Средства эффективного синтеза объектной программы

А. Н. Терехов, Г. С. Цейтин

В статье определяется язык для написания синтезирующих частей компиляторов с алгоритмических языков высокого уровня, позволяющий удобным способом записывать алгоритмы перевода на машинный язык. Важной особенностью предполагаемого языка является возможность выбора вариантов программирования, минимизирующих "стоимость" программы (в частности, в зависимости от последующего текста программы). Предлагается способ реализации этого языка, основанный на обработке двоичных деревьев; рассматриваются алгоритмы реализации основных конструкций.

Известно несколько подходов к созданию синтезирующих частей компиляторов, основанных на принципе последовательного просмотра текста программы (см., например, [1]). В частности, было предложено записывать программы преобразования конструкций языка в машинные команды в форме макроопределений на языке ассемблера, а также были определены методы передачи информации от конструкций к подконструкциям ("запросы на значения") и от конструкций к объемлющим конструкциям ("абстрактный стек периода компиляции"). Использование этих средств существенно повышает качество рабочей программы, однако макроязык ассемблера оказался не вполне подходящим средством для решения этой задачи, так как:

При этом для порождения команд удобно использовать процесс макрогенерации. В качестве первого варианта был использован макрогенератор языка ассемблер ЕС ЭВМ. Однако этот макрогенератор оказался не вполне подходящим средством, так как:

a) обычно это язык довольно низкого уровня, не предусматривающий широких возможностей для представления разнообразной информации, необходимой во время трансляции;

b) не вся информация, используемая ассемблером, доступна программисту;

c) макроязык ассемблера позволяет выбирать порождаемые команды в зависимости от предшествующей части программы, но не дает средств для получения информации о последующей части программы.

Из-за этих особенностей макроязыка ассемблера при выборе представления информации и при написании макроопределений приходится применять довольно искусственные приемы. В данной статье предлагается способ построения и реализации другого языка, специально созданного для фазы синтеза компилятора с Алгола-68; однако этот язык имеет и самостоятельное значение и может быть использован в других компиляторах. Описываются некоторые обобщения упомянутых выше методов, связанные с двухпросмотровой схемой синтеза и направленные на дальнейшее повышение эффективности рабочей программы.

"Программирование", №6, 1975, стр. 38-48

§1. ЗАДАЧА УЧЕТА ИНФОРМАЦИИ О ПОСЛЕДУЮЩЕМ ТЕКСТЕ ПРОГРАММЫ

Как оказалось, метод однопросмотрового синтеза недостаточно эффективен в тех случаях, когда нужно выбирать какое-то представление значения или вариант программирования в зависимости от будущей ситуации. В особенности это касается использования регистров. Ответ на вопрос: "Оставить ли данное значение на регистре или сразу выгрузить его в память?" – зависит не только от наличия свободных регистров в данный момент, но и от потребности в регистрах в ходе дальнейшей работы. Это можно проиллюстрировать на примере программирования последовательных и условных предложений, а также предложений выбора (далее эти конструкции будем называть ветвящимися). Пусть перед началом программирования ветвящейся конструкции было занято несколько анонимных (т.е. содержащих промежуточные результаты вычислений) регистров. Пусть далее, при программировании некоторых ветвей этой конструкции мы вынуждены из-за нехватки регистров выгрузить внешний по отношению к данной конструкции анонимный регистр. В этом случае лучше было бы выгрузить этот регистр перед входом в ветвящуюся конструкцию, так как:

a) конструкция, следующая за ветвящейся, должна иметь фиксированную информацию об этом регистре (во время трансляции не известно, какая ветвь будет исполняться во время счета);

b) появляется возможность уменьшить длину рабочей программы (одна команда выгрузки вместо нескольких в разных ветвях). При этом в тех ветвях, в которых удалось воспользоваться значением, хранящимся в этом регистре, можно пользоваться и после выгрузки (на правах остаточной информации).

Задача эффективного программирования с учетом вариантов, возникающих в дальнейшем, может быть решена с помощью двухпроходной схемы синтеза: во время первого просмотра входного текста порождается рабочая программа, содержащая несколько вариантов для каждого спорного случая; во время второго (обратного) просмотра происходит сравнение вариантов и выбор лучшего из них. В частности, решение поставленной выше задачи о выгрузке анонимного регистра в память может быть следующим:

- в первом просмотре для каждого анонимного значения, которое находится в регистре и не используется сразу же после получения, программируем условную выгрузку этого регистра в память. Далее помечаем каким-либо образом те анонимные значения в регистрах, которые из-за нехватки регистров должны быть выгружены в память до своего использования;
- во втором просмотре исключаем из текста ненужные выгрузки.

Заметим, что решение оказалось столь простым благодаря тому, что к моменту использования анонимного значения в регистре уже известно, что этот регистр не был использован для другой цели. В более сложных задачах, связанных с повышением эффективности рабочей программы, могут возникнуть ситуации, в которых представление значения не выбрано окончательно к моменту его использования. Если при появлении подобной альтернативы мы всегда будем начинать независимое программирование двух (или более) вариантов, его придется снова раздроблять на подварианты и т.д. и в результате число рассматриваемых вариантов быстро выйдет за разумные границы. Однако поиск наилучшего варианта при учете всех таких альтернатив может быть облегчен за счет следующих двух обстоятельств:

a) две разные альтернативы могут оказаться "ортогональными" друг другу, т.е. изменения в порождаемой программе, обусловленные выбором решения по одной из альтернатив, не будут зависеть от решения по другой альтернативе;

b) каждое принимаемое решение касается программирования определенной конструкции и, вообще говоря, не влияет на программу после окончания этой конструкции.

Предлагаемый язык как раз и рассчитан на то, чтобы описывать порождение рабочей программы с учетом альтернатив, разрешаемых при дальнейшей работе, но без явного разветвления генерируемой программы на много вариантов. В последующем тексте дается описание первого варианта такого языка и реализующей его системы. Предполагается, что описываемый метод будет использован в трансляторе с языка Алгол-68, работы над которым ведутся в ВЦ ЛГУ.

§2. ОПИСАНИЕ ЯЗЫКА

Многие конструкции в языках высокого уровня имеют весьма широкие области применения и разнообразные варианты исполнения. Так, например, в АЛГОЛе 68 присваивание может работать с простыми переменными, массивами, процедурами и т.п. Более того, при одном и том же виде величин могут порождаться разные команды в зависимости от способа размещения этих величин. Очевидно, что для повышения эффективности рабочей программы необходимо подставить вместо вхождения конструкции языка последовательность машинных команд, кратчайшим образом реализующих данное действие. Для этой цели удобно использовать процесс макрогенерации, в котором роль макроопределений играют программы перевода различных конструкций. Макрогенератор, реализующий такие макроопределения, должен удовлетворять следующим требованиям:

a) язык макрогенератора должен быть языком высокого уровня, т.к. построение эффективной рабочей программы требует использования сложных макроопределений;

b) язык макрогенератора должен включать такие специфические для задачи трансляции средства, как магазины, списки, деревья и т.д.

c) макроопределения, входящие в состав транслятора, используются весьма интенсивно, поэтому макрогенератор должен удовлетворять жестким требованиям эффективности.

Последнее требование можно ослабить, заметив, что работа макрогенератора делится на два этапа: подготовка внутреннего машинного представления макроопределения, которая происходит сравнительно редко (во время генерации транслятора в рамках операционной системы), и работа подготовленных макроопределений над конкретными программами. Ясно, что повышение эффективности первого этапа не так важно по сравнению со вторым.

В данной реализации в качестве основы языка макрогенератора предлагается Алгол-68. Это новый, чрезвычайно мощный язык программирования, имеющий стройное ортогональное описание, благодаря которому можно добиться эффективной его реализации. Некоторые ограничения на язык, обусловленные описываемой реализацией, будут рассмотрены далее.

Разумеется, практическое применение данного подхода не предполагает, что этот язык будет сразу реализован с полным объемом средств Алгола-68 (наоборот, он строится до реализации полного Алгола-68 и с целью ускорить такую реализацию). На первом этапе будет использоваться вариант языка с минимальным набором средств, а полный вариант можно построить методом раскрутки.

В тексте на промежуточном языке, который поступает на вход фазы синтеза, каждая конструкция входного языка, допускающая внутренние конструкции, имеет "рамочную" структуру, т.е. образы этих внутренних конструкций вставлены между символами, относящимися собственно к данной конструкции. При использовании макроязыка ассемблера каждый такой символ трактуется как отдельная макрокоманда. Эти макрокоманды в тексте удалены друг от друга, хотя по существу они составляют одно целое и должны пользоваться общей информацией. Для того, чтобы обеспечить это средствами макроязыка ассемблера, приходится вводить специальные магазины.

В предлагаемом языке каждая конструкция рассматривается как одна процедура. Во время синтеза рабочей программы выполнение этой процедуры начинается при чтении первого символа, относящегося к такой конструкции, и продолжается до ее последнего символа. Чтение внутренних конструкций, находящихся между этими словами, связано с выполнением процедур, относящихся к этим конструкциям. Вызов этих процедур происходит внутри внешней процедуры, для инициирования такого вызова внутри внешней процедуры вставляется специальный символ **inner**.

Информация, используемая только внутри данной процедуры, изображается при помощи идентификаторов, описанных в этой процедуре. Часть информации, используемой данной процедурой, может быть глобальной, т.е. описанной вне всех процедур.

В данном языке используется еще один вид описаний. Идентификаторы, вводимые с помощью таких описаний, будем называть *сменяемыми*. Если в данном блоке нет описания сменяемого идентификатора, то значением этого идентификатора является то же значение, которое было у него во внешнем блоке или в вызвавшей процедуре (если оно было там определено), в противном случае значение определяется его описанием. Таким образом, значение сменяемого идентификатора изменяется при входе в любой блок или процедуру, где имеется его новое описание, а при выходе восстанавливается прежнее значение. Сменяемый идентификатор реализуется следующим образом: такому идентификатору отводится глобальная ячейка памяти; при входе в блок, где есть новое описание этого сменяемого идентификатора, его значение отправляется в магазин, а при выходе из этого блока оно восстанавливается. Очевидно, что глобальные идентификаторы являются частным случаем сменяемых.

Для того, чтобы иметь возможность учитывать информацию о последующем тексте программы, в язык вводятся особого вида логические переменные – *предсказатели*. Если при программировании некоторой конструкции нужно выбрать один из двух вариантов в зависимости от последующих конструкций, то в транслируемой конструкции вводится новый предсказатель и затем допускаются ветвления по этому предсказателю. Условие, задающее действительное значение предсказателя, не задается явно. Вместо этого все порождаемые команды получают определенную оценку

"стоимости". Транслятор выбирает значения предсказателей таким образом, чтобы стоимость порождаемой программы была минимальной.

§3. УСЛОВНЫЕ ЗНАЧЕНИЯ

1. Некоторые операторы в программе могут находиться внутри условных предложений и выполняться или не выполняться в зависимости от значений определенных предсказателей. Среди таких операторов могут оказаться и присваивания, и, таким образом, на выходе условного предложения, управляемого предсказателем, окажется, что значение, именуемое некоторым именем, зависит от этого предсказателя. Далее эта зависимость может распространяться на результаты операций с этим значением, на условные предложения, управляемые результатом такой операции и т. д. Поэтому считается, что любое значение, над которым в этом языке производится действие, может быть условным, например, иметь вид: **if** *A* **then** *x* **elif** *B* **then** *y* **else** *z* **fi**.

Однако учет этих условностей не должен усложнять написание макроопределений, и для автора макроопределений эти значения должны выглядеть как простые. Система, реализующая описываемый способ синтеза (далее будем говорить просто система), использует специальный способ представления этих значений в виде двоичного дерева; каждый узел соответствует некоторому предсказателю, а две дуги, выходящие из этого узла, соответствуют значениям предсказателя **true** и **false**. В висячих вершинах этого дерева находятся конкретные значения. Операции над такими деревьями (арифметические операции, присваивания, сравнения и т.п.) выполняются при помощи специальных процедур.

2. Наиболее сложная задача при реализации такой системы – это уметь в каждый момент определить, какое значение именуется данным именем в результате произведенных до данного момента присваиваний, каждое из которых было выполнено под некоторым условием. В частности, трудно установить, исполнялось ли некоторое присваивание, если в программе есть переходы и метки. Мы будем рассматривать такой язык, в котором метки и произвольные переходы запрещены, но зато допускаются условные предложения и циклы. Это ограничение не очень существенно, так как известно, что любая программа с переходами в принципе может быть преобразована к виду, содержащему только условные предложения и циклы.

Для правильного исполнения условных присваиваний вводится специальная переменная, образующая текущее условие. Значением этой переменной является условное логическое выражение, т.е. двоичное дерево описанного выше вида, в висячих вершинах которого находятся логические значения. Перед началом генерации программы эта переменная получает значение **true**. Другие значения она получает в условных предложениях и циклах.

Кроме того, значение текущего условия изменяется, если обнаруживается, что при некотором сочетании значений предсказателей программирование не может продолжаться (например, из-за отсутствия свободных регистров); в этих случаях к текущему условию конъюнктивно присоединяется условие продолжимости генерации.

Значение текущего условия влияет на исполнение присваиваний и генерацию рабочей программы. Если при текущем условии в исполняется

присваивание *x* := *y*, то фактически имени *x* присваивается значение **if** *B* **then** *y* **else** *x* **fi** (технически эта операция несколько сложнее, поскольку *B*, *x*, *y* могут быть условными). Если же при текущем условии *B* генерируется текст рабочей программы, то он должен быть снабжен пометкой, что в окончательную программу он войдет, лишь если значение в в конечном счете окажется **true**. Фактически такие пометки в рабочей программе достаточно ставить лишь в тех точках, где текущее условие изменяется.

3. Рассмотрим условное предложение **if** *A* **then** *S* **fi**. Пусть перед входом в это предложение значением текущего условия было *B*. Тогда *S* исполняется при значении текущего условия *B* & *A*. После выхода из условного предложения для текущего условия можно восстановить значение *B* (фактически мы поступаем иначе, см. §5, п.6).

Условное предложение **if** *A* **then** *S1* **else** *S2* **fi** в принципе реализуется так же, как

bool *AA*= *A*; **if** *AA* **then** *S1* **fi**; **if not** *AA* **then** *S2* **fi**.

4. Рассмотрим цикл **while** *A* **do** *S* **od**. Предположим в начале, что известное целое число *N*, ограничивающее сверху число повторений цикла. Тогда для данного цикла можно определить эквивалентное выражение S_N следующим образом:

- в качестве S_0 берем **skip**;
- в качестве S_{k+1} берем **if** *A* **then** *S*; S_k **fi**.

Способ реализации таких предложений уже описан. На самом деле не требуется определять такое *N* заранее. При выполнении цикла мы будем входить во вложенные условные выражения достаточное число раз, пока текущее условие не окажется тождественным **false**. Этот процесс может, вообще говоря, привести к бесконечному циклу, но это будет значить, что такой бесконечный цикл возможен и при некоторых конкретных, а не условных значениях используемых величин, т.е. в этом случае макроопределение и не должно оканчивать работу.

§4. ВЫБОР ЗНАЧЕНИЙ ПРЕДСКАЗАТЕЛЯ

Как уже говорилось, новый предсказатель вводится во время генерации кодов для некоторой конструкции, допускающей альтернативные варианты программирования; далее в процессе генерации возникают ссылки на этот предсказатель, либо непосредственные, либо через условные значения переменных, в которые вошел этот предсказатель. Окончательное значение предсказателя выбирается так, чтобы минимизировать "стоимость" порожденной программы. Это можно было бы сделать в конце программы сразу для всех предсказателей, но для упрощения работы желательно хотя бы для части предсказателей делать это раньше.

В принципе значение предсказателей может быть выбрано без ущерба для минимизации в любой момент, когда кончились все ссылки на этот предсказатель как прямые, так и косвенные, и единственным условным значением, зависящим от этого предсказателя, является текущая стоимость порожденной части программы. Практически в большинстве случаев удобнее

выбирать значение предсказателей при окончании той конструкции, для которой он был введен. Прямых ссылок на этот предсказатель после этого момента уже не будет, а потерями, связанными с тем, что при минимизации не будет учтен эффект возможных в будущем косвенных ссылок, мы пренебрегаем. Предсказатели, для которых принят такой режим, будем называть *локальными*. В принципе можно рассматривать *глобальные* предсказатели, значение для которых выбираются или в момент, указанный автором макроопределения, или после того, как станут недоступными все условные значения, использующие этот предсказатель (кроме текущей стоимости).

Выбор значения предсказателя *A* состоит из следующих шагов:

a) в текущее условное значение стоимости *C* вместо *A* подставляются поочередно **true** и **false**, что дает условные значения *CT* и *CF* соответственно;

b) вычисляем условное логическое значение *CT* < *CF* и присваиваем его *A* (таким образом, выбранное значение *A* может зависеть от других предсказателей); информация об этом присваивании выдается в выходной текст (см. §5, п.5);

c) во всех условных значениях, доступных в данный момент и содержащих *A*, вместо *A* подставляем присвоенное ему значение.

Описанный процесс будем называть *исключением* предсказателя.

В связи с тем, что локальные предсказатели исключаются автоматически при выходе из макроопределения, в котором они введены, для них применяется магазинный способ присвоения номеров. Именно, при входе в макроопределение запоминается максимально занятый номер предсказателя, далее предсказателям присваиваются последовательные номера, а при выходе из макроопределения все предсказатели с номерами, большими начального, исключаются; в последующих конструкциях эти номера могут быть использованы повторно.

§5. ПРЕДСТАВЛЕНИЕ УСЛОВНЫХ ЗНАЧЕНИЙ

1. Опишем подробнее представление условных значений в памяти машины и процедуры, работающие с этим значениями. Как уже было сказано, условные значения представляются двоичными деревьями; вид таких деревьев можно описать следующим образом:

mode узелX = struct (int предсказатель, **значениеX** да, нет);
mode значениеX = union (X, ref узелX);

Такие описания даются для каждого вида ***X***, допускаемого реализацией.

В начале работы системы вся память, отведенная для хранения деревьев, разбивается на ячейки, доступные для хранения значений видов **узелX**; эти ячейки связываются в список свободных ячеек. Имеется процедура, доставляющая очередную свободную ячейку, а также процедура "сборки мусора", которая вызывается при исчерпании списка свободных ячеек.

Система обрабатывает все идентификаторы простых переменных, а также элементы массивов как значения видов **значениеX**. Такая замена видов происходит автоматически.

2. Основным способом выполнения различных действий над значениями, представленными в виде двоичных деревьев, является копирование двоичного дерева в свободных ячейках памяти с выполнением соответствующего действия

на каждой висячей вершине этого дерева. В результате такого действия вместо висячей вершины может быть, в частности, вставлена копия другого дерева, полученная аналогичным процессом. Например, для того, чтобы получить сумму двух значений, заданных в виде двоичных деревьев, мы можем копировать дерево первого слагаемого, выполняя в каждой его висячей вершине следующее действие: запоминаем число, стоящее в этой вершине, затем копируем второе слагаемое с увеличением числа в каждой его висячей вершине на запомненное число из первого слагаемого, получившееся дерево вставляем на место этого числа в копию первого слагаемого. Фактически такое копирование не обязательно будет копированием полного дерева. Именно, при копировании первого дерева мы запоминаем в магазине путь от его корня к текущей вершине и, таким образом, мы имеем список значений всех предсказателей, по которым производилось ветвление на этом пути. Когда в висячей вершине первого дерева мы начнем копирование второго дерева, то оно будет выполняться уже при определенных значениях предсказателей из первого дерева; поэтому, встречая во втором дереве ветвление по предсказателю, значение которого уже записано в магазине, мы не копируем узел с таким ветвлением, а просто сразу идем по нужной ветви и рассматриваем встретившуюся там вершину вместо данного узла. Такое неполное копирование может применяться и к первому копируемому дереву, если текущее условие конъюнктивно содержит некоторые предсказатели или их отрицания. Список таких предсказателей выделяется из текущего условия посредством специальной операции (см. §5, п.4) и добавляется к началу магазина, используемого для копирования.

Рассмотрим как используется описанная техника при действиях над условными значениями.

Для присваивания переменной значения сначала копируется дерево текущего условия, после чего висячие вершины со значением **true** заменяется копией дерева источника, а висячие вершины со значением **false** – копией дерева значения, именуемого получателем. Полученное дерево объявляется новым значением переменной. Таким образом, при тех значениях предсказателей, при которых текущее условие есть **false**, переменная своего значения не изменяет.

Для получения вырезки (в данной реализации используются только одномерные массивы; в индексных позициях могут стоять только индексы) сначала копируется дерево индекса, после чего каждая висячая вершина со значением i заменяется копией дерева i-го элемента массива. Полученное дерево объявляется значением вырезки.

3. Особого рассмотрения заслуживает присваивание, в котором значение получателя может быть условным. Пусть, например, присваивание имеет вид: $a\,[i] := b$. Здесь i и b условные значения; кроме того, для каждого конкретного значения j идентификатора i значение $a\,[j]$ также может быть условным.

Обходим дерево индекса, при этом заносим в список различные значения, которые записаны в висячих вершинах. Затем для каждого значения j из этого списка осуществляем следующие действия: копируем дерево индекса, каждую висячую вершину со значением j заменяем на копию дерева текущего условия, а его висячие вершины со значениями **true** и **false** заменяем на копии деревьев b и $a\,[i]$ соответственно; висячие вершины дерева индекса со значениями, не равными j, заменяем на копии дерева $a\,[j]$. Полученное дерево объявляем новым значением $a\,[j]$.

Различные операции, соответствующие формулам Алгола-68 с операциями из стандартного вступления, реализуются аналогично описанной выше операции сложения.

Подстановка условного логического значения вместо предсказателя, входящего в некоторое условное значение, осуществляется следующим образом: происходит обход дерева, в котором нужно выполнить подстановку; если встречается узел с заменяемым предсказателем, то запоминаются условные значения, находившиеся на ветвях "да" и "нет" этого узла, а вместо самого этого узла вставляется копия подставляемого условного логического значения, в которой вместо висячих вершин со значениями **true** и **false** в свою очередь подставляются копии запомненных значений ветвей "да" и "нет" соответственно.

4. В описываемой системе существенно используются условные значения, представленные в виде деревьев. Каждое действие с условными значениями связано с копированием деревьев, подстановкой деревьев вместо висячих вершин и т. д. При этом, если не принять специальных мер, объем памяти, отведенной для хранения деревьев, может оказаться недостаточным даже для сравнительно простых случаев.

Очевидно, что задача упрощения деревьев с произвольными конкретными значениями включает в себя упрощение деревьев с булевскими значениями, т.е. задачу минимизации булевской формулы. Известно, однако, что полная минимизация булевской формулы в общем виде является исключительно трудоемкой задачей. Исходя из этого, в данной реализации не предполагается полная оптимизация деревьев. Используются лишь сравнительно несложные алгоритмы, позволяющие заметить следующие ситуации:

a) дерево эквивалентно константе;

b) дерево может быть преобразовано к виду:

if *A* **then** *x* **elif** *B* **then** *x* **elif** … **elif** *произвольное дерево* **fi**.

Здесь *x* обозначает конкретное значение, *A*, *B*, … обозначают предсказатели или их отрицания.

В частности, если в таком виде представлено значение текущего условия, причем *x* есть **false**, то для генерируемого в данный момент варианта значения определенных предсказателей зафиксированы таким образом, что *A*, *B*, … все имеют значение **false**. Это используется для упрощения копирования деревьев (см. 2). Эквивалентность константе также используется в этой реализации (см. §3, п.4). Для приведения дерева к виду b) используется следующий алгоритм.

Строится таблица из двух столбцов, каждая строка этой таблицы соответствует одному предсказателю. Сначала таблица заполняется кодом "пусто". Далее обходится дерево, которое нужно оптимизировать. В каждой висячей вершине со значением `A` для каждой строки делается следующее. Если значение соответствующего показателя определено в магазине обхода дерева, то в нужный столбец, а в противном случае – в оба столбца записывается:

- если было "пусто", то *A*;
- если было *A*, то значение не изменится;
- в других случаях пишется код "∗".

После окончания обхода дерева просматривается таблица. Если найдется строка, в обоих столбцах которой записаны значения, отличные от кода "∗", то результирующее значение будет либо константой (если эти два значения

совпадают), либо будут состоять из одного узла, обе ветви которого – конкретные значения, взятые из этой строки таблицы.

Если найдется строка, в одном из столбцов которой записано конкретное значение, то пристраиваем дерево так, чтобы в его корне стояло ветвление по предсказателю, соответствующему этой строке таблицы. Одна из ветвей, исходящих из этого узла будет указанным конкретным значением, на второй ветви расположен "остаток" исходного дерева.

Может оказаться несколько строк таблицы, содержащих конкретное значение в одном из столбцов таблицы, (и тогда, как нетрудно догадаться, это конкретное значение будет всегда одним и тем же). В этом случае в корень дерева помещается узел, построенный для одной из тех строк, в начале остатка дерева становится узел для другой такой строки и т. д. После того, как все такие строки исчерпаны, окончательный остаток дерева, присоединяемый к последнему из построенных узлов, получается копированием исходного дерева, выполняемым с учетом значений тех предсказателей, которые вошли в начальные узлы.

Заметим, что в данной реализации построение таблицы, используемой в алгоритме оптимизации деревьев, может быть совмещено с выполнением копирования. Эквивалентность дерева константе также выясняется во время копирования: при копировании очередного узла проверяется, являются ли обе ветви, исходящие из этого узла, совпадающими конкретными значениями. При положительном ответе этот узел не копируется, вместо него рассматривается конкретное значение. Таким образом, обнаруживается не только эквивалентность дерева константе, но также и эквивалентность константам его поддеревьев.

В заключение отметим, что этот алгоритм, конечно, не дает полной оптимизации деревьев, например,

a) не будет замечено, что дерево не зависит от некоторого предсказателя, хотя его и содержит;

b) не будут найдены эквивалентные поддеревья.

5. Генерируемые команды рабочей программы помещаются в выходной текст, который рассматривается как набор данных с последовательным доступом. Между командами помещается специальная информация, позволяющая на обратном просмотре выбрать последовательность команд рабочей программы в соответствии с окончательно выбранными значениями предсказателей.

Перечислим возможные составные части выходного текста:

1) записи на языке ассемблера;
2) управляющая информация:
   a) символ "если". Этот символ ставится в начале условного генерируемого участка программы. Во время обратного просмотра этот символ воспринимается как признак конца условной генерации.
   b) символ "илсе". Ставится в конце условно генерируемого участка программы. Так как при обратном просмотре он будет встречен первым, то при нем записывается условие генерации этого участка.
   c) символ "иначе". Разделяет два участка выходного текста, генерируемые при противоположных условиях. Этот символ соответствует символу **else** в тексте макроопределения.

d) символ "установить". Записывается при исключении предсказателя. При нем указывается сам предсказатель и присваиваемое значение. При обратном просмотре этот символ предшествует всем условным значениям, зависящим от этого предсказателя.

В выходном тексте, как в записях на языке ассемблера, так и в управляющей информации, могут встречаться условные значения. Они представляются в виде линейной развертки соответствующих деревьев, аналогичной обратной польской записи, так что при обратном просмотре сначала читается предсказатель, по которому происходит ветвление, а за ним обе ветви.

Перейдем к описанию алгоритма обратного просмотра. Определяется логический массив predict для хранения значений предсказателей и целая переменная flag с нулевым начальным значением для подсчета скобок. Эта же переменная используется для определения условия генерации. Выходной текст просматривается с конца, при этом происходит распознавание управляющей информации среди записей на языке ассемблера. Записи на языке ассемблера при flag = 0 переносятся в окончательный текст с заменой содержащихся в них условных значений на конкретные.

Если встретится символ "установить", то считывается предсказатель и присваиваемое ему значение (значение может быть условным, тогда оно вычисляется); затем значение предсказателя запоминается в массиве predict.

Символы "илсе", "иначе" и "если" обрабатываются по-разному, в зависимости от значения переменой flag.

Пусть flag = 0, тогда:

a) если встретится символ "илсе", то вычисляется значение следующего за ним условия генерации этого участка программы. Если это значение есть **false**, то flag := 1;

b) если встретится символ "иначе", то flag := 1;

c) символ "если" при flag = 0 пропускается.

Рассмотрим теперь обработку символов "илсе", "иначе" и "если" при flag > 0.

a) по символу "илсе": flag := flag + 1;

b) по символу "иначе": если flag = 1, то flag := 0;

c) по символу "если": flag := flag - 1.

6. Рассмотрим подробнее работу конструкций, влияющих на текущее условие и на значения предсказателей.

Условное предложение if *A* then *S1* else *S2* fi.

Действия, исполняемые по символу **then**: в выходной текст помещается управляющий символ "если", в магазине запоминается конъюнкция текущего условия и отрицания *A*; текущему условию присваивается новое значение – конъюнкция прежнего значения и *A*.

Действия, выполняемые по символу **else**: в выходной текст помещается управляющий символ "иначе"; текущее условие и вершина магазина обмениваются значениями.

Действия, выполняемые по символу **fi**: текущее условие выдается в выходной текст, затем выдается управляющий символ "илсе"; в качестве текущего условия берется дизъюнкция текущего условия и значения, записанного в вершине магазина; из магазина удаляется последняя запись.

Цикл while *A* do *S* od.

Действия, исполняемые по символу **while:** в некоторой сменяемой переменной запоминается текущее значение указателя вершины магазина; в магазине запоминается адрес первой команды *A*; в вершину магазина записывается значение **false**.

Действия, исполняемые по символу **do**: в выходной текст помещается управляющий символ "если"; в вершину магазина записыается дизъюнкция старого значения и конъюнкции текущего условия и отрицания *A*; текущему условию присваивается новое значение – конъюнкция прежнего значения и *A*; вычисленное текущее условие запоминается в магазине.

Действия, исполняемые по символу **od**: если текущее значение не тождественно **false**, то управление передается на программу *A* (по адресу, записанному в магазине), в противном случае все текущие условия, запомненные в магазине, выдаются в выходной текст, причем, вслед за каждым из них выдается управляющий символ "илсе"; в качестве текущего условия берется дизъюнкция текущих условий, накопленная в вершине магазина.

Начало и конец макроопределения.

При входе в макроопределение в магазине запоминается максимальный занятый номер локального предсказателя и текущее значение стоимости. Затем текущее значение стоимости устанавливается на 0. При выходе из макроопределения производятся следующие действия:

a) поочередно исключаются все локальные предсказатели, введенные в данном макроопределении;

b) вычисленное значение стоимости данного макроопределения добавляется к значению стоимости, запомненному при входе, и результат становится новым текущим значением стоимости;

c) восстанавливается прежнее значение максимального номера локального предсказателя;

d) из магазина удаляются записи, сделанные при входе.

7. В системе имеется процедура "сборки мусора", которая вызывается, когда список свободных ячеек для записи узлов уже исчерпан, но требуется построить еще один узел. Эта процедура состоит, как обычно, в обходе деревьев всех доступных условных значений с нанесением метки на каждый пройденный узел, после чего просматривается весь массив ячеек для узлов, непомеченные узлы связываются в новый список свободных ячеек, а с помеченных снимаются метки. "Сборка мусора" может произойти во время копирования некоторого дерева, поэтому просматриваться должны не только "законченные" условные значения, но и частичные результаты копирования. В связи с этим копирование организовано таким образом, что новый узел создается лишь после того, как определены обе выходящие из него ветви (если окажется, что обе ветви совпадают, то есть представляют одно и то же конкретное значение, то узел не строится вовсе, см. п. 4). Если для будущего узла готова лишь одна из ветвей, то она записывается в магазин, где хранится до получения второй ветви. Этот магазин также просматривается во время сборки мусора.

Для перебора всех доступных в данный момент "законченных" условных значений ведется специальный список адресов таких значений; этот же список используется и при исключении предсказателей, когда нужно во всех доступных значениях подставить вместо исключаемого предсказателя выбранное для него значение (см. §4, п. 1). Этот список адресов организован как цепной список, используемый в режиме магазина, то есть новые элементы

присоединяются и убираются только в начале списка. В начале работы системы этот список содержит адреса нескольких глобальных системных переменных. При исполнении каждого локального описания идентификатора, используемого в макроопределении, его адрес заносится в указанный список (для массивов заносится пара из начального и конечного адресов). При выполнении нового описания сменяемой переменной в список заносится адрес его прежнего значения. Наконец, адрес начала этого списка сам рассматривается как сменяемая переменная, переописываемая в каждом блоке, содержащем описания; поэтому при выходе из блока вся часть списка, надстроенная внутри этого блока, отбрасывается, и локальные идентификаторы блока становятся, таким образом, объектами для сборки мусора.

Поступила в редакцию 30 июня 1975

ЛИТЕРАТУРА

1. *P. Branquart, J. Lewi.* A scheme of storage allocation and garbage collection for ALGOL 68. Report R133, MBLE Res. Lab., April 1970.