Евгений Налимов, Алексей Недоря

Перенацеливаемый оптимизирующий Модула-2/Оберон-2 компилятор

Евгений Налимов, Алексей Недоря (1995) «Перенацеливаемый оптимизирующий Модула-2/Оберон-2 компилятор» // Технология программирования, 1995, Т.1, No.2, с.

Сведения об авторах

Евгений Налимов является специалистом в области разработки компиляторов, окончил Новосибирский Государственный Университет, работал в ВЦ СО АН СССР, занимался школьной информатикой, принимал участие в разработке оптимизатора в проекте СОКРАТ (программное обеспечение для бортовых машин). В настоящее время работает в Институте систем информатики Российской Академии Наук (ИСИ РАН, Новосибирск) и является руководителем отдела кодогенерации в xTech Ltd.

Email: enal@iisnw.iis.nsk.su.

Алексей Недоря является специалистом в области разработки компиляторов и операционных систем, окончил Новосибирский Государственный Университет, работал сотрудником ВЦ СО АН СССР, был одним из ведущих разработчиков в проекте создания системного программного обеспечения и 32-разрядного процессора, ориентированного на язык Modula-2 (проект Kronos). В настоящее время работает в Институте систем информатики им.А.П.Ершова Российской Академии Наук (ИСИ РАН, Новосибирск) и является исполнительным директором фирмы по разработке переносимого инструментального программного обеспечения (xTech Ltd.), активный участник международного комитета по стандартизации языка Oberon-2. Email: ned@iisnw.iis.nsk.su.

Аннотация

В статье, являющейся независимым продолжением работы A.Nedorya, помещенной в предыдущем номере нашего журнала (The Technology of Programming, 1995, Vol.1, No.1, p.46-48), раскрываются особенности технологии построения перенацеливаемых (retargetable) компиляторов, которые отличаются от переносимых (portable) прежде всего тем, что позволяют легко адаптироваться к иной архитектуре. Причем это в значительной мере относится к кодогенератору — наиболее критичной компоненте компилятора при переходе на другую платформу. В работе затронуты такие вопросы, как внутреннее представление программ в генераторе кода, перевод программы в SSA-форму, вопросы оптимизации и собственно генерации кода. Идеи авторов были апробированы на коммерческой системе XDS (xTech Development System), которая предназначена для разработки переносимого программного обеспечения с использованием надежных языков высокого уровня Modula-2 (ISO стандарт) и Oberon-2. Авторы статьи, являющиеся одновременно и авторами системы XDS, поставили перед собой цель не просто максимально использовать достоинства объектно-ориентированного подхода в своей технологии построения перенацеливаемых компиляторов, но и добиться при этом крайне эффективной кодогенерации, на уровне лучших С-компиляторов. Как показало сравнительное тестирование, результаты во многих случаях не только не уступают таким, к примеру, промышленным компиляторам, как Symantec и Watcom, но и даже превосходят их.

На сегодняшний день семейство инструментальных средств XDS включает наряду с компилятором еще и транслятор с входных языков Modula-2/Oberon-2 на C (ANSI-стандарт), который в настоящее время перенесен на все популярные платформы, включая MS-DOS, OS/2, MacOS, UNIX с поддержкой требований большинства известных С-компиляторов, применяемых на этих платформах. При разработке одной программы можно использовать и компилятор, и транслятор на С.

В настоящее время имеется бета-версия 32-разрядного компилятора для MS-DOS (ее можно найти на CD-приложении к этому номеру журнала). Полностью готов кодогенератор для Intel x86 и идет работа по переносу компилятора в Win32, Windows NT, Windows 95, OS/2 и Linux. Следующий шаг — перенос на архитектуру Motorola 68K, затем последует перенос на RISC-архитектуру. Сам компилятор написан на языке Oberon (около 50.000 строк исходного текста), что и позволило обеспечить наиболее адекватную и эффективную поддержку разработанной авторами технологии построения перенастраиваемых компиляторов. При разработке архитектуры кодогенератора был использован опыт работ по российскому проекту SOCRAT — разработке программного обеспечения для бортовых компьютеров.

Ключевые слова

Перенацеливаемый компилятор, генерация кода, распределение регистров.

1. ВВЕДЕНИЕ

В первом номере журнала, в статье [Нед95] был описан подход к разработке переносимых компиляторов, принятый в системе XDS (xTech Development System). Система предназначена для разработки переносимого программного обеспечения с использование надежных языков высокого уровня Модулы-2 (ISO стандарт) и Оберона-2. Сама система написана на Обероне-2. В первой очереди системы была реализована трансляция входных языков в ANSI C. Такая трансляция имеет несомненные достоинства: наличие всего одного генератора (back-end'a) позволяет перенести систему на любую платформу для которой есть С компилятор. В настоящее время система перенесена на все популярные платформы, включая MS-DOS, OS/2, Mac, различные Юниксы. Далее мы будем использовать термин «транслятор» для системы порождающей С текст, и «компилятор», для системы, порождающей машинный код для конкретной платформы.

Наряду с достоинствами, трансляторы обладают серьезными недостатками. Перечислим некоторые из них:

1. Проверки времени исполнения (range checks, NIL checks, etc), как правило, реализуются в трансляторах процедурными вызовами, что приводит к заметному замедлению программы (в несколько раз, если все проверки включены). Выключение проверок ускоряет программу ценой снижения надежности. Некоторые классы проверок, например, контроль целочисленного переполнения, не реализованы в трансляторе, их реализация привела бы к еще более существенному замедлению сгенерированной программы.
2. Необходимость компиляции полученного текста С компилятором приводит к существенному замедлению процесса разработки. В некоторых случаях ошибки в каком-либо С компиляторе приводят к необходимости менять генерацию кода.
3. Сложной задачей является обеспечение отладки программы в терминах исходного текста. Эта задача решана в XDS только частично.

Для преодоления этих недостатков было принято решение разработать легко перенацеливаемый генератор, порождающий машинный код для каждой платформы. Мы используем термин перенацеливаемый (retargetable) вместо переносимый (portable), чтобы подчеркнуть простоту перехода на другую архитектуру. С нашей точки зрения в перенацеливаемом компиляторе объем текста, который переписывается при переходе на новую архитектуру, должен быть минимальным. Так, например, компилятор OP2 [Cre90] является переносимым, но не перенацеливаемым, так как весь генератор кода должен быть переписан при переносе. Заметим, что С генератор является идеально перенацеливаемым, любой другой генератор будет перенацеливаться существенно труднее.

Легкость перенацеливания — не единственное требование к генератору. Для того, чтобы компилятор был конкурентно-способным необходимо:

- порождать достаточно эффективный код;

- поддерживать стандарты ОС;
- обеспечить интерфейс с C;
- обеспечить максимальную легкость переноса;
- обеспечить полную совместимость транслятора и компилятора.

Кроме того, компилятор должен быть достаточно прост, поэтому при выборе набора оптимизаций мы оценивали как коэффициент усложнения компилятора при добавлении данной оптимизации, так и возможный выигрыш от нее. Если отношение усложнения к возможному выигрышу было слишком велико, то такая оптимизация отбрасывалась. Рассмотрим более подробно требования к компилятору:

Легкость перенацеливания

Для упрощения перенацеливания необходимо локализовать и минимизировать машинно-зависимые части компилятора. В нашей системе машинно-зависимым является только генератор кода, синтаксический анализатор (front-end) и оптимизатор машинно-независимы. Существенная часть генератора кода не пишется, а порождается из описания архитектуры. Для этого разработан язык описания правил и транслятор, который порождает текст на Обероне.

Порождение оптимального кода

Мы считаем, что наш компилятор должен порождать код на уровне лучших C компиляторов. В настоящее время реализованы те оптимизации, которые дают максимальный эффект (оценка эффекта от оптимизаций есть, например, в работе [Knu71]). В дальнейшем планируется увеличение набора оптимизирующих преобразований.

Стандартность компилятора

Компилятор должен быть максимально стандартным с точки зрения операционной системы. Он должен порождать стандартные объектные файлы, должны использоваться стандартные соглашения о связях и т.д. Пользователь должен иметь возможность использовать стандартные отладчики и другие полезные возможности.

Интерфейс с C

Язык C в настоящее время является существенно выделенным языком. Для этого языка реализовано существенно больше библиотек, чем для языков Модула-2 и Оберон-2. Удобный доступ к C библиотекам может существенно повысить применимость компиляторов. К сожалению, различные C компиляторы могут различатся в передаче параметров, именовании объектов и т.д., поэтому в компиляторе необходима возможность подстройки под конкретный C компилятор.

Легкость переноса

Для упрощения переноса программного обеспечения между различными платформами необходимо обеспечить одинаковую среду программирования, а именно, на всех платформах языки и библиотеки должны быть реализованы одинаково. Для этого особенно важно наличие стандартов на языки и библиотеки.

Совместимость компилятора и транслятора

Полная совместимость компилятора и транслятора дает возможность разработывать программное обеспечение на платформе, для которой есть компилятор, а затем переносить его с помощью транслятора на другую платформу. В нашей системе реализована еще более сильная возможность: при разработке одной программы можно использовать одновременно и компилятор, и транслятор для разных модулей.

В настоящее время реализована бета-версия компилятора с генерацией кода для старших моделей x86 (x>=3, flat memory model). Далее в статье мы описываем структуру генератора (back-end), основные решения и основные алгоритмы. При разработке генератора использовался опыт, приобретенный одним из авторов в рамках проекта СОКРАТ [Пот92]. Оттуда был взят ряд архитектурных решений, а также алгоритмы некоторых оптимизирующих преобразований. Замечание: далее в тексте слово генератор будет употребляться в двух значениях: для обозначения части компилятора (back-end) и для обозначения генератора машинных команд (т.е. части back-end'a). Мы надеемся, что из контекста всегда очевидно, какое из этих значений используется.

Генератор состоит из четырех частей:

- интерфейс с первым проходом, порождающий представление удобное для оптимизации
- оптимизатор
- генератор машинных команд
- выходной интерфейс (порождение объектного файла)

Первые две части являются машинно-независимыми.

В текущей версии генератора не реализованы межпроцедурный анализ или оптимизации. Планируется добавить эти возможности позднее.

2. ВНУТРЕННЕЕ ПРЕДСТАВЛЕНИЕ ПРОГРАММ В ГЕНЕРАТОРЕ

Первый проход компилятора строит представление всего компилируемого модуля в виде дерева, после чего для каждой процедуры вызывает генератор.

Генератор строит свое представление процедуры — управляющий граф (типичное представление для оптимизаторов и генераторов кода); вершинами его являются линейные участки программы, а те, в свою очередь, состоят из триад.

Граф имеет выделенный стартовый узел. Финишных узлов может быть несколько — это все те, из которых не выходит ни одной дуги.

В узел (кроме стартового) может входить любое количество дуг, выходят же обычно одна или две. Одна выходная дуга соответствует случаю явной или неявной передачи управления, две — случаю ветвления. Несколько выходных дуг может быть в случае оператора выбора.

Линейные участки состоят из триад, представляющих элементарные операции. Как правило, у триады фиксировано количество параметров; исключением являются т.н.«мультиоперации» — сложение, умножение, логические операции, т.е. коммутативные операции; у них может быть сколько угодно параметров. Введение мультиопераций расширяет возможности для оптимизации. Например, оператор

A := (1 + B) + 2

в классическом триадном представлении состоит из двух триад:

T := 1 + B

A := T + 2.

Объединение же их в одну мультиоперацию

A := 1 + B + 2

дает возможность вычислить константную часть и заменить триаду на

A := 3 + B.

Кроме того, у сложения есть обратная операция — вычитание; ее удобно обозначать соответствующей пометкой на параметре.

Система типов во внутреннем представлении чрезвычайно проста. Объекты языка делятся на два класса: длинные и короткие. К первым относятся все массивы и записи, а также множества длиной более 4 байт; работа с ними ведется через указатели; на регистрах они не размещаются. Вторые — те, которые могут быть размещены на регистрах (целые и вещественные числа, множества длиной до 4 байт, логические величины, значения перечислимых типов, указатели). Комплексные числа заменяются парами вещественных чисел. Заметим, что некоторые «короткие» значения (например, типа LONGREAL — 8 байт) могут быть длиннее некоторых «длинных» (например, массива длиной 2 байта).

Короткие объекты бывают нескольких фиксированных размеров; с каждым типом связано множество его допустимых размеров. Строго говоря, это множество машинно-зависимо; к счастью, большая часть оптимизатора и генератора не зависит от конкретных размеров.

Сложные языковые конструкции во внутреннем представлении, как правило, разбиты на составные части:

- индексации массивов разделены на проверку индекса, адресную арифметику, явную загрузку из памяти;
- при работе с записями (и объектами в языке Оберон-2) вставляются явные триады загрузки/записи в память;
- при вызовах процедур вместо VAR-параметров явно передаются адреса; при обращениях к VARпараметрам внутри процедур явно вставляются команды косвенной загрузки/записи;
- все переменные, к которым обращаются текстуально вложенные в данную процедуры, собираются в одну запись, и указатель на эту запись передается дополнительным параметром во вложенные процедуры; там работа с этими переменными и ведется как с полями записи;
- явно вставлены триады проверки указателей на NIL, диапазонов переменных, приема «длинных» объектов по значению и т.д.
- конструкции, связанные с выделением памяти и обработкой исключительных ситуаций, заменены на вызовы процедур;
- логические выражения разбиты на последовательность переходов и, если надо, на явно записанные присваивания 0 и 1.

Такая организация требует, чтобы смещения полей записей были вычислены еще до перевода программы во внутреннее представление; этим занимается модуль перевода программы из «древесного» во внутреннее представление.

Двумя важными исключениями являются операторы FOR и CASE; для их представления введены специальные триады. Эти операторы будут разбиты на более мелкие части уже самим оптимизатором. Сделано это потому, что для эффективной и корректной трансляции этих операторов необходимо иметь информацию о возможных диапазонах переменных (которая собирается оптимизатором), а также об использовании переменной цикла внутри него уже после проведения оптимизаций.

Таким образом, внутреннее представление, являясь низкоуровневым, остается тем не менее почти машинно-независимым; семантика операций и их набор не зависит от целевой архитектуры, в отличие от, например, RTL, используемого в компиляторе GNU C [Sta92].

3. ПЕРЕВОД ПРОГРАММЫ В SSA ФОРМУ И ИСПОЛЬЗОВАНИЕ ЕЕ СВОЙСТВ

Вегман (Mark Wegman) и Задек (Kenneth Zadek) предложили способ хранения use-def информации, удобный для модификации и требующий относительно немного памяти. Они назвали его SSA-формой, то есть формой программы со статически единственными присваиваниями (Static Single Assignment Form). Эта форма и способ ее построения описаны в [CFR91].

В этой форме каждая переменная имеет единственную точку определения (эта точка может быть исполнена несколько раз, поэтому и говорится о статически, а не динамически единственных присваиваниях). Для каждой точки определения переменной создаются уникальные переменные.

Если использованию переменной могут соответствовать несколько точек определения, то в программу вставляются специального рода функции, называемые Ф-функциями. Эти функции определяются следующим образом: Пусть в узел x графа управления входят n дуг. Тогда Ф-функция, стоящая на входе в этот узел, имеет n аргументов и выдает в качестве результата аргумент с номером, равным номеру входящей дуги, по которой в узел пришло управление. Такие Ф-функции должны стоять во всех линейных участках, куда по разным дугам могут приходить разные значения переменной.

Первое, что мы делаем с программой во внутреннем представлении — это переводим ее в SSA форму; при этом, в отличие от предложенного в [CFR91] способа, мы переводим в SSA форму только работу с «короткими» (т.е. скалярными) объектами. Как было сказано ранее, при обращении к «длинными» объектам стоят явные триады загрузки/записи в память; более того, при переводе в SSA форму приходится иногда вставлять такие триады и для «коротких» объектов — например, после косвенной записи в память по указателю надо вставить загрузку всех переменных, на которые этот указатель мог указывать. Чтобы вставить такие триады, приходится выполнять анализ указателей, а именно, выяснять, куда может указывать тот или иной указатель. В настоящий момент анализ весьма неточен: он использует только информацию о текущей процедуре, поэтому любой вызов процедуры либо использование глобальной переменной вынуждает «предполагать худшее»; тем не менее он собирает на удивление много полезной информации, которая используется не только здесь, но и при выполнении собственно оптимизаций.

Мы сочли удобным использовать SSA-форму не только как средство хранения use-def информации, но при проведении оптимизаций воспользоваться многими ее свойствами, в первую очередь следующими:

- присваивание переменной доминирует (в графовом смысле — см., например, [Нес77]) над всеми ее использованиями; это позволяет, в частности, эффективно организовать хранение def-use информации (хватает обычного списка);
- текстуальное совпадение переменных означает совпадение их значений;
- можно сколь угодно продлевать область жизни пере-

менных, не беспокоясь о корректности программы.

Разумеется, резкое упрощение оптимизатора не является бесплатным. При выполнении даже простейших трансформаций может оказаться, что два экземпляра одной переменной будут существовать одновременно; при этом программа (благодаря SSA форме) останется корректной, но связь переменных с исходной программой может быть потеряна. Мало того, архитектура современных процессоров не поддерживает исполнение программ прямо в SSA форме, поэтому перед собственно генерацией кода программу надо из этой формы вывести, чем и занимается достаточно объемный кусок кода. Те не менее нам представляется, что в целом благодаря применению SSA формы компилятор ощутимо сократился.

4. ОПТИМИЗАЦИИ

При выборе алгоритмов оптимизации особое внимание уделялось эквивалентности преобразований. При этом приходится учитывать наличие в языке Модула-2 механизма обработки исключительных ситуаций — от многих оптимизаций приходится отказываться, чтобы сохранить эквивалентность программ.. Большинство программ не пользуются обработкой исключительных ситуаций, но тем не менее выяснить это, не имея текста всей программы, невозможно. Может быть, имеет смысл предусмотреть специальный режим работы компилятора, в котором он будет игнорировать возможность перехвата ошибок. Оптимизатор выполняет следующие оптимизирующие преобразования:

- счет константных и частично константных выражений
- удаление явно избыточного кода
- протяжка присваиваний (copy propagation); при этом происходит и протяжка констант (constant propagation); после этой оптимизации в программе не остается ни одного оператора присваивания — это стало возможным благодаря SSA форме программы
- поиск общих подвыражений (common subexpression elimination)
- вынос инвариантных выражений из цикла
- уменьшение силы операций в циклах

В настоящий момент не реализованы, но планируются следующие оптимизации:

- глобальное удаление избыточного кода, аналогичное описанному в [CFR91]
- поиск монотонно возрастающих и убывающих в циклах переменных и связанные с ними оптимизации
- вынос из сложений в циклах инвариантных членов
- разворачивание циклов (loop unrolling)
- открытая подстановка процедур (procedure inlining)

Далее мы подробнее опишем некоторые оптимизации.

4.1. Поиск общих подвыражений

Мы обходим все триады в графе, и для каждой:

1) предположим, что в триаде один из параметров — переменная; тогда можно пройти по ее def-use списку, пытаясь найти использование, такое, что:
   - она доминирует (в графовом смысле) над нашей триадой,
   - все параметры совпадают;

   тогда можно всюду заменить все использования результата нашей триады на результат найденной, а нашу триаду удалить;
2) если триада — мультиоперация, то допустимо, чтобы совпадали не все параметры; так, триады

   A := B + C + D

   X := C + B + Z будут заменены на

   T := B + C

   A := T + D

   X := T + Z
3) если триада — это триада чтения из памяти по указателю, то одного равенства указателей недостаточно; надо, чтобы в промежутке между чтениями не было записей в эту память; здесь приходится явно обходить граф и проверять это условие — анализ, куда может записать та или иная триада, был выполнен в процессе перевода программы в SSA форму;
4) выражения из одних констант тоже бывают, например, загрузки по константным указателям; поэтому в процессе работы алгоритма строится список таких триад, и повстречав новую триаду-кандидат, достаточно просмотреть этот список в поисках подходящей триады.
5) алгоритм производит также объединение условий: если имеются два текстуально совпадающих условия, причем
   - одно из них доминирует над другим, и
   - одна из выходящих из него дуг доминирует над вторым условием, то вместо второго условия можно поставить безусловный переход по нужной ветви.

Следует заметить, что будут найдены (и удалены) и избыточные проверки (на NIL, индексов, диапазонов, типов и т.д.).

4.2. Вынос инвариантных выражений

Из циклов выносятся инвариантные там выражения (мы определяем цикл так же, как это было сделано в работе [ASU86] — цикл определяется своей обратной дугой); при этом надо, чтобы выражение либо было «безопасно» (т.е. при его вычислении не могло произойти исключительной ситуайии), либо находилось на связующем участке цикла и в цикле до него не было опасных операций; кроме того, из циклов может быть вынесено чтение по инвариантному в нем указателю, если будет выяснено, что нигде в цикле не пишется в память, на которую может указывать этот указатель.

4.3. Уменьшение силы операций

Для умножений в циклах производится уменьшение силы операций; используемый алгоритм по мощности не уступает описанному в [ACK81] и значительно превосходит описанный в [Wol92], который тоже имеет дело с программой в SSA форме. Подробно изложить алгоритм здесь не позволяют ограничения по месту, основная же идея такова:

1) найдем сначала все индуктивные в цикле переменные; это такие переменные, которые получаются:
   - либо как сумма или разность индуктивных переменных и констант (либо инвариантных в цикле переменных),
   - либо как результат Ф-функции, все параметры которых, в свою очередь, есть индуктивные переменные и константы либо инвариантные в цикле переменные;
2) рассмотрим «безопасное» умножение индуктивной переменной на константу:

   A := B * 5

если

B := X + Y + 1,

то вычисление A можно заменить на

A := X * 5 + Y * 5 + 5,

и итеративно применить алгоритм для X * 5, Y * 5, а если

B := Ф (1, Z)

то вычисление A можно заменить на

A := Ф (5, Z * 5)

и опять-таки итеративно применить алгоритм для Z * 5.

3) умножение двух индуктивных переменных тоже можно заменить, алгоритм усложняется несущественно.

5. ПОДГОТОВКА ПРОГРАММЫ К ГЕНЕРАЦИИ

После выполнения оптимизаций производятся следующие действия:

1) Мультиоперации снова разбиваются на части:

A := B + C - D

заменяется на

T := B + C

A := T - D.

2) Производится ряд машинно-ориентированных замен, например, «безопасные» умножения на константы расписываются на сдвиги и сложения, деления на степень двойки заменяются сдвигами и т.д.

3) Триада вызова процедуры разбивается на триады передачи параметров и собственно триаду вызова.

4) Некоторые операции (такие, как И-НЕ) тоже разбиваются на составные части.

Таким образом, программа, оставаясь еще в триадном представлении, становится еще более машинноориентирована.

Теперь мы начинаем выводить программу из SSA формы; для этого

1) Считаем зацепленность переменных; в SSA форме это сделать очень просто: две переменные зацеплены, если одна из них «жива» (т.е. уже получила значение и еще будет использована) в момент определения другой.

2) Теперь объединяем переменные в кластера — группы взаимно незацепленных переменных, которые можно разместить в одном и том же месте (ячейке памяти или регистре). При этом задача экономии памяти не ставится: переменные объединяются лишь из соображений экономии пересылок. Фактически раскрашивается граф зацепленности, причем известна выгода от размещения переменной в конкретной ячейке памяти. Возможные источники такой выгоды таковы:

- одна переменная — результат Ф-функции, другая — ее аргумент, и одна из них уже размещена в этой ячейке;
- переменная читается из или записывается в эту ячейку (например, переменная загружается из глобала в начале процедуры; тогда имеет смысл разместить ее в этом самом глобале).

Разумеется, далеко не всегда возможно разместить переменную в памяти, из которой она была загружена — в таком случае при генерации возникнет пересылка. Изощренные алгоритмы раскраски графа не применяются: для практических нужд хватает жадного алгоритма, размещающего переменные в порядке убывания выгод от размещения. Таким образом, каждому кластеру сопоставляется ячейка памяти, в которой он будет размещен, если не окажется на регистре.

3) Наконец, мы строим по программе ациклический граф (DAG — direct acyclic graph), т.е. снова преобразуем программу к древесному виду. При этом некоторые операции сливаются (например, для Intel x86 все операции сдвига заменяются одной командой «сдвиг»; конкретную команду подставит уже самый нижний уровень генератора кода) — это сделано исключительно для экономии кода в селекторе команд.

6. ГЕНЕРАЦИЯ МАШИННОГО КОДА

Для генерации кода применяется техника, аналогичная используемой в [ESL89] или [FrH95] — покрытие больших деревьев небольшими фрагментами-образцами, с заменой сопоставленного образцу фрагмента дерева на результирующую вершину. Так как покрытие, строго говоря, неоднозначно, с каждым образцом связывается его цена — в нашем случае это количество тактов процессора, необходимых для выполнения этой команды (или ее части, например, для способа адресации); из всех покрытий выбирается покрытие наименьшей стоимости.

Эту технику проще всего пояснить на примере; пусть есть набор правил

(1) statement ::= store (addr, reg) стоимость 1

(2) addr ::= based стоимость 0

(3) addr ::= based + scaled стоимость 1

(4) based ::= reg стоимость 0

(5) scaled ::= reg стоимость 0

(6) scaled ::= mult (reg, 2) стоимость 0

(7) scaled ::= mult (reg, 4) стоимость 0

(8) scaled ::= mult (reg, 8) стоимость 0

Пусть в триадном представлении было написано

T := 4 * I

U := A + T

U := B

Пусть, далее, переменные A, I и B находятся на регистрах и мы ищем оптимальное покрытие этого фрагмента программы (а точнее, полученного из него дерева); тогда имеется ровно одно такое покрытие:

- переменной T будет сопоставлено scaled по правилу (7),
- с помощью правила (4) из reg (для A) будет получено based,
- наконец, с помощью правила (1) вершине дерева будет сопоставлен нетерминал statement.

Этот алгоритм позволяет хорошо генерировать код для деревьев; беда, однако, в том, что у нас не деревья, а DAG. Если рассматривать его просто как лес деревьев и генерировать код для каждого дерева в отдельности, результат будет не очень хорош: так, программа

I. (1) T := 4 * I
U := A + T
(2) * U := B
W = C + T
(3) * W := D

будет состоять из трех деревьев (их вершины явно отмечены), и для процессора Intel x86 получится чегонибудь вроде

lea rT,rI*4

mov [rA+rT],rB

mov [rC+rT],rD

(здесь rX — это выделенный под переменную X регистр). В

то же время хочется сгенерировать

```
mov [rA+rI*4],rB
mov [rC+rI*4],rD
```

Видно, что не у всех деревьев вершины следует считать явно; результат некоторых деревьев хорошо бы посчитать неявно — например, вершины T в приведенном примере.

Именно так и сделано в нашем генераторе. При разметке DAG'a (так называется процесс поиска покрытия) для каждой вершины вызывается машинно-зависимая процедура, которая, исходя из некоторых эвристик, принимает решение, какой нетерминал следует сопоставить вершине дерева.

Следует учитывать, что далеко не всегда вершину можно посчитать неявно (например, на способе адресации) — так в примере

```
II. (1) T := 4 * I
    (2) I = I + 1
        W = C + T
    (3) * W := D
```

вершину (1) надо посчитать явно, нельзя оставить ее в способе адресации. В общем случае чтобы быть уверенным, что счет вершины можно отложить, приходится обходить граф. Более того, откладывание счета вершины может привести к изменению зацепленности переменных (и тем самым повлиять на распределение регистров), что тоже приходится учитывать.

7. РАСПРЕДЕЛЕНИЕ РЕГИСТРОВ

Решение о том, находится переменная на регистре или нет, необходимо принимать еще до разметки DAG'a. Очевидно, что регистров может не хватить для размещения всех переменных; поэтому для их распределения используется следующий алгоритм:

1) Вначале считаем, что все переменные находятся в регистрах.
2) Размечаем DAG.
3) Используя результат разметки, распределяем регистры (а фактически вызываем подпрограммы генерации кода; они все делат «по-настоящему», только не выдают полученный код).
4) Если в какой-то момент нам не хватает регистров, то мы принимаем решение, что какая-то переменная будет находиться в памяти; при этом вся разметка, строго говоря, становится некорректной. Мы, тем не менее, продолжаем распределение регистров — сделано это для того, чтобы на каждом шаге итерации вытеснять в память как можно больше переменных.
5) Продолжаем выполнять последовательность «Разметка-Распределение регистров» пока не хватает регистров.
6) Окончательно генерируем код.

При выборе, какую переменную следует вытеснять в память, используется простейшая эвристика: мы считаем количество использований каждой переменной (разумеется, с учетом вложенности циклов), делим это число на количество линейных участков, в которых жива даная переменная, и вытесняем в память переменную с наименьшим таким коэффициентом. Разумеется, возможно использование любой другой эвристической функции.

В алгоритме существенно используется тот факт, что у современных процессоров общего назначения регистры более или менее симметричны.

8. ЗАКЛЮЧЕНИЕ

Все результаты разработки можно разделить на две группы: машинно-независимые и машинно-зависимые. К машинно-независимым относятся оптимизатор, интерфейс с первым проходом компилятора, язык описания архитектуры и компилятор с этого языка в Оберон; к машинно-зависмым — генератор кода для Intel x86. Для реализации этого генератора была написана грамматика и реализованы выходные интерфейсы (порождение объектных файлов и т.п.). Размер грамматики 1000 строк, 200 правил, 10 нетерминалов, по грамматике строится около 4000 строк на Обероне.

Качество генерируемого кода сопоставимо с (а иногда лучше чем) код, генерируемым коммерческими Скомпиляторами, и это при том, что сейчас нашему генератору несколько месяцов от роду; реализованы не все запланированные оптимизации, отсутствует щелевой оптимизатор.

Размер компилятора 50.000 строк, из них синтаксический анализ (front-end) 20.000, генератор (back-end) 30.000. В состав генератора кода входит

- интерфейс с первым проходом, в т.ч. перевод во внутреннее представление генератора — 8.500 строк
- оптимизатор — 10.000 строк (будет увеличиватся при добавлении оптимизаций)
- генератор кода для Intel x86 — 10.000 строк (из них 1000 — упомянутая выше грамматика)
- выходные интерфейсы — 1.500 строк

Таким образом, всего 20% компилятора существенно машинно-зависимо. Конечно, 10.000 строк — это тоже много; авторы надеются, что при перенастройке генератора на вторую архитектуру удастся использовать как минимум половину из них.

В данный момент готова бета версия компилятора для MS-DOS. Компилятор порождает стандартные объектные файлы, содержащие минимальную информацию для отладчика. В betf версию не включены линкер, отладчик и extender; кроме того, библиотеки существенно используют функции из библиотек языка C. Поэтому для использования этой версии необходимо иметь C компилятор, порождающий код для плоской модели памяти (flat memory model), например Symantec или Watcom. Идет работа по переносу комилятора в Win32, NT, Win/95, OS/2 и Linux. Бета версии для этих платформ должны быть готовы к концу лета.

На основании опыта разработки генератора для Intel x86 мы можем оценить трудоемкость перенацеливания генератора на другие архитектуры. Мы полагаем, что перенос на похожую архитектуру (Motorola 68k) будет сводится к модификации грамматики и выходного интерфейса и займет 2-4 человекомесяца. Перенос на первую RISC архитектуру, видимо, потребует модификации некоторых алгоритмов и займет чуть больше времени. После этого переносы на следующие RISC архитектуры будут также очень простыми.

В качестве следующей архитектуры нами выбран PowerPC. Работы по переносу на PowerPC, а именно разработка грамматики, будут начаты в течении месяца. Бета версии для этой платформы мы ожидаем к концу года.

ЛИТЕРАТУРА

[Нед95] Недоря А.Е. (1995) «Объектно-ориентированная разработка компиляторов» // Технология

программирования, Т.1, No.1.

[Пот92] Поттосин И.В. (1992) «Система СОКРАТ: окружение программирования для встроенных ЭВМ» // Новосибирск, Сибирское отделение РАН, Институт систем информатики, препринт No.11.

[ACK81] Allen F.E., Cocke J., Kennedy, K. (1981) «Reduction of operator strength» // In «Program Flow Analysis: Theory and Applications», ed. by Steven S. Muchnick and Neil D. Jones // Prentice-Hall, p.79-101.

[ASU86] Aho A.V., Sethi R., Ullman J.D. (1986) «Compilers: Principles, Techniques, and Tools» // Addison-Wesley.

[CFR91] Cytron R., Ferrante J., Rosen B.K., Wegman M.N., Zadek F.K. (1991) «Efficiently computing static single assignment form and the control dependence graph» // ACM Transactions on Programming Languages and Systems, Vol.13, No.4, p.451-490.

[Cre90] Crelier R. (1990) «OP2: A Portable Oberon Compiler» // ETH Zurich, Technical Report No.125.

[ESL89] Emmelmann H., Schroer F.-W., Landwehr R. (1989) «BEG – a Generator for Efficient Back Ends» // SIGPLAN Notices, Vol.24, No.7, p.227-237.

[FrH95] Fraser C., Hanson D. (1995) «A Retargetable C Compiler: Design and Implementation» // Benjamin Cummings.

[Hec77] Hecht M.S. (1977) «Flow analysis of computer programs» // Elsevier North-Holland.

[Knu71] Knuth D.E. (1971) «An Empirical Study or FORTRAN Programs» // Software – Practice and Experience, Vol.1, No.12, p.105-134.

[Sta92] Stallman R. (1992) «Using and Porting GNU C» // Free Software Foundation.

[Wol92] Wolfe M. (1992) «Beyond Induction Variables» // SIGPLAN Notices, Vol.27, No.7, p.162-174.