

11. Оптимизация

- Задачи оптимизации
- Виды оптимизирующих преобразований
- Представления программы, используемые в оптимизирующих преобразованиях
- Примеры оптимизирующих преобразований

Лекция 11. Оптимизация

В этой лекции рассматриваются следующие вопросы:

- Задачи оптимизации
- Виды оптимизирующих преобразований
- Представления программы, используемые в оптимизирующих преобразованиях
- Примеры оптимизирующих преобразований

Оптимизация

Оптимизирующее преобразование:

- эквивалентное или почти эквивалентное преобразование программы, уменьшающее ее стоимость

Применение оптимизаций:

- улучшение качества кода
- выявление неочевидных ошибок

Оптимизация

Под оптимизацией понимают последовательность эквивалентных преобразований исходной программы, уменьшающих ее стоимость. Как набор, так и порядок выполнения этих преобразований зависят от того, что считается стоимостью программы. В качестве такой стоимости могут выступать, например, среднее время работы, объем кода и т.д. Эффективность оптимизации также зависит от отношения эквивалентности и от размера участка экономии, на котором эта оптимизация проводится (обычно оптимизированной программе разрешается иметь большую область определения, чем исходной). За счет оптимизации невозможно добиться существенного улучшения *алгоритма* программы, можно только говорить об улучшении *реализации* этого алгоритма. В удачных случаях оптимизация может ускорить программу в несколько раз. Полезность применения оптимизации обусловлена следующими причинами:

- Распространением языков сверхвысокого уровня, языков спецификации и систем проектирования. Как правило, подобные системы порождают программы на некотором языке высокого уровня. В этом случае оптимизация может нейтрализовать избыточность такого порождения, приблизив качество сгенерированной программы к ручному программированию.
- Необходимостью поддержки и сопровождения готовой программы. Такие требования зачастую приводят к тому, что программисты используют не самые эффективные решения в целях улучшения наглядности или легкости сопровождения программ (очень часто соображения эффективности и сопровождаемости противоречат друг другу). В то же время недостатки эффективности могут быть исправлены оптимизатором при генерации окончательной программы.
- Усилением контроля семантических ошибок. Такие ошибки требуют специального анализа исходной программы, который может являться побочным результатом действий оптимизатора.

Виды оптимизации

По уровню представления программы:

- на уровне исходного языка
- машинно-независимая (на уровне машинно-независимого промежуточного представления)
- машинно-зависимая (на уровне машинного языка)

По размеру фрагмента оптимизации

- локальная (линейный участок, группа операторов)
- квазилокальная (фрагменты выбранного вида)
- глобальная (процедура целиком)
- межпроцедурная

Виды оптимизации

Существует много различных классификаций оптимизирующих преобразований. Здесь мы рассмотрим классификации по уровню представления программы и по размеру участка экономии.

В зависимости от уровня представления программы различают следующие виды оптимизации:

- Оптимизацию на уровне исходного языка. При этом в результате трансформации получается программа, записанная в том же самом языке.
- Машинно-независимую оптимизацию. В этом случае преобразованию подвергается программа на уровне машинно-независимого промежуточного представления, общего для группы входных или машинных языков.
- Машинно-зависимую оптимизацию, то есть оптимизацию на уровне машинного языка.

С точки зрения эффективности наиболее предпочтительной является машинно-зависимая оптимизация, поскольку именно с ее помощью можно учесть особенности конкретной вычислительной среды, однако машинно-зависимый оптимизатор непереносим. С другой стороны, преобразование программы на уровне исходного языка позволяет получить более эффективную программу, допускающую дальнейшее развитие и сопровождение. Наконец, машинно-независимая оптимизация на уровне промежуточного представления является компромиссом между этими двумя крайними случаями.

Другим важным для качества оптимизации соображением является также размер участка экономии, то есть того фрагмента программы, в рамках которого производится оптимизирующее преобразование. Чем больше участок экономии, тем больше информации о свойствах программы доступно оптимизатору. Классификация оптимизации относительно участка экономии приведена на слайде.

Зависимость между оптимизациями

Виды зависимости:

- независимость (преобразования T_1 и T_2 не влияют на применимость друг друга к программе)
- повторность (преобразование T_1 открывает дополнительные возможности для проведения преобразования T_2)
- тупиковость (преобразование T_1 делает невозможным применение преобразования T_2)

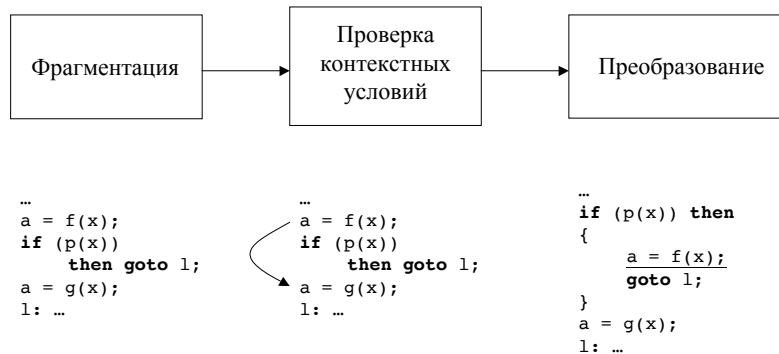
Зависимость между оптимизациями

Как правило, эффект оптимизации достигается применением серии разнородных оптимизирующих преобразований. Здесь мы рассмотрим возможные зависимости между ними и их влияние на результаты друг друга.

С одной стороны интуитивно понятно, что конкретное оптимизирующее преобразование применимо не к произвольной программе (например, никакое оптимизирующее преобразование не применимо к оптимальной программе). С другой стороны, последовательность оптимизирующих преобразований, применяемых к программе, можно трактовать как серию изолированных преобразований, применяемых к *разным* программам (а именно, к исходно программе, к программе после первого преобразования, к программе после второго преобразования и т.д.). Отсюда возникают следующие возможные зависимости между оптимизирующими преобразованиями.

- Независимость. Преобразования T_1 и T_2 называются независимыми, если применение одного из них к программе не влияет на применимость другого.
- Повторность. Преобразование T_1 повторно по отношению к преобразованию T_2 , если для произвольной программы p из применимости к ней T_2 следует применимость его к $T_1(p)$.
- Тупиковость. Преобразование T_1 тупиково по отношению к преобразованию T_2 , если для произвольной программы p T_2 неприменимо к $T_1(p)$.

Стадии оптимизации



Стадии оптимизации

Проведение оптимизирующего преобразования обычно осуществляется в несколько стадий. Рассмотрим каждую из них на конкретном примере.

- **Фрагментация.** Под фрагментацией понимается выделение некоторого участка программы, к которому может быть применено преобразование. В данном случае таким участком является последовательность операторов, один из которых – условный оператор без части *else*. Задачу фрагментации решает *анализ потока управления*. Заметим, что различные оптимизирующие преобразования требуют выделения различных фрагментов исходной программы, так что количество задач фрагментации достаточно велико. Некоторые алгоритмы фрагментации подробнее рассмотрены в лекции 12.
- **Проверка контекстных условий,** то есть выяснение применимости оптимизирующего преобразования к данному фрагменту. В данном случае необходимо проверить, что значение переменной *a* после первого присваивания фрагмента не используется в случае ложности условия *p(x)*. Задачи такого рода решаются с применением *анализа потоков данных*. Примеры задач анализа потоков данных и описание общего подхода к их решению приведены в лекции 13.
- **Преобразование.** Собственно применение оптимизации к выбранному фрагменту. В данном случае первое присваивание "втянуто" в *then*-часть условного оператора. Таким образом, в случае ложности условия *p(x)* программа выполняет на один оператор меньше.

Покадровая оптимизация

```
1 { .L1:      jmp      .L1      }
2 {          movl     %edx, %eax }
3 {          movl     %edx, %eax }
4 {          movl     %eax, %ecx }
5 {          movl     %ecx, %edx }
   {          movzbl   (%eax), %ecx }
   {          movl     $0, %eax }
   {          je       .L2      }
   {          movzbl   (%ecx), %edx }
   {          cmpb     %dl, (%esi) }
   {          jmp      .L4      }
.L2: {          jmp      .L3      }

→

.L1:  addl     %edx, %eax
      movl     %eax, %edx
      movzbl   (%eax), %ecx
      xorl     %eax, %eax
      je       .L3
      movzbl   (%ecx), %edx
      cmpb     %dl, (%esi)
      jmp      .L4
```

Покадровая оптимизация

Одним из самых простых видов оптимизаций является так называемая *покадровая оптимизация*, которая обычно применяется на уровне машинно-зависимого представления программы.

Суть покадровой оптимизации – поиск сравнительно короткой последовательности инструкций, удовлетворяющей определенным контекстным условиям, и замена такой последовательности на более эффективную.

Для реализации этого подхода используют понятие "окна" (или "кадра", *frame*), которое передвигается по программе. В каждый момент времени содержимое кадра исследуется на предмет соответствия тому или иному образцу и в случае совпадения производится преобразование.

На слайде приведен пример работы покадрового оптимизатора. В данном случае были распознаны следующие образцы:

- Переход на следующую инструкцию (обозначен номером 1).
- Две одинаковые инструкции пересылки (номер 2).
- Пересылка данных через вспомогательный регистр, значение которого далее не используется (номер 3).
- Засылка нуля в регистр (номер 4).
- Переход на переход (номер 5).

Результат применения покадровой оптимизации является программа, показанная на слайде справа.

Обозначения

Ориентированный граф $G=(V, E)$:

1. $V=\{v_1, v_2, \dots, v_k\}$ - множество вершин
2. $E=\{(v, w) \mid v \in V, w \in V\}$ - множество дуг

Обозначения:

1. $e=(v, w) \in E \Rightarrow \text{beg}(e)=v, \text{end}(e)=w$
2. $v \in V \Rightarrow \text{in}(v)=\{e \in E \mid \text{end}(e)=v\}$
3. $v \in V \Rightarrow \text{out}(v)=\{e \in E \mid \text{beg}(e)=v\}$

Обозначения

Для описания способов решения задач оптимизации, нам потребуются следующие обозначения.

Ориентированным графом назовем пару конечных множеств (V, E) , называемых соответственно множествами вершин и дуг, при этом множество дуг представляет собой совокупность пар вершин.

Для произвольной дуги $e=(v, w)$ вершину v назовем началом дуги e (обозначение $\text{beg}(e)$), вершину w – концом дуги e (обозначение $\text{end}(e)$).

Для произвольной вершины v обозначим через $\text{in}(v)$ множество входящих дуг, через $\text{out}(v)$ – множество исходящих дуг.

Представление программы

Определение:

$G=(V, E, start, stop)$ - граф потока управления \Leftrightarrow

1. (V, E) - ориентированный граф
2. $start \in G.V, stop \in G.V$
3. $|in(start)|=|out(stop)|=\emptyset$
4. $\forall v \in G.V \text{ } start \rightarrow^* v \rightarrow^* stop$

Разметка:

$\sigma : G.V \rightarrow \Omega$, где

$\Omega = \{\omega_1, \omega_2, \dots, \omega_k\}$ - алфавит операторов

Программа - это пара (G, σ)

Представление программы

Программу для оптимизации удобно представлять в виде размеченного графа потока управления.

Граф потока управления – это ориентированный граф, в котором выделены две вершины $start$ и $stop$, и удовлетворяющий следующим требованиям:

- В $start$ не входит ни одна дуга.
- Из $stop$ не выходит ни одна дуга.
- Любая вершина достижима из $start$.
- Из любой вершины достижима вершина $stop$.

Введем далее в рассмотрение некоторое конечное множество, которое мы будем называть алфавитом операторов Ω и назовем разметкой графа потока управления произвольное отображение σ множества вершин графа в множество операторов.

Пара (G, σ) , где G – граф потока управления, а σ – его разметка, и является представлением программы для оптимизации. От дополнительных свойств элементов множества операторов зависит характер оптимизаций, которые можно выразить, используя данное представление.

Заметим, что построение такого представления программы уже включает в себя широко известное оптимизирующее преобразование – удаление недостижимого кода, то есть той части программы, на которую никогда не передается управление.

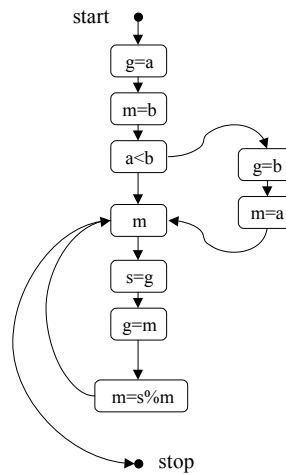
Пример

```
int F (int a, int b)
{
  int g = a, m = b;

  if (a < b) {g = b; m = a;}

  while (m)
  {
    int s = g;
    g = m;
    m = s % m;
  }

  return g;
}
```



Пример

На слайде приведен пример программы (слева) и ее представления в виде размеченного графа управления (справа). В качестве алфавита операторов выступает множество, состоящее из изображений элементарных операторов данной программы, то есть множество

$$\{ 'g=a', 'm=b', 'a<b', 'm', 's=g', 'g=m', 'g=b', 'm=a', 'm=s\%m' \}$$

Далее мы рассмотрим несколько полезных вариантов определения множества операторов, применимых для формализации основных оптимизирующих преобразований.

Поток управления

Алфавит операторов:

$$\Omega = \{\emptyset, \omega_1, \dots, \omega_k\}$$

$$\forall v \in G. V \quad \sigma(v) = \emptyset \Rightarrow |out(v)| \leq 1$$

Эквивалентность слов:

$$\forall w_1, w_2 \in \Omega^* \quad w_1 \sim w_2 \Leftrightarrow$$

1. $w_1 = \omega_i \ \& \ w_2 = \omega_i$ либо
2. $w_1 = u_1 \emptyset v_1, w_2 = u_2 \emptyset v_2$, где $u_1 \sim u_2 \ \& \ v_1 \sim v_2$

Слово, соответствующее пути:

$$p = (v_1, v_2, \dots, v_n), \quad \forall i \in [1, n-1] \quad (v_i, v_{i+1}) \in G.E$$

$$w_\sigma(p) = \sigma(v_1) \sigma(v_2) \dots \sigma(v_n)$$

Эквивалентность программ:

$$(G_1, \sigma_1) \sim (G_2, \sigma_2) \Leftrightarrow$$

$$\exists m : G_1.V \rightarrow G_2.V, \text{ такое, что}$$

$$\forall v \in G_1.V \quad \forall p = (v, \dots) \in G_1 \quad \exists p' = (m(v), \dots) \in G_2 : w_{\sigma_1}(p) \sim w_{\sigma_2}(p')$$

Поток управления

Данный способ описания программы применяется тогда, когда необходимо описать преобразования потока управления, которые не требуют анализа потоков данных.

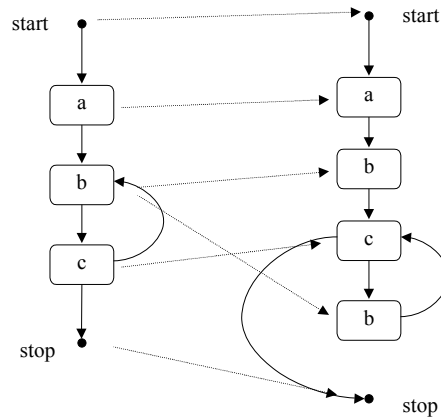
Алфавит операторов в этом случае содержит выделенный пустой оператор (\emptyset). Будем требовать, чтобы разметка графа потока управления сопоставляла пустой оператор только тем вершинам, для которых число исходящих дуг не превосходит единицы. Два слова в алфавите операторов объявляются эквивалентными в том и только том случае, когда они равны с точностью до вхождения пустых операторов.

Для определения эквивалентности программ, записанных в данном формализме, введем понятие слова, соответствующего пути в графе потока управления, а именно, для произвольного пути p слово $w(p)$ в алфавите операторов получается последовательным выписыванием символов, которыми помечены вершины p .

Будем говорить, что две программы (G_1, σ_1) и (G_2, σ_2) эквивалентны, тогда и только тогда, когда существует такое отображение m между вершинами G_1 и G_2 , что для произвольной вершины v графа G_1 и произвольного пути p , начинающегося в ней, найдется путь p' в графе G_2 , начинающийся в вершине $m(v)$, для которого слова $w(p)$ и $w(p')$ эквивалентны.

Пример

$$\Omega = \{\emptyset, a, b, c\}$$



Пример

На слайде приведен пример преобразования потока управления к форме, эквивалентной в описанном выше смысле (а именно, преобразование цикла с постусловием в цикл с предусловием). Пунктирными стрелками указаны образы вершин при отображении m .

Нетрудно заметить, что данное отображение необратимо, и, следовательно, определяемое им отношение несимметрично. Таким образом, введенное нами отношение, строго говоря, не является отношением эквивалентности. Это несоответствие может быть устранено, однако для наших целей этого не потребуется.

Def-Use Chains

Операторы:

$\Omega = \{\omega_1, \omega_2, \dots, \omega_n\}$ - алфавит операторов

$I = \{i_1, i_2, \dots, i_k\}$ - алфавит входов

$O = \{o_1, o_2, \dots, o_m\}$ - алфавит выходов

$input : \Omega \rightarrow 2^I$ - входы операторов,

$output : \Omega \rightarrow 2^O$ - выходы операторов

$\forall \omega_1, \omega_2 \in \Omega \quad \omega_1 \neq \omega_2 \Rightarrow$

$input(\omega_1) \cap input(\omega_2) = \emptyset \ \& \ output(\omega_1) \cap output(\omega_2) = \emptyset$

Def-Use разметка:

$DU : O \rightarrow 2^I$

$\forall \omega \in \Omega$

$def(\omega) = output(\omega)$

$use(\omega) = \bigcup_{i \in input(\omega)} \{o \in O \mid i \in DU(o)\}$

Def-Use Chains

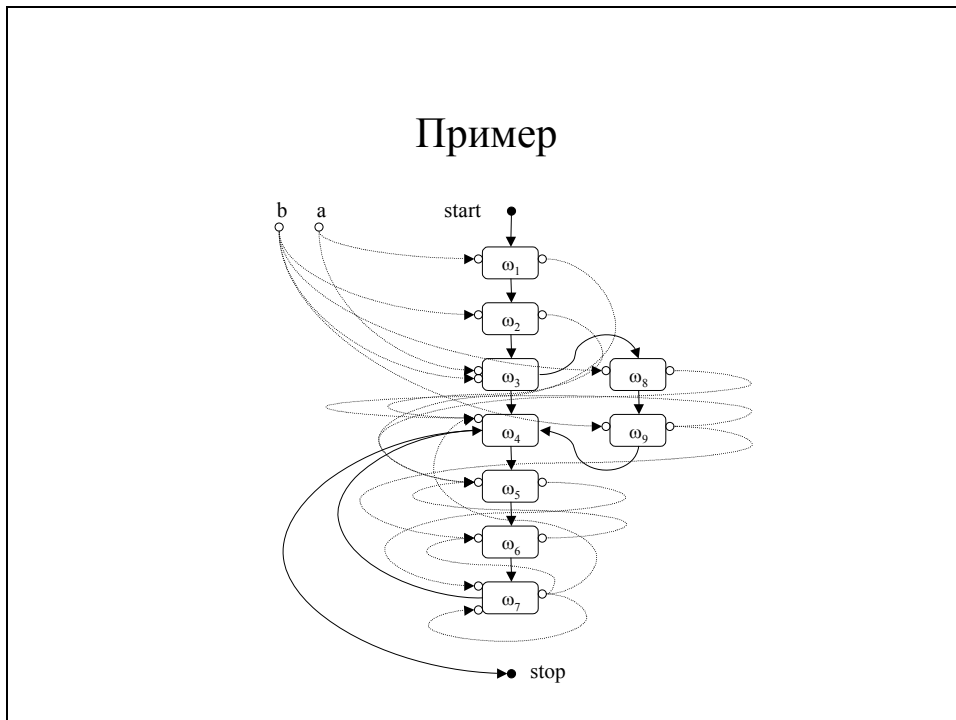
Следующей удобной формой представления программ для оптимизации является представление с использованием def-use chains.

Помимо алфавита операторов введем в рассмотрение алфавиты входов I и выходов O и определим для каждого оператора ω множество его входов $input(\omega)$ и выходов $output(\omega)$. При этом будем требовать, чтобы множества входов и выходов для разных операторов не пересекались.

Рассмотрим теперь отображение DU , которое каждому выходу сопоставляет некоторое подмножество множества входов. Тогда для каждого оператора определяются множества $def(\omega)$ и $use(\omega)$, которые есть соответственно множества его выходов и множество тех выходов других операторов, образы которых при отображении DU содержат хотя бы один вход оператора ω .

Неформально говоря, данное представление позволяет отследить потоки данных в программе без учета того, как именно эти данные преобразуются. Множества входов и выходов соответствуют интуитивному представлению об аргументах и результатах операторов, а отображение DU просто описывает, как используются выработанные операторами результаты.

Пример



Пример

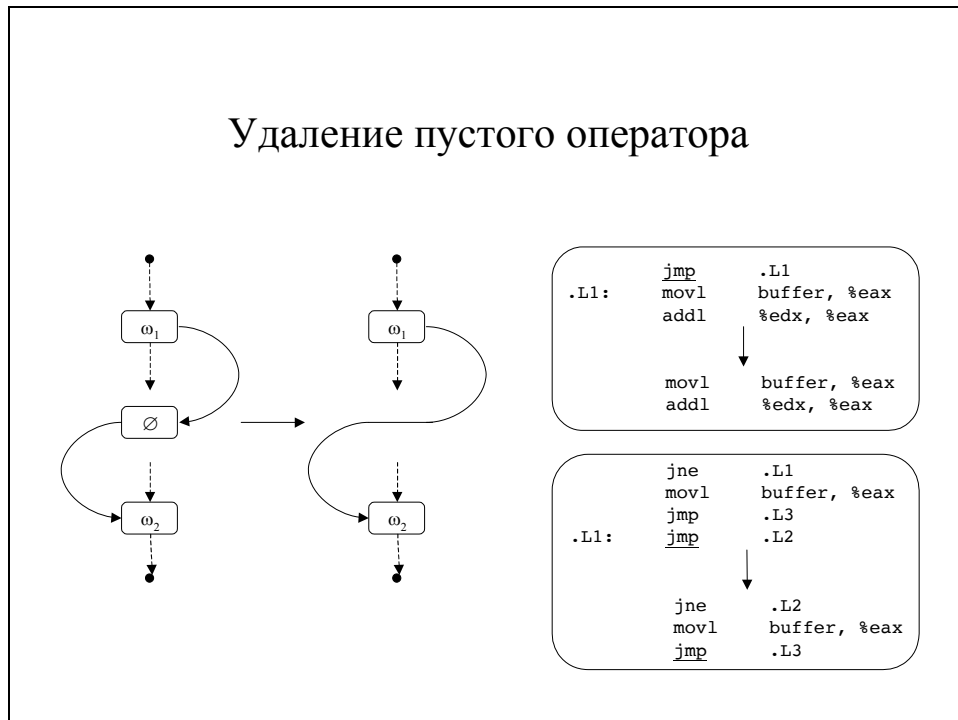
Рассмотрим представление программы, приведенной на иллюстрации 8, с использованием def-use chains.

Здесь входы операторов обозначены с помощью кружков, расположенных слева от операторов, выходы — с помощью кружков, расположенных справа. Пунктирные стрелки ведут из выходов одних операторов к входам других и обозначают образы при отображении *DU*.

Построение данного представления опирается на решение одной из задач из области анализа потоков данных, а именно — задачи о достижимых определениях. Подробно данная задача рассмотрена в лекции 13.

Далее мы рассмотрим примеры оптимизирующих преобразований и те представления, которые необходимо использовать для их проведения.

Удаление пустого оператора



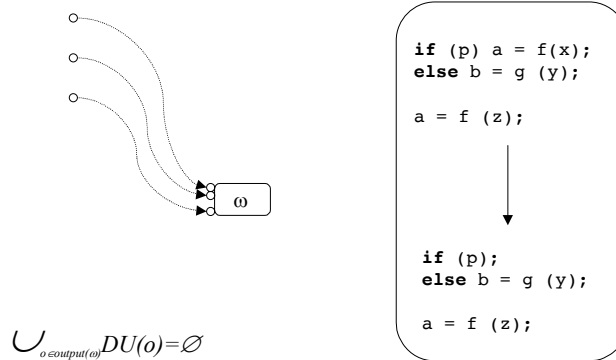
Удаление пустого оператора

Удаление пустого оператора – это одно из наиболее простых и распространенных оптимизирующих преобразований, которое реализуется практически во всех, даже не оптимизирующих, компиляторах. Для его формализации требуется представление программы в виде графа потока управления с пустым оператором.

Данное преобразование заключается в том, что вершина графа, помеченная пустым оператором, удаляется вместе с входящими и исходящими дугами, а в оставшийся граф добавляются дуги, соединяющие предшественников удаленного оператора с его потомком (если он есть). Напомним, что в выбранном нами представлении у вершины графа, помеченной пустым оператором, может быть не более одного потомка.

Несмотря на кажущуюся простоту данного преобразования, им покрывается большое количество частных случаев. В качестве примеров его применения справа на иллюстрации приведены удаление перехода на следующую инструкцию и удаление перехода на переход.

Удаление мертвого кода



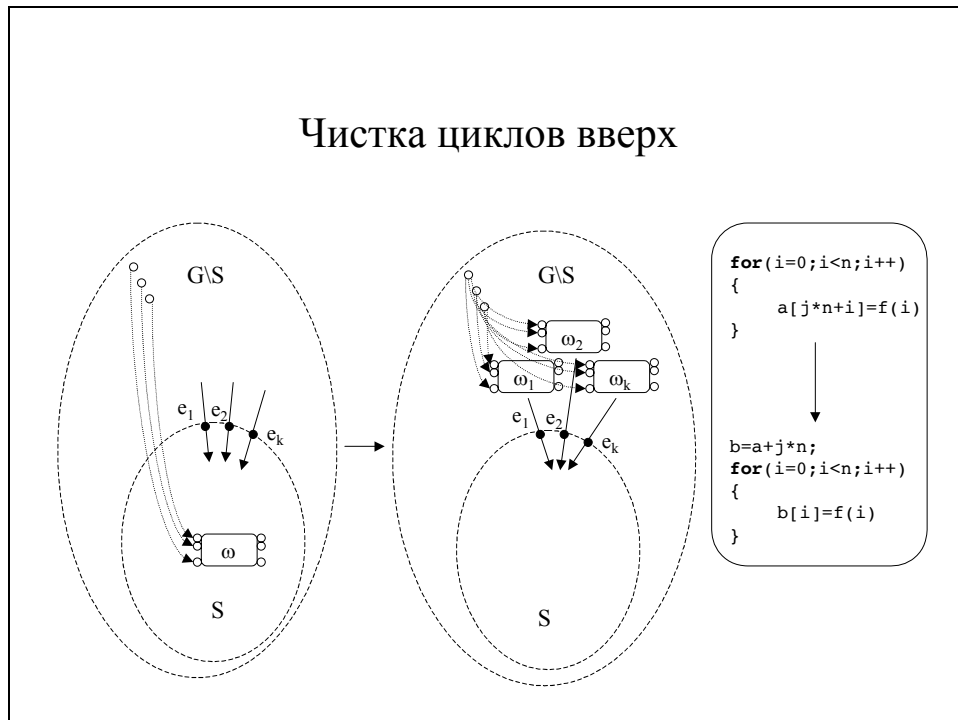
Удаление мертвого кода

Данное преобразование легко реализуется, если программа представлена с помощью def-use chains. Преобразование заключается в удалении такого оператора, у которого не используются его выходы.

Пример применения такого преобразования приведен на иллюстрации.

Данное преобразование повторно по отношению к самому себе, поскольку удаление одного оператора приводит к тому, что операторы, вырабатывающие для него данные, также могут оказаться неиспользуемыми.

Чистка циклов вверх



Чистка циклов вверх

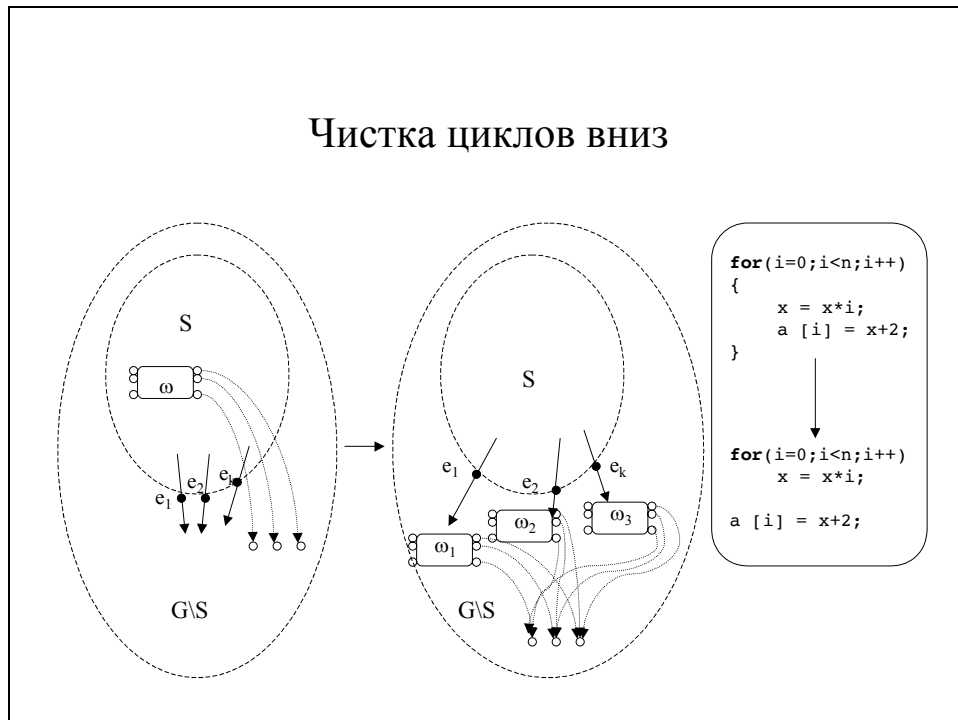
Данное преобразование применяется к сильно связным подграфам графа управления (т.е. подграфам, состоящим из взаимно достижимых вершин). Таким образом, для его проведения необходима фрагментация программы на уровне сильно связных подграфов. Алгоритм выделения сильно связных подграфов будет рассмотрен в лекции 12.

Если такой подграф выделен, то для него определяется множество входных вершин, то есть таких его вершин, в которые существует путь из вершины *start*, лежащий целиком за пределами этого подграфа. На иллюстрации входные вершины подграфа *S* обозначены темными кружками и символами e_1, \dots, e_k .

Чистка цикла вверх заключается в том, что вершина, лежащая в сильно связном подграфе, заменяется на несколько копий вне его, каждая из которых непосредственно предшествует входным вершинам. Такое преобразование возможно только в случае, когда оператор, которым помечена данная вершина, использует только данные операторов, недостижимых изнутри выбранного сильно связного подграфа.

Легко видеть, что данное преобразование также повторно по отношению к самому себе.

Чистка циклов вниз



Чистка циклов вниз

Данное преобразование также легко реализуется в представлении с использованием `def-use chains` и в каком-то смысле является симметричным предыдущему.

Теперь вершина изнутри сильно связного подграфа заменяется на несколько копий, каждая из которых является непосредственным преемником выходных вершин подграфа. Выходной вершиной называется такая вершина, среди непосредственных преемников которой есть вершина, не принадлежащая данному подграфу.

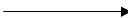
Легко видеть, что преобразование корректно только в том случае, когда ни одна вершина подграфа не достижима из тех вершин, которые используют результаты оператора, которым помечена выносимая вершина.

Данное преобразование также повторно по отношению к самому себе.

Далее мы перейдем к описанию преобразований, для которых недостаточно рассмотренных выше способов представления программы.

Объединение циклов

```
for (i=0; i<n; i++)  
    a [i] = i;  
  
for (i=0; i<n; i++)  
    b [i] = 2 * a [i];
```



```
for (i=0; i<n; i++)  
{  
    a [i] = i;  
    b [i] = 2 * a [i];  
}
```

Объединение циклов

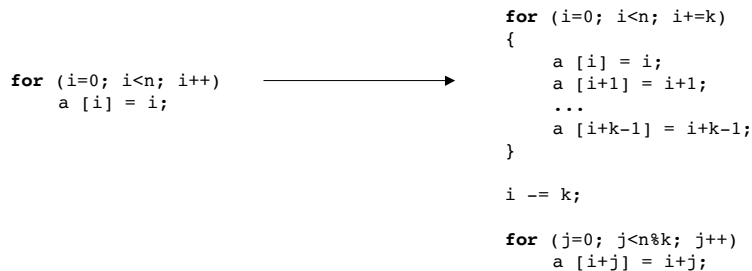
Объединение циклов производится с целью сокращения накладных расходов на организацию цикла.

Контекстные условия корректности этого преобразования следующие:

- Границы, переменная и инкремент циклов должны быть одинаковыми.
- Ни одна итерация операторов второго цикла не должна использовать результаты последующих итераций первого.

Заметим, что условие одинаковости переменной циклов не является самым точным. Более правильно было бы сказать, что должно быть осуществимо такое переименование переменных, при котором переменные цикла становятся одинаковыми.

Раскрутка циклов



```
for (i=0; i<n; i++)  
    a [i] = i;  
→  
for (i=0; i<n; i+=k)  
{  
    a [i] = i;  
    a [i+1] = i+1;  
    ...  
    a [i+k-1] = i+k-1;  
}  
i -= k;  
for (j=0; j<n%k; j++)  
    a [i+j] = i+j;
```

Раскрутка циклов

Раскрутка циклов заключается в дублировании тела цикла и сокращении числа итераций. Кроме того, необходимо позаботиться, чтобы все вхождения переменной цикла в тело были заменены выражениями, вычисляющими требуемые значения с учетом измененных значений переменной цикла.

В современных машинных архитектурах используется конвейерный принцип – некоторый участок кода загружается во внутреннюю память процессора заранее в предположении, что его все равно придется исполнять. В ситуации, когда осуществляется переход за границы этой предварительно загруженной области осуществляется так называемый сброс конвейера, что сопровождается существенной временной задержкой. Понятно, что источником таких сбросов являются условные и безусловные переходы, в частности, переходы на начало циклов. Раскрутка предпринимается с целью увеличить протяженность линейных участков для того, чтобы предотвратить сброс конвейера при организации очередной итерации.

Пример этого преобразования приведен на иллюстрации. Как видно, в общем случае помимо раскрученного цикла необходима организация дополнительного цикла, осуществляющего довычисления для остатка итераций.

Понижение силы операций

```
for (k=1; k<=M; k++)  
{  
    v = (k-1) * n + i;  
    a [v] = 0;  
}  
→  
v = i;  
for (k=1; k<=M; k++)  
{  
    a [v] = 0;  
    v = v + n;  
}
```

Понижение силы операций

Данное преобразование особенно эффективно при применении к вычислениям внутри тел циклов, хотя это не является его необходимым контекстным условием.

Основная идея понижения силы операций – это замена (в частных случаях) использования более дорогих операций более дешевыми (умножения – сложением или сдвигом, возведения в степень – умножением и т.д.) Обычно используются следующие тождества:

- $x * 2^k \equiv x \ll k$
- $x^2 \equiv x * x$
- $2 * x \equiv x + x$
- $n * x \equiv (n - 1) * x + x$
- $x^n \equiv x^{n-1} * x$

Два последних тождества и дают возможность применения преобразования к циклам. На иллюстрации приведен пример такого преобразования. В данном случае умножение внутри тела цикла было заменено сложением.

Упрощение выражений

Константные вычисления:

```
c=1;  
a=c+7-4*3+(g[i]-8);
```



```
c=1;  
a=g[i]-12;
```

Использование алгебраических свойств:

$$1 * a = a$$
$$0 + a = a$$

Упрощение выражений

Упрощение выражений представляет собой целый класс оптимизирующих преобразований. Наиболее часто встречающимися среди них являются константные вычисления и упрощения с использованием алгебраических тождеств. Понижение силы операций также может быть рассмотрено как преобразование из серии упрощения выражений.

Пример константных вычислений показан на иллюстрации.

Экономия общих подвыражений

$$d = (a+b+c) * (a+b) \longrightarrow \begin{array}{l} x = a+b; \\ d = (x+c) * x \end{array}$$

<pre> a = b+c+d; if (p(x)) x = b+c+e; else x = c+d+f; y = x+b+c; </pre>	\longrightarrow	<pre> a1 = b+c; a = a1+d; if (p(x)) x = a1+e; else x = c+d+f; y = x + a1; </pre>
---	-------------------	--

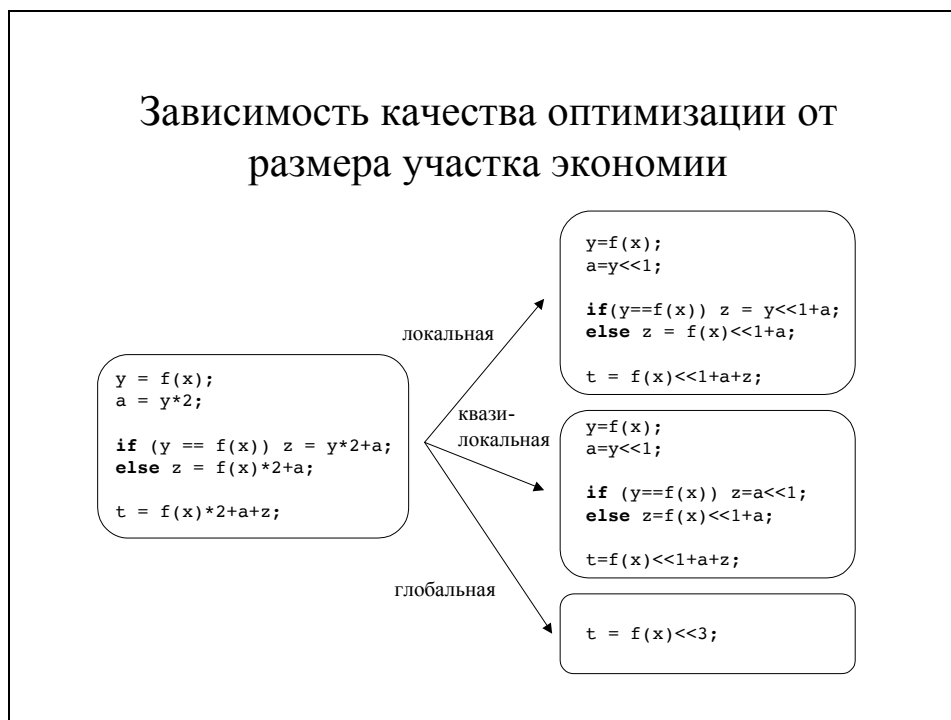
Экономия общих подвыражений

Экономия общих подвыражений заключается в том, что среди всех вычисляемых на участке экономии выражений выделяются эквивалентные, и затем их вхождения заменяются на вхождение новой переменной, хранящей заранее вычисленное значение этого общего выражения.

Поскольку выделение одних подвыражений закрывает возможность выделить другие, данное преобразование является неоднозначным.

На иллюстрации показаны примеры применения этого преобразования. Во втором случае видна неоднозначность при выделении подвыражений: выражения $b+c$ и $c+d$ не могут быть выделены одновременно. Поскольку подвыражение $b+c$ используется чаще, именно оно и было выделено.

Зависимость качества оптимизации от размера участка экономии



Зависимость качества оптимизации от размера участка экономии

Продemonстрируем теперь на примере зависимость качества оптимизации от размера участка экономии.

Фрагмент исходной программы показан в левой части иллюстрации. В правой части изображены возможные остаточные программы при локальной, квазилокальной и глобальной оптимизациях.

При локальной оптимизации произведено только понижение силы операций (умножение заменено на сдвиг).

При квазилокальной оптимизации помимо этого может быть упрощено выражение в then-части условного оператора. Для этого, однако, участок экономии должен включать в себя конструкцию ветвления.

Наконец, при глобальной оптимизации и при соответствующих свойствах функции f (ее значение должно полностью определяться значениями параметров и она не должна иметь побочных эффектов) исходный фрагмент может быть сокращен до вида, указанного на иллюстрации.

Литература к лекции

- А. Ахо, Р. Сети, Дж. Ульман. "Компиляторы: принципы, технологии и инструменты", М.: "Вильямс", 2001. 768 с.
- Steven S. Muchnik "Advanced Compiler Design And Implementation". Morgan Kaufmann Publishers, July 1997, 880 pp.
- В.Н.Касьянов "Оптимизирующие преобразования программ", М., "Наука", 1988. 336 с.

Литература к лекции

- А. Ахо, Р. Сети, Дж. Ульман "Компиляторы: принципы, технологии и инструменты", М.: "Вильямс", 2001. 768 с.
- Steven S. Muchnik "Advanced Compiler Design And Implementation", Morgan Kaufmann Publishers, July 1997. 880 pp.
- В.Н. Касьянов "Оптимизирующие преобразования программ", М., "Наука", 1988. 336 с.