КОНСТАНТИН ВЛАДИМИРОВ

ОПТИМИЗИРУЮЩИЕ КОМПИЛЯТОРЫ

СТРУКТУРА И АЛГОРИТМЫ

Издательство АСТ
Москва
2024

УДК 004.43
ББК 32.973
В57

Владимиров, Константин.

В57 Оптимизирующие компиляторы. Структура и алгоритмы / Владимиров К.; — Москва: Издательство АСТ, 2024. — 272 с. — (Программирование для всех)

ISBN 978-5-17-167965-1

«Оптимизирующие компиляторы» — настольная книга специалиста, который решил не просто укрепить свои знания, но и вывести навыки на новый уровень.

Вместе с Константином Владимировым вы разберете теорию оптимизирующей компиляции — все те сложные преобразования, которые происходят с текстом программы на его пути к исполняемому файлу, узнаете, что такое тулчейны и каким этапам трансформации подвергается программа до того, как будет впервые запущена, а также закрепите полученные знания, выполняя задания.

УДК 004.43
ББК 32.973

ISBN 978-5-17-167965-1

© Константин Владимиров, текст
© ООО Издательство «АСТ»

Содержание

Введение 6

1 Тулчейны 10
1.1 Общий обзор 11
1.2 Компиляторы 13
1.2.1 Лексический анализ 15
1.2.2 Синтаксический анализ 18
1.2.3 Промежуточное представление 22
1.2.4 Машинно-независимые оптимизации 27
1.2.5 Кодогенерация 29
1.3 Ассемблеры 32
1.3.1 Ассемблирование функций и понятие ABI 34
1.3.2 Дизассемблирование и релокации 37
1.4 Линкеры 40
1.4.1 Оптимизации времени линковки 42
1.4.2 Релаксации 44
1.5 Задачи на самостоятельную проработку 46

2 Введение в оптимизации 48
2.1 Поток управления 49
2.2 Решётки и распространение констант 55
2.2.1 Бинарные отношения 55
2.2.2 Решётки 57
2.2.3 Продвижение констант 60
2.3 Поток данных 64
2.3.1 Достигающие определения 66
2.3.2 Доступные выражения 69
2.3.3 Активные переменные и обратный анализ 70
2.3.4 Трансформации и анализ в HIR 72
2.4 Нумерация значений 75
2.5 Задачи на самостоятельную проработку 79

3 Построение SSA 81
3.1 SSA представление 82

3.1.1 Наивный способ построения SSA 85
3.2 Отношения и структуры доминирования 88
3.2.1 Дерево доминаторов 90
3.2.2 Фронт доминирования. 91
3.2.3 Графы схождения 94
3.3. Построение SSA 96
3.3.1 Свойства и вилы SSA 96
3.3.2 Основной алгоритм 98
3.3.3 Построение SSA в LLVM 102
3.4 Задачи на самостоятельную проработку 105

4 Базовые оптимизации для SSA 107
4.1 Начинаем использовать SSA 108
4.1.1 Работа с SSA представлением 109
4.1.2 Граф зависимостей SSA 111
4.2 Продвижение констант и не только 113
4.2.1 Простое продвижение констант 113
4.2.2 Добавляем информацию из CFG 116
4.2.3 Другие продвижения информации. 119
4.3 Глобальная нумерация значений 121
4.3.1 Построение классов конгруэнтности 121
4.3.2 Улучшение эффективности разбиения 125
4.4 Устранение частичной избыточности 127
4.4.1 Разбиение критического ребра. 128
4.4.2 Граф избыточных выражений 130
4.4.3 Алгоритм SSAPRE 133
4.5 Задачи на самостоятельную проработку 137

5 Цикловые оптимизации 139
5.1 Естественные циклы 143
5.1.1 Обходы в графах и обращение дуг 145
5.1.2 Сводимость. 148
5.1.3 Канонический вид цикла. 151
5.1.4 Замкнутое цикловое SSA. 155
5.2 Поиск и оптимизации индуктивностей. 158
5.2.1 Индуктивные переменные 158
5.2.2 Поиск базовых индуктивностей 162
5.2.3 Скалярная эволюция. 165
5.2.4 Упрощение вычислений в циклах 169
5.2.5 Раскрутка циклов 171

5.3 Инварианты и квазиинварианты 174
5.3.1 Вынос инвариантного кода из цикла 174
5.3.2 Пилинг циклов 176
5.4 Структурные оптимизации 180
5.4.1 Распределение и соединение циклов 181
5.4.2 Оптимизации для локальности данных 183
5.4.3 Разглаживание циклов 184
5.4.4 Вынос инвариантного управления 185
5.4.5 Управление цикловыми оптимизациями 186
5.5 Задачи на самостоятельную проработку 189

6 Межпроцедурные оптимизации 190
6.1 Клонирование и трансформации функций 194
6.1.1 Продвижение констант и прочей информации 195
6.1.2 Эвристики и управление эвристиками 198
6.1.3 Частичная девиртуализация 201
6.1.4 Хвостовая рекурсия 204
6.2 Подстановка и факторизация функций 208
6.2.1 Подстановка функций 208
6.2.2 Вынос функций 211
6.3 Задачи на самостоятельную проработку 213

7 Разрушение SSA 215
7.1 Выбор инструкций 218
7.1.1 Перевод в представление SDHIR 220
7.1.2 Легализация и выбор инструкций 224
7.1.3 Альтернативный подход: модификация LIR. 228
7.1.4 Конструкция Пробстинга 230
7.1.5 Подход на основе PBQF 232
7.2 Распределение регистров. 236
7.2.1 Разрушение SSA 237
7.2.2 Подходы, основанные на раскраске графа 240
7.2.3 Линейный подход к распределению регистров 244
7.2.4 Подходы, основанные на PBQP и ILP 247
7.3 Планирование инструкций 250
7.3.1 Планирование на виртуальных регистрах 252
7.3.2 Планирование на физических регистрах 255
7.4 Задачи на самостоятельную проработку 258

Заключение 260
Предметный указатель. 263

Введение

Tolle numerum omnibus rebus et omnia pereunt.

– st. Isidore of Seville

В этой книге речь в основном пойдёт о компиляторах.

Изначально, когда программирование ещё не было так широко известно, как сейчас, термин "компилятор" относился исключительно к литературе. Так назывался человек, осуществлявший соединение (то есть компиляцию) нескольких, иногда разноязычных, текстов. Такой человек решал две основные задачи. Во-первых, ему надо было перевести (транслировать) те тексты, которые были написаны не на целевом языке. При переводе хороший компилятор вполне мог соптимизировать излагаемый материал, сделав его короче или упростив формулировки (но стараясь не потерять суть). Во-вторых, ему нужно было скомпоновать нужные части переведённых текстов в единый кодекс, свод или энциклопедию. Например, святой Исидор Севильский был первым компилятором мосарабского богослужебного обряда. И он же был его компоновщиком.

В программировании всё очень похоже. Есть разные модули, написанные на разных языках разными программистами. И должна быть некая программа, которая помогает собрать их воедино и перевести на язык, который поймёт машина. Эта программа и есть компилятор в широком смысле. Она не столько выполняет, сколько организует (драйвит) этот процесс, запуская много других программ, которые и делают всё на пути от исходного кода к исполняемому файлу – то есть и трансляцию, и оптимизацию, и ассемблирование, и линковку. И чаще всего люди, когда говорят про компилятор, имеют в виду именно драйвер компиляции (например программу `gcc`).

Компилятор, в узком смысле, это одна из программ, вызываемых драйвером компиляции. Эта программа переводит код других программ с высокоуровневого языка на низкоуровневый, то есть делает их оптими-

зирующую трансляцию. Далее, когда я буду говорить о компиляторе как о программе, я буду иметь в виду именно “компилятор в узком смысле”.

На самом деле даже в узком смысле компилятор – это слишком широкая тема, которую будет сложно описать в рамках одной книги. Отмечу сразу, в этой работе мы не будем касаться динамической (just in time, JIT) компиляции, бинарной трансляции и всего, что с ними связано. JIT-компиляторы – это, несомненно, тоже компиляторы в узком смысле. Но речь дальше пойдет исключительно про статические компиляторы. Многие из тех алгоритмов, которые будут рассмотрены ниже, вполне применимы и используются в системах динамической компиляции, но там есть сложные детали и нюансы, которые мне хотелось бы опустить. Будем считать, что здесь и далее я буду иметь в виду исключительно статический компилятор и исключительно такой, который в итоге осуществляет трансляцию с языка программирования общего назначения в язык ассемблера целевой архитектуры.

Центральной идеей, вокруг которой в наше время строится практика оптимизаций в компиляторах, является идея промежуточного представления со статически единственными присвоениями (static single assignment, SSA). В этой книге мы пройдём путь от изобретения SSA к его построению, далее к использованию и далее к разрушению. Само это представление будет строго определено в разделе 3.1. Почему так поздно? Дело в том, что SSA исторически возникла очень поздно и есть гигантский пласт доисторической теории компиляторов, которые ей предшествовали и из которых она выросла. Необходимо минимальное знакомство с фундаментальными концепциями, предшествовавшими SSA, чтобы понимать откуда что растёт.

Структура книги довольно проста. Первая глава будет посвящена обзору современных систем компиляции (они также известны как тулчейны). В процессе я рассмотрю устройство компилятора и его окружение с высоты птичьего полёта. Эту главу можно считать общеобразовательной, и я рекомендую с ней ознакомиться даже если вы не будете далее читать эту книгу и вообще не собираетесь изучать оптимизации в компиляторах. Знание об устройстве и функциях современных компиляторов полезно для любого их пользователя, то есть для каждого программиста. Вторая глава будет посвящена практикам оптимизации, подводящим к идее SSA. Я введу понятие решёток, расскажу про рекуррентные соотношения для анализа потока данных и приведу примеры такого рода оптимизаций. Третья глава будет посвящена построению SSA. В теоретическом смысле она будет самой интересной и математически нагруженной.

Мы посмотрим на самые разные графы и научимся их трансформировать и строить над ними интересные структуры данных. Четвёртая глава будет посвящена базовым оптимизациям над SSA, пятая – цикловым (она по объёму будет вдвое больше четвёртой) и шестая – межпроцедурным оптимизациям. В седьмой главе мы научимся выбирать инструкции, разрушать SSA и поговорим о распределении регистров и о планировании исполнения. Седьмая глава будет тоже очень большой.

Автор этой книги является (в литературном смысле) в основном компилятором по отношению к её источникам. В этой книге будет мало нового и самостоятельного материала, и я почти не буду хвастаться своими исследованиями или исследованиями моих студентов. В основном я постараюсь аккуратно и точно изложить теорию оптимизирующей компиляции с учётом указанных выше ограничений. Книга написана от первого лица, но иногда я буду говорить во множественном числе – в основном, когда хочу подчеркнуть, что ожидаю вовлечения читателя в рассуждения. То есть во фразах "а теперь мы могли бы задаться вопросом", я пишу "мы", подразумевая нас с вами, уважаемый читатель. Когда я хочу сказать что-то отстранённое о себе, или сообщить какой-то факт, который от первого лица прозвучит странно, я буду использовать третье лицо как в начале этого абзаца.

Немного про используемые слова и термины. Такие слова как драйвер, тулчейн, трансляция и линковка являются англицизмами. В случае со словом "трансляция", этот англицизм к тому же неприятно пересекается с русским словом, обозначающим передачу звука и изображений на дальнее расстояние. Увы, в этой книге будет неизбежно много англицизмов. Автор является практикующим программистом и крутится в мире профессионального сленга. У меня не повернётся язык сказать "компоновщик" вместо "линкер" или "система компиляции" вместо "тулчейн".

Очень часто я буду вводить новые слова, образованные от англицизмов. Например, англицизм "спилл слот" (spill slot), обозначает временное хранилище в памяти для содержимого регистра. Образованное от него слово "спи́ллить" в свою очередь означает процесс сохранения компилятором регистров в эти слоты. Вводя такого рода новообразования, я постараюсь каждый раз подробно объяснять, что я имею в виду. Иногда я буду пользоваться английскими аббревиатурами вроде SSA или CFG. Опять-таки они являются частью профессионального сленга, а соответствующие им русские эквиваленты СЕП (статическое единственное присвоение) или ГПУ (граф потока управления) будут смотреться непонятно и инородно для специалиста, а у новичка сформируют неправильные

языковые привычки, которые помешают в дальнейшем профессиональном общении.

Автор имеет определённые языковые предпочтения: много лет используя и преподавая языки C и C++ (и разрабатывая компиляторы в основном на них и для них), я к ним в некотором роде привык. Поэтому, когда говорю о "каком-то" языке программирования, не уточняя что это за язык, обычно я имею в виду что-то вроде C или C++: что-то статически типизированное, что-то компилируемое, что-то с явным управлением ресурсами, включая память и т.д. Увы, это исключает из этой книги интереснейшие алгоритмы оптимизирующей компиляции функциональных языков: не ожидайте здесь всякой дефорестации и прочего. Я считаю, что этому надо было бы посвящать отдельную книгу.

Тот язык псевдокода, на котором я буду описывать компиляторные алгоритмы, также будет подозрительно похож на современный высокоуровневый C++ (без некоторых технических деталей, выкинутых для краткости). Надеюсь, он окажется для читателя интуитивно понятен. Но будьте с этим псевдокодом аккуратны: он не компилировался, не тестировался и служит только для иллюстрации идей. Настоящие реализации в компиляторах сильно отличаются педантичностью рассмотрения краевых случаев и краткость кода в них никто не считает достоинством. Десяток строчек псевдокода алгоритма, с учётом реальности, может стать тысячами строк обработки в конкретной реализации. Сам автор, читая литературу по компиляторам, иногда встречался с тем, что авторы отводили целую главу для описания сложного развитого псевдокода, а потом публиковали на нём процедуры на два-три разворота. При этом этот псевдокод также не исполнялся, не тестировался, так что взять как есть и просто использовать его всё равно было нельзя, а понять было куда сложнее. Поэтому я в псевдокоде стараюсь быть настолько кратким и ясным, насколько возможно, иногда сознательно опуская важные для практических реализаций оптимизации и проверки.

По опыту преподавания курса, лежащего в основе этой книги, оптимизации в компиляторах вызывают большой интерес и их довольно легко понять. Но даже лучший материал выветривается из головы без заданий и упражнений. Книга сопровождается двумя видами задач. Во-первых, заданиями непосредственно в тексте, которые позволяют что-то быстро попробовать и посмотреть и, во-вторых, несколько более сложными упражнениями в конце каждой главы.

Глава 1

Тулчейны

First we build the tools, then they build us.

– Marshall McLuhan

Так исторически сложилось, что компиляторы не работают в изоляции. На практике пользователь вообще редко запускает компилятор. И gcc и clang и даже cl это просто программы-оболочки (также говорят "драйверы"), которые запускают из-под себя не только компилятор, но и много что ещё. Конечно, пользователя интересует результат: превращение текста программы в исполняемый файл. Но именно для достижения такого результата одного только компилятора мало. Когда в обыденном языке мы говорим "компилятор gcc" мы обычно имеем в виду как раз ту самую программу-оболочку, под которой скрывается GNU Toolchain – громадный и разветвлённый набор компонентов и библиотек.

Если вы интересуетесь именно компилятором, то очень важно знать и понимать его место в тулчейне. И даже если рассматривать компилятор отдельно, он занимается не только оптимизациями. Поэтому не менее важно знать место оптимизатора в компиляторе. Общий ландшафт вокруг компиляторов сформирован самыми разными инструментами разработки – ассемблерами, линкерами, отладчиками, профилировщиками. Конечно, мне не хватит места поговорить о них всех. То немногое о чём я успею здесь рассказать, как мне кажется, важно знать не только если вы разрабатываете компиляторы, но даже если вы просто пользуетесь ими.

Со следующей главы мы углубимся в оптимизатор, а пока что давайте попробуем взлететь повыше и увидеть весь ландшафт и весь контекст,

в котором будет происходить всё то, о чём мы поговорим позже в этой книге.

1.1 Общий обзор

Давайте начнём с простого вопроса. Вы написали какую-то программу, например такую, как на листинге 1.1 и запускаете компилятор (на самом деле – драйвер компиляции) `gcc` чтобы получить исполняемый файл, как показано на том же листинге.

```
$ cat hello.c
#include <stdio.h>
int main() {
  printf("%s\n", "Hello, world!");
}
$ gcc hello.c -o hello.x
$ ./hello.x
Hello, world!
```

Листинг 1.1: Пример компиляции простейшего приложения

Прежде чем читать дальше постарайтесь перечислить все программы, которые будут при этом вызваны.

Задание 1.1. *Какие программы будут вызваны при вызове gcc, показанном на листинге 1.1?*

Вы можете посмотреть в справочные материалы по GNU Toolchain чтобы понять какие из его опций предназначены для того, чтобы открыть вам детали происходящего под капотом.

```
$ gcc hello.c --verbose -save-temps
cc1 -E -quiet -v -imultiarch x86_64-linux-gnu hello.c \
   -mtune=generic -march=x86-64 -fpch-preprocess -o hello.i
cc1 -fpreprocessed hello.i -quiet -mtune=generic \
   -march=x86-64 -o hello.s
as -v --64 -o hello.o hello.s
collect2 crti.o hello.o -lgcc -lc -lgcc crtn.o -o a.out
```

Листинг 1.2: Детали компиляции приложения

Правильный ответ: будут вызваны компилятор `cc1` ассемблер `as` и линкер `ld`. Убедиться в этом можно, модифицировав строчку вызова и

добавив опции для детального отображения процесса компиляции и сохранения временных файлов. На листинге 1.2 приведена немного упрощённая картинка, в реальности вы увидите гораздо больше визуального мусора, установки переменных окружения и всего такого.

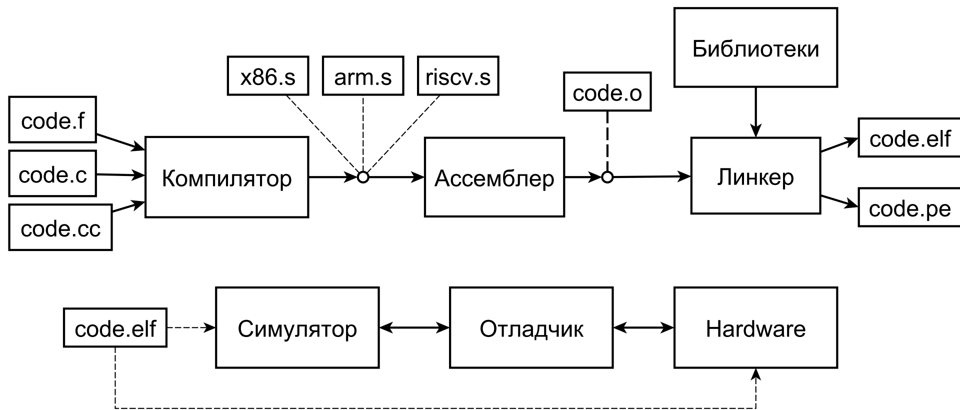

Рис. 1.1: Упрощённая схема тулчейна

Упрощённая схема тулчейна приведена на рис. 1.1. Основными её частями являются компилятор ассемблер и линкер. Кроме того показано, что получившийся бинарник (binary file, сленговое выражение для файла, содержащего не текст, а двоичный код) можно отлаживать и запускать на железе или на симуляторе. Запуски на симуляторе могут быть полезны при кросс-компиляции, например если вы на одной архитектуре компилируете код для исполнения на другой.

Задание 1.2. *Скачайте кросс-тулчейн для любой архитектуры по вашему выбору и попробуйте скомпилировать программу и запустить её под симулятором.*

Итак, под общей крышей программы-драйвера в основном живут компилятор ассемблер и линкер. Настало время посмотреть на эти основные части поближе.

1.2 Компиляторы

If you lie to the compiler,

it will get its revenge.

– Henry Spencer

Компилятор это основа, главная и самая сложная часть тулчейна. Как можно судить по рис. 1.1 компилятор решает задачу, кажущуюся простой – он берёт программу на высокоуровневом языке и строит из неё ассемблер. Чтобы показать, что эта задача не тривиальна, давайте рассмотрим довольно простой код на листинге 1.3.

```
int sum(int n) {
  if (n == 1)
    return 1;
  return n + sum(n - 1);
}
```

Листинг 1.3: Сумма чисел от 1 до n

Как компилятор должен транслировать эту программу? Он может просто построить ассемблер как написано, но такой ассемблер будет не слишком хорош. Улучшая этот код, компилятор может заменить рекурсию на цикл. Тогда ему придётся ввести дополнительную переменную для счётчика цикла. Возможная трансформация такого рода показана на листинге 1.4.

```
int possible_sum1(int n) {
  int i, s = 1;
  for (i = n; i != 1; --i)
    s += i;
  return s;
}
```

Листинг 1.4: Возможная трансформация суммы чисел компилятором

Наконец компилятор может просто вернуть известное значение суммы, как на листинге 1.5.

```
int possible_sum2(int n) {
  return n * (n + 1) / 2;
}
```

Листинг 1.5: Ещё более агрессивная трансформация

Проблема в том, что этот вариант не эквивалентен листингу 1.3 для отрицательных аргументов. С другой стороны, если число изначально отрицательное, то и исходный код ведёт себя некорректно, переполняясь вниз. Вариант с циклом с листинга 1.4 сохраняет ошибочное поведение ошибочным причём точно таким же. Но он медленней, так как делает цикл там, где можно просто сразу вычислить значение для корректного случая. Наконец компилятор может выбрать гибридный вариант, оставив сокращённое вычисление при корректном поведении и оставив всё как есть при ошибочной семантике, как на листинге 1.4.

```
// compiler can rewrite things even this way
int possible_sum3(int n) {
  if (n > 0)
    return n * (n + 1) / 2;
  return n + possible_sum3(n - 1);
}
```

Листинг 1.6: Компромисс между оптимизацией и сохранением семантики

Представьте объём анализа, который нужен для принятия такого решения. На момент написания этой книги два разных промышленных тулчейна – clang и gcc строят два разных варианта ассемблера для такого, казалось бы, простого примера. И даже просто глядя на сгенерированный ассемблер, пример которого приведён на листинге 1.7, мы иногда испытываем сложности с тем, чтобы соотнести его с исходным кодом.

```
possible_sum3(int):
    mov    eax, 1
    cmp    edi, 1
    je     .L1
    lea    eax, [rdi - 1]
    lea    ecx, [rdi - 2]
    imul   eax, ecx
    lea    edx, [rdi - 3]
    imul   rdx, rcx
    shr    rdx
    add    eax, edi
    sub    eax, edx
    inc    eax
.L1:
    ret
```

Листинг 1.7: Сумма чисел на ассемблере x86