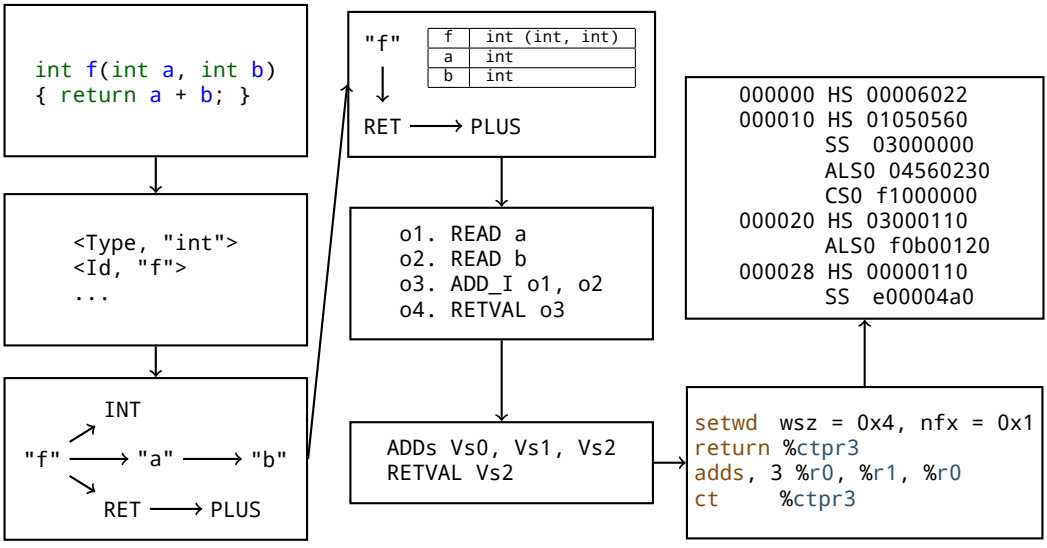
Проектирование компиляторов

Лекция 9.
Локальные оптимизации и оптимизации потока данных.

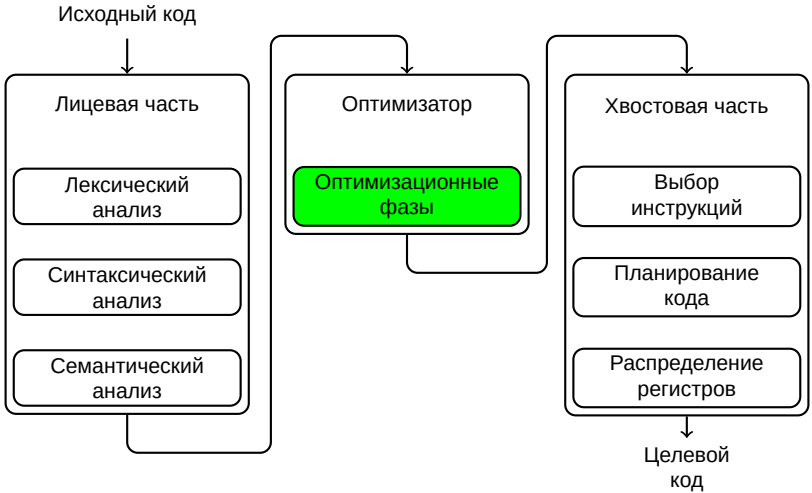
Маркин А. Л.
alexanius@gmail.com

2021

Компиляция программы



Компиляция программы



Локальные оптимизации

Локальные оптимизации — класс оптимизаций, работающих с небольшим участком представления. Обычно это линейный участок или его часть. Иногда в рассмотрение могут попадать несколько окружающих линейных участков.

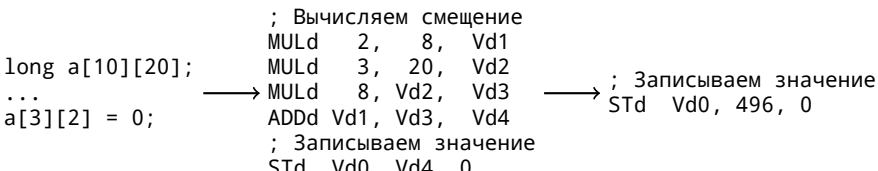
Локальные оптимизации предназначены для уменьшения времени исполнения отдельных линейных участков а также для устранения избыточности кода.

Большинство локальных оптимизаций работают с потоком данных, т.е. применяют граф определений и использований.

Локальные оптимизации

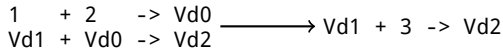
Возможные источники контекста для локальных оптимизаций:

- Высокоуровневый язык программирования.
- Неоптимальность кода (иногда оправданная).
- Другие оптимизации.



Свёртка констант

Свёртка констант (Constant folding) — вычисление выражений, аргументы которых являются константами:



Аналогичным образом оптимизация работает для битовых и адресных операций.

Свёртка констант

При выполнении свёртки констант могут встречаться ошибки в коде.
Например:

```
1 / 0 -> Vd0
```

Свёртка констант

При выполнении свёртки констант могут встречаться ошибки в коде.
Например:

```
1 / 0 -> Vd0
```

Возможны разные проблемные ситуации:

- Целочисленные переполнения
- Деление на ноль
- Изменение результата плавающих вычислений

Свёртка констант

Свёртка плавающих констант представляет особую проблему:

- Выполнение плавающих операций внутри компилятора и при исполнении на реальной машине может различаться
- Стандарт, описывающий плавающие числа (IEEE-754), содержит много особых ситуаций: бесконечности, NaN, денормализованные значения и другие. Необходимо внимательно его изучить и аккуратно перенести в оптимизации.

Алгебраические упрощения

Алгебраические упрощения используют алгебраические свойства операций для упрощения выражений:

```
Vd0 + 0       = 0 + Vd0   = Vd0 - 0 = Vd0
0   - Vd0     = -Vd0
Vd0 * 1       = 1 * Vd0   = Vd0 / 1 = Vd0
Vd0 * 0       = 0 * Vd0   = 0
Vd0 + (-Vd1) = Vd0 - Vd1
-(-Vd0)       = Vd0

Vd2 || true  = true  || Vd2 = true
Vd2 || false = false || Vd2 = Vd2
Vd2 && true  = true  && Vd2 = Vd2
Vd2 && false = false && Vd2 = false
```

Алгебраические упрощения

Некоторые упрощения могут влиять на результат работы программы:

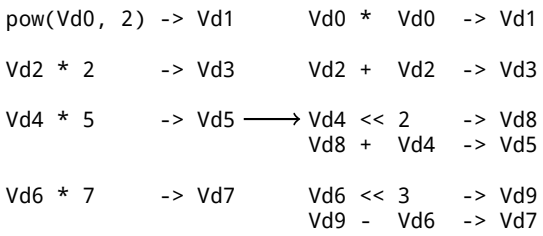
```
1   / Vd0 -> Vd2
Vd1 * Vd2 -> Vd3
```

```
Vd1 / Vd0 -> Vd3
```

Данное преобразование корректно математически, но результат в случае вычислений с плавающей точкой будет различаться

Понижение силы операций

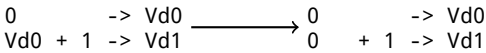
Некоторые исходные вычисления можно заменить более дешёвыми аналогами:



Такое преобразование называется **понижение силы операций**.

Распространение констант

Осуществление свёртки констант не всегда возможно напрямую:



Распространение констант (пропагация) — процесс подстановки значений, известных на этапе компиляции в выражения.

Оконные оптимизации

Оптимизации, занимающиеся мелкими локальными заменами и улучшениями обычно группируют в одну оптимизацию, называемую **оконной** (peephole).

Это набор оптимизаций, основанный на шаблонах, который видит несколько операций из линейного участка, т.е. смотрит на них через «окно» или «замочную скважину».

Компилятор **lcc** содержит более 200 шаблонов для данной оптимизации.

Оконные оптимизации

Эффекты от оконных оптимизаций:

- Уменьшение времени исполнения кода
- Уменьшение количества использованных регистров
- Уменьшение размера кода
- Создание контекста для других оптимизаций
- Возможно изменение поведения программы для отдельных случаев (например оптимизация плавающих вычислений)

Алгебраическое переупорядочивание

Даже после применения оконных оптимизаций не всегда возможно сразу произвести дальнейшую оптимизацию:

```
for(int i = 0; i < N; i++)
    for(int j = 0; j < M; j++)
        ind = (i * 2) + 4 + j + 2 + k + 1;
        ...
```

```
i   + i -> Vs0
Vs0 + 4 -> Vs1
Vs1 + j -> Vs2
Vs2 + 2 -> Vs3
Vs3 + k -> Vs4
Vs4 + 1 -> ind
```

Несмотря на очевидную необходимость, оконные оптимизации не смогут примениться, т.к. не увидят подходящего шаблона

Алгебраическое переупорядочивание

Используя свойства операций, возможно произвести **алгебраическое переупорядочивание**, т.е. перестановку операндов таким образом чтобы оконные оптимизации увидели контекст применения:

```
for(int i = 0; i < N; i++)
    for(int j = 0; j < M; j++)
        ind = (i * 2) + 4 + j + 2 + k + 1;
```

```
i   * 2   -> Vs0
Vs0 + 4   -> Vs1
Vs1 + j   -> Vs2
Vs2 + 2   -> Vs3
Vs3 + k   -> Vs4
Vs4 + 1   -> ind
```

```
for(int i = 0; i < N; i++)
    for(int j = 0; j < M; j++)
        ind = j + i * 2 + k + 4 + 2 + 1;
```

```
1   + 2   -> Vs0
Vs0 + 4   -> Vs1
Vs1 + k   -> Vs2
i   * 2   -> Vs3
Vs3 + Vs2 -> Vs4
Vs4 + j   -> ind
```

Алгебраическое переупорядочивание

После переупорядочивания оконные оптимизации снова смогут улучшить код:

```
for(int i = 0; i < N; i++)
    for(int j = 0; j < M; j++)
        ind = j + i * 2 + k + 4 + 2 + 1;
```

```
1   + 2   -> Vs0
Vs0 + 4   -> Vs1
Vs1 + k   -> Vs2
i   * 2   -> Vs3
Vs3 + Vs2 -> Vs4
Vs4 + j   -> ind
```

```
for(int i = 0; i < N; i++)
    for(int j = 0; j < M; j++)
        ind = j + i * 2 + k + 7;
```

```
7   + k   -> Vs2
i   * 2   -> Vs3
Vs3 + Vs2 -> Vs4
Vs4 + j   -> ind
```

Кроме того, в данном коде улучшился контекст и для других оптимизаций.

Алгебраическое переупорядочивание

Принцип применения алгебраического переупорядочивания:

1. Для каждого подвыражения считаем ранг:
   - Наименьший ранг у констант
   - Повышается ранг у инвариантов для каждого цикла
   - Повышается ранг для индуктивных переменных для каждого цикла
2. Сортируем подвыражения в соответствии с их рангом

Алгебраическое переупорядочивание

Преимущества:

- Создание контекста для других оптимизаций (особенно свёртки констант и выноса инвариантов)

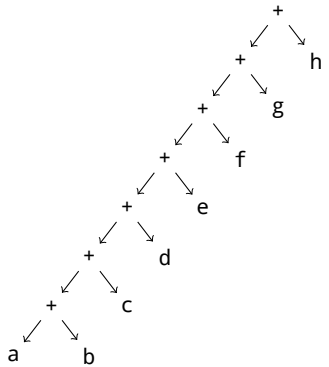
Недостатки:

- Изменение работы программы (если для операции отсутствует ассоциативность)
- Может испортить контекст для других оптимизаций (например для удаления общих подвыражений)

Проблема

Рассмотрим дерево разбора и промежуточный код выражения:

```
a + b + c + d + e + f + g + h
```

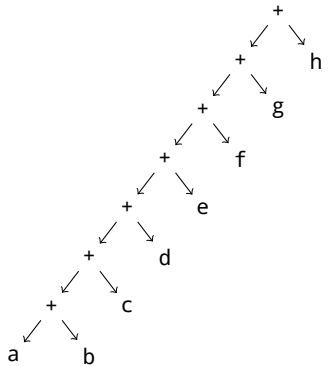



```
a   + b -> Vs0
Vs0 + c -> Vs1
Vs1 + d -> Vs2
Vs2 + e -> Vs3
Vs3 + f -> Vs4
Vs4 + g -> Vs5
Vs5 + h -> Vs6
```

Проблема

Рассмотрим дерево разбора и промежуточный код выражения:

```
a + b + c + d + e + f + g + h
```

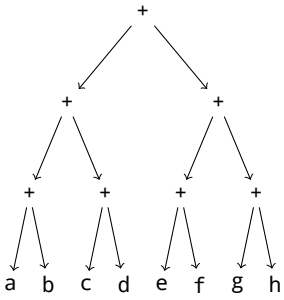



```
a   + b -> Vs0
Vs0 + c -> Vs1
Vs1 + d -> Vs2
Vs2 + e -> Vs3
Vs3 + f -> Vs4
Vs4 + g -> Vs5
Vs5 + h -> Vs6
```

Проблема: код является последовательным и не позволяет использовать параллелизм уровня инструкций.

Балансировка выражений

Дерево выражений можно сбалансировать:




```
a    + b    -> Vs0
c    + d    -> Vs1
e    + f    -> Vs2
g    + h    -> Vs3
Vs0  + Vs1  -> Vs4
Vs2  + Vs3  -> Vs5
Vs4  + Vs5  -> Vs6
```

Балансировка выражений

Планирование операций до и после балансировки:

```
0:  a   + b -> Vs0 // АЛК 0
4:  Vs0 + c -> Vs1 // АЛК 0
8:  Vs1 + d -> Vs2 // АЛК 0
12: Vs2 + e -> Vs3 // АЛК 0
16: Vs3 + f -> Vs4 // АЛК 0
20: Vs4 + g -> Vs5 // АЛК 0
24: Vs5 + h -> Vs6 // АЛК 0
```

```
0: a   + b -> Vs0   // АЛК 0
0: c   + d -> Vs1   // АЛК 1
0: e   + f -> Vs2   // АЛК 2
0: g   + h -> Vs3   // АЛК 3
4: Vs0 + Vs1 -> Vs4 // АЛК 0
4: Vs2 + Vs3 -> Vs5 // АЛК 1
8: Vs4 + Vs5 -> Vs6 // АЛК 0
```

Увеличение параллелизма уровня инструкций ускорило исполнение на 16 тактов.

Балансировка выражений

Преимущества:

- Увеличение параллелизма уровня инструкций

Недостатки:

- Изменение работы программы (если для операции отсутствует ассоциативность)
- Может испортить контекст для других оптимизаций (например для удаления общих подвыражений)

Проблема

В коде часто можно встретить дублирующиеся вычисления:

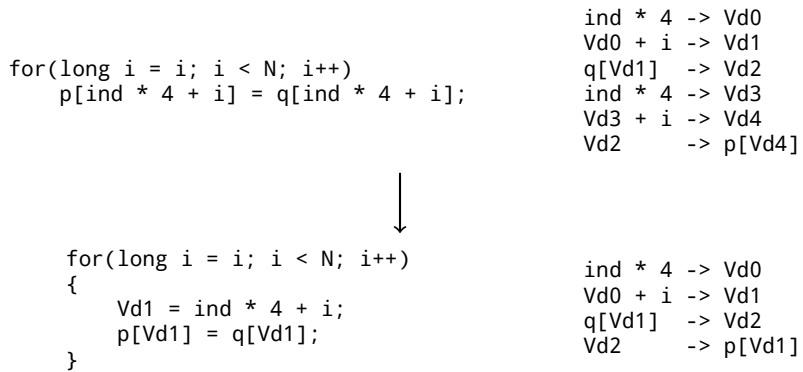
```
for(long i = i; i < N; i++)
    p[ind * 4 + i] = q[ind * 4 + i];
```

Промежуточное представление ациклического участка:

```
ind * 4 -> Vd0
Vd0 + i -> Vd1
q[Vd1]  -> Vd2
ind * 4 -> Vd3
Vd3 + i -> Vd4
Vd2     -> p[Vd4]
```

Удаление общих подвыражений

Компилятор умеет определять общие подвыражения и удалять дублирующие вычисления:



Такая оптимизация называется **удалением общих подвыражений** (common subexpression elimination, cse).

Нумерация значений

Для поиска общих подвыражений применяется специальная техника, называемая **нумерация значений**.

Класс эквивалентности (конгруэнтности) — подмножество операций, безусловно имеющих одинаковый результат.

Нумерация значений (value numbering) — анализ, определяющий класс эквивалентности у различных операций.

Нумерация значений

Основная идея нумерации значений:

Каждой операции присвоить класс эквивалентности — символьное значение, которое определяет её результат.

Класс эквивалентности основывается на действии, выполняемом операцией и классах эквивалентности её аргументов.

```
ind    (c1) * 4 (c2) -> Vd0    | (c3)
Vd0    (c3) + i (c4) -> Vd1    | (c5)
q[Vd1] (c6)          -> Vd2    | (c7)
ind    (c1) * 4 (c2) -> Vd3    | (c3)
Vd3    (c3) + i (c4) -> Vd4    | (c5)
Vd2    (c7)          -> p[Vd4] | (c8)
```

Удаление общих подвыражений

Преимущества:

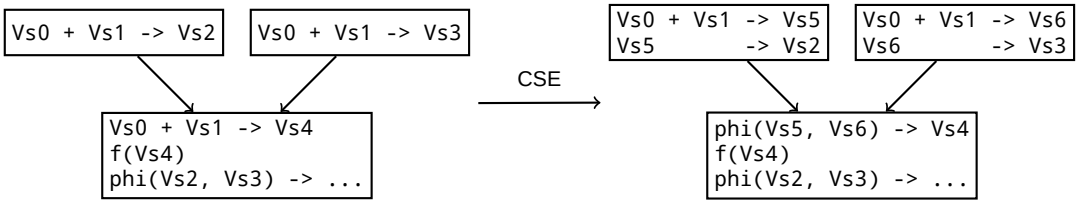
- Ускорение программы за счёт уменьшения количества вычислений.

Недостатки:

- Увеличение времени жизни регистров.

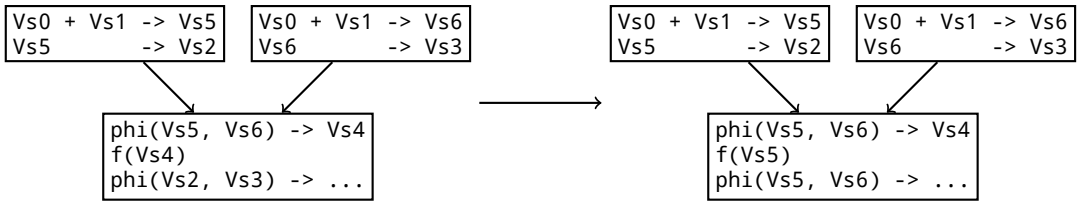
Проблема

Работа различных оптимизаций может привести к лишним операциям копирования значений:



Распространение копий

Распространение копий (copy propagation) — оптимизация, при наличии пересылок вида `Vs0 -> Vs1` заменяющая использование копии значения на само значение.



В дальнейшем избыточный код будет удалён другими оптимизациями

Распространение копий

Преимущества:

- Создание контекста для устранения избыточных операций.

Недостатки:

- Увеличение времени жизни регистров.

Проблема

```
int f(int x, int y)
{
    int w = x * y;
    return x + y;
}
```

Проблема

```
int f(int x, int y)
{
    int w = x * y;
    return x + y;
}
```

В данном коде присутствует избыточность: выполняются операции, результат которых никак не используется далее.

Удаление мёртвого кода

Переменная называется **мёртвой**, если на пути от её определения до точки выхода у неё нет использований.

Операция называется **мёртвой** если вычисляемые ей значения не используются на всём пути исполнения.

```
int f(int x, int y)
{
    int w = x * y; // Мёртвый код
    return x + y;
}
```

Удаление мёртвого кода

Причины появления мёртвого кода:

- Работа оптимизаций
- Раскрытие языковых конструкций (например макросов)
- Неаккуратная работа программиста (довольно редко)

```
int f(int x, int y)
{
    int w = x * y; // Мёртвый код
    return x + y;
}
```

Удаление мёртвого кода

Алгоритм **удаления мёртвого кода** (Dead Code Elimination, dce):

1. Необходимо пометить маркером все априори живые операции:
   - Переходы
   - Запись в память
   - Вызовы функций
2. Для каждой живой операции поднимаемся по графу определений и использований и помечаем все операции как живые
3. Все операции, не помеченные как живые считаются мёртвыми, и их можно удалять

```
int f(int x, int y)
{
    int w = x * y; // Мёртвый код
    return x + y;
}
```

Удаление мёртвого кода

Эффекты от применения **удаления мёртвого кода**:

- Ускорение программы за счёт удаления лишних операций
- Уменьшение нагрузки на регистры
- Облегчение работы других оптимизаций

Литература

- *S. S. Muchnick* Advanced Compiler Design and Implementation.
- *K. D. Cooper, L. Torczon* Engineering a compiler.
- *А. В. Ахо, М. С. Лам, Р. Сети, Д. Д. Ульман* Компиляторы: принципы, технологии и инструменты.
- *Chirag H. Bhatt, Harshad B. Bhadka* Peephole Optimization Technique for analysis and review of Compiler Design and Construction