

Clang v.s. LLVM

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

Comparison of Clang and LLVM

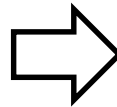
| | Clang | LLVM |
|-------------|--|---|
| Pros | <ul style="list-style-type: none">• Source code information (e.g., line/column number) is available• Clang supports source-to-source transformation | <ul style="list-style-type: none">• Complex high-level language semantics are lowered to relatively simple instructions• An analysis tool using LLVM can be programming language independent |
| Cons | <ul style="list-style-type: none">• A user should handle complex C/C++ language semantics (e.g., side effect, various AST node types) | <ul style="list-style-type: none">• Source code information is lost |
| Application | <ul style="list-style-type: none">• C's undefined behavior checker• Source code refactoring tool• Source code browser (e.g., Source Insight) | <ul style="list-style-type: none">• 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)

```
1 define i32 @example() {  
2     store i32 1, i32* %a  
3     %1 = load i32* %a  
4     %2 = add i32 %1, 1  
5     store i32 %2, i32* %a  
6     %3 = load i32* %a  
7     %4 = add i32 %3, 1  
8     store i32 %4, i32* %a  
9     %5 = add i32 %1, %4  
10    store i32 %5, i32* %b  
11    %6 = load i32* %b  
12    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

```
1 int example(int x){
2     int a[10], b[10];
3     ... omitted code ...
4     // Want to check array bound
5     b[++a[x++]]=0;
6 ...}
```



An instrumented program

```
1 int example(int *a,int x){
2     int b[10];
3     ... omitted code ...
4     assert(++a[x++]<10) ;
5     b[++a[x++]]=0;
6 ...}
```

- 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

```
1 int example(int x){
2     int a[10], b[10];
3     ... omitted code ...
4     // Want to check array bound
5     b[++a[x++]]=0;
6 ...}
```



An instrumented program rev. 2

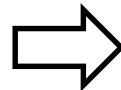
```
1 int example(int *a,int x){
2     int b[10];
3     ... omitted code ...
4     int tmp1=x++;
5     assert(tmp1<10);
6     int tmp2=++a[tmp1]
7     assert(tmp2<10);
9     b[ tmp2 ]=0;
10 ...}
```

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

```
1 int example(int *a,int x){
2     int b[10];
3     // Want to check array bound
4     b[++a[x++]]=0;
5 ...}
```



LLVM bytecode (Simplified version)

```
1 define i32 @example(i32 %x) {
2     %1 = alloca i32
3     %a = alloca [10 x i32]
4     %b = alloca [10 x i32]
5     ... omitted code ...
9     %4 = sext i32 %2 to i64
10    %5 = getelementptr [10 x i32]*
        %a, i32 0, i64 %4
        ; access to array a[10]
11    ... omitted code ...
14    %8 = sext i32 %7 to i64
15    %9 = getelementptr [10 x i32]*
        %b, i32 0, i64 %8
        ; access to array b[10]
```