LLVM, in Greater Detail

Thanks to Gabe Weisz for some content.
Outline

• Navigating and changing the IR
• The machine type system
• Using and writing passes
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
  - Many classes have dump() member functions that print instances to standard error
  - In GDB, use `p obj->dump()` to see the contents of obj
LLVM IR

• Mostly machine-independent assembly
  – Target triples define alignment, pointer sizes

• Arbitrary number of “registers”
  – Really, stack locations or SSA values
  – Virtual registers appear in lower-level IRs

• Locals start with %, globals with @
  – Instructions that produce values can be named
Iterators – navigating the IR

- **Module::iterator**
  - Modules are translation units
  - Iterates through functions in the module
- **Function::iterator**
  - Iterates through a function's basic blocks
- **BasicBlock::iterator**
  - Iterates through instructions in a block
- **Value::use_iterator**
  - Iterates through *uses*
  - Instructions are values; so are constants
  - How does SSA help?
- **User::op_iterator**
  - Iterates over operands.
    (Instructions are users!)
  - Many instruction classes define convenient accessors
    - LoadInst::getPointerOperand
More Iterators

• Some iterators wrap other iterators
  – `inst_iterator` walks over all instructions in a function
  – `for (inst_iterator II = inst_begin(f), IE = inst_end(f); II != IE; ++II)`
  – In `Transforms/Utils/FunctionUtils.h`

• Prefer `++i` over `i++` and precompute the end iterator
  – Especially for fancier iterators

• Most iterators automatically cast to a pointer to the object type (`inst_iterator` does not)

• Be careful about modifying the object you're iterating over during iteration
Instructions

• Instruction is subclassed for various types of operation
  – LoadInst, StoreInst, CmpInst, BranchInst, etc
• Most arithmetic operations are BinaryOperators that contain a code for the operation
• Some instructions can only appear in certain places
  – Branches are only at the end of a basic block
  – Phi instructions are only at the beginning
(Re)moving Instructions

- **eraseFromParent()**
  - Remove from basic block, drop all references, delete

- **removeFromParent()**
  - Remove from basic block
  - Use if you will re-attach the instruction
  - Does not drop references (or clear the use list), so if you don't re-attach it Bad Things will happen

- **moveBefore/insertBefore/insertAfter** are also available

- **ReplaceInstWithValue** and **ReplaceInstWithInst** are also useful to have
IR Types

• Primitive types
  - Integers (iN of N bits, N from 1 to $2^{23} - 1$)
  - Floating point (half, float, double, x86_fp80, ...)
  - Weirdos (x86mmx, void, ...)

• Derived types
  - Arrays ([# elements (>= 0) x element type])
  - Functions (returntype (paramlist))
  - Pointers (type*, type addrspace(N)*)
  - Vectors (<# elements (> 0) x element type>)
  - Structures ({ typelist }) ...
IR Types

- `getelementptr` instruction gives you the address of a field or an array cell (why have this?)

- Types in the LLVM IR are structural
  - Mostly compared by shape, not by name
    - Once allocated, there is only one 32-bit integer until the end of time
  - Only one instance of a given shape exists at once
    - Benefit?
  - Exception: “identified” structures
    - Problem: how do you write down the type of a singly-linked list?
Identified Structs

• LLVM 3 came with a redesign of the IR type system mainly over the issue of recursive and abstract types

• Literal structs are compared by shape and must not be recursive; all fields must be known
  – {i32, i32}; StructType::get

• Only identified structs can be recursive; declaration and definition of fields is separate
  – %T1 = type { type list }; StructType::create
  – type list may directly or indirectly refer to %T1
  – %T1 = type opaque - “I'll fill in the fields later.”

• LLVM may rename your identifiers
Examples of Types

[40 x i32]
[3 x [4 x i32]]

%sll =
    type { i32, %sll* }
i32 (%sll*)*

%struct.a =
    type { %struct.b* }%
%struct.b =
    type opaque

{ i32, [0 x float] }
Last words about opaque types

- Opaque types are not void* (C void* is i8*)

- Consider `%local_a = alloca %a, align 4`
  - `%a = type { %b* }`
    - `%b = type opaque`
      - OK!
  - `%a = type { %b }`
    - `%b = type opaque`
      - “Cannot allocate unsized type”
  - `%a = type { %b }`
    - `%b = type { %a }`
      - “Segmentation fault”
Passes

• For assignments, don't use provided LLVM passes unless instructed to
  – We want you to implement them yourself to understand how they really work
• For projects, use whatever you want
• Two major kinds of passes
  – Analysis: provide information
  – Transform: modify the program
Module Verifier (-verify)

• *opt* runs this automatically unless you disable it

• Sanity-checks passes
  
  – You may need to break module invariants while operating on them, eg:
    
    • Types of binary operator parameters are the same
    • Terminators (branches) only at the end of basic blocks
    • Functions are called with correct argument types
    • Instructions belong to basic blocks
    • Constants in a switch are the right type
    • Entry node has no predecessors (*and so on...*)
The mem2reg transform pass

- The LLVM IR is natively SSA
  - An Instruction is the same thing as the Value it produces
  - `%foo = inst` in the LLVM IR just gives a name to `inst` in the syntax; `%foo` does not exist inside the compiler

- It may be nontrivial for frontends to emit SSA directly

- mem2reg understands certain use patterns that frontends use to emit “variables”
```c
int ssa1() {
    int z = f() + 1;
    return z;
}
```

```c
define i32 @ssa1() nounwind {
    entry:
        %call = call i32 @f()
        %add = add nsw i32 %call, 1
        ret i32 %add
}
```

**alloca in the entry block**

**only used by load and store**
mem2reg might add SSA features

```c
int ssa2() {
    int y, z;
    y = f();
    if (y < 0)
        z = y + 1;
    else
        z = y + 2;
    return z;
}
```

define i32 @ssa2() nounwind {
    entry:
        %call = call i32 @f()
        %cmp = icmp slt i32 %call, 0
        br i1 %cmp, label %if.then, label %if.else
    if.then:
        %add = add nsw i32 %call, 1
        br label %if.end
    if.else:
        %add1 = add nsw i32 %call, 2
        br label %if.end
    if.end:
        %z.0 = phi i32 [%add, %if.then ], [%add1, %if.else ]
        ret i32 %z.0
}
```
Rules for Phi instructions

• \texttt{phi type [ val1, inedge1 ], [ val2, inedge2 ]}
  – Select \texttt{val1} if coming from \texttt{in\textit{edge1}}; \texttt{val2} if from \texttt{in\textit{edge2}}

• Phi nodes may refer to themselves (loops!) and may select undef (undefined) values for certain in-edges

• Placement requirements:
  – must be at the beginning of the block
  – must have one entry for each predecessor
  – must have at least one entry
mem2reg confuses easily

```c
int ssa3() {
    int z;
    return *(&z + 1 - 1);
}
```

```assembly
define i32 @ssa3() nounwind {
  entry:
    %z = alloca i32, align 4
    %add.ptr = getelementptr inbounds i32* %z, i32 1
    %add.ptr1 = getelementptr inbounds i32* %add.ptr, i32 -1
    %0 = load i32* %add.ptr1, align 4
    ret i32 %0
}
```

Why not make mem2reg smarter?
(note that compiling with -O2 optimizes this down to ret undef)
Loop information (-loops)

- Analysis/LoopInfo.h
- Basic blocks in a loop
- Headers and pre-headers
- Exit and exiting blocks
- Back-edges
- "Canonical induction variable"
  - Starts at 0 and counts up by 1?
  - Starts at some number and counts down to 0
- Loop count
Using an analysis pass

• #include "llvm/Analysis/LoopInfo.h"

• AU.addRequired<LoopInfo>();  
  \textit{in getAnalysisUsage()}

• LoopInfo& LI = getAnalysis<LoopInfo>(F);  
  \textit{in a function called from runOnModule}  
  \textit{with function F inside that module}

• LI.dump()  
  “Loop at depth 1 containing:  
  \%for.cond\langle\text{header}\rangle\langle\text{exiting}\rangle,\%for.body,  
  \%for.inc\langle\text{latch}\rangle” \textit{(from loop.c)}

• PassManager sequences both kinds of passes using  
  getAnalysisUsage()
Scalar Evolution (-scalar-evolution)

• Tracks changes to variables through multiple loop nests
• Gives start value, step size, kind of evolution
  – Constant
  – Add a value each iteration
  – Multiply a value each iteration
  – More complicated relationships as well
• Useful to aggregate accesses into arrays into larger blocks or to improve cache performance
Target Data (-targetdata)

- Endian-ness
- Pointer sizes
- Alignment
- Actual size (in bits) of variables
- Actual layout of structures
  - (taking into account platform alignment requirements)
Alias Analyses

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

If we know that
%A != %B != %C
we have more freedom to reordered code, promote to registers, etc.

• LLVM includes a number of passes that collect various kinds of alias information
• Can get information about both global and local variables
• Included passes take into account the behavior of the C standard library (eg, \sin() will not make new aliases)
Simplify CFG

- A cleanup pass
- Removes unnecessary basic blocks by merging unconditional branches if the second block has only one predecessor
- Removes unreachable blocks
- Removes Phi instructions in blocks with single predecessors
Other useful passes

- Liveness-based dead code elimination
  - Assumes code is dead unless proven otherwise
- Sparse conditional constant propagation
  - Aggressively search for constants
- Correlated propagation
  - Replace select instructions that select on a constant
- Loop invariant code motion
  - Move code out of loops where possible
- Dead global elimination
- Canonicalize induction variables
  - All loops start from 0
- Canonicalize loops
  - Put loop structures in standard form
Notes on Writing passes

• Declare which passes you use (and what your pass mutates) in getAnalysisUsage

• The CommandLine library allows you to add command line parameters very quickly
  – Conflicts in parameter names won't show up until runtime, since passes are loaded dynamically

• Be mindful of correctness; the module verifier is like a syntax checker
  – Does your pass make sense in a multithreaded environment?
Transformations and memory

- For regular loads/stores, LLVM forbids introducing new stores to externally observable locations:

```c
int x;
void f(int* a) {
    for (int i = 0; i < 100; i++) {
        if (a[i])
            x += 1;
    }
}
```

- volatile marks memory operations that cannot be reordered (wrt volatile operations) or removed

- To support new features in C++11, LLVM provides other *atomic orderings* that can be applied to loads and stores

```c
int x;
void f(int* a) {
    int xtemp = x;
    for (int i = 0; i < 100; i++) {
        if (a[i])
            xtemp += 1;
    }
    x = xtemp;
}
```
Projects using LLVM

• Just a few from llvm.org:
  – Clang: a C-family language frontend
  – LLDB: an improved debugger using Clang data
  – vmkit: building Java/.NET VMs
  – klee: a symbolic virtual machine for LLVM IR

• Emscripten: LLVM bitcode → JavaScript
  – http://emscripten.org/

• Vellvm: a formalization of the LLVM IR
  – http://www.cis.upenn.edu/~jianzhou/Vellvm/
Links

• When in doubt, read the documentation—and the code!
  – http://llvm.org/doxygen/

• Articles on the LLVM site are very useful
  – http://llvm.org/docs/Passes.html
  – http://llvm.org/docs/Atomics.html
  – http://llvm.org/docs/LangRef.html
Supplemental
High-level view of atomic orderings

- NotAtomic – ordinary loads/stores; races are undefined
- Unordered – races have “somewhat sane” results
  - A load cannot see a value which was never stored
  - May not be supported for all types on all platforms
  - Used for shared variables in Java, “safe” languages
- Monotonic – single locations have consistent order
- Acquire/Release – when paired together, strong enough to write a lock
- SequentiallyConsistent – Acquire loads, Release stores, and all SequentiallyConsistent operations have a total order
  - Java volatile
Atomic orderings and you

• If your project is about fine-grained parallelism or lock-free data structures, you need to think about these things
  – LLVM also exposes cmpxchg, atomicrmw, fence
  – LLVM does not expose LL/SC

• If not, just don't, in general:
  – reorder LLVM volatile operations wrt other volatile operations
  – introduce new stores to (shared) locations that would not have been previously stored to

• More details at http://llvm.org/docs/LangRef.html#memmodel
Your module, post-IL

- LLVM still has to generate machine code!
- Your module goes through ~3 more stages:

  LLVM IL
  →
  SelectionDAG
  →
  MachineFunction
  →
  Object File
define i32 @foo(i32 %a, i32 %b) nounwind {
  entry:
      %cmp = icmp sgt i32 %a, %b
      br i1 %cmp, label %if.then, label %if.else

  if.then:
      %add = add nsw i32 %a, %b
      br label %return

  if.else:
      %sub = sub nsw i32 %a, %b
      br label %return

  return:
      %retval.0 = phi i32 [ %add, %if.then ],
                   [ %sub, %if.else ]
      ret i32 %retval.0
}
LLVM IL → SelectionDAG

if.then:
%add = add nsw i32 %a, %b
br label %return

return:
%retval.0 = phi i32 ...
ret i32 %retval.0
SelectionDAG transformations

- Build initial DAG from LLVM IR
- Simplify!
- Legalize types (vectors → scalars)
- Simplify!
- Legalize ops (x86 doesn't do byte-size CMOVs)
- Simplify!
- Select instructions
- Schedule instructions
Instruction selection
Target-Independent Code Generation

- There are a lot of different machines!
  - And even on x86, there are a lot of different ADDs!
- Make the process data-driven
The `tablegen` tool is run when compiling the LLVM library for each target. It accepts a custom text-based record description format and generates C++ definitions using various backends. For example:

- `defm ADD : ArithBinOp_RF<0x00, 0x02, 0x04, "add", MRM0r, MRM0m, X86add_flag, add, 1, 1>;`
- `ArithBinOp_RF` is actually another macro...

There is still a lot of human-written code in the backends (X86 instruction encoding, for one)
def ADD32rr {
    Format BinOpRR:f = MRMDestReg;
    Domain I:d = GenericDomain;
    string Namespace = "X86";
    dag OutOperandList = (outs GR32:$dst);
    dag InOperandList = (ins GR32:$src1, GR32:$src2);
    string AsmString = "add{l}  {$src2, $src1|$src1, $src2}"
    list<dag> Pattern =
        [(set GR32:$dst, EFLAGS,
          (X86add_flag GR32:$src1, GR32:$src2))];
    list<Register> Uses = [];
    list<Register> Defs = [EFLAGS];
    list<Predicate> Predicates = [];
    InstrItinClass Itinerary = IIC_BIN_NONMEM;
    string Constraints = "$src1 = $dst";
    bits<8> Opcode = { 0, 0, 0, 0, 0, 0, 0, 1 };
    Format Form = MRMDestReg;
    bits<6> FormBits = { 0, 0, 0, 0, 1, 1 };
    ... lots more ...
Scheduling

We have choices to make
(here, which CopyFromReg executes first?)
Chains add control dependency.

Glue forbids breaking up instructions.
Lowering to MC

BB#1: derived from LLVM BB %if.then

%vreg0<def, tied1> = ADD32rr
%vreg3<tied0>,
%vreg4,
%EFLAGS<imp-def,dead>;
GR32:%vreg0,%vreg3,%vreg4
JMP_4 <BB#3>

BB#3: derived from LLVM BB %return

%vreg2<def> = PHI
%vreg1, <BB#2>, %vreg0, <BB#1>;
GR32:%vreg2,%vreg1,%vreg0
%EAX<def> = COPY %vreg2; GR32:%vreg2
RET
MC transformations

• SSA-based MC optimizations
• Register allocation
  – 2AC correction and Leave SSA; copy coalescing; add spillcode
• Prolog/epilog insertion
• Stack layout
• Last-chance MC optimizations/spillcode scheduling
• Encoding
Thanks, abstraction!

You don't have to look here! *(Unless you're studying it...)*