



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 2026



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, 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 ::= ...
A

types:
base type

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

Base Types

- Identity function

$\lambda x:A. x;$

$\langle \text{fun} \rangle: A \rightarrow A$

$\lambda x:B. x;$

$\langle \text{fun} \rangle: B \rightarrow B$

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

$\lambda f: A \rightarrow A. \lambda x: A. f (f (x))$

$\langle \text{fun} \rangle: (A \rightarrow A) \rightarrow A \rightarrow A$

The Unit Type

- The singleton type (like void in C).

New syntactic forms

$t ::= \dots$ *terms:*
unit *constant unit*

$v ::= \dots$ *values:*
unit *constant unit*

$T ::= \dots$ *types:*
Unit *unit type*

New typing rules

$$\Gamma \vdash t : T$$

$\Gamma \vdash \text{unit} : \text{Unit}$ (T-UNIT)

New derived forms

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

- Unit-type expressions **care more about** “*side effects*” rather than “*results*” (assignment to a reference cell)
 - unit is the only possible result** of evaluating an expression of type Unit



Derived Form: Sequencing $t_1; t_2$

- A direct extension λ^E

– $t ::= \dots$

$t_1; t_2$

- New evaluation relation rules

$$\frac{t_1 \rightarrow t'_1}{t_1; t_2 \rightarrow t'_1; t_2}$$

(E-SEQ)

$$\text{unit}; t_2 \rightarrow t_2$$

(E-SEQNEXT)

- New typing rules

$$\frac{\Gamma \vdash t_1 : \text{Unit} \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1; t_2 : T_2}$$

(T-SEQ)

Derived Form: Sequencing $t_1 ; t_2$

- Derived form: syntactic sugar

$$t_1 ; t_2 \stackrel{\text{def}}{=} (\lambda x : \text{Unit}. t_2) t_1$$

where $x \notin FV(t_2)$

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

Let $e \in \lambda^E \rightarrow \lambda^I$ be the *elaboration function* (desugaring)

that translates from the external to the internal language by replacing *every occurrence* of $t_1 ; t_2$ with $(\lambda x : \text{Unit}. t_2) t_1$.

$$t \rightarrow_E t' \text{ iff } e(t) \rightarrow_I e(t')$$

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



Derived Form: Wildcard

- A derived form

$$\lambda_ : S. t \rightarrow \lambda x : S. t$$

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 T* to *a given term t*
 - 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

$$t ::= \dots$$

$$t \text{ as } T$$

terms
ascription

New evaluation rules

$$v_1 \text{ as } T \longrightarrow v_1 \quad (\text{E-ASCRIIBE})$$

$$\frac{t_1 \longrightarrow t'_1}{t_1 \text{ as } T \longrightarrow t'_1 \text{ as } T} \quad (\text{E-ASCRIIBE1})$$

New typing rules

$$\frac{\Gamma \vdash t_1 : T}{\Gamma \vdash t_1 \text{ as } T : T} \quad (\text{T-ASCRIIBE})$$

verification

Ascription as a derived form

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

Let Bindings

- To *give names* to some of its *subexpressions*.

New syntactic forms

$$t ::= \dots$$

$$\text{let } \boxed{x=t} \text{ in } \boxed{t}$$

terms

let binding

New evaluation rules

$$\frac{\text{let } x=v_1 \text{ in } t_2 \longrightarrow [x \mapsto v_1]t_2 \quad t_1 \longrightarrow t'_1}{\text{let } x=t_1 \text{ in } t_2 \longrightarrow \text{let } x=t'_1 \text{ in } t_2}$$

(E-LETV)

(E-LET)

New typing rules

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma, x:T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x=t_1 \text{ in } t_2 : T_2}$$

(T-LET)

Let Bindings

- Is “let binding” a derived form?

Yes?

$$\text{let } x = t_1 \text{ in } t_2 \rightarrow (\lambda x:T_1. t_2) t_1$$

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

$$\begin{array}{c}
 \vdots \\
 \hline
 \Gamma \vdash t_1 : T_1
 \end{array}
 \quad
 \begin{array}{c}
 \vdots \\
 \hline
 \Gamma, x:T_1 \vdash t_2 : T_2
 \end{array}
 \quad
 \text{T-LET}
 \quad
 \hline
 \Gamma \vdash \text{let } x=t_1 \text{ in } t_2 : T_2$$

↓

$$\begin{array}{c}
 \vdots \\
 \hline
 \Gamma, x:T_1 \vdash t_2 : T_2
 \end{array}
 \quad
 \text{T-ABS}
 \quad
 \begin{array}{c}
 \vdots \\
 \hline
 \Gamma \vdash t_1 : T_1
 \end{array}
 \quad
 \text{T-APP}
 \quad
 \hline
 \Gamma \vdash (\lambda x:T_1. t_2) t_1 : T_2$$



Pairs, Tuples, and Records

- Compound data structures -

Pairs



$t ::= \dots$
 $\{t, t\}$
 $t.1$
 $t.2$

terms
pair
first projection
second projection

$v ::= \dots$
 $\{v, v\}$

values
pair value

$T ::= \dots$
 $T_1 \times T_2$

types
product type

Evaluation rules for pairs

$$\{v_1, v_2\}.1 \longrightarrow v_1 \quad (\text{E-PAIRBETA1})$$

$$\{v_1, v_2\}.2 \longrightarrow v_2 \quad (\text{E-PAIRBETA2})$$

$$\frac{t_1 \longrightarrow t'_1}{t_1.1 \longrightarrow t'_1.1} \quad (\text{E-PROJ1})$$

$$\frac{t_1 \longrightarrow t'_1}{t_1.2 \longrightarrow t'_1.2} \quad (\text{E-PROJ2})$$

$$\frac{t_1 \longrightarrow t'_1}{\{t_1, t_2\} \longrightarrow \{t'_1, t_2\}} \quad (\text{E-PAIR1})$$

$$\frac{t_2 \longrightarrow t'_2}{\{v_1, t_2\} \longrightarrow \{v_1, t'_2\}} \quad (\text{E-PAIR2})$$

Evaluation rules for pairs

- examples

$\{ \text{pred } 4, \text{ if true then false else false} \}.1$
 $\rightarrow \{3, \text{ if true then false else false} \}.1$
 $\rightarrow \{3, \text{ false} \}.1$
 $\rightarrow 3$

$(\lambda x:\text{Nat} \times \text{Nat}. x.2) \{ \text{pred } 4, \text{ pred } 5 \}$
 $\rightarrow (\lambda x:\text{Nat} \times \text{Nat}. x.2) \{3, \text{ pred } 5 \}$
 $\rightarrow (\lambda x:\text{Nat} \times \text{Nat}. x.2) \{3, 4 \}$
 $\rightarrow \{3, 4 \}.2$
 $\rightarrow 4$



Typing rules for pairs

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

$$\frac{\Gamma \vdash t_1 : T_{11} \times T_{12}}{\Gamma \vdash t_1.1 : T_{11}} \quad (\text{T-PROJ1})$$

$$\frac{\Gamma \vdash t_1 : T_{11} \times T_{12}}{\Gamma \vdash t_1.2 : T_{12}} \quad (\text{T-PROJ2})$$

Tuples

- Generalization: binary \rightarrow n-ary products

New syntactic forms

$t ::= \dots$
 $\{t_i^{i \in 1..n}\}$
 $t.i$

$v ::= \dots$
 $\{v_i^{i \in 1..n}\}$

$T ::= \dots$
 $\{T_i^{i \in 1..n}\}$

New evaluation rules

$\{v_i^{i \in 1..n}\}.j \rightarrow v_j$

terms:
tuple
projection

values:
tuple value

types:
tuple type

$t \rightarrow t'$

(E-PROJTUPLE)

$$\frac{t_1 \rightarrow t'_1}{t_1.i \rightarrow t'_1.i} \quad \text{(E-PROJ)}$$

$$\frac{t_j \rightarrow t'_j}{\{v_i^{i \in 1..j-1}, t_j, t_k^{k \in j+1..n}\} \rightarrow \{v_i^{i \in 1..j-1}, t'_j, t_k^{k \in j+1..n}\}} \quad \text{(E-TUPLE)}$$

$$\Gamma \vdash t : T$$

New typing rules

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

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

Records

- Generalization: n-ary products \rightarrow labeled records

New syntactic forms

$t ::= \dots$
 $\{\lambda_i = t_i \quad i \in 1..n\}$
 $t.l$

terms:
record
projection

$v ::= \dots$
 $\{\lambda_i = v_i \quad i \in 1..n\}$

values:
record value

$T ::= \dots$
 $\{\lambda_i : T_i \quad i \in 1..n\}$

types:
type of records

New evaluation rules

$\{\lambda_i = v_i \quad i \in 1..n\}.l_j \rightarrow v_j$ (E-PROJRCD)

$t \rightarrow t'$

$\frac{t_1 \rightarrow t'_1}{t_1.l \rightarrow t'_1.l}$ (E-PROJ)

$\frac{t_j \rightarrow t'_j}{\{\lambda_i = v_i \quad i \in 1..j-1, \lambda_j = t_j, \lambda_k = t_k \quad k \in j+1..n\} \rightarrow \{\lambda_i = v_i \quad i \in 1..j-1, \lambda_j = t'_j, \lambda_k = t_k \quad k \in j+1..n\}}$ (E-RCD)

New typing rules

$\Gamma \vdash t : T$

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

$\frac{\Gamma \vdash t_1 : \{\lambda_i : T_i \quad i \in 1..n\}}{\Gamma \vdash t_1.l_j : T_j}$ (T-PROJ)

Question: $\{\text{partno}=5524, \text{cost}=30.27\} = \{\text{cost}=30.27, \text{partno}=5524\}$?



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

$t ::= \dots$ $\text{inl } t$ $\text{inr } t$ $\text{case } t \text{ of } \text{inl } x \Rightarrow t \mid \text{inr } x \Rightarrow t$	<i>terms</i> <i>tagging (left)</i> <i>tagging (right)</i> <i>case</i>
$v ::= \dots$ $\text{inl } v$ $\text{inr } v$	<i>values</i> <i>tagged value (left)</i> <i>tagged value (right)</i>
$T ::= \dots$ $T+T$	<i>types</i> <i>sum type</i>

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

New evaluation rules

$$\begin{array}{l} \text{case (inl } v_0) \\ \text{of inl } x_1 \Rightarrow t_1 \mid \text{inr } x_2 \Rightarrow t_2 \end{array} \longrightarrow [x_1 \mapsto v_0]t_1 \quad (\text{E-CASEINL})$$
$$\begin{array}{l} \text{case (inr } v_0) \\ \text{of inl } x_1 \Rightarrow t_1 \mid \text{inr } x_2 \Rightarrow t_2 \end{array} \longrightarrow [x_2 \mapsto v_0]t_2 \quad (\text{E-CASEINR})$$
$$\frac{t_0 \longrightarrow t'_0}{\text{case } t_0 \text{ of inl } x_1 \Rightarrow t_1 \mid \text{inr } x_2 \Rightarrow t_2 \longrightarrow \text{case } t'_0 \text{ of inl } x_1 \Rightarrow t_1 \mid \text{inr } x_2 \Rightarrow t_2} \quad (\text{E-CASE})$$
$$\frac{t_1 \longrightarrow t'_1}{\text{inl } t_1 \longrightarrow \text{inl } t'_1} \quad (\text{E-INL})$$
$$\frac{t_1 \longrightarrow t'_1}{\text{inr } t_1 \longrightarrow \text{inr } t'_1} \quad (\text{E-INR})$$

Sums (with Unique Typing)

New typing rules

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

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

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



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 \rightarrow Labeled variants
 - $T_1 + T_2 \rightarrow \langle l_1:T_1, l_2:T_2 \rangle$
 - $\text{inl } t \text{ as } T_1 + T_2 \rightarrow \langle l_1 = t \rangle \text{ as } \langle l_1:T_1, l_2:T_2 \rangle$
- Example:

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

 - ▶ `a : Addr`

```
getName =  $\lambda a:\text{Addr}.$ 
  case a of
    <physical=x>  $\Rightarrow$  x.firstlast
  | <virtual=y>  $\Rightarrow$  y.name;
```

 - ▶ `getName : Addr \rightarrow String`

Variants

New syntactic forms

$t ::= \dots$ *terms:*
 $\langle l=t \rangle \text{ as } T$ *tagging*
 $\text{case } t \text{ of } \langle l_i=x_i \rangle \Rightarrow t_i^{i \in 1..n}$ *case*

$T ::= \dots$ *types:*
 $\langle l_i : T_i^{i \in 1..n} \rangle$ *type of variants*

New evaluation rules

$$\boxed{t \rightarrow t'}$$

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

$$\frac{t_0 \rightarrow t'_0}{\text{case } t_0 \text{ of } \langle l_i=x_i \rangle \Rightarrow t_i^{i \in 1..n} \rightarrow \text{case } t'_0 \text{ of } \langle l_i=x_i \rangle \Rightarrow t_i^{i \in 1..n}}$$
(E-CASE)

$$\frac{t_i \rightarrow t'_i}{\langle l_i=t_i \rangle \text{ as } T \rightarrow \langle l_i=t'_i \rangle \text{ as } T}$$
(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)

$$\frac{\Gamma \vdash t_0 : \langle l_i : T_i^{i \in 1..n} \rangle \quad \text{for each } i \quad \Gamma, x_i : T_i \vdash t_i : T}{\Gamma \vdash \text{case } t_0 \text{ of } \langle l_i=x_i \rangle \Rightarrow t_i^{i \in 1..n} : T}$$
(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 (with just one single lable)

$V = \langle l: T \rangle$

- 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 λ_{\rightarrow}

- In simply typed lambda-calculus λ_{\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

- What type for **fix**?

General Recursions

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

New syntactic forms

$t ::= \dots$
 $\text{fix } t$

terms
fixed point of t

New evaluation rules

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

$$\frac{t_1 \longrightarrow t'_1}{\text{fix } t_1 \longrightarrow \text{fix } t'_1}$$
 (E-FIX)



General Recursions

New typing rules

$$\boxed{\Gamma \vdash t : T}$$
$$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_1}{\Gamma \vdash \text{fix } t_1 : T_1}$$

(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)};
```

▶ $ff : \{iseven:Nat \rightarrow Bool, isodd:Nat \rightarrow Bool\} \rightarrow \{iseven:Nat \rightarrow Bool, isodd:Nat \rightarrow Bool\}$

```
r = fix ff;
```

▶ $r : \{iseven:Nat \rightarrow Bool, isodd:Nat \rightarrow Bool\}$

```
iseven = r.iseven;
```

▶ $iseven : Nat \rightarrow Bool$

```
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

$\text{fix } (\lambda x:T. x)$

$\text{diverge}_T = \lambda_:\text{Unit}. \text{fix } (\lambda x:T.x);$

▶ $\text{diverge}_T : \text{Unit} \rightarrow T$

General Recursions

- *A convenient form*

`letrec x:T1=t1 in t2 $\stackrel{\text{def}}{=} \text{let } x = \text{fix } (\lambda x:T_1.t_1) \text{ in } t_2$`

```
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

$t ::= \dots$
 $\text{nil}[T]$
 $\text{cons}[T] t t$
 $\text{isnil}[T] t$
 $\text{head}[T] t$
 $\text{tail}[T] t$

terms:
 empty list
 list constructor
 test for empty list
 head of a list
 tail of a list

$v ::= \dots$
 $\text{nil}[T]$
 $\text{cons}[T] v v$

values:
 empty list
 list constructor

$T ::= \dots$
 $\text{List } T$

types:
 type of lists

New evaluation rules

$\frac{t_1 \rightarrow t'_1}{\text{cons}[T] t_1 t_2 \rightarrow \text{cons}[T] t'_1 t_2}$ (E-CONS1)

$\frac{t_2 \rightarrow t'_2}{\text{cons}[T] v_1 t_2 \rightarrow \text{cons}[T] v_1 t'_2}$ (E-CONS2)

$\text{isnil}[S] (\text{nil}[T]) \rightarrow \text{true}$ (E-ISNILNIL)

$\text{isnil}[S] (\text{cons}[T] v_1 v_2) \rightarrow \text{false}$ (E-ISNILCONS)

$t \rightarrow t'$

$\frac{t_1 \rightarrow t'_1}{\text{isnil}[T] t_1 \rightarrow \text{isnil}[T] t'_1}$ (E-ISNIL)
 $\text{head}[S] (\text{cons}[T] v_1 v_2) \rightarrow v_1$ (E-HEADCONS)

$\frac{t_1 \rightarrow t'_1}{\text{head}[T] t_1 \rightarrow \text{head}[T] t'_1}$ (E-HEAD)

$\text{tail}[S] (\text{cons}[T] v_1 v_2) \rightarrow v_2$ (E-TAILCONS)

$\frac{t_1 \rightarrow t'_1}{\text{tail}[T] t_1 \rightarrow \text{tail}[T] t'_1}$ (E-TAIL)

New typing rules

$\Gamma \vdash t : T$

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

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

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

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

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



Homework 😊

- Read Chapter 11.
- Do Exercise 11.9.1 & 11.12.1

- 11.9.1 EXERCISE [★★]: Note the similarity between the typing rule for `case` and the rule for `if` in Figure 8-1: `if` can be regarded as a sort of degenerate form of `case` where no information is passed to the branches. Formalize this intuition by defining `true`, `false`, and `if` as derived forms using sums and `Unit`. □
- 11.12.1 EXERCISE [★★★]: Verify that the progress and preservation theorems hold for the simply typed lambda-calculus with booleans and lists. □