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 2025

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	::=	...	*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 which is intended in a particular instance.

Base Types

- Identity function

  $\lambda x{:}A.\ x;$

  <fun>: $A \rightarrow A$

  $\lambda x{:}B.\ x;$

  <fun>: $B \rightarrow B$

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

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

  <fun>: $(A \rightarrow A) \rightarrow A \rightarrow A$

The Unit Type

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

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

New derived forms

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

- Unit-type expressions care more about "side effects" rather than "results"
  - **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 ::= …

    $t_1;\ t_2$
  - New evaluation relation rules

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

$$\text{unit};t_2 \longrightarrow t_2 \quad \text{(E-SEQNEXT)}$$

  - New typing rules

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

Derived Form: Sequencing $t_1 ; t_2$

- Derived form (λ^E): syntactic sugar

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

- **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 $(\lambda x\text{: Unit. } t_2)\ t_1$.

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

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

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 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

$$\texttt{t} ::= \ldots \qquad \textit{terms}$$
$$\texttt{t as T} \qquad \textit{ascription}$$

New evaluation rules

$$\texttt{v}_1 \texttt{ as T} \longrightarrow \texttt{v}_1 \qquad \text{(E-Ascribe)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{t}_1 \texttt{ as T} \longrightarrow \texttt{t}_1' \texttt{ as T}} \qquad \text{(E-Ascribe1)}$$

New typing rules

$$\boxed{\frac{\Gamma \vdash \texttt{t}_1 : \texttt{T}}{\Gamma \vdash \texttt{t}_1 \texttt{ as T} : \texttt{T}} \qquad \text{(T-Ascribe)}}$$

verification

Ascription as a derived form

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

Let Bindings

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

New syntactic forms

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

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

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

Let Bindings

- Is “let binding” a derived form?

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

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

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

$$\Downarrow$$

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

Pairs, Tuples, and Records

- Compound data structures -

Pairs

$\mathtt{t}$	$::=$	$\ldots$	*terms*
		$\boxed{\{\mathtt{t},\mathtt{t}\}}$	*pair*
		$\mathtt{t.1}$	*first projection*
		$\mathtt{t.2}$	*second projection*
$\mathtt{v}$	$::=$	$\ldots$	*values*
		$\{\mathtt{v},\mathtt{v}\}$	*pair value*
$\mathtt{T}$	$::=$	$\ldots$	*types*
		$\boxed{\mathtt{T}_1 \times \mathtt{T}_2}$	*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

$$
\begin{array}{ll}
 & \{\text{pred } 4, \text{ if true then false else false}\}.1 \\
\longrightarrow & \{3, \text{ if true then false else false}\}.1 \\
\longrightarrow & \{3, \text{ false}\}.1 \\
\longrightarrow & 3
\end{array}
$$

$$
\begin{array}{ll}
 & (\lambda x{:}\text{Nat}\times\text{Nat}.\ x.2)\ \{\text{pred } 4, \text{ pred } 5\} \\
\longrightarrow & (\lambda x{:}\text{Nat}\times\text{Nat}.\ x.2)\ \{3, \text{ pred } 5\} \\
\longrightarrow & (\lambda x{:}\text{Nat}\times\text{Nat}.\ x.2)\ \{3,4\} \\
\longrightarrow & \{3,4\}.2 \\
\longrightarrow & 4
\end{array}
$$

Typing rules for pairs

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \{\mathtt{t}_1, \mathtt{t}_2\} : \mathtt{T}_1 \times \mathtt{T}_2} \qquad \text{(T-PAIR)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \times \mathtt{T}_{12}}{\Gamma \vdash \mathtt{t}_1.1 : \mathtt{T}_{11}} \qquad \text{(T-PROJ1)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \times \mathtt{T}_{12}}{\Gamma \vdash \mathtt{t}_1.2 : \mathtt{T}_{12}} \qquad \text{(T-PROJ2)}$$

Tuples

- Generalization: binary ➔ n-ary products

New syntactic forms

$\mathtt{t}$	::=	...	*terms:*
		$\{\mathtt{t}_i\ ^{i\in 1..n}\}$	*tuple*
		$\mathtt{t.i}$	*projection*
$\mathtt{v}$	::=	...	*values:*
		$\{\mathtt{v}_i\ ^{i\in 1..n}\}$	*tuple value*
$\mathtt{T}$	::=	...	*types:*
		$\{\mathtt{T}_i\ ^{i\in 1..n}\}$	*tuple type*

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

$$\{\mathtt{v}_i\ ^{i\in 1..n}\}.\mathtt{j} \longrightarrow \mathtt{v}_j \qquad \text{(E-ProjTuple)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1.\mathtt{i} \longrightarrow \mathtt{t}_1'.\mathtt{i}} \qquad \text{(E-Proj)}$$

$$\frac{\mathtt{t}_j \longrightarrow \mathtt{t}_j'}{\begin{array}{l}\{\mathtt{v}_i\ ^{i\in 1..j-1}, \mathtt{t}_j, \mathtt{t}_k\ ^{k\in j+1..n}\}\\ \longrightarrow \{\mathtt{v}_i\ ^{i\in 1..j-1}, \mathtt{t}_j', \mathtt{t}_k\ ^{k\in j+1..n}\}\end{array}} \qquad \text{(E-Tuple)}$$

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

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

$$\frac{\Gamma \vdash \mathtt{t}_1 : \{\mathtt{T}_i\ ^{i\in 1..n}\}}{\Gamma \vdash \mathtt{t}_1.\mathtt{j} : \mathtt{T}_j} \qquad \text{(T-Proj)}$$

Records

- Generalization: n-ary products ➔ labeled records

New syntactic forms

$\mathtt{t}$	::=	...	*terms:*
		$\{\mathtt{l}_i\mathtt{=t}_i\ ^{i\in 1..n}\}$	*record*
		$\mathtt{t.l}$	*projection*
$\mathtt{v}$	::=	...	*values:*
		$\{\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..n}\}$	*record value*
$\mathtt{T}$	::=	...	*types:*
		$\{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}$	*type of records*

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

$$\{\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..n}\}.\mathtt{l}_j \longrightarrow \mathtt{v}_j \qquad \text{(E-ProjRcd)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1.\mathtt{l} \longrightarrow \mathtt{t}_1'.\mathtt{l}} \qquad \text{(E-Proj)}$$

$$\frac{\mathtt{t}_j \longrightarrow \mathtt{t}_j'}{\begin{array}{l}\{\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..j-1},\mathtt{l}_j\mathtt{=t}_j,\mathtt{l}_k\mathtt{=t}_k\ ^{k\in j+1..n}\}\\ \longrightarrow \{\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..j-1},\mathtt{l}_j\mathtt{=t}_j',\mathtt{l}_k\mathtt{=t}_k\ ^{k\in j+1..n}\}\end{array}} \qquad \text{(E-Rcd)}$$

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

$$\frac{\text{for each } i \quad \Gamma \vdash \mathtt{t}_i : \mathtt{T}_i}{\Gamma \vdash \{\mathtt{l}_i\mathtt{=t}_i\ ^{i\in 1..n}\} : \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}} \qquad \text{(T-Rcd)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}}{\Gamma \vdash \mathtt{t}_1.\mathtt{l}_j : \mathtt{T}_j} \qquad \text{(T-Proj)}$$

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)

$$\text{inl} : \text{“PhysicalAddr} \rightarrow \text{PhysicalAddr+VirtualAddr”}$$
$$\text{inr} : \text{“VirtualAddr} \rightarrow \text{PhysicalAddr+VirtualAddr”}$$

  – 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*
		$\boxed{\texttt{inl t}}$	*tagging (left)*
		$\boxed{\texttt{inr t}}$	*tagging (right)*
		$\boxed{\texttt{case t}}$ `of inl x⇒t	inr x⇒t`	*case*
v	::=	...	*values*
		`inl v`	*tagged value (left)*
		`inr v`	*tagged value (right)*
T	::=	...	*types*
		$\boxed{\texttt{T+T}}$	*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

$$\begin{array}{ll} \texttt{case (inl } \texttt{v}_0\texttt{)} & \longrightarrow [\texttt{x}_1 \mapsto \texttt{v}_0]\texttt{t}_1 \\ \texttt{of inl } \texttt{x}_1 \Rightarrow \texttt{t}_1 \mid \texttt{inr } \texttt{x}_2 \Rightarrow \texttt{t}_2 & \end{array} \quad \text{(E-CaseInl)}$$

$$\begin{array}{ll} \texttt{case (inr } \texttt{v}_0\texttt{)} & \longrightarrow [\texttt{x}_2 \mapsto \texttt{v}_0]\texttt{t}_2 \\ \texttt{of inl } \texttt{x}_1 \Rightarrow \texttt{t}_1 \mid \texttt{inr } \texttt{x}_2 \Rightarrow \texttt{t}_2 & \end{array} \quad \text{(E-CaseInr)}$$

$$\frac{\texttt{t}_0 \longrightarrow \texttt{t}_0'}{\begin{array}{c} \texttt{case } \texttt{t}_0 \texttt{ of inl } \texttt{x}_1 \Rightarrow \texttt{t}_1 \mid \texttt{inr } \texttt{x}_2 \Rightarrow \texttt{t}_2 \\ \longrightarrow \texttt{case } \texttt{t}_0' \texttt{ of inl } \texttt{x}_1 \Rightarrow \texttt{t}_1 \mid \texttt{inr } \texttt{x}_2 \Rightarrow \texttt{t}_2 \end{array}} \quad \text{(E-Case)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{inl } \texttt{t}_1 \longrightarrow \texttt{inl } \texttt{t}_1'} \quad \text{(E-Inl)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{inr } \texttt{t}_1 \longrightarrow \texttt{inr } \texttt{t}_1'} \quad \text{(E-Inr)}$$

Sums (with Unique Typing)

New typing rules

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1}{\Gamma \vdash \mathtt{inl}\ \mathtt{t}_1 : \underline{\mathtt{T}_1\mathtt{+}\mathtt{T}_2}} \quad \text{(T-INL)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_2}{\Gamma \vdash \mathtt{inr}\ \mathtt{t}_1 : \underline{\mathtt{T}_1\mathtt{+}\mathtt{T}_2}} \quad \text{(T-INR)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{t}_0 : \mathtt{T}_1\mathtt{+}\mathtt{T}_2 \\ \Gamma, \mathtt{x}_1\mathtt{:}\mathtt{T}_1 \vdash \mathtt{t}_1 \underline{: \mathtt{T}} \qquad \Gamma, \mathtt{x}_2\mathtt{:}\mathtt{T}_2 \vdash \mathtt{t}_2 \underline{: \mathtt{T}}\end{array}}{\Gamma \vdash \mathtt{case}\ \mathtt{t}_0\ \mathtt{of}\ \mathtt{inl}\ \mathtt{x}_1 \Rightarrow \mathtt{t}_1 \mid \mathtt{inr}\ \mathtt{x}_2 \Rightarrow \mathtt{t}_2 : \underline{\mathtt{T}}} \quad \text{(T-CASE)}$$

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
  - $T_1 + T_2$ ➔ $<l_1{:}T_1, l_2{:}T_2>$
  - $\text{inl } t \text{ as } T_1 + T_2$ ➔ $< l_1 = t > \text{ as } <l_1{:}T_1, l_2{:}T_2>$
- 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

$$\begin{array}{llll}
\mathtt{t} & ::= & \ldots & \textit{terms:} \\
 & & \mathtt{<l=t>\ as\ T} & \textit{tagging} \\
 & & \mathtt{case\ t\ of\ <l_i=x_i>\Rightarrow t_i}\ ^{i\in 1..n} & \textit{case}
\end{array}$$

$$\begin{array}{llll}
\mathtt{T} & ::= & \ldots & \textit{types:} \\
 & & \mathtt{<l_i:T_i}\ ^{i\in 1..n}\mathtt{>} & \textit{type of variants}
\end{array}$$

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

$$\mathtt{case\ (<l_j=v_j>\ as\ T)\ of\ <l_i=x_i>\Rightarrow t_i}\ ^{i\in 1..n} \longrightarrow [\mathtt{x}_j \mapsto \mathtt{v}_j]\mathtt{t}_j \qquad \text{(E-CASEVARIANT)}$$

$$\frac{\mathtt{t}_0 \longrightarrow \mathtt{t}_0'}{\mathtt{case\ t_0\ of\ <l_i=x_i>\Rightarrow t_i}\ ^{i\in 1..n} \longrightarrow \mathtt{case\ t_0'\ of\ <l_i=x_i>\Rightarrow t_i}\ ^{i\in 1..n}} \qquad \text{(E-CASE)}$$

$$\frac{\mathtt{t}_i \longrightarrow \mathtt{t}_i'}{\mathtt{<l_i=t_i>\ as\ T} \longrightarrow \mathtt{<l_i=t_i'>\ as\ T}} \qquad \text{(E-VARIANT)}$$

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

$$\frac{\Gamma \vdash \mathtt{t}_j : \mathtt{T}_j}{\Gamma \vdash \mathtt{<l_j=t_j>\ as\ <l_i:T_i}\ ^{i\in 1..n}\mathtt{>} : \mathtt{<l_i:T_i}\ ^{i\in 1..n}\mathtt{>}} \qquad \text{(T-VARIANT)}$$

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{<l_i:T_i}\ ^{i\in 1..n}\mathtt{>} \qquad \text{for each } i \quad \Gamma, \mathtt{x}_i:\mathtt{T}_i \vdash \mathtt{t}_i : \mathtt{T}}{\Gamma \vdash \mathtt{case\ t_0\ of\ <l_i=x_i>\Rightarrow t_i}\ ^{i\in 1..n} : \mathtt{T}} \qquad \text{(T-CASE)}$$

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 = <l: 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

- What type for fix?

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

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

New evaluation rules

$$\begin{array}{c} \mathtt{fix}\ (\lambda \mathtt{x}{:}\mathtt{T_1}.\mathtt{t_2}) \\ \longrightarrow [\boxed{\mathtt{x} \mapsto (\mathtt{fix}\ (\lambda \mathtt{x}{:}\mathtt{T_1}.\mathtt{t_2}))}]\mathtt{t_2} \end{array} \quad \text{(E-FixBeta)}$$

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{t_1'}}{\mathtt{fix}\ \mathtt{t_1} \longrightarrow \mathtt{fix}\ \mathtt{t_1'}} \quad \text{(E-Fix)}$$

General Recursions

New typing rules

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

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \rightarrow \mathtt{T}_1}{\Gamma \vdash \mathtt{fix}\ \mathtt{t}_1 : \mathtt{T}_1} \qquad \text{(T-FIX)}$$

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)

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

General Recursions

- *A convenient form*

$$\texttt{letrec x:T}_1\texttt{=t}_1\texttt{ in t}_2 \stackrel{\text{def}}{=} \texttt{let x = fix (}\lambda\texttt{x:T}_1\texttt{.t}_1\texttt{) in t}_2$$

```
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

$\mathtt{t}$	::=	...	*terms:*
		$\mathtt{nil[T]}$	*empty list*
		$\mathtt{cons[T]\ t\ t}$	*list constructor*
		$\mathtt{isnil[T]\ t}$	*test for empty list*
		$\mathtt{head[T]\ t}$	*head of a list*
		$\mathtt{tail[T]\ t}$	*tail of a list*
$\mathtt{v}$	::=	...	*values:*
		$\mathtt{nil[T]}$	*empty list*
		$\mathtt{cons[T]\ v\ v}$	*list constructor*
$\mathtt{T}$	::=	...	*types:*
		$\mathtt{List\ T}$	*type of lists*

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

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{cons[T]\ t_1\ t_2} \longrightarrow \mathtt{cons[T]\ t_1'\ t_2}} \quad \text{(E-Cons1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{cons[T]\ v_1\ t_2} \longrightarrow \mathtt{cons[T]\ v_1\ t_2'}} \quad \text{(E-Cons2)}$$

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

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

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{isnil[T]\ t_1} \longrightarrow \mathtt{isnil[T]\ t_1'}} \quad \text{(E-Isnil)}$$

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

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{head[T]\ t_1} \longrightarrow \mathtt{head[T]\ t_1'}} \quad \text{(E-Head)}$$

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

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{tail[T]\ t_1} \longrightarrow \mathtt{tail[T]\ t_1'}} \quad \text{(E-Tail)}$$

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

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

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

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

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

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

Homework ☺

- Read Chapter 11.
- Do Exercise 11.11.1 & 11.12.1