Министерство образования и науки Российской Федерации
Новосибирский государственный технический университет
Кафедра прикладной математики

# Языки программирования и методы трансляции

## Расчетно-графическая работа

| | |
|---|---|
| Факультет | прикладной математики и информатики |
| Группа | ПМ-01 |
| Студенты | Александров М.Е. |
| | Жигалов П.С. |
| Преподаватели | Еланцева И.Л. |
| | Полетаева И.А. |
| Вариант | 2 |

Новосибирск
2013

## 1. Тема

Методы оптимизации кода. Оптимизации выражений с константами.

## 2. Введение

На сегодняшний день все компиляторы C/C++ близки по предоставляемым возможностям, поэтому оптимизация остается одним из важнейших отличий, по которому их можно дифференцировать. Работа компилятора заключается в том, чтобы перевести процедурное описание задачи и эффективно отобразить его в выделенное множество машинных команд процессора. Степень эффективности генерации компилятором низкоуровневого кода может иметь существенное влияние на скорость выполнения программы и ее размер.

Основной целью оптимизации служит выработка более экономичного по потребляемым ресурсам кода.

Немаловажную роль играет тщательный анализ задачи и хорошая программная реализация, которые могут обеспечить возможность последующих усовершенствований, которые могут быть внесены оптимизирующим компилятором. Нельзя ожидать, что оптимизация кода компенсирует программам плохо продуманный алгоритм и неверный выбор структур данных.

## 3. Критерии оптимизации

Наилучшие преобразования программы – те, которые приводят к максимальному результату при минимальных усилиях. Преобразования, выполняемые оптимизирующим компилятором, должны обладать рядом свойств.

Во-первых, преобразование должно ускорять работу программы или уменьшать выделенный ей объем памяти, т.е. она должна потреблять меньшее количество ресурсов, как по памяти, так и по времени процессора.

Во-вторых, преобразование должно быть устойчивым, т.е. при одинаковых входных данных оптимизированная и неоптимизированная программа должны выдавать один и тот же результат.

В-третьих, преобразования должны стоить затрачиваемых усилий.

## 4. Виды оптимизаций

Существуют различные методы машинно-зависимой и машинно-независимой оптимизации кода. Они могут применяться на всех синтаксических уровнях.

В общем случае, одни и те же конструкции языка высокого уровня могут быть по-разному реализованы в машинных кодах.

Все следующие оптимизации призваны создать оптимальный набор машинных инструкций соответствующих коду программы:

- скалярные оптимизации
- цикловые оптимизации
- векторизация
- межпроцедурный анализ
- …

Более подробно остановимся на скалярных оптимизациях. Скалярные оптимизации обычно разделяют на следующие виды:

- Свертка констант (Constant folding)
- Размножение констант (constant propagation)
- Размножение копий (copy propagation)
- Удаление общих подвыражений (Common subexpression elimination)
- Удаление мертвого кода (Dead code elimination)
- …

Остановимся подробнее на методах оптимизации вычислений с константами.

## 5. Оптимизации выражений с константами

### *5.1. Свертка констант*

Свертка констант – оптимизация, вычисляющая константные выражения на этапе компиляции. Прежде всего, упрощаются константные выражения, содержащие числовые литералы. Также могут быть упрощены выражения, содержащие никогда неизменяемые переменные или переменные, объявленные как константы. Например, в случае непосредственного использования константных данных:

```
i = 320 * 200 * 32;
```

компилятор, поддерживающий свертку констант, не будет генерировать две инструкции умножения и запись полученного результата. Вместо этого он распознает эту конструкцию как константное выражение и заменит ее на выражение

```
i = 2048000;
```

Константные данные также могут быть объявлены косвенно, с использованием объявленных для препроцессора манифестных констант. В этом случае свертка констант сведет строки

```
#define TWO 2

a = 1 + TWO;
```

к их эквивалентной форме

```
a = 3;
```

В Си сворачивание констант применяют как к целым константам, так и к константам с плавающей точкой.

### *5.2. Алгебраические упрощения*

Как частный случай свертки констант также выделяют так называемые алгебраические упрощения. Алгебраические упрощения – это такой вид свертки, который удаляет арифметические тождества. Другими словами, код, сгенерированный для таких операторов, как

```
x = y + 0;          x = 0 + y;          x = y - 0;

x = y * 1;          x = 1 * y;          x = y / 1.0;

x = y * 0;          x = 0 * y;          x = y / 0;
```

должен быть простым присваиванием и не должен содержать команд для выполнения арифметических операций. Важно, что компилятор должен пометить последний оператор как ошибочный и не генерировать код для него.

### *5.3. Распространение констант*

Распространение констант – оптимизация, заменяющее выражение, которое при выполнении всегда возвращает одну и ту же константу, самой этой константой. Например:

```
int x = 14;

int y = 7 - x / 2;

return y * (28 / x + 2);
```

Распространение x возвращает:

```
int x = 14;

int y = 7 - 14 / 2;
```

```
return y * (28 / 14 + 2);
```

Далее, свёртка констант и распространение y возвращают следующее (присваивания x и y, вероятно, в дальнейшем будут удалены оптимизацией удаления мёртвого кода):

```
int x = 14;

int y = 0;

return 0;
```

### *5.4. Распространение копий*

Похоже на распространение констант распространение копий, только вместо константных значений используются переменные. Это позволяет уменьшить число избыточных переменных, содержащих одно и то же значение.

```
x = y;

z = x;

if(a < x && b < z)

       c = x;
```

переводится в

```
if(a < y && b < y)

       c = y;
```

Таким образом, удалось избавиться от двух переменных, x и z и лишних присваиваний. Строго говоря, данная оптимизация не работает напрямую с константами, однако ее применение вместе с различными видами оптимизаций констант позволяет добиться значительно большей их эффективности.

### *5.5. Прямое преобразование*

Метод прямого преобразования позволяет значительно ускорить выполнение программы за счет снижения стоимости выполнения операций. Суть его заключается в том, что для некоторых операций возможно подобрать аналог, стоимость выполнения которого ниже.

Например, можно заменить операции деления и умножения целых чисел и выражений, возвращающих целый результат, на операции побитового сдвига. Например, код:

```
int x;

...

x = x * 4;

x = x / 2;
```

эквивалентен следующему:

```
int x;

...

x = x << 2;

x = x >> 1;
```

но выполняться он будет быстрее.

## 6. Реализация в лабораторной работе

В рамках лабораторной работы были реализованы следующие методы оптимизаций выражений с константами:

- свертка констант (частично)
- алгебраические упрощения (частично)

Распространение констант и распространение копий реализовывать в лабораторной работе несколько бессмысленно, поскольку при правильном применении данных оптимизаций, программа выродится в так называемый "мертвый код".

Метод прямого преобразования реализовать затруднительно, так как по заданию лабораторной работы, число операций ограничено.

### *6.1. Алгоритм свертки констант*

Результат работы синтаксического анализатора представлен в виде постфиксной записи из элементов специального типа, это накладывает определенные ограничения на работу оптимизатора.

Рассмотренный алгоритм позволяет упрощать константные выражения в том случае, если при обходе обратной польской записи стандартным алгоритмом Дейкстры при выполнении операции оба операнда будут константами. То есть алгоритм упростит следующее выражение:

```
a = (2 + 5 + b) * (6 - 8 + c);
```

но не сможет упростить это:

```
a = (2 + b + 5) * (6 + c - 8);
```

Сам алгоритм:

Дано выражение в постфиксной записи. Нам понадобится стек для хранения смешанных данных (чисел и операторов). Просматриваем выражение слева направо.

Пока есть символы для чтения:

- Читаем очередной символ.
- Если символ является числом, помещаем его в стек.
- Если символ является переменной, считая что переменная имеет значение *null*, помещаем символ в стек.
- Если символ является оператором:
- (если все аргументы оператора, лежащие в стеке, имеют значение, отличное от *null*) выталкиваем аргументы оператора из стека и помещаем в стек результат операции;
- (если хотя бы один из аргументов имеет значение *null*) считая что результат операции *null*, кладём символ оператора в стек.

После того, как всё выражение просмотрено, то, что осталось в стеке, является оптимизированным выражением (операторы выражения лежат в стеке в обратном порядке).

Примечание: так как операторы лежат в обратном порядке, удобнее будет воспользоваться деком вместо стека.

### *6.2. Алгоритм алгебраических упрощений*

Так как алгоритм является дополнением к предыдущему, то он имеет аналогичные недостатки.

Для выполнения алгебраических упрощений можно перед проверкой условия последнего шага предыдущего алгоритма проверить, удовлетворяют ли операнды и операция выражениям, приведенным в пункте 5.2. и, если удовлетворяют, положить в стек результат операции.

Следует отметить, что после алгебраических упрощений могут оказаться выражения, которые можно еще упростить, в уже просмотренной части постфиксной записи. Поэтому, при необходимости, следует повторно выполнить оптимизацию.

## 7. Тесты

### *7.1. Хорошо оптимизирующийся тест*

Выражение до: `a = (2 + 5 + b) * 1 * 15 - (6 - 8 + c) * 1 + a * 0 - 0;`

ОПЗ до: `a 2 5 + b + 1 * 15 * 6 8 - c + 1 * - a 0 * + 0 - =`

ОПЗ после: `a 7 b + 15 * -2 c + - =`

Выражение после: `a = (7 + b) * 15 – (-2 + c)`

### *7.2. Плохо оптимизирующийся тест*

Выражение до: `a = 1 * (2 + b + 5) * 15 - (6 + c - 8) * 1 + 0 * a - 0;`

ОПЗ до: `a 1 2 b + 5 + * 15 * 6 c + 8 - 1 * - 0 a * + 0 – =`

ОПЗ после: `a 1 2 b + 5 + * 15 * 6 c + 8 - - =`

Выражение после: `a = 1 * (2 + b + 5) * 15 – (6 + c - 8)`

## 8. Код функций, связанных с оптимизацией

```
    // Структура элемент постфиксной записи
    class postfix_elem
    {
    public:
        string id;
        short int type;
        short int table;
        postfix_elem()
        {
            id = "", type = 0, table = -1;
        }
        postfix_elem(string id_, int type_, int table_)
        {
            id = id_, type = type_, table = table_;
        }
        postfix_elem(string id_, int type_)
        {
            id = id_, type = type_, table = -1;
        }
        postfix_elem(string id_)
        {
            id = id_, type = 1, table = -1;
        }
        friend bool operator == (const postfix_elem& f, const postfix_elem& l)
        {
            if(f.type == l.type && f.table == l.table && f.id == l.id) return true;
            return false;
        }
        friend ostream& operator << (ostream& ostream_, const postfix_elem& pe_)
        {
            ostream_ << pe_.id;
            return ostream_;
        }
    };
```

```
// Алгебраические упрощения
bool translator::special_case(postfix_elem oper1p, postfix_elem oper2p, postfix_elem operation, postfix_elem&
result)
{
    // Штуки вроде a*0 или 0*a
    if(operation.id == "*" && (oper2p.id == "0.0" || oper2p.id == "0" ||
                               ((oper1p.id == "0.0" || oper1p.id == "0") &&
                                (oper2p.table == 6 || oper2p.table == 5)) ))
    {
        if(oper2p.id == "0.0" || oper2p.id == "0")
            result = oper2p;
        else
            result = oper1p;
        return true;
    }
    // Штуки вроде a*1 или 1*a
    else if(operation.id == "*" && (oper2p.id == "1.0" || oper2p.id == "1" ||
```

```
                                    ((oper1p.id == "1.0" || oper1p.id == "1") &&
                                     (oper2p.table == 6 || oper2p.table == 5)) ))
    {
        if(oper2p.id == "1.0" || oper2p.id == "1")
            result = oper1p;
        else
            result = oper2p;
        return true;
    }
    // Штуки вроде a+0 или 0+a
    else if(operation.id == "+" && (oper2p.id == "0.0" || oper2p.id == "0" ||
                                    ((oper1p.id == "0.0" || oper1p.id == "0") &&
                                     (oper2p.table == 6 || oper2p.table == 5)) ))
    {
        if(oper2p.id == "0.0" || oper2p.id == "0")
            result = oper1p;
        else
            result = oper2p;
        return true;
    }
    // Штуки вроде a-0
    else if(operation.id == "-" && (oper2p.id == "0.0" || oper2p.id == "0"))
    {
        result = oper1p;
        return true;
    }
    return false;
}

// Свертка констант
void translator::constant_folding(vector<postfix_elem>& postfix_tmp)
{
    deque<postfix_elem> optimize_deque;
    bool is_changed = true;
    while(is_changed)
    {
        is_changed = false;
        for(int i = 0; i < (int)postfix_tmp.size(); i++)
        {
            if(postfix_tmp[i].table == 5 || postfix_tmp[i].table == 6)
            {
                optimize_deque.push_back(postfix_tmp[i]);
            }
            else
            {
                int oper1i = 0, oper2i = 0;
                float oper1f = 0.0, oper2f = 0.0;
                int type1 = 0, type2 = 0, typer = 0;
                postfix_elem oper1p, oper2p;
                postfix_elem result_special;
                deque<postfix_elem>::iterator it = optimize_deque.end() - 1;
                oper2p = *it--;
                oper1p = *it;
                if(oper2p.table == 6)
                {
                    if(oper1p.table == 6)
                    {
                        //is_local_change = true;
                        optimize_deque.pop_back();
                        stringstream a;
                        lexeme lex;
                        constants.get_lexeme(oper1p.id, lex);
                        type1 = lex.type;
                        a.str("");
                        a.clear();
                        a.str(oper1p.id);
                        if(type1 == 2)
                            a >> oper1f;
                        else
                            a >> oper1i;
                        constants.get_lexeme(oper2p.id, lex);
                        type2 = lex.type;
                        a.str("");
                        a.clear();
                        a.str(oper2p.id);
                        if(type2 == 2)
                            a >> oper2f;
                        else
                            a >> oper2i;
                        // Приведение типов
                        if(type1 == 2)
```

```
{
    if(type2 != 2)
    {
        oper2f = (float)oper2i;
        type2 = 2;
    }
    typer = 2;
}
else
{
    if(type2 == 2)
    {
        oper1f = (float)oper1i;
        type1 = 2;
        typer = 2;
    }
    else
    {
        typer = 1;
    }
}
// Собственно выполнение операции
a.str("");
a.clear();
if(postfix_tmp[i].id == "+") // +
{
    if(typer == 2)
        a << oper1f + oper2f;
    else
        a << oper1i + oper2i;
    string result = a.str();
    constants.add(result);
    constants.set_type(result, typer);
    optimize_deque.push_back(postfix_elem(result, 1, 6));
    is_changed = true;
}
else if(postfix_tmp[i].id == "-") // -
{
    if(typer == 2)
        a << oper1f - oper2f;
    else
        a << oper1i - oper2i;
    string result = a.str();
    constants.add(result);
    constants.set_type(result, typer);
    optimize_deque.push_back(postfix_elem(result, 1, 6));
    is_changed = true;
}
else if(postfix_tmp[i].id == "*") // *
{
    if(typer == 2)
        a << oper1f * oper2f;
    else
        a << oper1i * oper2i;
    string result = a.str();
    constants.add(result);
    constants.set_type(result, typer);
    optimize_deque.push_back(postfix_elem(result, 1, 6));
    is_changed = true;
}
else if(postfix_tmp[i].id == "==") // ==
{
    if(typer == 2)
        a << (oper1f == oper2f);
    else
        a << (oper1i == oper2i);
    string result = a.str();
    constants.add(result);
    constants.set_type(result, 1);
    optimize_deque.push_back(postfix_elem(result, 1, 6));
    is_changed = true;
}
else if(postfix_tmp[i].id == "!=") // !=
{
    if(typer == 2)
        a << (oper1f != oper2f);
    else
        a << (oper1i != oper2i);
    string result = a.str();
    constants.add(result);
    constants.set_type(result, 1);
```

```
                        optimize_deque.push_back(postfix_elem(result, 1, 6));
                        is_changed = true;
                    }
                    else if(postfix_tmp[i].id == ">") // >
                    {
                        if(typer == 2)
                            a << (oper1f > oper2f);
                        else
                            a << (oper1i > oper2i);
                        string result = a.str();
                        constants.add(result);
                        constants.set_type(result, 1);
                        optimize_deque.push_back(postfix_elem(result, 1, 6));
                        is_changed = true;
                    }
                    else if(postfix_tmp[i].id == "<") // <
                    {
                        if(typer == 2)
                            a << (oper1f < oper2f);
                        else
                            a << (oper1i < oper2i);
                        string result = a.str();
                        constants.add(result);
                        constants.set_type(result, 1);
                        optimize_deque.push_back(postfix_elem(result, 1, 6));
                        is_changed = true;
                    }
                    else // Неразрешенная операция, откат
                    {
                        optimize_deque.push_back(oper1p);
                        optimize_deque.push_back(oper2p);
                        optimize_deque.push_back(postfix_tmp[i]);
                    }
                }
                else if(special_case(oper1p, oper2p, postfix_tmp[i], result_special))
                {
                    optimize_deque.pop_back();
                    optimize_deque.push_back(result_special);
                    is_changed = true;
                }
                else
                {
                    optimize_deque.push_back(oper2p);
                    optimize_deque.push_back(postfix_tmp[i]);
                }
            }
            else if(special_case(oper1p, oper2p, postfix_tmp[i], result_special))
            {
                optimize_deque.pop_back();
                optimize_deque.pop_back();
                optimize_deque.push_back(result_special);
                is_changed = true;
            }
            else
            {
                optimize_deque.push_back(postfix_tmp[i]);
            }
        }
    }
    postfix_tmp.clear();
    for(deque<postfix_elem>::iterator it = optimize_deque.begin(); it != optimize_deque.end(); it++)
        postfix_tmp.push_back(*it);
    optimize_deque.clear();
  }
}
```

**9. Список литературы**


1. Практикум "Оптимизирующие компиляторы" (на примере GCC). – НГУ им. Лобачевского.
2. Бьерн Страуструп, Язык программирования C++. Специальное издание. – М.: Бином, 2011.
3. http://ru.wikipedia.org/wiki/Обратная_польская_запись
4. Ахо А.В. и др, Компиляторы - принципы, технологии, инструменты. – Вильямс, 2003.
5. http://en.wikipedia.org/wiki/Optimizing_compiler
6. http://en.wikipedia.org/wiki/Constant_folding