Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования

“Московский государственный технический университет имени Н.Э. Баумана” (МГТУ им. Н.Э. Баумана)

Факультет: Информатика и системы управления
Кафедра: Программное обеспечение ЭВМ и информационные технологии

Курсовая работа

по теме

“Бекенд компилятора языка C–”

Студент: Гребенюк Александр, ИУ7-29

Москва 2015

Реферат

Отчёт 10с., 0 табл., 0 листингов, 4 источника, 0 приложений.
Компиляторы, LLVM, JIT, Haskell.

Предмет работы составляет разработка бекенда компилятора упрощённой версии языка C– с целью изучения его механизмов работы.

Цель работы - создание программного обеспечения, осуществляющего трансляцию абстрактное синтсаксического дерева в код LLVM IR с последующей JIT-компиляцией.

Было разработано ПО, осуществляющее трансляцию абстрактное синтсаксического дерева в промежуточное представление кода виртуальной машины LLVM – LLVM IR и JIT-компиляцию полученного кода. Был изучен механизм линковки сторонних библиотек для обеспечения работы инструкций ввода-вывода языка C–.

В результате было разработан бекенд компилятора C– и приведены примеры трансляции АСТ в LLVM IR.

Оглавление

1. Введение

В настоящей работе в рамках курса “конструирование компиляторов” реализуется бекенд компилятора упрощённой версии языка C–. В работе приводится описание трансляции токенов абстрактного синтсаксического дерева в LLVM IR и примеры обнаруживаемых ошибок компиляции. Также приводится описание реализации JIT-компилятора.

2. Абстрактное синтаксическое дерево

Выходными данными фронтенда компилятора является синтаксическое дерево следующего вида:

```
data Type =
    Number
    | Reference Int
    | Void

data Vardecl = Vardecl String Type

data Declaration =
    VD Vardecl
    | Funcdecl Type String [Vardecl] Statement
    | Extdecl Type String [Vardecl]

data Statement =
    Complex [Vardecl] [Statement]
    | Ite Expression Statement (Maybe Statement)
    | While Expression Statement
    | Expsta Expression
    | Return (Maybe Expression)

data Expression =
    ConstInt Int                        -- 7
    | ConstArr [Int]                    -- [7,8,9]
    | Takeval Expression
    | Takeadr String
    | Call String [Expression]
    | Assign [Expression] Expression  -- adr1 = adr2 = adr3 = 7
    | BracketOp Expression Expression
```

Абстракцией самого верхнего уровня является Declaration. Код C– модуля де-факто представляет собой лес корневых элементов Declaration, отражающих инструкции объявления глобальных переменных и функций языка C–.

3. Трансляция

3.1. Переменные

Переменные в AST представляются абстракцией Vardecl и имеют имя и тип. Глобальные переменные объявляются вне тел функций и представляются абстракцией VD.

Следующий пример LLVM IR объявляет глобальную переменную “gvar” типа “i32” с начальным значением 0 и глобальный i32 массив “garr” размера 10, инициализированный нулями.

```
@gvar = common global i32 0
@garr = common global <10 x i32> zeroinitializer
```

LLVM не предоставляет возможности объявить неинициализированную глобальную переменную, поэтому, несмотря на отсутствие поддержки инициализации переменных при объявлении в AST, компилятор производит инициализацию переменных нулевыми значениями, в зависимости от типа.

Ключевое слово “common” означает тип линковки переменной. Согласно документации, именно этот тип линковки используется для неинициализированных глобальных переменных в C-подобных языках. Далее так же будет использован тип “external” при объявлении внешних(библиотечных) функций, реализующих ввод-вывод.

Данная реализация компилятора поддерживает локальные переменные, однако производит выделение памяти на стеке а не в куче.

Пример объявления и инициализации локальной переменой:

```
//C-- some function scope:
int j;
j = 'Z';
```

```
; LLVM IR
%2 = alloca i32
store i32 90, i32* %2
```

3.2. Функции

Функции в AST представляются абстракцией Funcdecl и имеют: тип возвращаемого значения, имя, список параметров и тело.

Для удобства трансляции в AST была добавлена абстракция Extdecl, представляющая собой Funcdecl без тела, отражающая объявление внешних функций. Т.к. упрощенная версия языка C– не поддерживает инструкций объявления внешних функций, Extdecl невозможно получить путем синтаксического разбора, т.е. добавление двух необходимых Extdecl(для реализа-

ции инструкций “read” и “write” языка C–) производится непосредтсвенно в коде компилятора, после этапа генерации АСТ.

Пример объявления внешней функции “void putChar(int)”, отвечающей инструкции “write”, и функции языка C–:

```
//C-- instruction ``write'', uses external putChar.
void print(int sym)
{
    write sym;
    return;
}
```

```
; LLVM IR
declare void @putChar(i32)

define void @print(i32 %sym) {
entry:
  %0 = alloca i32
  store i32 %sym, i32* %0
  %1 = load i32* %0
  call void @putChar(i32 %1)
  ret void
}
```

3.3. Выражения

АСТ поддерживает следующие типы выражений:

- Целочисленная константа
- Целочисленный массив
- Вычисление значения выражения
- Вычисление значения переменной
- Вызов функции
- Множественное присваивание (“a = b = 6”)
- Индексация в массиве

Ввиду особенности реализации фронтенда, по умолчанию АСТ поддерживает арифметические операции языка C– посредством вызова “Call <operator> [Expression]”. Т.к. LLVM поддерживает множество арифметических операций над различными типами данных, и язык C– не является семантически объектно-ориентированным, т.е. не поддерживает перегрузку операторов, было принято решение транслировать Call арифметических операторов в существующие LLVM-инструкции вместо генерации кода поддержки операторов.

LLVM поддерживает объявление, в числе прочих, целочисленных констант и многомерных константных массивов. Вычисление значения переменной проихводится при помощи инструкции “load”, вызов функции – при помощи “call”.

Пример сложного кода на C– и его трансляции в LLVM IR:

```
//C--: m, k -- local, gvar -- global, print -- declared function
m = k = gvar = gvar + 1;
print(m);
```

```
; LLVM IR
%7 = load i32* @gvar
%8 = add i32 %7, 1
store i32 %8, i32* @gvar  ; gvar = gvar + 1
%9 = load i32* @gvar
store i32 %9, i32* %3     ; k = gvar
%10 = load i32* %3
store i32 %10, i32* %4    ; m = k
%11 = load i32* %4
call void @print(i32 %11) ; print(k)
```

3.4. Ветвление

ACT поддерживает следующие инструкции ветвления:

- while
- if-then
- if-then-else

Ветвление в LLVM производится при помощи создания нескольких блоков кода и объяления условных и безусловных переходов между ними. Каждый блок, кроме первого и последнего должен иметь хотя бы один вход и один выход.

while

```
; LLVM IR
while.cond:                                       ; preds = %if.exit, %entry
  ; ...
  br i1 %6, label %while.loop, label %while.exit

while.loop:                                       ; preds = %while.cond
  ; ...
  br label %while.cond

while.exit:                                       ; preds = %while.cond
  ; ...
```

if-then, if-then-else

```
; LLVM IR
br i1 %13, label %if.then, label %if.else
if.then:                                          ; preds = %while.loop
  ; ...
  br label %if.exit

if.else:                                          ; preds = %while.loop
  ; ...
  br label %if.exit

if.exit:                                          ; preds = %if.else, %if.then
  ; ...
```

Инструкция ветвления с отсутствующим блоком “else” эмулирует поведение полной инструкции, т.е. создает блок “if.else” из единственной терминальной инструкции безусловного перехода в конец цикла.

3.5. Возврат управления

В данной версии языка C– инструкция возврата управления может присутствовать только в конце тела функции. В LLVM IR для возврата управления используется инструкция “ret”.

Примеры:

```
; function1
; ...
ret i32 %14
; function2
; ...
ret void
```

4. Реализация

Данный проект был реализован на языке Haskell с использованием библиотек llvm-general-pure и llvm-general, предоставляющих привязки для языка Haskell к C++ LLVM - библиотекам, поставляемым разработчиками LLVM и находящимися в открытом доступе.

Для реализации инструкций “read” и “write” была написана динамически линкуемая библиотека cbits.so.

```
#include <stdio.h>
#include <stdint.h>

void putChar(int32_t X) {
  putchar((char)X);
  fflush(stdout);
}

int32_t getChar()
{
  return getchar();
}
```

Линковка и сборка Haskell-кода производится при помощи утилиты “ghc” (Glasgow Haskell Compiler, Version 7.10.1)

JIT-компиляция производится при помощи вызова LLVM ExecutionEngine, предоставляемого библиотекой llvm-general.