Lecture 8

CIS 341: COMPILERS

Announcements

- Homework 3: Compiling LLVMlite
- Goal:
 - Familiarize yourself with (a subset of) the LLVM IR
 - Implement a translation down to (inefficient) X86lite
- **Due:** Thursday, Feb. 23rd
- Update: (small) clarification in the project description
 - fixes some minor discrepancies in terminology: "ctxt" vs "layout"

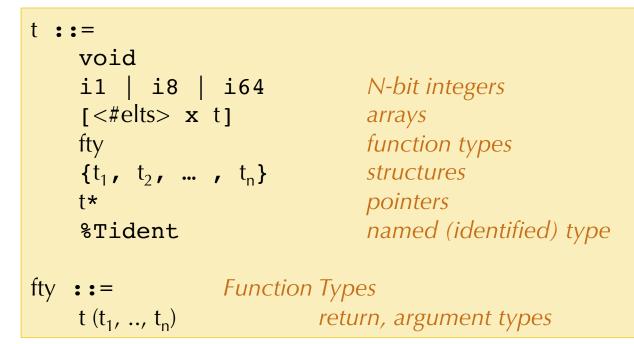
START EARLY!!

DATATYPES IN THE LLVM IR

Zdancewic CIS 341: Compilers

Structured Data in LLVM

• LLVM's IR is uses types to describe the structure of data.



- <#elts> is an integer constant >= 0
- Structure types can be named at the top level:

 $T1 = type \{t_1, t_2, ..., t_n\}$

- Such structure types can be recursive

Example LL Types

- An array of 341 integers: [341 x i64]
- A two-dimensional array of integers: [3 x [4 x i64]]
- Structure for representing arrays with their length:

{ i64 , [0 x i64] }

- There is no array-bounds check; the static type information is only used for calculating pointer offsets.
- C-style linked lists (declared at the top level):

%Node = type { i64, %Node*}

 Structs from the C program shown earlier: %Rect = { %Point, %Point, %Point, %Point } %Point = { i64, i64 }

getelementptr

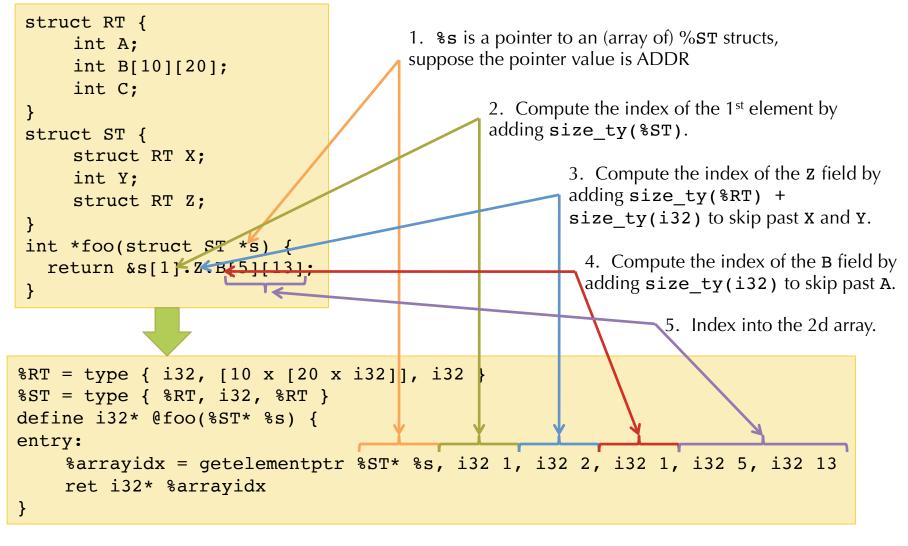
- LLVM provides the getelementptr instruction to compute pointer values
 - Given a pointer and a "path" through the structured data pointed to by that pointer, getelementptr computes an address
 - This is the abstract analog of the X86 LEA (load effective address). It does not access memory.
 - It is a "type indexed" operation, since the sizescomputations involved depend on the type

```
insn ::= ...
| getelementptr t* %val, t1 idx1, t2 idx2 ,...
```

• Example: access the x component of the first point of a rectangle:

```
%tmp1 = getelementptr %Rect* %square, i32 0, i32 0
%tmp2 = getelementptr %Point* %tmp1, i32 0, i32 0
```

GEP Example*



Final answer: ADDR + size_ty(%ST) + size_ty(%RT) + size_ty(i32) + size_ty(i32) + 5*20*size_ty(i32) + 13*size_ty(i32)

Zdancewic CIS 341: Compile *adapted from the LLVM documentaion: see http://llvm.org/docs/LangRef.html#getelementptr-instruction

getelementptr

- GEP *never* dereferences the address it's calculating:
 - GEP only produces pointers by doing arithmetic
 - It doesn't actually traverse the links of a datastructure
- To index into a deeply nested structure, need to "follow the pointer" by loadingfrom the computed pointer
 - See list.ll from HW3

Compiling Datastructures via LLVM

- 1. Translate high level language types into an LLVM representation type.
 - For some languages (e.g. C) this process is straight forward
 - The translation simply uses platform-specific alignment and padding
 - For other languages, (e.g. OO languages) there might be a fairly complex elaboration.
 - e.g. for Ocaml, arrays types might be translated to pointers to length-indexed structs.

 $[int array] = \{ i32, [0 x i32]\}*$

- 2. Translate accesses of the data into getelementptr operations:
 - e.g. for Ocaml array size access:
 [length a]] =
 %1 = getelementptr {i32, [0xi32]}* %a, i32 0, i32 0

Bitcast

- What if the LLVM IR's type system isn't expressive enough?
 - e.g. if the source language has subtyping, perhaps due to inheritance
 - e.g. if the source language has polymorphic/generic types
- LLVM IR provides a **bitcast** instruction
 - This is a form of (potentially) unsafe cast. Misuse can cause serious bugs (segmentation faults, or silent memory corruption)

```
%rect2 = type { i64, i64 } ; two-field record
%rect3 = type { i64, i64, i64 } ; three-field record
define @foo() {
  %1 = alloca %rect3 ; allocate a three-field record
  %2 = bitcast %rect3* %1 to %rect2* ; safe cast
  %3 = getelementptr %rect2* %2, i32 0, i32 1 ; allowed
  ...
}
```

see HW3 **LLVMLITE SPECIFICATION**

Zdancewic CIS 341: Compilers

COMPILING LLVMLITE TO X86

Zdancewic CIS 341: Compilers

Compiling LLVMlite Types to X86

- [[i1]], [[i64]], [[t*]] = quad word (8 bytes, 8-byte aligned)
- raw i8 values are not allowed (they must be manipulated via i8*)
- array and struct types are laid out sequentially in memory
- getelementptr computations must be relative to the LLVMlite size definitions
 - i.e. [[i1]] = quad

Compiling LLVM locals

- How do we manage storage for each %uid defined by an LLVM instruction?
- Option 1:
 - Map each %uid to a x86 register
 - Efficient!
 - Difficult to do effectively: many %uid values, only 16 registers
- Option 2:
 - Map each %uid to a stack-allocated space
 - Less efficient!
 - Simple to implement
- For HW3 we will follow Option 2

Other LLVMlite Features

- Globals
 - NOTE: must use %rip relative addressing:

address global at label @G: translates to X86 operand as: leag _G(%rip), DEST

- On OS X to support
- Calls
 - Follow x64 AMD ABI calling conventions
 - Should interoperate with C programs
- getelementptr
 - trickiest part

LLVMlite notes

• Recall LLVM requires that constants appearing in getelementptr be declared with type i32:

```
%struct = type { i64, [5 x i64], i64}
@gbl = global %struct {i64 1,
    [5 x i64] [i64 2, i64 3, i64 4, i64 5, i64 6], i64 7}
define void @foo() {
  %1 = getelementptr %struct* @gbl, i32 0, i32 0
...
}
```

- LLVMlite ignores the i32 annotation and treats these as i64 values
 - we keep the i32 annotation in the syntax to retain compatibility with the clang compiler

see HW3 and README

II.ml, using main.native, clang, etc.

TOUR OF HW 3