Компиляторные технологии

Майер Алексей

Содержание

Лекция 0. Вводная лекция [ТРЕБУЕТ ДОРАБОТКИ]

О курсе

Лектор

Иван Иванович

Элементы контроля

14 лекций, 14 семинаров

4 кр и 6 домашек

Домашки сдаются через ejudje + сдача семеру

4 контрольные в сумме дают 100 баллов

Блокируются контрольные суммы или каждая в отдельности – неизвестно (на усмотрение лектора)

1. 20
2. 30
3. 30
4. 20

6 домашек тоже 100 баллов, весят:
1. 5
2. 10
3. 15
4. 15
5. 15
6. 40

Итоговая оценка это

$$0,4 * О_{кр} + 0,6 * О_{дз}$$

Литература

https://github.com https://github.com https://github.com

Транслятор, компилятор, интерпретатор

Это 3 программы. В чём их отличие?

Транслятор преобразует один язык на другой язык.

Компилятор преобразует программу в машинный код.

Интерпретатор исполняет исходную программу.

Гибридный компилятор сначала транслирует код в байт-код, а затем исполняет.

Например в *java*, сначала транслятор преобразует код в байт-код, а затем *java virtual machine (JVM)* исполняет байт-код.

Виды компиляторов.

AOT (ahead of time) – полная траснялция программы до её выполнения.

JIT (just in time) – выполняет динамическую трансляцию промежуточного кода (байт-кода) в исполняемый код целевой архитектуры.

Гетерогенный компилятор

Гетерогенный компилятор – выполняет трансляцию программы с частями на разных языках (диалектах) для разных целевых архитектур.

Примеры:

Код разделяется на код хоста и код устройства

1. C/C++/Fortran + директивы для расспаралеливания
2. C/C++ + NVIDIA CUDA
3. C/C++ + OpenCL
4. SYCL

Кросс-компиляция – трансляция программы под архитектуру процессора, отличающуюся от архитекуты процессора, на котором выполняется компиляция.

1. Сборка на x86-64 для ARMv8
2. Сборка на x86-64 (little-endian) для (big-endian)
3. Сборка на x86-64 для

Если архитектура хоста не совпадает с архитектурой целевого устройства.

Почему интересны компиляторы?

Компилятор – большая и сложная программа.

Там куча интересных алгоритмов.

1.
2.
3.
4.
5.
6.

Набор инструметов для программирования (Toolchain)

main.cpp → компилятор

Мы будем изучать только блочок компилятор

Заглянем в компилятор

Структура компилятора

Лексический анализ, препроцессинг, синтаксический анализ, семантический анализ

Чем отличается абстрактное синтаксическое дерево от конкретного синтаксическое дерево?

AST – синтаксическое дерево в терминах языковых конструкций

Конкретного синтаксическое дерево – в терминах результирующего кода

Такое AST поступает в оптимизирующий компилятор

Инструкция

Задачи компилятора

Frontend
1. ....
2. Построение HIR

Краткий обзор процесса компиляции

В чем разница между токеном и лексемой? Лексема – последовательность символов. Токен это уже определенная последовательность символов, чем она является

Нисходящий анализ в промышленных компиляторах.

Семантический енализтор – условно проверяет совместимость типов, корректность сгенерированного кода.

Лекция 1. Тема лекции [ТРЕБУЕТ ДОРАБОТКИ]

Представления программ.

Абстрактное синтаксическое дерево.

AST – структурированное представление программы в терминах языковых конструкций.

1. АСД пригодно для автоматического анализа
2. Содержит высокоуровневую семантическую информацию
3. Удообно для интерпретирования
4. Неудобно для выполнения сложных оптимизирующий преобразований и генерации кода.

Высокоуровневая информация прокидывается в промежуточное (Middle IR. MIR) что не очень удобно, но иначе нельзя, а так удобнее анлизировать.

Последовательность инструкций

LLVM IR – промежуточное представление программы. Каждая интсрукция обладает регулярной формой.

Промежуточное представление MIR

У нас будет учебное промежуточное представление, чтобы показать

1. x = y OP Z
2. x = OP y
3. x = y
4. x[i] = y
5. x = y[i]
6. goto
7. if ...

8. Функции
    1. function f(p_1, p_2, ..., p_n)
    2. call f(p1, p2, ..., p_n)
    3. ret t

——— Пример функции быстрой сортировки ———

——— Чуть более удобное представление ———

Граф потока управления – набор базовых блоков

Базовый блок – последовательность инструкций не содержащая инструкций передачи управления (условных, безусловных, возврата из функции).

Также в базовый блок можно войти только в первую инструкцию. В другую – нельзя.

——— Алгоритм построения графа потока управления ———

Вход: последовательность трехадресных инструкций

Выход: список баховых блоков

Метод:
1. Строится упорядоченное множество НББ (начал базовых блоков)
2.
3.

——— Промежуточное представление MIR ———

——— Локальная оптимизация (оптмизация базовых блоков) ———

Оптимизации бывают:

1. Локальные (На уровне BB – Basic block)
2. Глобальные (На уровне CFG – control flow graph)
3. LTO (возможно успеем в конце курса)

Локальная оптимизация – устранение избыточностей в рамках одного базового блока.

——— Пример локальной Оптимизации ———

——— Постановка задачи локальной оптимизации ———

Базовый блок – множества:

B = <P, Input, Output>

P – последовательность инструкций.

Input – множество переменных, определенных до блока B

Output – множество переменных, используемых после выхода из блока B

Оптимизация, в ББ это:

- Удаление общих подвыражений.
- Удаление мертвого кода
- Сворачивание констант

- Изменение порядка инструкций, там, гле это возможно, чтобы сократить время хранения временного значения на регистре.

Представление базового блока в виде ориентированного ациклического графа

$$a + a * (b - c) + (b - c) * d$$

AST -> DAG.

В DAG – каждое значение представляется только раз. Узлы представляющие в АСД разные значения – склеиваются.

$$a = a + y * (b + (y - z) * b) + (y - z) * b$$

Как мы это программно решаем? Делаем хэш таблицу поддеревьев. (ХЭШИ!)

Метод нумерации значений (SSA алгоритм, но внутри одного блока)

Алгортм построения ОАГ для базового блока B

Представление ОАГ в виде таблицы значений. Пример 1

Присоединённая переменная – переменная в левой части присваивания.