Compiler II
CS 3423 - Aug 2019
Mini Assignment 1:

An Introduction to the LLVM Infrastructure, AST,

IR and Compiler Options

INSTRUCTOR: Dr. Ramakrishna Upadrasta

Report By:
Vijay Tadikamalla
CS17BTECH11040

भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

AST Structure

General Properties

- The top-level declaration in every AST starts with the **translation unit declaration** followed by some typedefs.
- A lot of information is stored in the AST like the line number, column number, type of the statement, used/unused variables.

```
-VarDecl 0x55f03d64f0d0 <col:9, col:17> col:13 used i 'int' cinit
```

- Storing this information helps in various ways like:
    - Semantic analysis
    - Throwing error messages with line and column numbers.
    - Showing type-incompatibility error messages.
    - Throwing warnings like implicit type conversion, unused variable, etc.
- The following command dumps the AST of the code in a serialized form

```
lang -Xclang -ast-dump -fsyntax-only filename.cpp
```

- This serialized version of AST shows us the **scope of each block of code**
    - All the global functions and variables are children of the root node.
    - Each inner block is the child node of the immediate outer block.
- All constants and expressions are passed in an unreduced form.

Types of Nodes

The following are some of the nodes which can be observed in the AST

1. **If-else node**
    - A single ***if*** *statement* is represented as follows:

```
IfStmt 0x55f03d64f0d0 <line:2:5, line:20:5>
```

- **'has_else'** indicates that the ***if*** statement also has an ***else*** part.

```
IfStmt 0x55f03d64fd47 <line:9:13, line:16:13> has_else
```

2. **For node**
   It has four children which represent the declaration, predicate for loop termination, increment, and body of loop respectively.
   Any missing value is represented as **<<<NULL>>>** in the AST dump.

3. **While node**
   It has two children which represent the predicate for loop termination, and the body of loop respectively.

4. **RecordDecl (class/struct/union) node**
   Some of its child nodes are DefinitionData, FieldDecl, AccessSpecDecl, CXXMethodDecl.

5. **Function declaration node**
   It has two children ParamVarDecl and CompoundStmt (body).

AST Traversals

FrontEnd Action

- ASTFrontendAction is an interface that allows the execution of user specific actions as part of the compilation.
- It has the following signature
  ```
  std::unique_ptr<clang::ASTConsumer> CreateASTConsumer(
  clang::CompilerInstance &Compiler, llvm::StringRef InFile)
  ```

ASTConsumer

- ASTConsumer is an interface that allows writing generic actions on AST.
- ASTConsumer provides many different entry points, such as HandleTranslationUnit, which is called the ASTContext for the translation unit.
- It has the following signature
  ```
  void HandleTranslationUnit(clang::ASTContext &Context)
  ```

RecursiveASTVisitor

- RecursiveASTVisitor is an interface provided by LLVM that is used to extract the relevant information from AST.
- The Visitor class implementing the RecursiveASTVisitor interface provides methods that are invoked when a certain type of node is visited.

- It has the following signature bool VisitTYPE(TYPE *node_ptr)

Error messages

LLVM classifies errors into two categories:

- **Recoverable error -** Recoverable errors represent an error in the program's environment, for example a resource failure (a missing file, a dropped network connection, etc.), or malformed input.
- **Programmatic error -** Programmatic errors are violations of program invariants or API contracts, and represent bugs within the program itself. Our aim is to document invariants, and to abort quickly at the point of failure (providing some basic diagnostic) when invariants are broken at runtime.

The following are some sample examples of error message:

```
assert(Ty->isPointerType() && "Can't allocate a non-pointer type!");
assert((Opcode == Shl || Opcode == Shr) && "ShiftInst Opcode invalid!");
assert(idx < getNumSuccessors() && "Successor # out of range!");
assert(V1.getType() == V2.getType() && "Constant types must be identical!");
assert(isa<PHINode>(Succ->front()) && "Only works on PHId BBs!");
```

LLVM-IR

LLVM-IR is the **core of LLVM**. It is a low-level intermediate representation used by the LLVM compiler framework. It looks like a low-level programming language similar to assembly (RISC-like three address code).

The LLVM-IR has **three** different forms :
1. In-memory forms
2. Binary - An on-disk bitcode representation (suitable for fast loading by a JIT compiler)
3. Text - A human-readable assembly language representation.
This allows LLVM to provide powerful intermediate representation for efficient compiler transformations and analysis.
The three different forms of LLVM are all equivalent and LLVM provides constructs to interconvert them into each other.

Report on LLVM-IR

1. LLVM-IR has a **type system** that consists of
    a. **Simple types** (Fixed size) like
        i. **Label:** if.then, if.else
        ii. **Void**
        iii. **Null**
    b. **Derived types** like
        i. **Integers:** They can be of arbitrary sizes like ***i8, i32, i64, i112, etc.***
        ii. **Floats:** They can be of size like ***16, 32,64, 80, 128 bit.***
        iii. **Array:** [40 x i32], [2 x [3 x [4 x i16]]]
        iv. **Structure:** { [40 x i32], i32 }
2. It allows **type-casting**. So, it can be used to express weakly-typed languages like C. Some instructions like **'bitcast'** are used to convert value to another type without changing any bits.
3. Front-ends for type-safe languages like Haskell can also be implemented by using a type-safe subset of LLVM.
4. Any LLVM program has the following structure:
    a. It is a collection of **Modules.** A **module** can be thought of as a unit of compilation.
    b. A Module consists of **Functions** and **Global variables.**
    c. Each function consists of **Basic block and arguments.**
    d. A basic block is a **list of instructions.**
5. **Identifiers:** They come in two basic types: **global and local**. Global identifiers (functions, global variables) begin with the **('@')** character. Local identifiers (register names, types) begin with the **('%')** character.
6. A module may specify a **target-specific data layout** string that specifies how data is to be laid out in memory. The layout specification consists of a list of specifications separated by the minus sign character **('-')**. Each specification starts with a letter and may include other information after the letter to define some aspect of the data layout.
7. A module may also specify a **target triple** string that describes the **target host**.

```
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"
```

8. To generate the IR of any c/cpp file we can use the following command:

```
clang -S -emit-llvm filename.cpp -o output.ll
```

9. For generating LLVM-IR (for Specific Target use flag **-target)**

```
clang -S -target sparc -emit-llvm file.c -o file.ll
```

10. LLVM instruction set consists of several classes of instructions like
    a. Memory instructions like store, load, etc.
    b. Binary instructions like add, fadd, sub, fsub, etc.
    c. Terminator instructions like ret, br, switch, etc.
    d. Vector instructions like extractelement, insertelement, shufflevector, etc.

Assembly language

Name mangling means encoding of external names, i.e, the variables, functions which can be accessed outside the module. This process is essential because at assembly level when we merge/combine multiple source files, there is always a chance of a repetition of variable/function names.

To prevent this problem, LLVM follows a method similar to the **Itanium C++ ABI** method of mangling. The following BNF rules show the general structure of this method:

```
<mangled-name> ::= _Z <encoding>
               ::= _Z <encoding> . <vendor-specific suffix>
<encoding> ::= <function name> <bare-function-type>
           ::= <data name>
           ::= <special-name>
```

These rules show that a name is mangled by prefixing "**_Z**" to an encoding of its name and in the case of functions its type (to support overloading).

Name mangling is not generally required because C because it does not support function overloading. Instead, name mangling is used in some cases to provide additional information about a function.

Compiler toolchain and options

LLVM tools are the executables built out of the libraries. They form the main part of the user interface. The following are some of the LLVM tools:

- **llvm-ar:** The archiver produces an archive containing the given LLVM bitcode files
- **llvm-as:** assemble a human-readable .ll file into bitcode
- **llvm-dis:** disassemble a bitcode file into a human-readable .ll file
- **opt:** run a series of LLVM-to-LLVM optimizations on a bitcode file
- **llc**: generate native machine code for a bitcode file
- **lli:** directly run a program compiled to bitcode using a JIT compiler or interpreter
- **llvm-link:** link several bitcode files into one
- **clang:** C, C++, Object C front-end for LLVM

Here are some commands that show how to use the above-mentioned tools

1. Compile a C file into a native executable and run it:

```
clang filename.c  && ./a.out
```

2. Generate assembly code

```
clang -S file.c -o file.s
```

3. For generating assembly for Specific Target, use flag **-target**

```
clang -target sparc -S file.c -o file.s
```

4. Compile a C file into an LLVM bitcode file and run it:

```
clang -O3 -emit-llvm filename.c -c -o filename.bc && lli filename.bc
```

5. Compile a bitcode program to native assembly using the LLC code generator:

```
llc file.bc -o file.s
```

6. We can play with **llvm-as** and **llvm-dis** and we observe no/some minor changes in a **.ll file** after a round-trip.

```
llvm-as filename.ll
llvm-dis filename.bc -o filename_round_trip.ll
```

```
diff  filename.ll filename_round_trip.ll
```

7. Clang provides different level of optimizations. Different flag provide different type of optimization **-O0**, **-O1**, **-O2**, **-O3**, **-Ofast**, **-Os**, **-Oz**, **-Og**, **-O**, **-O4.**
8. **-Os, -Oz** are used to reduce code size, -Og is used to improve improve debuggability.

Kaleidoscope

Lexer

- The lexer returns tokens [0-255] if it is an unknown character, otherwise, it will either be one of the Token enum values

```
enum Token {
  tok_eof = -1,

  // commands
  tok_def = -2,
  tok_extern = -3,

  // primary
  tok_identifier = -4,
  tok_number = -5,
};

static std::string IdentifierStr; // Filled in if tok_identifier
static double NumVal;             // Filled in if tok_number
```

- The lexer is a single function named **gettok()**.
- The gettok function is called to return the next token from standard input.
- It needs to recognize identifiers and specific keywords like “def” and “extern”.

Regex:

Identifier: [a-zA-Z][a-zA-Z0-9]*
Number: [0-9.]+

Parser

It has basic parsing mechanisms for:

- **Numeric literals -** A routine ParseNumberExpr() is called when the current token is a tok_number token.
- **Handling variable references and function calls -** A routine **ParseIdentifierExpr()** is called when the current token is a tok_identifier token. It has recursion and error handling
- **Arbitrary primary expression -** We need a helper function to determine what sort of Arbitrary primary expression it is. So a helper function ParsePrimary() is used here.
- **Binary Expressions -** Binary expressions are significantly harder to parse because they are often ambiguous. So the parser uses a technique called Operator-Precedence Parsing. This parsing technique uses the precedence of binary operators to guide recursion.
- **Handling of function prototypes -** Kaleidoscope uses ‘extern’ for both function declarations as well as function body definitions. So a routine ParsePrototype() is used here.

References

- https://itanium-cxx-abi.github.io/cxx-abi/abi.html#mangling
- https://en.wikipedia.org/wiki/Name_mangling
- https://cs.lmu.edu/~ray/notes/ir/
- https://llvm.org/docs/CodingStandards.html#assert-liberally
- http://clang.llvm.org/docs/IntroductionToTheClangAST.html
- http://clang.llvm.org/docs/RAVFrontendAction.html
- http://llvm.org/docs/GettingStarted.html#directory-layout