An Introduction to LLVM Infrastructure
About LLVM

- **LLVM**: Low-level virtual machine
- A framework for writing compilers (including tools for static analysis)
- Written in C++
- Main author: Chris Lattner
LLVM IR: Intermediate Representation

- Input to LLVM
Example 1

```c
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    return 0;
}
```
Example 2

- Partial-SSA form
Example 3

```c
size1 = strlen(s1);
size2 = strlen(s2);
ret_val = xmalloc(size1 + size2 + 1);
strcpy(ret_val, s1);
strcpy(&ret_val[size1], s2);
return ret_val;
```

```assembly
%call = call i64 @strlen(i8* %s1.addr.0)
%conv = trunc i64 %call to i32
%4 = bitcast i32 %conv to i32
%call4 = call i64 @strlen(i8* %s2.addr.0)
%conv5 = trunc i64 %call4 to i32
%5 = bitcast i32 %conv5 to i32
%add = add nsw i32 %4, %5
%add6 = add nsw i32 %add, 1
%call7 = call i8* @xmalloc(i32 %add6)
%6 = bitcast i8* %call7 to i8*
%call8 = call i8* @strcpy(i9* %6, i8* %s1.addr.0)
%idxprom = sext i32 %4 to i64
%arrayidx = getelementptr inbounds i8* %6, i64 %idxprom
%call9 = call i8* @strcpy(i8* %arrayidx, i8* %s2.addr.0)
ret i8* %6
```
How do LLVM work – analyzed object

- Input: IR
  - Analyzing unit: Module

- Modules can be combined to a larger module
  - Useful for linking
How do LLVM work - implementation

- Composition
  - Header files
  
  ```
  root@ubuntu: /mnt/data/llvm/include/llvm# ls
  ADT Analysis Assembly AutoUpgrade.h Bitcode CMakeLists.txt CodeGen Config DebugInfo DebugInfo.h DIBuilder.h ExecutionEngine GVMaterializer.h InitializePasses.h InstVisitor.h IR IRRewriter LinkAllIR.h LinkAllPasses.h Linker.h LTO MC Object Option PassAnalysisSupport.h Pass.h PassManager.h PassRegistry.h PassSupport.h Summary Support TableGen Target Transforms
  ```

- Source files

  ```
  root@ubuntu: /mnt/data/llvm/lib# ls
  Analysis AsmParser Bitcode CMakeLists.txt CodeGen DebugInfo ExecutionEngine IR IRRewriter Linker LLVMBuild.txt MC LTO Option Makefile Summary TableGen Target Transforms
  ```
Using LLVM

- Writing frontend compilers
- Writing backend tools
- Writing tools using both frontend and backend
Writing frontend compilers

- Only need to compile source code to IR

Existing compilers that compile to IR:
- C/C++
- Ruby
- Python
- Haskell
- Java
- D
- PHP
- Pure
- Lua
- etc.
Clang: LLVM Frontend C/C++ Compiler

- Similar to gcc: easy to use
- Faster speed
- Better modularity
- Can be used separately from LLVM: Compile to executables
Writing backend tools

- Simpliest way: Using LLVM Passes
  - Module pass
  - Function pass
  - BasicBlock pass
Writing backend tools

- All passes are registered and managed by pass manager
  - Each pass is identified by its field address: ID
  - Running order of passes are written by tool developer

```cpp
void AllocIdentify::getAnalysisUsage(AnalysisUsage &AU) const {
    AU.addRequired<LoopInfo>();
    AU.setPreservesAll();
}

char AllocIdentify::ID = 0;
static RegisterPass<AllocIdentify>
X("alloc-identify", "Identify allocator wrapper functions");
```
Program analysis using LLVM

- Writing intra-procedure analysis tools
  - Using Clang CFG
  - Using LLVM passes

- Define-use chains are already provided
  - Value::use_iterator

- Alias analysis
  - Inherit alias analysis base class

- Pointer analysis
  - DSA
Combining Clang and LLVM

- Clang provides ASTs in source level code
- LLVM provides more powerful program analysis tools
- LLVM gold plugin
  - Used to perform link-time optimization
  - Based on GNU gold linker
Example: Memory-leak fixing

- Pointer analysis is performed at IR during linking
  - Existing tool: DSA

- Data flow analysis is performed via Clang CFG
  - Contain the information of source code location for fixing
Thanks!