Overview of LLVM
Chris Lattner created LLVM for his MSc thesis at University of Illinois at Urbana-Champain published in 2002.

LLVM 1.0 was released in 2003.

The initial purpose was to make a virtual machine which could optimize programs before, during, and after they are executed.

Now the focus is more on being a more normal compiler which also supports just-in-time compilation.

LLVM no longer stands for "low-level virtual machine.”

Google and Apple support LLVM to a large extent.
LLVM version 1.0

- Used C/C++ front-end from GCC
- Supported X86 and SPARC
- SSA Form
- Examples of implemented optimizations:
  - Function inlining
  - Dead code elimination
  - Constant propagation
  - Scalar replacement of aggregates
  - Loop-invariant code motion
  - Common subexpression elimination
  - Register allocation
More development

- LLVM 1.2 supported feedback-directed optimization
- Focus on the Clang C/C++ front-end to replace the GCC front-end
- In 2010 Clang could recompile itself, i.e. self-host
- In 2012 some demanding open-source projects switched to Clang, including FreeBSD
- In 2013 the Polly optimizer for parallelism became an official LLVM project in LLVM 3.1
- Polly is lead by Tobias Grosser who studied compilers for Christian Lengauer in Passau
- In LLVM 3.2 and 3.3 SIMD vectorization was added
- Most new work has been on better optimizations and supporting more CPU architectures
LLVM front-ends

- C/C++ with OpenMP extensions in Clang
- Scala
- Rust
- Haskell
- Julia
- Fortran (in development by Nvidia)
Source code structure

- Two main directories:
  - include
  - lib
- They typically exist in a src directory but the name ”src” does not matter
- include has two subdirectories: one for each of C++ and C
- lib contains e.g.
  - Analysis
  - CodeGen
  - IR
  - Target
  - Transform
- To create an LLVM-based tool, these libraries are used — different from GCC and other compilers which typically are much less modular
- Alias analysis
- Basic blocks
- Call graph
- Constant folding
- Control flow graph
- Data dependence analysis
- Dominance frontiers
- Inlining costs
- Loop analysis
- Post-dominance analysis
The intermediate representation is implemented in lib/IR
The types are declared in include/llvm/IR in 78 header files (39k lines)
A program is represented as:
- Module — one per translation unit or one for whole program
  - Function
  - Basic Block
  - Instruction
This IR is based on SSA form and can be saved to disk in text or binary form
The type LLVMContext keeps tracks of one or more modules
Each LLVM thread needs their own context
An instruction has a parent basic block, a basic block a parent function and a function a parent module
A module corresponds to a translation unit when compiling source such as with -c option.

During link-time optimization there is one module for the whole program.

A module has lists of global variables, constants, and functions.
Function

- Arguments
- Basic blocks
- Symbol table
- Attributes to improve optimizations:
  - The function may call `setjmp`
  - The function takes a variable number of arguments
  - The function may read memory
  - The function may modify memory
  - The function may be called speculatively (e.g. abs)
  - The function may call itself recursively directly or indirectly
Basic Block

- It contains the parent pointer to a function
- It contains a list of instructions
- The last instruction must be a TerminatorInst
- It is possible to move a basic block from one function to another
- It is possible to split a basic block in two parts
- `include/llvm/IR/CFG.h` contains iterators for successors and predecessors
There is a base class Instruction
It contains numerous functions for manipulating instructions such as moving them between basic blocks
There are functions to classify an instruction to determine e.g.:
- Whether it can terminate a basic block
- Whether it is a unary operation
- Whether it is a binary operation

An instruction has an opcode which is a constant such as Instruction::Xor
The opcodes are not defined in a C++ header file — instead they are defined in IR/Instruction.def, included by Instruction.h several times
This is due to the implementation based on macros
In InstrTypes.h most details about various instructions are declared
Examples: LoadInst, StoreInst, CallInst, PHINode, and ReturnInst
Translation unit types

- Types used by the application program are mapped to Type in the IR
- New types can be constructed using declarations in DerivedTypes.h
- Classes: StructType, ArrayType, VectorType, FunctionType, PointerType
- ArrayType is for normal arrays
- VectorType is for SIMD
The operands of an instruction have the type `Value`.

A value has a list of uses of that value.

A value represents what is computed, and both instructions and functions are values.

A user of a value is of the type `User` and therefore an instruction is a user.

To find all users of a value, the type `Use` is used.
Optimizations are performed in passes

A pass operates at a certain level:
- ModulePass
- FunctionPass
- LoopPass
- BasicBlockPass

An optimization in LLVM is declared as a derived type of a pass at a suitable level

For instance, constant propagation is declared as:

```cpp
struct ConstantPropagation : public FunctionPass {
    /* ... */
};
```
Pass managers

- At each level, there is a pass manager which controls the passes.
- A pass manager has a queue of optimizations to perform.
- A pass consists of two parts:
  - an analysis part
  - a transformation part
- An optimization specifies which analyses it requires to be performed as input (e.g. building the dominator tree).
- The pass manager then makes sure the request analyses have been done before the pass.
- A transformation may destroy some analyses and if that happens, it must tell the pass manager.
- There is a virtual function `getAnalysisUsage` which the different optimizations implement and it is called by a pass manager.
For ConstantPropagation the function is defined as:

```c
void getAnalysisUsage(AnalysisUsage &AU) const override {
    AU.setPreservesCFG();
    AU.addRequired<TargetLibraryInfoWrapperPass>();
}
```

- This says it will not modify the control flow graph
- And it needs certain target specific info
- This ConstantPropagation is not the advanced optimization you will implement in vcc, which in LLVM is called SCCP — sparse conditional constant propagation.
Creating and running an optimization pass

- LLVM uses "create" functions to instantiate an optimization pass:
  ```cpp
  FunctionPass *llvm::createConstantPropagationPass() {
    return new ConstantPropagation();
  }
  ```

- For a function pass, there is a boolean function:
  ```cpp
  bool ConstantPropagation::runOnFunction(Function& F) {
    /* ... */
  }
  ```

- It returns true if the function was modified

- An optimization pass is identified using a string which makes it possible to specify optimizations as arguments to an optimizer, such as `opt` as we will see below
The Transform library contains most optimizations.
The Transform/Scalar directory has 67 C++ files.
We can find familiar optimization techniques such as:
  - SCCP – sparse conditional constant propagation
  - GVN – global value numbering
  - DCE – dead code elimination
Recall the main tasks of a back-end:
- instruction selection
- instruction scheduling
- register allocation

The CodeGen library contains the machine-independent code for this

The Target library contains machine-specific code for the backend
LLVM source code

- LLVM — what we have seen
- Clang — C/C++ front-end installed in src/tools/clang
- Compiler-rt — runtime libraries: builtin functions, sanitizer, profiling etc
- Download with browser or wget:

  wget https://releases.llvm.org/6.0.1/llvm-6.0.1.src.tar.xz \  
  https://releases.llvm.org/6.0.1/cfe-6.0.1.src.tar.xz \  
  https://releases.llvm.org/6.0.1/compiler-rt-6.0.1.src.tar.xz
Then extract the files:

$ ls *.xz | xargs -n1 tar xf

Move the extracted files to appropriate locations:

$ mv llvm-6.0.1.src src
$ mv cfe-6.0.1.src src/tools/clang
$ mv compiler-rt-6.0.1.src src/projects/compiler-rt

Create and configure the build directory:

$ mkdir build && cd build
$ cmake ../src -DCMAKE_BUILD_TYPE=Release \
   -DLLVM_ENABLE_ASSERTIONS=ON \
   -DLLVM_TARGETS_TO_BUILD=PowerPC \
   -DCMAKE_INSTALL_PREFIX=/opt/llvm/6.0.1

Finally make and install:

$ make && make install
Using LLVM

- export PATH=/opt/llvm/6.0.1/bin:$PATH
- To produce LLVM IR instead of assembler or machine code when compiling a file a.c, use either:
  - clang a.c -emit-llvm -S
  to produce textual IR, or
  - clang a.c -emit-llvm -c
  to produce binary IR.
- The textual file has suffix .ll and the binary .bc.
- Use opt to perform machine-independent optimizations of a file:
  - opt a.ll -S -O3 -o b.ll
  We get almost identical output with
  - opt a.bc -S -O3 -o c.ll
  with the only difference between b.ll and c.ll being the module id.
We can specify individual optimizations to perform such as:

```bash
opt a.bc -S -inline -o c.ll
```

Use `-help` to list all parameters of `opt`.

Code generation is performed with `llc` which can produce an assembler file with suffix `.s` with the command

```bash
llc c.ll -o c.s
```

or an object file with suffix `.o` with the command

```bash
llc c.ll -filetype=obj -o c.o
```

The latter file can be disassembled with a standard Unix command

```bash
objdump -d c.o
```
Making and running your own LLVM pass

- First use the existing Hello pass:
  ```
  make -f lib/Transforms/Hello/Makefile
  ```
- You should get a shared library `lib/LLVMHello.so`
- Invoke it with
  ```
  bin/opt -load lib/LLVMHello.so -hello a.ll
  ```
- Make a new directory EDAN75 by copying Hello
- Replace Hello with EDAN75 and also LLVMHello with EDAN75
- Add EDAN75 to `src/lib/Transforms/CMakeLists.txt`
- Go back to the build directory
- Run cmake as
  ```
  cmake -DCMAKE_INSTALL_PREFIX=/opt/llvm/6.0.1 ../src
  ```
- Type: `make -f lib/Transforms/EDAN75/Makefile`
- Then you can run it with
  ```
  bin/opt -load lib/EDAN75.so -EDAN75 a.ll -S
  ```