Clang v.s. LLVM

Moonzoo Kim
Software Testing and Verificaton (SWTV) group
CS Dept. KAIST

Comparison of Clang and LLVM

	Clang	LLVM
Pros	• **Source code information** (e.g., line/column number) is available • Clang supports **source-to-source transformation**	• Complex high-level language semantics are lowered to relatively **simple instructions** • An analysis tool using LLVM can be **programming language independent**
Cons	• A user should handle **complex C/C++ language semantics** (e.g., side effect, various AST node types)	• **Source code information is lost**
Application	• C's undefined behavior checker • Source code refactoring tool • Source code browser (e.g., Source Insight)	• Static analyzer for bug detection • Test generator • Runtime monitoring tool

An Example of Clang's Use Cases

- You need to use Clang to develop a checker for C/C++'s undefined behaviors in source code
  - Undefined behaviors in C code will be removed in transformed LLVM IR
  - Line 4 of C code containing an undefined behavior is transformed into well-defined LLVM instructions

C code

```
1 int example(){
2   int a = 1, b;
3   // Undefined behavior
4   b = a++ + ++a;
5   return b;}
```

LLVM bytecode (Simplified version)

```
define i32 @example() {
  store i32 1, i32* %a
  %1 = load i32* %a
  %2 = add i32 %1, 1
  store i32 %2, i32* %a
  %3 = load i32* %a
  %4 = add i32 %3, 1
  store i32 %4, i32* %a
  %5 = add i32 %1, %4
  store i32 %5, i32* %b
  %6 = load i32* %b
  ret i32 %6 }
```

An Example of LLVM’s Use Cases (1/3)

- Using LLVM to develop a run-time checker by inserting assertions is easier than using Clang
  - When we use Clang for analyzing C source code, we need to handle C’s complex language semantics including side effects
- Suppose that we would like to do array bound checking by inserting `assert()` before array accesses
  - One possible solution is to use Clang to insert `assert()` to check array subscription expression can be greater than the size of array

An example program

```
int example(int x){
  int a[10], b[10];
  … omitted code …
  // Want to check array bound
  b[++a[x++]]=0;
…}
```

An instrumented program

```
int example(int *a,int x){
  int b[10];
  … omitted code …
  assert(++a[x++]<10);
  b[++a[x++]]=0;
…}
```

- Will it Okay?

An Example of LLVM’s Use Cases (2/3)

- The array subscription expression `++a[x++]` has side effects
  - Executing `assert(++a[x++])` changes the value of `x` and `a[x]`
  - We should execute the array subscription expression once and store the result to use both `assert()` and array access
- In addition, we should do array bound check for the array subscription expression `++a[x++]` itself.
- If we choose Clang to develop a run-time checker to insert `assert()`, we should consider such complex semantics of C program code

An example program

```
int example(int x){
  int a[10], b[10];
  … omitted code …
  // Want to check array bound
  b[++a[x++]]=0;
…}
```

An instrumented program rev. 2

```
int example(int *a,int x){
  int b[10];
  … omitted code …
  int tmp1=x++;
  assert(tmp1<10);
  int tmp2=++a[tmp1]
  assert(tmp2<10);
  b[tmp2]=0;
…}
```

An Example of LLVM's Use Cases (3/3)

- If we use LLVM to perform array bound check, we can simply instrument the `getelementptr` instruction (LLVM instruction for array accesses) to check the 3rd parameter (array index) of the instruction
  - We do not suffer side effects because all side effects in C code are removed by LLVM front-end

An example program

```
int example(int *a,int x){
  int b[10];
  // Want to check array bound
  b[++a[x++]]=0;
…}
```

LLVM bytecode (Simplified version)

```
define i32 @example(i32 %x) {
  %1 = alloca i32
  %a = alloca [10 x i32]
  %b = alloca [10 x i32]
  … omitted code …
  %4 = sext i32 %2 to i64
  %5 = getelementptr [10 x i32]*
         %a, i32 0, i64 %4
         ; access to array a[10]
  … omitted code …
  %8 = sext i32 %7 to i64
  %9 = getelementptr [10 x i32]*
         %b, i32 0, i64 %8
         ; access to array b[10]
```