Lecture 15

CIS 341: COMPILERS

Announcements

- HW4: OAT v. 1.0
  - Parsing & basic code generation
  - **Due: Wednesday, March 23rd**

UNTYPED LAMBDA CALCULUS

(Untyped) Lambda Calculus

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

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*

Operational Semantics

- Key operation: *capture-avoiding substitution*: $e_2\{e_1/x\}$
  - replaces all free occurrences of x in e_2 by e_1
  - must respect scope and alpha equivalence (renaming)

- *Reduction Strategies*
  Various ways of simplifying (or "*reducing*") lambda calculus terms.
  - call-by-value evaluation:
    - simplify the function argument *before* substitution
    - *does not* reduce under lambda (a.k.a. fun)
  - call-by-name evaluation:
    - *does not* simplify the argument before substitution
    - *does not* reduce under lambda
  - weak-head normalization:
    - does not simplify the argument before substitution
    - does not reduce under lambda
    - works on open terms, "suspending" reduction at variables
  - normal order reduction:
    - *does* reduce under lambda
    - first does weak-head normalization and then recursively continues to reduce
    - works on open terms – guaranteed to find a "normal form" if such a form exists

A "normal form" is one that has no substitution steps possible, i.e., there are no subterms of the form
(fun x → e1) e2
anywhere.

CBV Operational Semantics

- This is *call-by-value* semantics:
  function arguments are evaluated before substitution

$$\frac{}{v \Downarrow v}$$

"Values evaluate to themselves"

$$\frac{exp_1 \Downarrow (\texttt{fun}\ x \rightarrow exp_3) \qquad exp_2 \Downarrow v \qquad exp_3\{v/x\} \Downarrow w}{exp_1\ exp_2 \Downarrow w}$$

"To evaluate function application: Evaluate the function to a value, evaluate the argument to a value, and then substitute the argument for the function. "

CBN Operational Semantics

- This is *call-by-name* semantics:
  function arguments are evaluated before substitution

$$\frac{}{v \Downarrow v}$$

"Values evaluate to themselves"

$$\frac{exp_1 \Downarrow (\texttt{fun}\ x \rightarrow exp_3) \qquad exp_3\{exp_2/x\} \Downarrow w}{exp_1\ exp_2 \Downarrow w}$$

"To evaluate function application: Evaluate the function to a value, substitute the argument into the function body, and then keep evaluating. "

See fun.ml

Examples of encoding Booleans, integers, conditionals, loops, etc., in untyped lambda calculus.

IMPLEMENTING THE INTERPRETER

Adding Integers to Lambda Calculus

exp ::=	
| …	
| n	*constant integers*
| exp_1 + exp_2	*binary arithmetic operation*
val ::=	
| `fun` x → exp	*functions are values*
| n	*integers are values*
n{v/x} = n	*constants have no free vars.*
$(e_1 + e_2)\{v/x\} = (e_1\{v/x\} + e_2\{v/x\})$	*substitute everywhere*

$$\frac{exp_1 \Downarrow n_1 \quad exp2 \Downarrow n_2}{exp_1 + exp_2 \Downarrow (n1 \; [\![+]\!] \; n2)}$$

Object-level '+'

Meta-level '+'

Scope, Types, and Context

STATIC ANALYSIS

Scope-Checking Lambda Calculus

- 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" $fv(e) \subseteq G$

$fv(x)$	=	$\{x\}$
$fv(\texttt{fun}\ x \rightarrow exp)$	=	$fv(exp) \setminus \{x\}$ *('x' is a bound in exp)*
$fv(exp_1\ exp_2)$	=	$fv(exp_1) \cup fv(exp_2)$

$$\frac{x \in G}{G \vdash x}$$

"the variable x is free"

$$\frac{G \vdash e_1 \qquad G \vdash e_2}{G \vdash e_1\ e_2}$$

"G contains the free variables of e_1 and e_2"

$$\frac{G \cup \{x\} \vdash e}{G \vdash \texttt{fun}\ x \rightarrow e}$$

"x is available in the function bodye"

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
```

$$\frac{x \in G}{G \vdash x} \qquad \frac{G \vdash e_1 \qquad G \vdash e_2}{G \vdash e_1\ e_2} \qquad \frac{G \cup \{x\} \vdash e}{G \vdash \texttt{fun}\ x \rightarrow e}$$

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)

```
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?

Contexts and Inference Rules

- Need to keep track of contextual information.
  - What variables are in scope?
  - What are their types?

- How do we describe this process?
  - In the compiler there's a mapping from variables to information we know about them.
  - This is ″contextual information″

- How do we use that information to implement a scope checker?

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 ⊢ 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 ⊢ src ⇒ 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` ⊢ 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.

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

holds if `G` ;L⊢ e : bool and `G;L;rt` ⊢ s_1 and `G;L;rt` ⊢ s_2 all hold.

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

Premises

Conclusion

$$\frac{\texttt{G;L} \vdash e : \text{bool} \qquad \texttt{G;L;rt} \vdash s_1 \qquad \texttt{G;L;rt} \vdash s_2}{\texttt{G;L;rt} \vdash \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2}$$

- This rule can be used for *any* substitution of the syntactic metavariables `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 type checker: verify that such a tree exists.
- Example1: Find a tree for the following program using the inference rules in oat-v1-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

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

$$\dfrac{\dfrac{\mathcal{D}_1 \quad \mathcal{D}_2 \quad \mathcal{D}_3 \quad \mathcal{D}_4}{G_0\,;\cdot\,;\texttt{int} \vdash \texttt{var}\ x_1 = 0; \texttt{var}\ x_2 = x_1 + x_1; x_1 = x_1 - x_2; \texttt{return}\ x_1; \Rightarrow \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}\ [\textsc{stmts}]}{\vdash \texttt{var}\ x_1 = 0; \texttt{var}\ x_2 = x_1 + x_1; x_1 = x_1 - x_2; \texttt{return}\ x_1;}\ [\textsc{prog}]$$

Example Derivation

$$\mathcal{D}_1 \quad = \quad \dfrac{\dfrac{\dfrac{\dfrac{}{G_0;\cdot \vdash 0 : \texttt{int}}\,[\textsc{int}]}{G_0;\cdot \vdash 0 : \texttt{int}}\,[\textsc{const}]}{G_0;\cdot \vdash \texttt{var}\ x_1 = 0 \Rightarrow \cdot, x_1 : \texttt{int}}\,[\textsc{decl}]}{G_0;\cdot;\texttt{int} \vdash \texttt{var}\ x_1 = 0; \Rightarrow \cdot, x_1 : \texttt{int}}\,[\textsc{sdecl}]$$

$$\mathcal{D}_2 \quad = \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\vdash + : (\texttt{int},\texttt{int}) \to \texttt{int}}\,[\textsc{add}] \quad \dfrac{x_1 : \texttt{int} \in \cdot, x_1 : \texttt{int}}{G_0;\cdot, x_1 : \texttt{int} \vdash x_1 : \texttt{int}}\,[\textsc{var}] \quad \dfrac{x_1 : \texttt{int} \in \cdot, x_1 : \texttt{int}}{G_0;\cdot, x_1 : \texttt{int} \vdash x_1 : \texttt{int}}\,[\textsc{var}]}{G_0;\cdot, x_1 : \texttt{int} \vdash x_1 + x_1 : \texttt{int}}\,[\textsc{bop}]}{G_0;\cdot, x_1 : \texttt{int};\texttt{int} \vdash \texttt{var}\ x_2 = x_1 + x_1; \Rightarrow \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}\,[\textsc{decl}]}{G_0;\cdot, x_1 : \texttt{int};\texttt{int} \vdash \texttt{var}\ x_2 = x_1 + x_1; \Rightarrow \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}\,[\textsc{sdecl}]$$

Example Derivation

$$\mathcal{D}_3 \quad \dfrac{\dfrac{}{\vdash \texttt{-} : (\texttt{int},\texttt{int}) \to \texttt{int}}\ [\textsc{add}] \quad \dfrac{x_1 : \texttt{int} \in \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}{G_0\,;\cdot, x_1 : \texttt{int}, x_2 : \texttt{int} \vdash x_1 : \texttt{int}}\ [\textsc{var}] \quad \dfrac{x_2 : \texttt{int} \in \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}{G_0\,;\cdot, x_1 : \texttt{int}, x_2 : \texttt{int} \vdash x_2 : \texttt{int}}\ [\textsc{var}]}{\dfrac{G_0\,;\cdot, x_1 : \texttt{int}, x_2 : \texttt{int} \vdash x_1 \texttt{ - } x_2 : \texttt{int}}{G_0\,;\cdot, x_1 : \texttt{int}, x_2 : \texttt{int}\,;\texttt{int} \vdash x_1 \texttt{ = } x_1 \texttt{ - } x_2\texttt{;} \Rightarrow \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}\ [\textsc{assn}]}\ [\textsc{bop}]$$

$$\mathcal{D}_4 \quad = \quad \dfrac{\dfrac{x_1 : \texttt{int} \in \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}{G_0\,;\cdot, x_1 : \texttt{int}, x_2 : \texttt{int} \vdash x_1 : \texttt{int}}\ [\textsc{var}]}{G_0\,;\cdot, x_1 : \texttt{int}, x_2 : \texttt{int}\,;\texttt{int} \vdash \texttt{return}\ x_1\texttt{;} \Rightarrow \cdot, x_1 : \texttt{int}, x_2 : \texttt{int}}\ [\textsc{Ret}]$$

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?

$$[\![C \vdash e : t]\!] = \ ?$$

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

- INVARIANT: if $[\![C \vdash e : t]\!] = ty, operand, stream$
  then the type (at the target level) of the operand is $ty = [\![t]\!]$

Example

- C ⊢ 341 + 5 : int what is ⟦ C ⊢ 341 + 5 : int⟧ ?

⟦ ⊢ 341 : int ⟧ = `(i64, Const 341, [])` ⟦⊢ 5 : int⟧ = `(i64, Const 5, [])`

⟦C ⊢ 341 : int⟧ = `(i64, Const 341, [])` ⟦C ⊢ 5 : int⟧ = `(i64, Const 5, [])`

⟦C ⊢ 341 + 5 : int⟧ = `(i64, %tmp, [%tmp = add i64 (Const 341) (Const 5)])`

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?

$$\frac{x : t \in L}{G ; L \vdash x : t}\ \textbf{TYP\_VAR}$$

as expressions
(which denote values)

$$\frac{x : t \in L \quad G ; L \vdash exp : t}{G ; L ; rt \vdash x = exp ; \Rightarrow L}\ \textbf{TYP\_ASSN}$$

as addresses
(which can be assigned)

Interpretation of Contexts

- ⟦C⟧ = a map from source identifiers to types and target identifiers

- INVARIANT:
  x:t ∈ 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:

$$\left[\!\!\left[\frac{x : t \in L}{G;L \vdash x : t}\ \textbf{TYP\_VAR} \right]\!\!\right] = \texttt{(\%tmp, [\%tmp = load i64* \%id\_x])}$$

as expressions
(which denote values)

where (`i64`, `%id_x`) = lookup ⟦L⟧ x

- What about statements?

$$\left[\!\!\left[\frac{x : t \in L \quad G;L \vdash exp : t}{G;L;rt \vdash x = exp; \Rightarrow L}\ \textbf{TYP\_ASSN} \right]\!\!\right] = \text{stream @ } \texttt{[store ⟦t⟧ opn, ⟦t⟧* \%id\_x]}$$

as addresses
(which can be assigned)

where (t, `%id_x`) = lookup ⟦L⟧ x
and ⟦G;L ⊢ exp : t⟧ = (⟦t⟧, `opn`, stream)

Other Judgments?

- Statement:
  ⟦C; rt ⊢ stmt ⇒ C′⟧ = ⟦C′⟧ , stream

- Declaration:
  ⟦G;L ⊢ t x = exp ⇒ G;L,x:t ⟧ = ⟦G;L,x:t⟧, stream

  INVARIANT: stream is of the form:

```
stream' @
[ %id_x = alloca ⟦t⟧;
  store ⟦t⟧ opn, ⟦t⟧* %id_x ]
```

  and ⟦G;L ⊢ exp : t ⟧ = (⟦t⟧, `opn`, 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.

⟦`C;rt ⊢ while(e) s ⇒ C'`⟧ = ⟦`C'`⟧,

```
lpre:
    opn = ⟦C ⊢ e : bool⟧
    %test = icmp eq i1 opn, 0
    br %test, label %lpost, label %lbody
lbody:
     ⟦C;rt ⊢ s ⇒ C'⟧
     br %lpre
lpost:
```

- Note: writing `opn =` ⟦`C ⊢ e : bool`⟧ is pun
  - translating ⟦C ⊢ e : bool⟧ 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.

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

```
    opn = ⟦C ⊢ e : bool⟧
    %test = icmp eq i1 opn, 0
    br %test, label %else, label %then
then:
     ⟦C;rt ⊢ s1 ⇒ C'⟧
     br %merge
else:
    ⟦C; rt s2 ⇒ 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:

```
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 ⊢ e : t⟧ = (ty, operand, stream)
  - e.g. ⟦C ⊢ e_1 + e_2 : int⟧ = (i64, `%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 us implement different functionality too: e.g. short-circuiting boolean expressions

Idea: Use a different translation for tests

Usual Expression translation:

$[\![C \vdash e : t]\!] = (ty, operand, stream)$

Conditional branch translation of booleans, without materializing the value:

$[\![C \vdash e : bool@]\!]\ ltrue\ lfalse = stream$

$[\![C, rt \vdash \texttt{if (e) then s1 else s2} \Rightarrow C']\!] = [\![C']\!],$

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

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…

where

$[\![C, rt \vdash s_1 \Rightarrow C']\!] = [\![C']\!], insns_1$

$[\![C, rt \vdash s_2 \Rightarrow C'']\!] = [\![C'']\!], insns_2$

$[\![C \vdash e : bool@\ ]\!]\ \texttt{then else} = insns_3$

Short Circuit Compilation: Expressions

- ⟦C ⊢ e : bool@⟧ ltrue lfalse = insns

$$\frac{}{⟦C ⊢ \text{false} : \text{bool@}⟧\ \text{ltrue lfalse} = \texttt{[br \%lfalse]}}\ \text{FALSE}$$

$$\frac{}{⟦C ⊢ \text{true} : \text{bool@}⟧\ \text{ltrue lfalse} = \texttt{[br \%ltrue]}}\ \text{TRUE}$$

$$\frac{⟦C ⊢ e : \text{bool@}⟧\ \text{lfalse ltrue} = \text{insns}}{⟦C ⊢ !e : \text{bool@}⟧\ \text{ltrue lfalse} = \text{insns}}\ \text{NOT}$$

Short Circuit Evaluation

Idea: build the logic into the translation

$$\frac{[\![C \vdash e1 : bool@]\!]\ ltrue\ right = insns_1 \quad [\![C \vdash e2 : bool@]\!]\ ltrue\ lfalse = insns_2}{[\![C \vdash e1 \,|\, e2 : bool@]\!]\ ltrue\ lfalse =}$$

```
    insns1
right:
    insn2
```

$$\frac{[\![C \vdash e1 : bool@]\!]\ right\ lfalse = insns_1 \quad [\![C \vdash e2 : bool@]\!]\ ltrue\ lfalse = insns_2}{[\![C \vdash e1\&e2 : bool@]\!]\ ltrue\ lfalse =}$$

```
    insns1
right:
    insn2
```

where `right` is a fresh label

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
```