Тема 10. Генерация кода (из главы 8 в Ахо, 2-е изд.)

Генерация кода является последней фазой компиляции. Входом генератора кода является промежуточное представление исходной программы.

К генератору кода предъявляются жесткие требования. Получаемый код должен сохранять семантическое значение исходной программы и быть высококачественным, что означает эффективное использование доступных ресурсов целевой машины. Кроме того, эффективно должен работать и сам генератор кода.

Математически проблема генерации оптимальной целевой программы для данной исходной является неразрешимой; многие из подзадач, встречающихся при генерации кода, таких как распределение регистров, вычислительно трудноразрешимы. На практике используются эвристические методы, генерирующие хороший, но не обязательно оптимальный код. Эти эвристики достаточно стары и проверены, так что тщательно разработанный генератор может давать код в несколько раз более быстрый, чем получаемый о простейшего генератора без их применения.

Хорошие компиляторы перед началом генерации кода включают фазу оптимизации. Оптимизатор превращает одно промежуточное представление в другое, из которого может быть сгенерирован более эффективный код. В общем случае фазы оптимизации и генерации кода, известные как заключительная стадия компиляции, могут выполнять несколько проходов по промежуточному представлению перед генерацией целевой программы.

Генерация абсолютной программы на машинном языке имеет то преимущество, что она может быть помещена в фиксированное место в памяти и тут же выполнена. Программа может быть быстро скомпилирована и выполнена.

Генерация переносимой программы на машинном языке (обычно называют *объектным модулем*) обеспечивает возможность раздельной компиляции подпрограмм. Набор переместимых объектных модулей может быть скомпонован в одно целое и загружен для выполнения. Ценой дополнительных действий по компоновке и загрузке объектных модулей получается гибкое решение, которое допускает раздельную компиляцию подпрограмм и вызов из объектного модуля других ранее скомпилированных программ. Если целевая машина не обрабатывает перемещение автоматически, компилятор должен явно предоставить необходимую информацию загрузчику для компоновки отдельно скомпилированных модулей.

Несколько проще получение в качестве выхода генератора программы на языке ассемблера. При этом можно генерировать символьные команды и использовать возможности макросов ассемблера при генерации кода. Платой за эту простоту является дополнительный шаг ассемблирования по окончанию генерации кода.

Перед генератором кода стоят три основные задачи:

1) *выбор команд*, означает выбор машинных команд целевой машины для реализации инструкций промежуточного представления.

2) *распределение и назначение регистров*, означает принятие решения о том, какие значения в каких регистрах будут храниться.

3) *упорядочение команд*, предусматривает принятие решения о том, в каком порядке должны выполняться сгенерированные команды.

Детали реализации этих задач зависят от промежуточного представления, целевого языка и системы времени выполнения.

10.1. Выбор команд

Генератор кода должен отобразить программу в промежуточном представлении на последовательность машинных команд, которые могут быть выполнены целевой машиной. Сложность этого отображения определяется такими факторами, как

- уровень промежуточного представления;
- природа архитектуры набора команд;
- требуемое качество генерируемого кода.

Если использовать высокоуровневое промежуточное представление, то генератор кода может транслировать каждую инструкцию промежуточного представления в последовательность машинных команд с использованием шаблона кода. Однако генерация кода инструкция за инструкцией часто дает низкокачественный код, требующий дальнейшей оптимизации. Если промежуточное представление отражает некоторые низкоуровневые особенности машины, то генератор кода может использовать эту информацию для генерации более эффективной последовательности команд.

Природа набора команд целевой машины сильно влияет на сложность выбора инструкций. Например, важными факторами являются единообразие и полнота набора команд. Если целевая машина не поддерживает единообразно все типы данных, то каждое исключение из общего правила требует отдельной обработки. На некоторых машинах, например, операции с числами с плавающей точкой выполняются с использованием отдельных регистров.

Другим важным фактором является скорость выполнения команд. Если не беспокоиться об эффективности целевой программы, выбор инструкций достаточно прост. Для каждого типа трехадресных инструкций можно разработать шаблон целевого кода, генерируемого для данной конструкции. Например, каждая трехадресная инструкция вида $x := y + z$, где x, y и z распределяются статически, может быть транслирована в следующий код:

```
LD      R0, y          //загрузка y в регистр R0
ADD     R0, R0, z      //прибавление z к R0
ST      x, R0          //сохранение R0 в x
```

Такая стратегия часто приводит к избыточным сохранениям и загрузкам. Например, последовательность трехадресных инструкций

$$a := b + c$$
$$d := a + e$$

будет транслирована в

```
LD      R0, b          //загрузка b в регистр R0
ADD     R0, R0, c      //прибавление c к R0
ST      a, R0          //сохранение R0 в a
LD      R0, a          //загрузка a в регистр R0
ADD     R0, R0, e      //прибавление e к R0
ST      d, R0          //сохранение R0 в d
```

Четвертая команда совершенно излишня, поскольку она загружает значение, которое только что было сохранено; если в дальнейшем *a* не используется, то излишня и третья команда.

Качество сгенерированного кода обычно определяется его скоростью выполнения и размером. На большинстве машин заданная программа в промежуточном представлении может быть реализована множеством различных последовательностей кодов, существенно отличающихся друг от друга. Непосредственная простейшая трансляция промежуточного кода может давать корректный, но неприемлемо неэффективный код.

Например, если целевая машина имеет команду инкремента (INC), то трехадресная инструкция $a := a + 1$ может быть эффективно реализована одной командой `INC a` вместо обычной последовательности, состоящей из загрузки *a* в регистр, прибавления к регистру 1 и сохранения результата в *a*:

```
LD      R0, a           //загрузка a в регистр R0
ADD     R0, R0, #1      //прибавление 1 к R0
ST      a, R0           //сохранение R0 в a
```

Для получения эффективной последовательности команд необходимо знать стоимость каждой команды; к сожалению, получить точную информацию зачастую весьма сложно. Принятие решения о том, какая именно последовательность машинных команд наилучшим образом подходит для данной трехадресной конструкции, может потребовать также знания контекста, в котором появляется эта конструкция.

В разделе 8.9 Ахо показано, что такой выбор может быть смоделирован с использованием представления машинных команд и инструкций промежуточного представления в виде деревьев. Затем делается попытка покрытия дерева промежуточного представления множеством поддеревьев, соответствующим машинным командам. Если каждому поддереву машинной команды назначить стоимость, то для генерации оптимальной последовательности команд можно использовать динамическое программирование.

10.2. Распределение регистров

Ключевой проблемой при генерации кода является принятие решения о том, какие значения в каких регистрах должны храниться. Обычно регистров слишком мало, чтобы хранить все значения. Значения, которые не хранятся в регистрах, должны находиться в памяти. Команды, использующие в качестве операндов регистры, обычно короче и выполняются быстрее, чем команды, работающие с операндами, расположенными в памяти. Следовательно, эффективное использование регистров – еще одна важная составляющая генерации хорошего целевого кода.

Использование регистров часто разделяется на две подзадачи:

1. В процессе *распределения регистров* (register allocation) выбирается множество переменных, которые будут находиться в регистрах в каждой точке программы.
2. В последующей фазе *назначения регистров* (register assignment) выбираются конкретные регистры для размещения в них переменных.

Поиск оптимального назначения регистров переменным представляет собой сложную задачу даже на машине с единственным регистром. Математически эта задача – NP-полная. Проблема усложняется еще и тем, что аппаратное обеспечение и/или операционная система целевой машины может накладывать дополнительные ограничения на использование регистров.

Пример. Некоторые машины требуют для некоторых операндов и результатов операций пар регистров (регистра с четным номером и регистра со следующим за ним нечетным номером). Например, на некоторых машинах целочисленные умножение и деление требуют использования пар регистров. Команда умножения, например, имеет вид

```
M x, y
```

Здесь сомножитель *x* представляет собой нечетный регистр пары, состоящей из четного и нечетного регистров, а сомножитель *y* может храниться в любом регистре. Произведение занимает пару из четного и нечетного регистров. Команда деления имеет вид

```
D x, y
```

Здесь делимое занимает пару регистров, четный регистр которой – *x*; *y* представляет собой делитель. После деления в четном регистре хранится остаток, а в нечетном – частное.

Рассмотрим две последовательности трехадресных кодов (отличаются вторые инструкции):

$t := a + b$	$t := a + b$
$t := t * c$	$t := t + c$
$t := t / d$	$t := t / d$

Кратчайшие последовательности ассемблерных кодов:

```
L  R1, a          L    R0, a
A  R1, b          A    R0, b
M  R0, c          A    R0, c
D  R0, d          SRDA R0, 32
ST R1, t          D    R0, d
                  ST   R1, t
```

Команда `SRDA R0, 32` сдвигает делимое в R1 и очищает R0, делая все биты равными его знаковому биту. Команды L, ST и A означают соответственно загрузку в регистр, сохранение регистра и суммирование. Следует заметить, что оптимальный выбор регистра для загрузки *a* зависит от того, что в конечном счете произойдет с *t*.

Стратегии распределения регистров рассматриваются в разделе 8.8 Ахо. В разделе 8.10 показано, что для некоторых классов машин можно построить последовательность кодов, которые будут выполнять вычисление выражения с использованием минимально возможного количества регистров.

10.3. Порядок вычислений

Порядок, в котором выполняются вычисления, также может существенно влиять на эффективность целевого кода. Изменение порядка вычислений может привести к уменьшению количества регистров, необходимых для хранения промежуточных результатов. Однако в общем случае выбор оптимального порядка вычислений представляет собой сложную NP-полную задачу.