

LLVM Pet Project

Simple compiler using LLVM's codegen infrastructure and JIT execution engine

Demonstrate how easy it is to create a new compiler from scratch

And benefit from the powerful code generation and optimizations routines of LLVM

Renato Golin

rengolin@systemcall.org

Why LLVM?

- With LLVM you can:
 - Easily create a powerful IR
 - Create new optimizations and metadata
- With LLVM you get for free:
 - Validation of the IR structure
 - Static optimization of the generated IR
 - Machine code generator for many platforms
 - A virtual machine (JIT)
 - Many run-time optimizations

Basics

- Simple parser / tokenizer
 - Build AST based on token type & value
 - Error messages are horrible (no line number)
- Simple codegen
 - Generate IR based on node's value
 - Each generator knows when to change insert point
- Using execution engine provided
 - No modifications, only simple optimizations

Codegen

- Each AST node type triggers a different generator
 - Some are recursive (expressions)
 - Some are global (states and variables)
 - Some are constant (numbers)
 - Some change insertion point (state, if/else)
- Global states, no function call
 - Each state is a BasicBlock inside the same function
 - Extern functions are declared but not defined
 - JIT can use `extern` "C" functions linked to it

Codegen – Global Function

- **First, create a Module***
 - `context = getGlobalContext();`
 - `mod = new Module("MyModule", context);`
- **Then, create a FunctionType***
 - `vararg = false;`
 - `ret_t = Type::getVoidTy(context);`
 - `ft = FunctionType::get(ret_t, vararg);`
- **Now, the Function***
 - `func = Function::Create(ft, Function::ExternalLinkage, "func", mod);`

Codegen - BasicBlock

- All code must be inserted in BasicBlock*s
- Each function can have multiple blocks
 - `global = BasicBlock::Create(context, "global", func);`
- The IRBuilder must know where to insert
 - `builder.SetInsertPoint(global);`
- Still, it's orthogonal to blocks and functions
 - Variables and states can be easily *mapped*
 - Validation makes sure everything is correct

Codegen - Variables

- We can allocate some variables..
 - `Value* var = builder.CreateAlloca(Type::getDoubleTy(context), 0, "var"); // SSA as varN`
- Create a constant...
 - `Value* two = ConstantFP::get(context, APFloat(2.0));`
- And assign it to the variable
 - `builder.CreateStore(two, var);`

Codegen - Expressions

- We can create simple expressions...
 - `Value* add = builder.CreateFAdd(two, two, "four"); // SSA as fourN`
 - `Value* mul = builder.CreateFMul(two, two, "four"); // SSA as fourN+1`
- To compare them together,
 - `Value* eq = builder.CreateFCmpOEq(add, mul, "eq"); // SSA as eqN`
- Or Assign to a variable
 - `builder.CreateStore(mul, var);`

Codegen - FunctionCall

- First, you need to know which function to call
 - I store functions in fields (only a few hard-coded)
 - states and variables in maps (lots user-defined)
- Then you simply create the call
 - `Function* func = funcs["func"];`
 - `builder.CreateCall(func, var, "retVal");`
- Parameters (val) can be `Value*` or iterators
- `retVal` is the SSA name of the returned value
 - Or empty for void

Codegen - Branch

- Every state is a `BasicBlock*`
- Jumps to `BasicBlocks` only
 - Param is an object and not a name
 - It guarantees you have created it, at least
- Jumping between states (blocks) is easy
 - `BasicBlock* bb = states["myState"];`
 - `builder.CreateBr(bb);`

Codegen - If/Else

- If needs a condition
 - `Value* cond = eq;`
- And three basic blocks
 - if-block: to be executed if cond is true
 - else-block: otherwise
 - merge-block: where both go at the end
- You need to keep track of PHIs manually
 - Though, optimizations do it for you later

Codegen - If/Else

- **Example:**

```
BasicBlock *then = BasicBlock::Create(context, "then", func);
BasicBlock *else = ..., *merge = ...; // same thing
builder.CreateCondBr(cond, then, else);

builder.SetInsertPoint(then);

// generate code for true condition

builder.CreateBr(merge);

// Same for else

builder.SetInsertPoint(merge);

// Continue code gen for that state/function
```

Codegen - Validate

- LLVM has a verification method after you're done with your function
 - `verifyFunction(*func);`
- It'll check for basic things like
 - Multiple unconditional branches in the same block
 - No branch at the end of a block
 - Unused blocks (unreachable code)

Optimizer - FunctionPass

- The optimization is done via FunctionPass
 - They run through the LLVM IR
 - Can use metadata for advanced passes
- Create a FunctionPassManager
 - `provider = new ExistingModuleProvider(mod);`
 - `fpm = new FunctionPassManager(provider);`
- And add function passes
 - `fpm->add(createGVNPass()); // and others`
 - `fpm->doInitialization();`

JIT

- The JIT gets a Function*

- `Function* f = funcs["myFunc"];`

- You can optimize it now

- `fpm->run(*f);`

- Transforms into a function pointer

- `void *fv = engine->getPointerToFunction(f);`

- `void (*fp)() = (void (*)())(intptr_t) fv;`

- And execute

- `fp();`

Source Organization

- Source organization:
 - token.cpp: static tokenizer function
 - ast.cpp/h: AST nodes definitions
 - parser.cpp/h: get token, build AST
 - codegen.cpp/h: get AST, build IR
 - jit.cpp/h: optimize and execute
- Dependencies
 - boost::shared_ptr / pointer_cast (polymorph.)
 - LLVM 2.6-svn

References

- Project page
 - <http://systemcall.org/rengolin/stuff/compiler/>
- LLVM tutorial (Kaleidoscope)
 - <http://llvm.org/docs/tutorial/LangImpl1.html>
- LLVM Documentation
 - <http://llvm.org/docs/>
- LLVM mailing list
 - <http://lists.cs.uiuc.edu/mailman/listinfo/llvmdev>