Types and Programming Languages

Lecture 5. Extensions of simple types

Xiaojuan Cai

cxj@sjtu.edu.cn

BASICS Lab, Shanghai Jiao Tong University

Fall, 2016
Simply typed λ-calculus has enough structure to make its theoretical properties interesting, but it is not yet much of a programming language.
Simply typed \( \lambda \)-calculus has enough structure to make its theoretical properties interesting, but it is not yet much of a programming language.

Close the gap with more familiar languages by introducing: Base types, Unit type, Pairs, Tuples, Records, Sum, etc.
Simply typed $\lambda$-calculus has enough structure to make its theoretical properties interesting, but it is not yet much of a programming language.

Close the gap with more familiar languages by introducing: Base types, Unit type, Pairs, Tuples, Records, Sum, etc.

An important theme throughout the part is the concept of derived forms.
Outline

Simple extensions

Base types
Unit type and sequencing
Ascription
let bindings
Pairs and tuples
Records

More extensions

Sums and variants
General recursion
Base types

- Every programming language provides base types, such as numbers, booleans, or characters, plus appropriate primitive operations for manipulating these values.
Base types

- Every programming language provides **base types**, such as numbers, booleans, or characters, plus **appropriate primitive operations** for manipulating these values.
- For theoretical purposes, we abstract away from the details of particular base types and their operations.

\[
\text{New types. } T ::= \cdots | A
\]

where \(A\) denotes some base type.
Base types

- Every programming language provides base types, such as numbers, booleans, or characters, plus appropriate primitive operations for manipulating these values.
- For theoretical purposes, we abstract away from the details of particular base types and their operations.

New types. \( T ::= \cdots | A \)

where \( A \) denotes some base type. For example,
\[
\lambda x : A. x : A \rightarrow A \\
\lambda f : A \rightarrow A. \lambda x : A. f(f(x)) : \\
\]
Base types

- Every programming language provides base types, such as numbers, booleans, or characters, plus appropriate primitive operations for manipulating these values.
- For theoretical purposes, we abstract away from the details of particular base types and their operations.

New types.  \[ T ::= \cdots \mid A \]

where \( A \) denotes some base type. For example,
\[
\lambda x : A. x : A \rightarrow A
\]
\[
\lambda f : A \rightarrow A. \lambda x : A. f(f(x)) : (A \rightarrow A) \rightarrow A \rightarrow A
\]
The Unit type

New terms $t ::= \cdots \mid unit$

New values $\nu ::= \cdots \mid unit$

New types $T ::= \cdots \mid \text{Unit}$

New typing rules $\text{T-UNIT} \quad \Gamma \vdash unit : \text{Unit}$

- Unit type can be found in the ML family.
- The main application is in languages with side effects, such as assignments to reference cells.
- Similar to void in languages like C and Java.
Derived forms: Sequencing and Wildcards

Two ways to add *sequencing*

1. add new syntax, evaluation and typing rules for sequencing:

   \[ t ::= \cdots \mid t_1; t_2 \]

   $\text{E-SEQ} \quad \frac{}{t_1 \rightarrow t'_1} \quad \frac{t_1; t_2 \rightarrow t'_1; t_2}{t_1; t_2 \rightarrow t'_1; t_2}$

   $\text{E-SEQ\_NEXT} \quad \frac{\Gamma \vdash t_1 : \text{Unit}}{\Gamma \vdash t_1; t_2 : T}$

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

2. Define it as *derived forms*:

   \[ t_1; t_2 \overset{\text{def}}{=} (\lambda x : \text{Unit} . t_2) t_1 \quad \text{where} \ x \not\in \text{FV}(t_2) \]
Derived forms: Sequencing and Wildcards

- Derived form has another name: **syntactic sugar**.
- The advantage is that we can extend the surface syntax without adding any complexity about theorems to be proved.
- Derived form has been heavily used in modern language definitions.
- Another derived form: **wildcard** \( \lambda_.t \overset{\text{def}}{=} \lambda x.t \) where \( x \not\in FV(t) \).
Ascription

Another simple feature is the ability to explicitly ascribe a particular type to a given term.

**New terms**  \( t ::= \cdots | t \text{ as } T \)

**New evaluation rules**

\[
\begin{align*}
\text{E-Ascribe} & \quad \frac{}{v_1 \text{ as } T \rightarrow v_1} & \quad \text{E-Ascribe1} & \quad \frac{t_1 \rightarrow t'_1}{t_1 \text{ as } T \rightarrow t'_1 \text{ as } T}
\end{align*}
\]

**New typing rules**

\[
\begin{align*}
\text{T-Ascribe} & \quad \frac{}{\Gamma \vdash t_1 : T} & \quad \frac{}{\Gamma \vdash t_1 \text{ as } T : T}
\end{align*}
\]
Purpose of ascription

- For documentation and maintenance
- For controlling the printing of complex types
- For abstract away some types, especially in PLs with type inference, such as SML.

Note that we add new syntax, semantics rules to add ascription. How to consider ascription as derived forms? (See Homework.)
let bindings

It is often useful — both for avoiding repetition and for increasing readability — to give names to some of its subexpressions.
let bindings

It is often useful — both for *avoiding repetition* and for *increasing readability* — to give names to some of its subexpressions.

let bindings are very common syntax in a lot of PLs, such as ML family, Scheme, but with slightly different *scoping rules*.

**New terms**  
\[ t ::= \cdots \mid \text{let } x = t \text{ in } t \]
let bindings

It is often useful — both for avoiding repetition and for increasing readability — to give names to some of its subexpressions.

let bindings are very common syntax in a lot of PLs, such as ML family, Scheme, but with slightly different scoping rules.

New terms  \[ t ::= \cdots \mid \text{let } x = t \text{ in } t \]

New evaluation rules
\[
\begin{align*}
\text{E-LetV} & \quad \frac{}{\text{let } x = v_1 \text{ in } t_2 \rightarrow [x \mapsto v_1]t_2} \\
\text{E-Let} & \quad \frac{t_1 \rightarrow t'_1}{\text{let } x = t_1 \text{ in } t_2 \rightarrow \text{let } x = t'_1 \text{ in } t_2}
\end{align*}
\]

New typing rules
\[
\begin{align*}
\text{T-Let} & \quad \frac{\Gamma \vdash t_1 : T_1 \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2}
\end{align*}
\]
let bindings, as derived forms
let bindings, as derived forms

\[
\text{let } x = t_1 \text{ in } t_2 \overset{\text{def}}{=} (\lambda x : T_1 . t_2) t_1
\]
let bindings, as derived forms

\[ \text{let } x = t_1 \text{ in } t_2 \overset{\text{def}}{=} (\lambda x : T_1.t_2) t_1 \]

Where \( T_1 \) comes from?
let bindings, as derived forms

\[
\text{let } x = t_1 \text{ in } t_2 \stackrel{\text{def}}{=} (\lambda x : T_1 . t_2) t_1
\]

Where \( T_1 \) comes from? Type checker!

The desugaring of sequencing is a transformation on terms. However, The desugaring of let binding is a transformation on typing derivations.

We will NOT treat let bindings as a derived form.
Pairs

The simplest compound data structure is pairs, or more generally tuples, of values.

New terms \[ t ::= \cdots | \{t, t\} | t.1 | t.2 \]
New values \[ t ::= \cdots | \{v, v\} \]
New types \[ t ::= \cdots | T_1 \times T_2 \text{ product type} \]
Pairs

The simplest compound data structure is **pairs**, or more generally **tuples**, of values.

**New terms**

\[ t ::= \cdots \mid \{t, t\} \mid t.1 \mid t.2 \]

**New values**

\[ t ::= \cdots \mid \{v, v\} \]

**New types**

\[ t ::= \cdots \mid T_1 \times T_2 \text{ product type} \]

**New evaluation rules**
Pairs

The simplest compound data structure is pairs, or more generally tuples, of values.

**New terms**  
\[ t ::= \cdots \mid \{ t, t \} \mid t.1 \mid t.2 \]

**New values**  
\[ t ::= \cdots \mid \{ v, v \} \]

**New types**  
\[ t ::= \cdots \mid T_1 \times T_2 \text{ product type} \]

**New evaluation rules**

- **E-PairBeta1**
  \[
  \{ v_1, v_2 \}.1 \longrightarrow v_1
  \]

- **E-PairBeta2**
  \[
  \{ v_1, v_2 \}.2 \longrightarrow v_2
  \]

- **E-Proj1**
  \[
  t_1 \longrightarrow t_1'
  t_1.1 \longrightarrow t_1'.1
  \]

- **E-Proj2**
  \[
  t_1 \longrightarrow t_1'
  t_1.2 \longrightarrow t_1'.2
  \]

- **E-Pair1**
  \[
  t_1 \longrightarrow t_1'
  \{ t_1, t_2 \} \longrightarrow \{ t_1', t_2 \}
  \]

- **E-Pair1**
  \[
  t_2 \longrightarrow t_2'
  \{ v_1, t_2 \} \longrightarrow \{ v_1, t_2' \}
  \]

**New typing rules**
Pairs

The simplest compound data structure is pairs, or more generally tuples, of values.

New terms \[ t ::= \cdots \mid \{ t, t \} \mid t.1 \mid t.2 \]
New values \[ t ::= \cdots \mid \{ v, v \} \]
New types \[ t ::= \cdots \mid T_1 \times T_2 \text{ product type} \]

New evaluation rules

\[ \begin{align*}
\text{E-PairBeta1} & : & \{ v_1, v_2 \}.1 & \rightarrow v_1 \\
\text{E-PairBeta2} & : & \{ v_1, v_2 \}.2 & \rightarrow v_2 \\
\text{E-Proj1} & : & t_1 & \rightarrow t'_1 \\
& & t_1.1 & \rightarrow t'_1.1 \\
& & t_1.1 & \rightarrow t'_1.1 \\
\text{E-Proj2} & : & t_1 & \rightarrow t'_1 \\
& & t_1.2 & \rightarrow t'_1.2 \\
\text{E-Pair1} & : & t_1 & \rightarrow t'_1 \\
& & \{ t_1, t_2 \} & \rightarrow \{ t'_1, t_2 \} \\
& & \{ t_1, t_2 \} & \rightarrow \{ t'_1, t_2 \} \\
\text{E-Pair1} & : & t_2 & \rightarrow t'_2 \\
& & \{ v_1, t_2 \} & \rightarrow \{ v_1, t'_2 \} \\
& & \{ v_1, t_2 \} & \rightarrow \{ v_1, t'_2 \} \\
\end{align*} \]

New typing rules

\[ \begin{align*}
\text{T-Pair} & : & \Gamma \vdash t_1 : T_1 & \Gamma \vdash t_2 : T_2 \\
& & \Gamma \vdash \{ t_1, t_2 \} : T_1 \times T_2 \\
\text{T-Proj1} & : & \Gamma \vdash t_1 : T_1 \times T_2 \\
& & \Gamma \vdash t_{1.1} : T_1 \\
\text{T-Proj1} & : & \Gamma \vdash t_1 : T_1 \times T_2 \\
& & \Gamma \vdash t_{1.2} : T_2 \\
\end{align*} \]
Tuples

It’s easy to generalize pairs to tuples. Pair is a 2-tuple.

**New terms**
\[ t ::= \cdots \mid \{ t_i^{i\in 1..n} \} \mid t.i \]

**New values**
\[ t ::= \cdots \mid v_i^{i\in 1..n} \]

**New types**
\[ t ::= \cdots \mid \{ T_i^{i\in 1..n} \} \text{ product type} \]

**New evaluation rules**

\[
\begin{align*}
\text{E-TupleBeta} & \quad \{ v_i^{i\in 1..n} \}.j \longrightarrow v_j \\
\text{E-PROJ} & \quad t_1 \longrightarrow t'_1 \\
\text{E-Tuple} & \quad t_j \longrightarrow t'_j \\
\end{align*}
\]

\[
\begin{align*}
\{ v_i^{i\in 1..j-1}, t_j, t_k^{k\in j+1..n} \} \longrightarrow \{ v_i^{i\in 1..j-1}, t'_j, t_k^{k\in j+1..n} \}
\end{align*}
\]

**New typing rules**

\[
\begin{align*}
\text{T-Tuple} & \quad \forall i \quad \Gamma \vdash t_i : T_i \\
\Gamma \vdash \{ t_i^{i\in 1..n} \} : \{ T_i^{i\in 1..n} \}
\end{align*}
\]

\[
\begin{align*}
\text{T-PROJ} & \quad \Gamma \vdash t_1 : \{ T_i^{i\in 1..n} \} \\
\Gamma \vdash t_1.j : T_j
\end{align*}
\]
### Records

#### New syntactic forms

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathbf{t}$ ::= ...</td>
<td>\n { $l_i = t_i$ }_{i \in 1..n}</td>
</tr>
<tr>
<td>$\mathbf{t} . l$</td>
<td>$t_j$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathbf{v}$ ::= ...</td>
<td>\n { $l_i = v_i$ }_{i \in 1..n}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathbf{T}$ ::= ...</td>
<td>\n { $l_i : T_i$ }_{i \in 1..n}</td>
</tr>
</tbody>
</table>

#### New evaluation rules

$\{ l_i = v_i \}_{i \in 1..n} . l_j \rightarrow v_j$

#### New typing rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash t : T$</td>
<td>for each $i$ $\Gamma \vdash t_i : T_i$</td>
</tr>
<tr>
<td>$\Gamma \vdash { l_i = t_i }<em>{i \in 1..n} : { l_i : T_i }</em>{i \in 1..n}$</td>
<td></td>
</tr>
<tr>
<td>$\Gamma \vdash t_1 : { l_i : T_i }_{i \in 1..n}$</td>
<td>$\Gamma \vdash t_1 . l_j : T_j$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1 \rightarrow t'_1$</td>
<td>$t_1 . l \rightarrow t'_1 . l$</td>
</tr>
<tr>
<td>$t_j \rightarrow t'_j$</td>
<td>${ l_i = v_i }<em>{i \in 1..j-1}, l_j = v_j, l_k = t_k }</em>{k \in j+1..n}$</td>
</tr>
<tr>
<td>$\frac{t_j \rightarrow t'<em>j}{{ l_i = v_i }</em>{i \in 1..j-1}, l_j = v_j, l_k = t_k }_{k \in j+1..n}$</td>
<td></td>
</tr>
</tbody>
</table>

#### Diagram

![Diagram of records and their rules](Image)

---

**Figure 11-7: Records**
Records

Surface difference:

- Fields in records has names, pairs does not.
- We can treat pairs as special records labeled 1, 2, 3, ....

Deeper difference: Compilers may implement them in distinct ways. Pair has strict orders. However, PLs treat the order of fields in records differently, e.g.

- In SML, order does not matter: `{l=2, m=4} = {m=4, l=2};`
- In rules of Figure 11-7, orders matter, i.e., `{l=2, m=4} \neq {m=4, l=2};`
- We will later use subtyping to make records with different field permutations equivalent.

Rules for records are similar to those for tuples. Please refer to Figure 11-7 in the textbook.
Records

Surface difference:
- Fields in records have names, pairs do not.
- We can treat pairs as special records labeled 1, 2, 3, ....

Deeper difference: Compilers may implement them in distinct ways. Pair has strict orders. However, PLs treat the order of fields in records differently, e.g.
Records

Surface difference:
- Fields in records has names, pairs does not.
- We can treat pairs as special records labeled 1, 2, 3, ....

Deeper difference: Compilers may implement them in distinct ways. Pair has strict orders. However, PLs treat the order of fields in records differently, e.g.
- in SML, order does not matter:
  \[ \{ l = 2, m = 4 \} = \{ m = 4, l = 2 \}; \]
- in rules of Figure 11-7, orders matter, i.e.,
  \[ \{ l = 2, m = 4 \} \neq \{ m = 4, l = 2 \} \]
Records

Surface difference:
- Fields in records has names, pairs does not.
- We can treat pairs as special records labeled 1, 2, 3, ....

Deeper difference: Compilers may implement them in distinct ways. Pair has strict orders. However, PLs treat the order of fields in records differently, e.g.
- in SML, order does not matter:
  \[[l = 2, m = 4] = [m = 4, l = 2]\];
- in rules of Figure 11-7, orders matter, i.e.,
  \[[l = 2, m = 4] \neq [m = 4, l = 2]\]
- we will latter use subtyping to make records with different field permutations equivalent.

Rules for records are similar to those for tuples. Please refer to Figure 11-7 in the textbook.
Records

Surface difference:
- Fields in records has names, pairs does not.
- We can treat pairs as special records labeled 1, 2, 3, ....

Deeper difference: Compilers may implement them in distinct ways. Pair has strict orders. However, PLs treat the order of fields in records differently, e.g.
- in SML, order does not matter:
  \[ \{ l = 2, m = 4 \} = \{ m = 4, l = 2 \}; \]
- in rules of Figure 11-7, orders matter, i.e.,
  \[ \{ l = 2, m = 4 \} \neq \{ m = 4, l = 2 \} \]
- we will latter use subtyping to make records with different field permutations equivalent.

Rules for records are similar to those for tuples. Please refer to Figure 11-7 in the textbook.
Outline

Simple extensions
- Base types
- Unit type and sequencing
- Ascription
- let bindings
- Pairs and tuples
- Records

More extensions
- Sums and variants
- General recursion
Variant types

We need *heterogeneous collections of values* in many cases:

- a node in a tree can be a *leaf* or an *interior* node with children;
- a list cell can be either *nil* or a *cons* cell carrying a head and a tail,
- a node of an abstract syntax tree in a compiler can represent a *variable*, an *abstraction*, an *application*, etc.
Variant types

We need *heterogeneous collections of values* in many cases:

- a node in a tree can be a *leaf* or an *interior* node with children;
- a list cell can be either *nil* or a *cons* cell carrying a head and a tail,
- a node of an abstract syntax tree in a compiler can represent a *variable*, an *abstraction*, an *application*, etc

Type-theoretic mechanism that supports this kind of programming is *variant types*. 
Variant types

We need heterogeneous collections of values in many cases:

- a node in a tree can be a leaf or an interior node with children;
- a list cell can be either nil or a cons cell carrying a head and a tail,
- a node of an abstract syntax tree in a compiler can represent a variable, an abstraction, an application, etc

Type-theoretic mechanism that supports this kind of programming is variant types.
A more familiar name for variant type is union, or more precisely, disjoint union. 
Sum is the binary version of variant type.
Sums

Constructors and accessors

\[
\begin{aligned}
t & ::= \cdots \\
\text{inl } t & \quad \text{tagging (left)} \\
\text{inr } t & \quad \text{tagging (right)} \\
\text{case } t \text{ of} & \quad \text{case} \\
\text{inl } x & \Rightarrow t \\
\mid \text{inr } x & \Rightarrow t
\end{aligned}
\]

\[
\begin{aligned}
v & ::= \cdots \\
\text{inl } v & \quad \text{tagged value (left)} \\
\text{inr } v & \quad \text{tagged value (right)}
\end{aligned}
\]

\[
\begin{aligned}
T & ::= \cdots \\
T + T & \quad \text{sum type}
\end{aligned}
\]
Sums, example

Your score may be an integer number, or a grade (P/F).

*Types*: Score = Int + Char

*Constructor*: \( t_1 = \text{inl} \, 59 : \text{Score}, \quad t_2 = \text{inr} \, 'F' : \text{Score} \)

*Accessor*: \( \text{a_good_teacher} = \lambda t.\text{case } t \text{ of} \)
\[
\begin{align*}
\text{inl } x & \Rightarrow \text{inl} \, (\text{max}(x, 60)) \\
\text{inr } x & \Rightarrow \text{inr} \, 'P'
\end{align*}
\]
Sums, example

Your score may be an integer number, or a grade (P/F).

Types: Score = Int + Char

Constructor: \( t_1 = \text{inl} \ 59 : \text{Score}, \quad t_2 = \text{inr} \ ‘F’ : \text{Score} \)

Accessor: \( a \text{good teacher} = \lambda t. \text{case } t \text{ of} \)
\[
\begin{align*}
inl \ x & \Rightarrow \text{inl} \ (\text{max} (x, 60)) \\
| \ inr \ x & \Rightarrow \text{inr} \ ‘P’
\end{align*}
\]

Quiz.
1. Give the evaluation rule and typing rule for \text{sum}.
2. Does the "uniqueness of types" still hold for languages with \text{sum}? Why?
Sums, semantics

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

\[
\text{E-Case} \quad t_0 \quad \rightarrow \quad t_0'
\]

\[
\text{E-Case} \quad \text{case} \ t_0 \ \text{of} \ \text{inl} \ x_1 \Rightarrow t_1 \ | \ \text{inr} \ x_2 \Rightarrow t_2 \quad \rightarrow \quad \text{case} \ t_0' \ \text{of} \ \text{inl} \ x_1 \Rightarrow t_1 \ | \ \text{inr} \ x_2 \Rightarrow t_2
\]

\[
\text{E-Inl} \quad t_1 \quad \rightarrow \quad t_1'
\]

\[
\text{E-Inl} \quad \text{inl} \ t_1 \quad \rightarrow \quad \text{inl} \ t_1'
\]

\[
\text{T-Inl} \quad \Gamma, x_1 : T_1 \vdash t_1 : T \quad \Gamma, x_2 : T_2 \vdash t_2 : T \quad \rightarrow \quad \Gamma \vdash \text{inl} \ t_1 : T_1 + T_2
\]

\[
\text{T-Case} \quad \Gamma \vdash t_0 : T_1 + T_2 \quad \Gamma, x_1 : T_1 \vdash t_1 : T \quad \Gamma, x_2 : T_2 \vdash t_2 : T \quad \rightarrow \quad \text{case} \ t_0 \ \text{of} \ \text{inl} \ x_1 \Rightarrow t_1 \ | \ \text{inr} \ x_2 \Rightarrow t_2 : T
\]

Symmetric rules for \(\text{inr}\) are omitted.
Sums and uniqueness of types

This rule breaks the uniqueness of types:

\[
\frac{T\text{-INL}}{
\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash \text{inl} \ t_1 : T_1 + T_2
}
\]

Solutions include:

- Keep it as a “variable” which will be instantiated later. Mainly used in PLs with type inference.
Sums and uniqueness of types

This rule breaks the uniqueness of types:

\[
\begin{array}{c}
\text{T-INL} \quad \Gamma \vdash t_1 : T_1 \\
\hline
\Gamma \vdash \text{inl } t_1 : T_1 + T_2
\end{array}
\]

Solutions include:

- Keep it as a “variable” which will be instantiated later. Mainly used in PLs with type inference.
- Allow any \( T_2 \). We will explore this option when discussing subtyping.
Sums and uniqueness of types

This rule breaks the uniqueness of types:

\[
\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \text{inl } t_1 : T_1 + T_2}
\]

Solutions include:

- Keep it as a “variable” which will be instantiated later. Mainly used in PLs with type inference.
- Allow any \( T_2 \). We will explore this option when discussing subtyping.
- Ascription: use explicit annotation to tell the compiler or type checker which type \( T_2 \) is intended.

\[
\frac{\text{inl } t \text{ as } T}{\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \text{inl } t_1 \text{ as } T_1 + T_2 : T_1 + T_2}}
\]
Variants

Like the relation between pair and records. Variants are extensions of sum type with fields.

\[ t_1 = \langle \text{none} = \text{unit} \rangle \; \text{as} \; \langle \text{none} : \text{Unit}, \text{some} : \text{Nat} \rangle \]

\[ t_2 = \langle \text{some} = 20 \rangle \; \text{as} \; \langle \text{none} : \text{Unit}, \text{some} : \text{Nat} \rangle \]

\[ f = \lambda x : \langle \text{none} : \text{Unit}, \text{some} : \text{Nat} \rangle. \]

\[
\begin{cases}
\text{case } x \text{ of} \\
\langle \text{none} = u \rangle \Rightarrow 999 \\
| \langle \text{some} = v \rangle \Rightarrow v
\end{cases}
\]

In ML family, this type is called option.
Enumerations and Single-Field variants

- We can construct enumerations by using variants, each field has type Unit.

\[
\text{Weekday} = \langle \text{monday: Unit}, \text{tuesday: Unit}, \ldots, \text{friday: Unit} \rangle
\]

The access of this enumeration is very annoying. We will have alternatives later.

- The single-field variants looks silly, but is useful to abstract/hide information.

\[
\text{DollarAmount} = \langle \text{dollars: Float} \rangle
\]

\[
\text{EuroAmount} = \langle \text{euros: Float} \rangle
\]
Discussion: variants v.s. Datatypes

Variant type is analogous to the ML datatype

$$\text{type } T = l_1 \text{ of } T_1 \mid \cdots \mid l_n \text{ of } T_n$$

But there are several differences worth noticing

- For ML datatype, we do not use $l_i(t_i)$ as $T$ to explicitly tell the compiler $T$, instead the constructor $l_i$ has type $T_i \rightarrow T$.
- Enumeration is much easier with datatype, we omit of Unit:

  $$\text{type Weekday = monday} \mid \cdots \mid \text{friday}$$

- ML datatype has several additional important features:
  - Recursive datatype:
    $$\text{type NatList = nil} \mid \text{cons of Nat * NatList}$$
  - Parametric datatype:
    $$\text{type } 'a \text{ List = nil} \mid \text{cons of } 'a * 'a \text{ List}$$

List is called a type operator.
Another facility found in most programming languages is the ability to define \textit{recursive functions}. Here is one way to define a function \texttt{iseven}:

\[
\texttt{ff} = \lambda \texttt{ie} : \texttt{Nat} \to \texttt{Bool}. \lambda \texttt{x} : \texttt{Nat}.
\begin{align*}
& \quad \text{if iszero } \texttt{x} \text{ then true} \\
& \quad \text{else if iszero (pred } \texttt{x} \text{) then false} \\
& \quad \text{else ie (pred \ (pred } \texttt{x})\); \\
\texttt{iseven} = \texttt{fix ff};
\end{align*}
\]

\textbf{Quiz.}  
What's the type of \texttt{ff}?
General recursion

fix itself cannot be defined in the simply typed lambda-calculus. We simply add it as primitives.

**New terms** \( t ::= \cdots \mid \text{fix } t \)**

**New evaluation rules**
General recursion

fix itself cannot be defined in the simply typed lambda-calculus. We simply add it as primitives.

**New terms** \( t ::= \cdots | \text{fix } t \)

**New evaluation rules**

\[
E\text{-FixBeta}
\]

\[
\frac{\text{fix } (\lambda x : T_1. t_2) \rightarrow [x \mapsto \text{fix } (\lambda x : T_1. t_2)] t_2}{t_1 \rightarrow t_1'}\]

\[
E\text{-Fix}
\]

\[
\frac{t_1 \rightarrow t_1'}{\text{fix } t_1 \rightarrow \text{fix } t_1'}\]

**New typing rules**
General recursion

`fix` itself cannot be defined in the simply typed lambda-calculus. We simply add it as primitives.

**New terms**  \( t ::= \cdots | \text{fix} \ t \)

**New evaluation rules**

**E-FixBeta**

\[
\text{fix} \ (\lambda x : T_1. t_2) \to [x \mapsto \text{fix} \ (\lambda x : T_1. t_2)]t_2
\]

**E-Fix**

\[
t_1 \to t'_1 \\
\text{fix} \ t_1 \to \text{fix} \ t'_1
\]

**New typing rules**

**T-Fix**

\[
\Gamma \vdash t_1 : T_1 \to T_1 \\
\Gamma \vdash \text{fix} \ t_1 : T_1
\]
General recursion

fix itself cannot be defined in the simply typed lambda-calculus. We simply add it as primitives.

**New terms** \( t ::= \cdots \mid \text{fix } t \)

**New evaluation rules**

\[
\text{E-FixBeta} \quad \text{fix } (\lambda x : T_1.t_2) \rightarrow [x \mapsto \text{fix } (\lambda x : T_1.t_2)]t_2
\]

\[
\text E-\text{Fix}\quad t_1 \rightarrow t'_1 \quad \Rightarrow \quad \text{fix } t_1 \rightarrow \text{fix } t'_1
\]

**New typing rules**

\[
\text{T-Fix} \quad \Gamma \vdash t_1 : T_1 \rightarrow T_1 \quad \Rightarrow \quad \Gamma \vdash \text{fix } t_1 : T_1
\]

**New derived forms**

\[
\text{letrec } x : T_1 = t_1 \text{ in } t_2 \overset{\text{def}}{=} \text{let } x = \text{fix } (\lambda x : T_1.t_1) \text{ in } t_2
\]
More on `fix`

- Notice that the type $T_1$ in rule $T$-Fix is not restricted to function types.
- `fix` implies that every type is inhabited by some term.

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

`diverge_T(unit)` has type $T$, and has non-terminating evaluation.

- The simply typed lambda-calculus with numbers and `fix`, called PCF (Programming Computable Functions), is the simplest language with a range of subtle semantic phenomena.
List

Typing features can be classified into

- **base types** such as `Bool` and `Unit`
- **type constructors** such as `→` and `×`
- List is also a type constructor: For every `T`, `List T` returns a type describing finite length lists whose elements typed `T`.

Please refer to **Figure 11-13** for syntax and semantics of List.
Conclusion

- Real programming languages usually include: Base types, Unit type, Pairs, Tuples, Records, Sum, etc.
- `fix` can not be typed in simply typed lambda calculus. Most languages do not have explicit `fix`, but allow recursive definitions of functions.
- An important theme throughout the part is the concept of derived forms.
Homework

- 11.4.1, 11.5.2, 11.8.2, 11.11.1, 11.11.2