Implementing Domain-Specific Languages with LLVM

David Chisnall

February 5, 2012

What are Domain-Specific Languages?

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

Some are *also* general-purpose programming languages.

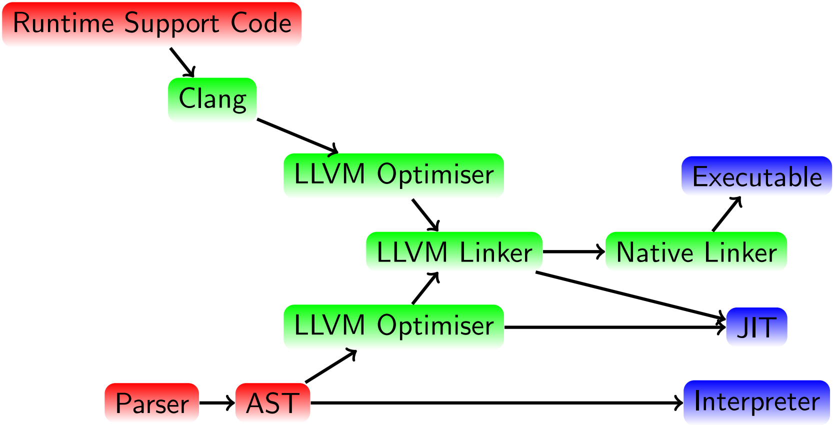
What is LLVM?

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

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 DSL Implementation



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

The Structure of an LLVM Compilation Unit

- Compilation units are LLVM Modules, each one contains one or more...
- Functions, each of which contains one or more...
- Basic Blocks, each of which contains one or more...
- Instructions

LLVM Instructions

- `alloca` allocates space on the stack
- `add` and so on: arithmetic instructions
- `jeq`, `jne`, `ret` flow control
- `call`, `invoke` structured flow control
- LLVM also provides some *intrinsic functions* for things like atomic operations

Instructions are Values

- LLVM uses an infinite SSA register set
- The result of (almost) every instruction is a register
- These can be operands to other instructions

Basic Blocks

- Start with (zero or more) `phi` instructions
- End with a flow control instruction (terminator)
- No flow control inside a basic block

Functions

- Start with an entry basic block.
- All locals should be `allocas` in the entry block.
- Must end (if it terminates) with a `ret` instruction

Hello World in LLVM

```
@.str=private constant [12 x i8] c"hello world
    \00"
@.str1=private constant [9 x i8] c"hello %s\00"
define i32 @main(i32 %argc, i8** %argv) {
  %1 = icmp eq i32 %argc, 1
  br i1 %1, label %world, label %name
world:
  %2 = getelementptr [12 x i8]* @.str, i64 0,
     i64 0
  call void @puts(i8* %2)
  ret i32 0
name:
  %3 = getelementptr inbounds i8** %argv, i64 1
  %4 = load i8** %3, align 8
  %5 = getelementptr [9 x i8]* @.str1, i64 0,
     i64 0
  call void (i8*, ...)* @printf(i8* %5, i8* %4)
  ret i32 0
}
```

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);
}
```

GEP? WTF? BBQ?

- GEP is short for GetElementPtr
- Returns a pointer to an element of a structure or array
- Does not dereference the pointer
- Architecture-agnostic representation of complex addressing modes

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

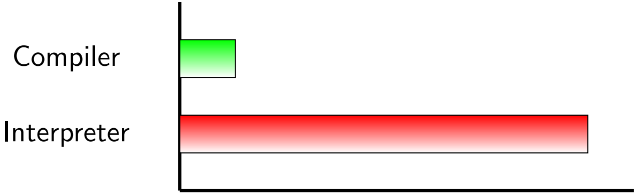
```
std::string error;
ExecutionEngine *EE = ExecutionEngine::create(
    Mod, false, &error);
if (!EE) {
  fprintf(stderr, "Error: %s\n", error.c_str());
  exit(-1);
}
return (automaton)EE->getPointerToFunction(Mod->
    getFunction("automaton"));
```

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?

Writing a New Pass

- Tutorial:
  `http://llvm.org/docs/WritingAnLLVMPass.html`
- `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

Other Libraries

- LLVM is not the end, when designing a language
- It's trivial to call into other libraries with C APIs from LLVM-generated code.
- libdispatch gives cheap concurrency
- libgc gives garbage collection
- libobjc2 gives a dynamic object model

This language is conceptually parallel - why not use libdispatch to run 16x16 blocks concurrently?

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

Questions?