Lecture 16

CIS 4521/5521: COMPILERS

Announcements

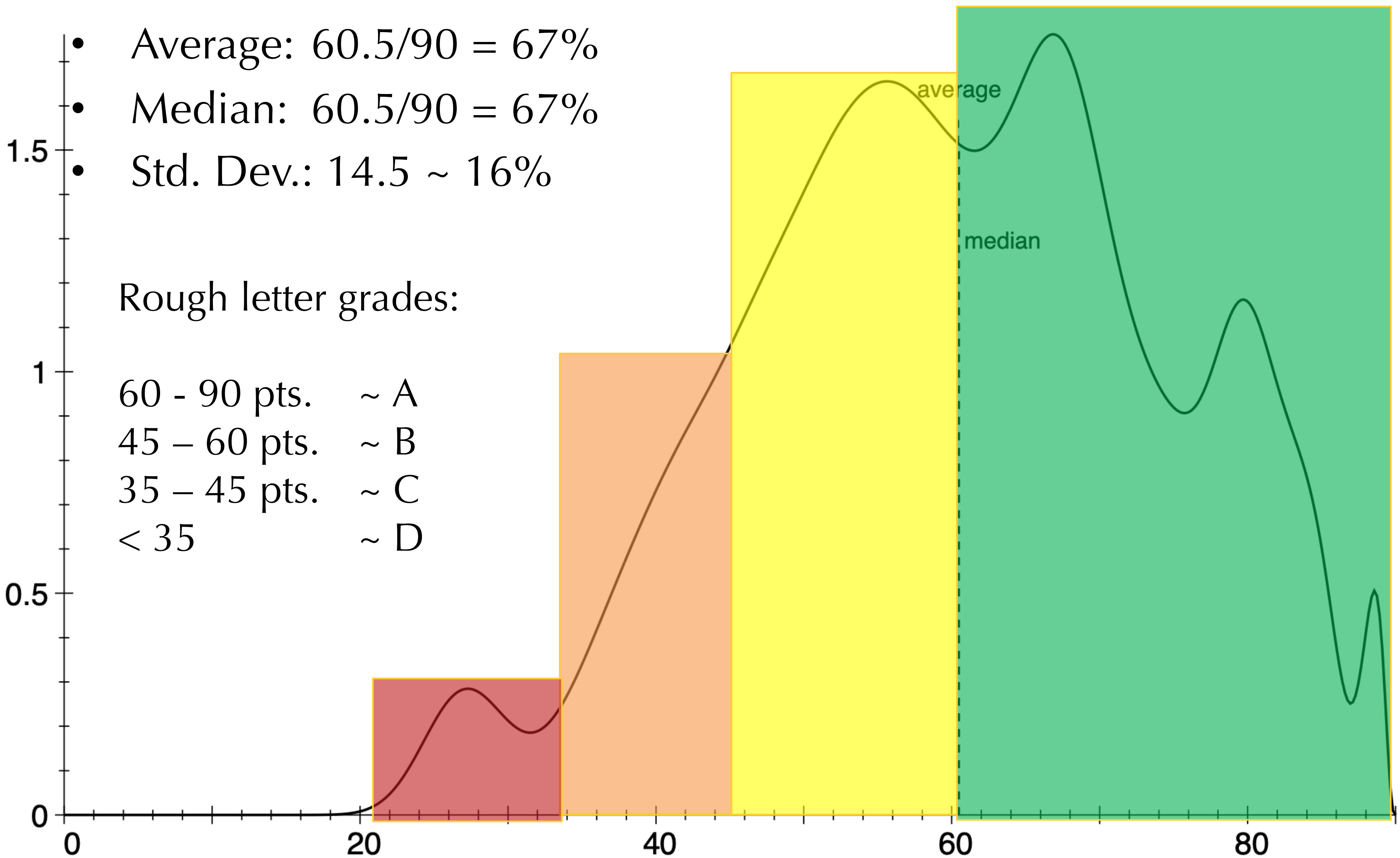
- HW4: OAT v. 1.0
  - Parsing & basic code generation
  - **Due: Wednesday, March 26th**
  - **Test case Due: TUESDAY, March 25th**

Midterm 2024

- Average: 60.5/90 = 67%
- Median: 60.5/90 = 67%
- Std. Dev.: 14.5 ~ 16%

Rough letter grades:

60 - 90 pts. ~ A
45 – 60 pts. ~ B
35 – 45 pts. ~ C
< 35 ~ D



See fun.ml

Eval2, Eval3

ENVIRONMENT BASED INTERPRETERS

Environment Based Interpreters

- Thread through an ***environment***, which maps variables to their values.
  - extend the environment when doing a function call
  - lookup variables in the current environment

- To properly handle first-class functions: use closures
  - a ***closure*** is a pair of a
    (1) a datastructure representing the saved environment, and
    (2) the function body definition

See cc.ml

CLOSURE CONVERSION

Closure Conversion Summary

- A ***closure*** is a pair of an environment and a code pointer
  - the environment is a map data structure binding variables to values
  - environment could just be a list of the values (with known indices)

- Building a closure value:
  - code pointer is a function that takes an extra argument for the environment: $A \to B$ becomes $(Env * A \to B)$
  - body of the closure "projects out" then variables from the environment
  - creates the environment map by bundling the free variables

- Applying a closure:
  - project out the environment, invoke the function (pointer) with the environment and its "real" argument

- Hoisting:
  - Once closure converted, all functions can be lifted to the top level

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 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?

Inference Rules

- We can read a judgment $G \vdash e$ as
  "the expression e is well scoped and has free variables in G"
- For any environment G, expression e, and statements s_1, s_2.

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

  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

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

Conclusion

- 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 ⊢ e "the free variables of e are included in G"

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

"the variable x is free, but in scope"

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

"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 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
```

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

- 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 $\mathcal{P}$(Var) be the (finite) powerset of variables (set of all finite sets)
  - Define *well-scoped* ⊆ ($\mathcal{P}$(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*

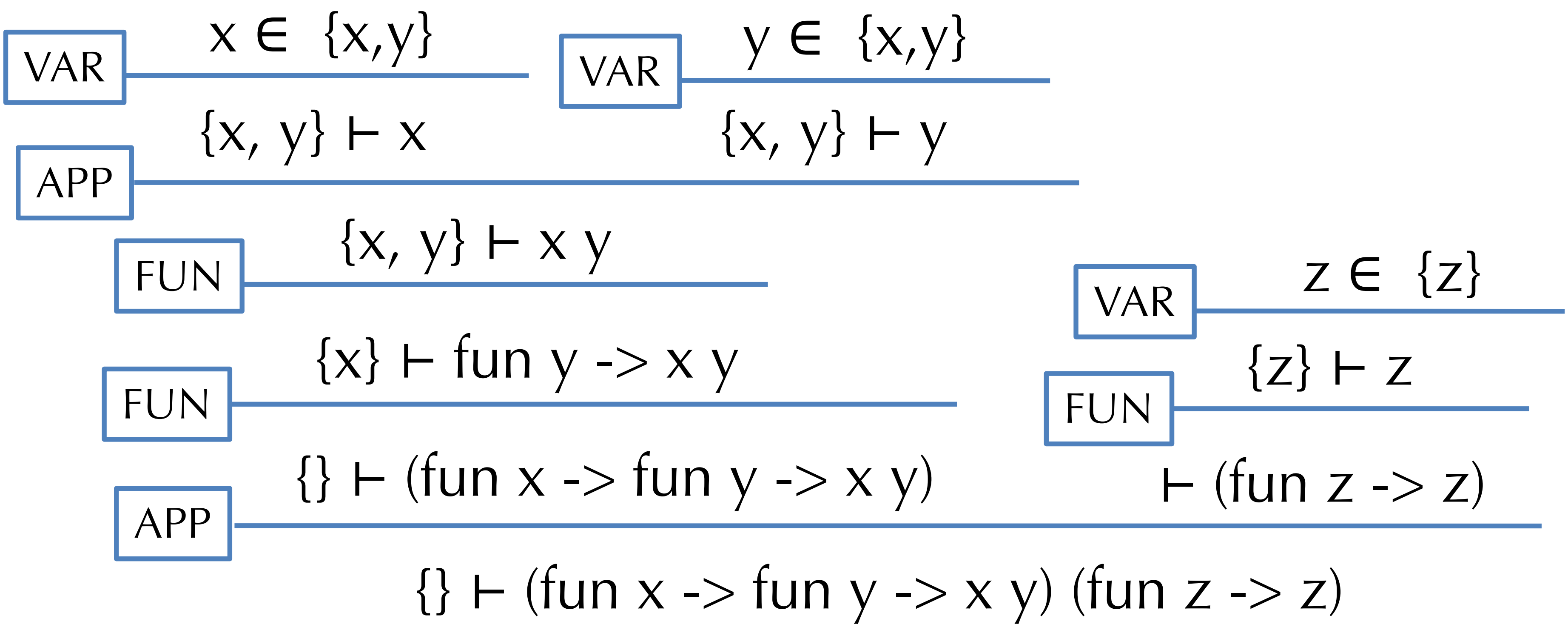
Checking Derivations

- 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:

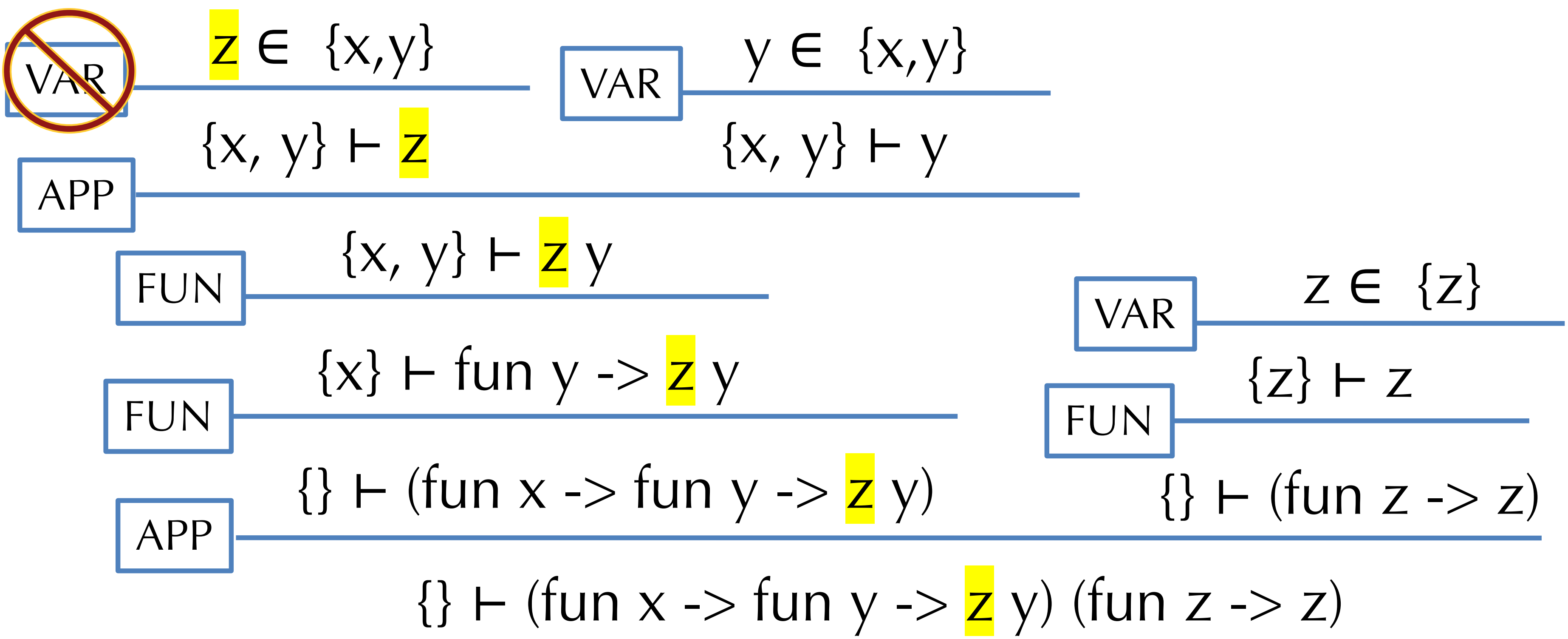
(fun x -> fun y -> x y) (fun z -> z)

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 ∈ 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 $G \vdash e$.

– case: VAR then we have $e = x$ (for some variable x) and $x \in G$. But $fv(e) = fv(x) = \{x\}$, but then $\{x\} \subseteq G$.

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

– 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 $fv(e_1\ e_2) = fv(e_1) \cup fv(e_2) \subseteq G$

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

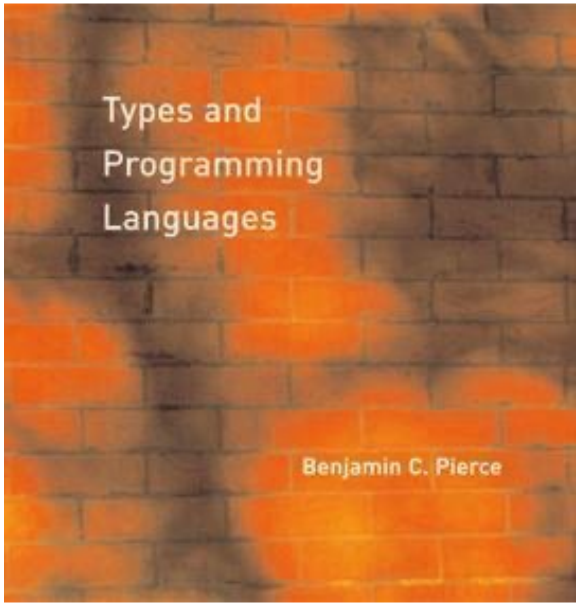
– case: FUN then we have $e = (\text{fun } x \text{ -> } e_1)$ for some x, e_1 and, by induction, we have $fv(e_1) \subseteq G \cup \{x\}$, but then we also have $fv(\text{fun } x \text{ -> } e_1) = fv(e_1) \setminus \{x\} \subseteq ((G \cup \{x\}) \setminus \{x\}) \subseteq G$

$$\frac{G \cup \{x\} \vdash e_1}{G \vdash \texttt{fun}\ x \rightarrow e_1}$$

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

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 5000 next Fall if you're interested in type systems!
  - ***Types and Programming Languages*** by Pierce



*Here (and later) we'll write context C for `G;L`, the combination of the global and local contexts.

CBV Operational Semantics

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

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

"Values evaluate to themselves"

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

"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. "

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*: $G \vdash e : T$ means "the expression e computes a value of type T, assuming its free variables have the types given in G"

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

"the variable x has type T an is in scope"

$$\frac{G \vdash e_1 : T \rightarrow S \qquad G \vdash e_2 : T}{G \vdash e_1\ e_2 : S}$$

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

$$\frac{G, x : T \vdash e : S}{G \vdash \texttt{fun}\ (x{:}T) \rightarrow e : T \rightarrow S}$$

"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:

INT

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

VAR

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

ADD

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

FUN

$$\frac{G, x : T \vdash e : S}{G \vdash \texttt{fun (x:T) -> e} : T \rightarrow S}$$

APP

$$\frac{G \vdash e_1 : T \rightarrow S \qquad G \vdash e_2 : T}{G \vdash e_1\ e_2 : S}$$

- 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:

$$\vdash \texttt{(fun (x:int) → x + 3) 5} : \text{int}$$

Example Derivation Tree

$x : \mathrm{int} \in x : \mathrm{int}$

VAR ──────────────── INT ────────────────

$x : \mathrm{int} \vdash$ `x` $: \mathrm{int}$ $x : \mathrm{int} \vdash$ `3` $: \mathrm{int}$

ADD ────────────────────────────────

$x : \mathrm{int} \vdash$ `x + 3` $: \mathrm{int}$

FUN ──────────────────────────────── INT ────────

$\vdash$ `(fun (x:int) →x + 3)` $: \mathrm{int} \longrightarrow \mathrm{int}$ $\vdash$ `5` $: \mathrm{int}$

APP ──

$\vdash$ `(fun (x:int) → 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`

Ill-typed Programs

- Programs without derivations are ill-typed

  Example: There is no type T such that

  $\vdash$ `(fun (x:int) → x 3) 5` : T

x : int → T ∉ x : int

VAR

x : int ⊢ x : int → T   x : int ⊢ 3 : int

APP

x : int ⊢ x 3 : T

FUN

⊢ `(fun (x:int) →x 3)` : int → T   ⊢ 5 : int

APP

⊢ `(fun (x:int) → 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`?