Классическая теория компиляторов

Лекция 6

ОБ ОПЕРАТОРАХ И ВЫРАЖЕНИЯХ

Базовые синтаксические категории:

- программа
- оператор
- выражение

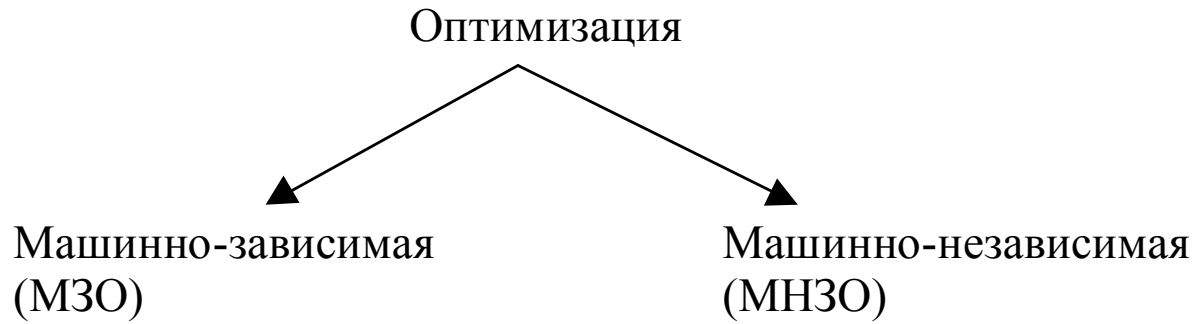
Например, в языке Си выражения считаются операторами

```
void main(void)
{
  int a, b;
  3-4*b;
  1+2;
  for(2+a;1;)
    b+1;
}
```

Что такое пустой оператор?

Как сделать так, чтоб была возможность ставить или не ставить разделитель операторов (‘;’) перед закрывающей операторной скобкой?

ОПТИМИЗАЦИЯ ПРОГРАММ



Оптимизации подлежат:

- время выполнения;
- емкостные ресурсы (память).

МЗО связана с типом генерируемых команд и включается в фазу генерации кода (т.е. оптимизации подлежат машинные коды). МЗО напрямую зависит от архитектуры вычислительной машины.

МНЗО – отдельная фаза компилятора, предшествующая генерации кода. Она включается на этапе генерации промежуточного кода – внутренней формы представления программы.

Исключение общих подвыражений

Оптимизация линейных участков

Процедура, позволяющая не рассчитывать одно и то же выражение несколько раз, исключая общие подвыражения:

- представить выражение в форме, пригодной для обнаружения общих подвыражений;
- определить эквивалентность двух и более подвыражений;
- исключить повторяющиеся;
- изменить команды так, чтобы учесть это исключение.

Пример: A = c*d*(d*c+b) * c d T1
* d c T2
+ T2 b T3
* T1 T3 T4
= A T4

1. Упорядочим операнды:
   1) * c d T1
   2) * c d T2
   3) + T2 b T3
   4) * T1 T3 T4
   5) = A T4

2. Определим границы операторов и найдем общие подвыражения (это (1) и (2)) и затем исключим подвыражение (2). После чего заменим далее везде T2 на T1:
   * c d T1
   + T1 b T3
   * T1 T3 T4
   = A T4

Опасности: побочные эффекты, затраты на поиск и анализ

Примеры

Пример:

```
if( a[y*3] < 0 || b[y*3] > 10)
        a[y*3] = 0;
```

Выражения "y*3" и "a[y*3]" являются общими подвыражениями.

```
T1 = y*3;
A1 = &a[T1];
A2 = &b[T1];
if( *A1 < 0 || *A2 > 10)
    *A1 = 0;
```

Пример:

```
if(a == 0)
    a = y * 3;
else
    b = y * 3;
```

приводит к логическому эквиваленту:

```
T1 = y * 3;
if(a == 0)
    a = T1;
else
    b = T1;
```

Вычисления на этапе компиляции

Если в программе есть участки, в которых присутствуют подвыражения, состоящие из **констант**, то эти подвыражения можно просчитать на этапе компиляции.

Метод "свертки констант" (константная арифметика)

Например:

A := 1.5 * 2/3; => A := 1;

Но:

A := 3*b/4/d*2 ?

```
#define TWO 2
a = 1 + TWO;
```

=>

```
a = 3;
```

"Алгебраические упрощения" (вид свертки констант, который удаляет арифметические тождества)

```
x := y + 0;     => x := y;
x := y * 0;     => x := 0;
x := y / 1.0;   => x := y;
x := y / 0;     => x :???;
```

Размножение констант

Любая ссылка на константное значение замещается самим значением:

```
x = 2;
if( a < x && b < x)
    c = x;
```

=>

```
x = 2;
if(a < 2 && b < 2)
    c = 2;
```

Оптимизация булевых выражений

Использование свойств булевых выражений.

Например, вместо

if (a and b and c) then <операторы> endif

надо сгенерировать команды таким образом, чтобы исключались лишние проверки:

```
if not a then goto Label
if not b then goto Label
if not c then goto Label
<операторы>
Label:                                  //метка перехода
```

Проблемы:

- могут проявиться побочные эффекты в тех случаях, когда аргументами являются функции
- "оптимизированный" код может получиться более громоздким по сравнению с оригиналом.

Оптимизация циклов

Вынесение инвариантов за пределы цикла

Причины, которые могут привести к уменьшению скорости работы программы в циклах

- Итерации цикла зависимы и не могут исполняться параллельно.
- Тело цикла большое и требуется слишком много регистров.
- Тело цикла или количество итераций мало и выгоднее совсем отказаться от использования цикла.
- Цикл содержит вызовы функций и процедур из сторонних библиотек.
- Цикл интенсивно использует какое-то одно исполняющее устройство процессора.
- В цикле имеются условные переходы.

Вынесение инвариантных вычислений за пределы цикла

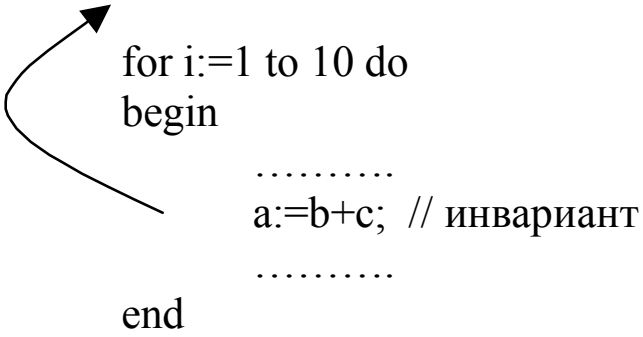
Один из наиболее эффективных методов оптимизации, дающий весьма ощутимые результаты.

Для реализации метода необходимо:

- распознать инвариант;
- определить место переноса;
- перенести инвариант.

Неудобства метода:

- отследить инвариант нелегко, т.к. аргументы могут косвенно зависеть от переменной цикла;
- не учитываются побочные эффекты, если аргументы инварианта являются функциями (или зависимыми от них).



Подсчет единиц

```
// Медленная, но простая процедура
// подсчета единиц
R00 = R11;
  for(i=0;i<8;i++)
  {
    if(e0&(1<<i)) R00++;
    if(e1&(1<<i)) R11++;
}
```

```
// Быстрая, но сложная процедура подсчета единиц

  Mask=0b10101010;
  r0=((R00&Mask)>>1);
  r1=((R11&Mask)>>1);

  Mask=0b01010101;
  R00 &= Mask;
  R11 &= Mask;

  R00 += r0;
  R11 += r1;
```

```
Mask=0b11001100;
r0=((R00&Mask)>>2);
r1=((R11&Mask)>>2);

Mask=0b00110011;
R00 &= Mask;
R11 &= Mask;

R00 += r0;
R11 += r1;

Mask=0b11110000;
r0=((R00&Mask)>>4);
r1=((R11&Mask)>>4);

Mask=0b00001111;
R00 &= Mask;
R11 &= Mask;
R00 += r0;
R11 += r1;
```

Развертка циклов

- Такая оптимизация выполняется, когда тело цикла мало.
- Многократно дублируют тело цикла в зависимости от количества исполняющих устройств
- Эта оптимизация может вызвать зависимость по данным => вводятся дополнительные переменные.

До	После-1	После-2

`for(i=0; i < iN; i++)`
`{`
`  res *= a[i];`
`}`
	`for(i=0; i < iN; i+=3)`
	`{`
	`  res *= a[i];`
	`  res *= a[i+1];`
	`  res *= a[i+2];`
	`}`
		`for(i=0; i < iN; i+=3)`
		`{`
		`  res1 *= a[i];`
		`  res2 *= a[i+1];`
		`  res3 *= a[i+2];`
		`}`
		`res = res1 * res2 * res3;`

Объединение циклов

В цикле могут быть долго выполняющиеся инструкции (например, извлечение корней, логарифмы...).

Или есть несколько циклов, которые выполняются по одинаковому интервалу индексов.

Целесообразно объединить циклы для более сбалансированной нагрузки исполняющих устройств.

До

```
for(i = 0; i < iN; i++)
{
  a[i] = b[i] - 5;
}
for(i = 0; i < iN-1; i++)
{
  d[i] = e[i] * 3;
}
```

После

```
for(i = 0; i < iN-1; i++)
{
  a[i] = b[i] - 5;
  d[i] = e[i] * 3;
}
a[iN-1] = b[iN-1] - 5;
```

Дополнительно

Что эффективнее: ++x или x++?

Для пользовательских типов ++x лучше, т.к. постинкремент сопровождается созданием копии объекта, хранящего старое состояние, и возвратом его по значению.

for(i=0;i<10;**i++**)

for(i=0;i<10;**++i**)

a = b/4

a = b>>2

Проблемы, связанные с оптимизацией

- Необходимо сопоставлять ожидаемый выигрыш с дополнительными накладными расходами
- Возможное ухудшение кода, сложность трассировки оптимизированных программ.
- Сложность выявления возникающих "побочных" эффектов.

Например

A := pop();

B := pop();

C := A || B;

push(C);

"Компактный" вариант

push(pop() || pop()),

Может быть некорректно оптимизирован в *push(pop()*, т.к. A || A ≡ A).

Выводы

- Необходимо сопоставлять затраты на оптимизацию с ожидаемым эффектом.
- Оптимизация не всегда приводит к улучшению кода – могут появиться побочные эффекты. Либо после оптимизации получается более громоздкий код.
- Чем больше вычислительной работы перекладывается на этап компиляции, тем эффективнее будет выполняться программа.
- Более предпочтительной для МНЗО является внутренняя форма представления программы в виде тетрад.
- Лучший метод оптимизации – писать хорошие (грамотные) программы.