CS451 – Software Analysis

Lecture 19
The LLVM Tools

Elias Athanasopoulos
elathan@cs.ucy.ac.cy

LLVM in your VM

- LLVM is pre-installed in your VM
  - There is a directory called llvm-project
- The entire distribution of libraries and tools has been built from sources
  - The build is located at llvm-project/llvm/build

Building LLVM form scratch

- Install needed software
  - C/C++ compiler (gcc), git, Python, GNU make, Cmake
- Checkout LLVM

  `$ git clone` https://github.com/llvm/llvm-project.git
- Create a build directory and go there
- Create Makefiles

  `$ cmake -DLLVM_ENABLE_PROJECTS=clang -G 'Unix Makefiles' ..`
- Minimum size build

  `$ cmake -DLLVM_ENABLE_PROJECTS=clang -DCMAKE_BUILD_TYPE=MinSizeRel -G 'Unix Makefiles' ..`

LLVM versions

- List all versions

  ```
  $ git tag -l
  ```

- Checkout a specific version for building

  ```
  $ git checkout llvmorg-8.0.0
  ```

- Show current version

  ```
  $ ./build/bin/llvm-config --version
  13.0.0
  ```

Check the installation

```
$ bin/clang --version
```

clang version 13.0.0 (https://github.com/llvm/llvm-project.git d7b669b3a30345cfcdb2fde2af6f48aa4b94845d)

Target: x86_64-unknown-linux-gnu

Thread model: posix

InstalledDir: /home/elathan/llvm-project/llvm/build/bin

Compile a first program using Clang

- Compiling a C program using Clang is very similar to gcc

```
$ clang –Wall toy.c -o toy
$ ./toy
Hello World.
```

LLVM bitcode

- The strength of the LLVM framework is the intermediate representation form, known as LLVM IR

- There are three representations of the LLVM IR
  - One that resides in memory and is processed by LLVM passes
  - One binary form that can be stored on the disk (.bc)
  - One textual form that can be stored on the disk (.ll)

- There are tools to go from one representation to the other

Examples

- Produce .bc
```
$ clang -emit-llvm -c toy.c -o toy.bc
```
- Produce .ll
```
$ clang -emit-llvm -S toy.c -o toy.ll
```
- Going from one format to the other
```
$ llvm-dis toy.bc # generates .ll
$ llvm-as toy.ll  # generates .bc
```
- Execute .bc
```
$ lli toy.bc
Hello World.
```

Extracting part of the bitcode

```
$ llvm-extract -func=foo toy.bc  -o foo-fn.bc
$ llvm-dis foo-fn.bc
$ cat foo-fn.ll
; Function Attrs: noinline nounwind optnone uwtable
define dso_local void @foo() #0 {
  %1 = load %struct._IO_FILE*, %struct._IO_FILE** @stderr,
align 8
  %2 = call i32 (%struct._IO_FILE*, i8*, ...)
@fprintf(%struct._IO_FILE* %1, i8* getelementptr inbounds
([14 x i8], [14 x i8]* @.str, i64 0, i64 0))
  ret void
}
```

Manipulate the IR

```c
#include <stdio.h>

void print_number(void) {
    int number = 41;
    fprintf(stderr, "The answer of life is: %d.\n", number);
}

int main(int argc, char *argv[]) {
    print_number();
    return 1;
}
```

- Compile to .ll
- Change 41 to 42
- Assemble to .bc
- Execute .bc with lli

LLVM IR syntax

- Assume an LLVM file with some bitcode
  - For instance, the bitcode of the function foo() from toy.c
  - This is a module, which contain a series of functions, that contain a series of instructions
- Modules may contain additional data
  - Global variables, target data layout, external function prototypes, declaration of data structures

LLVM local variables

- Local values can be thought as h/w registers storing a value
  - They have a name starting with the token '%'

- Examples
  - 32-bit addition of %0 to %add, which can produce an overflow

```
%add = add nsw i32 %0
```

  - 32-bit addition of %6 with %7, which can produce an overflow, and the result is stored in %8

```
%8 = add nsw i32 %6, %7
```

LLVM IR instruction

- Each instruction is expressed in three-address format
  - One instruction with maximum two operands, and the result of the operation is stored in a third variable
- No value is reassigned
  - Each value can be easily traced back to the instruction that produced it, without complex data-flow analaysis
  - Useful for computing use-def chains and performing optimizations

Target host

- The module initially contains target information about type sizes and the architecture

  ```
  target datalayout = "e-m:e-p270:32:32-
  p271:32:32-p272:64:64-i64:64-f80:128-
  n8:16:32:64-S128"
  ```

  ```
  target triple = "x86_64-unknown-linux-
  gnu"
  ```

- The target is x86 (64-bit) with GNU Linux

- The target is little endian (letter 'e') and uses ELF ('m:e')

- Supports the following types given with the format
  ```
  type:=<size>:<abi>:<preferred>
  ```

Function declaration

- Defines a function with the name @add

- The function takes two arguments, %0 and %1, which are integers of 32 bits

- It returns an integer of 32 bits

- The function resolves to a symbol within the same linkage unit

```
define dso_local i32 @add(i32 %0, i32 %1) #0 {
...
}
```

Attributes

- The tag #0 specifies common compiler flags
    - attributes #0 = { noinline nounwind optnone
      uwtable "frame-pointer"="all" "min-legal-
      vector-width"="0" "no-trapping-math"="true"
      "stack-protector-buffer-size"="8" "target-
      cpu"="x86-64" "target-
      features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
      "tune-cpu"="generic" }

Basic blocks

```llvm
define dso_local i32 @add(i32 %0, i32 %1) #0 {
    %3 = alloca i32, align 4
    ...
9:
    %10 = load i32, i32* %3, align 4
 ...
13:
    %14 = load i32, i32* %3, align 4
   ...
15:
    %16 = load i32, i32* %5, align 4
    ret i32 %16
}
```

```c
int add(int a, int b) {
    int c = 0;
    if (a > b)
        c = a + b;
    else
        c = a;
    return c;
}
```

Allocation

- The `alloca` instruction reserves space on the stack frame
  - The amount of space is determined by the data type and the alignment is specified

    ```
    %3 = alloca i32, align 4
    ```

Homework

- Write a C program that uses a function that multiplies two numbers, with the prototype

  `int mymul(int a, int b)`

  - Compile the program using Clang and produce the IR
  - Change multiplication to addition and produce a native executable

- Modify the IR so that we have two functions, one for addition (`myadd`) and one for multiplication (`mymul`), but *without* modifying the C source

References

- LLVM Documentation
  - https://llvm.org/docs/index.html
- LLVM Command Guide
  - https://llvm.org/docs/CommandGuide/index.html