COS320: Compiling Techniques

Zak Kincaid

February 18, 2020
• Reminder: HW1 due today
• HW2 on course webpage. Due March 3
  • You will implement an LLVMlite-to-X86lite compiler
  • You may work individually or in pairs
LLVM
Low-Level Virtual Machine (LLVM)

• Open-source compiler infrastructure
  • Created by Chris Lattner (advised by Vikram Adve) at UIUC in 2003
  • Apple XCode 3.1
  • Several OpenCL implementations (NVIDIA, Intel, Apple, ...)
  • PlayStation™4 compiler
  • Used widely in academia

• Many components. The ones we’re interested in:
  • LLVM IR
  • llc: code generator (for various targets)
  • opt: LLVM IR → LLVM IR optimization
Compiler phases (simplified)

Source text
  ↓ Lexing
Token stream
  ↓ Parsing
Abstract syntax tree
  ↓ Translation
Intermediate representation
  ↓ Code generation
Assembly

LLVM

Optimization
Many front-ends & back-ends

- C
- C++
- Rust
- Go
- Swift
- LLVM
- x86
- ARM
- PowerPC
- C++
- MIPS
LLVMlite IR

- LLVMlite is a small subset of the LLVM IR
- Broadly similar to the let-based IR from last week
  - Each procedure $P$ is represented as a control flow graph: a directed, rooted graph where
    - The nodes are basic blocks of $P$
    - There is an edge $BB_i \rightarrow BB_j$ iff $BB_j$ may execute immediately after $BB_i$
    - There is a distinguished entry block where the execution of the procedure begins
  - Local variables must satisfy the static single assignment property
LLVMlite IR

- LLVMlite is a small subset of the LLVM IR
- Broadly similar to the let-based IR from last week
  - Each procedure $P$ is represented as a control flow graph: a directed, rooted graph where
    - The nodes are basic blocks of $P$
    - There is an edge $BB_i \rightarrow BB_j$ iff $BB_j$ may execute immediately after $BB_i$
    - There is a distinguished entry block where the execution of the procedure begins
  - Local variables must satisfy the static single assignment property
- Some differences:
  - Memory allocation
  - Functions
  - Types
```c
#define i64 @factorial(i64 %arg) {
    %tmp = alloca i64
    %tmp1 = alloca i64
    %tmp2 = alloca i64
    store i64 %arg, i64* %tmp
    store i64 1, i64* %tmp2
    store i64 1, i64* %tmp1
bb3:
    %tmp4 = load i64, i64* %tmp1
    %tmp5 = load i64, i64* %tmp
    %tmp6 = icmp sle i64 %tmp4, %tmp5
bb7:
    %tmp8 = load i64, i64* %tmp1
    %tmp9 = load i64, i64* %tmp2
    %tmp10 = mul i64 %tmp9, %tmp8
    store i64 %tmp10, i64* %tmp2
bb11:
    %tmp12 = load i64, i64* %tmp1
    %tmp13 = add i64 %tmp12, 1
    store i64 %tmp13, i64* %tmp1
bb14:
    %tmp15 = load i64, i64* %tmp2
    ret i64 %tmp15
}

@.str = global [18 x i8] c"Factorial_is_%ld\0A\00"
#define i64 @main(i32 %arg, i8** %arg1) #0 {
    %tmp1 = bitcast [18 x i8]* @.str to i8*
    %tmp2 = call i64 @factorial(i64 6)
    %tmp3 = call i64 (i8*, ...) @printf(i8* %tmp1, i64 %tmp2)
    ret i64 0
}

declare i64 @printf(i8*, ...)
```
LLVMlite memory

- Local variables / temporaries / “abstract registers” (%uid)
  - E.g., %t4 = mul i64 %t1, %t3

- Global declarations (e.g., for functions, string constants): @gid
  - E.g., @.str = constant [18 x i8] c"Factorial is %ld\0A\00"
  - E.g., %r = call @factorial(i64 6)

- Stack allocated storage
  - %count = alloca i64

- Heap-allocated storage, created by external calls (malloc)
• Program has four components:
  • Type declarations
    • E.g., %node = { i64, %node* } 
  • Global declarations
    • E.g., @.str = global [18 x i8] c"Factorial is %ld\n\0"
  • Function declarations
    • E.g., define i64 @factorial(i64 %n) { ... } 
  • External declarations
    • E.g., declare i32 @printf(i8*, ...)
Functions

• Function declaration
  • define i64 @factorial(i64 %n) { <cfg> }
  • type fdecl = { f_ty : fty; f_param : uid list; f_cfg : cfg }

• Function call
  • Direct call: %r = call @factorial(i64 6)
  • Indirect call: %r = call %5(i64 1, i64 10)
define i64 @factorial(i64 %n) {
    %count = alloca i64
    %result = alloca i64
    store i64 %n, i64* %count
    store i64 1, i64* %result
    br label %loop

    loop:
    %t1 = load i64, i64* %count
    %t2 = icmp sgt i64 %t1, 1
    br i1 %t2, label %body, label %exit

    body:
    %t3 = load i64, i64* %result
    %t4 = mul i64 %t1, %t3
    store i64 %t4, i64* %result
    %t5 = sub i64 %t1, 1
    store i64 %t5, i64* %count
    br label %loop

    exit:
    %t6 = load i64, i64* %result
    ret i64 %t6
}

%count = alloca i64
%result = alloca i64
store i64 %n, i64* %count
store i64 1, i64* %result
br label %loop

%t1 = load i64, i64* %count
%t2 = icmp sgt i64 %t1, 1
br i1 %t2, label %body, label %exit

%t3 = load i64, i64* %result
%t4 = mul i64 %t1, %t3
store i64 %t4, i64* %result
%t5 = sub i64 %t1, 1
store i64 %t5, i64* %count
br label %loop

%t6 = load i64, i64* %result
ret i64 %t6
Static Single Assignment (SSA)

- Each variable appears on the left-hand-side of at most one assignment in a CFG
  
  ```
  x = x + y;
y = 2 * x;
x = x + 1;
z = x - 1;
y = x & z;
return y;
  ```

  ```
  x_1 = x_0 + y_0;
y_1 = 2 * x_1;
x_2 = x_1 + 1;
z_1 = x_2 - 1;
y_2 = x_2 & z_1;
return y_2;
  ```

- Simplifies analysis and optimization
- Makes connections between variable definitions and uses explicit
- More freedom in memory allocation
- No need for `x_0` and `x_2` to be stored in the same register or stack slot
- Simple application: dead code elimination
  - If `%uid` is never used, can elide the assignment to `%uid` (e.g., `y_1` above)
### Static Single Assignment (SSA)

- Each variable appears on the left-hand-side of at most one assignment in a CFG
  
  ```plaintext
  x = x + y;
  y = 2 * x;
  x = x + 1;
  z = x - 1;
  y = x & z;
  return y;
  
  x_1 = x_0 + y_0;
  y_1 = 2 * x_1;
  x_2 = x_1 + 1;
  z_1 = x_2 - 1;
  y_2 = x_2 & z_1;
  return y_2;
  ```

- Simplifies analysis and optimization
  - Make connections between variable definitions and uses explicit
  - More freedom in memory allocation
    - No need for $x_0$ and $x_2$ to be stored in the same register or stack slot
  - Simple application: dead code elimination
    - If variable is never used, can elide the assignment to that variable (e.g., $y_1$ above)
Stack storage

• Unlike our let-based IR, LLVM does not have mutable symbolic variables
  • store n = tmp8

• alloca instruction allocates stack space and returns a pointer to it
  • %count = alloca i64 allocates a 8 bytes of stack space, %count points to the space

• load and store read/write memory
  • %t6 = load i64, i64* %result
    read 64-bit int from the memory addressed by the 64-bit int pointer %result, store it in %t6
  • store i64 %n, i64* %count
    store 64-bit int %n in the memory addressed by the 64-bit int pointer %count

• No stack de-allocation. Implementation of return must de-allocate.
Types

• LLVM IR is statically typed
• Types:
  • Integer types: i1, i64
  • Pointers: i8*, i64*
  • Function pointers: i64(i64, i64*)
  • Tuples: i64, i64, i64 (integer triples)
  • Arrays: [18 x i8] (array of 18 characters)
  • Named types
    • Allows recursive types (e.g., lists, trees, graphs, ...)
    • %node = { i64, %node* }
LLVM’s type system is *inexpressive*

- No generics
- No subtyping

LLVM IR provides a *bitcast* instruction to circumvent the type system

```ll
%pair = type { i64, i64 } ; two-field record
%triple = type { i64, i64, i64 } ; three-field record

@g = global %triple { i64 0, i64 1, i64 2 } ; allocate global triple
define @foo() {
    %c = bitcast %triple* @g to %pair* ; cast
}
```

- *bitcast* does not change any bits
- Potentially unsafe!
  - Can cause segfaults or memory corruption

More casting instructions in real LLVM IR, LLVMlite has only *bitcast*
define i64 @factorial(i64) #0 {
  %2 = alloca i64, align 8
  %3 = alloca i64, align 8
  %4 = alloca i64, align 8
  store i64 %0, i64* %2, align 8
  store i64 1, i64* %4, align 8
  store i64 1, i64* %3, align 8
  br label %5
  ; <label>:5:
  %6 = load i64, i64* %3, align 8
  %7 = load i64, i64* %2, align 8
  %8 = icmp slt i64 %6, %7
  br i1 %8, label %9, label %16
  ; <label>:9:
  %10 = load i64, i64* %3, align 8
  %11 = load i64, i64* %4, align 8
  %12 = mul nsw i64 %11, %10
  store i64 %12, i64* %4, align 8
  br label %13
  ; <label>:13:
  %14 = load i64, i64* %3, align 8
  %15 = add nsw i64 %14, 1
  store i64 %15, i64* %3, align 8
  br label %5
  ; <label>:16:
  %17 = load i64, i64* %4, align 8
  ret i64 %17
}
(Some) comparisons to LLVMlite:

- More (optional) type and alignment annotations
- Numeric identifiers
- Keeps track of block predecessors

```c
if (x < 0) {
  y := y - x;
} else {
  y := y + x;
}
return y

if (x < 0) {
  y1 := y0 - x0;
} else {
  y2 := y0 + x0;
}
```

\( y3 := \phi(y1, y2) \)
(Some) comparisons to LLVMlite:

- More (optional) type and alignment annotations
- Numeric identifiers
- Keeps track of block predecessors
- $\phi$ instructions (more on these later...)

```diff
if (x < 0) {
    y := y - x;
} else {
    y := y + x;
}
return y
```

```diff
if (x_0 < 0) {
    y_1 := y_0 - x_0;
} else {
    y_2 := y_0 + x_0;
}
y_3 := \phi(y_1, y_2)
return y_3
```
Using LLVM

- `clang file.c -emit-llvm -S`: produce LLVM IR in `file.ll`
- `opt [options] -S file.ll -o file-opt.ll`: optimize
  - **Options:** -O2, -O3, -mem2reg, ...
  - **Recommended:** -instnamer
- `llc file-opt.ll`: produce x86 assembly in `file-opt.s`
- `gcc file-opt.s -o file`: produce file executable