Announcements

• Poll indicated preference for keeping the midterm on Feb. 27\textsuperscript{th}
  – Talk to Prof. Zdancewic if you have a conflict

• HW3: LLVM lite
  – Available on the course web pages.
  – Due: Wednesday, Mar. 4th at 11:59:59pm
  – Only one group member needs to submit
  – Three submissions per group

\textbf{START EARLY!!}
STRUCTURED DATA
ARRAYS
Arrays

void foo() {
    char buf[27];

    buf[0] = 'a';
    buf[1] = 'b';
    ...
    buf[25] = 'z';
    buf[26] = 0;
}

void foo() {
    char buf[27];

    *(buf) = 'a';
    *(buf+1) = 'b';
    ...
    *(buf+25) = 'z';
    *(buf+26) = 0;
}

• Space is allocated on the stack for buf.
  – Note, without the ability to allocated stack space dynamically (C’s *alloca* function) need to know size of buf at compile time…

• 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[i][j]` compiles to?

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

  |---------|---------|---------|---------|---------|---------|---------|-----|

- In ML and Java, there are no multi-dimensional arrays:
  - `(int array) array` is represented as an array of pointers to arrays of ints.
- Why is knowing these memory layout strategies important?
Array Bounds Checks

- Safe languages (e.g. Java, C#, ML but not C, C++) check array indices to ensure that they're 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:
  - 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]`:
  
  ```
  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
  - Hardware can improve performance: executing instructions in parallel, branch prediction

- 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
C-style Strings

- A string constant "foo" is represented as global data:
  
  _string42: 102 111 111 0

- 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.

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

- Instead, must allocate space on the heap:

```c
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

\[
[\text{switch}(e) \ {\{\text{case tag1: s1; case tag2 s2; ...}\}}] =
\]

- Each $\text{tag}_1...\text{tag}_N$ is just a constant int tag value.

- Note: \[\text{break;}\] (within the switch branches) is:
  
  \begin{verbatim}
  br %merge
  \end{verbatim}

\begin{verbatim}
%tag = [e];
br label %l1
l1: %cmp1 = icmp eq %tag, $tag1
    br %cmp1 label %b1, label %merge
b1: [%s1]
    br label %l2

l2: %cmp2 = icmp eq %tag, $tag2
    br %cmp2 label %b2, label %merge
b2: [%s2]
    br label %l3
...

lN: %cmpN = icmp eq %tag, $tagN
    br %cmpN label %bN, label %merge
bN: [%sN]
    br label %merge

merge:
\end{verbatim}
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, 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 [min…max]
  - Let N = max – min
  - Create a branch table Branches[N] where Branches[i] = branch_label for tag i.
  - Compute tag = [e] and then do an *indirect jump*: J Branches[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 uses types to describe the structure of data.

\[
t ::= \\
\text{void} \\
i1 \mid i8 \mid i64 \\
[<\#elts> \times t] \\
fty \\
\{t_1, t_2, \ldots, t_n\} \\
t^* \\
\%Tident
\]

\[
fty ::= \text{Function Types} \\
t (t_1, .., t_n) \text{ return, argument types}
\]

- `<\#elts>` is an integer constant >= 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: \[ 341 \times \text{i64} \]

• A two-dimensional array of integers: \[ 3 \times [ 4 \times \text{i64} ] \]

• Structure for representing arrays with their length:
  \{ \text{i64} , [0 \times \text{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 = \text{type} \{ \text{i64}, \%Node* \}

• Structs from the C program shown earlier:
  \%Rect = \{ \%Point, \%Point, \%Point, \%Point \}
  \%Point = \{ \text{i64}, \text{i64} \}
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
  ```
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 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 loading from 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.

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

2. Translate accesses of the data into getelementptr operations:
   - e.g. for OCaml array size access:

   \[
   \text{[length a]} = \\
   \%1 = \text{getelementptr} \{\text{i32, [0xi32]}\}^* \%a, \text{i32} 0, \text{i32} 0
   \]
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)

```asm
%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
LLVMlite notes

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

```cpp
%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
  – we assume that the i64 value will always fit in 32 bits
COMPILING LLVM-LITE 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
see HW3 and README

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

TOUR OF HW 3