CS 421 Lecture 21 – Proof systems

- Defining proof systems
  - Judgments
  - Axioms
  - Rules of inference
  - Proofs
- T_{simp}: Simple proof system for types in OCaml
- OS_{simp}: Simple proof system for operational semantics of OCaml

Proof systems

- Proof system: formalized representation of mathematical proofs based on *axioms* and *rules of inference*.
- Can be used to formalize deductions for many purposes
  - **Type-checking** axioms and rules of inference allow proofs of assertions (a.k.a. "judgments") of the form "expression e has type τ".
  - **Operational semantics** rules allow proofs of judgments of the form "e evaluates to *v*".
  - **Axiomatic semantics** rules allow proofs of judgments of the form "If the variables in a program initially satisfy some conditions *C*, then after executing statement *S*, they will satisfy conditions *C'* ".

Proof systems

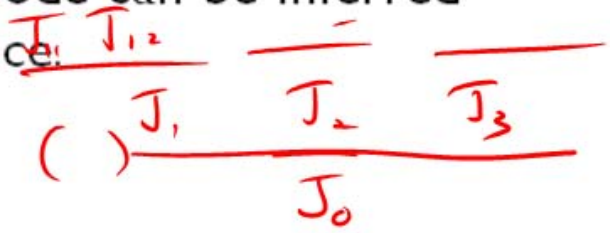
To define a proof system, we need to define three things:

- **Judgments**: A judgment is an assertion whose truth is subject to proof.
- **Axioms**: Judgments that are assumed to be true without proof. There are usually an infinite number of axioms, so they can't all be listed, but they need to be described in some way. Written: $\frac{}{J}$
- **Rules of inference**: Rules that allow you to infer a judgment from one or more previously-inferred judgments. Written: $\frac{J_1 \ \ldots \ J_n}{J}$

Proofs

Given a proof system, a **proof** is a tree labeled with judgments, such that:

- Every judgment labeling a leaf node is an axiom
- Every judgment labeling an internal node can be inferred from its children by a rule of inference:



Notational notes:

1. Axioms and rules of inference are usually given names, and these names are placed in the proof tree
2. Proof trees are written with the root – the main judgment being proved at the bottom.

T_{simp} – simplified Ocaml type system

Types: int | $\tau \longrightarrow \tau'$ (for any types τ and τ')

e.g. $(int \to int) \to int$, ---

Type environments Γ: partial mapping from variables to types, written as identical envs

$\{ x : int,\ y : int \to int \to int \}$

$\emptyset[x : int][y : int \to int \to int]$

Judgments: $\Gamma \vdash e : \tau$

$\{x : int \to int\} + x\ 0 : int$

Expressions: $x \mid e_1 e_2 \mid fun\ x \to e \mid 0 \mid 1 \mid \ldots \mid + \mid - \mid \ldots$

T_{simp} – simplified Ocaml type system

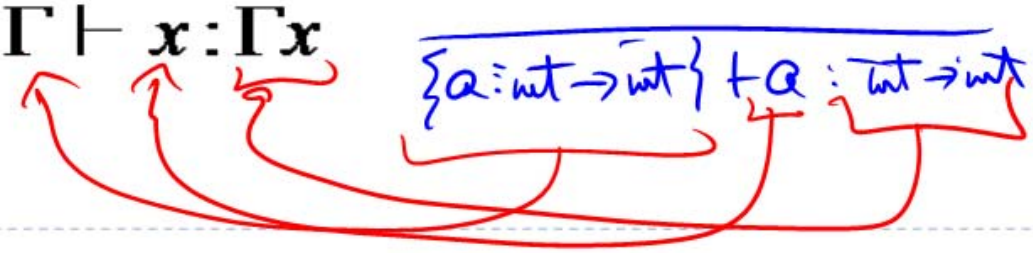
Axioms:

(Const) $\dfrac{}{\Gamma \vdash 0 : int}$ $\dfrac{}{\Gamma \vdash 1 : int}$

$$\frac{}{\Gamma \vdash + : \mathbf{int} \to \mathbf{int} \to \mathbf{int}}$$

(and many more)

(Var) $\dfrac{}{\Gamma \vdash x : \Gamma x}$



T_{simp} – simplified Ocaml type system

Rules of inference:

(Application)
$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'}$$

(Abstraction)
$$\frac{\Gamma[x : \tau] \vdash e : \tau'}{\Gamma \vdash \text{fun } x \text{ -> } e : \tau \to \tau'}$$

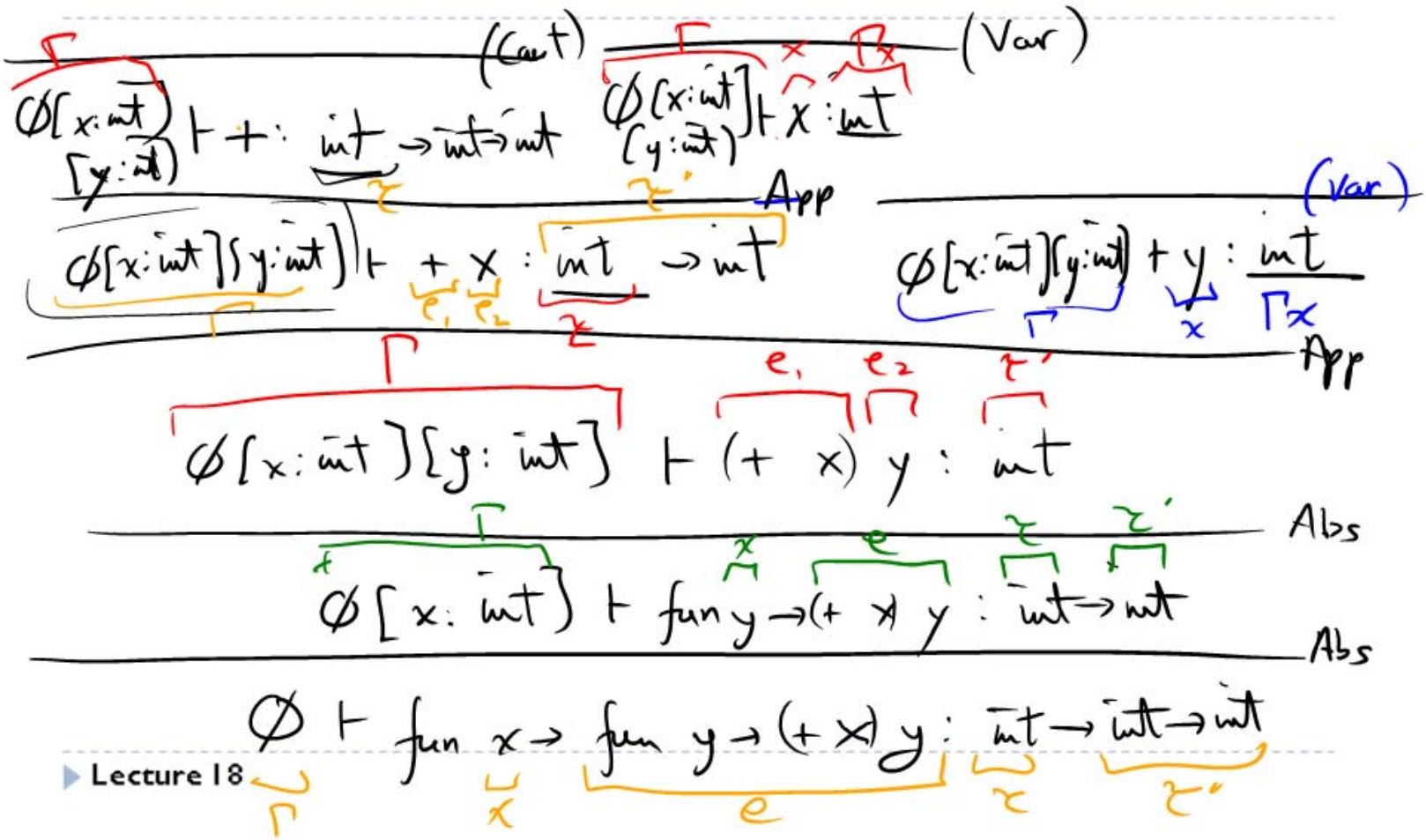
How to use axioms and rules of inference

Symbols occurring in rules are (mostly) *meta-variables* standing in for actual symbols in the language of judgments. A valid *instance* of a rule is obtained by substituting uniformly for the meta-variables in the rule.

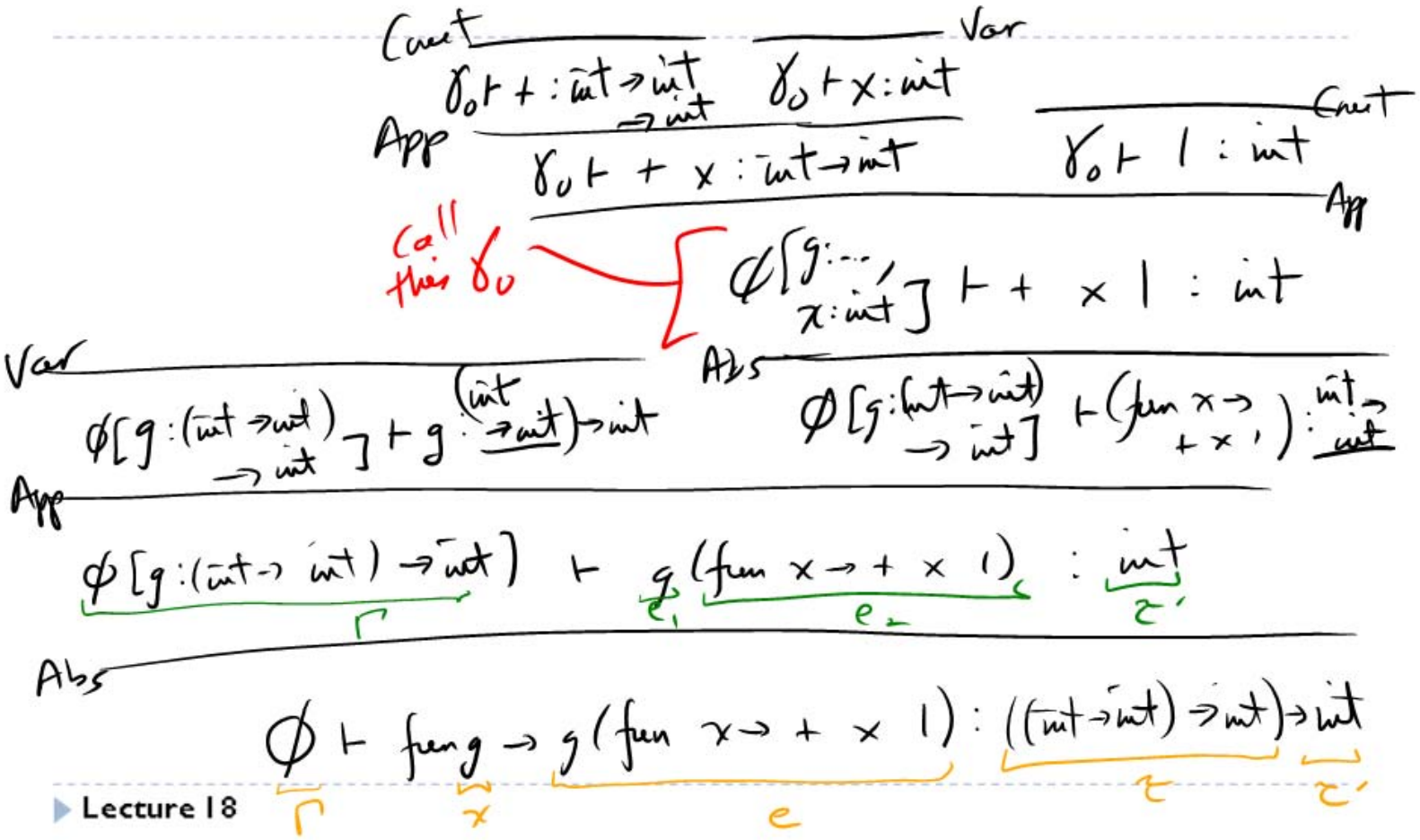
$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'}$$

$$\frac{\{x : \text{int} \to \text{int}\} \vdash x : \text{int} \to \text{int} \qquad \{x : \text{int} \to \text{int}\} \vdash 0 : \text{int}}{\{x : \text{int} \to \text{int}\} \vdash x\ 0 : \text{int}}$$

Example: fun x -> fun y -> (+ x) y : int→int →int



Example: fun g -> g(fun x -> + x 1): ((int→int) →int) →int



$$\gamma_0 \vdash + : int \to int \to int \qquad \gamma_0 \vdash x : int$$

$$\gamma_0 \vdash + \; x : int \to int \qquad \gamma_0 \vdash 1 : int$$

$$\phi[g : \ldots, x : int] \vdash + \; x \; 1 : int$$

$$\phi[g : (int \to int) \to int] \vdash g : (int \to int) \to int \qquad \phi[g : (int \to int) \to int] \vdash (\text{fun } x \to + \; x \; 1) : int \to int$$

$$\underbrace{\phi[g : (int \to int) \to int]}_{\Gamma} \vdash \underbrace{g}_{e_1} \; \underbrace{(\text{fun } x \to + \; x \; 1)}_{e_2} : \underbrace{int}_{\tau'}$$

$$\underbrace{\phi}_{\Gamma} \vdash \text{fun } \underbrace{g}_{x} \to \underbrace{g(\text{fun } x \to + \; x \; 1)}_{e} : \underbrace{((int \to int) \to int) \to int}_{\tau \to \tau'}$$

Notes on T_{simp}

(1) Given Γ, e, and τ, the structure of the proof tree is completely determined by e – it is the same as the abstract syntax. The content of the proof tree is almost completely determined by e and τ; however, in the application rule, even given Γ, e1, e2, and τ', τ is not uniquely determined.

(2) *Proving* $\varnothing \vdash e : \tau$ is called **type checking**. *Finding* τ such that $\varnothing \vdash e : \tau$ can be proved is called **type inference.**

OS_{simp} – simplified Ocaml operational semantics

The **operational semantics** of a language says, in an abstract way, how programs in a language are executed.

For a functional language like Ocaml, the operational semantics should say how expressions are evaluated.

E.g. "(fun x -> x*x) 4" evaluates to "16".

To simplify the formal presentation, we will take the view that the evaluation of an expression involves transforming it to another, simpler expression.

OS_{simp} – simplified Ocaml operational semantics

We give the operational semantics of a very simplified Ocaml as a proof system. We need to define the judgments of the system, and then give the axioms and rules of inference.

Expressions (simplified Ocaml): constants, variables, fun x -> e, e1 e2

Values: constants, closed abstractions (i.e. fun x -> e, where e has no free variables other than x)

Judgments: $e \Downarrow v$ (where e is closed)

$(\text{fun } x \to x * x)\,4 \Downarrow 16$

OS_{simp} – simplified Ocaml operational semantics

Axioms:
(Const) $\dfrac{}{k \Downarrow k}$ for constants k

(Abstr) $\dfrac{}{\textbf{fun } x \textbf{ -> } e \Downarrow \textbf{fun } x \textbf{ -> } e}$ (fun x -> e closed)

OS_{simp} – simplified Ocaml operational semantics

Rule of inference:
(Application)

subst. v' for x in e

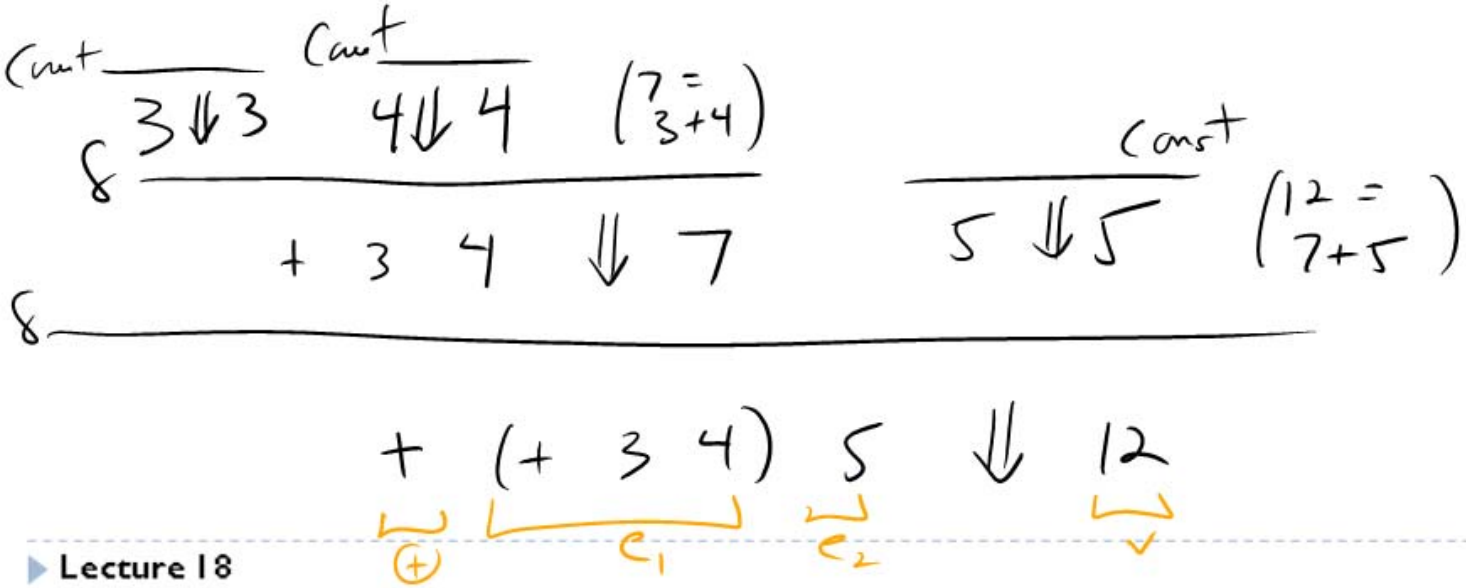
$$\frac{e_1 \Downarrow \text{fun } x \text{ -> } e \quad e_2 \Downarrow v' \quad e[v'/x] \Downarrow v}{e_1 e_2 \Downarrow v}$$

(δ rules)

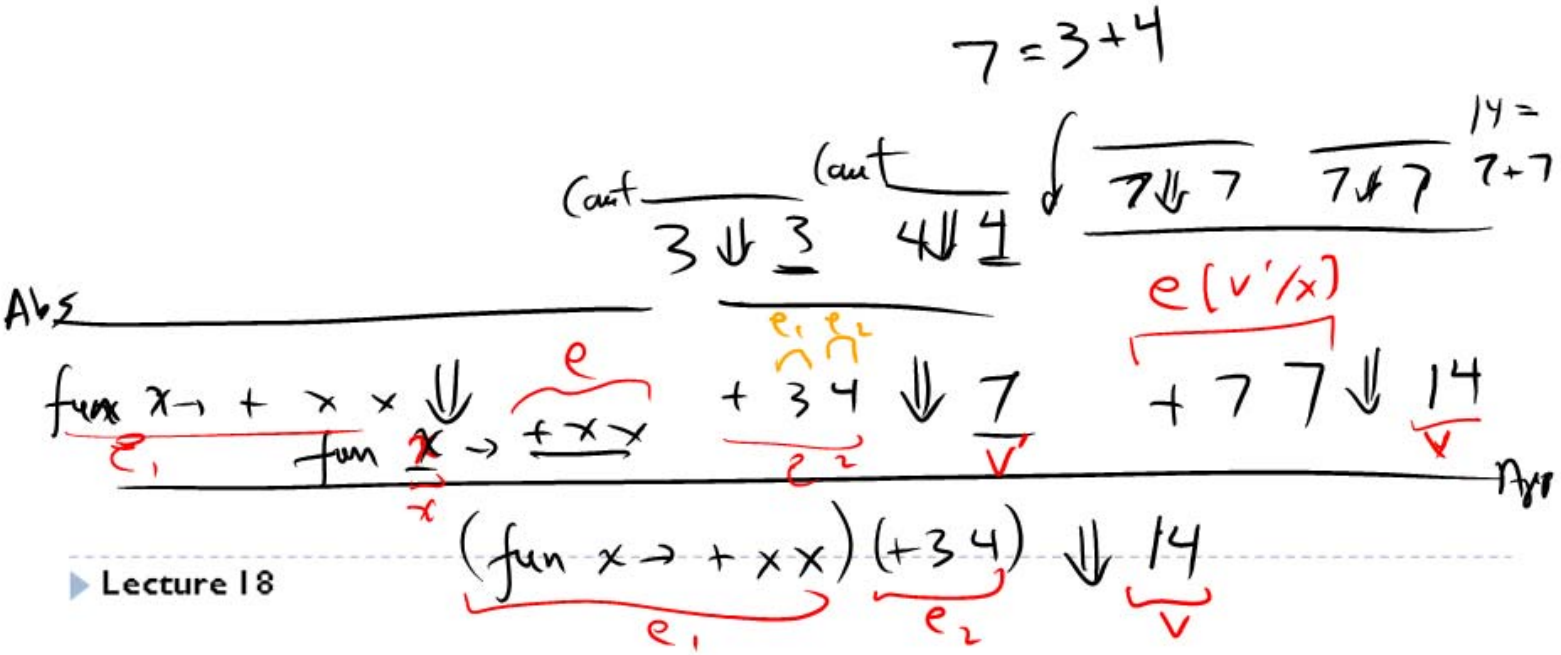
$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v = v_1 \oplus v_2}{e_1 \oplus e_2 \Downarrow v}$$

where $\oplus$ is any built-in function

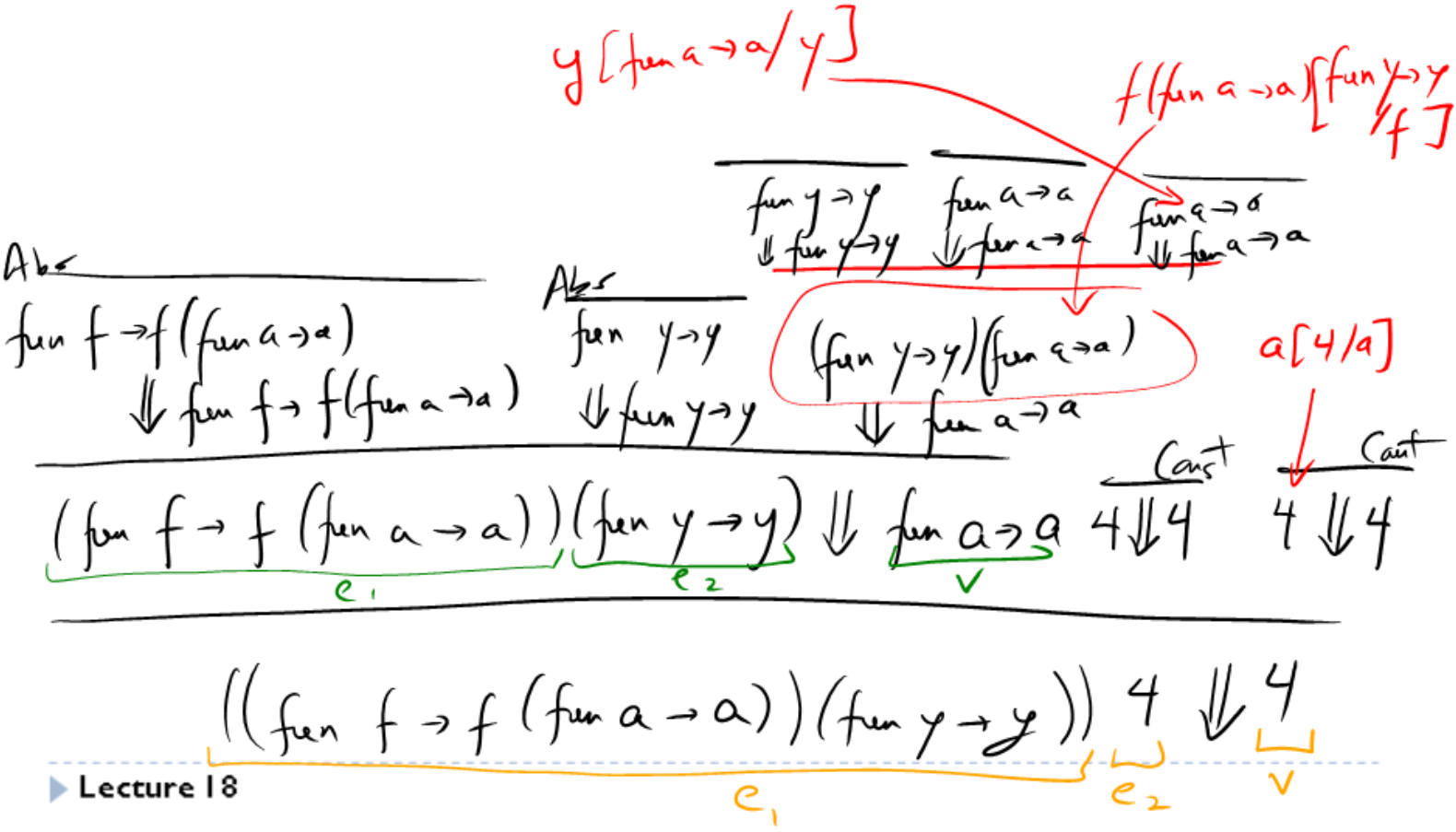
Example: + (+3 4) 5 ⇓ 12



Example: (fun x -> + x x)(+ 3 4) ⇓ 14



Example: (fun f -> f (fun a -> a))(fun y -> y) 4 ⇓ 4



Notes on OS_{simp}

(1) The structure of the proof tree for $e \Downarrow v$ is *similar* to the structure of e, but not the same. (It would be less similar if our language had recursion.)

(2) However, the proof tree – structure and content – are completely, unambiguously determined by the expression e. There is no intelligence or insight required; building the proof tree is completely mechanical.

Preview of following lectures

Will present more complex and realistic proof systems for type-checking and operational semantics of OCaml.

(1) Type system

- Polymorphism and the special role of "let".
- Type-checking of references (i.e. assignable variables)

(2) Operational semantics

- Handling recursion