A Brief Introduction to Using LLVM

Nick Sumner

Spring 2013

What is LLVM?

- A compiler?

What is LLVM?

- A compiler?
- A set of formats, libraries and tools.

What is LLVM?

- A compiler?
- A set of formats, libraries and tools.
  - A simple, typed IR (*bitcode*)
  - Program analysis / optimization libraries
  - Machine code generation libraries
  - Tools that compose the libraries to perform task

What is LLVM?

- A compiler?
- A set of formats, libraries and tools.
  - A simple, typed IR (*bitcode*)
  - Program analysis / optimization libraries
  - Machine code generation libraries
  - Tools that compose the libraries to perform tasks
- Easy to add / remove / change functionality

How will you be using it?

- Compiling programs to bitcode:

```
clang -g -c -emit-llvm <sourcefile> -o <bitcode>.bc
```

How will you be using it?

- Compiling programs to bitcode:

```
clang -g -c -emit-llvm <sourcefile> -o <bitcode>.bc
```

- Analyzing the bitcode:

```
opt -load <plugin>.so --<plugin> -analyze <bitcode>.bc
```

How will you be using it?

- Compiling programs to bitcode:

```
clang -g -c -emit-llvm <sourcefile> -o <bitcode>.bc
```

- Analyzing the bitcode:

```
opt -load <plugin>.so --<plugin> -analyze <bitcode>.bc
```

- Reporting properties of the program:

```
[main] : [A], [C], [F]
[A] : [B]
[C] : [E], [D]
```

What is LLVM Bitcode?

- A (Relatively) Simple IR

Code

```c
#include<stdio.h>

void
foo(unsigned e) {
  for (unsigned i = 0; i < e; ++i) {
    printf("Hello\n");
  }
}

int
main(int argc, char **argv) {
  foo(argc);
  return 0;
}
```

clang -c -emit-llvm
(and llvm-dis)

IR

```llvm
@str = private constant [6 x i8] c"Hello\00"

define void @foo(i32 %e) {
  %1 = icmp eq i32 %e, 0
  br i1 %1, label %._crit_edge, label %.lr.ph

.lr.ph:                          ; preds = %.lr.ph, %0
  %i = phi i32 [ %2, %.lr.ph ], [ 0, %0 ]
  %str1 = getelementptr
                  [6 x i8]* @str, i64 0, i64 0
  %puts = tail call i32 @puts(i8* %str1)
  %2 = add i32 %i, 1
  %cond = icmp eq i32 %2, %e
  br i1 %cond, label %.exit, label %.lr.ph

.exit:                           ; preds = %.lr.ph, %0
  ret void
}

define i32 @main(i32 %argc, i8** %argv) {
  tail call void @foo(i32 %argc)
  ret i32 0
}
```

What is LLVM Bitcode?

- A (Relatively) Simple IR

```c
#include<stdio.h>

void
foo(unsigned e) {
  for (unsigned i = 0; i < e; ++i) {
    printf("Hello\n");
  }
}

int
main(int argc, char **argv) {
  foo(argc);
  return 0;
}
```

clang -c -emit-llvm
(and llvm-dis)

```llvm
@str = private constant [6 x i8] c"Hello\00"

define void @foo(i32 %e) {
  %1 = icmp eq i32 %e, 0
  br i1 %1, label %._crit_edge, label %.lr.ph

.lr.ph:                         ; preds = %.lr.ph, %0
  %i = phi i32 [ %2, %.lr.ph ], [ 0, %0 ]
  %str1 = getelementptr
                   [6 x i8]* @str, i64 0, i64 0
  %puts = tail call i32 @puts(i8* %str1)
  %2 = add i32 %i, 1
  %cond = icmp eq i32 %2, %e
  br i1 %cond, label %.exit, label %.lr.ph

.exit:                          ; preds = %.lr.ph, %0
  ret void
}

define i32 @main(i32 %argc, i8** %argv) {
  tail call void @foo(i32 %argc)
  ret i32 0
}
```

What is LLVM Bitcode?

- A (Relatively) Simple IR

```c
#include<stdio.h>

void
foo(unsigned e) {
  for (unsigned i = 0; i < e; ++i) {
    printf("Hello\n");
  }
}

int
main(int argc, char **argv) {
  foo(argc);
  return 0;
}
```

Functions

```llvm
@str = private constant [6 x i8] c"Hello\00"

define void @foo(i32 %e) {
  %1 = icmp eq i32 %e, 0
  br i1 %1, label %._crit_edge, label %.lr.ph

.lr.ph:                          ; preds = %.lr.ph, %0
  %i = phi i32 [ %2, %.lr.ph ], [ 0, %0 ]
  %str1 = getelementptr
                  [6 x i8]* @str, i64 0, i64 0
  %puts = tail call i32 @puts(i8* %str1)
  %2 = add i32 %i, 1
  %cond = icmp eq i32 %2, %e
  br i1 %cond, label %.exit, label %.lr.ph

.exit:                           ; preds = %.lr.ph, %0
  ret void
}

define i32 @main(i32 %argc, i8** %argv) {
  tail call void @foo(i32 %argc)
  ret i32 0
}
```

What is LLVM Bitcode?

- A (Relatively) Simple IR

```c
#include<stdio.h>

void
foo(unsigned e) {
  for (unsigned i = 0; i < e; ++i) {
    printf("Hello\n");
  }
}

int
main(int argc, char **argv) {
  foo(argc);
  return 0;
}
```

```llvm
@str = private constant [6 x i8] c"Hello\00"

define void @foo(i32 %e) {
  %1 = icmp eq i32 %e, 0
  br i1 %1, label %._crit_edge, label %.lr.ph

.lr.ph:                          ; preds = %.lr.ph, %0
  %i = phi i32 [ %2, %.lr.ph ], [ 0, %0 ]
  %str1 = getelementptr
                  [6 x i8]* @str, i64 0, i64 0
  %puts = tail call i32 @puts(i8* %str1)
  %2 = add i32 %i, 1
  %cond = icmp eq i32 %2, %e
  br i1 %cond, label %.exit, label %.lr.ph

.exit:                           ; preds = %.lr.ph, %0
  ret void
}

define i32 @main(i32 %argc, i8** %argv) {
  tail call void @foo(i32 %argc)
  ret i32 0
}
```

Basic Blocks

What is LLVM Bitcode?

- A (Relatively) Simple IR

```
#include<stdio.h>

void
foo(unsigned e) {
  for (unsigned i = 0; i < e; ++i) {
    printf("Hello\n");
  }
}

int
main(int argc, char **argv) {
  foo(argc);
  return 0;
}
```

```
@str = private constant [6 x i8] c"Hello\00"

define void @foo(i32 %e) {
  %1 = icmp eq i32 %e, 0
  br i1 %1, label %._crit_edge, label %.lr.ph

.lr.ph:                          ; preds = %.lr.ph, %0
  %i = phi i32 [ %2, %.lr.ph ], [ 0, %0 ]
  %str1 = getelementptr
                  [6 x i8]* @str, i64 0, i64 0
          call i32 @puts(i8* %str1)
  %2 = add i32 %i, 1
  %cond = icmp eq i32 %2, %e
  br i1 %cond, label %.exit, label %.lr.ph

.exit:                           ; preds = %.lr.ph, %0
  ret void
}

define i32 @main(i32 %argc, i8** %argv) {
  tail call void @foo(i32 %argc)
  ret i32 0
}
```

labels & predecessors

Basic Blocks

What is LLVM Bitcode?

- A (Relatively) Simple IR

```c
#include<stdio.h>

void
foo(unsigned e) {
  for (unsigned i = 0; i < e; ++i) {
    printf("Hello\n");
  }
}

int
main(int argc, char **argv) {
  foo(argc);
  return 0;
}
```

```llvm
@str = private constant [6 x i8] c"Hello\00"

define void @foo(i32 %e) {
  %1 = icmp eq i32 %e, 0
  br i1 %1, label %._crit_edge, label %.lr.ph

.lr.ph:                              ; preds = %.lr.ph, %0
  %i = phi i32 [ %2, %.lr.ph ], [ 0, %0 ]
  %str1 = getelementptr
                  [6 x i8]* @str, i64 0, i64 0
                  call i32 @puts(i8* %str1)
  %2 = add i32 %i, 1
  %cond = icmp eq i32 %2, %e
  br i1 %cond, label %.exit, label %.lr.ph

.exit:                               ; preds = %.lr.ph, %0
  ret void
}

define i32 @main(i32 %argc, i8** %argv) {
  tail call void @foo(i32 %argc)
  ret i32 0
}
```

branches & successors

Basic Blocks

What is LLVM Bitcode?

- A (Relatively) Simple IR

```
#include<stdio.h>

void
foo(unsigned e) {
  for (unsigned i = 0; i < e; ++i) {
    printf("Hello\n");
  }
}

int
main(int argc, char **argv) {
  foo(argc);
  return 0;
}
```

```
@str = private constant [6 x i8] c"Hello\00"

define void @foo(i32 %e) {
  %1 = icmp eq i32 %e, 0
  br i1 %1, label %._crit_edge, label %.lr.ph

.lr.ph:                        ; preds = %.lr.ph, %0
  %i = phi i32 [ %2, %.lr.ph ], [ 0, %0 ]
  %str1 = getelementptr
                  [6 x i8]* @str, i64 0, i64 0
  %puts = tail call i32 @puts(i8* %str1)
  %2 = add i32 %i, 1
  %cond = icmp eq i32 %2, %e
  br i1 %cond, label %.exit, label %.lr.ph

.exit:                         ; preds = %.lr.ph, %0
  ret void
}

define i32 @main(i32 %argc, i8** %argv) {
  tail call void @foo(i32 %argc)
  ret i32 0
}
```

Instructions

Inspecting Bitcode

- LLVM libraries help examine the bitcode
  - Easy to examine and/or manipulate

Inspecting Bitcode

- LLVM libraries help examine the bitcode
  - Easy to examine and/or manipulate

```
Module &module = ...;
for (Function &fun : module) {
  for (BasicBlock &bb : fun) {
    for (Instruction &i : bb) {

...
```

Iterate over the:
- Functions in a Module
- BasicBlocks in a Function
- Instructions in a BasicBlock

Inspecting Bitcode

- LLVM libraries help examine the bitcode
  - Easy to examine and/or manipulate
  - Many helpers (e.g. CallSite,)

```
Module &module = ...;
for (Function &fun : module) {
  for (BasicBlock &bb : fun) {
    for (Instruction &i : bb) {
      CallSite cs(&i);
      if (!cs.getInstruction()) {
        continue;
      }



...
```

CallSite helps you extract information from Call and Invoke instructions.

Inspecting Bitcode

- LLVM libraries help examine the bitcode
  - Easy to examine and/or manipulate
  - Many helpers (e.g. CallSite, outs(),)

```
Module &module = ...;
for (Function &fun : module) {
  for (BasicBlock &bb : fun) {
    for (Instruction &i : bb) {
      CallSite cs(&i);
      if (!cs.getInstruction()) {
        continue;
      }
      outs() << "Found a function call: " << i << "\n";



...
```

Inspecting Bitcode

- LLVM libraries help examine the bitcode
  - Easy to examine and/or manipulate
  - Many helpers (e.g. CallSite, outs(), dyn_cast)

```
Module &module = ...;
for (Function &fun : module) {
  for (BasicBlock &bb : fun) {
    for (Instruction &i : bb) {
      CallSite cs(&i);
      if (!cs.getInstruction()) {
        continue;
      }
      outs() << "Found a function call: " << i << "\n";
      Value *called = cs.getCalledValue()->stripPointerCasts();
      if (Function *f = dyn_cast<Function>(called)) {
        outs() << "Direct call to function: " << f->getName() << "\n";
...
```

dyn_cast() efficiently checks the runtime types of LLVM IR components.

Dealing with SSA

- You may ask where certain values came from
  - Useful for tracking dependencies
  - “Where was this variable defined?”

Dealing with SSA

- You may ask where certain values came from
- LLVM IR is in SSA form
  - How many acronyms can I fit into one line?
  - What does this mean?
  - Why does it matter?

Dealing with SSA

- You may ask where certain values came from
- LLVM IR is in SSA form
  - How many acronyms can I fit into one line?
  - What does this mean?
  - Why does it matter?

```
void foo()
  unsigned i = 0;
  while (i < 10) {
    i = i + 1;
  }
}
```

Dealing with SSA

- You may ask where certain values came from
- LLVM IR is in SSA form
  - How many acronyms can I fit into one line?
  - What does this mean?
  - Why does it matter?

```
void foo()
  unsigned i = 0;
  while (i < 10) {
    i = i + 1;
  }
}
```

What is the single definition of `i` at this point?

Dealing with SSA

- Thus the phi instruction
  - It selects which of the definitions to use
  - Always at the start of a basic block

Dealing with SSA

- Thus the phi instruction
  - It selects which of the definitions to use
  - Always at the start of a basic block

```
void foo()
  unsigned i = 0;
  while (i < 10) {
    i = i + 1;
  }
}
```

```
define void @foo() {
  br label %1

; <label>:1
  %i.phi = phi i32 [ 0, %0 ], [ %2, %1 ]
  %2 = add i32 %i.phi, 1
  %exitcond = icmp eq i32 %2, 10
  br i1 %exitcond, label %3, label %1

; <label>:3
  ret void
}
```

Dealing with SSA

- Thus the phi instruction
  - It selects which of the definitions to use
  - Always at the start of a basic block

```
void foo()
  unsigned i = 0;
  while (i < 10) {
    i = i + 1;
  }
}
```

```
define void @foo() {
  br label %1

; <label>:1
  %i.phi = phi i32 [ 0, %0 ], [ %2, %1 ]
  %2 = add i32 %i.phi, 1
  %exitcond = icmp eq i32 %2, 10
  br i1 %exitcond, label %3, label %1

; <label>:3
  ret void
}
```

Dependencies in General

- You can loop over the values an instruction uses

```
for (auto i = inst->op_begin(), e = inst->op_end(); i != e; ++i) {
  // inst uses the Value i
}
```

Dependencies in General

- You can loop over the values an instruction uses

```
for (auto i = inst->op_begin(), e = inst->op_end(); i != e; ++i) {
  // inst uses the Value i
}
```

for %a = %b + %c:

[%b, %c]

Dependencies in General

- You can loop over the values an instruction uses

```
for (auto i = inst->op_begin(), e = inst->op_end(); i != e; ++i) {
  // inst uses the Value i
}
```

- You can loop over the instructions that use a particular value

```
Instruction *inst = ...;
for (auto i = inst->use_begin(), e = inst->use_end(); i != e; ++i)
  if (auto *user = dyn_cast<Instruction>(*i)) {
    // inst is used by Instruction user
  }
```

Dealing with Types

- LLVM IR is *strongly typed*
  - Every value has a type → getType()

Dealing with Types

- LLVM IR is *strongly typed*
  - Every value has a type → getType()
- A value must be explicitly cast to a new type

```
define i64 @trunc(i16 zeroext %a) {
  %1 = zext i16 %a to i64
  ret i64 %1
}
```

Dealing with Types

- LLVM IR is *strongly typed*
  - Every value has a type → getType()
- A value must be explicitly cast to a new type

```
define i64 @trunc(i16 zeroext %a) {
  %1 = zext i16 %a to i64
  ret i64 %1
}
```

Dealing with Types

- LLVM IR is *strongly typed*
  - Every value has a type → getType()
- A value must be explicitly cast to a new type

```
define i64 @trunc(i16 zeroext %a) {
  %1 = zext i16 %a to i64
  ret i64 %1
}
```

- Also types for pointers, arrays, structs, etc.
  - Strong typing means they take a bit more work

Dealing with Types: GEP

- We sometimes need to extract elements/fields from arrays/structs
  - Pointer arithmetic
  - Done using GetElementPointer (GEP)

Dealing with Types: GEP

- We sometimes need to extract elements/fields from arrays/structs
  - Pointer arithmetic
  - Done using GetElementPointer (GEP)

```c
struct rec {
  int x;
  int y;
};

struct rec *buf;

void foo() {
  buffer[5].y = 7;
}
```

```llvm
%struct.rec = type { i32, i32 }

@buf = global %struct.rec* null

define void @foo() {
  %1 = load %struct.rec** @buf
  %2 = getelementptr %struct.rec* %1, i64 5, i32 1
  store i32 7, i32* %2
  ret void
}
```

Dealing with Types: GEP

- We sometimes need to extract elements/fields from arrays/structs
  - Pointer arithmetic
  - Done using GetElementPointer (GEP)

```
struct rec {
  int x;
  int y;
};

struct rec *buf;

void foo() {
  buffer[5].y = 7;
}
```

```
%struct.rec = type { i32, i32 }

@buf = global %struct.rec* null

define void @foo() {
  %1 = load %struct.rec** @buf
  %2 = getelementptr %struct.rec* %1, i64 5, i32 1
  store i32 7, i32* %2
  ret void
}
```

Where Can You Get Info?

- The online documentation is extensive:
  - LLVM Programmer’s Manual
  - LLVM Language Reference Manual

Where Can You Get Info?

- The online documentation is extensive:
  - LLVM Programmer's Manual
  - LLVM Language Reference Manual
- The header files!
  - All in llvm-3.x.src/include/llvm/

    Function.h
    BasicBlock.h
    Instructions.h
    InstrTypes.h
    Support/CallSite.h
    Support/InstVisitor.h
    Type.h
    DerivedTypes.h

Making a New Analysis

- Analyses are organized into individual *passes*
  - ModulePass
  - FunctionPass
  - LoopPass
  - …

Derive from the appropriate base class to make a Pass

Making a New Analysis

- Analyses are organized into individual *passes*
  - ModulePass
  - FunctionPass
  - LoopPass
  - …

Derive from the appropriate base class to make a Pass

3 Steps

1) Declare your pass
2) Register your pass
3) Define your pass

Making a New Analysis

- Analyses are organized into individual *passes*
  - ModulePass
  - FunctionPass
  - LoopPass
  - …

Derive from the appropriate base class to make a Pass

3 Steps

1) Declare your pass
2) Register your pass
3) Define your pass

Let's count the number of direct calls to each function.

Making a ModulePass (1)

- Declare your ModulePass

```
struct CallPrinterPass : public llvm::ModulePass {

  static char ID;

  DenseMap<Function*, uint64_t> counts;

  CallPrinterPass()
    : ModulePass(ID)
      { }

  virtual bool runOnModule(Module &m) override;

  virtual void print(raw_ostream &out, const Module *m) const override;

  void handleInstruction(CallSite cs);
};
```

Making a ModulePass (1)

- Declare your ModulePass

```
struct CallPrinterPass : public llvm::ModulePass {

  static char ID;

  DenseMap<Function*, uint64_t> counts;

  CallPrinterPass()
    : ModulePass(ID)
      { }

  virtual bool runOnModule(Module &m) override;

  virtual void print(raw_ostream &out, const Module *m) const override;

  void handleInstruction(CallSite cs);
};
```

Making a ModulePass (1)

- Declare your ModulePass

```
struct CallPrinterPass : public llvm::ModulePass {

  static char ID;

  DenseMap<Function*, uint64_t> counts;

  CallPrinterPass()
    : ModulePass(ID)
      { }

  virtual bool runOnModule(Module &m) override;

  virtual void print(raw_ostream &out, const Module *m) const override;

  void handleInstruction(CallSite cs);
};
```

Making a ModulePass (2)

- Register your ModulePass
  - This allows it to by dynamically loaded as a plugin

```
char CallPrinterPass::ID = 0;

RegisterPass<CallPrinterPass> CallPrinterPassReg("callprinter",
                      "Print the static count of direct calls");
```

Making a ModulePass (3)

- Define your ModulePass
  - Need to override `runOnModule()` and `print()`

```
bool
CallPrinterPass::runOnModule(Module &m) {
  for (auto &f : m)
    for (auto &bb : f)
      for (auto &i : bb)
        handleInstruction(&i);
  return false; // False because we didn't change the Module
}
```

Making a ModulePass (3)

- analysis continued...

```
void
CallPrinterPass::handleInstruction(CallSite cs) {
  // Check whether the instruction is actually a call
  if (!cs.getInstruction()) { return; }

  // Check whether the called function is directly invoked
  auto called = cs.getCalledValue()->stripPointerCasts();
  auto fun = dyn_cast<Function>(called);
  if (!fun) { return; }

  // Update the count for the particular call
  auto count = counts.find(fun);
  if (counts.end() == count) {
    count = counts.insert(std::make_pair(fun, 0)).first;
  }
  ++count->second;
}
```

Making a ModulePass (3)

- analysis continued...

```
void
CallPrinterPass::handleInstruction(CallSite cs) {
  // Check whether the instruction is actually a call
  if (!cs.getInstruction()) { return; }

  // Check whether the called function is directly invoked
  auto called = cs.getCalledValue()->stripPointerCasts();
  auto fun = dyn_cast<Function>(called);
  if (!fun) { return; }

  // Update the count for the particular call
  auto count = counts.find(fun);
  if (counts.end() == count) {
    count = counts.insert(std::make_pair(fun, 0)).first;
  }
  ++count->second;
}
```

Making a ModulePass (3)

- analysis continued...

```
void
CallPrinterPass::handleInstruction(CallSite cs) {
  // Check whether the instruction is actually a call
  if (!cs.getInstruction()) { return; }

  // Check whether the called function is directly invoked
  auto called = cs.getCalledValue()->stripPointerCasts();
  auto fun = dyn_cast<Function>(called);
  if (!fun) { return; }

  // Update the count for the particular call
  auto count = counts.find(fun);
  if (counts.end() == count) {
    count = counts.insert(std::make_pair(fun, 0)).first;
  }
  ++count->second;
}
```

Making a ModulePass (3)

- Printing out the results

```
void
CallPrinterPass::print(raw_ostream &out, const Module *m) const {
  out << "Function Counts\n"
      << "===============\n";
  for (auto &kvPair : counts) {
    auto *function = kvPair.first;
    uint64_t count = kvPair.second;
    out << function->getName() << " : " << count << "\n";
  }
}
```

Putting it all Together

- LLVM organizes groups of passes and tools into *projects*

Putting it all Together

- LLVM organizes groups of passes and tools into *projects*

- Easiest way to start is by using their sample project
  - llvmsrc/projects/sample

Putting it all Together

- LLVM organizes groups of passes and tools into *projects*
- Easiest way to start is by using their sample project
  - llvmsrc/projects/sample
- For the most part, you can follow the directions online & in project description

Notes on Creating Projects

- Posted online, read on your own time:
  - Building
    - Copy the sample project to a new directory <proj>
    - Make another directory for building <projbuild>
    - <proj>/configure --disable-optimized --enable-debugging –with-clang=/path/to/clang
  - Customizing
    - You build your entire project in <proj>/lib/sample/
    - Delete the existing source and write your module there instead
    - Add these lines to the Makefile in the library directory:

```
LOADABLE_MODULE=1
CPPFLAGS += -std=c++11
```

Extra Tips

- I have a pointer to something. What is it?
  - The getName() method works on most things.
  - You can usually: outs() << X
- How do I see the C++ API calls for constructing a module?
  - **llc -march=cpp <bitcode>.bc -o <cppapi>.cpp**