая такая программа вычисляет фрагмент в порядке, при котором каждый узел исполняется непосредственно после его наследников. Следовательно, для генерации оптимального кода требуется определить места выгрузок (т.е. разбиение исходного дерева на фрагменты) и для каждого фрагмента найти покрытие минимального размера. Что касается линеаризации, то она поностью определяется порядком вычисления поддеревьев для каждой вершины. Эти соображения позволяют решить рассматриваемую задачу с помощью алгоритма динамического программирования.

4.2 Алгоритм Ахо-Джонсона

С каждой вершиной дерева v свяжем вектор стоимостей (v) длины $N + 1$. i-й элемент этого вектора будет символизировать минимальную стоимость программы в строгой нормальной форме $P_1 J_1 P_2 J_2 \ldots P_k J_k Q$, вычисляющей дерево с корнем v, где $width(Q) \leq i$. Нулевой элемент будет обозначать стоимость вычисления программы в ячейку памяти. Перед началом работы алгоритма все элементы векторов стоимости для каждой вершины принимаются равными бесконечности. Процедура инициализации показана на рис. 18. Здесь `c(v)[j]` — j-й элемент вектора стоимостей для вершины v.

Следующая стадия алгоритма — нахождение значений всех элементов векторов стоимости для всех вершин дерева. Поскольку помимо самих стоимостей нам позже потребуется сама программа, вместе с ними для каждой вершины дерева v мы будем запоминать последнюю инструкцию программы, вычисляющей поддерево с корнем v и порядок, в котором в этой программе вычисляются ее регистровые операнды.

Алгоритм построения описанной выше разметки приведен на рис. 19. Дерево обходится снизу вверх (строка `01`) и в каждой вершине v вычисляется элемент вектора стоимости на основе векторов стоимостей для дочерних вершин. Эта процедура состоит в следующем:

```
    init (t)
    {
01    for all v in t.V do
02      for i=0 to N do
03        c(v)[i] = infinity;
    }
```

Рис. 18: Алгоритм Ахо-Джонсона, инициализация

1. если v — лист, то стоимость программы, вычисляющей ее в ячейку памяти, есть 0 (строка 02);

2. в системе команд IS выбираются все инструкции, которые покрывают v (строка 03) (один из способов нахождения таких инструкций описан в части 2);

3. для каждой такой инструкции I внутри поддерева с корнем v определяются два множества вершин: тех, значения которых должны быть вычислены в регистрах и тех, значения которых должны быть вычислены в ячейках памяти (строки 04-05);

4. для произвольной перестановки p множества $\{1, 2, \ldots, k\}$ и для произвольного $j \in [0, N]$ вычисляется минимальная стоимость программы, вычисляющей дерево с корнем v программой в строгой нормальной форме, последняя инструкция которой есть I (строка 08). Стоимость каждой такой программы складывается из стоимостей программ, вычисляющих нерегистровые операнды I в памяти (они хранятся в нулевых элементах соответствующих векторов стоимостей) и стоимостей программ, вычисляющих ее регистровые операнды в заданном порядке и при заданной ширине;

5. Если вычисленная таким образом стоимость является минимальной, то вместе с ней запоминается инструкция и перестановка, на которых этот минимум был достигнут (строки 09-13).

6. После вычисления элементов вектора стоимостей описанным выше способом необходимо учесть еще два случая: когда поддерево с корнем v вычисляется, используя все регистры, и затем сохраняется в ячейку памяти (строки 16-20), и когда оно сначала вычисляется в ячейке памяти, а затем загружается в регистр (строки 16-26).

После вычисления стоимостей необходимо определить, какие вершины дерева в оптимальной программе должны быть вычислены в ячейках памяти. Для этогов обще говоря надо знать ширину последней части программы в строгой нормальной форме, которая вычисляет все дерево. На рис. 20 приведен алгоритм определения выгрузок для заданной ширины. Здесь j

```
    label(v) :
01    for all w in succ(v) do label(w);
02    if leaf (v) then c(v)[0] = 0;
03    for all I in IS where covers (I, v) do
04       {S(1), S(2), ..., S(k)} = registers(I, v);
05       {T(1), T(2), ..., T(m)} = memories(I, v);
06       for each p in permutation(k) do
07         for j=0 to N do
08           cost =
                sum (i=1..m) c(T(i))[0] +
                sum (i=1..k) c(S(pi(i)))[j-i+1] + 1;
09           if cost < c(v)[j] then begin
10             instruction(v)[j] = I;
11             permutation(v)[j] = p;
12             c(v)[j] = cost;
13           end
14         done
15    done;
16    if c(v)[0] > c(v)[N]+1 then begin
17      c(v)[0] = c(v)[N] + 1;
18      permutation(v)[0] = id;
19      instruction(v)[j] = 'm <- rN';
20    end;
21    for j=1 to N do
22      if c(v)[j] > c(v)[0]+1 then begin
23        c(v)[j] = c(v)[0] + 1;
24        permutation(v)[0] = id;
25        instruction(v)[j] = 'rj <- m';
26      end;
```

Рис. 19: Алгоритм Ахо-Джонсона, вычисление стоимостей

— предзаданная ширина программы, `out l` — параметр-результат, хранящий список всех выгрузок в порядке, здаваемом какой-либо линеаризацией дерева (точный вид этой линеаризации зависит от порядка обхода). Данный алгоритм проходит дерево снизу вверх и помещает в список выгрузок те вершины, оптимальные инструкции которых есть выгрузки. Дочерние вершины обходятся в порядке, задаваемом оптимальной перестановкой.

После определения выгрузок можно сгенерировать код для подпрограмм без выгрузок. Для этого также необходимо знать ширину этих подпрограмм. Алгоритм порождения кода приведен на рис. 21. Генерация кода происходит при обходе снизу-вверх, при этом порядок обхода определяется оптимальной перестановкой. Примитив `alloc()` доставляет один свободный регистр, через `free(r)` обозначено освобождение регистра `r`. Порождение

```
     spill(out l, v, j) :
01      I = instruction(v)[j];
02      p = permutation(v)[j];
03      {S(1), S(2), ..., S(k)} = registers(I, v);
04      {T(1), T(2), ..., T(m)} = memories(I, v);
05      for i=1 to k do spill (l, S(p(i)), j-i+1);
06      for i=1 to m do spill (l, T(i), 0);
07      if j==0 && I=='m<-r' then l = l | v;
```

Рис. 20: Алгоритм Ахо-Джонсона, определение выгрузок

```
     code (v, j) :
01      I = instruction(v)[j];
02      p = permutation(v)[j];
03      {S(1), S(2), ..., S(k)} = registers(I, v);
04      {T(1), T(2), ..., T(m)} = memories(I, v);
05      for i=1 to k do r(i) = code (S(p(i)), j-i+1);
06      if k==0
07        then r = alloc();
08        else r = any_of (r(1), r(2), ..., r(k));
09      emit(
         'r<-body(I)[r(i),r(2),...,r(k); T(1),T(2),...,T(m)]'
        );
10      for each r(i) where r(i) <> r do free(r(i));
11      return r;
```

Рис. 21: Алгоритм Ахо-Джонсона, генерация кода

конкретных инструкций записано в строке 09: здесь в правую часть оптимальной инструкции подставляются регистры и ячейки памяти, содержащие вычисленные на предыдущих шагах ее операнды.

Наконец, общий вид алгоритма Ахо-Джонсона прведен на рис. 22. После инициализации и вычисления стоимостей (строки 01-02) следует выбор той программы, которая доставляет минимальную стоимость — для этого в векторе стоимостей надо найти минимальный элемент (строки 03-09). Индекс этого элемента определит ширину последнего фрагмента программы в строгой нормальной форме, вычисляющей данное дерево, и, следовательно, оптимальную инструкцию и оптимальную перестановку для его корня.

После определения ширины необходимо построить список выгрузок (строки 10-11). Обход этого списка с генерацией финальной программы показан в строках 12-19. Здесь `new()` — примитив, доставляющий свежую ячейку памяти, через `replace(x,m)` обозначена замена вершины x исходного дерева на лист, помеченный ячейкой памяти m.

Вполне очевидно, что элемент вектора стоимости $c(v)[i]$ действительно

```
    generate(t) :
01     init(t);
02     label(root(t));
03     j = 0;
04     c = intinity;
05     for i=0 to N do
06       if c(root(t))[i] < c then begin
07         j = i;
08         c = c(root(t))[i];
09       end;
10     l = emptylist;
11     spill(l, root(t), j);
12     for each x in l where x <> root(t) do
13       r = code(x, N);
14       m = new();
15       emit('m<-r');
16       free(r);
17       replace(x, m);
18     done;
19     code(root(t), N);
```

Рис. 22: Алгоритм Ахо-Джонсона, общая структура

вычисляется алгоритмом Ахо-Джонсона как длина минимальной программы в строгой нормальной форме с шириной последнего фрагмента, равной i, значение которой есть поддерево с корнем v. Так же очевидно, что длина порожденной алгоритмом программы есть минимум по всем стоимостям для корня дерева. Наконец, можно показать, что алгоритм Ахо-Джонсона корректен и удовлетворяет критерию оптимальности алгоритма генерации кода для дерева [2].

Что касается сложности данного алгоритма, то он очевидно линеен относительно числа вершин дерева и экспоненциален относительно максимального числа операндов инструкций, которое обычно ограничено небольшим числом.

Основным достоинством приведенного алгоритма является то, что он позволяет сгенерировать конечный машинный код, включая распределение регистров и построение линеаризации. Система команд, для которой данный алгоритм генерирует оптимальную программу, является подмножеством системы команд RISC-процессоров, что свидетельствует о возможности практического применения. Некоторые свойства системы команд (например, то, что результат каждой операции размещен в регистре) не являются существенными.

Действительно принципиальными свойствами системы команд является взаимозаменяемость всех регистров и одинаковый размер всех значений.

Как показано в работе [4], уже наличие в системе команд регистровых пар делает неверным утверждение о нормализации. В оптимальной программе для такой системы возможны "колебания" между вычислениями поддеревьев данной вершины, что существенно усложняет задачу поиска оптимальной линеаризации. Как показано там же, для систем, допускающих спаривание произвольных регистров количество колебаний в оптимальной программе не превосходит двух. В общем же случае оценка числа колебаний неизвестна.

Алгоритм Ахо-Джонсона явился по сути результатом, завершающим серию исследований в области генерации кода для конкретных архитектур. Более подробно с результатами этих исследований можно ознакомиться в [7, 8, 30, 2, 3, 4].

5 Восходящее переписывание деревьев

Приведенный в предыдущей части алгоритм генерации кода для деревьев, несмотря на все его очевидные достоинства, обладает большим недостатком: он работает только для фиксированной системы команд с равнозначными регистрами, причем основные свойства этой системы команд с точки зрения алгоритма существенны. Здесь мы рассмотрим обобщение алгоритма Ахо-Джонсона для произвольной системы команд, которое позволяет получать оптимальные в некотором смысле программы для деревьев. Более точно — этот подход гарантирует оптимальный выбор инструкций, который можно превратить в оптимальную программу, если для него существует распределение регистров. Речь идет о системах *восходящего переписывания деревьев*, или *BURS* (*Bottom-Up Rewrite Systems*). Собственно сокращение "BURS" первоначально было введено для обозначения систем переписывания термов специального вида [26], частными случаями которых являются деревянные грамматики; позже этот термин стал ассоциироваться именно с грамматиками.

BURS можно рассматривать как вариант деревянных грамматик, позволяющий из всего разнообразия выводов выбрать только один. Для решения этой задачи используется алгоритм динамического программирования, который близок алгоритму Ахо-Джонсона.

В основе BURS лежит деревянная грамматика специального вида. Формально, грамматика BURS получается из обыкновенной деревянной грамматики добавлением к каждому правилу численного атрибута — стоимости. Далее для правила r его стоимость будем обозначать через $c(r)$. Поскольку вывод дерева в деревянной грамматике соответствует программе, вычисляющей это дерево, а вхождение правила в вывод — вхождению инструкции в программу, генерации оптимальной программы соответствует нахождение вывода, минимального в смысле суммы стоимостей входящих в него правил.

Очевидно, что понятие грамматики в нормальной форме и сама процедура нормализации сохраняют свой смысл и в BURS-случае. Единственное

```
    init (t, G) :
01    for each v in t.V do
02      for each K in G.N do
03        cost(v)[K] = infinity

    boolean labels (N, v) :
01    return cost(v)[N] < infinity;

    update (v, N, r, c) :
01    if c  < cost(v)[N] then begin
02      cost(v)[N] = c;
03      rule(v)[N] = r
04    end
```

Рис. 23: Алгоритм BURS: вспомогательные функции

отличие от нормализации обычных деревянных грамматик заключается в корректном вычислении стоимостей для вновь вводимых правил: при удалении цепных правил стоимость нового правила полагается равной сумме стоимости самого цепного правила и стоимости правила для нетерминала в его правой части, а при нормализации деревянного образца стоимости правил для вновь вводимых нетерминалов полагаются равными 0.

Рассмотрим алгоритмы построения покрытия минимальной стоимости с использованием BURS.

5.1 Динамическое программирование периода компиляции

В простейшем случае для нахождения оптимального покрытия достаточно правильно воспользоваться общим алгоритмом вывода для деревянных грамматик. Как и там, первой задачей является построение множества выводов. Поскольку теперь нас интересует вывод минимальной стоимости, то разметка вершины v должна содержать, во-первых, стоимость минимального вывода поддерева с корнем v из данного нетерминала (для каждого нетерминала), и, во-вторых, то правило, которое начинает этот минимальный вывод.

Вспомогательные процедуры построения BURS-разметки показаны на рис. 23. Здесь `init` — процедура начальной инициализации. Через `cost(v)[K]` обозначена стоимость минимального вывода поддерева с корнем `v` из нетерминала `K`, первоначально эта стоимость полагается равной бесконечности для каждой вершины и каждого нетерминала. Процедура `labels(N, v)` проверяет, что поддерево с корнем `v` выводится из нетерминала `N` — для этого соответствующая стоимость должна быть меньше бесконечности. Наконец, процедура `update(v, N, r, c)` используется для вы-

```
    labeling (v) :
01    for each w in succ(v) do labeling(w);
02    for each r in G.R do
03      if r='K:x' and terminal(v) == x and leaf(v)
04      then update (v, K, r, c(r))
05      else if
06        r='K:t(L(1),...,L(k))' and
07        terminal(v) == t   and
08        (for i=1 to k) labels(L(i), son(v, i))
09      then update (v, N, r, c(r) + sum (i=1..k) cost(son(v, i))[L(i)])
```

Рис. 24: Алгоритм BURS: разметка

```
    reduce (K, v) :
01    append(v, rule(v)[K]);
02    if not leaf(v) then begin
03      'M:x(L1,L2,...,Lk)' = rule(v)[K];
04       for i=1 to k do reduce(Li,son(v,i))
      end
```

Рис. 25: Алгоритм BURS: свертка

числения минимальной стоимости вывода и его начального правила. Здесь через `rule(v)[N]` обозначено правило, начинающее минимальный вывод поддерева с корнем v из нетерминала N.

Основная часть алгоритма построения BURS-разметки показана на рис. 24. Видно, что единственным отличием этого алгоритма от такового для деревянных грамматик является учет стоимостей выводов.

Легко видеть, что данный алгоритм действительно строит вывод минимальной в смысле суммы весов входящих в него правил стоимости. Так же очевидно, что построение разметки линейно относительно размера дерева.

После построения разметки изо всех возможных выводов выбирается вывод минимальной стоимости. Очевидно,что ему соответствует вывод из стартового нетерминала. Процедура построения минимального вывода (свертка) приведена на рис. 25. Здесь через `append(v, rule(v)[K])` обозначено добавление в вывод правила `rule(v)[K]`, применяемого к вершине v.

Рассмотрим пример использования BURS. В качестве системы команд рассмотрим ту же систему, что и в части 2, только исключим из системы F терминал $index$ и заменим команду '$r_i \leftarrow index(m_j, r_k)$' на команду '$r_i \leftarrow fetch(\text{`}+\text{'}(m_j, r_k))$'.

BURS-грамматика для этой системы команд имеет следующий вид (в скобках рядом с именем правила указана его стоимость):

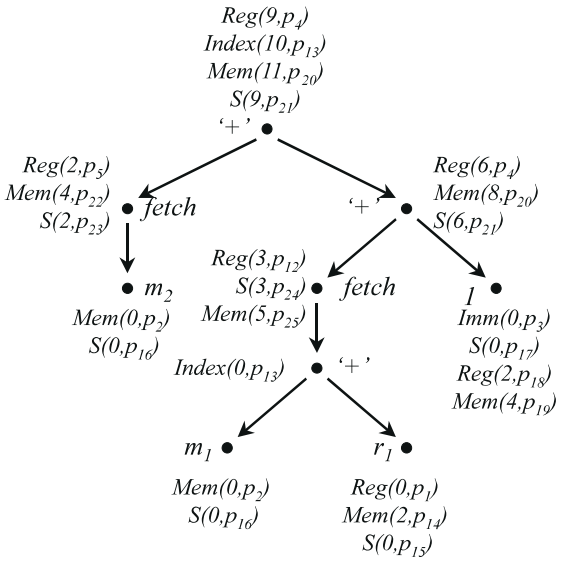

Рис. 26: Пример BURS: разметка

$$p_1(0) = \text{`}Reg : r_i\text{'}$$
$$p_2(0) = \text{`}Mem : m_i\text{'}$$
$$p_3(0) = \text{`}Imm : c\text{'}$$
$$p_4(1) = \text{`}Reg : \text{`} + \text{'}(Reg, Reg)\text{'}$$
$$p_5(2) = \text{`}Reg : fetch(Mem)\text{'}$$
$$p_6(3) = \text{`}Reg : fetch(\text{`} + \text{'}(Mem, Reg))\text{'}$$
$$p_7(2) = \text{`}Reg : Imm\text{'}$$
$$p_8(2) = \text{`}Mem : Reg\text{'}$$
$$p_9(0) = \text{`}S : Reg\text{'}$$
$$p_{10}(0) = \text{`}S : Mem\text{'}$$
$$p_{11}(0) = \text{`}S : Imm\text{'}$$

Стоимости правил в данном случае назначены из общих соображений: правилам $p_1 - p_3$ и $p_9 - p_{11}$ установлена стоимость 0 (поскольку им не соответствует никакая машинная команда), регистровому сложению (p_1) назначена стоимость 1, инструкциям, работающим с памятью (p_5, p_7, p_8), назначена стоимость 2, сложной инструкции (p_6) — стоимость 3.

После приведения этой грамматики к нормальной форме вместо правила p_6 появляются два правила

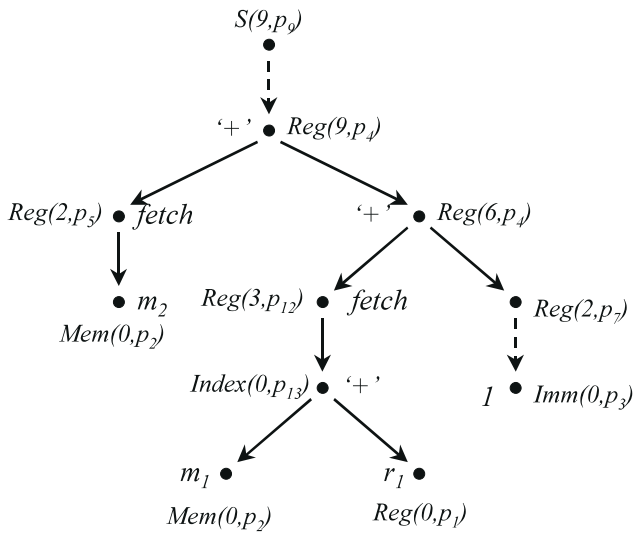

Рис. 27: Пример BURS: свертка

$$p_{12}(3) = \text{`}Reg : fetch(Index)\text{'}$$
$$p_{13}(0) = \text{`}Index : \text{`}+\text{'}(Mem, Reg)\text{'}$$

и вместо цепных правил — правила

$$p_{14}(2),_{15}(0) = \text{`}\{Mem, S\} : r_i\text{'}$$
$$p_{16}(0) = \text{`}S : m_i\text{'}$$
$$p_{17}(0),_{18}(2),_{19}(4) = \text{`}\{S, Reg, Mem\} : c\text{'}$$
$$p_{20}(3),_{21}(1) = \text{`}\{Mem, S\} : \text{`}+\text{'}(Reg, Reg)\text{'}$$
$$p_{22}(4),_{23}(2) = \text{`}\{Mem, S\} : fetch(Mem)\text{'}$$
$$p_{24}(3),_{25}(5) = \text{`}\{S, Mem\} : fetch(Index)\text{'}$$

На рис. 26 показан результат разметки для некоторого дерева в соответствии с приведенным алгоритмом. Каждая вершина помечена нетерминалом, из которого она выводится, в скобках указаны стоимость вывода и первое правило. Результат свертки приведен на рис. 27. Пунктирными линиями указаны восстановленные цепные правила.

Описанный алгоритм называют алгоритмом динамического программирования периода компиляции, поскольку стоимости выводов вычисляются при компиляции целевой программы. Такой подход применим не только в случае, когда стоимость каждого правила задана константой, но и тогда, когда она вычисляется в процессе построения вывода, исходя из более

```
    normalize_costs (s) :
01    m = min{s[K].cost | K in N};
02    for each K in N do
03      s[K].cost = s[K].cost - m
```

Рис. 28: Динамическое программирование периода компиляции компилятора: нормализация стоимостей

тонких соображений (например, результатов локального распределения регистров). Далее мы рассмотрим другой вариант решения той же задачи, ориентированный на построение быстрых кодогенераторов для грамматик с постоянными стоимостями.

5.2 Динамическое программирование периода компиляции компилятора

В разделе 2.2 нами рассматривался способ проверки выводимости данного дерева в деревянной грамматике, основанный на применении деревянных автоматов. Выбор оптимального вывода в некотором смысле можно рассматривать как способ детерминизации исходной грамматики для конкретного дерева, поскольку фактически из нее выбираются только те правила, которые участвуют в минимальном выводе. При этом оказывается, что для каждого нетерминала, появившегося в разметке, выводящее его правило однозначно определяется на основе сравнения стоимостей. Таким образом, если учесть стоимости при построении деревянного автомата, то этот автомат может быть использован для получения той же самой разметки, которая строится интерпретативным BURS-алгоритмом, изложенным в предыдущем разделе. Очевидно, что скорость генерации кода при этом оказывается существенно выше, поскольку все решения динамического программирования переносятся на этап построения автомата. В силу этого данный подход получил наименование “динамическое программирование периода компиляции компилятора”. Опишем его.

Будем считать, что деревянная грамматика удовлетворяет тем же ограничениям, которые мы сформулировали для простоты, когда описывали алгоритм построения обычного деревянного автомата. Состояние теперь будет содержать в себе не только нетерминалы, но и правила вместе со стоимостями. Удобно представлять состояние вектором, который индексируется нетерминалом и содержит пару — стоимость минимального вывода из данного нетерминала и правило, с которого этот вывод начинается. Для состояния `s` и нетерминала `K` через `s[K].rule` и `s[K].cost` обозначим соответственно это правило и стоимость. Нашей задачей теперь является построение деревянного автомата над множеством таких состояний.

Прежде всего нам потребуется процедура нормализации. Основанием для нормализации является то соображение, что с точки зрения построе-

```
    build_leaf_states (G) :
01    states = empty;
02    for each x in T where arity(x)=0 do begin
03      for all K in N do state_x[K].cost = infinity;
04      for each r='K:x (c)' in R do begin
05        if state_x[K].cost > c then begin
06          state_x[K].cost = c;
07          state_x[K].rule = r;
08        end;
09      nomalize_costs (state_x);
10      т(x) = state_x;
11      states += {state_x}
12    end;
13    return (states, т);
```

Рис. 29: Динамическое программирование периода компиляции компилятора: вычисление начальных состояний

ния минимального вывода для нас важны не истинные стоимости выводов из каждого нетерминала состояния, а их соотношения между собой. Поэтому мы всегда будем рассматривать состояния, в которых стоимости вывода каждого нетерминала соотнесены со стоимостью вывода, минимального среди всех нетерминалов данного состояния. В противном случае невозможно ограничить число состояний, поскольку истинная стоимость вывода с ростом дерева может расти неограниченно.

Процедура нормализации состояния приведена на рисунке 28. Прежде всего, среди всех элементов состояния выбирается имеющий минимальную стоимость `m` (строка `01`), после чего стоимости всех элементов состояния уменьшаются на `m` (строки `02-03`). Очвидно, что в нормализованном состоянии стоимость минимального вывода есть 0.

Алгоритм построения автомата в целом аналогичен таковому для обычной грамматики и отличается только нормализацией состояний и вычислением стоимостей. Сначала строится базовый набор состояний применением всех правил к операндам (рис. 29), при этом в каждом состоянии для каждого нетерминала сохраняется правило, выводящее его с минимальной стоимостью, и эта стоимость (строки `04-08`). Полученные состояния нормализуются, добавляются в текущее множество состояний, определяется функция перехода для операндов (строки `09-11`). Затем множество состояний пополняется применением остальных правил. Для этого из него выбираются всевозможные пары и строятся состояния, которые можно получить при условии, что выбранная пара размечает сыновей вершины, к которой применяется правило (см. рис. 30, строки `4-14`). После генерации состояния оно нормализуется, добавляется в текущее множество состояний, функция перехода автомата доопределяется (строки `15-18`). Процесс завершается,

```
    build_automaton (G) :
01    (states, т) = build_leaf_states (G);
02    do
03      for each x in T where arity(x)>0 do begin
04        for each (left, right) in (states x states) do begin
05          for each K in N do state_x[K].cost = infinity;
06          for each r='K:x(P,Q) (c)' in R do begin
07            if left[P].cost < infinity and right[Q].cost < infinity then
08            begin
09              d = left[P].cost+right[Q].cost+c;
10              if state_x[K].cost > d then begin
11                 state_x[K].cost = d;
12                 state_x[K].rule = r
13              end
14            end;
15            if state_x <> empty then begin
16              normalize_costs (state_x);
17              т(x,left,right)=state_x;
18              states += {state_x}
19            end
20          end
21        end
22      end
23    while changed(states);
24    return (states, т);
```

Рис. 30: Динамическое программирование периода компиляции компилятора: построение автомата

если никакое применение правила не меняет множества состояний.

В качестве примера работы алгоритма рассмотрим грамматику с множествами терминалов и нетерминалов

$$T = \{a, b\}$$
$$N = \{X, Y\}$$

и правилами

$$p_1(1) = \text{`}X : a\text{'}$$
$$p_2(1) = \text{`}Y : b\text{'}$$
$$p_3(2) = \text{`}X : c(X, X)\text{'}$$
$$p_4(2) = \text{`}X : c(X, Y)\text{'}$$

Обработка листьев дает следующие состояния и функцию перехода:

$$\tau(a) = \{(X, p_1, 0)\} = s_1$$
$$\tau(b) = \{(Y, p_2, 0)\} = s_2$$

Здесь состояния записаны в виде множеств троек (нетерминал, правило, нормализованная стоимость)). Из полученных состояний можно составить четыре пары и применить к ним правила p_3 и p_4, что дает одно дополнительное состояние:

$$\tau(c, s_1, s_1) = \tau(c, s_1, s_2) = \{X, p_3, 0\} = s_3$$

Заметим, что если в дереве левый сын произвольной вершины α помечен состоянием s_2, то к вершине α нельзя применить никакое правило, поскольку все они требуют наличия нетерминала X в его пометке. Следовательно, $\tau(c, s_2, s)$ не определена для всех состояний s.

Дальнейший перебор всевозможных пар состояний и применение правил не увеличивает их число; функция перехода окончательно доопределяется следующим образом:

$$\tau(c, s_1, s_3) = \tau(c, s_3, s_1) = \tau(c, s_3, s_3) = \tau(c, s_3, s_2) = s_3$$

Важным свойством BURS-грамматик является то, что для них не всегда можно построить конечное множество состояний. Рассмотрим, например, следующую грамматику:

$$p_1(1) = \text{`}X : a\text{'}$$
$$p_2(2) = \text{`}Y : a\text{'}$$
$$p_3(2) = \text{`}X : c(Y, Y)\text{'}$$
$$p_4(1) = \text{`}Y : c(X, X)\text{'}$$

Состояние, получаемое обработкой правил для листьев, есть

$$s_1 = \{(X, p_1, 0), (Y, p_2, 1)\}$$

Применение правил p_3 и p_4 к паре (s_1, s_1) дает состояние

$$s_2 = \{(X, p_3, 3), (Y, p_4, 0)\}$$

Применяя те же правила к паре (s_2, s_2), получаем состояние

$$s_3 = \{(X, p_3, 0), (Y, p_4, 5)\}$$

Еще раз применим те же правила к паре (s_3, s_3) и получим

$$s_4 = \{(X, p_3, 11), (Y, p_4, 0)\}$$

Видно, что такая схема применения правил ведет к тому, что нормализованная стоимость вывода нетерминала X неограниченно растет во всех

четных состояниях, следовательно, число состояний для данной грамматики бесконечно.

Подобные грамматики называются *расходящимися* (*diverging*); описанный выше алгоритм для них не завершается. Практика показывает, что грамматики, используемые для описания реальных систем команд, как правило не расходятся.

Нетрудно показать, что если работа описанного выше алгоритма завершается, то построенный автомат будет размечать вершины произвольного дерева теми же состояними (с точностью до нормализации стоимостей), что и интерпретативный алгоритм BURS. Следовательно, обычная операция свертки, примененная после этого к нетерминалу состояния, доставляющему минимальную стоимость, будет конструировать минимальный вывод для произвольной вершины дерева, размеченной этим состоянием.

Использование изложенного алгоритма на практике требует применения различных оптимизаций, поскольку в противном случае размер получаемого автомата оказывается неприемлемо большим. Обсуждение возможных способов оптимизации можно найти в [9, 27, 29].

Отметим, что в англоязычной литературе термин BURS принято относить именно к автоматному способу кодогенерации. Несмотря на то, что данный способ имеет преимущества с точки зрения скорости работы кодогенератора, у интерпретативного подхода есть свои сильные стороны. В простейшем случае возможно динамическое вычисление стоимости применения правила в зависимости от разметки дочерних вершин; вообще же говоря данный подход позволяет совместить разметку с вычислением различных атрибутов в вершинах дерева, значения которых могут в свою очередь влиять на процесс разметки. В качестве таких атрибутов могут выступать данные, необходимые для распределения регистров, планирования и т.д. Данный подход описан в работах [6, 11]. Следует отметить, что описываемые там средства применяют нисходящий обход дерева; это различие не является существенным.

Наконец, возможен вариант, когда динамическое программирование не используется вовсе. Это позволяет еще более увеличить скорость работы кодогенератора, но приводит к неоптимальному выбору инструкций. Техника такого рода может быть использована, например, для динамической генерации кода. Среди подходов, использующих эту идею, можно отметить жадный алгоритм, описанный в [18].

5.3 Переписывание дэгов

Как мы видели в разделе 1, фрагмент программы, предназначенный для генерации кода, адекватно описывается дэгом. Задача порождения оптимального кода для дэгов оказалась труднорешаемой для почти всех практически значимых случаев. Восходящее переписывание деревьев, рассматриваемое здесь, есть не способ генерации кода, а лишь способ оптимального выбора инструкций. В связи с этим возникает вопрос: возможно ли такое обобщение

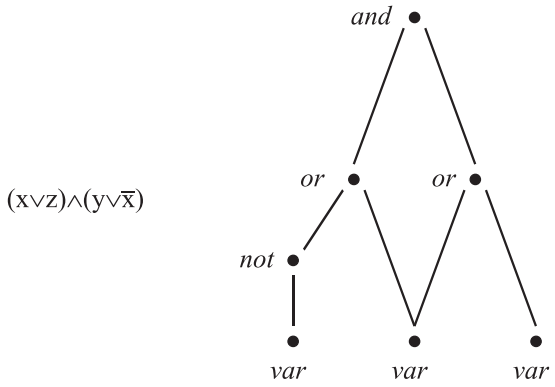

Рис. 31: Пропозициональная формула и построенный для нее дэг

алгоритма BURS, которое позволяет эффективно получить оптимальный выбор инструкций для дэгов?

Дэг отличает от дерева наличие сложных вершин. Довольно очевидно, что алгоритм разметки, примененный к дэгу, сохраняет свой смысл, поскольку разметка каждой вершины определяется разметкой ее сыновей, которые и в дереве, и в дэге определены однозначно. Неоднозначность возникает на этапе свертки: для сложной вершины вообще говоря определено несколько предзаданных нетерминалов (по одному для каждого родителя), вывод из которых может давать оптимальное с точки зрения этого родителя покрытие. Поэтому построение минимального вывода для всего дэга может потребовать перебора всех нетерминалов во всех сложных вершинах.

Теорема 5.1. *(Прэбстинг, [28]) Задача построения оптимального вывода для дэга NP-трудна.*

Доказательство. Рассмотрим следующую грамматику:

$$T = \{var, and, or, not\}$$

$$N = \{TRUE, FALSE\}$$

$$p_1(1) = \text{`}TRUE : var\text{'}$$

$$p_2(1) = \text{`}FALSE : var\text{'}$$

$$p_3(1) = \text{`}TRUE : not(FALSE)\text{'}$$

$$p_4(1) = \text{`}FALSE : not(TRUE)\text{'}$$

$$p_5(1) = \text{`}TRUE : and(TRUE, TRUE)\text{'}$$

$$p_6(1) = \text{`}FALSE : and(FALSE, TRUE)\text{'}$$

$$p_7(1) = \text{`}FALSE : and(FALSE, FALSE)\text{'}$$

$$p_8(1) = \text{`}FALSE : and(TRUE, FALSE)\text{'}$$
$$p_9(1) = \text{`}FALSE : or(FALSE, FALSE)\text{'}$$
$$p_{10}(1) = \text{`}TRUE : or(FALSE, TRUE)\text{'}$$
$$p_{11}(1) = \text{`}TRUE : or(TRUE, TRUE)\text{'}$$
$$p_{12} = \text{`}TRUE : or(TRUE, FALSE)\text{'}$$

Возьмем произвольную пропозициональную формулу F, построенную из символов переменных и связок конъюнкции, дизъюнкции и отрицания. Эта формула однозначно определяет дэг, в котором сложным вершинам соответствуют подформулы, имеющие более одного вхождения в F. Каждой переменной F в этом дэге соответствует лист, помеченный терминалом var, логическим связкам сопоставляются внутренние вершины, помеченные соответствующими терминалами and, or, not. Поскольку в исходной формуле существовало самое "внешнее" применение пропозициональной связки, построенный дэг будет иметь единственный корень. Пример соответствия между формулой и дэгом показан на рисунке 31. Сложной вершине (самому левому листу) соответствуют два вхождения в формулу переменной x.

Пусть K — число вершин дэга. Очевидно, что произвольный вывод дэга в приведенной выше грамматике имеет стоимость, не меньше K, поскольку каждое правило имеет стоимость 1 и покрывает только одну вершину. Очевидно также, что если стоимость вывода равна K, то к каждой вершине (в том числе сложной) в нем применяется только одно правило, и, следовательно, поддэг с корнем в этой вершине выводится из одного нетерминала.

Пусть данная пропозициональная формула выполнима. Очевидно, что тогда построенный по ней дэг выводится из нетерминала $TRUE$ выводом длины K. Теорема доказана в силу NP-полноты задачи о выполнимости [1]. □

Несмотря на этот негативный результат, практическое применение получило следующее эвристическое решение при синтезе кода для дэгов.

Рассмотрим дэг и его разметку, определяемую некоторой деревянной грамматикой. Очевидно, что если при свертке для данной сложной вершины α все предзаданные нетерминалы, определяемые ее предками, совпадают, то наилучшим кодом для поддэга с корнем α будет именно код, соответствующий выводу из этого нетерминала, поскольку тогда такой вывод может быть "переиспользован" в выводах, соответствующих всем его предкам. Неопределенность возникает только тогда, когда "интересы" разных предков требуют вывода поддэга с корнем α из разных нетерминалов. Упомянутая выше эвристика заключается в том, что при наличии такого конфликта поддэг с корнем α выводится по одному разу для каждого требуемого нетерминала.

Применимость данной эвристики может быть обоснована с помощью теста [12], который для данной грамматики определяет, генерирует ли эвристический алгоритм оптимальный код. Отметим, что этот тест не является

точным, так как некоторые грамматики, для которых эвристика сохраняет оптимальность порожденного кода, классифицируются им как "неоптимальные". Оценка многих грамматик, описывающих системы команд реальных процессоров, показывает, что применение эвристики не нарушает оптимальности алгоритма генерации кода.

Заключение

В данной статье мы рассмотрели основные понятия и наиболее известные результаты в области генерации кода. Эта область является активно развивающейся, поскольку прогресс вычислительной техники ставит перед разработчиками компиляторов новые задачи. В силу этого невозможно в одной статье охватить все разнообразие предлагаемых подходов.

Среди методов, не рассмотренных выше, но требующих упоминания, следует отметить синтаксически-ориентированный подход на основе постфиксных грамматик [20], кодогенерацию на основе *аффиксных* (атрибутных) грамматик [19]; алгоритм динамического программирования для решетчатых диаграмм (*trellis diagrams*), который позволяет интегрировать выбор инструкций и распределение регистров для архитектур с гетерогенными регистрами [32]; генерацию кода на основе целочисленного линейного программирования [33]; интегрированный алгоритм динамического программирования для одновременного выбора инструкций, распределения регистров и планирования [22]; подход к генерации кода для VLIW-архитектур на основе двоичного покрытия (*binate covering*) [23].

Список литературы

[1] **Гэри М., Джонсон Д.** Вычислительные машины и труднорешаемые задачи: Пер. с англ. — М.: Мир, 1982, 416 с.

[2] **Aho A.V., Johnson S.C.** Optimal Code Generation for Expression Trees // Journal of the ACM. — 1979. — Vol. 23, No. 3 — P. 488-501.

[3] **Aho A.V., Johnson S.C., Ullman J.D.** Code Generation for Expressions with Common Subexpressions // Proc. of the 3rd ACM SIGACT-SIGPLAN symp. on Principles of Programming Languages. — 1976. — P. 19-31.

[4] **Aho A.V., Johnson S.C., Ullman J.D.** Code Generation for Machines with Multiregister Operations // Proc. of the 4th ACM SIGACT-SIGPLAN symp. on Principles of Programming Languages. — 1977. — P. 21-28.

[5] **Aho A.V., Sethi R., Ullman J.D.** Compilers: Principles, Techniques, and Tools. — Addison-Wesley, 1986. — 500 p.

[6] **Aho A.V., Ganapathi M., Tjiang S.W.K.** Code Generation Using Tree Matching and Dynamic Programming // ACM Transactions on Programming Languages and Systems. — 1989. — Vol. 11, No. 4 — P. 491-516.

[7] **Bruno J., Lassagne T.** The Generation of Optimal Code for Stack Machines // Journal of the ACM. — 1975. — Vol. 22, Issue 3.

[8] **Bruno J., Sethi R.** Code Generation for a One-register Machine // Journal of the ACM. — 1976. — Vol. 23, No. 3 — P. 502-510.

[9] (Chase D.R.) An Improvement to Bottom-up Tree Pattern Matching // Proc. of 14th Annual Symp. on Principles of Programming Languages. — 1987. — P. 168-177.

[10] **Comon H., Dauchet M. et al.** Tree Automata Techniques and Applications // http://www.grappa.univ-lille3.fr/tata.

[11] **Emmelmann H., Schröer F.W., Landwehr R.** BEG — a Generator for Efficient Back Ends // Proceedings of the SIGPLAN'89 Conference on Programming Languages Design and Implementation. — 1989. — P. 227-237.

[12] **Ertl M.A.** Optimal Code Selection in DAGs // Proceedings of the 26th ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages. — 1999. — P. 242-249.

[13] **Fraser C.W., Hanson D.R.** A Retargetable C Compiler: Design and Implementation. — Addison-Wesley Pub Co., 1995. — P. 564.

[14] **Fraser C.W., Hanson D.R.** A Retargetable Compiler for ANSI C // ACM SIGPLAN Notices. — 1991. — Vol. 26, No. 10 — P. 29-43.

[15] **Fraser C.W., Hanson D.R.** A Code Generation Interface for ANSI C // Software — Practice and Experience. — 1991. — Vol. 21, No. 9 — P. 963-988.

[16] **Fraser C.W., Hanson D.R., Proebsting T.A.** Engeneering a simple, efficient code generator generator // ACM Letters on Programming Languages and Systems. — 1992. — Vol. 1, No. 3 — P. 213-226.

[17] **Fraser C.W., Henry R.R, Proebsting T.A.** BURG — Fast Optimal Instruction Selection and Tree Parsing // SIGPLAN Notices. — 1992. — Vol. 27, No. 4 — P. 68-76.

[18] **Fraser C.W., Proebsting T.A.** Finite-State Code Generation // ACM SIGPLAN Notices. — 1999. — Vol. 34, Issue 5.

[19] **Ganapathi M., Fischer C.N.** Affix grammar driven code generation // ACM Transactions on Programming Languages and Systems. — 1985. — Vol. 7, No. 4 — P. 560-599.

[20] **Glanville R.S., Graham S.L.** A New Method for Compiler Code Generation // Proc. of the 5th ASM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. — 1978. — P. 231-240.

[21] **Hoffmann C.M., O'Donnell M.J.** Pattern Matching in Trees // Journal of the ACM. — 1983. — Vol. 29, No. 1 — P. 68-95.

[22] **Keβler C., Bednarski A.** A Dynamic Programming Approach to Optimal Integrated Code Generation // ACM SIGPLAN Notices. — 2001. — Vol. 36, Issue 8 — P. 165-174.

[23] **Liao S., Devadas S., Keutzer K., Tjiang S.** Instruction Selection' Using Binate Covering for Code Size Optimization // Proc. of the IEEE/ACM Internationall Conference on Computer-aided Design. — 1995.

[24] **Morgan C.R.** Building an Optimizing Compiler. — Digital Press, 1998. — 450 p.

[25] **Muchnik S.S.** Advanced Compiler Design and Implementation. — Morgan Kaufmann Publishers, 1997. — 880 p.

[26] **Pelegrí-Llopart E., Graham S.L.** Optimal Code Generation for Expression Trees: An Application of BURS Theory // Proceedings of the Conference On Principles of Programming Languages. — 1988. — P. 294-308.

[27] **Proebsting T.A.** BURS Automata Generation // ACM Transactions on Programming Languages and Systems. — 1995. — Vol. 17, No. 3 — P. 461-486.

[28] **Proebsting T.A.** Least-Cost Instruction Selection in DAGs is NP-Complete // http://research.microsoft.com/~toddpro/papers/proof.htm

[29] **Proebsting T.A.** Simple and Efficient BURS Table Generation // Proc. of the SIGPLAN'92 Conference on Programming Languages Design and Implementation. — 1992. — P. 331-340.

[30] **Sethi R., Ullman J.D.** The Generation of Optimal Code for Arithmetic Expressions // Journal of the ACM. — 1970. — Vol. 17 No. 4 — P. 715-728.

[31] **Stallman R.M.** Using and Porting the GNU Compiler Collection. — Free Software Foundation, 1999. — 588 p.

[32] **Wess B.** Code Generation Based On Trellis Diagrams // Code Generation for Embedded Processors. — Kluwer Academic Publishers, 1995. — P. 188-202.

[33] **Wilson T., Grewal G., Henshall S., Banerji D.** An ILP-based Approach to Code Generation // Code Generation for Embedded Processors. — Kluwer Academic Publishers, 1995. — P. 103-117.