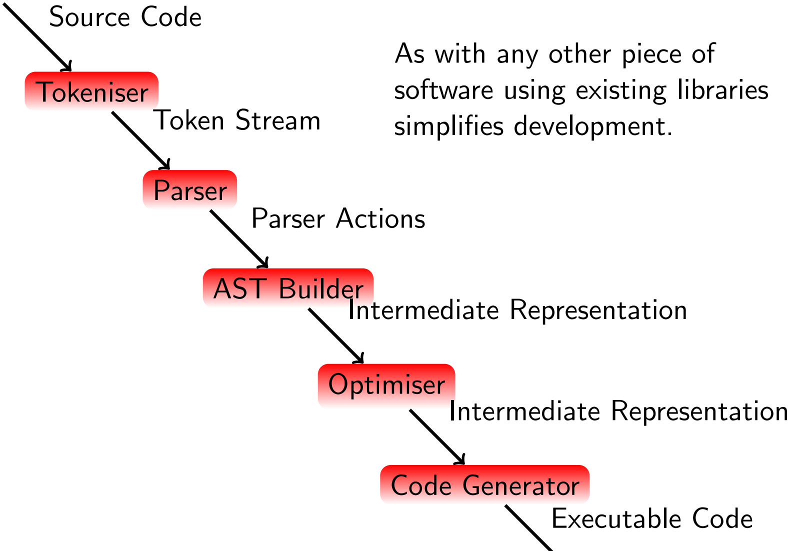
What LLVM Can Do For You

David Chisnall

April 13, 2012

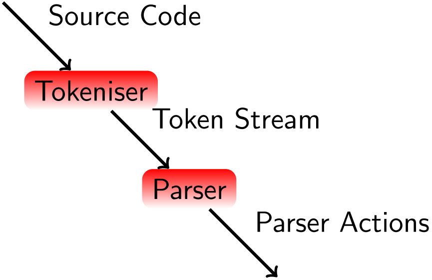
Part 1: Introduction to LLVM

Overview of a Compiler



As with any other piece of software using existing libraries simplifies development.

Building a Front End



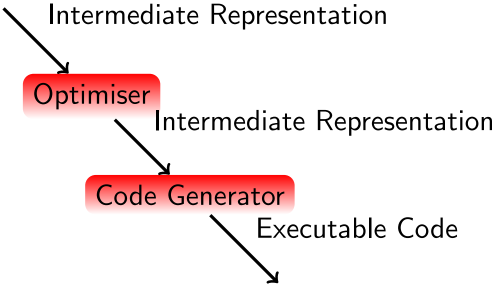
Many existing tools:

- Lex + yacc
- Lemon
- ANTLR
- OMeta
- ...

And the Middle?

- ASTs tend to be very language-specific
- You're (mostly) on your own there

What About the Back End?



This is where LLVM comes in.

What is LLVM?

- A set of libraries for implementing compilers
- Intermediate representation (LLVM IR) for optimisation
- Various helper libraries

Great for Compiler Writers!

- Other tools help you write the front end
- LLVM gives you the back end
- A simple compiler can be under 1000 lines of (new) code

What About Library Developers?

- LLVM optimisations are modular
- Does your library encourage some common patterns among users?
- Write an optimisation that makes them faster!

All programmers use compilers. Now all programmers can improve their compiler.

What Is LLVM IR?

- Unlimited Single-Assignment Register machine instruction set
- Three common representations:
  - Human-readable LLVM assembly (.ll files)
  - Dense 'bitcode' binary representation (.bc files)
  - C++ classes

Unlimited Register Machine?

- Real CPUs have a fixed number of registers
- LLVM IR has an infinite number
- New registers are created to hold the result of every instruction
- CodeGen's register allocator determines the mapping from LLVM registers to physical registers

Static Single Assignment

- Registers may be assigned to only once
- Most (imperative) languages allow variables to be... variable
- This requires some effort to support in LLVM IR

Multiple Assignment

```
int a = someFunction();
a++;
```

- One variable, assigned to twice.

Translating to LLVM IR

```
%a = call i32 @someFunction()
%a = add i32 %a, 1
```

```
error: multiple definition of local value named 'a'
  %a = add i32 %a, 1
    ^
```

Translating to *Correct* LLVM IR

```
%a = call i32 @someFunction()
%a2 = add i32 %a, 1
```

- How do we track the new values?

Translating to LLVM IR The Easy Way

```
; int a
%a = alloca i32, align 4
; a = someFunction
%0 = call i32 @someFunction()
store i32 %0, i32* %a
; a++
%1 = load i32* %a
%2 = add i32 %0, 1
store i32 %2, i32* %a
```

- Numbered register are allocated automatically
- Each expression in the source is translated without worrying about data flow
- Memory is not SSA in LLVM

Isn't That Slow?

- Lots of redundant memory operations
- Stores followed immediately by loads
- The mem2reg pass cleans it up for is

```
%0 = call i32 @someFunction()
%1 = add i32 %0, 1
```

Sequences of Instructions

- A sequence of instructions that execute in order is a *basic block*
- Basic blocks must end with a terminator
- Terminators are flow control instructions.
- `call` is not a terminator because execution resumes at the same place after the call

Intraprocedural Flow Control

- Assembly languages typically manage flow control via jumps / branches
- LLVM IR has conditional and unconditional branches
- Branch instructions go at the end of a basic block
- Basic blocks are branch targets
- You can't jump into the middle of a basic block

What About Conditionals?

```
int b = 12;
if (a)
    b++;
return b;
```

- Flow control requires one basic block for each path
- Conditional branches determine which path is taken

‘Phi, my lord, phi!’ - Lady Macbeth, Compiler Developer

- PHI nodes are special instructions used in SSA construction
- Their value is determined by the preceding basic block
- PHI nodes must come before any non-PHI instructions in a basic block

```
entry:
 ; int b = 12
 %b = alloca i32
 store i32 12, i32* %b
 ; if (a)
 %0 = load i32* %a
 %cond = icmp ne i32 %0, 0
 br i1 %cond, label %then, label %end
```

```
then:
 ; b++
 %1 = load i32* %b
 %2 = add i32 %1, 1
 store i32 %2, i32* %b
 br label %end
```

```
end:
 ; return b
 %3 = load i32* %b
 ret i32 %3
```

In SSA Form...

```
entry:
 ; if (a)
 %cond = icmp ne i32 %a, 0
 br i1 %cond, label %then, label %end
```

```
then:
 ; b++
 %inc = add i32 12, 1
 br label %end
```

```
end:
 ; return b
 %b.0 = phi i32 [ %inc, %then ], [ 12, %entry ]
 ret i32 %b.0
```

The output from the mem2reg pass

And After Constant Propagation...

```
entry:
 ; if (a)
 %cond = icmp ne i32 %a, 0
 br i1 %cond, label %then, label %end
```

```
then:
 br label %end
```

The output from the constprop pass. No add instruction.

```
end:
 ; b++
 ; return b
 %b.0 = phi i32 [ 13, %then ], [ 12, %entry ]
 ret i32 %b.0
```

And After CFG Simplification...

```
entry:
  %tobool = icmp ne i32 %a, 0
  %0 = select i1 %tobool, i32 13, i32 12
  ret i32 %0
```

- Output from the simplifycfg pass
- No flow control in the IR, just a `select` instruction

Why Select?

x86:

```
testl %edi, %edi
setne %al
movzbl %al, %eax
orl $12, %eax
ret
```

ARM:

```
mov r1, r0
mov r0, #12
cmp r1, #0
movne r0, #13
mov pc, lr
```

PowerPC:

```
cmplwi 0, 3, 0
beq 0, .LBB0_2
li 3, 13
blr
.LBB0_2:
li 3, 12
blr
```

Branch is only needed on some architectures.

Functions

- LLVM functions contain at least one basic block
- Arguments are explicitly typed

```
@hello = private constant [13 x i8] c"Hello
   world!\00"

define i32 @main(i32 %argc, i8** %argv) {
entry:
  %0 = getelementptr [13 x i8]* @hello, i32 0,
      i32 0
  call i32 @puts(i8* %0)
  ret i32 0
}
```

Get Element Pointer?

- Often shortened to GEP (in code as well as documentation)
- Represents pointer arithmetic
- Translated to complex addressing modes for the CPU
- Also useful for alias analysis: result of a GEP is the same object as the original pointer (or undefined)

F!@£ing GEPs! HOW DO THEY WORK?!?

```
struct a {
    int c;
    int b[128];
} a;
int get(int i) { return a.b[i]; }
```

```
%struct.a = type { i32, [128 x i32] }

define i32 @get(i32 %i) {
entry:
  %arrayidx = getelementptr %struct.a* @a, i32
      0, i32 1, i32 %i
  %0 = load i32* %arrayidx
  ret i32 %0
}
```

As x86 Assembly

```
define i32 @get(i32 %i) {
entry:
  %arrayidx = getelementptr inbounds %struct.a*
     @a, i32 0, i32 1, i32 %i
  %0 = load i32* %arrayidx
  ret i32 %0
}
```

```
get:
    movl    4(%esp), %eax          # load parameter
    movl    a+4(,%eax,4), %eax     # GEP + load
    ret
```

As ARM Assembly

```
define i32 @get(i32 %i) {
entry:
  %arrayidx = getelementptr inbounds %struct.a*
     @a, i32 0, i32 1, i32 %i
  %0 = load i32* %arrayidx
  ret i32 %0
}
```

```
get:
    ldr     r1, .LCPI0_0        // Load global address
    add     r0, r1, r0, lsl #2 // GEP
    ldr     r0, [r0, #4]        // load return value
    bx      lr
.LCPI0_0:
    .long    a
```

Part 2: Writing a Simple Front End

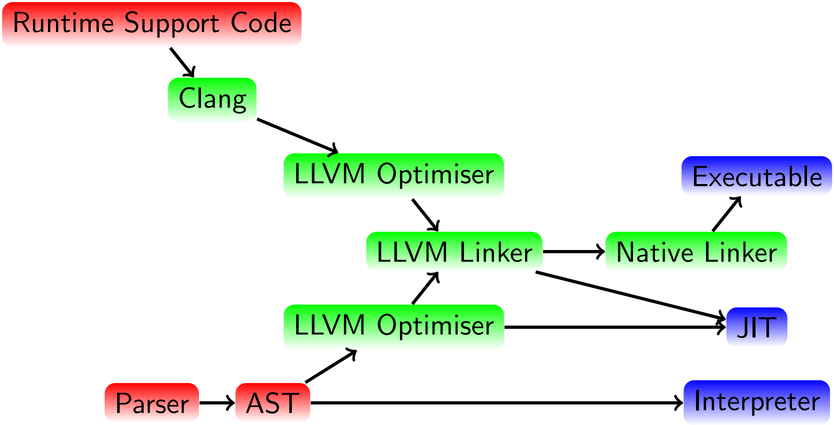
What Applications Need Compilers?

- UNIX `bc` / `dc`
- Graphviz
- JavaScript
- AppleScript / Visual Basic for Applications
- Firewall filter rules
- EMACS Lisp

How Do I Use LLVM?

- Generate LLVM IR from your language
- Link to some helper functions written in C and compiled to LLVM IR with clang
- Run optimisers
- Emit code: object code files, assembly, or machine code in memory (JIT)

A Typical LLVM-based Compiler Implementation



A Note About LLVM Types

- LLVM is strongly typed
- Types are structural (e.g. 8-bit integer - signed and unsigned are properties of operations, not typed)
- Arrays of different length are different types
- Pointers and integers are different
- Structures with the same layout are different if they have different names (new in LLVM 3.)
- Various casts to translate between them

A Worked Example

Full source code:

`http://cs.swan.ac.uk/~csdavec/FOSDEM12/examples.tbz2`

Compiler source file:

`http://cs.swan.ac.uk/~csdavec/FOSDEM12/compiler.cc.html`

A Simple DSL

- Simple language for implementing cellular automata
- Programs run on every cell in a grid
- Lots of compromises to make it easy to implement!
- 10 per-instance accumulator registers (a0-a9)
- 10 shared registers (g0-g9)
- Current cell value register (v)

Arithmetic Statements

```
{operator} {register} {expression}
```

- Arithmetic operations are statements - no operator precedence.

Neighbours Statements

```
neigbours ( {statements} )
```

- Only flow control construct in the language
- Executes the statements once for every neighbour of the current cell

Select Expressions

```
[ {register} |
    {value or range) => {expression},
    {value or range) => {expression}...
]
```

- Maps a value in a register to another value selected from a range
- Unlisted ranges are implicitly mapped to 0

Examples

Flash every cell:

```
= v [ v | 0 => 1 ]
```

Count the neighbours:

```
neighbours ( + a1 1)
= v a1
```

Connway's Game of Life:

```
neighbours ( + a1 a0 )
= v [ v |
    0 => [ a1 | 3 => 1] ,
    1 => [ a1 | (2,3) => 1]
]
```

AST Representation

- Nodes with two children
- Registers and literals encoded into pointers with low bit set

Implementing the Compiler

- One C++ file
- Uses several LLVM classes
- Some parts written in C and compiled to LLVM IR with clang

The Most Important LLVM Classes

- `Module` - A compilation unit.
- `Function` - Can you guess?
- `BasicBlock` - a basic block
- `GlobalVariable` (I hope it's obvious)
- `IRBuilder` - a helper for creating IR
- `Type` - superclass for all LLVM concrete types
- `ConstantExpr` - superclass for all constant expressions
- `PassManagerBuilder` - Constructs optimisation passes to run
- `ExecutionEngine` - The thing that drives the JIT

The Runtime Library

```
void automaton(int16_t *oldgrid, int16_t *
    newgrid, int16_t width, int16_t
     height) {
  int16_t g[10] = {0};
  int16_t i=0;
  for (int16_t x=0 ; x<width ; x++) {
    for (int16_t y=0 ; y<height ; y++,i++) {
      newgrid[i] = cell(oldgrid, newgrid, width,
          height, x, y, oldgrid[i], g);
    }
  }
}
```

Generate LLVM bitcode that we can link into our language:

```
$ clang -c -emit-llvm runtime.c -o runtime.bc -O0
```

Setup

```
// Load the runtime module
OwningPtr<MemoryBuffer> buffer;
MemoryBuffer::getFile("runtime.bc", buffer);
Mod = ParseBitcodeFile(buffer.get(), C);
// Get the stub function
F = Mod->getFunction("cell");
// Stop exporting it
F->setLinkage(GlobalValue::PrivateLinkage);
// Set up the first basic block
BasicBlock *entry =
    BasicBlock::Create(C, "entry", F);
// Create the type used for registers
regTy = Type::getInt16Ty(C);
// Get the IRBuilder ready to use
B.SetInsertPoint(entry);
```

Creating Space for the Registers

```
for (int i=0 ; i<10 ; i++) {
  a[i] = B.CreateAlloca(regTy);
}
B.CreateStore(args++, v);
Value *gArg = args;
for (int i=0 ; i<10 ; i++) {
  B.CreateStore(ConstantInt::get(regTy, 0), a[i
      ]);
  g[i] = B.CreateConstGEP1_32(gArg, i);
}
```

Compiling Arithmetic Statements

```
Value *reg = B.CreateLoad(a[val]);
Value *result = B.CreateAdd(reg, expr);
B.CreateStore(result, a[val]);
```

- LLVM IR is SSA, but this isn't
- Memory is not part of SSA
- The Mem2Reg pass will fix this for us

Flow Control

- More complex, requires new basic blocks and PHI nodes
- Range expressions use one block for each range
- Fall through to the next one

Range Expressions

```
PHINode *phi = PHINode::Create(regTy, count, "
    result", cont);
...
// For each range:
  Value *min = ConstantInt::get(regTy, minVal);
  Value *max = ConstantInt::get(regTy, maxVal);
  match = B.CreateAnd(B.CreateICmpSGE(reg, min),
      B.CreateICmpSLE(reg, max));
  BasicBlock *expr = BasicBlock::Create(C, "
     range_result", F);
  BasicBlock *next = BasicBlock::Create(C, "
     range_next", F);
  B.CreateCondBr(match, expr, next);
  B.SetInsertPoint(expr); // (Generate the
     expression after this)
  phi->addIncoming(val, B.GetInsertBlock());
  B.CreateBr(cont);
```

Optimising the IR

```
PassManagerBuilder PMBuilder;
PMBuilder.OptLevel = optimiseLevel;
PMBuilder.Inliner =
    createFunctionInliningPass(275);
FunctionPassManager *FPM =
    new FunctionPassManager(Mod);
PMBuilder.populateFunctionPassManager(*FPM);
for (Module::iterator I = Mod->begin(),
     E = Mod->end() ; I != E ; ++I) {
    if (!I->isDeclaration()) FPM->run(*I);
}
FPM->doFinalization();
PassManager *MP = new PassManager();
PMBuilder.populateModulePassManager(*MP);
MP->run(*Mod);
```

Generating Code

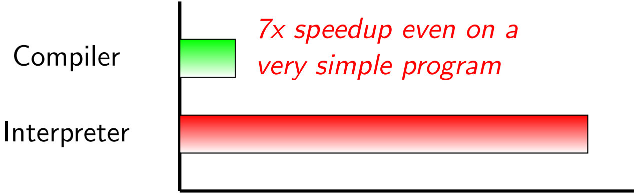
```
std::string error;
ExecutionEngine *EE = ExecutionEngine::create(
    Mod, false, &error);
if (!EE) {
  fprintf(stderr, "Error: %s\n", error.c_str());
  exit(-1);
}
return (automaton)EE->getPointerToFunction(Mod->
    getFunction("automaton"));
```

Now we have a function pointer, just like any other function pointer!

Some Statistics

Component	Lines of Code
Parser	400
Interpreter	200
Compiler	300

Running 200000 iterations of Connway's Game of Life on a 50x50 grid:



Improving Performance

- Can we improve the IR we generate?
- Can LLVM improve the IR for us?
- Can we improve the overall system?

Improving the IR

- Optimsers work best when they have lots of information to work with.
- Split the inner loop into fixed-size blocks?
- Generate special versions of the program for edges and corners?

Make Better Use of Optimisations

- This version uses the default set of LLVM passes
- Try changing the order or explicitly adding others
- Writing new LLVM parses is quite easy - maybe you can write some specific to your language?

Part 2: Writing a Simple Optimisation

Writing a New Pass

LLVM optimisations are self-contained classes:

- `ModulePass` subclasses modify a whole module
- `FunctionPass` subclasses modify a function
- `LoopPass` subclasses modify a function
- Lots of analysis passes create information your passes can use!

Example Language-specific Passes

ARC Optimisations:

- Part of LLVM
- Elide reference counting operations in Objective-C code when not required
- Makes heavy use of LLVM's flow control analysis

GNUstep Objective-C runtime optimisations:

- Distributed with the runtime.
- Can be used by clang (Objective-C) or LanguageKit (Smalltalk)
- Cache method lookups, turn dynamic into static behaviour if safe

Writing A Simple Pass

- Memoise an expensive library call
- Call maps a string to an integer (e.g. string intern function)
- Mapping can be expensive.
- Always returns the same result.

Declaring the Pass

```
class MemoiseExample : public ModulePass {
  /// Module that we're currently optimising
  Module *M;
  /// Static cache.
  llvm::StringMap<GlobalVariable*> statics;
  // Lookup - call plus its argument
  typedef std::pair<CallInst*,std::string>
     ExampleCall;
  bool runOnFunction(Function &F);
  public:
  static char ID;
  MemoiseExample() : ModulePass(ID) {}
  virtual bool runOnModule(Module &Mod);
};
RegisterPass<MemoiseExample> X("example-pass",
        "Memoise example pass");
```

The Entry Point

```
bool MemoiseExample::runOnModule(Module &Mod) {
  statics.empty();
  M = &Mod;
  bool modified = false;

  for (auto &F : Mod) {

    if (F.isDeclaration()) { continue; }

    modified |= runOnFunction(F);
  }

  return modified;
};
```

Finding the Call

```
for (auto &i : F) {
 for (auto &b : i) {
  if (CallInst *c = dyn_cast<CallInst>(&b)) {
   if (Function *func = c->getCalledFunction()){
    if (func->getName() == "example") {
      ExampleCall lookup;
      GlobalVariable *arg =
        dyn_cast<GlobalVariable>(
        c->getOperand(0)->stripPointerCasts());
      if (0 == arg) { continue; }
      ConstantDataArray *init =
        dyn_cast<ConstantDataArray>(
        arg->getInitializer());
```

Creating the Cache

- Once we've found all of the replacement points, we can insert the caches.
- Don't do this during the search - iteration doesn't like the collection being mutated...

```
GlobalVariable *cache = statics[arg];
if (!cache) {
 cache = new GlobalVariable(*M, retTy, false,
  GlobalVariable::PrivateLinkage,
  Constant::getNullValue(retTy),
  "._cache");
  statics[arg] = cache;
}
```

Restructuring the CFG

```
BasicBlock *beforeLookupBB=lookup->getParent();
BasicBlock *lookupBB =
  SplitBlock(beforeLookupBB, lookup, this);
BasicBlock::iterator iter = lookup;
iter++;
BasicBlock *afterLookupBB =
  SplitBlock(iter->getParent(), iter, this);
removeTerminator(beforeLookupBB);
removeTerminator(lookupBB);
PHINode *phi = PHINode::Create(retTy, 2, arg,
   afterLookupBB->begin());
lookup->replaceAllUsesWith(phi);
```

Adding the Test

```
IRBuilder<> B(beforeLookupBB);
llvm::Value *cachedClass =
  B.CreateBitCast(B.CreateLoad(cache), retTy);
llvm::Value *needsLookup =
  B.CreateIsNull(cachedClass);
B.CreateCondBr(needsLookup, lookupBB,
    afterLookupBB);
B.SetInsertPoint(lookupBB);
B.CreateStore(lookup, cache);
B.CreateBr(afterLookupBB);
phi->addIncoming(cachedClass, beforeLookupBB);
phi->addIncoming(lookup, lookupBB);
```

A Simple Test

```
int example(char *foo) {
  printf("example(%s)\n", foo);
  int i=0;
  while (*foo)
    i += *(foo++);
  return i;
}
int main(void) {
  int a = example("a contrived example");
  a += example("a contrived example");
  a += example("a contrived example");
  a += example("a contrived example");
  a += example("a contrived example");
  return a;
}
```

Running the Test

```
$ clang example.c ; ./a.out ; echo $?
example(a contrived example)
example(a contrived example)
example(a contrived example)
example(a contrived example)
example(a contrived example)
199
$ clang `llvm-config --cxxflags --ldflags ` memo.cc \
  -std=c++0x -fPIC -shared -o memo.so
$ clang example.c -c -emit-llvm
$ opt -load ./memo.so -example-pass example.o | llc > e.s
$ clang e.s ; ./a.out ; echo $?
example(a contrived example)
199
```

Part 4: Using Libclang

FFI Aided by Clang

- libclang allows you to easily parse headers.
- Can get names, type encodings for functions.
- No explicit FFI
- Pragmatic Smalltalk uses this to provide a C alien: messages sent to C are turned into function calls

LanguageKit's C Alien

Smalltalk code:

```
C sqrt: 42.
C fdim: {60. 12}.
C NSLocation: l InRange: r.
```

Calls these C functions:

```
double sqrt(double x);
double fdim(double x, double y);
BOOL NSLocationInRange(NSUInteger loc, NSRange
    range);
```

Correct argument types are generated and return types interpreted automatically.

Using libclang

```
CXIndex idx = clang_createIndex(1, 1);
CXTranslationUnit tu =
  clang_createTranslationUnitFromSourceFile(idx,
      filename, argc, argv, unsavedFileCount,
     unsavedFiles);
```

- Clang uses a single shared index for cross-referencing between source files.
- A translation unit is a source file, plus includes, interpreted as if compiled with the set of command line options.
- Unsaved (in-memory) files can be passed via the last two arguments.

Using libclang

```
clang_visitChildrenWithBlock(
 clang_getTranslationUnitCursor(tu),
 ^enum CXChildVisitResult (CXCursor c, CXCursor
    parent) {
  if (c.kind == CXCursor_FunctionDecl) {
   CXString n= clang_getCursorSpelling(c);
   const char *name= clang_getCString(n);
   CXString t= clang_getDeclObjCTypeEncoding(c)
   const char *type= clang_getCString(t);
   storeFunctionNameAndType(name, type);
   clang_disposeString(n);
   clang_disposeString(t);
  }
  return CXChildVisit_Continue
 });
```

Questions?