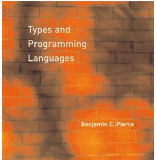
Typed λ-Calculus: Simple extensions

UNIVERSIDADE DA CORUÑA

GRAO EN ENXEÑERÍA INFORMÁTICA

DESEÑO DAS LINGUAXES DE PROGRAMACIÓN

Based on chapter 11 of: Benjamin C. Pierce, *Types and Programming Languages*. The MIT Press, 2002



Basic Types

- Programming languages provide a set of *base types* (ssts of simple, unstructured values such as numbers, booleans and characters) plus appropiate primitive operations for manipulating these values.
- We abstract away from the details of particular base types and their operations, and instead we assume that typed $\lambda_{\rightarrow}$ provides a set A of *uninterpreted* base types, with no primitive operations on them at all.

Unit type

- A single element, the term constant `unit`.
- Its main application is in languages with side effects, such as assignments: It is often the side effect, not the result, of an expression that we care about.
- `Unit` similar to `void` in C or Java.
- Extensions to the language $\lambda_{\rightarrow}$:
  - New syntactic forms:

    ```
    t::= ...
       unit
    v::= ...
       unit
    T::= ...
       Unit
    ```

  - New typing rules:

    $\Gamma \vdash$ `unit: Unit`

Derived forms

- Derived forms $\equiv$ *syntactic sugar*.

- Make easier to read and wite terms
- Can be eliminated without affecting language semantics
- *Desugaring*: replacing a derived form with its lower-level definition.

Derived forms: sequencing

- In languages with side effects, it is often useful to evaluate two or more expressions in sequence $\mathtt{t}_1\mathtt{;}\mathtt{t}_2$
- Evaluating $\mathtt{t}_1$, throwing away its trivial result, and going on to evaluate $\mathtt{t}_2$. Two different ways to formalize sequencing:

1. Add $\mathtt{t}_1\mathtt{;}\mathtt{t}_2$ as a new alternative in the syntax of terms and then add two evaluation rules:

$$\frac{\mathtt{t}_1 \rightarrow \mathtt{t}_1'}{\mathtt{t}_1;\mathtt{t}_2 \rightarrow \mathtt{t}_1';\mathtt{t}_2} \quad \mathtt{E-Seq}$$

$$\mathtt{unit};\mathtt{t}_2 \rightarrow \mathtt{t}_2 \quad \mathtt{E-SeqNext}$$

   and a typing rule:

$$\frac{\Gamma\vdash\mathtt{t}_1:\mathtt{Unit} \qquad \Gamma\vdash\mathtt{t}_2:\mathtt{T}_2}{\Gamma\vdash\mathtt{t}_1;\mathtt{t}_2:\mathtt{T}_2}$$

2. To regard $\mathtt{t}_1\mathtt{;}\mathtt{t}_2$ as an *abbreviation* of $(\lambda\mathtt{x}:\mathtt{Unit}.\mathtt{t}_2)\mathtt{t}_1$ where $\mathtt{x}$ is chosen *fresh*.

Derived forms: wildcards

- A term of the form λ`x:S.t`, where `x` is not used in the body of `t`.
- We would like to replace `x` by a wildcard binder λ`_:S.t`.

Ascription

- To explicitely ascribe a particular type to a given term.
- Syntactic form:

```
t::= ...
   t as T
```

- Evaluation rules (throws away an ascription as soon as it is reached):

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

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

- Typing rule:

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

- Use of ascription: documentation (to improve program readability); debugging (to “hide” some types by telling the typechecker to treat a term as if it had a given type).

Ascription

- **Exercise:** Show how to formulate ascription as a derived form

Ascription

- **Exercise:** Show how to formulate ascription as a derived form

$(\lambda$x:T.x$)$t

  - Following a *call-by-value* evaluation strategy, t will be evaluated to a value before applying the abstraction.
  - According to T-App, the type of t should be T.

let bindings

- Give names to some subexpressions, both for avoiding repetition and for increasing readability.
- Syntax similar to ML:

```
t::= ...
     let x = t in t
```

- Evaluation rules: in *call-by-value*, the let-bound term must be fully evaluated before evaluation of the let-body can begin

$$\texttt{let x} = \texttt{v}_1 \texttt{ in t}_2 \rightarrow [\texttt{x} \mapsto \texttt{v}_1]\texttt{t}_2 \quad (\texttt{E} - \texttt{LetV})$$

$$\frac{\texttt{t}_1 \rightarrow \texttt{t}_1'}{\texttt{let x} = \texttt{t}_1 \texttt{ in t}_2 \rightarrow \texttt{let x} = \texttt{t}_1' \texttt{ in t}_2} \quad (\texttt{E} - \texttt{Let})$$

- Typing rule: calculate the type of the let-bound term, extending the context with a binding with this type, and calculate the type of the body, which is the type of the whole expression

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{T}_1 \qquad \Gamma, \texttt{x} : \texttt{T}_1 \vdash \texttt{t}_2 : \texttt{T}_2}{\Gamma \vdash \texttt{let x} = \texttt{t}_1 \texttt{ in t}_2 : \texttt{T}_2} \quad (\texttt{T} - \texttt{Let})$$

`let` bindings

- `let` can also be defined as a derived form:

  $\texttt{let x = t}_1\ \texttt{in t}_2 \equiv (\lambda\texttt{x:T}_1\texttt{.t}_2)\texttt{t}_1$.
- but...
  - The abbreviation includes the type annotation T_1
  - While *desugaring*, the typechecker must determine T_1
  - Then, a specific typeing rule mist be included in the language

Pairs

- A way of building compound data structures. The new type $\mathsf{T}_1 \times \mathsf{T}_2$ is called a *product*

- Left-to-right evaluation order:

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

- When a pair is used as an argument, it is evaluated before the execution of the abstraction body:

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

- Syntactic rules:

```
t::= ...
      {t,t}
      t.1
      t.2
```

```
v::= ...
      {v,v}
T::= ...
```

$T_1 \times T_2$

- Evaluation rules:

$$\{v_1, v_2\}.1 \to v_1 \quad (\mathrm{E-PairBeta1})$$

$$\{v_1, v_2\}.2 \to v_2 \quad (\mathrm{E-PairBeta2})$$

$$\frac{t_1 \to t_1'}{t_1.1 \to t_1'.1} \quad (\mathrm{E-Proj1})$$

$$\frac{t_1 \to t_1'}{t_1.2 \to t_1'.2} \quad (\mathrm{E-Proj2})$$

$$\frac{t_1 \to t_1'}{\{t_1, t_2\} \to \{t_1', t_2\}} \quad (\mathrm{E-Pair1})$$

$$\frac{t_2 \to t_2'}{\{v_1, t_2\} \to \{v_1, t_2'\}} \quad (\mathrm{E-Pair2})$$

- Typing rules:

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

$$\frac{\Gamma\vdash \mathtt{t}_1\mathtt{:T}_{11}\times\mathtt{T}_{12}}{\Gamma\vdash \mathtt{t}_1\mathtt{.1:T}_{11}} \quad (\mathtt{T}-\mathtt{Proj1})$$

$$\frac{\Gamma\vdash \mathtt{t}_1\mathtt{:T}_{11}\times\mathtt{T}_{12}}{\Gamma\vdash \mathtt{t}_1\mathtt{.2:T}_{12}} \quad (\mathtt{T}-\mathtt{Proj2})$$

Tuples

- A generalization of pairs to n-ary products: $\{t_i{}^{i=1..n}\}:\{T_i{}^{i=1..n}\}$.

  `{1,2,true}:{Nat,Nat,Bool}`.

  `{}` when `n = 0` (empty tuple).

  5 is a `Nat` value

  `{5}` is a `{Nat} value)`.

- New syntactic forms:

$$
\begin{array}{ll}
\texttt{t::=} & \texttt{...} \\
 & \{\texttt{t}_i{}^{i=1..n}\} \\
 & \texttt{t.i}
\end{array}
\qquad
\begin{array}{ll}
\texttt{v::=} & \texttt{...} \\
 & \{\texttt{v}_i{}^{i=1..n}\} \\
\texttt{T::=} & \\
 & \{\texttt{T}_i{}^{i=1..n}\}
\end{array}
$$

- Evaluation rules:

$$\{\mathtt{v}_{\mathtt{i}}^{\mathtt{i=1..n}}\}.\mathtt{j} \rightarrow \mathtt{v}_{\mathtt{j}} \qquad (\mathtt{E-ProjTuple})$$

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

$$\frac{\mathtt{t}_{\mathtt{j}} \rightarrow \mathtt{t}_{\mathtt{j}}'}{\{\mathtt{v}_{\mathtt{i}}^{\mathtt{i=1..j\text{-}1}},\mathtt{t}_{\mathtt{j}},\mathtt{t}_{\mathtt{k}}^{\mathtt{k=j+1..n}}\} \rightarrow \{\mathtt{v}_{\mathtt{i}}^{\mathtt{i=1..j\text{-}1}},\mathtt{t}_{\mathtt{j}}',\mathtt{t}_{\mathtt{k}}^{\mathtt{k=j+1..n}}\}} \qquad (\mathtt{E-Tuple})$$

- Typing rules:

$$\frac{\forall \mathtt{i} \quad \Gamma \vdash \mathtt{t}_{\mathtt{i}} : \mathtt{T}_{\mathtt{i}}}{\Gamma \vdash \{\mathtt{t}_{\mathtt{i}}^{\mathtt{i=1..n}}\} : \{\mathtt{T}_{\mathtt{i}}^{\mathtt{i=1..n}}\}} \qquad (\mathtt{T-Tuple})$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \{\mathtt{T}_{\mathtt{i}}^{\mathtt{i=1..n}}\}}{\Gamma \vdash \mathtt{t}_1.\mathtt{j} : \mathtt{T}_{\mathtt{j}}} \qquad (\mathtt{T-Proj})$$

Records

- A generalization from tuples to labeled records: annotate each field $\mathtt{t_i}$ with a label $\mathtt{l_i}$ (all labels in a given record are distinct)

  `{x=5}` are `{num=5524,const=30.27}` are records of types `{x:Nat}` and `{num:Nat,const:Float}`

- Tuples are a special case of records with labels `1,2,3,...`.

  `{Bool,Nat,Bool}` would be equivalent to `{1:Bool,2:Nat,3:Bool}`.

- Is the order of fields relevant?
  Have `{num=5524,const=30.27}` and `{const=30.27, num=5524}` the same meaning and the same type?

- New syntactic forms:

```
t::= ...
      {li=ti i=1..n}
      t.l
v::= ...
      {li=vi i=1..n}
T::=
      {li:Ti i=1..n}
```

$$
\begin{aligned}
\texttt{t::=}\ & \texttt{...} \\
& \{\texttt{l}_{\mathrm{i}}\texttt{=t}_{\mathrm{i}}{}^{\mathrm{i}=1..\mathrm{n}}\} \\
& \texttt{t.l} \\
\texttt{v::=}\ & \texttt{...} \\
& \{\texttt{l}_{\mathrm{i}}\texttt{=v}_{\mathrm{i}}{}^{\mathrm{i}=1..\mathrm{n}}\} \\
\texttt{T::=}\ & \\
& \{\texttt{l}_{\mathrm{i}}\texttt{:T}_{\mathrm{i}}{}^{\mathrm{i}=1..\mathrm{n}}\}
\end{aligned}
$$

- Evaluation rules:

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

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

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

- Typing rules:

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

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

Sums

- Heterogeneous collections of values
- The simpler case of a binary *sum* type describes a set of differente values drawn from exactly two given types. Example:
  - Two sorts of address records:

    `PhysicalAddr = {firstLast:String,addr:String}`
    `VirtualAddr = {name:String,email:String}`
  - To manipulate them uniformly:

    `Addr = PhysicalAddr + VirtualAddr`
  - `inl` and `inr` "inject" elements into the left and right components of the sum type:

    `inl : PhysicalAddr -> PhysicalAddr+VirtualAddr`
    `inr : VirtualAddr -> PhysicalAddr+VirtualAddr`

- The elements of T_1+T_2 consists of elements of T_1 tagged with `inl` and elements of T_2 tagged with `inr`.
- `case` allows us to distinguish whether a value comes from the left or right branch. Example:

```
getName = λa:Addr.
   case a of
      inl x => x.firstLast
      |inr y => y.name
> getName : Addr→String
```

- New syntactic forms:

```
t::= ...
      inl t
      inr t
      case t of inl x => t | inr x => t
v::= ...
      inl v
      inr v
T::= ...
       T+T
```

- Evaluation rules:

$$\texttt{case (inl } v_0\texttt{) of inl } x_1 \;\texttt{=>}\; t_1 \mid \texttt{inr } x_2 \;\texttt{=>}\; t_2 \to [x_1 \mapsto v_0]t_1 \qquad (\texttt{E} - \texttt{CaseInl})$$

$$\texttt{case (inr } v_0\texttt{) of inl } x_1 \;\texttt{=>}\; t_1 \mid \texttt{inr } x_2 \;\texttt{=>}\; t_2 \to [x_2 \mapsto v_0]t_2 \qquad (\texttt{E} - \texttt{CaseInr})$$

$$\frac{t_0 \to t_0'}{\texttt{case } t_0 \texttt{ of inl } x_1 \texttt{ => } t_1 \mid \texttt{inr } x_2 \texttt{ => } t_2 \to \texttt{case } t_0' \texttt{ of inl } x_1 \texttt{ => } t_1 \mid \texttt{inr } x_2 \texttt{ => } t_2} \qquad (\texttt{E} - \texttt{Case})$$

$$\frac{t_1 \to t_1'}{\texttt{inl } t_1 \to \texttt{inl } t_1} \qquad (\texttt{E} - \texttt{Inl})$$

$$\frac{t_1 \to t_1'}{\texttt{inr } t_1 \to \texttt{inr } t_1} \qquad (\texttt{E} - \texttt{Inr})$$

- Typing rules:

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

$$\frac{\Gamma \vdash t_2 : T_2}{\Gamma \vdash \texttt{inr } t_2 : T_1 + T_2} \qquad (\texttt{T} - \texttt{Inr})$$

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

Sums and uniqueness of types

- Most of the properties of the typing relation of $\lambda_{\rightarrow}$ extend to the system with sums, but one important fails: The Uniquieness of Types. Example:
  - Following `T-Inl`, if $\mathtt{t}_1$:$\mathtt{T}_1$ then `inl` $\mathtt{t}_1$:$\mathtt{T}_1$+$\mathtt{T}_2$ for any $\mathtt{T}_2$.
  - `inl 5` can be of type `Nat+Nat` or `Nat+Bool`.
  - The same occurs for `T-Inr`.
- Possible solutions:
  1. We can complicate the typecheking algorithm so that it can “guess” $\mathtt{T}_2$.
  2. We can refine the language of types to allow all possible $\mathtt{T}_2$ be represented uniformly
  3. **We can demand that the programmer provide an explicit annotation to indicate which type $\mathtt{T}_2$ is intended.**

- New syntax forms:

```
t::= ...
    inl t as T
    inr t as T
    case t of inl x => t | inr x => t
```

```
v::= ...
    inl v as T
    inr v as T
T::= ...
    T+T
```

- Evaluation rules:

$$\texttt{case (inl } v_0 \texttt{ as } T_0\texttt{) of inl } x_1 \Rightarrow t_1 \mid \texttt{inr } x_2 \Rightarrow t_2 \rightarrow [x_1 \mapsto v_0]t_1 \qquad (E-CaseInl)$$

$$\texttt{case (inr } v_0 \texttt{ as } T_0\texttt{) of inl } x_1 \Rightarrow t_1 \mid \texttt{inr } x_2 \Rightarrow t_2 \rightarrow [x_2 \mapsto v_0]t_2 \qquad (E-CaseInr)$$

$$\frac{t_0 \rightarrow t_0'}{\texttt{case } t_0 \texttt{ of inl } x_1 \Rightarrow t_1 \mid \texttt{inr } x_2 \Rightarrow t_2 \rightarrow \texttt{case } t_0' \texttt{ of inl } x_1 \Rightarrow t_1 \mid \texttt{inr } x_2 \Rightarrow t_2} \qquad (E-Case)$$

$$\frac{t_1 \rightarrow t_1'}{\texttt{inl } t_1 \texttt{ as } T \rightarrow \texttt{inl } t_1' \texttt{ asT}} \qquad (E-Inl)$$

$$\frac{t_1 \rightarrow t_1'}{\texttt{inr } t_1 \texttt{ as } T \rightarrow \texttt{inr } t_1' \texttt{asT}} \qquad (E-Inr)$$

- Typing rules:

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

$$\frac{\Gamma \vdash t_2 : T_2}{\Gamma \vdash \texttt{inr } t_2 \texttt{ as } T_1 + T_2 : T_1 + T_2} \qquad (T-Inr)$$

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

Variants

- Binary sums generalize to labeled *variants*.
  - Instead of $\mathtt{T}_1\mathtt{+T}_2$ we write $<\mathtt{l}_1\mathtt{:T}_1\mathtt{,l}_2\mathtt{:T}_2>$.
  - Instead of `inl t as` $\mathtt{T}_1\mathtt{+T}_2$ we write $<\mathtt{l}_1\mathtt{=t}>$ `as` $<\mathtt{l}_1\mathtt{:T}_1\mathtt{,l}_2\mathtt{:T}_2>$.
  - Instead of labeling the branches of the case with `inl` and `inr` we use the same labels as the corresponding sum type

- 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
```

- Sums and Variants are *disjoint unions*.

- New syntax forms:

```
t::= ...
      <l=t> as T
      case t of < l_i=x_i > => t_i^{i=1..n}
T::= ...                            v::= ...
      <l_i:T_i^{i=1..n} >                  <l=v> as T
```

- Evaluation rules:

$$\texttt{case}\ (<\texttt{l}_j = \texttt{v}_j>\ \texttt{as T})\ \texttt{of}\ <\texttt{l}_i = \texttt{x}_i> => \texttt{t}_i^{i=1..n} \ \rightarrow\ [\texttt{x}_j \mapsto \texttt{v}_j]\texttt{t}_j \qquad (\texttt{E} - \texttt{CaseVariant})$$

$$\frac{\texttt{t}_0 \rightarrow \texttt{t}_0'}{\texttt{case t}_0\ \texttt{of}\ <\texttt{l}_i=\texttt{x}_i> => \texttt{t}_i^{i=1..n} \rightarrow \texttt{case t}_0'\ \texttt{of}\ <\texttt{l}_i=\texttt{x}_i> => \texttt{t}_1^{i=1..n}} \qquad (\texttt{E} - \texttt{Case})$$

$$\frac{\texttt{t}_i \rightarrow \texttt{t}_i'}{<\texttt{l}_i=\texttt{t}_i>\ \texttt{as T} \rightarrow <\texttt{l}_i=\texttt{t}_i'>\ \texttt{as T}} \qquad (\texttt{E} - \texttt{Variant})$$

- Typing rules:

$$\frac{\Gamma \vdash \texttt{t}_j : \texttt{T}_j}{\Gamma \vdash <\texttt{l}_j=\texttt{t}_j>\ \texttt{as}\ <\texttt{l}_j=\texttt{T}_j^{j=1..n}> : <\texttt{l}_j=\texttt{T}_j^{j=1..n}>} \qquad (\texttt{T} - \texttt{Variant})$$

$$\frac{\Gamma \vdash \texttt{t}_0 : <\texttt{l}_j=\texttt{T}_j^{j=1..n}> \qquad \forall \texttt{i}\ \Gamma, \texttt{x}_i:\texttt{T}_i \vdash \texttt{t}_i:\texttt{T}}{\Gamma \vdash \texttt{case t}_0\ \texttt{of} <\texttt{l}_i=\texttt{x}_i> => \texttt{t}_i^{i=1..n} : \texttt{T}} \qquad (\texttt{T} - \texttt{Case})$$

Options

- Optional values. Example: `OptionalNat = <none:Unit,some:Nat>` represents finite mapping from numbers to numbers
- Example: Managing tables in a restaurant
  `Restaurant = Nat → OptionalNat`
  - At the begining, all tables are empty:
    ```
    emptyRestaurant: λn:Nat. <none=unit> as OptionNat;
    > emptyRestaurant : Restaurant
    ```
  - `modifyTable` modifies the value associated to table `m`:
    ```
    modifyTable = λt:Restaurant.λm:Nat.λv:Nat.
          λn:Nat.if eq n m
                 then if eq v 0
                      then <nome=unit> as OptionalNat
                      else <some=v> as OptionalNat
                else t n;
    > modifyTable : Restaurant → Nat → Nat → Restaurant
    ```
  - To check the number fo people at a given table:
    ```
    lookup = λt:Restaurant.λn:Nat.case (t n) of
          <none=u> => 0
          |<some=v> => v;
    >lookup: Restaurant → Nat → Nat
    ```

Enumerations

- When every field in a variant if `Unit`.
- Example: a type for representing the days of a working week:

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

- Elements of the form `<monday=unit> as Weekday`.
- The type `Weekday` is inhabited by precisely five values
- The case construct can be used to define computations on enumerations:

```
nextBusinessDay = λw.Weekday.
   case w of <monday=x> => <tuesday=unit> as Weekday
      | <tuesday=x> => <wednesday=unit> as Weekday
      | <wednesday=x> => <thursday=unit> as Weekday
      | <thursday=x> => <friday=unit> as Weekday
      | <friday=x> => <monday=unit> as Weekday
```

Single-field variants

- A variant with just a single label: `V = <l:T>`.
- Purpose: avoid direct access to fields
- Example: financial calculations in multiple currencies
  - Currencies represented as `Float`:

```
dollars2euros = λd:Float. timesFloat d 0.74
> dollars2euros : Float → Float
euros2dollars = λe:Float.timesFloat e 1.34
> euros2dollars : Float → Float
```

  - Confusing: there are no ways the type system can help prevent nonsense conversions (e.g. euros to euros)

- A better solution:
  - Define euros and dollars as different variant types:

```
DollarAmount = <dollars:Float>
EuroAmount = <euros:Float>
```

  - Define safe versions of the conversion functions:

```
dollars2euros =
   λd:DollarAmount.
      case d of <dollars=x> =>
      <euros = timesFloat x 0.74> as EuroAmount;
> dollars2euros : DollarAmount → EuroAmount
```

Recursion

- In λ, recursion was defined with the aid of `fix` combinator.
- In $\lambda_\to$ recursion can be defined in a similar way. Example

```
iseven_aux = λie:Nat→Bool.λx:Nat.
                  if iszero x then true
                  else if iszero (pred x) then false
                  else ie (pred (pred x));
> iseven_aux : (Nat → Bool) → Nat → Bool
iseven = fix iseven_aux
> iseven : Nat → Bool
iseven 7
> false : Bool
```

- Problem: it is not possible to give a type to `fix` in $\lambda_\to$.
- Solution: add `fix` as a new primitive of the language with the corrending evaluation and typing rules

- Syntax rules:

```
t::= ...
      fix t
```

- Evaluation rules:

$$\mathtt{fix}(\lambda \mathtt{f} : \mathtt{T}_1.\mathtt{t}_2) \rightarrow [\mathtt{f} \mapsto (\mathtt{fix}(\lambda \mathtt{f} : \mathtt{T}_1 : \mathtt{t}_2))]\mathtt{t}_2 \quad (\mathtt{E} - \mathtt{FixBeta})$$

$$\frac{\mathtt{t}_1 \rightarrow \mathtt{t}_1'}{\mathtt{fix}\ \mathtt{t}_1 \rightarrow \mathtt{fix}\ \mathtt{t}_1'} \quad (\mathtt{E} - \mathtt{Fix})$$

- Typing rule:

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \rightarrow \mathtt{T}_1}{\Gamma \vdash \mathtt{fix}\ \mathtt{t}_1 : \mathtt{T}_1} \quad (\mathtt{T} - \mathtt{Fix})$$

Lists

- For every T, the type `List T` describes finite-length lists whose elements are drawn from T.
- New syntax forms:

```
t::= ...
      nil[T]
      cons[T] t t
      isnil[T] t
      head[T] t
      tail[T] t

v::= ...
      nil[T]
      cons[T] v v
T::= ...
      List T
```

- Empty list: `nil[T]`.
- List resulting of adding t_1 to the from of t_2: `cons[T]` t_1 t_2.

- Evaluation rules:

$$\frac{\mathtt{t_1 \to t_1'}}{\mathtt{cons\ [T]\ t_1\ t_2 \to cons\ [T]\ t_1'\ t_2}} \qquad (\mathtt{E-Cons1})$$

$$\frac{\mathtt{t_2 \to t_2'}}{\mathtt{cons\ [T]\ v_1\ t_2 \to cons\ [T]\ v_1\ t_2'}} \qquad (\mathtt{E-Cons2})$$

$$\mathtt{isnil[S]\ (nil[T]) \to true} \qquad (\mathtt{E-IsNilNil})$$

$$\mathtt{isnil[S]\ (cons[T]\ v_1\ v_2) \to false} \qquad (\mathtt{E-IsNilCons})$$

$$\frac{\mathtt{t_1 \to t_1'}}{\mathtt{isnil\ [T]\ t_1 \to isnil\ [T]\ t_1'}} \qquad (\mathtt{E-IsNil})$$

$$\mathtt{head[S]\ (cons[T]\ v_1\ v_2) \to v_1} \qquad (\mathtt{E-HeadCons})$$

$$\frac{\mathtt{t_1 \to t_1'}}{\mathtt{head\ [T]\ t_1 \to head\ [T]\ t_1'}} \qquad (\mathtt{E-Head})$$

$$\mathtt{tail[S]\ (cons[T]\ v_1\ v_2) \to v_2} \qquad (\mathtt{E-TailCons})$$

$$\frac{\mathtt{t_1 \to t_1'}}{\mathtt{tail\ [T]\ t_1 \to tail\ [T]\ t_1'}} \qquad (\mathtt{E-Tail})$$

- Typing rules:

$$\Gamma \vdash \mathtt{nil}[\mathtt{T}_1] : \mathtt{List}\ \mathtt{T}_1 \qquad (\mathtt{T} - \mathtt{Nil})$$

$$\frac{\Gamma\vdash\mathtt{t}_1 : \mathtt{T}_1 \qquad \Gamma\vdash\mathtt{t}_2 : \mathtt{List}\ \mathtt{T}_1}{\Gamma\vdash\mathtt{cons}[\mathtt{T}_1]\ \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{List}\ \mathtt{T}_1} \qquad (\mathtt{T} - \mathtt{Cons})$$

$$\frac{\Gamma\vdash\mathtt{t}_1 : \mathtt{List}\ \mathtt{T}_1}{\Gamma\vdash\mathtt{isnil}[\mathtt{T}_1]\ \mathtt{t}_1 : \mathtt{Bool}} \qquad (\mathtt{T} - \mathtt{IsNil})$$

$$\frac{\Gamma\vdash\mathtt{t}_1 : \mathtt{List}\ \mathtt{T}_1}{\Gamma\vdash\mathtt{head}[\mathtt{T}_1]\ \mathtt{t}_1 : \mathtt{T}_1} \qquad (\mathtt{T} - \mathtt{Head})$$

$$\frac{\Gamma\vdash\mathtt{t}_1 : \mathtt{List}\ \mathtt{T}_1}{\Gamma\vdash\mathtt{tail}[\mathtt{T}_1]\ \mathtt{t}_1 : \mathtt{List}\ \mathtt{T}_1} \qquad (\mathtt{T} - \mathtt{Tail})$$