CS4212: Compiler Design

Week 9: Types and Type Checking

Ilya Sergey

ilya@nus.edu.sg

ilyasergey.net/CS4212/

(Untyped) Lambda Calculus

- The **lambda calculus** is a minimal programming language.
  - Note: we're writing (fun x -> e) lambda-calculus notation: λ x. e
- It has **variables**, **functions**, and **function application**.
  - That's it!
  - It's Turing Complete.
  - It's the foundation for a *lot* of research in programming languages.
  - Basis for "functional" languages like Scala, OCaml, Haskell, etc.

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
     | exp1 exp2            function application
     | ( exp )              parentheses
```

More Examples

Pairs and zero-checking

Recap: Operational Semantics of Lambda Calculus

- Substitution function (in Math):

x{v/x} = v		*(replace the free x by v)*
y{v/x} = y		*(assuming y ≠ x)*
(fun x → exp){v/x}	= (fun x → exp)	*(x is bound in exp)*
(fun y → exp){v/x}	= (fun y → exp{v/x})	*(assuming y ≠ x)*
(e_1 e_2){v/x}	= (e_1{v/x} e_2{v/x})	*(substitute everywhere)*

- Examples:

(x y) {(fun z → z z)/y}

= x (fun z → z z)

(fun x → x y){(fun z → z z)/y}

= fun x → x (fun z → z z)

(fun x → x){(fun z → z z)/x}

= fun x → x // x is not free!

Free Variables and Scoping

```
let add = fun x → fun y → x + y
let inc = add 1
```

- The result of add 1 is a function
- After calling add, we can't throw away its argument (or its local variables) because those are needed in the function returned by add.
- We say that the variable x is *free* in fun y → x + y
  - Free variables are defined in an outer scope
- We say that the variable y is *bound* by "fun y" and its scope is the body "x + y" in the expression fun y → x + y
- A term with no free variables is called *closed*.
- A term with one or more free variables is called *open*.

Free Variable Calculation

- An OCaml function to calculate the set of free variables in a lambda expression:

```
let rec free_vars (e:exp) : VarSet.t =
  begin match e with
    | Var x        -> VarSet.singleton x
    | Fun(x, body) -> VarSet.remove x (free_vars body)
    | App(e1, e2)  -> VarSet.union (free_vars e1) (free_vars e2)
  end
```

- A lambda expression `e` is *closed* if `free_vars e` returns `VarSet.empty`
- In mathematical notation:

$$\begin{aligned} fv(x) &= \{x\} \\ fv(\mathbf{fun}\ x \rightarrow exp) &= fv(exp) \setminus \{x\} \qquad \textit{('x' is a bound in exp)} \\ fv(exp_1\ exp_2) &= fv(exp_1) \cup fv(exp_2) \end{aligned}$$

Operational Semantics

- Specified using just two inference rules with judgments of the form $\text{exp} \Downarrow \text{val}$
  - Read this notation a as "program exp evaluates to value val"
  - This is *call-by-value* semantics: function arguments are evaluated before substitution

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

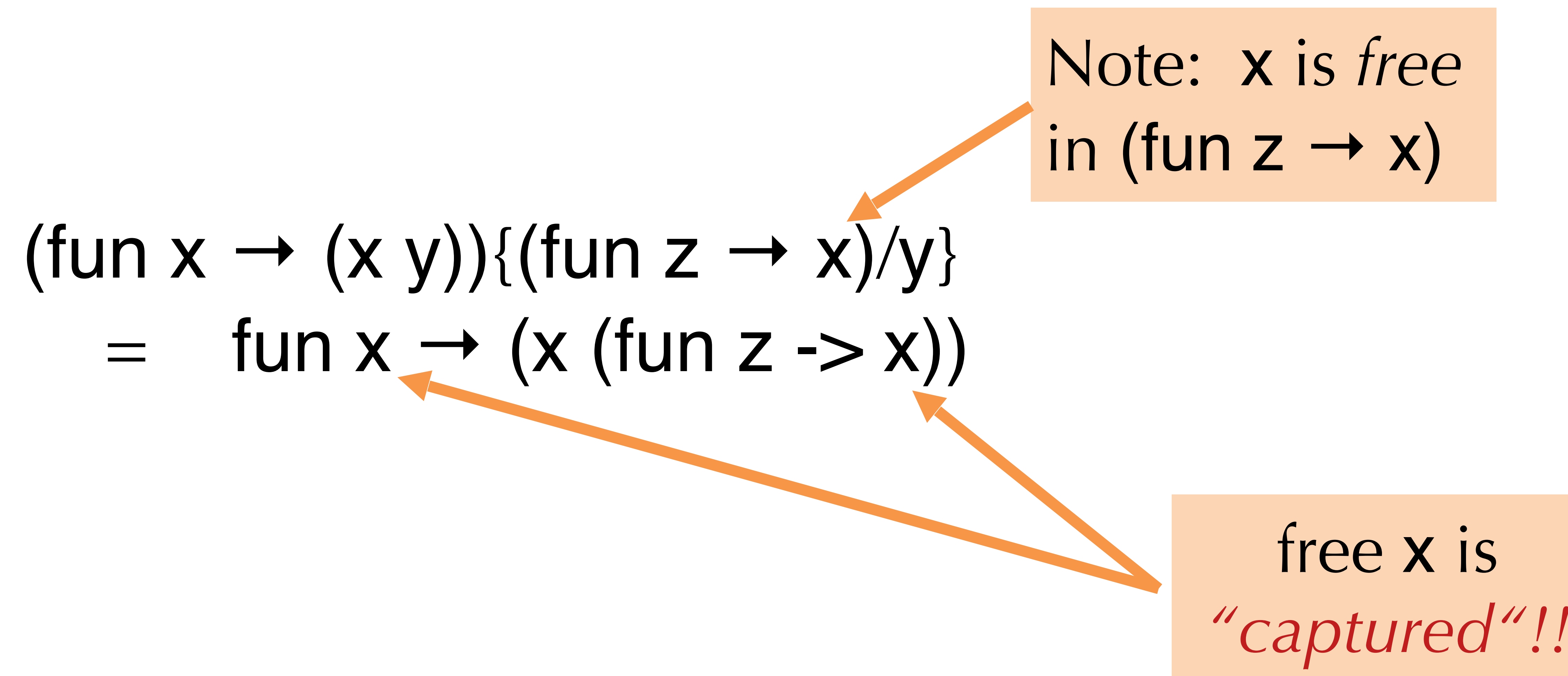
"Values evaluate to themselves"

$$\frac{\text{exp}_1 \Downarrow (\text{fun } x \rightarrow \text{exp}_3) \qquad \text{exp}_2 \Downarrow v \qquad \text{exp}_3\{v/x\} \Downarrow w}{\text{exp}_1\ \text{exp}_2 \Downarrow w}$$

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

Variable Capture

- Note that if we try to naively "substitute" an open term, a bound variable might ***capture*** the free variables:



- Usually *not* the desired behaviour
  - This property is sometimes called "dynamic scoping"
    The meaning of "x" is determined by where it is bound dynamically,
    not where it is bound statically.
  - Some languages (e.g. emacs lisp) are implemented with this as a "feature"
  - But: it leads to hard-to-debug scoping issues

Alpha Equivalence

- Note that the names of bound variables don't matter to the semantics
  - i.e. it doesn't matter which variable names you use, as long as you use them consistently:

(fun x → y x) is the "same" as (fun z → y z)

the choice of "x" or "z" is arbitrary, so long as we consistently rename them

Two terms that differ only by consistent renaming of *bound* variables are called *alpha equivalent*

- The names of *free* variables **do** matter:

(fun x → y x) is *not* the "same" as (fun x → z x)

Intuitively: y an z can refer to different things from some outer scope

Students who cheat by "renaming variables" are trying to exploit alpha equivalence…

Fixing Substitution

- Consider the substitution operation:

$$e_1\{e_2/x\}$$

- To avoid capture, we define substitution to pick an alpha equivalent version of e_1 such that the bound names of e_1 don't mention the free names of e_2.
  - Then do the "naïve" substitution.

For example: (fun x → (x y)){(fun z → x)/y}

= (fun x' → (x' (fun z → x))

rename x to x'

This is fine:

(fun x → (x y)){(fun x → x)/y}

= (fun x → (x (fun x → x))

= (fun a → (a (fun b → b))

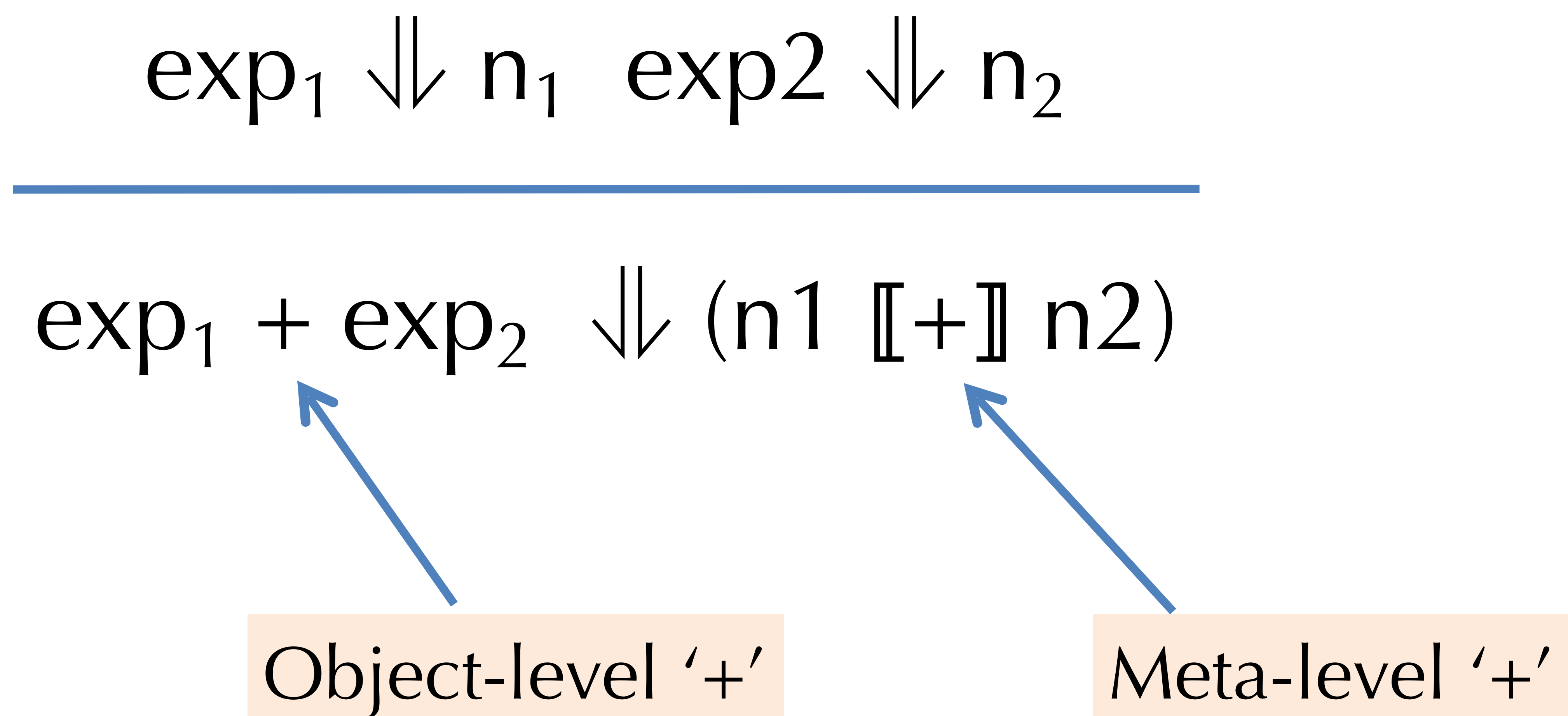
Demo: Implementing the Interpreter

- https://github.com/cs4212/week-08-lambda-2023

- lambda.ml – untyped lambda-calculus
- lambda_int.ml – untyped lambda-calculus with integers
- stlc.ml – simply-typed lambda-calculus

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*



Semantic 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.
  Why?

```
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 for *Static* Semantic Analysis

- Recall the interpreter from the Eval2 module in lambda_int.ml:

```
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

Semantic Analysis

- The *semantic analysis* phase
  - Resolve symbol occurrences to declarations / binders

    ```
    ex.c:3:11: error: 'i' undeclared (first use in this function)
    ```

  - Type-check AST

    ```
    ex.c:4:5: warning: assignment makes integer from pointer without a cast
    ```

- Main data structure manipulated by semantic analysis: *symbol table*
  - Mapping from symbols to information about those symbols (its type, location in source text, ...)
  - Symbol table is used to help translation into IR
  - Semantic analysis may also decorate AST (e.g., attach type information to expressions, or replace symbols with references to their symbol table entry).
  - Semantic analysis may not be a separate phase – e.g., may be incorporated into IR translation

Warm-Up: 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" ($fv(e) \subseteq G$)

$fv(x)$	=	$\{x\}$	
$fv(\text{fun } x \rightarrow exp)$	=	$fv(exp) \setminus \{x\}$	*('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"

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

"x is available in the function body"

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
```

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

Semantic Analysis via Types

What is a Type?

- *Intrinsic view (Church-style)*: a type is syntactically part of a program.
  - A program that cannot be typed is not a program at all
  - Types do not have inherent meaning – they are just used to define the syntax of a program
- *Extrinsic view (Curry-style)*: a type is a *property* of a program.
  - For any program and every type, either the program has that type or not
  - A program may have multiple types
  - A program may have no types

Why Types?

- ***Type checking*** (ensuring that the program is ascribed a "correct" type) catches errors at compile time, eliminating a class of mistakes that would otherwise lead to run-time errors, provided type information

- ***Type inference*** derives type information from the code (think function parameters in OCaml vs Java)

- ***Type information*** is sometimes necessary for code generation
  - Floating-point + is not the same instruction as integer + is not the same as pointer/integer +
  - pointer/integer compiled differently depending on pointer type
  - Assignment x = y compiled differently if y is an **int** or a **struct**

What is a type system?

- A type system consists of a system of judgements and inference rules
  - (Extrinsic view) A **judgement** is a *claim*, which may or may not be valid
    - $\vdash 3 : \texttt{int}$ – "3 has type integer"
    - $\vdash (1 + 2) : \texttt{bool}$ – "(1+2) has type boolean"
  - **Inference rules** are used to derive *valid* judgements from other valid judgements.

$$\textsc{Add}\;\frac{\vdash e_1 : \texttt{int} \qquad \vdash e_2 : \texttt{int}}{\vdash e_1 + e_2 : \texttt{int}}$$

Read: "If e_1 and e_2 have type $\texttt{int}$, so does $e_1 + e_2$"

- Type system might involve many different kinds of judgement
  - Well-typed expressions
  - Well-formed types
  - Well-formed statements
  - …

Inference Rules, General Form

- An *inference rule* consists of a list of **premises** $J_1, ..., J_n$ and one **conclusion** J (optionally: a side-condition):

$$\frac{J_1 \qquad J_2 \qquad \cdots \qquad J_n}{J}\ \textsc{Side-condition}$$

  - Side-condition: additional premise, but not a judgement
  - Read *top-down*: If J_1 and J_2 and ... and J_n are valid, and the side condition holds, then J is valid.
  - Read *bottom-up*: To prove J is valid, sufficient to prove J_1, J_2, ... J_n are valid

Type Judgments

- In the judgment: $G \vdash e : t$
  - G is a *typing environment* or a *type context*
  - G maps variables to types. It is just a set of bindings of the form:
    $x_1 : t_1, x_2 : t_2, \ldots, x_n : t_n$
- A type judgement takes the form $G \vdash e : t$
  "Under the type environment G, the expression e has type t"
- For example:
  $x : int, b : bool \vdash$ if (b) 3 else $x : int$

- What do we need to *check* 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. $x : int, b : bool \vdash 3 : int$
  - x must be an int   i.e. $x : int, b : bool \vdash x : int$

Simply-typed Lambda Calculus with Integers

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

INT

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

VAR

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

ADD

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

FUN

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

APP

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

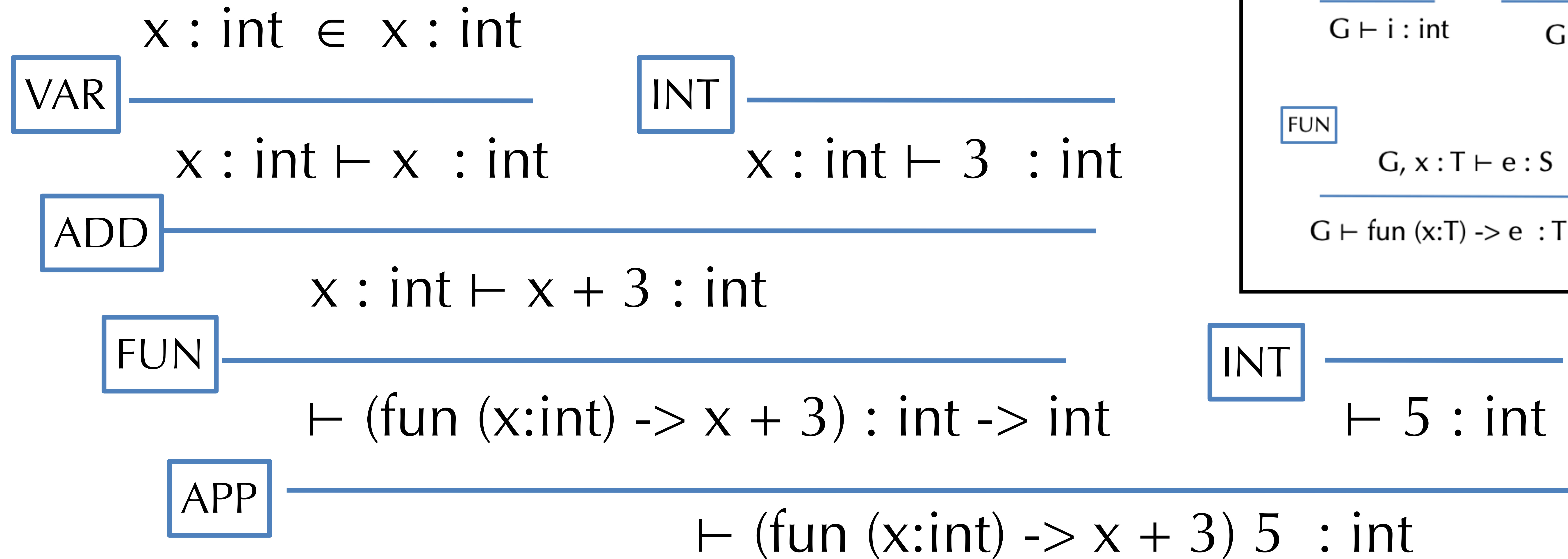
- Note how these rules correspond to the OCaml code.

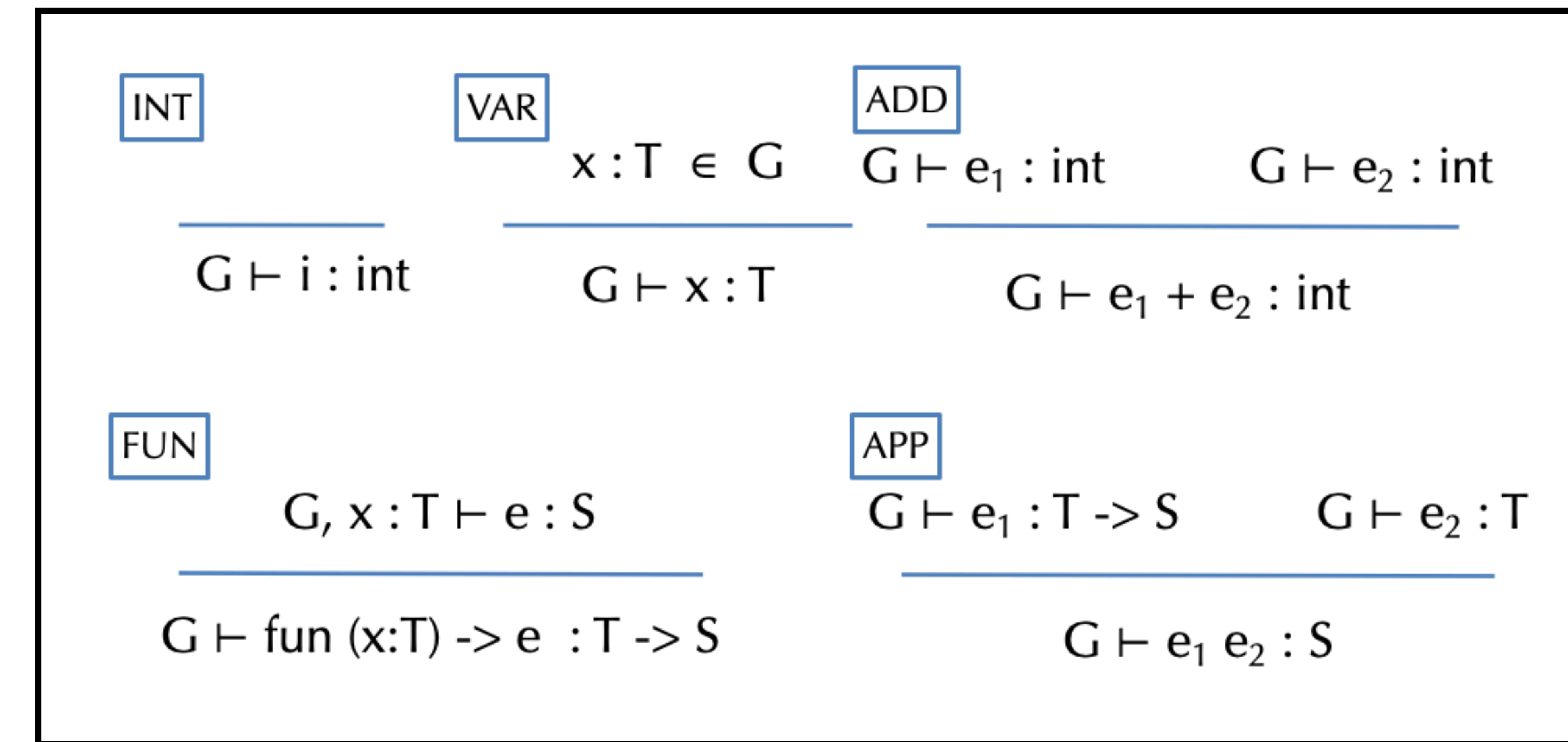
Model for Type Checking

- 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*.
- **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





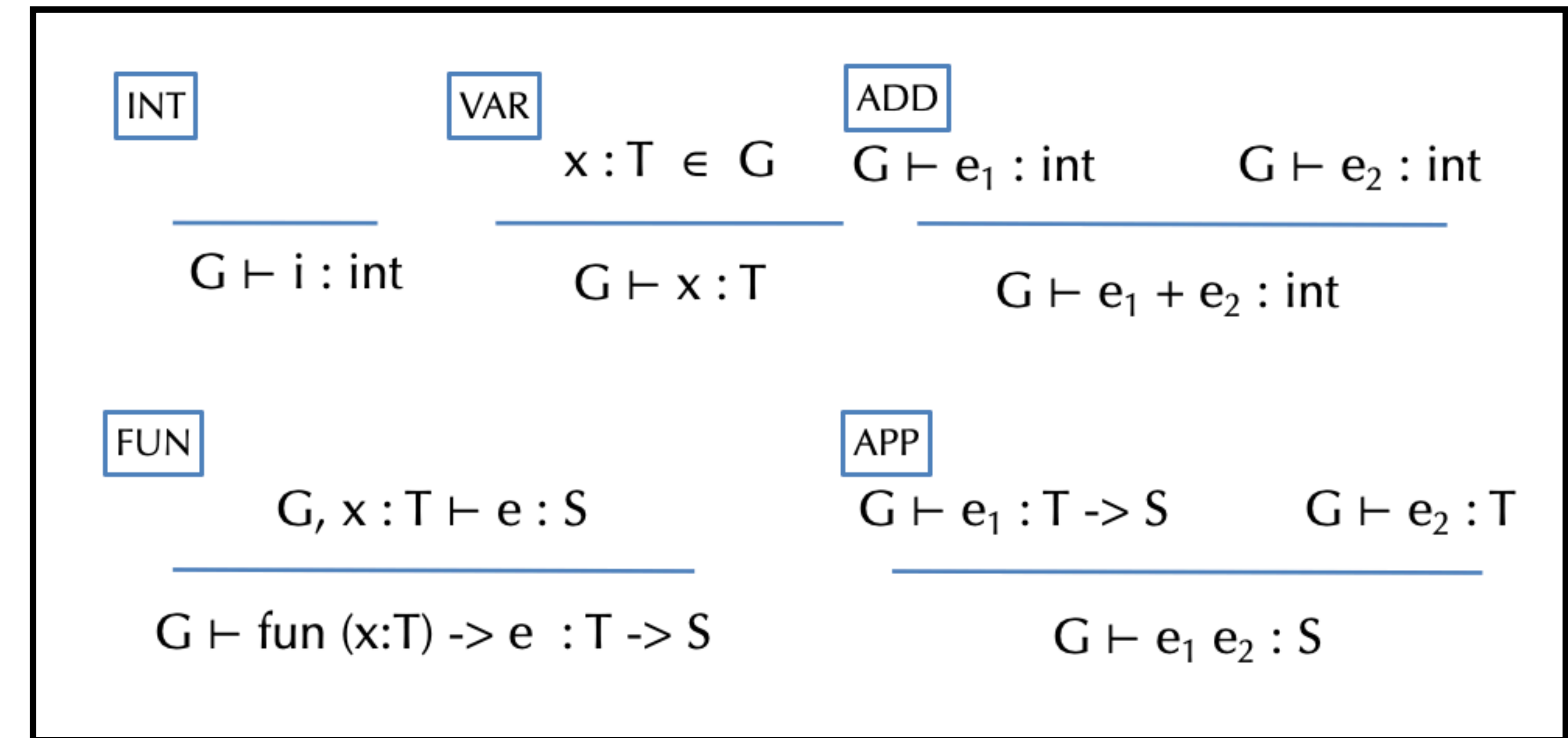
- Note: the OCaml function `typecheck` verifies the existence of this tree. The structure of the recursive calls when running `typecheck` is the same shape as this tree!
- Note that x : int ∊ E is implemented by the function `lookup`

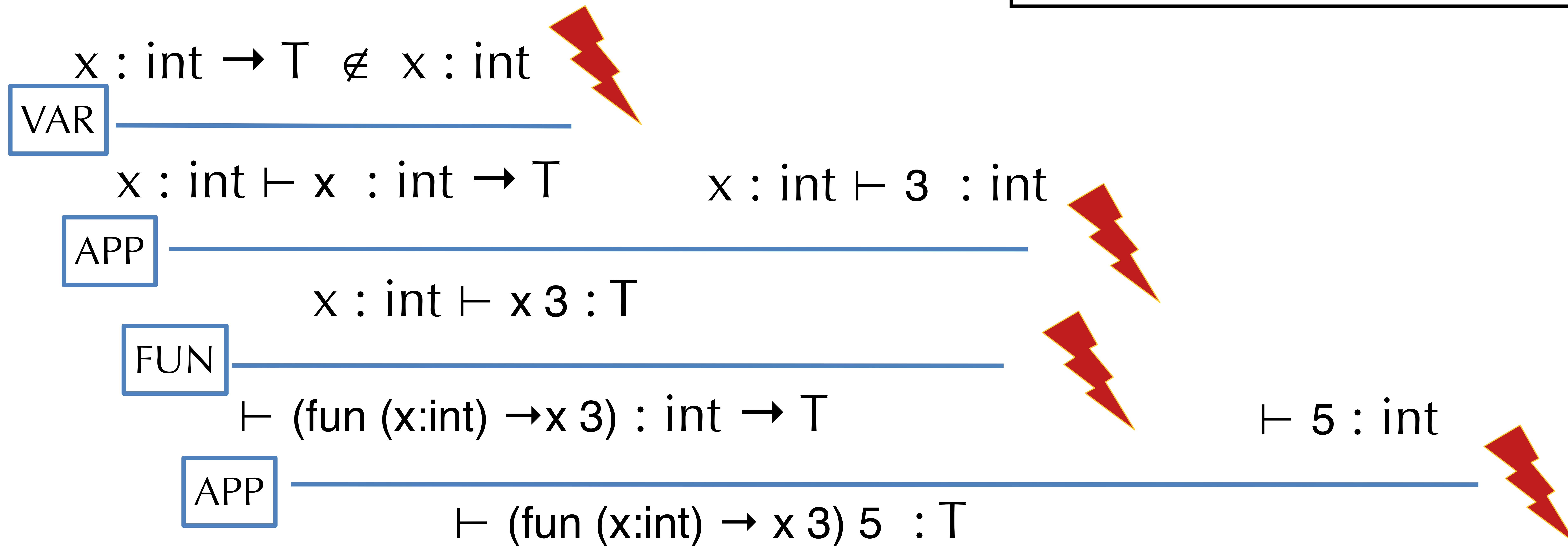
Ill-typed Programs

- Programs without derivations are ill-typed

  Example: There is no type T such that

  $\vdash$ (fun (x:int) → x 3) 5 : T





Simply-typed Lambda Calculus with Integers

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

INT

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

VAR

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

ADD

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

FUN

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

APP

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

- Note how these rules correspond to the OCaml code.

Implementing a Type Checker for Lambda Calculus

See stlc.ml

Exercise

- Implement the missing parts of the type-checker

Notes about this Type Checker

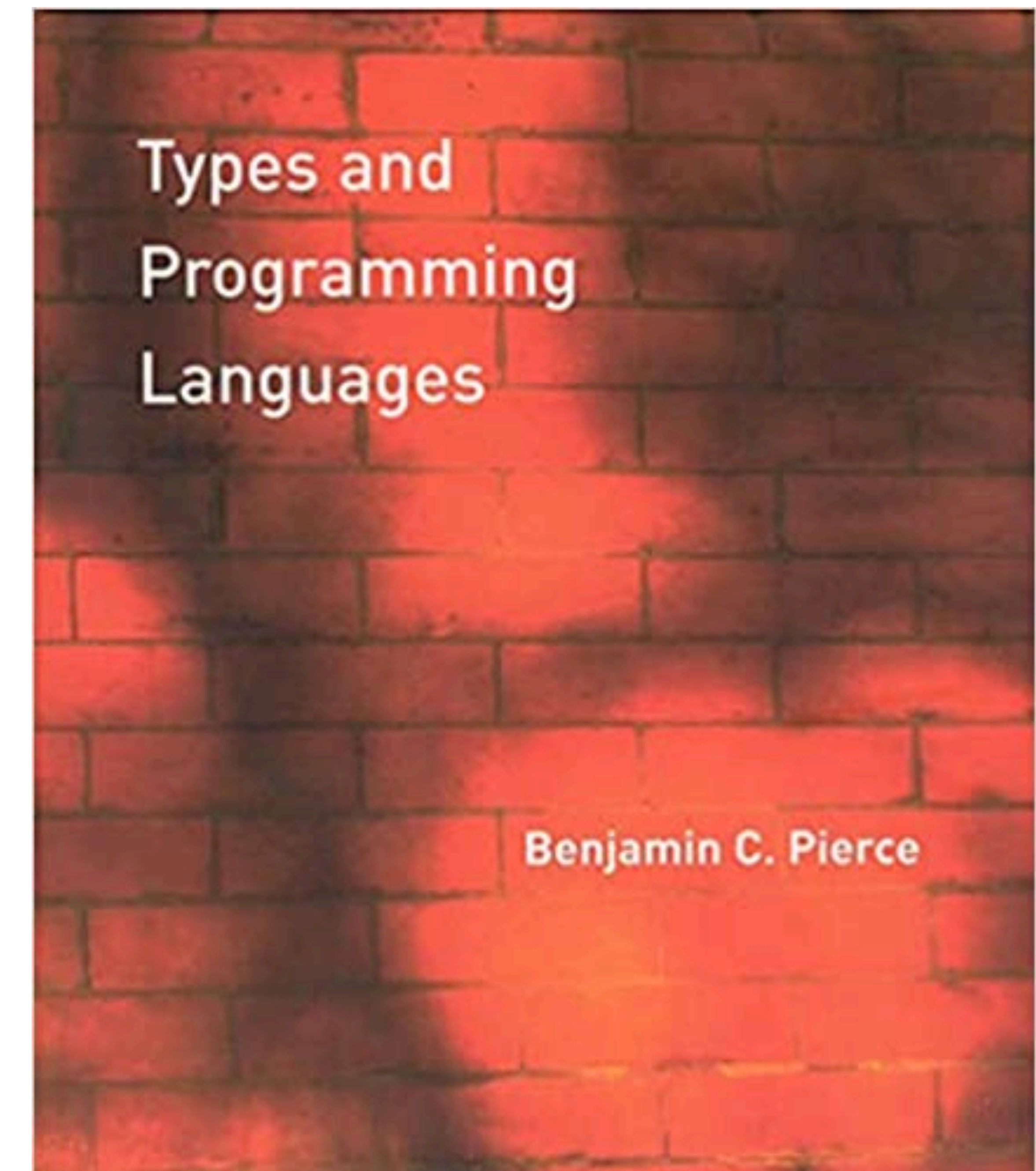
- In the interpreter, we only evaluate the body of a function when it's applied.
- In the type checker, we always check the body of the function (even if it's never applied.)
  - We *assume* the input has some type (say t_1) and reflect this in the type of the function (t_1 -> t_2).
- Dually, at a call site (e_1 e_2), we don't know what *closure* we're going to get as e_1.
  - But we can calculate e_1's type, check that e_2 is an argument of the right type, and also determine what type will (e_1 e_2) have.
- **Question**: Why is this a valid approximation of the dynamic program behaviour?

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.

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.
  - Check out **oat-v1-typing.pdf**
- 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 judgment ($G \vdash e : t$) by searching backwards through the rules.
- Strong mathematical foundations
  - The "Curry-Howard correspondence":
    - Programming Language ~ Logic,
    - Program ~ Proof,
    - Type ~ Proposition
- Talk to me you're interested in type systems!



Type Safety

Theorem: (type soundness of simply typed lambda calculus with integers)

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

"Well typed programs do not go wrong."

– Robin Milner, 1978

Next Lecture

- More on types and type safety
- Advanced type features
- Mistakes in type system design