# CS 565

## Programming Languages (graduate)

## Spring 2025

Week 8

# Type Systems and Simply-Typed Lambda Calculus

# Types



## Today

- Identify key concepts in type systems:
- Type systems as inductive relations
- Type safety
-

# Ill-Typed Imp+



- Let's weaken IMP's expression language slightly:

```
e  ::= B | N | e * e | e + e
     | true | false | ¬ e | e ∧ e
     | Id | e = e | e < e | e ? e : e
```

- Looks good, we can now write (and evaluate):

```
x * ((y > 3) ? 3 : y)
```

- But we can also write:

```
x * ((3 + (6 ∧ 5)) ? 3 : y)
```

- How do we evaluate this? What's the problem?

# Bad Behaviors



- What constitutes a "bad" expression in our IMP variant?
  * One that adds two booleans: `true + 3` $\longrightarrow$ ?
  * One with a non-boolean conditional: `3 ? x : y` $\longrightarrow$ ?
  * A use of an unassigned variable: `x + y` $\longrightarrow$ ?
- What about Coq?
  * Bad pattern match discriminees: `match 0 with [ ] -> …`
  * Function applied to wrong argument types: `plus 9 minus`
  * Application of non-function: `9 minus`

What about other languages?

Badness is language specific!

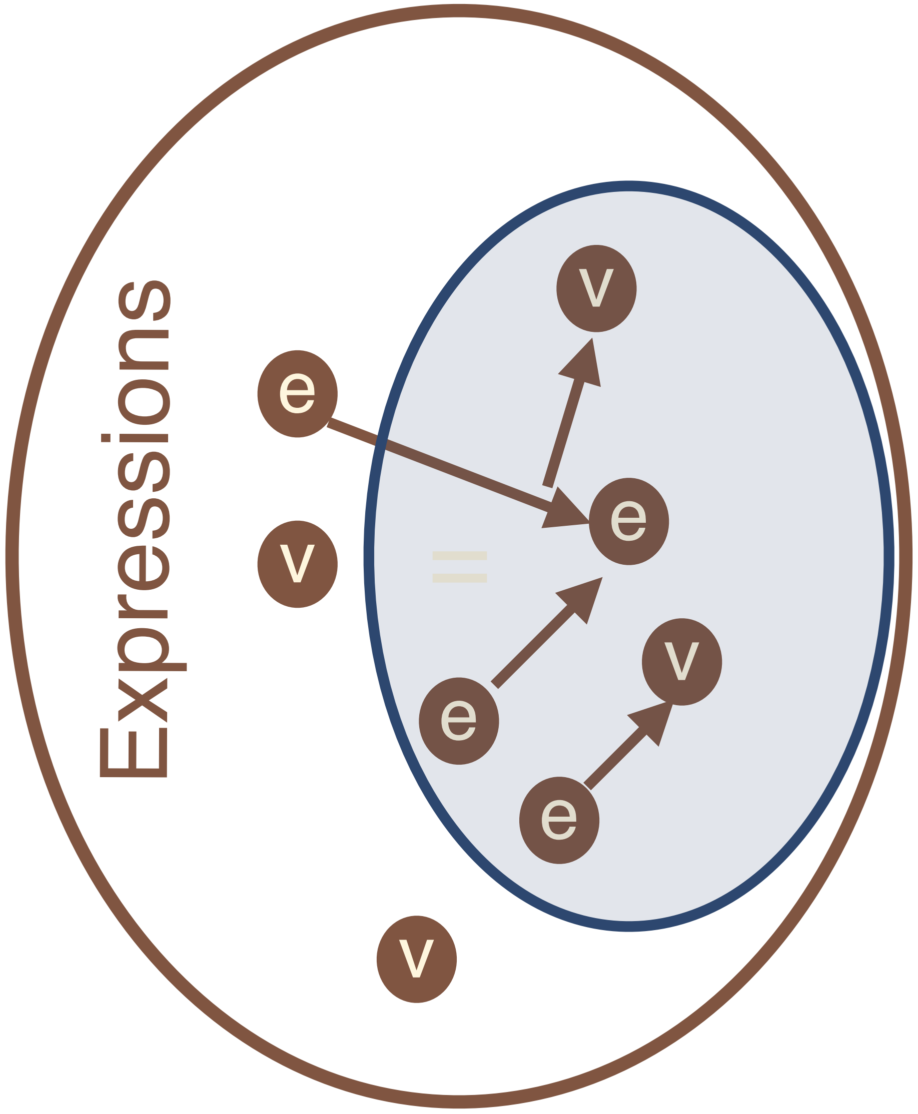
# Static Semantics



A recipe for defining a language:

1. Syntax:
   - What are the valid expressions?
2. Semantics (Dynamic Semantics):
   - How do I evaluate valid expressions?
3. Sanity Checks (Static Semantics):
   - What expressions are "good", i.e have meaningful evaluations?

Type systems identify a subset of good expressions



# Typing Imp$^+$



A recipe for type systems:

1. Define bad programs
2. Define typing rules for classifying programs
3. Show that the type system is sound, i.e. that it only identifies good programs

# Typing Imp$^+$



A recipe for type systems:

1. **Define bad programs**
2. Define typing rules for classifying programs
3. Show that the type system is sound, i.e. that it only identifies good programs

# Typing Imp+



- First step is to define badness:
  - Needs to be broad, program-independent properties
  - Some user-provided specification is okay (type annotations)
- What are bad Imp expressions?

```
3 ? true : 4
true + 3
x * ((y > 3) ? 3 : y)
```

- Those that evaluate to a stuck expression: a normal form that isn't a value

# Typing Imp+



- First step is to define badness:
  - Needs to be broad, program ... erties
  - Some user-p...
    anno...
- What ar...

“Well-typed programs cannot go wrong”

A Theory of Type Polymorphism in Programming (Milner 78)

x * ((y > 3) ? 3 : y)

- Those that evaluate to a stuck expression: a normal form that isn’t a value

# Typing Imp$^{+}$



A recipe for type systems:

1. Define bad programs
2. **Define typing rules for classifying programs**
3. Show that the type system is sound, i.e. that it only identifies good programs

# Typing Rules



Next, define a classifier for good, well-formed programs:

$$\vdash e : T$$

Goal is to classify good uses of each type of expression:

$$\frac{n \in N}{\vdash \mathbf{n} : \text{nat}} \ \text{TNUM} \qquad \frac{\vdash e_1 : \text{nat} \quad \vdash e_2 : \text{nat}}{\vdash e_1 + e_2 : \text{nat}} \ \text{TADD}$$

$$\frac{}{\vdash x : \text{nat}} \ \text{TVAR}$$

$$\frac{\vdash e1 : \text{nat} \quad \vdash e2 : \text{nat}}{\vdash e_1 * e_2 : \text{nat}} \ \text{TMULT}$$

# Typing Rules



Goal is to classify good uses of each type of expression:

$$\frac{}{\vdash \textbf{true} : \text{bool}} \ \text{TT}{\small\text{RUE}}$$

$$\frac{\vdash e : \text{bool}}{\vdash \neg\, e : \text{bool}} \ \text{TN}{\small\text{OT}}$$

$$\frac{}{\vdash \textbf{false} : \text{bool}} \ \text{TF}{\small\text{ALSE}}$$

$$\frac{\vdash e_1 : \text{bool} \quad \vdash e_2 : \text{bool}}{\vdash e_1 \wedge e_2 : \text{bool}} \ \text{TA}{\small\text{ND}}$$

# Typing Rules



Goal is to classify good uses of each type of expression:

$$\frac{\vdash e_1 : \text{nat} \qquad \vdash e_2 : \text{nat}}{\vdash e_1 < e_2 : \text{bool}} \; \textsc{TLe}$$

$$\frac{\vdash e_1 : T \qquad \vdash e_2 : T}{\vdash e_1 = e_2 : \text{bool}} \; \textsc{TEq}$$

$$\frac{\vdash e_1 : \text{bool} \qquad \vdash e_2 : T \qquad \vdash e_3 : T}{\vdash e_1 \;?\; e_2 : e_3 : T} \; \textsc{TCond}$$

# Typing Rules



Goal is to classify good uses of each type of expression:

$$\frac{\vdash e_1 : \text{bool} \qquad \vdash e_2 : T \qquad \vdash e_3 : T}{\vdash e_1 \ ?\ e_2 : e_3 : T}\ \text{TCOND}$$

$$\frac{\vdash e_1 : \text{nat} \qquad \vdash e_2 : \text{nat}}{\vdash e_1 + e_2 : \text{nat}}\ \text{TADD}$$

```
3 ? true : 4
true + 3
⊢ x + ((y > 3) ? true : y)
```

# Typing Imp$^+$



A recipe for type systems:

1. Define bad programs
2. Define a typing rules for classifying programs
3. **Show that the type system is sound, i.e. that it only identifies good programs**

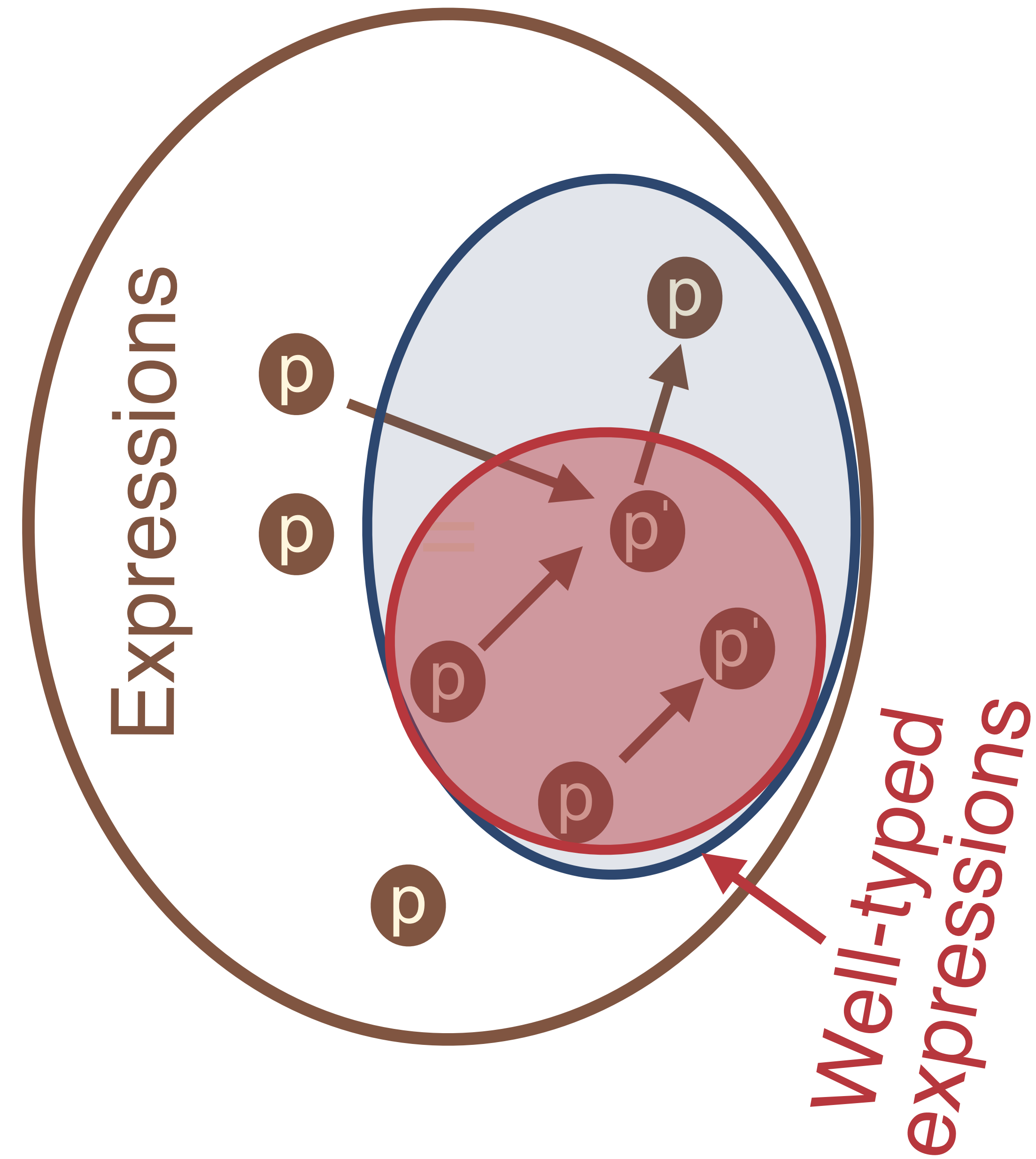
# Type Safety



- When is a type system correct?
  - ★ Need to show this classification is sound. i.e. no false positives:

$$\vdash e : T \quad \rightarrow \quad \sim e \text{ is bad!}$$

- If the a language's type system is sound, it is said to be type-safe.
- Soundness relates provable claims to semantic property



# Progress

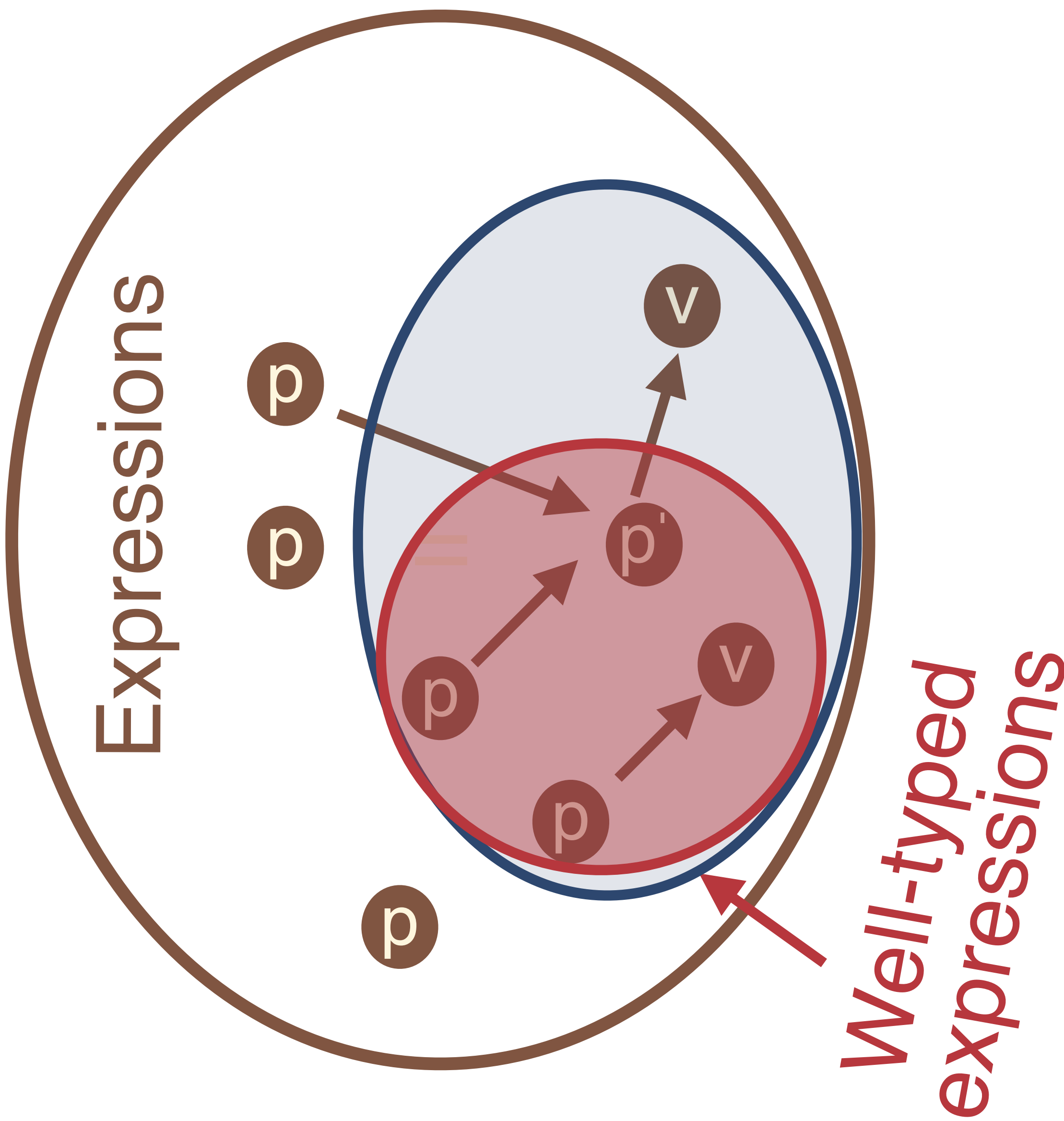


**Theorem** [PROGRESS]: Suppose e is a well-typed expression ($\vdash e:T$). Then either e is a value or there exists some e' such that e evaluates to e' ($\sigma, e \longrightarrow e'$).

Values:

$$\frac{}{\text{value true}}\ \text{TVALUE}$$

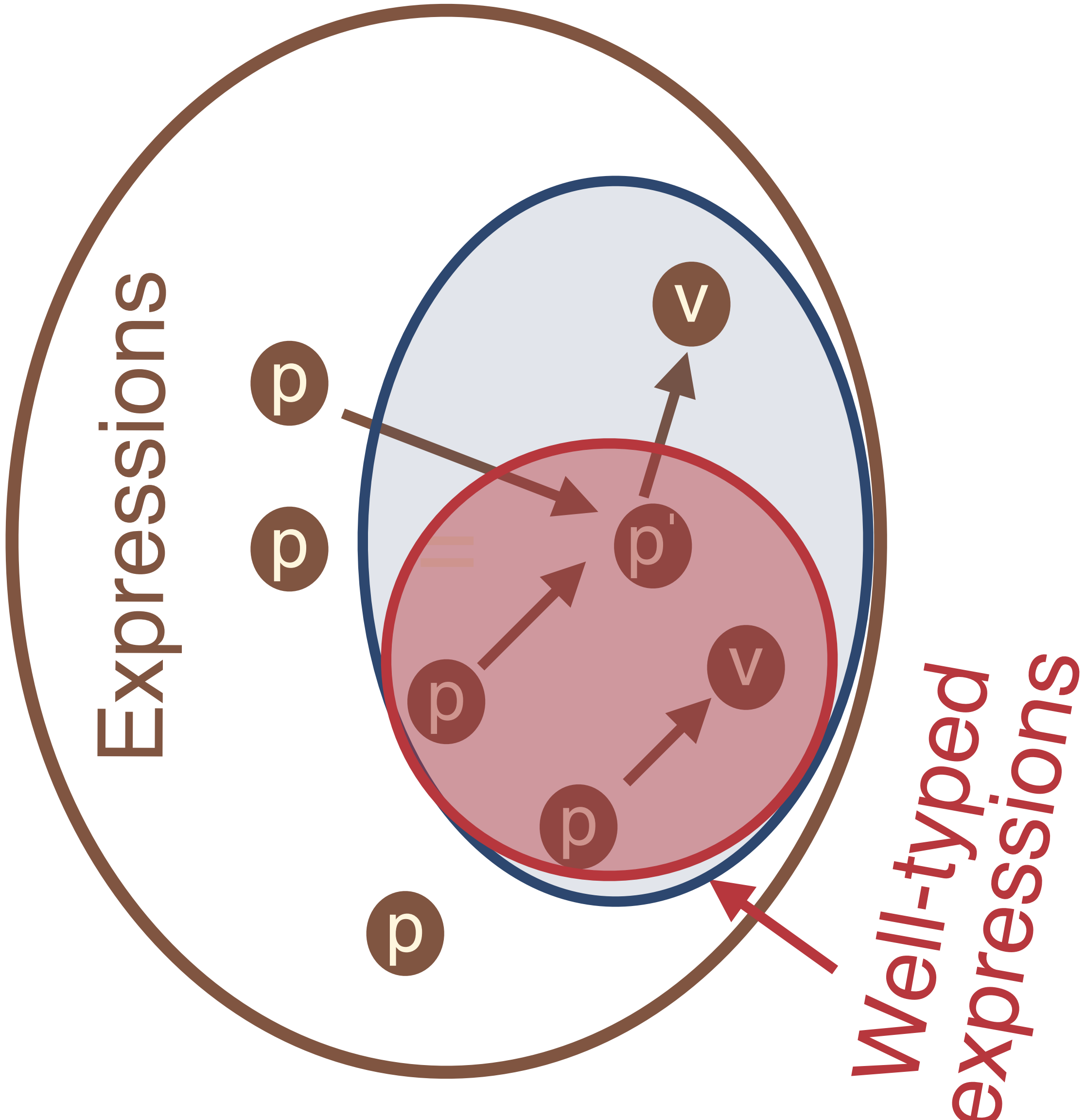
$$\frac{n \in N}{\text{value } n}\ \text{NUMVALUE}$$



# Preservation



- **Theorem** [PRESERVATION]: Suppose e is a well-typed term ($\vdash$ e : T). Then, if e evaluates to e', e' is also a well-typed term under the empty context, with the same type as e ($\vdash$ e' : T).



# Type Soundness



Theorem [Type Soundness]: If an expression e has type T, and e reduces to e' in zero or more steps, then e' is not a stuck term.

**Proof.**

By induction on σ, e $\longrightarrow^*$ e'…

Qed.

- Corollary [Normalization]: If an expression e has type T, e reduces to a value in zero or more steps.

# Example



$$\frac{\vdash e_1 : \text{bool} \qquad \vdash e_2 : \text{nat}}{\vdash e_1 + e_2 : \text{nat}} \ \text{TBADD}$$

# Example



$$\frac{}{0 \ ?\ e_1 : e_2 \longrightarrow e_1}$$

# Example



$$\frac{}{0 \; ? \; e_1 : e_2 \longrightarrow e_1}$$

$$\frac{\vdash e_1 : \text{nat} \qquad \vdash e_2 : T \qquad \vdash e_3 : T}{\vdash e_1 \; ? \; e_2 : e_3 : T} \; \text{TC}_{\text{OND}_2}$$

# Recap



- Type systems classify semantically meaningful expressions
- Our recipe for defining a type system

  1. Define bad states (irreducible, non-value expressions )
  2. Define a typing judgement and rules classifying good expressions ($\vdash e : T$)
  3. Show that the type system is sound, i.e. that good expressions don't reduce to bad states

# Simply-Typed Lambda Calculus



- A language with constants (numbers)
- Function abstraction (variables introduced as function arguments)
- Function application

(The text also considers Booleans, and conditionals)

- What are bad states for terms in this language?
  - Applying a non-function to an argument: λy. 1 y
  - Adding a function: (λy.y) + 1
  - Terms with free variables? x 1

# Typing STLC



- We first define the syntax of terms
- Updated Syntax: (notice that functions (also known as abstractions) have their types annotated)

```
T ::= T → T | nat
n ∈ ℕ

t ::= x | λx : T. t  | t t  | n | t + 1
```

$$\frac{\text{value } t_1 \quad t_2 \longrightarrow t_2'}{t_1\ t_2 \longrightarrow t_1\ t_2'}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}$$

$$\frac{\text{value } t_2}{(\lambda x{:}T.\ t_1)\ t_2 \longrightarrow [x{:=}t_2]t_1}$$

...

$$\frac{n \in N}{\text{value } n}$$

$$\frac{}{\text{value } (\lambda x{:}T.t)}$$

# Typing STLC



$$\Gamma \vdash t : T$$

Γ maps bound variables to their types

- Here are the typing rules:

$$\frac{}{\Gamma \vdash n : nat} \text{ TNUM}$$

$$\frac{\Gamma[x \mapsto T_1] \vdash t : T_2}{\Gamma \vdash \lambda x{:}T_1.t : T_1 \rightarrow T_2} \text{ TABS}$$

$$\frac{\Gamma \vdash t : nat}{\Gamma \vdash t+1 : nat} \text{ TINC}$$

$$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2} \text{ TAPP}$$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \text{ TVAR}$$

# Concept Check



- Can you type this term:

$$((\lambda x{:}\square.x)\ (\lambda x{:}\square.\lambda y{:}\square.y\ x))\ 1\ (\lambda x\ {:}\square.x)$$

- Can you type $(\lambda y\ {:}\square.x\ y)$?
- What about $\Omega$: $(\lambda x\ {:}\square.x\ x)\ (\lambda x\ {:}\square.x\ x)$?

# Type Soundness



- **Theorem** [TYPE SOUNDNESS]: If an STLC term t has type T in the empty context, and t reduces to t' in zero or more steps, either t' is a value, or it can be reduced further (i.e. t' isn't a stuck term).
- This is an example of a **metatheory** proof.
  - The prefix meta- (μετα) means 'beyond' in Greek.
- **theory**: noun | the·o·ry | ˈthē-ə-rē: the general or abstract principles of a body of fact or a science.
- In this sense, a type system is a theory for deducing whether a program is well-formed.
- Properties of that theory are thus meta-theoretic properties

# Progress



- **Theorem** [PROGRESS]: Suppose t is a closed, well-typed term (i.e. $\vdash$ t : T). Then either t is a value or there exists some t' such that t evaluates to t'.
- Proof relies on following lemmas:
- **Lemma** [CANONICAL FORM OF NAT]: If t has type nat in the empty context and t is a value, then t is a number.
- **Lemma** [CANONICAL FORM OF ARROW]: If t has type T -> T in the empty context and t is a value, then t is a lambda abstraction.

# Preservation



- **Theorem** [PRESERVATION]: Suppose t is a well-typed term under the empty context (i.e. ⊢ t : T). Then, if t evaluates to t', t' is also a well-typed term under the empty context, with the same type as t.
- Proof relies on following Lemma:
- **Lemma** [PRESERVATION OF TYPES UNDER SUBSTITUTION]: Suppose t is a well-typed term under context Γ[x↦S] (Γ[x↦S] ⊢ t : T). Then, if s is a well-typed term under Γ with type S, t[x↦s] is a well-typed term under context Γ with type T ( Γ⊢ t[x↦s] : T).

# Normalization



- **Theorem** [NORMALIZATION]: If an expression e has type T in the empty context, e reduces to a value in zero or more steps.

Why is STLC normalizing but not IMP?

# STLC+Pairs



- Updated Syntax:

```
T ::= T → T | nat | T * T
t ::= x   | N
          | λx : T. t
          | t t
          | (t,t)
          | fst t
          | snd t
```

# STLC+Pairs



- Updated Semantics:

$$\frac{t_1 \longrightarrow t_1'}{(t_1,\ t_2) \longrightarrow (t_1',\ t_2)} \qquad \frac{\text{value } t_1 \quad t_2 \longrightarrow t_2'}{(t_1,\ t_2) \longrightarrow (t_1,\ t_2')}$$

$$\frac{t_1 \longrightarrow t_1'}{\text{fst } t_1 \longrightarrow \text{fst } t_1'} \qquad \frac{\text{value } t_1 \quad \text{value } t_2}{\text{fst } (t_1,\ t_2) \longrightarrow t_1}$$

$$\frac{t_1 \longrightarrow t_1'}{\text{snd } t_1 \longrightarrow \text{snd } t_1'} \qquad \frac{\text{value } t_1 \quad \text{value } t_2}{\text{fst } (t_1,\ t_2) \longrightarrow t_2}$$

$$\frac{\text{value } t_1 \quad \text{value } t_2}{\text{value } (t_1,\ t_2)}$$

# STLC+Pairs



- Updated Typing Rules:

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash (t_1, t_2) : T_1 * T_2} \quad \text{TPAIR}$$

$$\frac{\Gamma \vdash t_1 : T_1 * T_2}{\Gamma \vdash \text{fst } t_1 : T_1} \quad \text{TFST}$$

$$\frac{\Gamma \vdash t_1 : T_1 * T_2}{\Gamma \vdash \text{snd } t_1 : T_2} \quad \text{TSND}$$

# STLC+Sums



- Updated Syntax:

```
T ::= ...   | T + T
t ::= ...   | inL T t
            | inR T t
            | case t of
                inL x => t
              | inR x => t
```

$$\frac{\text{value } t_1}{\text{value } in_L\ T\ t_1} \qquad \frac{\text{value } t_1}{\text{value } in_R\ T\ t_1}$$

# STLC+Sums



- Updated Semantics:

$$\frac{t_1 \longrightarrow t_1'}{in_L\ T\ t_1 \longrightarrow in_L\ T\ t_1'} \qquad \frac{t_1 \longrightarrow t_1'}{in_R\ T\ t_1 \longrightarrow in_R\ T\ t_1'}$$

$$\frac{t \longrightarrow t'}{\text{case } t \text{ of } in_L\ x \Rightarrow t_1 \mid in_R\ x \Rightarrow t_2 \longrightarrow \text{case } t' \text{ of } in_L\ x \Rightarrow t_1 \mid in_R\ x \Rightarrow t_2}$$

$$\frac{\text{value } t}{\text{case } in_L\ T\ t \text{ of } in_L\ x \Rightarrow t_1 \mid in_R\ x \Rightarrow t_2 \longrightarrow [x{:=}t]t_1}$$

$$\frac{\text{value } t}{\text{case } in_R\ T\ t \text{ of } in_L\ x \Rightarrow t_1 \mid in_R\ x \Rightarrow t_2 \longrightarrow [x{:=}t]t_2}$$

# STLC+Sums



- Updated Typing Rules:

$$\frac{\Gamma \vdash t : T_1}{\Gamma \vdash in_L\ T_2\ t : T_1 + T_2} \quad \text{TIN}_L$$

$$\frac{\Gamma \vdash t : T_2}{\Gamma \vdash in_R\ T_1\ t : T_1 + T_2} \quad \text{TIN}_L$$

$$\frac{\Gamma \vdash t : T_1 + T_2 \qquad \Gamma[x \mapsto T_1] \vdash t_1 : T_3 \qquad \Gamma[x \mapsto T_2] \vdash t_2 : T_3}{\Gamma \vdash \text{case } t \text{ of } in_L\ x \Rightarrow t_1 \mid in_R\ x \Rightarrow t_2 : T_3} \quad \text{TCASE}$$

# STLC+Fix



- Updated Syntax:

  t ::= … | fix t

- Updated Semantics:

$$\frac{t_1 \longrightarrow t_1'}{\text{fix } t_1 \longrightarrow \text{fix } t_1'}$$

$$\frac{}{\text{fix } (\lambda x{:}T.t_1) \longrightarrow [x{:=}\text{fix } (\lambda x{:}T.t_1)]t_1}$$

# STLC+Fix



let F = (\f. \x. test x=0 then 1 else x * (f (pred x))) in (fix F) 3

$\longrightarrow$ (\x. test x=0 then 1 else x * (fix F (pred x))) 3

$\longrightarrow$ test 3=0 then 1 else 3 * (fix F (pred 3))

$\longrightarrow$ 3 * (fix F (pred 3))

$\longrightarrow$ 3 * ((\x. test x=0 then 1 else x * (fix F (pred x))) (pred 3))

$\longrightarrow$ 3 * ((\x. test x=0 then 1 else x * (fix F (pred x))) 2)

$\longrightarrow$ 3 * test 2=0 then 1 else 2 * (fix F (pred 2))

$\longrightarrow$ 3 * 2 * (fix F (pred 2))

$\longrightarrow^*$ 3 * 2 * 1 * 1

# STLC+Fix



- Updated Typing Rules:

$$\frac{\Gamma \vdash t : T_1 \to T_1}{\Gamma \vdash \text{fix } t : T_1} \quad \text{TFix}$$

# STLC+Records



- Updated Syntax:

$$T ::= \ldots \quad | \quad \{i_1{:}T_1,\ \ldots,\ i_n{:}T_n\}$$

$$t ::= \ldots \quad | \quad \{i_1{=}t_1,\ \ldots,\ i_n{=}t_n\}$$

$$| \quad t.i$$

$$\frac{\text{value } t_1 \quad \ldots \quad \text{value } t_n}{\text{value } \{i_1{=}t_1,\ \ldots,\ i_n{=}t_n\}}$$

# STLC+Records



- Updated Semantics:

$$\frac{\text{value } t_1 \quad \ldots \quad \text{value } t_{m-1} \qquad t_m \longrightarrow t_m'}{\{i_1{=}t_1, \ldots, i_m{=}t_m, \ldots, i_n{=}t_n\} \longrightarrow \{i_1{=}t_1, \ldots, i_m{=}t_m', \ldots, i_n{=}t_n\}}$$

$$\frac{t \longrightarrow t'}{t.i \longrightarrow t'.i}$$

$$\frac{\text{value } t_1 \quad \ldots \quad \text{value } t_n}{\{i_1{=}t_1, \ldots, i_n{=}t_n\}.i_j \longrightarrow t_j}$$

# STLC+Records



- Updated Typing Rules:

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2 \qquad \ldots \qquad \Gamma \vdash t_n : T_n}{\Gamma \vdash \{i_1{=}t_1,\ \ldots,\ i_n{=}t_n\} : \{i_1{:}T_1,\ \ldots,\ i_n{:}T_n\}} \quad \text{TRCD}$$

$$\frac{\Gamma \vdash t : \{i_1{:}T_1,\ \ldots,\ i_n{:}T_n\}}{\Gamma \vdash t.i_j : T_j} \quad \text{TPROJ}$$

# The Limitations of $F_1$ (STLC)



- In $F_1$ each function works exactly for one type
- Example: the identity function
  - id = $\lambda x{:}\tau.\ x : \tau \to \tau$
  - We need to write one version for each type
  - Even more important: sort : $(\tau \to \tau \to \text{bool}) \to \tau\ \text{array} \to \text{unit}$
- The various sorting functions differ only in typing
  - At runtime they perform exactly the same operations
  - We need different versions only to keep the type checker happy
- Two alternatives:
  - Circumvent the type system (see C, Java, ...), or
  - Use a more flexible type system that lets us write only one sorting function

# Polymorphism



- Informal definition

  A function is polymorphic if it can be applied to "many" types of arguments

- Various kinds of polymorphism depending on the definition of "many"
  - subtype (or bounded) polymorphism

    "many" = all subtypes of a given type
  - ad-hoc polymorphism

    "many" = depends on the function

    choose behavior at runtime (depending on types, e.g. sizeof)
  - parametric predicative polymorphism

    "many" = all monomorphic types
  - parametric impredicative polymorphism

    "many" = all types