Using LLVM For Program Transformation

ANDREW RUEF
UNIVERSITY OF MARYLAND
COMPUTER SCIENCE
LLVM Overview

- Research project at UIUC
- Modular compiler tool chain
- Integrated in many open source and commercial projects
- Licensed under an open-source license
Introduction
Components of LLVM

- Mid-level compiler Intermediate Representation (IR)
- C/C++ compiler frontend (clang)
- Target-specific (X86, ARM, etc) code generators

- Divide between ‘clang’ and ‘LLVM’
- Clang is a C/C++ compiler with an LLVM backend
- LLVM is ‘everything else’
Todays Agenda

- We’ll talk about existing LLVM tools
- We’ll do a few demos using those tools
- We’ll talk about how to build tools on top of LLVM
- We’ll build two analysis tools
- We’ll look at a program re-writing tool
Lab: Where we’re going

- **clang** – C language frontend, translates C into LLVM bitcode
- **opt** – Analyze and transform LLVM bitcode
- **llc** – Code generator for LLVM bitcode to native code
$ clang -c -emit-llvm -o test.bc test.c
$ opt -O1 -o test.bc test.bc
$ llc -o test.s test.bc
$ gcc -o test test.s
Lab: What just happened?

- Full translation of C program to executable program
- At each stage we can look at what the compiler infrastructure is doing
  - C to un-optimized bitcode
  - Optimized bitcode
  - Machine code
  - Executable
- Very good blog post on the life of an LLVM instruction
LLVM Intermediate Representation
Lab: Find Non-Constant Format String

- **Condition to check for:**
  - Any time the first parameter to `printf`, `sprintf` (others?) is non-constant, alert for potential security badness

- Can we statically detect this in LLVM IR?
Algorithm For Detection

- Visit every call instruction in the program
- Ask if that call instruction is a format-string accepting routine
- If it is, retrieve the first parameter
- If the first parameter is not a constant global, raise an alert
Structure of Provided Driver

- Very basic driver that uses a PassManager
- Reads in LLVM bitcode and runs the VarPrintf pass on it
- Produce bitcode file using `clang -c -emit-llvm`
- Using the driver might seem clunky, this is easier than integrating with opt
- The pass can later be integrated with opt
Building the drivers

$ cd tutorial
$ mkdir build
$ cd build
$ cmake -DLLVM_ROOT=/usr/local ..
$ make
CMake

- CMake is a “meta make”
  - Why? Why not
- CMake generates your build environment
  - Makefiles
  - XCode solution
  - Visual Studio solution
- CMake has its own build specification system for describing building code
  - It might be saner than what you are used to
- LLVM can be built with cmake or automake/autoconf
LLVM Intermediate Representation

- Language allows for expression of computation
- Instructions produce unique values
- Collection of statements:
  - `%5 = add nsw i32 %3, %4`
    - `%N` — a value
    - `add` — a binary instruction
    - `nsw` — no signed wrap
- The language is Static Single Assignment (SSA)
- Values defined by statements are never re-defined
A compilation unit is a Module, contains functions
A function is a Function, contains basic blocks
A basic block is a BasicBlock, contains instructions
An instruction is an Instruction
Instructions can contain operands, each is a Value
All of the above, except Module, is a Value
Types

- No implicit casting in LLVM IR, all values must be explicitly converted
- All values have a static type
- Integers are specified at arbitrary bitwidth
  - `i1, i2, i3, ... , i32, ... i398`
- Floating point types
- Derived types specify arrays, vectors, functions, pointers, structures
  - Structures have types like `{i32, i32, i8}
  - Pointers have types like “pointer to i32”
Note on Integer Types

- There are no signed or unsigned integers
- LLVM views integers as bit vectors
- Frontends destroyed signed/unsigned information
  - Really, C programmers destroyed signed/unsigned information...
- Research prototypes exist that analyze integer wrapping in LLVM IR (http://code.google.com/p/wrapped-intervals/)
- Operations are interpreted as signed or unsigned based on instructions they are used in
Memory Model

- LLVM has a low level view of memory
  - Just a key -> value map
  - Keys are pointer values
  - Values stored in LLVM memory must be integers, floating point, pointers, vectors, structures, or arrays

- LLVM has a concept of creating function-local memory via alloca
The Module

- Highest level concept
- Contains a set of global values
  - Global variables
  - Functions
The Function

- Name
- Argument list
- Return type
- Calling convention
- Extends from `GlobalValue`, has properties of linkage visibility
The BasicBlock

- Contains a list of Instructions
- All BasicBlocks must end in a TerminatorInst
- BasicBlocks descend from values, and are used as values in branching instructions
The Instruction

- Terminator instructions
- Binary instructions
- Bitwise instructions
- Aggregate instructions
- Memory instructions
- Type conversion instructions
- Control and misc instructions
Language By Example

Produced with `opt -dot-cfg -o fib.bc fib.bc` and `graphviz`
Language By Example, Part 2

```assembly
%0:
  %1 = icmp eq %struct._Foo* %k, null
  br i1 %1, label %._crit_edge, label %._lr.ph

.T .F

._lr.ph:
  %i.08 = phi i32 [ %2, %._lr.ph ], [ 1, %0 ]
  %acc.07 = phi i32 [ %7, %._lr.ph ], [ 0, %0 ]
  %cur.06 = phi %struct._Foo* [ %9, %._lr.ph ], [ %k, %0 ]
  %2 = add nsw i32 %i.08, 1
  %3 = getelementptr inbounds %struct._Foo* %cur.06, i64 0, i32 1
  %4 = load i32* %3, align 4
  %5 = shl i32 %4, %2
  %6 = mul nsw i32 %5, %b
  %7 = add nsw i32 %6, %acc.07
  %8 = getelementptr inbounds %struct._Foo* %cur.06, i64 0, i32 0
  %9 = load %struct._Foo** %8, align 8
  %10 = icmp eq %struct._Foo* %9, null
  br i1 %10, label %._crit_edge, label %._lr.ph

.T .F

._crit_edge:
  %acc.0.lcssa = phi i32 [ 0, %0 ], [ %7, %._lr.ph ]
  ret i32 %acc.0.lcssa
```

CFG for 'xform_all' function
Static Single Assignment

- LLVM contains a pass to promote variable-using functions to value-using functions
- Once transformed by this pass, an LLVM module is in SSA form
- Most LLVM analyses and transformations expect to operate on an SSA IR
- SSA allows for Def-Use and Use-Def chain analysis
int foo(int a, int b) {
    int i = a;
    int j = b;

    return i+j+1;
}
define i32 @foo(i32 %a, i32 %b) nounwind uwtable ssp {
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  %i = alloca i32, align 4
  %j = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  %0 = load i32* %a.addr, align 4
  store i32 %0, i32* %i, align 4
  %1 = load i32* %b.addr, align 4
  store i32 %1, i32* %j, align 4
  %2 = load i32* %i, align 4
  %3 = load i32* %j, align 4
  %add = add nsw i32 %2, %3
  %add1 = add nsw i32 %add, 1
  ret i32 %add1
}
define i32 @foo(i32 %a, i32 %b) nounwind uwtable ssp {
entry:
  %add = add nsw i32 %a, %b
  %add1 = add nsw i32 %add, 1
ret i32 %add1
}
The Phi-Node

- To support conditional assignments, we introduce an imaginary function
- Phi defines a value and accepts a list of tuples as an argument
- Each tuple is a (BasicBlock * Value)
- Interpret the phi node as defining a value conditionally based on the previous basic block
int foo(int a, int b) {
    int r;

    if( a > b )
        r = a;
    else
        r = b;

    return r;
}
define i32 @foo(i32 %a, i32 %b) nounwind uwtable ssp {
entry:
    %a.addr = alloca i32, align 4
    %b.addr = alloca i32, align 4
    %r = alloca i32, align 4
    store i32 %a, i32* %a.addr, align 4
    store i32 %b, i32* %b.addr, align 4
    %0 = load i32* %a.addr, align 4
    %1 = load i32* %b.addr, align 4
    %cmp = icmp sgt i32 %0, %1
    br i1 %cmp, label %if.then, label %if.else

if.then:
    %2 = load i32* %a.addr, align 4
    store i32 %2, i32* %r, align 4
    br label %if.end

if.else:
    %3 = load i32* %b.addr, align 4
    store i32 %3, i32* %r, align 4
    br label %if.end

if.end:
    %4 = load i32* %r, align 4
    ret i32 %4
}
define i32 @foo(i32 %a, i32 %b) nounwind uwtable ssp {
  entry: 
    %cmp = icmp sgt i32 %a, %b
    br i1 %cmp, label %if.then, label %if.else
  
  if.then: br label %if.end
  
  if.else: br label %if.end
  
  if.end: %r.0 = phi i32 [ %a, %if.then ], [ %b, %if.else ] 
    ret i32 %r.0
}
int aa(int a, int b) {
    int i = 0;
    int k = 0;
    while( k < b) {
        i += a;
    }
    return i;
}
LLVM CFG

entry:
  br label %while.cond

while.cond:
  %i.0 = phi i32 [ 0, %entry ], [ %add, %while.body ]
  %cmp = icmp slt i32 0, %b
  br i1 %cmp, label %while.body, label %while.end

while.body:
  %add = add nsw i32 %i.0, %a
  br label %while.cond

while.end:
  ret i32 %i.0

CFG for 'aa' function
The GetElementPtr instruction

- An instruction so frequently misunderstood, it has its own documentation page about how it is misunderstood
- Frequently abbreviated as GEP
- GEP instructions compute offsets from pointer bases
  - Similar to ‘lea’ instructions in X86 assembler
- GEP instructions are type aware
  - Asking for ‘the 5th field’ of a pointer to structure operand will ‘do the right thing’
Well-Formed LLVM

- There are specific rules as to what constitutes “Well-Formed” LLVM
  - Phi-nodes dominate their uses
  - Instruction arguments are defined before use
  - All blocks end in a terminator
  - All branch targets are defined values
- There is an automatic verification pass that will alert when IR is not well formed
C++ API
Value Hierarchy

- Value has a very rich class hierarchy
- LLVM API allows the manipulation of every Value
- Any degree of transformation is possible
Everything From Value

- Every item contained in a Module inherits from Value
- This allows for some useful APIs
  - Def-Use / Use-Def iteration
  - Replace any Value with another Value
  - Sub
- Allows for classification
  - Instructions can be UnaryInstructions or BinaryInstructions
  - GlobalValues can be Functions or GlobalVariables
LLVM Context

- Frequent argument to LLVM API functions
- These can normally be retrieved from a Value via `getContext`
  - There is also a `getGlobalContext`
- The same LLVMContext should always be used across code that interacts with the same Values
  - LLVM objects are created in a specific context and are unique by pointer values
  - For example, type objects can be pointer-compared for equality between types of different instructions
An evil C++ concept

If you have a function that accepts a parameter of an abstract class and it could be one of any specific implementations, how to choose?

“Normal” C++ methods
- `dynamic_cast<T>` and friends

Compiler stores information about object types off to the side so that it can be used at run-time
LLVM and Run Time Type Information

- The LLVM codebase implements its own RTTI for LLVM objects
  - When writing passes, you use LLVM specific helpers
  - `isa<T>` - True or false if pointer/reference is of type T
  - `cast<T>` - “Checked cast”, asserts on failure if not type T
  - `dyn_cast<T>` - unchecked cast, null if not type T
- The project advises you not to use big chains of these to approximate ‘match’ from ML
- Instead they give you a Visitor pattern (yay)
- You might find these insufficient (or distasteful)
Common Patterns

- “Iterate over BasicBlock in a Function”
  - Use `begin()`, `end()` iterators of `Function`
- “Iterate over Instructions in a Function”
  - Use `inst_iterator`
- “Iterate over Def-Use chains”
  - Use `use_begin`, `use_end`
InstVisitor

- Pattern to avoid giant blocks of
  
  \[
  \text{if}(T \ *n = \text{dyn\_cast}<T>(\text{foo}))
  \]
- Inherit from InstVisitor class and define a visitTInst method
- Could work for your purposes
- Could confuse control flow even more
Including LLVM In Your Project

- **llvm-config** – executable that will provide useful info about the installed LLVM
- Provide paths to headers, library files, etc
- If LLVM is built with Cmake, it will add a `FindLLVM.cmake` to your `/usr/share`
- Compiling your code with `–fno-rtti` will probably be required
- If you compiled LLVM yourself, you can pass `LLVM_REQUIRES.RegisterType` to cmake
- Needed if combining boost and llvm
Passes and transformations
In the previous lab, we wrote a pass. Compiling is the act of passing over and analyzing/transforming IR. Most things that happen in LLVM happen in the context of a pass. Passes can have complicated actions.
Pass Dependencies

- Passes can depend on the output of other passes
  - Analysis passes for alias analysis
- Passes note their dependencies on other passes
  - By overriding the `getAnalysisUsage` method
- PassManager figures out the dependency graph
  - It also attempts to optimize the traversal of the graph
- Each Pass returns a bool, PassManager runs until everyone stops
Pass Manager

- **PassManager** performs dependency maintenance
  - Note that **PassManager** invocations could be multi-threaded!
  - Importance of multiple LLVMContexts
- **PassManager** also performs optimizations of pass ordering
- **PassManager** defines different kinds of Passes that can be run
  - **ModulePass** – Run on entire module
  - **FunctionPass** – Run on individual functions
  - **BasicBlockpass** – Run on individual basic blocks
Pass Rules

- Non-analysis passes should not ‘remember’ any information about a function or basic block
- Analysis passes should remember some information
  - Otherwise why run them
- Transformation passes should be idempotent
Lab: Escape Analysis

- If a variable is allocated on the local stack, a pointer to that variable should not outlive the stack.
- This could happen if a pointer to a local is returned or assigned to a global.
- clang currently includes a check for this, but the check is kind of busted.
Algorithm For Escape Analysis

- Populate a set of values that escape the function via return or store
- Traverse the set checking for *alloca*-ed values in the Values descending from the escapes
Structure of Provided Driver

- Driver is laid out similarly to before
- Collection of tests are included
Projects built on LLVM

- Google AddressSanitizer/ThreadSanitizer
  - http://code.google.com/p/address-sanitizer/

- Utah Integer Overflow Checker
  - http://embed.cs.utah.edu/ioc/

- Emscripten, LLVM to Javascript
  - https://github.com/kripken/emscripten/wiki

- Dagger, decompilation from x86 to LLVM
Important LLVM subprojects

- poolalloc – field-sensitive, context-sensitive alias analysis
- lldb – llvm debugger
- klee – symbolic execution for LLVM

- FreeBSD compiles with clang, soon will switch to building exclusively with clang
Conclusion

- LLVM enables powerful transformations
- Includes an “industry grade” C/C++ frontend
  - clang is default compiler on OSX, supported by Apple
  - Can compile much of Linux userspace
- Well defined Intermediate Language
- Modular and pluggable framework for analysis and transformation
Project Documentation

- Good documentation online
  - [http://www.llvm.org/docs](http://www.llvm.org/docs)
- Documentation covers many aspects of the LLVM project
  - Programmers manual details finer points of the C++ API
  - Language reference is ultimate source for language details and semantics
- Relatively responsive IRC channel on OFTC
- Active and responsive mailing list