Lecture 8

#### **CIS 4521/5521: COMPILERS**

#### Announcements

- HW2: X86lite
  - Due: TOMORROW Feb. 12th at 10:00pm

# **Intermediate Representations**

- IR1: Expressions
  - *immutable* global variables
  - simple arithmetic *expressions*
- IR2: Commands
  - *mutable* global variables
  - commands for update and sequencing
- IR3: Local control flow
  - conditional commands & while loops
  - basic blocks
- IR4: Procedures (top-level functions)
  - local variables
  - call stack
- IR5: "almost" LLVM IR
  - missing *phi-nodes* (explained when we get there)

#### **Basic Blocks**

- A sequence of instructions that is always executed starting at the first instruction and always exits at the last instruction.
  - Starts with a label that names the *entry point* of the basic block.
  - Ends with a control-flow instruction (e.g., branch or return) the "link"
  - Contains no other control-flow instructions
  - Contains no interior label used as a jump target
- Basic blocks can be arranged into a *control-flow graph* 
  - Nodes are basic blocks
  - There is a directed edge from node A to node B if the control flow instruction at the end of basic block A might jump to the label of basic block B.

#### **Control-flow Graphs**



See Ilvm.org



Zdancewic CIS 4521/5521: Compilers

# Low-Level Virtual Machine (LLVM)

- Open-Source Compiler Infrastructure
  - see llvm.org for full documentation
- Created by Chris Lattner (advised by Vikram Adve) at UIUC
  - LLVM: An infrastructure for Mult-stage Optimization, 2002
  - LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation, 2004
- 2005: Adopted by Apple for XCode 3.1
- Front ends:
  - Ilvm-gcc (drop-in replacement for gcc)
  - Clang: C, objective C, C++ compiler supported by Apple
  - various languages: Swift, ADA, Scala, Haskell, ...
- Back ends:
  - x86 / Arm / Power / etc.
- Used in many academic/research projects
  - Here at Penn: SoftBound, Vellvm



#### **LLVM Compiler Infrastructure**

[Lattner et al.]



# IR3/4/5 vs. LLVM

- "let in" and OCaml-style identifiers:
- let tmp1 = add 3L 4L in
- OCaml-style "let-rec" and functions for blocks:

```
let rec entry () =
    let tmp1 = ...
and foo () =
    let tmp2 = ...
```

OCaml-style global variables:
 let varX = ref 0L

• Omits let/in and prefixes local identifiers with %:

%tmp1 = add i64 3, i64 4

• Uses lighter-weight colon notation:

```
entry:
%tmp1 = ...
foo:
%tmp2 = ...
```

 Prefixes globals with @ define @X = i64 0

#### **Example LLVM Code**



}

# **Real LLVM**

Decorates values with type information ullet

factorial.

- i64 i64\* i 1
- Permits numeric ulletidentifiers
- Has alignment • annotations
- Keeps track of ٠ entry edges for each block: preds = %5, %0

```
; Function Attrs: nounwind ssp
define i64 @factorial(i64 %n) #0 {
  %1 = alloca i64, align 8
  %acc = alloca i64, align 8
  store i64 %n, i64* %1, align 8
  store i64 1, i64* %acc, align 8
  br label %2
                                      ; preds = %5, %0
; <label>:2
  %3 = load i64* %1, align 8
 %4 = icmp sgt i64 %3, 0
  br i1 %4, label %5, label %11
; <label>:5
                                      ; preds = \%2
  %6 = load i64* %acc, align 8
 %7 = load i64* %1, align 8
 %8 = mul nsw i64 %6, %7
  store i64 %8, i64* %acc, align 8
  %9 = load i64* %1, align 8
  %10 = sub nsw i64 %9, 1
  store i64 %10, i64* %1, align 8
  br label %2
                                      ; preds = \%2
; <label>:11
  %12 = load i64* %acc, align 8
  ret i64 %12
}
```

#### **Example Control-flow Graph**

define @factorial(%n) {



}

#### **LL Basic Blocks and Control-Flow Graphs**

- LLVM enforces (some of) the basic block invariants syntactically.
- Representation in OCaml:

```
type block = {
    insns : (uid * insn) list;
    term : (uid * terminator)
}
```

- A *control flow graph* is represented as a list of labeled basic blocks with these invariants:
  - No two blocks have the same label
  - All terminators mention only labels that are defined among the set of basic blocks
  - There is a distinguished, unlabeled, entry block:

```
type cfg = block * (lbl * block) list
```

# **LL Storage Model**

- Several kinds of storage:
  - Local variables (or temporaries): %uid
  - Global declarations (e.g., for string constants): @gid
  - Abstract locations: references to (stack-allocated) storage created by the alloca instruction
  - Heap-allocated structures created by external calls (e.g., to malloc)

### **LL Storage Model: Locals**

- Local variables:
  - Defined by the instructions of the form %uid = …
  - Analogous to "let %uid = e in ..." in OCaml

*static single assignment* (SSA) invariant:

Each **%uid** appears on the left-hand side of an assignment **only once** in the entire control flow graph.

- The value of a %uid remains unchanged throughout its lifetime
- Intended to be an abstract version of machine registers.
- We'll see soon how to extend SSA to allow richer use of local variables
  - *phi nodes* (allow "controlled mutation" of uids)

# LL Storage Model: alloca

- alloca instruction allocates stack space and returns a reference to it. •
  - The returned reference is stored in local:

%ptr = alloca type

- The amount of space allocated is determined by the type
- The contents of the slot are accessed via the load and store ۲ instructions:

%acc = alloca i64 store i64 4521, i64\* %acc ; store the integer value 4521 x = 10ad i64, i64\* acc = i0ad the value 4521 into <math>x

- ; allocate a storage slot

- Gives an abstract version of stack slots

# **STRUCTURED DATA**

Zdancewic CIS 4521/5521: Compilers

# **Compiling Structured Data**

- Consider C-style structures like those below.
- How do we represent **Point** and **Rect** values?

```
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**

#### 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**

struct Point { int x; int y; };

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

- Consider: **[[square.ul.y]]** = (x86.operand, x86.insns)
- Assume that %rcx holds the base address of square
- Calculate the offset relative to the base pointer of the data:
  - ul = sizeof(struct Point) + sizeof(struct Point)
  - y = sizeof(int)
- So: [[square.ul.y]] = (ans, Movq 20(%rcx) ans)

# Padding & Alignment

• How to lay out non-homogeneous structured data?



#### **Copy-in/Copy-out**

When we do an assignment in C as in:

```
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:

```
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:**

• 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...



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



Zdancewic CIS 4521/5521: Compilers

#### 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[0][0] M[0][1] M[0][2] M[1][0] M[1][1] M[1][2] M[2][0] ...

- M[i][j] compiles to?
- 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]	
--	---------	---------	---------	---------	---------	---------	---------	--

- 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

#### Size=7 A[0] A[1] A[2] A[3] A[4] A[5] A[6]

- 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]:

```
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</pre>
```

- 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.

• 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

Zdancewic CIS 4521/5521: Compilers

#### **C-style Enumerations / ML-style datatypes**

• In C:

enum Day {sun, mon, tue, wed, thu, fri, sat} today;

• In 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:

type foo = Bar of int | Baz of int \* foo

g

1

4

• Representation: a **foo** value is a pointer to a pair: (tag, data)

# **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

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

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

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

```
%tag = [[e]];
   br label %l1
l1: %cmp1 = icmp eq %tag, $tag1
   br %cmp1 label %b1, label %merge
b1: [s1]
   br label %12
l2: %cmp2 = icmp eq %tag, $tag2
   br %cmp2 label %b2, label %merge
b2: [s2]
   br label %13
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).</li>
- 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**

Zdancewic CIS 4521/5521: 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:

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

- Such structure types can be recursive

#### **Example LL Types**

- An array of 4521 integers: [ 4521 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 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

#### **GEP Example\***



Zdancewic CIS 4521/5521: C \*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:

# **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 lib/ll/ll.ml

# **LLVMLITE SPECIFICATION**

Zdancewic CIS 4521/5521: Compilers

# **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

# **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 (see today's lecture)

# **Other LLVMlite Features**

- Globals
  - must use %rip relative addressing
- Calls
  - Follow x64 AMD ABI calling conventions
  - Should interoperate with C programs
- More types: structured data records and arrays
- New instruction: getelementptr
  - LLVM IR's way of dealing with structured data
  - trickiest part of the compilation process
  - note: you can start HW3 before understanding getelementptr
- New instruction: bitcast
  - convert between pointer types

#### **LLVMlite notes**

• Real 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