Lecture 16

CIS 341: COMPILERS

Announcements

- HW4: OAT v. 1.0
  - Parsing & basic code generation
  - **Due: Wednesday, March 28th**

- HW5: OAT v. 2.0
  - records, function pointers, type checking, array-bounds checks, etc.
  - Due: Wednesday, April 11th

Scope, Types, and Context

STATIC ANALYSIS

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 '+'

NOTE: there are no rules for the case where exp1 or exp2 evaluate to functions! The semantics is *undefined* in those cases.

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?

Type Checking / Static Analysis

- Recall the interpreter from the Eval3 module:

```
let rec eval env e =
  match e with
  | …
  | Add (e1, e2) ->
    (match (eval env e1, eval env e2) with
       | (IntV i1, IntV i2) -> IntV (i1 + i2)
       | _ -> failwith "tried to add non-integers")
  | …
```

- The interpreter might fail at runtime.
  - Not all operations are defined for all values (e.g. 3/0, 3 + true, …)
- A compiler can't generate sensible code for this case.
  - A naïve implementation might "add" an integer and a function pointer

See tc.ml

STATICALLY RULING OUT PARTIALITY: TYPE CHECKING

Notes about this Typechecker

- In the interpreter, we only evaluate the body of a function when it's applied.
- In the typechecker, we always check the body of the function (even if it's never applied.)
  - We *assume* the input has some type (say t_1) and reflect this in the type of the function (t_1 **->** t_2).

- Dually, at a call site (e_1 e_2), we don't know what *closure* we're going to get.
  - But we can calculate e_1's type, check that e_2 is an argument of the right type, and also determine what type e_1 will return.

- Question: Why is this an approximation?
- Question: What if `well_typed` always returns `false`?

Contexts and Inference Rules

- Need to keep track of contextual information.
  - What variables are in scope?
  - What are their types?
  - What information doe we have about each syntactic construct?

- What relationships are there among the syntactic objects?
  - e.g. is one type a subtype of another?

- How do we describe this information?
  - In the compiler there's a mapping from variables to information we know about them – the "context".
  - The compiler has a collection of (mutually recursive) functions that follow the structure of the syntax.

Type Judgments

- In the judgment: $E \vdash e : t$
  - E is a *typing environment* or a *type context*
  - E maps variables to types. It is just a set of bindings of the form:
    $x_1 : t_1, x_2 : t_2, \ldots, x_n : t_n$
- For example: $x : \text{int}, b : \text{bool} \vdash \text{if (b) 3 else } x : \text{int}$
- What do we need to know to decide whether "if (b) 3 else x" has type int in the environment x : int, b : bool?
  - b must be a bool i.e. $x : \text{int}, b : \text{bool} \vdash b : \text{bool}$
  - 3 must be an int i.e. $x : \text{int}, b : \text{bool} \vdash 3 : \text{int}$
  - x must be an int i.e. $x : \text{int}, b : \text{bool} \vdash x : \text{int}$

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 ($G \vdash src \Rightarrow target$)
  - Moreover, the compilation rules are very similar in structure to the typechecking rules
- Strong mathematical foundations
  - The "Curry-Howard correspondence": Programming Language ~ Logic, Program ~ Proof, Type ~ Proposition
  - See CIS 500 if you're interested in type systems!

Inference Rules

- We can read a judgment $\texttt{G;L} \vdash e : t$ as "the expression e is well typed and has type t"
- We can read a judgment $\texttt{G;L} \vdash s$ as "the statement s is well formed"
- For any environment G, expression e, and statements s_1, s_2.

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

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

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

Premises

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

Conclusion

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

Simply-typed Lambda Calculus

- For the language in "tc.ml" we have five inference rules:

INT

$$\frac{}{E \vdash i : \text{int}}$$

VAR

$$\frac{x : T \in E}{E \vdash x : T}$$

ADD

$$\frac{E \vdash e_1 : \text{int} \qquad E \vdash e_2 : \text{int}}{E \vdash e_1 + e_2 : \text{int}}$$

FUN

$$\frac{E, x : T \vdash e : S}{E \vdash \text{fun } (x{:}T) \text{ -> } e \ : T \text{ -> } S}$$

APP

$$\frac{E \vdash e_1 : T \text{ -> } S \qquad E \vdash e_2 : T}{E \vdash e_1\ e_2 : S}$$

- Note how these rules correspond to the code.

Type 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 typechecker: verify that such a tree exists.
- Example: Find a tree for the following program using the inference rules on the previous slide:

$$\vdash \text{(fun (x:int) -> x + 3) 5} \;:\; \text{int}$$

Example Derivation Tree

$$\text{APP}\;\dfrac{\text{FUN}\;\dfrac{\text{ADD}\;\dfrac{\text{VAR}\;\dfrac{x : \mathrm{int} \;\in\; x : \mathrm{int}}{x : \mathrm{int} \vdash x : \mathrm{int}} \qquad \text{INT}\;\dfrac{}{x : \mathrm{int} \vdash 3 : \mathrm{int}}}{x : \mathrm{int} \vdash x + 3 : \mathrm{int}}}{\vdash (\mathrm{fun}\ (x{:}\mathrm{int}) \ {\text{->}}\ x + 3) : \mathrm{int}\ {\text{->}}\ \mathrm{int}} \qquad \text{INT}\;\dfrac{}{\vdash 5 : \mathrm{int}}}{\vdash (\mathrm{fun}\ (x{:}\mathrm{int})\ {\text{->}}\ x + 3)\ 5 : \mathrm{int}}$$

- Note: the OCaml function `typecheck` verifies the existence of this tree. The structure of the recursive calls when running `typecheck` is the same shape as this tree!
- Note that $x : \mathrm{int} \in E$ is implemented by the function `lookup`

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.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);
```

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

Example Derivation

$$\mathcal{D}_1 \quad = \quad \dfrac{\dfrac{\dfrac{}{G_0 ; \cdot \vdash 0 : \mathtt{int}}[\text{INT}]}{G_0 ; \cdot \vdash 0 : \mathtt{int}}[\text{CONST}]}{\dfrac{G_0 ; \cdot \vdash \mathtt{var}\ x_1 = 0 \Rightarrow \cdot, x_1 : \mathtt{int}}{G_0 ; \cdot ; \mathtt{int} \vdash \mathtt{var}\ x_1 = 0; \Rightarrow \cdot, x_1 : \mathtt{int}}[\text{SDECL}]}[\text{DECL}]$$

$$\mathcal{D}_2 \quad = \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\vdash + : (\mathtt{int}, \mathtt{int}) \to \mathtt{int}}[\text{ADD}] \quad \dfrac{x_1 : \mathtt{int} \in \cdot, x_1 : \mathtt{int}}{G_0 ; \cdot, x_1 : \mathtt{int} \vdash x_1 : \mathtt{int}}[\text{VAR}] \quad \dfrac{x_1 : \mathtt{int} \in \cdot, x_1 : \mathtt{int}}{G_0 ; \cdot, x_1 : \mathtt{int} \vdash x_1 : \mathtt{int}}[\text{VAR}]}{G_0 ; \cdot, x_1 : \mathtt{int} \vdash x_1 + x_1 : \mathtt{int}}[\text{BOP}]}{G_0 ; \cdot, x_1 : \mathtt{int} ; \mathtt{int} \vdash \mathtt{var}\ x_2 = x_1 + x_1 ; \Rightarrow \cdot, x_1 : \mathtt{int}, x_2 : \mathtt{int}}[\text{DECL}]}{G_0 ; \cdot, x_1 : \mathtt{int} ; \mathtt{int} \vdash \mathtt{var}\ x_2 = x_1 + x_1 ; \Rightarrow \cdot, x_1 : \mathtt{int}, x_2 : \mathtt{int}}[\text{SDECL}]$$

Example Derivation

$$x_1 : \texttt{int} \in \cdot, x_1 : \texttt{int}, x_2 : \texttt{int} \;;$$

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

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

Type Safety

"Well typed programs do not go wrong."

– Robin Milner, 1978

Theorem: (simply typed lambda calculus with integers)

If $\vdash e : t$ then there exists a value v such that $e \Downarrow v$.

- Note: this is a *very* strong property.
  - Well-typed programs cannot "go wrong" by trying to execute undefined code (such as 3 + (fun x -> 2))
  - Simply-typed lambda calculus is guaranteed to terminate! (i.e. it isn't Turing complete)

Type Safety For General Languages

Theorem: (Type Safety)

If $\vdash P : t$ is a well-typed program, then either:
(a) the program terminates in a well-defined way, or
(b) the program continues computing forever

- Well-defined termination could include:
  - halting with a return value
  - raising an exception

- Type safety rules out undefined behaviors:
  - abusing "unsafe" casts: converting pointers to integers, etc.
  - treating non-code values as code (and vice-versa)
  - breaking the type abstractions of the language
- What is "defined" depends on the language semantics…

Beyond describing "structure"… describing "properties"

Types as sets

Subsumption

TYPES, MORE GENERALLY

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?

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

- $[\![C]\!]$ translates contexts
- $[\![t]\!]$ is a target type
- $[\![e]\!]$ translates to a (potentially empty) stream 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

- $C \vdash 341 + 5 : \text{int}$ what is $[\![C \vdash 341 + 5 : \text{int}]\!]$?

$[\![\vdash 341 : \text{int}]\!] =$ `(i64, Const 341, [])`
$\overline{\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}$
$[\![C \vdash 341 : \text{int}]\!] =$ `(i64, Const 341, [])`

$[\![\vdash 5 : \text{int}]\!] =$ `(i64, Const 5, [])`
$\overline{\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}$
$[\![C \vdash 5 : \text{int}]\!] =$ `(i64, Const  5, [])`

$\overline{\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}$

$[\![C \vdash 341 + 5 : \text{int}]\!] =$ `(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?

$$\left[\!\left[\frac{x:t \in L \quad G;L \vdash exp : t}{G;L;rt \vdash x = exp;\ \Rightarrow L}\ \textbf{TYP\_ASSN} \right]\!\right] = \text{stream @}$$

`[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:
  $[\![C;\ rt \vdash \text{stmt} \Rightarrow C']\!]\ =\ \ \ \ [\![C']\!]$, stream

- Declaration:
  $[\![G;L \vdash t\ x = \exp \Rightarrow G;L,x{:}t\ ]\!] = \ [\![G;L,x{:}t]\!]$, stream

  INVARIANT: stream is of the form:

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

  and $[\![G;L \vdash \exp : t\ ]\!] = ([\![t]\!],$ `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` $\vdash$ `if (`e_1`)` s_1 `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

Arrays

- Array constructs are not hard
- First: add a new type constructor: T[]

NEW

$$\frac{E \vdash e_1 : \text{int} \qquad E \vdash e_2 : T}{E \vdash \text{new } T[e_1](e_2) \ : T[]}$$

e_1 is the size of the newly allocated array. e_2 initializes the elements of the array.

INDEX

$$\frac{E \vdash e_1 : T[] \qquad E \vdash e_2 : \text{int}}{E \vdash e_1[e_2] \ : T}$$

UPDATE

$$\frac{E \vdash e_1 : T[] \qquad E \vdash e_2 : \text{int} \qquad E \vdash e_3 : T}{E \vdash e_1[e_2] = e_3 \text{ ok}}$$

Note: These rules don't ensure that the array index is in bounds – that should be checked *dynamically*.

Tuples

- ML-style tuples with statically known number of products:
- First: add a new type constructor: $T_1 * \ldots * T_n$

TUPLE

$$\frac{E \vdash e_1 : T_1 \quad \ldots \quad E \vdash e_n : T_n}{E \vdash (e_1, \ldots, e_n) : T_1 * \ldots * T_n}$$

PROJ

$$\frac{E \vdash e : T_1 * \ldots * T_n \quad 1 \leq i \leq n}{E \vdash \#i\ e \ :\ T_i}$$

References

- ML-style references (note that ML uses only expressions)
- First, add a new type constructor: T ref

REF

$$\frac{E \vdash e : T}{E \vdash \text{ref } e : T \text{ ref}}$$

DEREF

$$\frac{E \vdash e : T \text{ ref}}{E \vdash \,!e \;: T}$$

ASSIGN

$$\frac{E \vdash e_1 : T \text{ ref} \qquad E \vdash e_2 : T}{E \vdash e_1 := e_2 \;: \text{unit}}$$

Note the similarity with the rules for arrays…