Lecture 11

CIS 341: COMPILERS

Announcements

- Project 3: Compiling Control Flow
  - Due: Monday, February 25th at 11:59pm
- Midterm Exam:
  - Thursday, February 28th
  - In class
  - Examples on the web

SCOPE AND CONTEXTS

Variable Scoping

- Consider the problem of determining whether a programmer-declared variable is in scope.

- See: Project 3 web pages for OAT's scoping rules.

- Issues:
  - Which variables are available at a given point in the program?
  - Shadowing – is it permissible to re-use the same identifier, or is it an error?

- Solution:
  - Contexts

Notation for Scope Checking

- Contexts (using OCaml list notation):

```
G ::= [ ]   |   IDENT::G
```

- Syntax-directed "functions" that say how to compositionally check the scope
  - One function for each syntactic category of the grammar.
  - Each function takes an input context (variables that are in scope)
  - May produce an output context (if new variables are introduced)

$G \vdash exp$

$G \vdash stmt$

$G \vdash vdecl \Rightarrow G$

$G \vdash vdecl\_list \Rightarrow G$

$G \vdash block \Rightarrow G$

$G \vdash prog$

Generalizing ‘if’ & Inference Rules

- We can read a judgment $\texttt{G} \vdash s$ as “The variables in statement s are well-scoped in the context $\texttt{G}$.”
- For any environment G, expression e, and statements s_1, s_2.

$$\texttt{G} \vdash \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2$$

holds if $\texttt{G} \vdash e$ and $\texttt{G} \vdash s_1$ and $\texttt{G} \vdash s_2$ all hold.

- More succinctly: we summarize these constraints as an *inference rule*:

Premises

$$\frac{\texttt{G} \vdash \texttt{e} \qquad \texttt{G} \vdash \texttt{s}_1 \qquad \texttt{G} \vdash \texttt{s}_2}{\texttt{G} \vdash \texttt{if}\ (\texttt{e})\ \texttt{s}_1\ \texttt{else}\ \texttt{s}_2}$$

Conclusion

- This rule can be used for *any* substitution of the syntactic metavariables $\texttt{G}$, e, s_1 and s_2.

Checking Derivations

- A *derivation* or *proof tree* has (instances of) judgments as its nodes and edges that connect premises to a conclusion according to an inference rule.
- Leaves of the tree are *axioms* (i.e. rules with no premises)
  - Example: the INT rule is an axiom
- Goal of the scope checker: verify that such a tree exists.
- Example1: Find a tree for the following program using the inference rules in oat0-defn.pdf:

```
int x1 = 0;
int x2 = x1 + x1;
x1 = x1 – x2;
return(x1);
```

Example2: There is no tree for this ill-scoped program:

```
int x2 = x1 + x1;
return(x2);
```

Why Inference Rules?

- They are a compact, precise way of specifying language properties.
  - E.g. ~20 pages for full Java vs. 100's of pages of prose Java Language Spec.

- Inference rules correspond closely to the recursive AST traversal that implements them

- Type checking (and type inference) is nothing more than attempting to prove a different judgment ($E \vdash e : T$) by searching backwards through the rules.
- Compiling in a context is nothing more than a collection of inference rules specifying yet a different judgment ($G \vdash \text{src} \Rightarrow \text{target}$)

- Strong mathematical foundations
  - The "Curry-Howard correspondence": Programming Language ~ Logic, Program ~ Proof, Type ~ Proposition
  - See CIS 500 next Fall if you're interested in type systems!

(BACK TO) LOCALS STORAGE

Abstract Storage: Locals

- Consider this factorial program:

```
int acc = 1;
int f = 6;
while (f > 0) {
  acc = acc * f;
  f = f – 1
}
return acc;
```

- When generating code for a declaration: `int acc = 1;`
  - Need to allocate some local storage space – a "stack slot" or a register
- When compiling the use of a variable: `acc = acc * f;`
  - the compiler needs to refer to the appropriate slot given the variable names.
- Managed by a *context* that maps variable identifiers to `%uids`

Locals and Contexts

- A local is just an abstract location with a unique identifier (uids)
  - The compiler can create new names as needed
  - Historically called "gen_sym" for "generate a symbol"
- The compiler manages a mapping from user-defined variable names (e.g. strings) to the uids
  - This mapping is a *context* (or *symbol table)*
  - It defines the *scope* of live variables just as in the "scope checking"

- There are many ways to store the map (e.g. Hash table); efficiency matters for industrial-scale compilers

Specifying Compilation with Judgments

- Just as for scope checking, there is one judgment form for each syntactic category:

$$C \vdash [\![\text{ exp }]\!] = \text{operand} * \text{insns}$$
$$C \vdash [\![\text{ stmt }]\!] = \text{insns}$$
$$C1 \vdash [\![\text{ vdecl }]\!] = \text{insns}, C2$$
$$C2 \vdash [\![\text{ vdecl list }]\!] = \text{insns}, C2$$
$$C1 \vdash [\![\text{ block }]\!] = \text{insns}, C2$$
$$\vdash [\![\text{ prog }]\!] = \text{insns}$$

- Unlike scope checking, contexts C *map* variables to LLVM's `%uid`'s:

C ::= [] | x ↦ `%uid`, C

Example Compilation Rules

$$\frac{}{C \vdash [\![\texttt{Cimm}(i)]\!] = (\text{Const } i, [\,])}$$

$$\frac{x \mapsto \texttt{\%uid} \in C \qquad \texttt{\%tmp} = \texttt{gen\_sym()}}{C \vdash [\![\texttt{Id}\ x]\!] = (\texttt{\%tmp}, \ [\texttt{\%tmp} = \ \texttt{load} \ \texttt{\%uid}\,])}$$

$$\frac{C \vdash [\![e]\!] = (\texttt{\%val}, \text{defn}) \qquad \texttt{\%uid} = \texttt{gen\_sym()}}{C \vdash [\![\texttt{int}\ x = e;]\!] = \text{defn}@[\texttt{\%uid} = \texttt{alloca; store \%val, \%uid}], \ (x \mapsto \texttt{\%uid}, C)}$$

Tracking alloca'ed Slots

- Consider this program and its contexts:

```
                    []                  // initially empty
int x = 1;          [x↦%uid0]
int y = 0;          [x↦%uid0, y↦%uid1]
{
  int x = 3;        [x↦%uid2, x↦%uid0, y↦%uid1] // shadowing
  y = x + y;        [x↦%uid2, x↦%uid0, y↦%uid1]
}
y = x + y;          [x↦%uid0, y↦%uid1] // first binding again
return y;
```

- The context reflects the block-structured scoping of variables
  - So that the last use of x refers to the appropriate local uid
  - Some languages limit shadowing to simplify context management

Compiling the Context

- To generate X86 code from LLVM code, the compiler must map %uids to either registers or stack space.

- There are many correct implementations:
  - Example 1: Calculate the total number of distinct %uid values and then allocate enough stack space to hold all of them. Map each %uid to a particular offset into the stack.
  - Example 2: Same as Example 1, but try to "reuse" slots once it's clear that their values are no longer used (for example when the variables they store leave scope).
  - Example 3: Register allocation: Try to optimally pack %uid values into registers, using the stack only when necessary. (Later in the class.)

- Different choices about when to allocate space:
  - Allocate all of the space at once (e.g. at the start of the program)
  - Allocate space upon entering into a new block/scope

COMPILING CONTROL

Translating while

- Consider translating "`while(e) s`":
  - Test the conditional, if true jump to the body, else jump to the label after the body.

⟦`while(e) s`⟧ =

```
lpre:
    %cnd = ⟦e⟧
    %test = icmp eq %cnd, 0
    br %test, label %lpost, label %lbody
lbody:
     ⟦s⟧
     br %lpre
lpost:
```

- Note: writing `%cnd = ⟦e⟧` is slight pun
  - translating ⟦e⟧ generates *code* that puts the result somewhere, the conditional tests against the result, must thread code through
- Note: must also thread the context through as appropriate:
  - The "C ⊢" part of the judgment "C ⊢ ⟦ e ⟧ = …" has been omitted

Translating if-then-else

- Similar to while except that code is slightly more complicated because if-then-else must reach a merge and the else branch is optional.

  $C \vdash [\![\texttt{if (}e_1\texttt{) }s_1\texttt{ else }s_2]\!] =$

```
    %cnd = ⟦e⟧
    %test = icmp eq %cnd, 0
    br %test, label %else, label %then
then:
     ⟦s1⟧
     br %merge
else:
    ⟦s2⟧
     br %merge
merge:
```

- The compiler must also thread through the context as appropriate

OPTIMIZING CONTROL

Standard Evaluation

- Consider compiling the following program fragment:

```
if (x & !y | !w)
   z = 3;
else
   z = 4;
return z;
```

```
    %tmp1 = icmp Eq ⟦y⟧, 0         ; !y
    %tmp2 = and ⟦x⟧ ⟦tmp1⟧
    %tmp3 = icmp Eq ⟦w⟧, 0
    %tmp4 = or %tmp2, %tmp3
    %tmp5 = icmp Eq %tmp4, 0
    br %tmp4, label %else, label %then

then:
    store ⟦z⟧, 3
    br %merge

else:
    store ⟦z⟧, 4
    br %merge

merge:
    %tmp5 = load ⟦z⟧
    ret %tmp5
```

Observation

- Usually, we want the translation ⟦e⟧ to produce a value
  - $C \vdash ⟦e⟧$ = (operand, insns)
  - e.g. $C \vdash ⟦e_1 + e_2⟧$ = (`%tmp`, [`%tmp = add` $⟦e_1⟧$ $⟦e_2⟧$])

- But when the expression we're compiling appears in a test, the program jumps to one label or another after the comparison but otherwise never uses the value.

- In many cases, we can avoid "materializing" the value (i.e. storing it in a temporary) and thus produce better code.
  - This idea also lets usimplement different functionality too:
    e.g. short-circuiting boolean expressions

Idea: Use a different translation for tests

Expression translation: $C \vdash \mathcal{E}[\![e]\!] = (\text{operand, insns})$

Conditional translation: $C \vdash \mathcal{C}[\![e]\!]$ ltrue lfalse = insns

$C \vdash [\![\text{if (e) then s1 else s2}]\!] =$

```
   insns₃
then:
    ⟦s1⟧
    br %merge
else:
   ⟦s₂⟧
    br %merge
merge:
```

where

$C \vdash [\![s_1]\!] = insns_1$

$C \vdash [\![s_2]\!] = insns_2$

$C \vdash \mathcal{C}[\![e]\!]$ `then` `else` $= insns_3$

Notes:

- $\mathcal{C}[\![e]\!]$ takes two extra arguments: a "true" branch label and a "false" branch label.
- Doesn't "return a value"
- Aside: this is a form of continuation-passing translation…

Short Circuit Compilation: Expressions

- $C \vdash \mathcal{C}[\![e]\!]$ ltrue lfalse = insns

$$\frac{}{\texttt{C} \vdash \mathcal{C}[\![\texttt{0}]\!] \text{ ltrue lfalse} = [\texttt{br \%lfalse}]} \quad \text{FALSE}$$

$$\frac{\texttt{n != 0}}{\texttt{C} \vdash \mathcal{C}[\![\texttt{n}]\!] \text{ ltrue lfalse} = [\texttt{br \%ltrue}]} \quad \text{TRUE}$$

$$\frac{\texttt{C} \vdash \mathcal{C}[\![e]\!] \text{ lfalse ltrue} = \text{insns}}{\texttt{C} \vdash \mathcal{C}[\![\texttt{!}e]\!] \text{ ltrue lfalse} = \text{insns}} \quad \text{NOT}$$

Short Circuit Evaluation

- $C \vdash \mathcal{C}[\![e]\!]$ ltrue lfalse = insns

$$\frac{C \vdash \mathcal{C}[\![\texttt{e1}]\!]\ \text{ltrue}\ \texttt{right} = \text{insns}_1 \quad C \vdash \mathcal{C}[\![\texttt{e2}]\!]\ \text{ltrue lfalse} = \text{insns}_2}{C \vdash \mathcal{C}[\![\texttt{e1 | e2}]\!]\ \text{ltrue lfalse} = \begin{array}{l}\quad \texttt{insns}_1 \\ \texttt{right:} \\ \quad \texttt{insn}_2\end{array}}$$

$$\frac{C \vdash \mathcal{C}[\![\texttt{e1}]\!]\ \texttt{right}\ \text{lfalse} = \text{insns}_1 \quad C \vdash \mathcal{C}[\![\texttt{e2}]\!]\ \text{ltrue lfalse} = \text{insns}_2}{C \vdash \mathcal{C}[\![\texttt{e1 \& e2}]\!]\ \text{ltrue lfalse} = \begin{array}{l}\quad \texttt{insns}_1 \\ \texttt{right:} \\ \quad \texttt{insn}_2\end{array}}$$

where `right` is a fresh label

Implementing C⟦e⟧(ctxt,ltrue,lfalse)

- Sketch of an implementation: (a few interesting cases)

```
let rec c_compile (c:ctxt) (e:exp) (ltrue:Label.t) (lfalse:Label.t) =
   begin match e with
   | Cint (0l) -> [Br lfalse]
   | Cint _    -> [Br ltrue]
   | Id x ->
       let (tmp1, insns) = compile c (Id x) in
       let tmp2 = gen_sym () in
       insns >@ [tmp2 = icmp Eq tmp1, 0; Cbr(tmp2, lfalse, ltrue)]
   | Binop (And, e1, e2) ->  (* short circuiting evaluation *)
       let lright = mk_label() in
       let insns1 = c_compile e1 c lright lfalse in
       let insns2 = c_compile e2 c ltrue lfalse in
           insns1 >@ (Label lright) >:: insn2
   | Unop (Lognot, e1) ->
       c_compile e1 ctxt lfalse ltrue
   | …
```

Short-Circuit Evaluation

- Consider compiling the following program fragment:

```
if (x & !y | !w)
  z = 3;
else
  z = 4;
return z;
```

```
    %tmp1 = icmp Eq ⟦x⟧, 0
    br %tmp1, label %right2, label %right1

right1:
    %tmp2 = icmp Eq ⟦y⟧, 0
    br %tmp2, label %then, label %right2

right2:
    %tmp3 = icmp Eq ⟦w⟧, 0
    br %tmp3, label %then, label %else

then:
    store ⟦z⟧, 3
    br %merge

else:
    store ⟦z⟧, 4
    br %merge

merge:
    %tmp5 = load ⟦z⟧
    ret %tmp5
```