Compiler construction

Lecture 4: Code generation for LLVM

Thomas Sewell

Spring 2020

Chalmers University of Technology — Gothenburg University

Overview

Last week: Anatomy of a program in LLVM IR

This week: How to generate an LLVM Module

- Structure
- Variables
- Types and elements

Unfinished Business: SSA

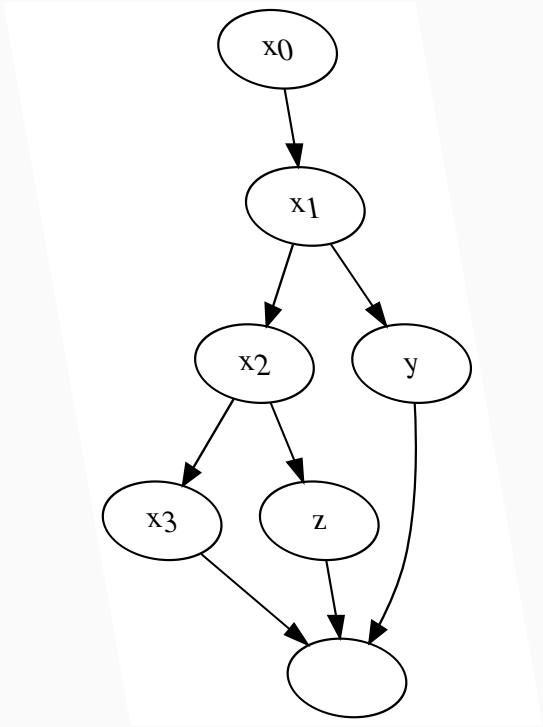
Promised last time: Why SSA?

```
int f (int x) {
  int y, z;
  x = x * 2;
  y = x + 3;
  x = x - 12;
  z = x / 4;
  x = x * 6;
  return x + y + z;
}
```

Unfinished Business: SSA

Promised last time: Why SSA?

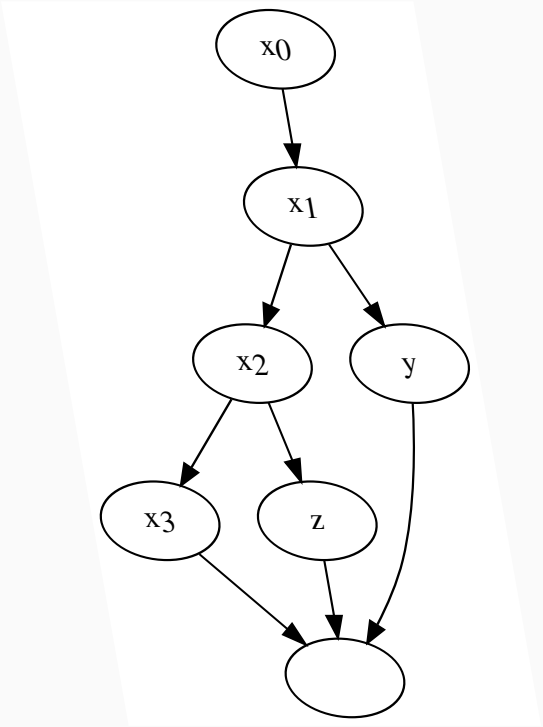
```
int f (int x) {
  int y, z;
  x = x * 2;
  y = x + 3;
  x = x - 12;
  z = x / 4;
  x = x * 6;
  return x + y + z;
}
```



Unfinished Business: SSA

Promised last time: Why SSA?

```
int f (int x) {
  int y, z;
  x = x * 2;
  y = x + 3;
  x = x - 12;
  z = x / 4;
  x = x * 6;
  return x + y + z;
}
```



It's the graph structure that matters.
SSA $\approx$ each value names a node in the graph.

LLVM Program Structure

LLVM modules

A LLVM compilation unit (a module) consists of a sequence of:

- type definitions
- global variable definitions
- function definitions
- (external) function declarations
- (external) global variable declarations

External global variables are not necessary for JAVALETTE; the only use of global variables is for naming string literals (as arguments to `@printString`).

Basic blocks in LLVM

CHALMERS

Recall

A basic block starts with a label and ends with a terminating instruction (`ret` or `br`).

Thus one cannot ‘fall through’ the end of a block into the next; an explicit branch to (the label of) the next instruction is necessary.

Basic blocks in LLVM

CHALMERS

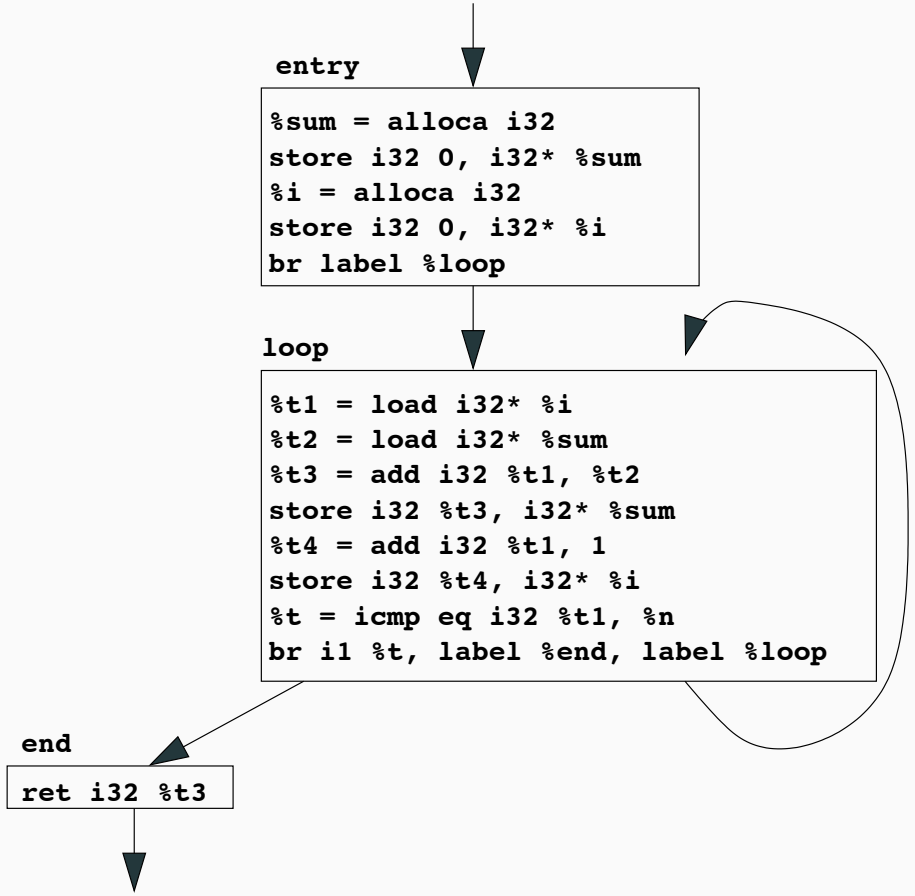
Recall

A basic block starts with a label and ends with a terminating instruction (`ret` or `br`).

Thus one cannot ‘fall through’ the end of a block into the next; an explicit branch to (the label of) the next instruction is necessary.

Consequence

The basic blocks of a LLVM function definition can be reordered arbitrarily; a function body is a graph of basic blocks (the control flow graph).



Generating LLVM

Compilation to LLVM

CHALMERS

General observations

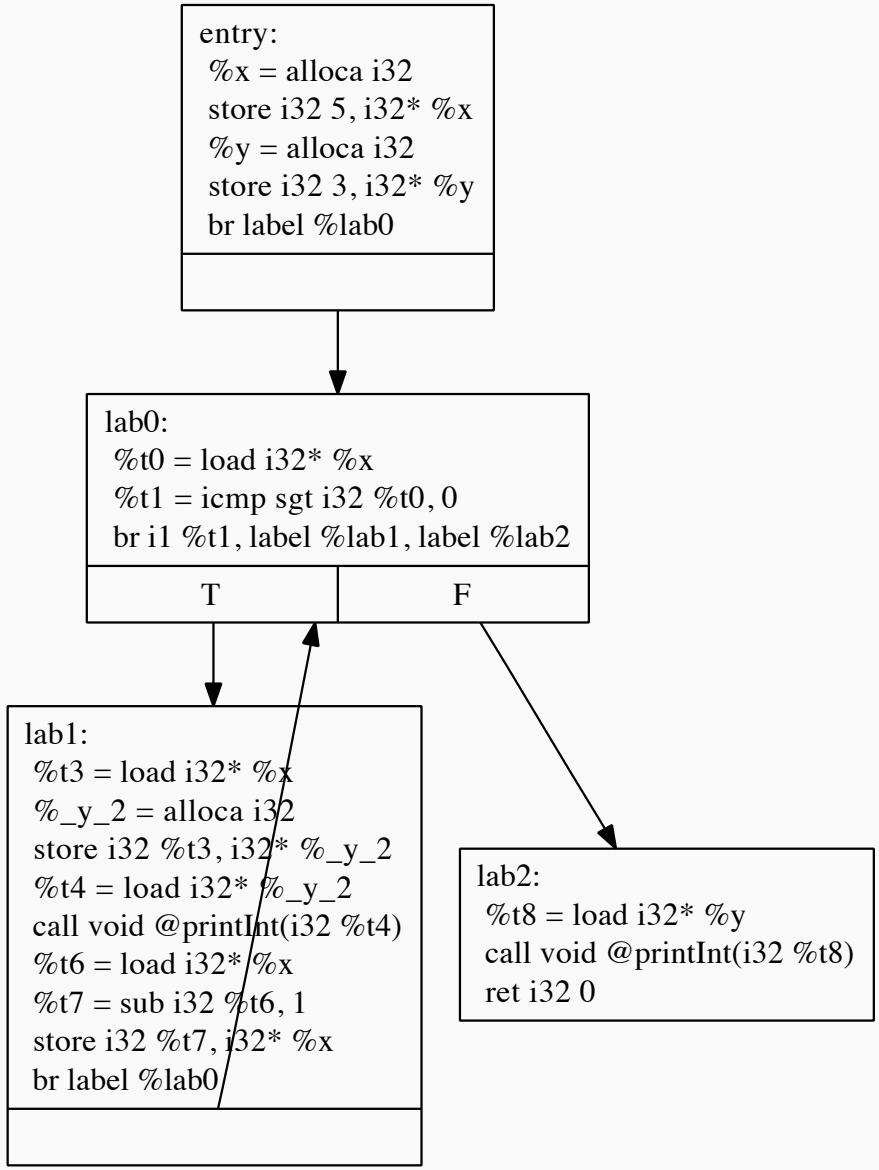
- Similarities to compilation schemes for JVM (e.g. discussed in the PLT course)
- Local variables and parameters should be treated as memory locations (`alloca`/`load`/`store` instructions)
- These will be removed by `opt` (and new memory references maybe introduced during register allocation)

Code generation for variables, 1

There are no nested scopes in LLVM. Thus JAVALETTE variables may need to be renamed.

Example

```
int main () {
  int x = 5;
  int y = 3;
  while (x > 0) {
    int y = x;
    printInt(y);
    x--;
  }
  printInt(y);
  return 0;
}
```



Optimizing code from previous slide

CHALMERS

```
> opt -std-compile-opts a.ll | llvm-dis

; ModuleID = '<stdin>'
declare void @printInt(i32)

define i32 @main() {
entry:
  tail call void @printInt(i32 5)
  tail call void @printInt(i32 4)
  tail call void @printInt(i32 3)
  tail call void @printInt(i32 2)
  tail call void @printInt(i32 1)
  tail call void @printInt(i32 3)
  ret i32 0
}
```

Code generation for variables, 2

- When a variable declaration is seen:
  - generate a (sufficiently) new name
  - generate `alloca` instruction
  - save (JavaLette name, LLVM name) pair in lookup table in the code generator
- Keep track of scope in lookup table `tab`
  - remember, we need to be able to revert at the end of a block
- JavaLette assign to `x` $\Rightarrow$ LLVM `store` of `lookup (tab, x)`
- JavaLette access to `x` $\Rightarrow$ LLVM `load` of `lookup (tab, x)`
- Parameters $\approx$ variables

Code generation for variables, alternative

Alternative approach: Some compilers include an initial α-renaming phase to rename all program variables such that variable names become unique. (This may simplify subsequent compiler phases.)

Before

```
int main () {
  int x = 5;
  int y = 3;
  while (x > 0) {
    int y = x;
    printInt(y);
    x--;
  }
  printInt(y);
  return 0;
}
```

After

```
int main () {
  int v1 = 5;
  int v2 = 3;
  while (v1 > 0) {
    int v3 = v1;
    printInt(v3);
    v1--;
  }
  printInt(v2);
  return 0;
}
```

Structural Strategy

General strategy for converting JavaLette to LLVM

- Sweep across JavaLette function body
- Work block by block, subdividing JavaLette blocks at `if` & `while`
- For each JavaLette sub-block produce an LLVM block
- Pay attention to variable mapping at each block

Treatment of labels

Labels are not instructions in LLVM

But it may be convenient for you to treat them as if they were!

Basic blocks without instructions are illegal

Depending on your compilation schemes, you may find yourself in the situation that a label has just been emitted and the function ends without further instructions.

Treatment of labels

CHALMERS

Labels are not instructions in LLVM

But it may be convenient for you to treat them as if they were!

Basic blocks without instructions are illegal

Depending on your compilation schemes, you may find yourself in the situation that a label has just been emitted and the function ends without further instructions.

The situation can then be saved by emitting the special terminator instruction `unreachable`.

Handling Pointers and LLVM Types

Types of local and global variables

CHALMERS

Local variables

The instruction

```
%x = alloca i32
```

introduces a new variable `%x` of type `i32*`

`%x` is a pointer to a newly allocated memory location on the stack.

Types of local and global variables

CHALMERS

Local variables

The instruction

```
%x = alloca i32
```

introduces a new variable `%x` of type `i32*`

`%x` is a pointer to a newly allocated memory location on the stack.

Global variables

The instruction

```
@hw = global [13 x i8] c"hello world\0A\00"
```

introduces a global name `@hw` of type `[13 x i8]*`

`@hw` is a pointer to a byte array.

The `getelementptr` instruction

CHALMERS

From reference manual

The `getelementptr` instruction is used to get the address of a subelement of an aggregate data structure. It performs address calculation only and does not access memory.

```
getelementptr %T, %T* %x, i32 0, i32 1, i32 1, i32 7

          ≈    & (x[0][1][1][7])
```

- provides an LLVM-level (low-level) equivalent to & in C
- upside: low-level, general purpose, type-checked
- downside: needs ≈ 5 slides to explain

Arguments of `getelementptr`

CHALMERS

Instruction arguments

Type to index `%T`, a variable `%x` that has pointer type `%T*`, and then indexes into the pointer. Each argument steps inside one more compound type.

Example type

```
%T = type
  {i32,
    {[4 x i32],
     [8 x i32]
    }
  }
```

Example use

```
define i32 @f (%T* %x) {
  %p = getelementptr %T, %T* %x,
       i32 0, i32 1, i32 1, i32 7
  %res = load i32, i32* %p
  ret i32 %res
}
```

We can index inside the arrays, and the compound structure $\{\ldots, \ldots\}$. The first zero indexes into the pointer argument.

Another `getelementptr` example

```
@mat = global [3 x [4 x i32]]
              [[4 x i32] [i32 1, i32 2, i32 3, i32 4],
               [4 x i32] [i32 5, i32 6, i32 7, i32 8],
               [4 x i32] [i32 9, i32 10, i32 11, i32 12]]

declare void @printInt(i32)

define i32 @main () {
  %t1 = getelementptr [3 x [4 x i32]], [3 x [4 x i32]]* @mat,
                      i32 0, i32 1, i32 2
  %t2 = load i32, i32* %t1
  call void @printInt(i32 %t2)
  ret i32 0
}
```

Executing this program prints 7. Note type of `@mat`.

Yet another `getelementptr` example

```
%T1 = type {i32, {[4 x i32]*, [8 x i32]*}}

define i32 @g (%T1* %x) {
  %p   = getelementptr %T1, %T1* %x, i32 0, i32 1, i32 1
  %p1  = load [8 x i32]*, [8 x i32]** %p
  %p2  = getelementptr [8 x i32], [8 x i32]* %p1, i32 0, i32 7
  %res = load i32, i32* %p2
  ret i32 %res
}
```

`@g` returns the last element of the 8-element array in `%x`.

We can not do this with just one `getelementptr` instruction; we need to access memory to get the pointer to the array.

Why the first `0`?

Most `getelementptr` instructions begin with `0`.

Note/recall that, in C

- `int[] xs_ptr` $\approx$ `int *xs_ptr`
- `xs_ptr[0]` $\equiv$ `*xs_ptr`

Most pointers are pointers are pointers to single elements, but some are to the first element of an array.

Why the first 0?

CHALMERS

```
struct Pair {
  int x, y;
};
int f(struct Pair *p) {
  return p[0].y + p[1].x;
}
```

Why the first 0?

CHALMERS

```
struct Pair {
  int x, y;
};
int f(struct Pair *p) {
  return p[0].y + p[1].x;
}

%Pair = type { i32, i32 }
define i32 @h(%Pair* %p) {
  %t1 = getelementptr %Pair, %Pair* %p, i32 0, i32 1
  %t2 = load i32, i32* %t1
  %t3 = getelementptr %Pair, %Pair* %p, i32 1, i32 0
  %t4 = load i32, i32* %t3
  %t5 = add i32 %t2, %t4
  ret i32 %t5
}
```

Computing the size of a type

CHALMERS

Size of a variable

With the size of a type `%T`, we mean the size (in bytes) of a variable of type `%T`. For a given LLVM type `%T`, this size can vary between target architectures (e.g. pointer types differ in size). So, how does one write portable code?

LLVM does not have a correspondence to C's `sizeof` macro.

Computing the size of a type

CHALMERS

Size of a variable
With the size of a type `%T`, we mean the size (in bytes) of a variable of type `%T`. For a given LLVM type `%T`, this size can vary between target architectures (e.g. pointer types differ in size). So, how does one write portable code?

LLVM does not have a correspondence to C's `sizeof` macro.

The trick
We use the `getelementptr` instruction:

```
%p = getelementptr %T, %T* null, i32 1
%s = ptrtoint %T* %p to i32
```

Now, `%s` holds the size of `%T`. Why?

Treatment of string literals

CHALMERS

- String literals occur in JavaLette only as argument to `@printString`
- When you encounter such a string you must introduce a definition that gives the string literal a global name
- Such a definition must not appear in the middle of the current function (recall the 'hello world' program)
- The type of a global variable is `[n x i8]*`, where `n` is the length of the string (after padding at the end)
- `@printString` is called with a global variable as argument

Quiz
What is the type of the parameter to `@printString`?

```
declare void @printString( ? )
```

String literals, 2

Answer

- We cannot let the parameter type be `[n x i8]*`, since `n` varies
- Let instead the parameter type be `i8*`, a pointer to the first byte
- How can we then call `@printString` in a type-correct way?

String literals, 2

CHALMERS

Answer

- We cannot let the parameter type be `[n x i8]*`, since `n` varies
- Let instead the parameter type be `i8*`, a pointer to the first byte
- How can we then call `@printString` in a type-correct way?

We use `getelementptr` to get a pointer to the first byte of the string (i.e. to the same address, but the type will change).

```
@hw = internal constant [13 x i8] c"hello world\0A\00"
declare void @printString(i8*)

define i32 @main () {
  %t1 = getelementptr [13 x i8], [13 x i8]* @hw, i32 0, i32 0
  call void @printString(i8* %t1)
  ret i32 0
}
```

State during code generation

CHALMERS

We need to keep some state information during code generation. This includes at least:

- next number for generating register names (and labels)
- definitions of global names for string literals
- lookup table to find LLVM name for JavaLette variable name
- lookup table to find type of function

Further properties of functions

CHALMERS

More syntactic features available in defining & calling functions:

In function definitions

- Linkage type, for example: `private`, `internal`
- Attributes, for example: `readnone`, `readonly`, `nounwind`
- Calling convention, for example: `ccc`, `fastcc`

In function calls

- Tail call indication
- Attributes
- Calling convention

Final example

CHALMERS

JavaLette code

```
boolean even(int n) {
  if (n == 0)
     return true;
  else
     return odd (n - 1);
}
boolean odd(int n) {
  if (n == 0)
     return false;
  else
     return even (n - 1);
}
```

JavaLette code

```
int main () {
  if (even (20))
     printString("Even!");
  else
     printString("Odd!");
  return 0;
}
```

To be done in class

- Let's hand-compile to naive LLVM code
- Send it through `opt` to get better code