Lecture 8

CIS 341: COMPILERS
Announcements

- HW3: LLVM lite
  - Available on the course web pages.
  - Due: Monday, Feb. 26th at 11:59:59pm
  - Only one group member needs to submit
  - Three submissions per group

START EARLY!!
STRUCTURED DATA
• Consider C-style structures like those below.
• How do we represent \texttt{Point} and \texttt{Rect} values?

```c
struct Point { int x; int y; };

struct Rect { struct Point ll, lr, ul, ur; }

struct Rect mk_square(struct Point ll, int len) {
    struct Rect square;
    square.ll = square.lr = square.ul = square.ur = ll;
    square.lr.x += len;
    square.ul.y += len;
    square.ur.x += len;
    square.ur.y += len;
    return square;
}
```
Representing Structs

```c
struct Point { int x; int y;};
• Store the data using two contiguous words of memory.
• Represent a Point value p as the address of the first word.

struct Rect { struct Point ll, lr, ul, ur;};
• Store the data using 8 contiguous words of memory.

• Compiler needs to know the size of the struct at compile time to allocate the needed storage space.
• Compiler needs to know the shape of the struct at compile time to index into the structure.
```
Assembly-level Member Access

Consider: $\text{square.ul.y} = (x86.\text{operand}, x86.\text{insns})$

Assume that $\%rcx$ holds the base address of $\text{square}$

Calculate the offset relative to the base pointer of the data:
- $ul = \text{sizeof(struct Point)} + \text{sizeof(struct Point)}$
- $y = \text{sizeof(int)}$

So: $\text{square.ul.y} = (\text{ans}, \text{Movq 20(\%rcx) ans})$

```c
struct Point { int x; int y; };
struct Rect { struct Point ll, lr, ul, ur; }
```
• How to lay out non-homogeneous structured data?

```c
struct Example {
    int x;
    char a;
    char b;
    int y;
};
```

32-bit boundaries

Padding

Not 32-bit aligned
Copy-in/Copy-out

When we do an assignment in C as in:

```c
struct Rect mk_square(struct Point ll, int elen) {
    struct Square res;
    res.lr = ll;
    ...
```

then we copy all of the elements out of the source and put them in the target. Same as doing word-level operations:

```c
struct Rect mk_square(struct Point ll, int elen) {
    struct Square res;
    res.lr.x = ll.x;
    res.lr.y = ll.x;
    ...
```

• For really large copies, the compiler uses something like `memcpy` (which is implemented using a loop in assembly).
C Procedure Calls

- Similarly, when we call a procedure, we copy arguments in, and copy results out.
  - Caller sets aside extra space in its frame to store results that are bigger than will fit in `%rax`.
  - We do the same with scalar values such as integers or doubles.
- Sometimes, this is termed "call-by-value".
  - This is bad terminology.
  - Copy-in/copy-out is more accurate.
- Benefit: locality
- Problem: expensive for large records...

- In C: can opt to pass *pointers* to structs: “call-by-reference”

- Languages like Java and OCaml always pass non-word-sized objects by reference.
Call-by-Reference:

```
void mkSquare(struct Point *ll, int elen,  
              struct Rect *res) {
    res->lr = res->ul = res->ur = res->ll = *ll;
    res->lr.x += elen;
    res->ur.x += elen;
    res->ur.y += elen;
    res->ul.y += elen;
}
void foo() {
    struct Point origin = {0,0};
    struct Square unit_sq;
    mkSquare(&origin, 1, &unit_sq);
}
```

- The caller passes in the address of the point and the address of the result (1 word each).
Stack Pointers Can Escape

• Note that returning references to stack-allocated data can cause problems...

```c
int* bad() {
    int x = 341;
    int *ptr = &x;
    return ptr;
}
```

– see unsafestack.c

• For data that persists across a function call, we need to allocate storage in the heap...
  – in C, use the malloc library
ARRAYS
• 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, \texttt{int M[4][3]} yields an array with 4 rows and 3 columns.
• Laid out in \textit{row-major} order:

\begin{verbatim}
\end{verbatim}

• \texttt{M[i][j]} compiles to?

• In Fortran, arrays are laid out in \textit{column major order}.

\begin{verbatim}
\end{verbatim}

• 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:
  - 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: 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:
  ```ocaml
  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:
  ```ocaml
  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
  ```ocaml
  [let f = Bar(3)] = f
  [let g = Baz(4, f)] = g
  ```
Switch Compilation

• Consider the C statement:

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

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

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

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

```assembly
%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:
```
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

```
match e with
  | Bar(z) -> e1
  | Baz(y, Bar(w)) -> e2
  | _ -> e3
```

- 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.

\[
\begin{align*}
t &::= \\
&\quad\text{void} \\
&\quad\text{i1 | i8 | i64} \quad \text{N-bit integers} \\
&\quad[<\#\text{elts}> \ x \ t] \quad \text{arrays} \\
&\quad\text{ftys} \\
&\quad\{t_1, t_2, \ldots, t_n\} \quad \text{function types} \\
&\quad t* \\
&\quad \%\text{Tident} \\
\end{align*}
\]

\[
\begin{align*}
\text{fty} &::= \\
&\quad\text{Function Types} \\
&\quad t\ (t_1, \ldots, t_n) \quad \text{return, argument types}
\end{align*}
\]

• \(<\#\text{elts}>\) 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: \[ 341 \times \text{i64} \]

• 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):
  \[
  \%\text{Node} = \text{type} \{ \text{i64}, \%\text{Node}* \}
  \]

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

   \[
   \begin{align*}
   \text{[int array]} &= \{ \text{i32}, [0 \times \text{i32}]\}^* \\
   \end{align*}
   \]

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

   \[
   \begin{align*}
   \text{[length a]} &= \\
   \%1 &= \text{getelementptr} \{ \text{i32}, [0\times\text{i32}]\}^* \%a, \text{i32 0, i32 0} \\
   \end{align*}
   \]
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)

```llvm
%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
Discussion: Defining a Language

• Premise: programming languages are purely ‘formal’ objects
  – We (as language designers) get to determine the meaning of the language constructs

• Question: How do we specify that meaning?

• Question: What are the properties of a good specification?

• Examples?
Approaches to Language Specification

• Implementation
  – It does what it does!

• Social
  – Authority figure says: “it means X”
  – English prose

• Technological
  – Multiple implementations
  – Reference interpreter
  – Test cases / Examples

• Translation
  – Semantics given in terms of (hopefully better specified) target

• Mathematical
  – “Informal” specifications
  – “Formal” specifications

Less “formal”: Techniques may miss problems in programs

This isn’t a tradeoff... all of these methods should be used.

Even the most “formal” can still have holes:

• Did you prove the right thing?
• Do your assumptions match reality?
• Knuth. “Beware of bugs in the above code; I have only proved it correct, not tried it.”

More “formal”: eliminate with certainty as many problems as possible.
LLVMlite notes

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

```assembly
global %struct {i64 1,
    [5 x i64] [i64 2, i64 3, i64 4, i64 5, i64 6], i64 7}
```

• 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

- $\llbracket i1 \rrbracket, \llbracket i64 \rrbracket, \llbracket t^* \rrbracket =$ 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. $\llbracket i1 \rrbracket =$ 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