A Brief Introduction to LLVM

Nick Sumner

(with a few modifications by Matt Dwyer)

What is LLVM?

- A compiler? (clang)

What is LLVM?

- A compiler? (clang)

- A set of formats, libraries, and tools.

What is LLVM?

- A compiler? (clang)

- 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? (clang)

- 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? (clang)

- 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? (clang)

- 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? (clang)

- 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 can you use it?

- Compiling programs to bitcode:

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

How can you use 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 can you use 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

- Writing your own tools:

  ./callcounter -static test.bc

How can you use 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

- Writing your own tools:

  ./callcounter -static test.bc

- Reporting properties of the program:

```
Function Counts
===============
b : 2
a : 1
printf : 3
```

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;
}
```

Code

clang -c -S -emit-llvm -O1 -g0

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

define void @foo(i32) {
  %2 = icmp eq i32 %0, 0
  br i1 %2, label %3, label %4

; <label>:3:                ; preds = %4, %1
  ret void

; <label>:4:                ; preds = %1, %4
  %5 = phi i32 [ %7, %4 ], [ 0, %1 ]
  %6 = tail call i32 @puts(i8* getelementptr
     ([6 x i8], [6 x i8]* @str, i64 0, i64 0))
  %7 = add nuw i32 %5, 1
  %8 = icmp eq i32 %7, %0
  br i1 %8, label %3, label %4
}

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

IR

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;
}
```

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

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

define void @foo(i32) {
  %2 = icmp eq i32 %0, 0
  br i1 %2, label %3, label %4

; <label>:3:                ; preds = %4, %1
  ret void

; <label>:4:                ; preds = %1, %4
  %5 = phi i32 [ %7, %4 ], [ 0, %1 ]
  %6 = tail call i32 @puts(i8* getelementptr
     ([6 x i8], [6 x i8]* @str, i64 0, i64 0))
  %7 = add nuw i32 %5, 1
  %8 = icmp eq i32 %7, %0
  br i1 %8, label %3, label %4
}

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

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;
}
```

Functions

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

define void @foo(i32) {
  %2 = icmp eq i32 %0, 0
  br i1 %2, label %3, label %4

; <label>:3:                ; preds = %4, %1
  ret void

; <label>:4:                ; preds = %1, %4
  %5 = phi i32 [ %7, %4 ], [ 0, %1 ]
  %6 = tail call i32 @puts(i8* getelementptr
     ([6 x i8], [6 x i8]* @str, i64 0, i64 0))
  %7 = add nuw i32 %5, 1
  %8 = icmp eq i32 %7, %0
  br i1 %8, label %3, label %4
}

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

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) {
  %2 = icmp eq i32 %0, 0
  br i1 %2, label %3, label %4

; <label>:3:                ; preds = %4, %1
  ret void

; <label>:4:                ; preds = %1, %4
  %5 = phi i32 [ %7, %4 ], [ 0, %1 ]
  %6 = tail call i32 @puts(i8* getelementptr
     ([6 x i8], [6 x i8]* @str, i64 0, i64 0))
  %7 = add nuw i32 %5, 1
  %8 = icmp eq i32 %7, %0
  br i1 %8, label %3, label %4
}

define i32 @main(i32, i8** nocapture readnone) {
  tail call void @foo(i32 %0)
  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 **arg
  foo(argc);
  return 0;
}
```

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

define void @foo(i32) {
  %2 = icmp eq i32 %0, 0
  br i1 %2, label %3, label %4

; <label>:3:                ; preds = %4, %1
  ret void

; <label>:4:                ; preds = %1, %4
  %5 = phi i32 [ %7, %4 ], [ 0, %1 ]
  call i32 @puts(i8* getelementptr
    [6 x i8]* @str, i64 0, i64 0))
  %7 = add nuw i32 %5, 1
  %8 = icmp eq i32 %7, %0
  br i1 %8, label %3, label %4
}

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

labels & predecessors

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) {
  %2 = icmp eq i32 %0, 0
  br i1 %2, label %3, label %4

; <label>:3:                ; preds = %4, %1
  ret void

                            ; preds = %1, %4
  %5 = phi i32 [ %7, %4 ], [ 0, %1 ]
  %6 = tail call i32 @puts(i8* getelementptr
     ([6 x i8], [6 x i8]* @str, i64 0, i64 0))
  %7 = add nuw i32 %5, 1
  %8 = icmp eq i32 %7, %0
  br i1 %8, label %3, label %4
}

define i32 @main(i32, i8** nocapture readnone) {
  tail call void @foo(i32 %0)
  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) {
  %2 = icmp eq i32 %0, 0
  br i1 %2, label %3, label %4

; <label>:3:                ; preds = %4, %1
  ret void

; <label>:4:                ; preds = %1, %4
  %5 = phi i32 [ %7, %4 ], [ 0, %1 ]
  %6 = tail call i32 @puts(i8* getelementptr
     ([6 x i8], [6 x i8]* @str, i64 0, i64 0))
  %7 = add nuw i32 %5, 1
  %8 = icmp eq i32 %7, %0
  br i1 %8, label %3, label %4
}

define i32 @main(i32, i8** nocapture readnone) {
  tail call void @foo(i32 %0)
  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?

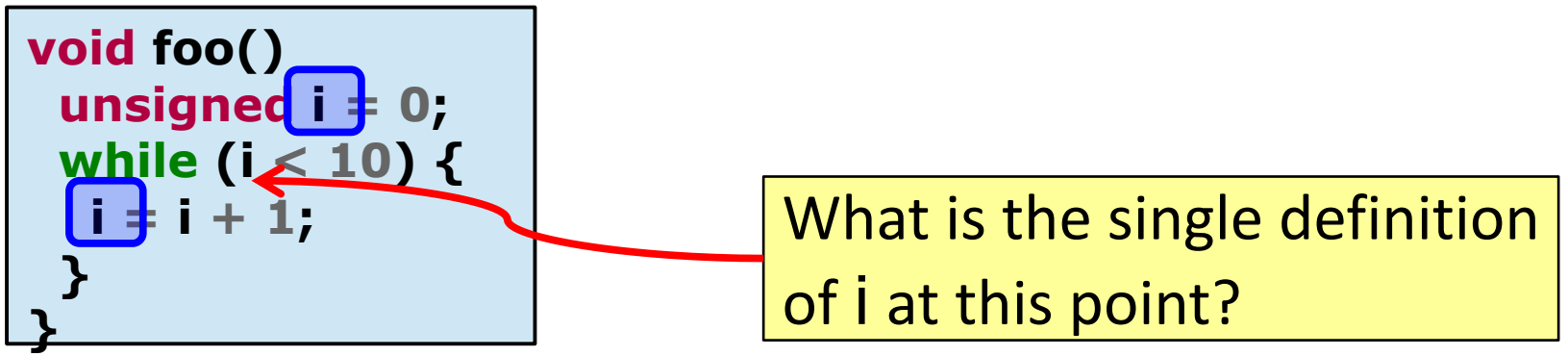
Dealing with SSA

- 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?



Dealing with SSA

- 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                ; preds = %1, %0
  %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                ; preds = %1
  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                ; preds = %1, %0
  %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                ; preds = %1
  ret void
}
```

Dependencies in General

- You can loop over the values an instruction uses

```
for (Use& u : inst->operands()) {
  // inst uses the Value* u
}
```

Dependencies in General

- You can loop over the values an instruction uses

```
for (Use& u : inst->operands()) {
  // inst uses the Value* u
}
```

for %a = %b + %c:

[%b, %c]

Dependencies in General

- You can loop over the values an instruction uses

```
for (Use& u : inst->operands()) {
  // inst uses the Value* u
}
```

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

```
Instruction* inst = ...;
for (User* user : inst->users())
  if (auto* i = dyn_cast<Instruction>(user)) {
    // inst is used by Instruction i
  }
```

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

The `tipc` TIP compiler

- Compiles a TIP program into LLVM bitcode
- Consists of several phases
  - Parsing TIP (uses ANTLR4)
  - Builds an AST
  - Walks the AST to generate bitcode
  - Applies building LLVM optimizations
  - Emits bitcode to a .bc file
- To build an executable we link with libraries

The `tipc` TIP compiler

```
$ ./tipc --help
OVERVIEW: tipc - a TIP to llvm compiler

USAGE: tipc [options] <tip source file>

OPTIONS:

Generic Options:

  -help      - Display available options (-help-hidden for more)
  -help-list - Display list of available options (-help-list-hidden
for more)
  -version   - Display the version of this program

tipc Options:
Options for controlling the TIP compilation process.

  -d         - disable bitcode optimization
  -l         - pretty print with line numbers
  -p         - pretty print
```