Announcements

• **CS Nights**: Tuesdays 8pm-10pm, MD119
  • Combined OH for CS153, CS61, CS121
  • Pizza and community!

• **Homework 2**: X86lite
  • Due today

• **Homework 3**: LLVMlite
  • Will be released today
  • Due in three weeks
  • Start early!!!
    • Challenging assignment; HW4 will be released in 2 weeks
Today

• Arrays
• Tagged datatypes (and switches)
• Datatypes in LLVM
• Brief tour of HW3
Arrays

- Space is allocated on the stack for \texttt{buf}
  - Note: without ability to allocate stack space dynamically (C's \texttt{alloca} function) need to know size of \texttt{buf} at compile time...

- \texttt{buf[i]} is really just: (base_of_array) + i * elt_size
Multi-dimensional Arrays

- In C int m[4][3] yields an array with 4 rows and 3 columns.
  - Laid out in row-major order:
  - m[0][0], m[0][1], m[0][2], m[1][0], m[1][1], ...

<table>
<thead>
<tr>
<th>m[0][0]</th>
<th>m[0][1]</th>
<th>m[0][2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>m[1][0]</td>
<td>m[1][1]</td>
<td>m[1][2]</td>
</tr>
<tr>
<td>m[2][0]</td>
<td>m[2][1]</td>
<td>m[2][2]</td>
</tr>
<tr>
<td>m[3][0]</td>
<td>m[3][1]</td>
<td>m[3][2]</td>
</tr>
</tbody>
</table>
Multi-dimensional Arrays

• In C `int m[4][3]` yields an array with 4 rows and 3 columns.
  • Laid out in row-major order:
    • `m[0][0], m[0][1], m[0][2], m[1][0], m[1][1], ...`

```
| m[0][0] | m[0][1] | m[0][2] | m[1][0] | m[1][1] | m[1][2] | m[2][0] | m[2][1] | m[2][2] |
```

• So `m[i][j]` is located where?
  • `(base address of m) + (i * 3 * sizeof(int)) + j * sizeof(int)`
Multi-dimensional Arrays

- In Fortran, arrays are laid out in column major order.

- In ML, there are no multi-dimensional arrays.
  - `(int array) array` is represented as an array of pointers to arrays of ints.

- Why is knowing the memory layout strategy important?
Multi-dimensional Arrays

• In Fortran, arrays are laid out in column major order

```
  m[0][0]  m[1][0]  m[2][0]  m[3][0]  m[0][1]  m[1][1]  m[2][1]  m[3][1]  m[0][2]  m[1][2]  m[2][2]  m[3][2]  m[0][3]  m[1][3]  m[2][3]  m[3][3]
```

• In ML, there are no multi-dimensional arrays
  • (int array) array is represented as an array of pointers to arrays of ints

• Why is knowing the memory layout strategy important?
Array Bounds Checks

• Safe languages (e.g. Java, C#, ML but not C, C++) check array indices to ensure that they are in bounds.
  • Compiler generates code to test that the computed offset is legal
• Needs to know the size of the array… where to store it?
  • One answer: Store the size **before** the array contents.

```
arr
```

```
```

• Other possibilities:
  • Store size and a pointer to array data
  • Pascal: only permit statically known array sizes (very unwieldy in practice)
  • What about multi-dimensional arrays?
Array Bounds Checks (Implementation)

• Example: Assume %rax holds the base pointer (arr) and %ecx holds the array index i. To read a value from the array arr[i]:

```assembly
  movq -8(%rax) %rdx       // load size into rdx
  cmpq %rdx %rcx           // compare index to bound
  j l __ok                // jump if 0 <= i < size
  callq __err_oob         // test failed, call the error handler
__ok:
  movq (%rax, %rcx, 8) dest // do the load from the array access
```

• Clearly more expensive: adds move, comparison & jump
  • More memory traffic
  • These overheads are particularly bad in an inner loop
  • Compiler optimizations can help remove the overhead
  • e.g. In a for loop, if bound on index is known, only do the test once

• Hardware support can improve performance: executing instructions in parallel, branch prediction
  • But speculative execution is behind the Spectre/Meltdown vulnerabilities...
C-style Strings

• A string constant "foo" is represented as global data:
  _string42: 0x66 0x6F 0x6F 0x00

• C uses null-terminated strings
• Strings are usually placed in the text segment so they are read only.
  • allows all copies of the same string to be shared.
• Rookie mistake (in C): write to a string constant.

```
char *p = "foo";
p[0] = 'b';
```

• Instead, must allocate space on the heap:

```
char *p = (char *)malloc(4 * sizeof(char));
strncpy(p, "foo", 4); /* include the null byte */
p[0] = 'b';
```
Tagged Datatypes
C-style Enumerations / ML-style datatypes

- In C:
  ```c
  enum Day {sun, mon, tue, wed, thu, fri, sat} today;
  ```

- In ML:
  ```ml
  type day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
  ```

- Associate an integer tag with each case: sun = 0, mon = 1, ...
  - C lets programmers choose the tags
- ML datatypes can also carry data:
  ```ml
  type foo = Bar of int | Baz of int * foo
  ```

- Representation: a foo value is a pointer to a pair: (tag, data)
- Example: tag(Bar) = 0, tag(Baz) = 1
  ```ml
  [let f = Bar(3)] =
  ```
  ```ml
  [let g = Baz(4, f)] =
  ```
Switch Compilation

• Consider the C statement:

```c
switch (e) {
    case sun: s1; break;
    case mon: s2; break;
    ...
    case sat: s3; break;
}
```

• How to compile this?
  • What happens if some of the break statements are omitted? (Control falls through to the next branch.)
Cascading ifs and Jumps

• Each \$tag1...\$tagN is just a constant int tag value.

• Note: \[break;\] (within the switch branches) is:
  
  br %merge

\[
\text{[switch(e) \{case tag1: s1; break; case tag2 s2; ...\}]} = \\
\%
tag = \[e];
\quad \text{br label \%l1}
\%
l1: \%cmp1 = icmp eq \%tag, \$tag1
\quad \text{br \%cmp1 label \%b1, label \%l2}
\%
b1: \[s1]\n\quad \text{br label \%merge}
\%
l2: \%cmp2 = icmp eq \%tag, \$tag2
\quad \text{br \%cmp2 label \%b2, label \%l3}
\%
b2: \[s2]\n\quad \text{br label \%l3}
\ldots
\%
lN: \%cmpN = icmp eq \%tag, \$tagN
\quad \text{br \%cmpN label \%bN, label \%merge}
\%
bN: \[sN]\n\quad \text{br label \%merge}

merge:
Alternatives for Switch Compilation

• Nested if-then-else works OK in practice if # of branches is small
  • (e.g. < 16 or so).

• For more branches, use better datastructures to organize the jumps:
  • Create a table of pairs \((v1, \text{branch\_label})\) and loop through
  • Or, do binary search rather than linear search
  • Or, use a hash table rather than binary search

• One common case: the tags are dense in some range \([\text{min}...\text{max}]\)
  • Let \(N = \text{max} – \text{min}\)
  • Create a branch table \(\text{Branches}[N]\) where \(\text{Branches}[i] = \text{branch\_label}\) for tag \(i\).
  • Compute tag = \([e]\) and then do an \textbf{indirect jump}: \text{J Branches}[\text{tag}]

• Common to use heuristics to combine these techniques.
ML-style Pattern Matching

- ML-style match statements are like C’s switch statements except:
  - Patterns can bind variables
  - Patterns can nest

- Compilation strategy:
  - “Flatten” nested patterns into matches against one constructor at a time.
  - Compile the match against the tags of the datatype as for C-style switches.
  - Code for each branch additionally must copy data from \([e]\) to the variables bound in the patterns.

- There are many opportunities for optimization, many papers about “pattern-match compilation”
  - Many of these transformations can be done at the AST level
Datatypes in the LLVM IR
Structured Data in LLVM

- LLVM’s IR is used to describe the structure of data.

\[ t ::= \]

\[ \text{void} \]

\[ \text{i1 | i8 | i64} \]

\[ [\text{<#elts> x t}] \]

\[ \text{fty} \]

\[ \{t_1, t_2, \ldots, t_n\} \]

\[ t^* \]

\[ \%\text{Tident} \]

\[ fty ::= \]

\[ t (t_1, \ldots, t_n) \]

- `<#elts>` is an integer constant \( \geq 0 \)
- Structure types can be named at the top level:

\[ \%T1 = \text{type} \{t_1, t_2, \ldots, t_n\} \]

- Such structure types can be recursive
Example LL Types

• An array of 341 integers: \([ 341 \times i64 ]\)

• A two-dimensional array of integers: \([ 3 \times [ 4 \times i64 ] ]\)

• Structure for representing arrays with their length:
  \[
  \{ i64 , [0 \times i64] \}
  \]
  • 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 size computations 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
  ```
```c
struct RT {
    int A;
    int B[10][20];
    int C;
}
struct ST {
    struct RT X;
    int Y;
    struct RT Z;
}
int *foo(struct ST *s) {
    return &s[1].Z.B[5][13];
}

%RT = type { i32, [10 x [20 x i32]], i32 }
%ST = type { %RT, i32, %RT }
define i32* @foo(%ST* %s) {
    entry:
        %arrayidx = getelementptr %ST* %s, i32 1, i32 2, i32 1, i32 5, i32 13
        ret i32* %arrayidx
}

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

*adapted from the LLVM documentation: see https://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 loading from the computed pointer
  - See list.ll from HW3
Compiling Data Structures via LLVM

1. Translate high level language types into an LLVM representation type.
   - For some languages (e.g. C) this process is straightforward
     - The translation simply uses platform-specific alignment and padding
   - For other languages, (e.g. OO languages) might be complex elaboration.
     - e.g. for OCaml, arrays types might be translated to pointers to length-indexed structs.
       \[\text{int array} = \{ \text{i32, [0 x i32]} \}^*\]

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

```plaintext
%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
  ...
}
```
LLVMlite Specification

• see HW3
LLVMlite notes

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

```plaintext
%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() {
    %l = 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
Compiling LLVMlite to x86
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
  • We will see how to do this later in the semester

• 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**
  - must use `%rip` relative addressing

- **Calls**
  - Follow x64 AMD ABI calling conventions
  - Should interoperate with C programs

- **`getelementptr`**
  - trickiest part