# 编程语言的设计原理
# Design Principles of Programming Languages

Haiyan Zhao, Di Wang

赵海燕，王迪

Peking University, Spring Term 2023

# $\lambda_{\to}$

*Syntax*

| | | |
|---|---|---|
| t ::= | | *terms:* |
| | x | *variable* |
| | $\lambda x:T.t$ | *abstraction* |
| | t t | *application* |
| v ::= | | *values:* |
| | $\lambda x:T.t$ | *abstraction value* |
| T ::= | | *types:* |
| | $T \to T$ | *type of functions* |
| $\Gamma$ ::= | | *contexts:* |
| | $\varnothing$ | *empty context* |
| | $\Gamma, x:T$ | *term variable binding* |

Assume: all variables in $\Gamma$ are different via renaming/internal

*Evaluation* $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-App2)}$$

$$(\lambda x:T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(E-AppAbs)}$$

*Typing* $\boxed{\Gamma \vdash t : T}$

$$\frac{x:T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-Var)}$$

$$\frac{\Gamma, x:T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x:T_1.t_2 : T_1 \to T_2} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-App)}$$

# Chapter 11: Simply Extensions

Basic Types / The Unit Type

Derived Forms: Sequencing and Wildcard

Ascription / Let Binding

Pairs / Tuples/Records

Sums / Variants

General Recursion / Lists

# Base Types

- Up to now, we've formulated "*base types*" (e.g. Nat) by adding them to the syntax of types, extending the *syntax of terms* with associated constants (zero) and operators (succ, etc.) and adding appropriate *typing and evaluation rules*.
- We can do this for **as many base types as** we like.
- For more theoretical discussions (as opposed to programming) we can often *ignore the term-level inhabitants* of base types, and just treat these types as uninterpreted constants.
  - E.g., suppose B and C are some base types. Then we can ask (without knowing anything more about B or C) whether there are any types S and T such that the term

    $$(\lambda f{:}\,S.\ \lambda g{:}\,T.\ f\ g)\ (\lambda x{:}\,B.\ x)$$

    is well typed.

# Base Types

- Base types in every programming language
  - sets of simple, unstructured *values* such as numbers, Booleans, or characters, and
  - primitive operations for manipulating these values.
- Theoretically, our language is equipped with some *uninterpreted base (atomic) types, with no primitive operations on them at all.*

*New syntactic forms*

| | | | |
|---|---|---|---|
| T | ::= | ... | *types:* |
| | | A | *base type* |

Using A, B, C, … both as the *names* of base types and *metavariables* ranging over base types, relying on context to tell us which is intended in a particular instance.

# Base Types

- Identity function

  λx:A. x;

  <fun>: A → A

  λx:B. x;

  <fun>: B → B

- Function repeating  the behavior of function f on argument x two times

  λf: A → A. λx: A. f (f(x))

  <fun>: (A → A) → A → A

# The Unit Type

- It is the singleton type (like void in C).

*New syntactic forms*

| | | | |
|---|---|---|---|
| t | ::= | ... | *terms:* |
| | | `unit` | *constant* `unit` |
| v | ::= | ... | *values:* |
| | | `unit` | *constant* ***`unit`*** |
| T | ::= | ... | *types:* |
| | | `Unit` | *unit type* |

*New typing rules*

$$\Gamma \vdash \texttt{unit} : \texttt{Unit}$$

$\Gamma \vdash \texttt{t} : \texttt{T}$

(T-UNIT)

*New derived forms*

$$\texttt{t}_1;\texttt{t}_2 \stackrel{\text{def}}{=} (\lambda \texttt{x:Unit}.\texttt{t}_2)\ \texttt{t}_1 \quad \text{where } \texttt{x} \notin FV(\texttt{t}_2)$$

- Application: Unit-type expressions care more about "side effects" rather than "results".

# Derived Form: Sequencing $t_1$; $t_2$

- A direct extension $\lambda^E$
  - t ::= …

    $t_1$; $t_2$
  - New evaluation relation rules

$$\frac{t_1 \longrightarrow t_1'}{t_1;t_2 \longrightarrow t_1';t_2} \qquad \text{(E-Seq)}$$

$$\texttt{unit};t_2 \longrightarrow t_2 \qquad \text{(E-SeqNext)}$$

  - New typing rules

$$\frac{\Gamma \vdash t_1 : \texttt{Unit} \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1;t_2 : T_2} \qquad \text{(T-Seq)}$$

# Derived Form: Sequencing $t_1 ; t_2$

- Derived form ($\lambda^I$): syntactic sugar

$$t_1 ; t_2 \stackrel{\text{def}}{=} (\lambda x{:}\text{Unit}.\, t_2)\ t_1 \quad \text{where } x \notin FV(t_2)$$

- **Theorem** [Sequencing is a derived form]:

  Let $e \in \lambda^E \to \lambda^I$

  be the *elaboration function* (desugaring) that translates from the external to the internal language by replacing every

  occurrence of $t_1; t_2$ with $(\lambda x{:}\,\text{Unit}.\, t_2)\ t_1$.

$$t \longrightarrow_E t' \text{ iff } e(t) \longrightarrow_I e(t')$$

$$\Gamma \vdash^E t : T \text{ iff } \Gamma \vdash^I e(t) : T$$

# Derived Form: Wildcard

- A derived form

$$\lambda\_ : S.\, t \longrightarrow \lambda x : S.\, t$$

  where $x$ is some variable not occurring in $t$.

- Useful in writing a *"dummy" lambda abstraction* in which the *parameter variable* is *not actually used in the body* of the abstraction.

# Ascription

- t as T
  - the ability to explicitly ascribe *a particular type* to *a given term*
  - checking if the term t has the type T, useful for
    - documentation and pinpointing error sources
    - controlling type printing
    - specializing types (after learning subtyping)

# Ascription

*New syntactic forms*

$$\begin{array}{lll} \mathtt{t} & ::= & \ldots \\ & & \mathtt{t\ as\ T} \end{array}$$

*terms*

*ascription*

*New evaluation rules*

$$\mathtt{v_1\ as\ T} \longrightarrow \mathtt{v_1} \qquad \text{(E-Ascribe)}$$

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{t_1'}}{\mathtt{t_1\ as\ T} \longrightarrow \mathtt{t_1'\ as\ T}} \qquad \text{(E-Ascribe1)}$$

*New typing rules*

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{T}}{\Gamma \vdash \mathtt{t_1\ as\ T} : \mathtt{T}} \qquad \text{(T-Ascribe)}$$

verification

*Ascription as a derived form*

$$\mathtt{t\ as\ T} \stackrel{\text{def}}{=} (\lambda \mathtt{x{:}T.\ x})\ \mathtt{t}$$

# Let Bindings

- To give names to some of its subexpressions.

*New syntactic forms*

$$\begin{array}{lll} \mathtt{t} & ::= & \ldots \qquad\qquad\qquad\qquad\qquad\qquad \textit{terms} \\ & & \mathtt{let\ x{=}t\ in\ t} \qquad\qquad\qquad\quad \textit{let binding} \end{array}$$

*New evaluation rules*

$$\mathtt{let\ x{=}v_1\ in\ t_2} \longrightarrow [\mathtt{x} \mapsto \mathtt{v_1}]\mathtt{t_2} \qquad \text{(E-LetV)}$$

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{t_1'}}{\mathtt{let\ x{=}t_1\ in\ t_2} \longrightarrow \mathtt{let\ x{=}t_1'\ in\ t_2}} \qquad \text{(E-Let)}$$

*New typing rules*

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{T_1} \qquad \Gamma, \mathtt{x{:}T_1} \vdash \mathtt{t_2} : \mathtt{T_2}}{\Gamma \vdash \mathtt{let\ x{=}t_1\ in\ t_2} : \mathtt{T_2}} \qquad \text{(T-Let)}$$

# Let Bindings

- Is “let binding” a derived form?

  Yes? $\quad \text{let } x = t_1 \text{ in } t_2 \longrightarrow (\lambda x{:}T_1.\ t_2)\ t_1$

- Desugaring is not on *terms* but *on typing derivations*

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash t_1 : T_1} \qquad \dfrac{\vdots}{\Gamma, x{:}T_1 \vdash t_2 : T_2}}{\Gamma \vdash \texttt{let } x{=}t_1 \texttt{ in } t_2 : T_2}\ \text{T-LET}$$

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, x{:}T_1 \vdash t_2 : T_2}}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1 \to T_2}\ \text{T-ABS} \qquad \dfrac{\vdots}{\Gamma \vdash t_1 : T_1}}{\Gamma \vdash (\lambda x{:}T_1.t_2)\ t_1 : T_2}\ \text{T-APP}$$

# Pairs, tuples, and records
# - Compound data structures -

# Pairs

| | | | |
|---|---|---|---|
| t | ::= | ... | *terms* |
| | | {t,t} | *pair* |
| | | t.1 | *first projection* |
| | | t.2 | *second projection* |
| v | ::= | ... | *values* |
| | | {v,v} | *pair value* |
| T | ::= | ... | *types* |
| | | $T_1 \times T_2$ | *product type* |

# Evaluation rules for pairs

$$\{v_1, v_2\}.1 \longrightarrow v_1 \qquad \text{(E-PairBeta1)}$$

$$\{v_1, v_2\}.2 \longrightarrow v_2 \qquad \text{(E-PairBeta2)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1.1 \longrightarrow t_1'.1} \qquad \text{(E-Proj1)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1.2 \longrightarrow t_1'.2} \qquad \text{(E-Proj2)}$$

$$\frac{t_1 \longrightarrow t_1'}{\{t_1, t_2\} \longrightarrow \{t_1', t_2\}} \qquad \text{(E-Pair1)}$$

$$\frac{t_2 \longrightarrow t_2'}{\{v_1, t_2\} \longrightarrow \{v_1, t_2'\}} \qquad \text{(E-Pair2)}$$

# Evaluation rules for pairs

- examples

```
    {pred 4, if true then false else false}.1
⟶   {3, if true then false else false}.1
⟶   {3, false}.1
⟶   3
```

```
    (λx:Nat×Nat. x.2) {pred 4, pred 5}
⟶   (λx:Nat×Nat. x.2) {3, pred 5}
⟶   (λx:Nat×Nat. x.2) {3,4}
⟶   {3,4}.2
⟶   4
```

# Typing rules for pairs

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash \{t_1, t_2\} : T_1 \times T_2} \quad \text{(T-Pair)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \times T_{12}}{\Gamma \vdash t_1.1 : T_{11}} \quad \text{(T-Proj1)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \times T_{12}}{\Gamma \vdash t_1.2 : T_{12}} \quad \text{(T-Proj2)}$$

# Tuples

- Generalization: binary ➔ n-ary products

*New syntactic forms*

| | | | |
|---|---|---|---|
| t | ::= | ... | *terms:* |
| | | $\{t_i{}^{i\in 1..n}\}$ | *tuple* |
| | | $t.i$ | *projection* |
| v | ::= | ... | *values:* |
| | | $\{v_i{}^{i\in 1..n}\}$ | *tuple value* |
| T | ::= | ... | *types:* |
| | | $\{T_i{}^{i\in 1..n}\}$ | *tuple type* |

*New evaluation rules*

$t \longrightarrow t'$

$$\{v_i{}^{i\in 1..n}\}.j \longrightarrow v_j \qquad \text{(E-ProjTuple)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1.i \longrightarrow t_1'.i} \qquad \text{(E-Proj)}$$

$$\frac{t_j \longrightarrow t_j'}{\{v_i{}^{i\in 1..j-1}, t_j, t_k{}^{k\in j+1..n}\} \longrightarrow \{v_i{}^{i\in 1..j-1}, t_j', t_k{}^{k\in j+1..n}\}} \qquad \text{(E-Tuple)}$$

*New typing rules*

$\Gamma \vdash t : T$

$$\frac{\text{for each } i \quad \Gamma \vdash t_i : T_i}{\Gamma \vdash \{t_i{}^{i\in 1..n}\} : \{T_i{}^{i\in 1..n}\}} \qquad \text{(T-Tuple)}$$

$$\frac{\Gamma \vdash t_1 : \{T_i{}^{i\in 1..n}\}}{\Gamma \vdash t_1.j : T_j} \qquad \text{(T-Proj)}$$

# Records

- Generalization: n-ary products ➔ labeled records

*New syntactic forms*

| | | | |
|---|---|---|---|
| t | ::= | ... | *terms:* |
| | | $\{l_i=t_i \ ^{i\in 1..n}\}$ | *record* |
| | | $t.l$ | *projection* |
| v | ::= | ... | *values:* |
| | | $\{l_i=v_i \ ^{i\in 1..n}\}$ | *record value* |
| T | ::= | ... | *types:* |
| | | $\{l_i:T_i \ ^{i\in 1..n}\}$ | *type of records* |

*New evaluation rules* $\boxed{t \longrightarrow t'}$

$$\{l_i=v_i \ ^{i\in 1..n}\}.l_j \longrightarrow v_j \qquad \text{(E-ProjRcd)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1.l \longrightarrow t_1'.l} \qquad \text{(E-Proj)}$$

$$\frac{t_j \longrightarrow t_j'}{\{l_i=v_i \ ^{i\in 1..j-1}, l_j=t_j, l_k=t_k \ ^{k\in j+1..n}\} \longrightarrow \{l_i=v_i \ ^{i\in 1..j-1}, l_j=t_j', l_k=t_k \ ^{k\in j+1..n}\}} \qquad \text{(E-Rcd)}$$

*New typing rules* $\boxed{\Gamma \vdash t : T}$

$$\frac{\text{for each } i \quad \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i=t_i \ ^{i\in 1..n}\} : \{l_i:T_i \ ^{i\in 1..n}\}} \qquad \text{(T-Rcd)}$$

$$\frac{\Gamma \vdash t_1 : \{l_i:T_i \ ^{i\in 1..n}\}}{\Gamma \vdash t_1.l_j : T_j} \qquad \text{(T-Proj)}$$

*Question*: {partno=5524, cost=30.27} = {cost=30.27, partno=5524}?

# Sums and variants

# Sums

- To deal with *heterogeneous collections* of values.
- e.g., Address books

```
PhysicalAddr = {firstlast:String, addr:String}
VirtualAddr  = {name:String, email:String}
Addr         = PhysicalAddr + VirtualAddr
```

  - Injection by ***tagging*** (disjoint unions)

```
inl  :  "PhysicalAddr → PhysicalAddr+VirtualAddr"
inr  :  "VirtualAddr → PhysicalAddr+VirtualAddr"
```

  - Processing by *case* analysis

```
getName = λa:Addr.
  case a of
    inl x ⇒ x.firstlast
  | inr y ⇒ y.name;
```

# Sums

- To deal with heterogeneous collections of values.

  *New syntactic forms*

| | | | |
|---|---|---|---|
| t | ::= | ... | *terms* |
| | | `inl t` | *tagging (left)* |
| | | `inr t` | *tagging (right)* |
| | | `case t of inl x⇒t \| inr x⇒t` | *case* |
| v | ::= | ... | *values* |
| | | `inl v` | *tagged value (left)* |
| | | `inr v` | *tagged value (right)* |
| T | ::= | ... | *types* |
| | | `T+T` | *sum type* |

$T_1+T_2$ is a *disjoint union* of $T_1$ and $T_2$ (the tags inl and inr ensure disjointness)

# Sums

*New evaluation rules*

$$\texttt{case (inl } v_0\texttt{) of inl } x_1 \Rightarrow t_1 \texttt{ | inr } x_2 \Rightarrow t_2 \longrightarrow [x_1 \mapsto v_0]t_1 \quad \text{(E-CaseInl)}$$

$$\texttt{case (inr } v_0\texttt{) of inl } x_1 \Rightarrow t_1 \texttt{ | inr } x_2 \Rightarrow t_2 \longrightarrow [x_2 \mapsto v_0]t_2 \quad \text{(E-CaseInr)}$$

$$\frac{t_0 \longrightarrow t_0'}{\texttt{case } t_0 \texttt{ of inl } x_1 \Rightarrow t_1 \texttt{ | inr } x_2 \Rightarrow t_2 \longrightarrow \texttt{case } t_0' \texttt{ of inl } x_1 \Rightarrow t_1 \texttt{ | inr } x_2 \Rightarrow t_2} \quad \text{(E-Case)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{inl } t_1 \longrightarrow \texttt{inl } t_1'} \quad \text{(E-Inl)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{inr } t_1 \longrightarrow \texttt{inr } t_1'} \quad \text{(E-Inr)}$$

# Sums (with Unique Typing)

*New typing rules*

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \texttt{inl}\ t_1 : T_1+T_2} \quad \text{(T-Inl)}$$

$$\frac{\Gamma \vdash t_1 : T_2}{\Gamma \vdash \texttt{inr}\ t_1 : T_1+T_2} \quad \text{(T-Inr)}$$

$$\frac{\Gamma \vdash t_0 : T_1+T_2 \qquad \Gamma, x_1:T_1 \vdash t_1 : T \qquad \Gamma, x_2:T_2 \vdash t_2 : T}{\Gamma \vdash \texttt{case}\ t_0\ \texttt{of}\ \texttt{inl}\ x_1 \Rightarrow t_1\ \mid\ \texttt{inr}\ x_2 \Rightarrow t_2 : T} \quad \text{(T-Case)}$$

# Sums and Uniqueness of Types

- Problem

  *If t has type T, then inl t has type T + U for every U.*

  the uniqueness of types is broken, a lot of *U.*

- Possible solutions
  - "Infer" *U* as needed during typechecking
  - Give constructors different names and only allow each name to appear in one sum type (requires generalization to "variants") — OCaml's solution
  - Annotate each *inl* and *inr* with the intended sum type (Figure 11-10)

# Variants

- Generalization: Sums ➔ Labeled variants
  - $T_1 + T_2$ ➔ $<l_1:T_1, l_2:T_2>$
  - inl t as $T_1 + T_2$ ➔ $< l_1 = t >$ as $<l_1:T_1, l_2:T_2>$
- Example:

```
Addr = <physical:PhysicalAddr, virtual:VirtualAddr>;
a = <physical=pa> as Addr;
▸ a : Addr

getName = λa:Addr.
  case a of
    <physical=x> ⇒ x.firstlast
  | <virtual=y> ⇒ y.name;
▸ getName : Addr → String
```

# Variants

*New syntactic forms*

$$
\begin{array}{lll}
t & ::= & \ldots \qquad \textit{terms:} \\
 & & \langle l{=}t \rangle \text{ as } T \qquad \textit{tagging} \\
 & & \text{case } t \text{ of } \langle l_i{=}x_i \rangle \Rightarrow t_i \;^{i \in 1..n} \qquad \textit{case} \\
 & & \\
T & ::= & \ldots \qquad \textit{types:} \\
 & & \langle l_i : T_i \;^{i \in 1..n} \rangle \qquad \textit{type of variants}
\end{array}
$$

*New evaluation rules* $\boxed{t \longrightarrow t'}$

$$\text{case } (\langle l_j{=}v_j \rangle \text{ as } T) \text{ of } \langle l_i{=}x_i \rangle \Rightarrow t_i \;^{i \in 1..n} \longrightarrow [x_j \mapsto v_j]t_j$$

(E-CaseVariant)

$$\frac{t_0 \longrightarrow t_0'}{\text{case } t_0 \text{ of } \langle l_i{=}x_i \rangle \Rightarrow t_i \;^{i \in 1..n} \longrightarrow \text{case } t_0' \text{ of } \langle l_i{=}x_i \rangle \Rightarrow t_i \;^{i \in 1..n}}$$

(E-Case)

$$\frac{t_i \longrightarrow t_i'}{\langle l_i{=}t_i \rangle \text{ as } T \longrightarrow \langle l_i{=}t_i' \rangle \text{ as } T}$$

(E-Variant)

*New typing rules* $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash t_j : T_j}{\Gamma \vdash \langle l_j{=}t_j \rangle \text{ as } \langle l_i : T_i \;^{i \in 1..n} \rangle : \langle l_i : T_i \;^{i \in 1..n} \rangle}$$

(T-Variant)

$$\frac{\Gamma \vdash t_0 : \langle l_i : T_i \;^{i \in 1..n} \rangle \qquad \text{for each } i \quad \Gamma, x_i : T_i \vdash t_i : T}{\Gamma \vdash \text{case } t_0 \text{ of } \langle l_i{=}x_i \rangle \Rightarrow t_i \;^{i \in 1..n} : T}$$

(T-Case)

# Special Instances of Variants

- Options

  OptionalNat = <none: Unit, some: Nat>;

- Enumerations

  Weekday = <monday: Unit, tuesday: Unit, wednesday: Unit,
  thursday:Unit, friday: Unit>;

- Single-Field Variants

  V = <I: T>

  — Operations on T cannot be applied to elements of V without first unpackaging them: a V cannot be accidentally mistaken for a T

# Recursion

# Recursions in $\lambda_{\rightarrow}$

- In simply typed lambda-calculus $\lambda_{\rightarrow}$, all programs terminate.
- Hence, untyped terms like omega and fix are not typable.
- We can extend the system with a (typed) fixed-point operator...

# Example

```
ff = λie:Nat→Bool.
       λx:Nat.
         if iszero x then true
         else if iszero (pred x) then false
         else ie (pred (pred x));

iseven = fix ff;

iseven 7;
```

- What types for ff and iseven ?

  ff : (Nat → Bool) → Nat → Bool

  iseven Nat → Bool

# General Recursions

- Introduce “fix” operator : fix f = f (fix f)
  - It cannot be defined as a derived form in simply typed lambda calculus

*New syntactic forms*

$$\begin{array}{lll} t & ::= & \ldots \qquad\qquad \textit{terms} \\ & & \texttt{fix}\ t \qquad\quad \textit{fixed point of } t \end{array}$$

*New evaluation rules*

$$\texttt{fix}\ (\lambda x{:}T_1.t_2) \longrightarrow [x \mapsto (\texttt{fix}\ (\lambda x{:}T_1.t_2))]t_2 \qquad \text{(E-FixBeta)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{fix}\ t_1 \longrightarrow \texttt{fix}\ t_1'} \qquad \text{(E-Fix)}$$

# General Recursions

*New typing rules*

$$\boxed{\Gamma \vdash t : T}$$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_1}{\Gamma \vdash \mathtt{fix}\ t_1 : T_1} \quad \text{(T-Fix)}$$

# General Recursions

- Another example:

```
ff = λieio:{iseven:Nat→Bool, isodd:Nat→Bool}.
       {iseven = λx:Nat.
                    if iszero x then true
                    else ieio.isodd (pred x),
        isodd = λx:Nat.
                    if iszero x then false
                    else ieio.iseven (pred x)};

▸ ff : {iseven:Nat→Bool,isodd:Nat→Bool} →
       {iseven:Nat→Bool, isodd:Nat→Bool}

r = fix ff;

▸ r : {iseven:Nat→Bool, isodd:Nat→Bool}

iseven = r.iseven;

▸ iseven : Nat → Bool

iseven 7;

▸ false : Bool
```

# General Recursions

- One more example: Given any type T, can you define a term that has type T?

x as T

fix (λx:T. x)

```
diverge_T = λ_:Unit. fix (λx:T.x);
▸ diverge_T : Unit → T
```

# General Recursions

- *A convenient form*

$$\texttt{letrec } x{:}T_1{=}t_1 \texttt{ in } t_2 \stackrel{\text{def}}{=} \texttt{let } x = \texttt{fix } (\lambda x{:}T_1.t_1) \texttt{ in } t_2$$

```
letrec iseven : Nat→Bool =
  λx:Nat.
    if iszero x then true
    else if iszero (pred x) then false
    else iseven (pred (pred x))
in
  iseven 7;
```

# Lists

*New syntactic forms*

| | | | |
|---|---|---|---|
| t | ::= | ... | *terms:* |
| | | `nil[T]` | *empty list* |
| | | `cons[T] t t` | *list constructor* |
| | | `isnil[T] t` | *test for empty list* |
| | | `head[T] t` | *head of a list* |
| | | `tail[T] t` | *tail of a list* |
| v | ::= | ... | *values:* |
| | | `nil[T]` | *empty list* |
| | | `cons[T] v v` | *list constructor* |
| T | ::= | ... | *types:* |
| | | `List T` | *type of lists* |

*New evaluation rules* $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{cons[T]}\ t_1\ t_2 \longrightarrow \texttt{cons[T]}\ t_1'\ t_2} \quad \text{(E-Cons1)}$$

$$\frac{t_2 \longrightarrow t_2'}{\texttt{cons[T]}\ v_1\ t_2 \longrightarrow \texttt{cons[T]}\ v_1\ t_2'} \quad \text{(E-Cons2)}$$

$$\texttt{isnil[S] (nil[T])} \longrightarrow \texttt{true} \quad \text{(E-IsnilNil)}$$

$$\texttt{isnil[S] (cons[T]}\ v_1\ v_2\texttt{)} \longrightarrow \texttt{false} \quad \text{(E-IsnilCons)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{isnil[T]}\ t_1 \longrightarrow \texttt{isnil[T]}\ t_1'} \quad \text{(E-Isnil)}$$

$$\texttt{head[S] (cons[T]}\ v_1\ v_2\texttt{)} \longrightarrow v_1 \quad \text{(E-HeadCons)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{head[T]}\ t_1 \longrightarrow \texttt{head[T]}\ t_1'} \quad \text{(E-Head)}$$

$$\texttt{tail[S] (cons[T]}\ v_1\ v_2\texttt{)} \longrightarrow v_2 \quad \text{(E-TailCons)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{tail[T]}\ t_1 \longrightarrow \texttt{tail[T]}\ t_1'} \quad \text{(E-Tail)}$$

*New typing rules* $\boxed{\Gamma \vdash t : T}$

$$\Gamma \vdash \texttt{nil [}T_1\texttt{]} : \texttt{List}\ T_1 \quad \text{(T-Nil)}$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : \texttt{List}\ T_1}{\Gamma \vdash \texttt{cons[}T_1\texttt{]}\ t_1\ t_2 : \texttt{List}\ T_1} \quad \text{(T-Cons)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{List}\ T_{11}}{\Gamma \vdash \texttt{isnil[}T_{11}\texttt{]}\ t_1 : \texttt{Bool}} \quad \text{(T-Isnil)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{List}\ T_{11}}{\Gamma \vdash \texttt{head[}T_{11}\texttt{]}\ t_1 : T_{11}} \quad \text{(T-Head)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{List}\ T_{11}}{\Gamma \vdash \texttt{tail[}T_{11}\texttt{]}\ t_1 : \texttt{List}\ T_{11}} \quad \text{(T-Tail)}$$

# Homework ☺

- Read Chapter 11.
- Do Exercise 11.5.2 & 11.12.1

11.5.2 EXERCISE [⋆⋆]: Another way of defining `let` as a derived form might be to desugar it by "executing" it immediately—i.e., to regard `let x=`$t_1$ `in` $t_2$ as an abbreviation for the substituted body $[x \mapsto t_1]t_2$. Is this a good idea? □

11.12.1 EXERCISE [⋆⋆⋆]: Verify that the progress and preservation theorems hold for the simply typed lambda-calculus with booleans and lists. □