Lecture 9

CIS 4521/5521: COMPILERS

Announcements

- HW2 – last minute git issues
  - all submitted test cases are fine, but there was some last-minute scrambling due to merge conflicts / accidental deletions
  - going forward: *please be careful*
  - we'll *require* test case submission 24 hours earlier
  - (this also gives everyone time to test their code)

- HW3: LLVM Backend
  - Available on the course web pages.
  - Due: Weds., February 26th at 10:00PM
  - Note: test cases should be submitted 24 hours earlier (so by Tues., Feb. 25th at 10pm)

START EARLY!!

- Midterm: March 6th
  - In class
  - One-page, letter-sized, double-sided "cheat sheet" of notes permitted
  - See Ed post (soon) for previous exams

Plan for Today

1. Continue tour of datatypes
   - tagged types / matching
2. LLVM IR's types
3. Overview of HW3

TAGGED DATATYPES

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

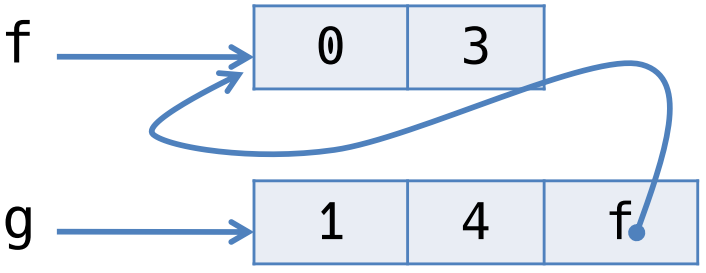
- ML datatypes can also carry data:

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

- Representation: a `foo` value is a pointer to a pair: (tag, data)
- Example: tag(`Bar`) = `0`, tag(`Baz`) = `1`

⟦`let f = Bar(3)`⟧ =

⟦`let g = Baz(4, f)`⟧ =



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
  rather than
  br %b_(i+1)

```
     %tag = ⟦e⟧;
     br label %l1
l1:  %cmp1 = icmp eq %tag, $tag1    ;; compare tags
     br %cmp1 label %b1, label %l2  ;; case 1 or case 2?
b1:  ⟦s1⟧
     br label %b2                   ;; fallthru to case 2

l2:  %cmp2 = icmp eq %tag, $tag2    ;; compare tags
     br %cmp2 label %b2, label l3   ;; case 2 or case 3?
b2:  ⟦s2⟧
     br label %b4 ;;                ;; use %merge if break
…
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]` is the 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
```

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

```
t ::=
    void
    i1 | i8 | i64          N-bit integers
    [<#elts> x t]          arrays
    fty                    function types
    {t1, t2, … , tn}       structures
    t*                     pointers
    %Tident                named (identified) type

fty  ::=                   Function Types
    t (t1, .., tn)                 return, argument types
```

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

```
%T1 = type {t1, t2, … , tn}
```

  - 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 }
```

Hiding Pointer Calculations

Question:

How do we abstract away from the details of pointer calculation?

- the low-level layout may be target dependent

getelementptr

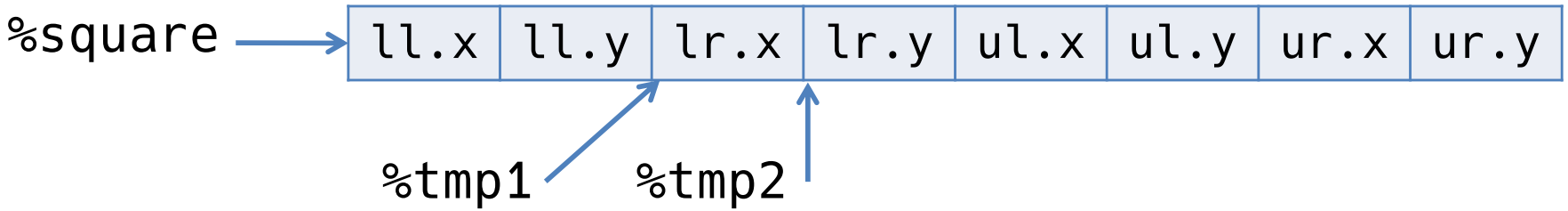
- LLVM provides the `getelementptr` instruction to compute pointer values
  - Given a pointer and a "path" through the structured data pointed to by that pointer, `getelementptr` computes an address
  - This is the abstract analog of the X86 `leaq` (load effective address) instruction. It does not access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

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

Example Uses of GEP

- Example: access the y component of the 2nd point of a rectangle:

```
;; Type information
%Rect  = { %Point, %Point, %Point, %Point }
%Point = { i64, i64 }
```




```
%tmp1 = getelementptr %Rect* %square, i32 0, i32 1
%tmp2 = getelementptr %Point* %tmp1, i32 0, i32 1
```

`%tmp2` can be equivalently calculated "all in one go" using a longer path:

```
%tmp2 = getelementptr %Rect* %square, i32 0, i32 1, i32 1
```

GEP Example*

```
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];
}
```

1. %s is a pointer to an (array of) %ST structs, suppose the pointer value is ADDR

2. Compute the index of the 1st element by adding size_ty(%ST).

3. Compute the index of the Z field by adding size_ty(%RT) + size_ty(i32) to skip past X and Y.

4. Compute the index of the B field by adding size_ty(i32) to skip past A.

5. Index into the 2d array.

```
%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 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 a linked-data structure that has embedded pointers, you need to "follow the pointer" by using the `load` instruction from the computed pointer
  - See `llprograms/list.ll` from HW3 for an example

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 and README

ll.ml.

TOUR OF HW 3

see HW3 lib/ll/ll.ml

LLVMLITE SPECIFICATION

LLVMlite vs "real" LLVM IR

- LLVM IR supports a few more types
  - arbitrary bitwidth integers: `i3, i17, i128, i12, iX`
  - `packed structures, vectors`

- LLVM IR has has more support for aggregate datatypes
  - `alloca` can allocate arbitrary types in the stack
  - there are operations for creating/manipulating such values (e.g., `extractelement`)

- There are a few other instructions:
  - `select` – choose between values
  - a few other kinds of control flow (for "exceptions" and "switch" statements)

- Pointer types can be written in an "undecorated form" as just `ptr`
  - this saves type annotations and reduced the need for `bitcast`
  - but is harder to debug

- So-called *intrinsics*
  - special-purpose instructions named `llvm.*` that are treated by the compiler
    e.g.: `llvm.memcpy` or `llvm.log2`

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

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 slot
  - 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 compile to use `%rip` relative addressing

- Calls/Function Bodies
  - 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