Introduction to LLVM
Part 2: Further Details
Gabe Weisz

LLVM Overview
- C++ based compiler framework
- (Fairly) well documented API
- Structures to help you process programs
  - Iterators for modules, functions, blocks, uses
  - Functions to inspect data types and constants
  - Almost every class has a dump() method that prints the object to standard error
  - Tip: in gdb, use p obj->dump() to see the object

LLVM IR
- Machine independent assembly
  - Not completely – integer sizes have sizes, ir files tagged with architecture
- Arbitrary # of registers
  - Target machine specific pass does actual allocation
- Locals start with %, globals with @
- All instructions that produce values can have a name
  - Not assignments – store, branch

Iterators
- Module::iterator – module == source file, iterates through functions in the module
- Function::iterator – iterates through basic blocks in the module
- BasicBlock::iterator – iterates through instructions in a block
- Value::use_iterator – iterates through uses
  - Instructions are subclasses of values, which also include constants
More on iterators

• User::op_iterator – iterates through operands
  (Instruction is a user)
  – Tip: Many instruction classes have members for particular
    operands – ie LoadInst::getPointerOperand()
• inst_iterator – goes through instructions in a function
  – for(inst_iterator i=inst_begin(f);i!=inst_end(f);i++)
  – Declared in <Transforms/Utils/FunctionUtils.h>
• Most iterators automatically cast to a pointer to the
  object type (except inst_iterator)
• Be careful if modifying the list while iterating
  – Problem if using ++ after removing an item

Instructions

• Instruction types have specific subclasses
  – LoadInst, StoreInst, CmpInst, BranchInst, etc
  – Subclasses may have subclasses
• Exception: Most math operations have
  BinaryOperator class with a code for operation
• Alloca – allocates memory on the stack – don’t use in a loop

Moving Instructions

• EraseFromParent() – remove from basic block, drop all references, deallocates
• RemoveFromParent() – just remove from basic block.
  – Use this if you will re-attach this instruction
  – Does not drop references (clear the use list), so if you
don’t reattach you’ll get a crash in the module verifier
  – dropAllReferences will clean up
• MoveBefore / InsertBefore / InsertAfter are available

Types

• Not exactly what PL people think of as types
• Integer (size as a property), Float (different class for
  float, double, half), Arrays (can often get # elements),
  Structures (can get members), Pointers
  – Tip: type->getPointerTo() gets a pointer to a particular type
  – Booleans are i1 (1 bit integer)
• Vector types are included, and can be added, subtracted, etc. Can turn into SIMD instructions when
  compiled if the target supports them.
• GEP instruction: finds a field within a complex pointer
  type (or array)
LLVM Passes
• For assignments, don’t use LLVM passes unless instructed to
  – Want you to implement them to really understand how they work
• For projects, use whatever you want
• Analysis passes – provide information
• Transform passes – modify the program
• Module Verifier – opt automatically runs this (unless you turn it off) to make sure that you haven’t broken the module

Module Verifier
• Opt automatically runs this unless you turn it off
• Sanity checks
• You may need to break the module temporarily while working on it
• Types of binary operator parameters are the same
• Terminators (branches) only at the end of basic blocks
• Void instructions are not named (store, branch)
• Function parameters are not void type

Module Verifier (more)
• Function call arguments match function prototype
• All instructions in a basic block (refer back to erasefromparent vs removefromparent)
• Constants in a switch are the right type
• Entry node of a function has no predecessors
• PHI nodes are valid (more later)

Loop Information (-loops)
• Analysis/LoopInfo.h
• Basic blocks in a loop
• Headers, pre-headers
• Exit and exiting blocks
• Back edges
• Canonical induction variable (more later but either:
  – Starts at 0 and counts up by 1 OR
  – Starts at some number and counts down to 0
• Loop count (in many cases)
Scalar Evolution (-scalar-evolution)

- Tracks changes to variables through multiple loop nests
- Start value, step size (pseudo machine independent way with sizeof, indexof – can use with TargetData)
- Constants, add a value each iteration, multiply a value each iteration, more complicated affine progression across loops
  - Can be used to aggregate accesses into larger blocks
  - Can be used to find out if accesses go the wrong way for caches (accessing row major arrays columnwise)

Target Data (-targetdata)

- Endian-ness
- Pointer sizes
- Alignment
- Actual size in bits of variables
- Actual structure layout

Alias Analyses

```
%1 = load i32* %A store i32 5, i32* %B
%3 = add i32 %1,i32 9 store i32 %3, i32* %C
```

What happens if $A==B==C$?

Notes on writing passes

- You must declare which analysis passes you use (and possibly what you change) in getAnalysisUsage
- LLVM includes a number of passes that perform different types of alias information
- Can get information about both global and local variables
- Included passes take into account the fact that many standard C library functions don’t access memory
- The CommandLine library allows you to implement command line parameters very easily
  - Will tell you if there are conflicts for parameter names at run time. Can’t tell at compile time because passes are loaded dynamically.
Memory To Register (-mem2reg)

• Removes memory operations where possible, puts code in SSA form
• EG: C=A+B
  D=C+2
%1 = load i32* %A
%2 = load i32* %B
%3 = add i32 %1,%2
store i32 %3, i32* %C
%4 = load i32* %C
%5 = add i32 %4, 2
store i32 %5, i32* %D

SSA Basics

• SSA = Single Static Assignment = each variable (abstract register) assigned exactly 1 time
• Phi nodes – construct to handle cases where a variable may have more than one value
  – May be self referential (in loops)
  – Inside a block – select statement sometimes used
• In LLVM:
  – Must be at the beginning of the block
  – Must have exactly 1 entry for every predecessor
  – Must have at least one entry
  – May include undef values

Simplify CFG

• Removes unnecessary basic blocks by merging unconditional branches if the second block only has one predecessor
• Removes unreachable blocks
• Removes phi nodes with only a single predecessor
• Many of the conditions that this pass cleans up are created by other optimization passes

Aggressive Dead Code Elimination

• Liveness based dead code elimination – assumes code is dead unless proven otherwise
• EG: C=A+B
  D=C+2
Assume this is the only code (other than initializing A,B somewhere) and that Mem2Reg has been run
%1 = load i32* %A
%2 = load i32* %B
%3 = add i32 %1,%2
store i32 %3, i32* %C
%5 = add i32 %4, 2
store i32 %5, i32* %D
%1 = load i32* %A
%2 = load i32* %B
%3 = add i32 %1,%2
%5 = add i32 %4, 2
store i32 %5, i32* %D
Other useful passes

- Sparse conditional constant propagation – aggressively searches for constants
- Correlated propagation – replaces select statements that depend on constants
- Loop invariant code motion – moves code out of loops
- Dead global elimination
- Canonicalize Induction Variables – all loops count up from 0
- Canonicalize Loops – puts loop structure in standard form

Links

- Pass Howto: http://llvm.org/docs/WritingAnLLVMPass.html
- LLVM IR: http://llvm.org/docs/LangRef.html
- GEP: http://llvm.org/docs/GetElementPtr.html
- CommandLine library: http://llvm.org/docs/CommandLine.html
- Built in passes: http://llvm.org/docs/Passes.html
- Everything else: http://llvm.org/docs