Contents

- Compiling LLVM
- Using `opt` and `llc`
- Some LLVM history
- LLVM files and data structures

Downloading and compiling LLVM

```
v=10.0.1 &&
sudo mkdir -p /opt/llvm/$v &&
sudo chown -Rv $USER:$USER /opt/llvm/$v &&
rm -rf llvm/$v &&
mkdir -p llvm/$v &&
cd llvm/$v &&
a=https://github.com/llvm/llvm-project/releases/download/llvmorg-$v &&
wget $a/llvm-$v.src.tar.xz \
   $a/clang-$v.src.tar.xz \
   $a/compiler-rt-$v.src.tar.xz &&
ls *.xz | xargs -n1 tar xf &&
mv llvm-$v.src src &&
mv clang-$v.src src/tools/clang &&
mv compiler-rt-$v.src src/projects/compiler-rt &&
mkdir build &&
cd build &&
cmake ../src -DCMAKE_BUILD_TYPE=Release \
    -DLLVM_ENABLE_ASSERTIONS=OFF \
    -DLLVM_TARGETS_TO_BUILD=PowerPC \
    -DCMAKE_INSTALL_PREFIX=/opt/llvm/$v &&
make &&
make install &&
echo ok
```

- This can be found in the tresor for the course

Using LLVM

- 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`.

Using opt

- Assume you are in the build directory
- We can then use `bin/opt` to perform machine-independet optimizations on a file, again using `-S` to output a textual file.

  ```
  $ bin/opt a.ll -S -O3 -o b.ll
  ```

- We get almost identical output with

  ```
  $ bin/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:

  ```
  $ bin/opt a.bc -S -inline -o c.ll
  ```

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

The EDAN75 LLVM project

- You will produce a shared library: `build/lib/EDAN75.so`

  ```
  $ bin/opt -load lib/EDAN75.so -S -o b.ll a.ll -EDAN75
  ```

- A new optimization pass is registered with a string that can be used as a switch to opt

Using llc

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

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

  or an object file with suffix o with the command

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

  and the latter file can be disassembled with a standard UNIX command

  ```
  $ objdump -d c.o
  ```

LLVM history

- 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, Apple and others 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
- 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
- New release naming from LLVM 4 (two major releases per year)
- 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
  - MC
  - Support
  - Target
  - Transform

clang

- Several tools, including `clang`, is in `src/tools`.
- Many tools share various libraries.
- This is an advancement compared to many other compilers which often are designed for only one purpose, which makes it more complicated to reuse its parts, in case that would be desired.
- This design has been important to make LLVM popular in many commercial and academic projects.

The lib directory

- Libraries are located in `lib`
- Header files in `include/llvm`.
- Most files are C++
- Machine specifications are implemented in a specialized language.
- Only the POWER specifications are more than 20,000 lines.

llvm source size of llvm 6.0

Directory	`du -ks`	**Files**	**Description**
`include/llvm`	14 MB	1066	Common declarations and some algorithms in templates such as Lengauer-Tarjan
`lib/Analysis`	3 MB	85	CFG, loops, inlining cost and more
`lib/CodeGen`	9 MB	221	Code generation
`lib/IR`	2 MB	51	Intermediate representation
`lib/MC`	1 MB	68	Machine code
`lib/Support`	2 MB	145	Data structures
`lib/Target`	39 MB	> 1000	Architecture specific code
`lib/Transform`	8 MB	203	Code optimization algorithms

The intermediate representation: IR

- The input to an LLVM optimizer is a program representation using LLVM IR
- Types in the `IR` library.
- The IR can be stored in three ways:
    - in memory,
    - in binary format on disk as bitcode, and
    - in textual form.

Optimization algorithms

- Machine-independent optimization use the `Analysis` and `Transform` libraries
- Machine-dependent optimization use the `CodeGen` and `Target` libraries:
    - instruction selection
    - instruction scheduling, and
    - register allocation

Object files

- The `MC` library is used to operate on machine instructions to, for instance, create object files
- On Linux: ELF
- On macOS: Mach-O

LLVM types

- Types declared in directory: `include/IR` in the namespace `llvm`.
- There usually is a file corresponding to the type name.
- For instance, the `Instruction` class is declared in the file `include/IR/Instruction.h`.

LLVM context

- The type `LLVMContext` stores the global state of LLVM.
- Each thread needs its own context
- A context contains tables of types and constants, and pointers to zero or more modules.

Module

- A `Module` normally corresponds to a translation unit.
- It has doubly linked lists of global variables, functions, aliases, certain meta data, and more, and provides iterators for simple access to them.
- Functions to lookup and install symbols are also provided.
- During link-time optimization multiple translation-units become one module.

Function

Some attributes

- a pointer to arguments and the number of arguments
- a symbol table,
- the control flow graph
- does it take a variable number of arguments?
- does it access memory? if so only reads memory?
- can it not return?
- has it side-effects?
- does it recurse (directly or indirectly)?
- does it return a structure?
- has it had its address taken?
- may it call the C function `setjmp`?
- the number of times the function was called if known.
- should it be optimized for size?

BasicBlock 1/2

- A `BasicBlock` contains pointer to its parent function, and a list of instructions.
- Any ϕ-functions must be first
- The last must have type `TerminatorInst`.
- For convenience, there are iterator types for the instruction list, and a function `getFirstNonPhi` which returns the first `Instruction` in the list which is not a ϕ-function.

BasicBlock 2/2

- It is possible to move a basic block from its parent function to another function.
- It is possible to split a basic block in two parts.
- The operands of an instruction have the type `Value`.
- Since basic blocks are operands of branch instructions they are also values
- Therefore the `BasicBlock` type inherits from the `Value` type, which we will explain after introducing the `Instruction` and `Type` classes.

Instruction

- The base `Instruction` class in the IR library contains a parent basic block pointer and is a `User` (explained below).
- There are numerous functions for manipulating the list of instructions of a basic block.
- Different kinds of instructions are declared in `InstrTypes.h`: e.g. `TerminatorInst`, unary, binary, and comparison instructions.
- Concrete instructions are declared in the file `Instructions.h`, e.g. `AllocaInst` to allocate stack space, `LoadInst`, `StoreInst`, `FenceInst`, `FCmpInst`, `CallInst`, `PHINode`, `ReturnInst`, and many more.

Type

- `Type` is the base class defined in `Type.h` and is used for primitive types such as C `float`, `double`, `void`, and integer types of different sizes.
- `DerivedTypes.h` has derived classes with which for example arrays and structs can be represented.
- To determine whether two type objects are equal, it is sufficient to check for pointer equality, as only one instance of each type object is ever created.

Value

- A `Value` represents what is computed
- Both instructions and functions are values.
- A value has a type and a list of users of it.
- The users of a value are also values, such as when one instruction has an operand which is the result of another instruction.
- It is possible to iterate through all users of a value and for this the type `Use` is used.

User

- A `User` is an entity which can use values as operands
- A user is also itself a value.
- An example of a user is an instruction. The function `getOperand` returns a pointer to the *i*th operand, i.e. a pointer to a value.

Use

- A `Use` represents the use of a value, e.g. the fact that a particular instruction uses a certain value as an operand.
- The type `Use` is used for connecting values with their users in LLVM.
- Due to it is one of, if not the most, memory intensive object in LLVM, it is implemented with saving memory in mind, while also being very flexible.
- A use has an explicit pointer to the value being used, while the pointer to the user, is not stored explicitly, but can still be found efficiently.

Useful functions

- get iterators to first and last basic blocks: `f.begin()` and `f.end()` where `f` is parameter to `bool runOnFunction(Function& f)` and the iterator should be declared e.g. as: `auto u = f.begin()`
- `runOnFunction` should return true if the function was modified
- get iterators to first and last instructions: `u->begin()` and `u->end()` where `u` is a basic block iterator
- check if two instructions are identical: `i->isIdenticalTo(j)`
  See `src/lib/IR/instruction.cpp`
- The call `i->replaceAllUsesWith(j)` does exactly what it says
  See `src/lib/IR/Value.cpp`
- The call `i->eraseFromParent()` erases an instruction from a basic block and returns an iterator pointing to the next instruction
- In general, to advance an iterator `i` one step, you can use `i++`

LLVM Passes

- Optimizations are performed in LLVM as passes which operate on the IR at a certain level:
  - module,
  - call graph,
  - function,
  - region of a function,
  - loop, or
  - basic block.

LLVM Passes

- The base type `Pass` is declared in `include/llvm/Pass.h` which also contains declarations for
  - `ModulePass`,
  - `FunctionPass`, and
  - `BasicBlockPass`.

  The other pass levels are declared in files, with corresponding names, in the directory `include/llvm/Analysis`:
  - `CallGraphSCCPass`,
  - `RegionPass`, and
  - `LoopPass`.

LLVM optimization

- An LLVM optimization is implemented as a derived type of a pass at a suitable level. For example, constant propagation is implemented by the type

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

  In general, an optimization is separated in an analysis part and a transformation part. An example of an analysis part is computing the dominator tree.
- A `PassManager` is a template class instantiated with a type corresonding to the appropriate level, such as function, i.e. `PassManager<Function>`, and in `llvm/IR/PassManager.h` there are corresponding type alias declarations of `ModulePassManager` and `FunctionPassManager`.

Pass manager

- A pass manager contains a queue of optimization passes to perform in order.
- The pass object is owned by the pass manager which hence also deletes it when it itself is deleted.
- It is the responsibility of the pass manager to perform the required analyses before an optimization pass is run. In a pass type, a virtual function getAnalysisUsage can be defined, and be used to communicate with the pass manager. For instance, ConstantPropagation defines it as:

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

Code generation

- After machine-independent optimizations, code generation is performed by translating the LLVM IR to a representation which is closer to the CPU architecture of the target.
- Code generation consists of three parts:
    - instruction selection,
    - instruction scheduling, and
    - register allocation.

Instruction selection

- Instruction selection is performed per basic block.
- The LLVM IR instructions, with their data dependences, are translated to a directed acyclic graph, called the selection DAG
- Then a matching of the nodes of the DAG to instructions of the CPU is performed.
- The type declarations for code generation are in files in the `include/CodeGen` directory.
- The selection DAG is declared in `SelectionDAG.h` and the nodes in `SelectionDAGNodes.h`.
- In addition files from the specific CPU architecture are used, which are located in one of the `lib/Target` subdirectories, such as `PowerPC`.

Instruction scheduling

- Instruction scheduling is performed both before and after register allocation.
- The main CPU implementations, each have a file with a description of the CPU pipeline.
- Instruction scheduling is performed on nodes from the selection DAGs.
- Scheduling before register allocation can use several different schedulers, and a suitable default scheduler is specified for each CPU architecture.
- List scheduling is used after register allocation with operations in a priority queue, and if the operation with highest priority is valid to schedule, it is scheduled.

Register allocation in LLVM

- Before LLVM 3.0 linear scan: scan basic blocks one at a time and note which registers are used and take suitable action at CFG edges
- Remarks copied from talk by Jakob Stoklund Olesen about LLVM 3.0:
  - *Need to tidy the infrastructure*
  - *Linear scan is not, in fact, linear*
  - *Major bookkeeping nightmare*
- LLVM has four official alternative register allocators:
  - fast — operates at a basic block level.
  - basic — small live ranges tend to be allocated before large
  - greedy — assigns to large live ranges first but is more flexible
  - PBQP — partitioned boolean quadratic programming for irregular architectures
- In addition, many academic research projects have evaluated others.
- The focus of the course is on Chaitin and the iterated register coalescing algorithm