Lecture 17

**CS 131: COMPILERS** 

#### **Announcements**

• Midterm: graded by November 28<sup>th</sup>

Scope, Types, and Context

#### **STATIC ANALYSIS**

## **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 wellformed. (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?

#### **Inference Rules**

- We can read a judgment G ⊢ e as "the expression e is well scoped and has free variables in G"
- For any environment G, expression e, and statements s<sub>1</sub>, s<sub>2</sub>.

$$G \vdash if (e) s_1 else s_2$$

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

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

Premises 
$$G \vdash e \qquad G \vdash s_1 \qquad G \vdash s_2$$

Conclusion  $G \vdash if (e) s_1 else s_2$ 

• Such a rule can be used for *any* substitution of the syntactic metavariables G, e,  $s_1$  and  $s_2$ .

## **Scope-Checking Lambda Calculus**

- Consider how to identify "well-scoped" lambda calculus terms
  - Given: G, a set of variable identifiers, e, a term of the lambda calculus
  - Judgment: G  $\vdash$  e "the free variables of e are included in G"

"the variable x is free, but in scope"

$$\frac{\mathsf{G} \vdash \mathsf{e}_1 \qquad \mathsf{G} \vdash \mathsf{e}_2}{\mathsf{G} \vdash \mathsf{e}_1 \, \mathsf{e}_2}$$

"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
```

$$x \in G$$
  $G \vdash e_1$   $G \vdash e_2$   $G \cup \{x\} \vdash e$   $G \vdash x$   $G \vdash e_1 e_2$   $G \vdash fun x \rightarrow e$  
$$G \vdash x \qquad APP \qquad FUN$$

- The inference rules are a *specification* of the intended behavior of this scope checking code.
  - they don't specify the order in which the premises are checked

## **Judgments**

- A judgment is a (meta-syntactic) notation that names a relation among one or more sets.
  - The sets are usually built from object-language syntax elements and other "math" sets (e.g., integers, natural numbers, etc.)
  - We usually describe them using metavariables that range over the sets.
  - Often use domain-specific notation to ease reading.
  - The meaning of judgments, i.e., which sets they represent, is defined by (collections of) inference rules
- Example: When we say "G ⊢ e is a judgment where G is a context of variables and e is a term, defined by these [...] inference rules" that is shorthand for this "math speak":
  - Let Var be the set of all (syntactic) variables
  - Let Exp be the set {e | e is a term of the untyped lambda calculus}
  - Let P(Var) be the (finite) powerset of variables (set of all finite sets)
  - Define well-scoped ⊆ (P(Var), Exp) to be a relation satisfying the properties defined by the associated inference rules [...]
  - Then "G ⊢ e" is notation that means that (G, e) ∈ well-scoped

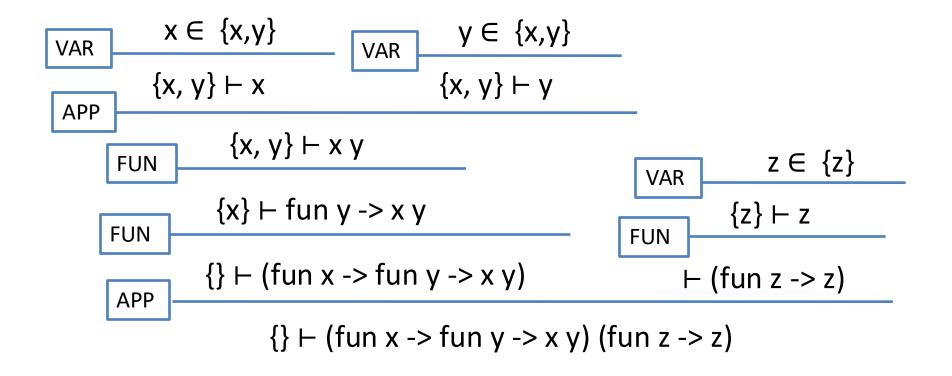
## **Checking Derivations**

- 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
  - axiom: rule with no premises that are judgments
  - Example: the VAR rule is an axiom (it doesn't have any ⊢
- Goal of the static checking algorithm: verify that such a tree exists.

Example: we can scope check the following lambda calculus term by finding a derivation tree for it:

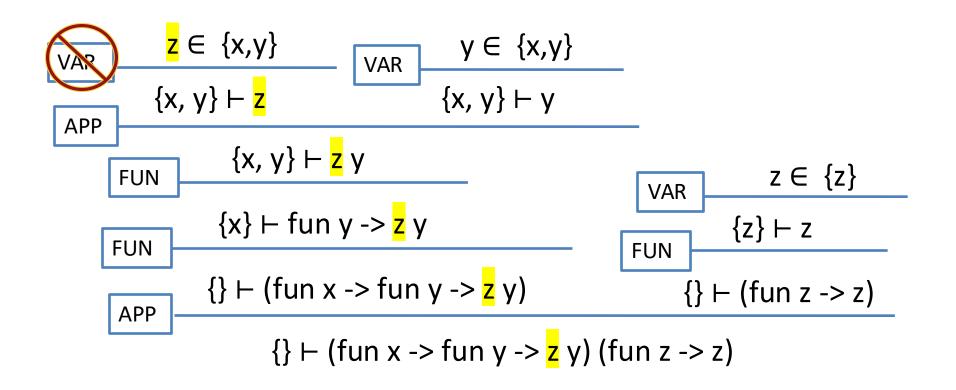
(fun  $x \rightarrow fun y \rightarrow x y$ ) (fun  $z \rightarrow z$ )

#### **Example Derivation Tree**



- Note: the OCaml function scope\_check verifies the existence of this tree.
   The structure of the recursive calls when running scope\_check is the same shape as this tree!
- Note that  $x \in E$  is implemented by the function VarSet.mem

#### **Example Failed Derivation**



- This program is not well scoped
  - The variable z is not bound in the body of the left function.
  - The typing derivation fails because the VAR rule cannot succeed
  - (The other parts of the derivation are OK, though!)

#### Uses of the inference rules

- We can do proofs by induction on the structure of the derivation.
- For example:

**Lemma:** If  $G \vdash e$  then  $fv(e) \subseteq G$ .

Proof.

By induction on the derivation that  $G \vdash e$ .

x ∈ G G ⊢ x

- case: VAR then we have e = x (for some variable x) and x ∈ G. But  $fv(e) = fv(x) = \{x\}$ , but then  $\{x\} \subseteq G$ .
- case: APP then we have  $e = e_1 e_2$  (for some  $e_1 e_2$ ) and, by induction, we have  $fv(e_1) \subseteq G$  and  $fv(e_2) \subseteq G$ , so  $fv(e_1 e_2) = fv(e_1) \cup fv(e_2) \subseteq G$
- $\frac{G \vdash e_1 \quad G \vdash e_2}{G \vdash e_1 e_2}$ 
  - $G \cup \{x\} \vdash e_1$
  - $G \vdash \text{fun } x \rightarrow e_1$
- case: FUN then we have  $e = (fun x -> e_1)$  for some x,  $e_1$  and, by induction, we have  $fv(e_1) \subseteq G \cup \{x\}$ , but then we also have  $fv(fun x -> e_1) = fv(e_1) \setminus \{x\} \subseteq ((G \cup \{x\}) \setminus \{x\}) \subseteq G$
- fv(x) =  $\{x\}$ fv(fun x  $\rightarrow$  exp) = fv(exp) \  $\{x\}$  ('x' is a bound in exp) fv(exp<sub>1</sub> exp<sub>2</sub>) = fv(exp<sub>1</sub>) U fv(exp<sub>2</sub>)

See tc.ml

# STATICALLY RULING OUT PARTIALITY: TYPE CHECKING

#### **Adding Integers to Lambda Calculus**

$$\exp_1 \Downarrow n_1 \exp_2 \Downarrow n_2$$

$$\exp_1 + \exp_2 \Downarrow (n1 [+] n2)$$
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.

# **Type Checking / Static Analysis**

Recall the interpreter from the Eval3 module:

- 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

## **Type Judgments**

- In the judgment: E ⊢ 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, ..., x_n : t_n
```

- For example: x : int, b : bool ⊢ if (b) 3 else x : 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 : int, b : bool \vdash b : bool
```

- 3 must be an int i.e. x : int, b : bool 3 : int
- x must be an int i.e.  $x : int, b : bool \vdash x : int$

## Simply-typed Lambda Calculus

- Consider how to identify "well-scoped" lambda calculus terms
  - Recall the free variable calculation
  - Given: G, a map of variable identifiers to types, e, a term of the lambda calculus
  - Judgment: G ⊢ e : T means "the expression e computes a value of type T, assuming its free variables have the types given in G"

$$x:T \in G$$
 "the variable x has type T an is in scope"  $G \vdash x : T$ 

$$G \vdash e_1 : T \rightarrow S$$
  $G \vdash e_2 : T$   
 $G \vdash e_1 e_2 : S$ 

" $e_1$  is a function from T2 to T and  $e_2$  is an expression of type T2"

G, x : T 
$$\vdash$$
 e : S  
G  $\vdash$  fun (x:T)  $\rightarrow$  e : T  $\rightarrow$  S

"Given an input of type T, this function computes a result of type S"

## **Adding Integers**

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

VAR  $X:T \in G$   $G \vdash e_1: int$   $G \vdash e_2: int$   $G \vdash i: int$   $G \vdash x:T$   $E \vdash e_1 + e_2: int$ 

FUN

 $G, x : T \vdash e : S$ 

 $G \vdash fun(x:T) \rightarrow e : T \rightarrow S$ 

APP

 $G \vdash e_1 : T \rightarrow S G \vdash e_2 : T$ 

 $G \vdash e_1 e_2 : S$ 

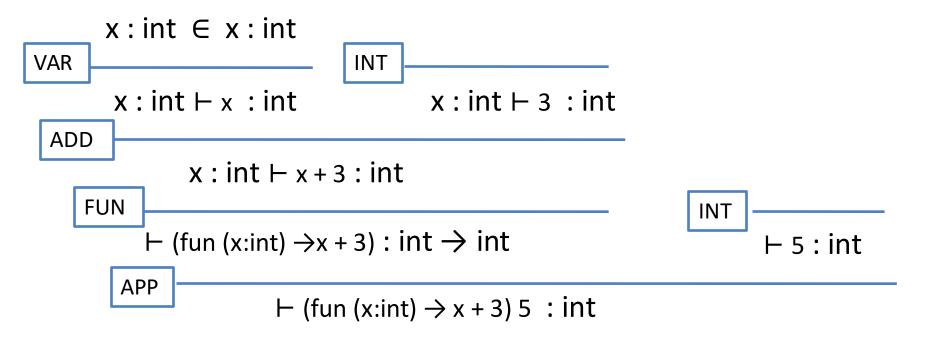
- Note how these rules correspond to the code.
- By convention, if G is empty we leave that spot blank.

## **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:

 $\vdash$  (fun (x:int)  $\rightarrow$  x + 3) 5 : int

#### **Example Derivation Tree**

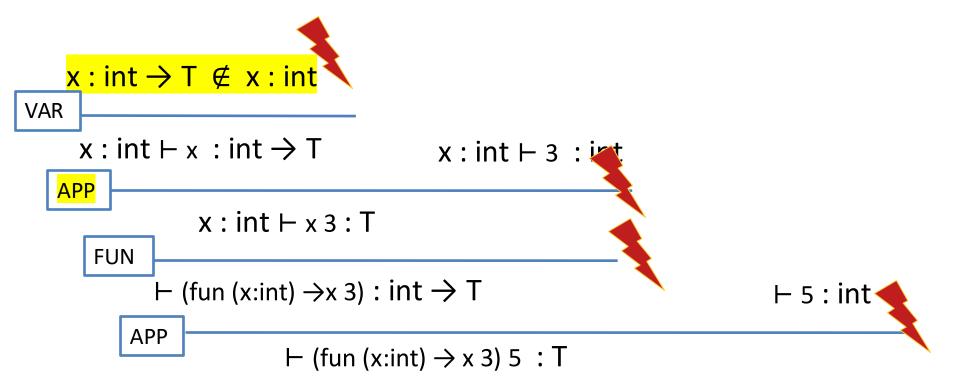


- Note: the OCaml function typecheck verifies the existence of this tree.
   The structure of the recursive calls when running typecheck is the same shape as this tree!
- Note that x: int ∈ E is implemented by the function lookup

# **Ill-typed Programs**

Programs without derivations are ill-typed

```
Example: There is no type T such that \vdash (fun (x:int) \rightarrow x 3) 5 : T
```



## **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)

## **Notes about this Typechecker**

- The interpreter evaluates the body of a function only when it's applied.
- The typechecker always checks 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 determine what type  $e_1$  will return.
- Question: Why is this an approximation?
- Question: What if well\_typed always returns false?

oat.pdf

#### **TYPECHECKING OAT**