Lecture 13

CIS 341: COMPILERS

Announcements

- Midterm Exam:
  - Thursday, February 28th
  - In class
  - Examples on the web

DATATYPES IN THE IR

Structured Data in LLVM

- LLVM's IR is uses types to describe the structure of data.

```
t ::=
    i32                          32-bit integers
    [<#elts> x t]                arrays
    r (t1, t2, … , tn)           function types
    {t1, t2, … , tn}             structures
    t*                           pointers
    %Tident                      named (identified) type

r ::=              Return Types
    t                            first-class type
    void                         no return value
```

- <#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 341 integers: `[ 341 x i32 ]`
- A two-dimensional array of integers: `[ 3 x [ 4 x i32 ] ]`
- Structure for representing arrays with their length:

```
{ i32 , [0 x i32] }
```

  - 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 { i32, %Node*}
```

- Structs from the C program shown earlier:

```
%Rect = { %Point, %Point, %Point, %Point }
%Point = { i32, i32 }
```

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 sizes computations involved depend on the type

```
insn ::= …
       |  %uid = 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
```

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 `sizeof(struct ST)`.

3. Compute the index of the `Z` field by adding `sizeof(struct RT) + sizeof(int)` to skip past `X` and `Y`.

4. Compute the index of the `B` field by adding `sizeof(int)` 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 + `sizeof(struct ST) + sizeof(struct RT) + sizeof(int) + sizeof(int) + 5*20*sizeof(int) + 13*sizeof(int)`

*adapted from the LLVM documentaion: see http://llvm.org/docs/LangRef.html#getelementptr-instruction

Other Adjustments to the IR

- LLVM's `alloca` instruction accepts a type:

```
%uid = alloca <type>
```

- External linkage to call C functions:
  - Needed for malloc

- Project 4 will provide a simplified interface to getelementptr
  - Only support for arrays (structs aren't needed until Project 5)

- getelementptr in practice:
  - See struct.c and struct.ll
  - Note that we will ignore lots of alignment and other annotations (e.g. `inbound`) seen in "real" LLVM code
  - We will generate them for you from the LL subset of LLVM.

FIRST-CLASS FUNCTIONS

"Functional" languages

- Languages like ML, Haskell, Scheme, Python, C#, (maybe eventually Java?) include *first-class* functions.
- Functions can be passed as arguments (e.g. map or fold)
- Functions can be returned as values (e.g. compose)
- Functions nest: inner function can refer to variables bound in the outer function

```
let add = fun x -> fun y -> x + y
let inc = add 1
let dec = add -1

let compose = fun f -> fun g -> fun x -> f (g x)
let id = compose inc dec
```

- How do we implement such functions?

Free Variables

```
let add = fun x -> fun y -> x + y
let inc = add 1
```

- The result of `add 1` is a function
- After calling `add`, we can't throw away its argument (or its local variables) because those are needed in the function returned by add.
- We say that the variable `x` is *free* in `fun y -> x + y`
  - Free variables are defined in an outer scope
- We say that the variable `y` is *bound* by "`fun y`" and its scope is the body "`x + y`" in the expression `fun y -> x + y`

- A term with no free variables is called *closed*.
- A term with one or more free variables is called *open*.

Substitution Semantics

- Consider how to evaluate such functions:

```
inc = add 1
   = (fun x -> fun y -> x + y) 1
   = fun y -> 1 + y
```

- Similarly

```
dec = fun y -> -1 + y
```

- So:

```
id = compose inc dec
   = (fun f -> (fun g -> fun x -> f (g x))) inc dec
   = (fun g -> fun x -> inc (g x)) dec
   = fun x -> (fun y -> 1 + y) (dec x)
   = fun x -> (1 + (dec x))
   = fun x -> (1 + ((fun y -> -1 + y) x))
   = fun x -> (1 + (-1 + x))
```

Substitutions Continued

- When we *substitute* a value `v` for some variable `x` in an expression `e` (written `subst v x e`), we replace all *free occurrences* of `x` in `e` by `v`.
- Function application can be interpreted by substitution:

```
   (fun x -> fun y -> x + y) 1
= subst 1 x (fun y -> x + y)
= (fun y -> 1 + y)
```

(Untyped) Lambda Calculus

- The lambda calculus is a minimal programming language.
  - Note: we're writing `(fun x -> e)` lambda-calculus notation: λ x. e
- It has variables, functions, and function application.
  - That's it! (Though for examples, I'll add int and +)
  - It's Turing Complete.
  - It's the foundation for a *lot* of research in programming languages.
  - Basis for "functional" languages like Scheme, ML, Haskell, etc.

Abstract syntax in OCaml:

```
type exp =
  | Var of var          (* variables                  *)
  | Fun of var * exp    (* functions: fun x -> e *)
  | App of exp * exp    (* function application   *)
```

Concrete syntax:

exp ::=

| x	*variables*
| `fun` x `->` exp	*functions*
| exp_1 exp_2	*function application*
| `(` exp `)`	*parentheses*