Lecture 16

CIS 341: COMPILERS
Announcements

• COVID19 Logistics
  – Changed due dates for remaining projects
  – Final exam still TBD: I will follow the university guidelines
    • my preference would be to allow a “take home” complete in your own time “homework-style” exam.
  – The university allows you to choose to take the class Pass/Fail
    • See Piazza post for ongoing discussion

• Zoom recordings uploaded in the afternoon after the lecture.

• HW4: OAT v. 1.0
  – Parsing & basic code generation
  – **Due: Monday, March 30**th

• HW5: now (tentatively) due: Friday, April 17th
• HW6: now (tentatively) due: Wednesday, April 29th
UNTYPED LAMBDA CALCULUS
(Untyped) Lambda Calculus

- The lambda calculus is a minimal programming language.
  - Note: we’re writing (fun x -> e) lambda-calculus notation: \( \lambda x. e \)

Abstract syntax in OCaml:

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

Concrete syntax:

```latex
exp ::= 
  | x                     variables
  | fun x -> exp           functions
  | exp_{1} exp_{2}        function application
  | ( exp )                parentheses
```
IMPLEMENTING THE INTERPRETER

See fun.ml
Scope, Types, and Context

STATIC ANALYSIS
Consider how to identify “well-scoped” lambda calculus terms

- Recall the free variable calculation
- Given: $G$, a set of variable identifiers, $e$, a term of the lambda calculus
- **Judgment:** $G \vdash e$ means “the free variables of $e$ are included in $G$”

$$\text{fv}(e) \subseteq G$$

- $\text{fv}(x) = \{x\}$
- $\text{fv}(\text{fun } x \rightarrow \text{exp}) = \text{fv}(\text{exp}) \setminus \{x\}$ (‘$x$’ is a bound in $\text{exp}$)
- $\text{fv}(\text{exp}_1 \text{exp}_2) = \text{fv}(\text{exp}_1) \cup \text{fv}(\text{exp}_2)$

$x \in G$  
\[ G \vdash x \]  
“the variable $x$ is free”

$G \vdash e_1$  
$G \vdash e_2$  
\[ G \vdash e_1 \text{ e}_2 \]  
“$G$ contains the free variables of $e_1$ and $e_2$”

$G \cup \{x\} \vdash e$  
\[ G \vdash \text{fun } x \rightarrow e \]  
“$x$ is available in the function body $e$”
Scope-checking Code

- Compare the OCaml code to the inference rules:
  - structural recursion over syntax
  - the check either “succeeds” or "fails"

```
let rec scope_check (g:VarSet.t) (e:exp) : unit =
  begin match e with
    | Var x -> if VarSet.member x g then () else failwith (x ^ "not in scope")
    | App(e1, e2) -> ignore (scope_check g e1); scope_check g e2
    | Fun(x, e) -> scope_check (VarSet.union g (VarSet.singleton x)) e
  end
```

\[
\begin{align*}
\frac{x \in G}{G \vdash x} & \quad \frac{G \vdash e_1 \quad G \vdash e_2}{G \vdash e_1 \; e_2} \quad \frac{G \cup \{x\} \vdash e}{G \vdash \text{fun } x \to e}
\end{align*}
\]
Variable Scoping

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

• 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?

• Example: The following program is syntactically correct but not well-formed. (y and q are used without being defined anywhere)

```java
int fact(int x) {
    var acc = 1;
    while (x > 0) {
        acc = acc * y;
        x = q - 1;
    }
    return acc;
}
```

Q: Can we solve this problem by changing the parser to rule out such programs?
• Need to keep track of contextual information.
  – What variables are in scope?
  – What are their types?

• How do we describe this?
  – In the compiler there's a mapping from variables to information we know about them.
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 \(( G;L \vdash e : t )\) by searching backwards through the rules.

• Compiling in a context is nothing more than a collection of inference rules specifying yet a different judgment \(( G \vdash src \Rightarrow target )\)
  – Moreover, the compilation judgment is similar to the typechecking judgment

• 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!
Inference Rules

• We can read a judgment $G;L \vdash e : t$ as “the expression $e$ is well typed and has type $t$”

• For any environment $G$, expression $e$, and statements $s_1$, $s_2$.

\[ G;L;rt \vdash \text{if } (e) \text{ s}_1 \text{ else } s_2 \]

holds if $G;L \vdash e : \text{bool}$ and $G;L;rt \vdash s_1$ and $G;L;rt \vdash s_2$ all hold.

• More succinctly: we summarize these constraints as an inference rule:

<table>
<thead>
<tr>
<th>Premises</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G;L \vdash e : \text{bool}$</td>
<td>$G;L;rt \vdash \text{if } (e) s_1 \text{ else } s_2$</td>
</tr>
<tr>
<td>$G;L;rt \vdash s_1$</td>
<td>$G;L;rt \vdash s_2$</td>
</tr>
</tbody>
</table>

• This rule can be used for any substitution of the syntactic metavariables $G$, $e$, $s_1$ and $s_2$. 
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 type checker: verify that such a tree exists.

Example1: Find a tree for the following program using the inference rules in oat0-defn.pdf:

```
var x1 = 0;
var x2 = x1 + x1;
x1 = x1 - x2;
return(x1);
```

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

```
var x2 = x1 + x1;
return(x2);
```
Example Derivation

```plaintext
var x1 = 0;
var x2 = x1 + x1;
x1 = x1 - x2;
return(x1);
```

\[ D_1 \quad D_2 \quad D_3 \quad D_4 \]

\[ G_0; \cdot; \text{int} \vdash \var x_1 = 0; \var x_2 = x_1 + x_1; x_1 = x_1 - x_2; \text{return } x_1; \Rightarrow \cdot, x_1: \text{int}, x_2: \text{int} \]

\[ \vdash \var x_1 = 0; \var x_2 = x_1 + x_1; x_1 = x_1 - x_2; \text{return } x_1; \]

[STMTS]

[PROG]
Example Derivation

\[
\begin{align*}
D_1 &= 
\frac{\quad G_0; \cdot \vdash 0: \text{int} \quad [\text{INT}]}{G_0; \cdot \vdash 0: \text{int} \quad [\text{CONST}]}
\hline
\frac{\quad G_0; \cdot \vdash \text{var } x_1 = 0 \Rightarrow \cdot, x_1: \text{int} \quad [\text{DECL}]}{G_0; \cdot, \text{int} \vdash \text{var } x_1 = 0; \Rightarrow \cdot, x_1: \text{int} \quad [\text{SDECL}]} \\
\end{align*}
\]

\[
\begin{align*}
\frac{\quad \vdash +: (\text{int}, \text{int}) \rightarrow \text{int} \quad [\text{ADD}]}{\frac{x_1: \text{int} \in \cdot, x_1: \text{int} \quad [\text{VAR}]}{G_0; \cdot, x_1: \text{int} \vdash x_1: \text{int} \quad [\text{BOP}]}}
\hline
\frac{\quad G_0; \cdot, x_1: \text{int} \vdash x_1 + x_1: \text{int}}{G_0; \cdot, x_1: \text{int}; \text{int} \vdash \text{var } x_2 = x_1 + x_1; \Rightarrow \cdot, x_1: \text{int}, x_2: \text{int} \quad [\text{DECL}]}
\hline
\frac{\quad G_0; \cdot, x_1: \text{int}; \text{int} \vdash \text{var } x_2 = x_1 + x_1; \Rightarrow \cdot, x_1: \text{int}, x_2: \text{int}}{G_0; \cdot, x_1: \text{int}; \text{int} \vdash \text{var } x_2 = x_1 + x_1; \Rightarrow \cdot, x_1: \text{int}, x_2: \text{int} \quad [\text{SDECL}]} \\
\end{align*}
\]
Example Derivation

\[ D_3 \]

\[
x_1: \text{int} \in \cdot, x_1: \text{int}, x_2: \text{int} ;
\]

\[ \vdash -(\text{int}, \text{int}) \to \text{int} \quad \text{[ADD]} \quad \frac{x_1: \text{int}, x_1: \text{int}, x_2: \text{int}}{G_0; :: x_1: \text{int}, x_2: \text{int} \vdash x_1: \text{int}} \quad \text{[VAR]} \quad \frac{x_2: \text{int} \in \cdot, x_1: \text{int}, x_2: \text{int}}{G_0; :: x_1: \text{int}, x_2: \text{int} \vdash x_2: \text{int}} \quad \text{[VAR]} \quad \frac{G_0; :: x_1: \text{int}, x_2: \text{int} \vdash x_1 - x_2: \text{int}}{G_0; :: x_1: \text{int}, x_2: \text{int}; \text{int} \vdash x_1 = x_1 - x_2; \Rightarrow •, x_1: \text{int}, x_2: \text{int}} \quad \text{[ASSN]}
\]

\[ D_4 \]

\[
\frac{x_1: \text{int} \in \cdot, x_1: \text{int}, x_2: \text{int}}{G_0; :: x_1: \text{int}, x_2: \text{int} \vdash x_1: \text{int}} \quad \text{[VAR]}
\]

\[
D_4 = \frac{x_1: \text{int} \in \cdot, x_1: \text{int}, x_2: \text{int}}{G_0; :: x_1: \text{int}, x_2: \text{int}; \text{int} \vdash \text{return } x_1; \Rightarrow •, x_1: \text{int}, x_2: \text{int}} \quad \text{[RET]}
\]
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

• Compiling in a context is nothing more an “interpretation” of the inference rules that specify typechecking*: \[ C \vdash e : t \]
  – Compilation follows the typechecking judgment

• 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!

*Here (and later) we’ll write context C for \( G; L \), the combination of the global and local contexts.
Compilation As Translating Judgments

- Consider the source typing judgment for source expressions:

  \[ C \vdash e : t \]

- How do we interpret this information in the target language?

  \[ \llbracket C \vdash e : t \rrbracket = ? \]

- \( \llbracket t \rrbracket \) is a target type
- \( \llbracket e \rrbracket \) translates to a (potentially empty) sequence of instructions, that, when run, computes the result into some operand

- INVARIANT: if \( \llbracket C \vdash e : t \rrbracket = ty \), operand, stream
  then the type (at the target level) of the operand is \( ty = \llbracket t \rrbracket \)
Example

- \( C \vdash 341 + 5 : \text{int} \)  
  
  What is \( \llbracket C \vdash 341 + 5 : \text{int} \rrbracket \) ?

\[
\begin{align*}
\llbracket C \vdash 341 : \text{int} \rrbracket &= (i64, \text{Const 341}, []) \\
\llbracket C \vdash 5 : \text{int} \rrbracket &= (i64, \text{Const 5}, [])
\end{align*}
\]

\[
\begin{align*}
\llbracket C \vdash 341 + 5 : \text{int} \rrbracket &= (i64, \%\text{tmp}, [\%\text{tmp} = \text{add i64 (\text{Const 341}) (\text{Const 5})}])
\end{align*}
\]
What about the Context?

• What is $[C]$?
• Source level C has bindings like: $x$:int, $y$:bool
  – We think of it as a finite map from identifiers to types

• What is the interpretation of C at the target level?

• $[C]$ maps source identifiers, “$x$” to source types and $[x]$

• What is the interpretation of a variable $[x]$ at the target level?
  – How are the variables used in the type system?

\[
\begin{align*}
\frac{x : t \in L}{G; L \vdash x : t} & \quad \text{TYP\_VAR} \\
\frac{G ; L \vdash exp : t}{G ; L ; rt \vdash x = exp \Rightarrow L} & \quad \text{TYP\_ASSN}
\end{align*}
\]
Interpretation of Contexts

• $[C]$ = a map from source identifiers to types and target identifiers

• INVARINT:
  $x : t \in C$ means that
  
  (1) lookup $[C] x = (t, \%id_x)$
  
  (2) the (target) type of $\%id_x$ is $[t]^*$ (a pointer to $[t]$)
Interpretation of Variables

- Establish invariant for expressions:

\[
\begin{align*}
    x : t &\in L \\
    G; L \vdash x : t &\quad \text{TYP_VAR} \\
    \text{as expressions} & \\
    \text{(which denote values)}
\end{align*}
\]

\[
G; L \vdash \text{exp : t} \\
\text{as addresses} \\
\text{(which can be assigned)}
\]

\[
G; L; rt \vdash x = \text{exp} ; \Rightarrow L \\
\text{where (i64, %id_x) = lookup [L] x}
\]

- What about statements?

\[
\begin{align*}
    x : t &\in L \\
    G; L \vdash \text{exp : t} &\quad \text{TYP_ASSN} \\
    \text{as addresses} & \\
    \text{(which can be assigned)}
\end{align*}
\]

\[
G; L; rt \vdash x = \text{exp} ; \Rightarrow L \\
\text{where (t, %id_x) = lookup [L] x} \\
\text{and [G;L \vdash exp : t] = ([t], opn, stream)}
\]
Other Judgments?

- **Statement:**
  \[
  \llbracket C; rt \vdash stmt \Rightarrow C' \rrbracket = \llbracket C' \rrbracket, \text{stream}
  \]

- **Declaration:**
  \[
  \llbracket G;L \vdash t \ x = \ exp \Rightarrow G;L,x:t \rrbracket = \llbracket G;L,x:t \rrbracket, \text{stream}
  \]

  **INVARIANT:** stream is of the form:
  \[
  \text{stream'} @
  \]
  \[
  [ \ %id_x = \text{alloca} [t];
  \quad \text{store} [t] \ opn, [t]^* \ %id_x ]
  \]

  and \[
  \llbracket G;L \vdash \text{exp} : t \rrbracket = (\llbracket t \rrbracket, \ opn, \text{stream'})
  \]

- **Rest follow similarly**
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.

\[
\llbracket C; \text{rt} \vdash \text{while}(e) \ s \Rightarrow C' \rrbracket = \llbracket C' \rrbracket,
\]

```c
lpre:
  \text{opn} = \llbracket C \vdash e : \text{bool} \rrbracket
  %\text{test} = \text{icmp eq i1 \ opn, 0}
  \text{br} %\text{test, label} %\text{lpost, label} %\text{lbody}

lbody:
  \llbracket C; \text{rt} \vdash s \Rightarrow C' \rrbracket
  \text{br} %\text{lpre}

lpost:
```

• Note: writing \text{opn} = \llbracket C \vdash e : \text{bool} \rrbracket \text{ is pun}
  – translating \llbracket C \vdash e : \text{bool} \rrbracket \text{ generates code that puts the result into opn}
  – In this notation there is implicit collection of the code
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.

\[
[C;rt \vdash \text{if } (e_1) \ s_1 \ \text{else } s_2 \Rightarrow C'] = [C']
\]

```plaintext
opn = [C \vdash e : \text{bool}]
%test = icmp eq i1 opn, 0
br %test, label %else, label %then
then:
   [C;rt \vdash s_1 \Rightarrow C']
   br %merge
else:
   [C; rt s_2 \Rightarrow C']
   br %merge
merge:
```
Connecting this to Code

- Instruction streams:
  - Must include labels, terminators, and “hoisted” global constants

- Must post-process the stream into a control-flow-graph

- See frontend.ml from HW4
OPTIMIZING CONTROL
Standard Evaluation

• Consider compiling the following program fragment:

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

Translated to intermediate representation:

```asm
%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
  – \([\mathcal{C} \vdash e : t]\) = (ty, operand, stream)
  – e.g. \([\mathcal{C} \vdash e_1 + e_2 : \text{int}]\) = (i64, %tmp, \([%\text{tmp} = \text{add} \ [e_1] [e_2]])\)

• 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 us implement different functionality too: e.g. short-circuiting boolean expressions
Idea: Use a different translation for tests

Usual Expression translation:
\[
\llbracket C \vdash e : t \rrbracket = (ty, operand, stream)
\]

Conditional branch translation of booleans, without materializing the value:
\[
\llbracket C \vdash e : \text{bool}@ \rrbracket \ ltrue \ lfalse = \text{stream}
\]

\[
\llbracket C, \text{rt} \vdash \text{if (e) then s1 else s2 } \Rightarrow C' \rrbracket = \llbracket C' \rrbracket,
\]

Notes:

- 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…

\[
\text{insns}_3
\]

\text{then:}
\[
\llbracket s1 \rrbracket\text{ br } \%\text{merge}
\]

\text{else:}
\[
\llbracket s2 \rrbracket\text{ br } \%\text{merge}
\]

\text{merge:}

where
\[
\llbracket C, \text{rt} \vdash s_1 \Rightarrow C' \rrbracket = \llbracket C' \rrbracket, \text{insns}_1
\]
\[
\llbracket C, \text{rt} \vdash s_2 \Rightarrow C'' \rrbracket = \llbracket C'' \rrbracket, \text{insns}_2
\]
\[
\llbracket C \vdash e : \text{bool}@ \rrbracket \text{ then else } = \text{insns}_3
\]
Short Circuit Compilation: Expressions

- $[[C \vdash e : \text{bool}]] \text{ltrue } \text{lfalse} = \text{insns}$

\[ [[C \vdash \text{false} : \text{bool}]] \text{ltrue } \text{lfalse} = [\text{br } \%\text{lfalse}] \]

\[ [[C \vdash \text{true} : \text{bool}]] \text{ltrue } \text{lfalse} = [\text{br } \%\text{ltrue}] \]

- $[[C \vdash \text{e} : \text{bool}]] \text{lfalse } \text{ltrue} = \text{insns}$

- $[[C \vdash \text{!e} : \text{bool}]] \text{ltrue } \text{lfalse} = \text{insns}$
Short Circuit Evaluation

Idea: build the logic into the translation

\[
\begin{align*}
&[[C \vdash e_1 : \text{bool}]] \ ltrue \ right = \ insns_1 \\
&[[C \vdash e_2 : \text{bool}]] \ ltrue \ lfalse = \ insns_2 \\
&[[C \vdash e_1 \mid e_2 : \text{bool}]] \ ltrue \ lfalse = \ insns_1 \\
&\quad \text{right:} \\
&\quad \quad \text{insn}_2 \\
&[[C \vdash e_1 \& e_2 : \text{bool}]] \ ltrue \ lfalse = \ insns_1 \\
&\quad \text{right:} \\
&\quad \quad \text{insn}_2 \\
\end{align*}
\]

where \text{right} is a fresh label
Short-Circuit Evaluation

- Consider compiling the following program fragment:

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

```assembly
%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
```
Compiling lambda calculus to straight-line code. Representing evaluation environments at runtime.
To implement first-class functions on a processor, there are two problems:

- First: we must implement substitution of free variables
- Second: we must separate ‘code’ from ‘data’

**Reify the substitution:**

- Move substitution from the meta language to the object language by making the data structure & lookup operation explicit
- The environment-based interpreter is one step in this direction

**Closure Conversion:**

- Eliminates free variables by packaging up the needed environment in the data structure.

**Hoisting:**

- Separates code from data, pulling closed code to the top level.
Example of closure creation

• Recall the “add” function:
  ```plaintext
  let add = fun x -> fun y -> x + y
  ```

• Consider the inner function: `fun y -> x + y`

• When run the function application: `add 4`
  the program builds a closure and returns it.
  – The closure is a pair of the environment and a code pointer.

  The code pointer takes a pair of parameters: `env` and `y`
  – The function code is (essentially):
    ```plaintext
    fun (env, y) -> let x = nth env 0 in x + y
    ```
Representing Closures

• As we saw, the simple closure conversion algorithm doesn’t generate very efficient code.
  – It stores all the values for variables in the environment, even if they aren’t needed by the function body.
  – It copies the environment values each time a nested closure is created.
  – It uses a linked-list datastructure for tuples.

• There are many options:
  – Store only the values for free variables in the body of the closure.
  – Share subcomponents of the environment to avoid copying
  – Use vectors or arrays rather than linked structures
Array-based Closures with N-ary Functions

\[
(\text{fun } (x \, y \, z) \rightarrow \\
(\text{fun } (n \, m) \rightarrow (\text{fun } p \rightarrow (\text{fun } q \rightarrow n + z) \, x))
\]

Note how free variables are “addressed” relative to the closure due to shared env.

“follow 1 nxt ptr then look up index 0”

“follow 2 nxt ptrs then look up index 2”
Adding Integers to Lambda Calculus

exp ::= 
| ... 
| n constant integers
| exp₁ + exp₂ binary arithmetic operation
val ::= 
| fun x -> exp functions are values
| n integers are values
n{v/x} = n constants have no free vars.
(e₁ + e₂){v/x} = (e₁{v/x} + e₂{v/x}) substitute everywhere

exp₁ \downarrow n₁  \quad exp₂ \downarrow n₂

\hline
exp₁ + exp₂ \downarrow (n₁ \llbracket + \rrbracket n₂)

Object-level ‘+’ \quad Meta-level ‘+’