Types and Programming Languages

Lecture 5. Extensions of simple types

Xiaojuan Cai

cxj@sjtu.edu.cn

BASICS Lab, Shanghai Jiao Tong University

Spring, 2016

Coming soon

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

Coming soon

- Simply typed λ-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.

Coming soon

- Simply typed λ-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.

$$\text{New types.} \quad \mathtt{T} ::= \cdots \mid \mathtt{A}$$

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.

$$\text{New types.} \quad \mathtt{T} ::= \cdots \mid \mathtt{A}$$

where A denotes some base type.For example,
$\lambda x : \mathtt{A}.x \;:\; \mathtt{A} \to \mathtt{A}$
$\lambda f : \mathtt{A} \to \mathtt{A}\,.\lambda x : \mathtt{A}.f(f(x)) \;:$

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.

$$\text{New types.}\quad \mathtt{T} ::= \cdots \mid \mathtt{A}$$

where A denotes some base type.For example,
$\lambda x : \mathtt{A}.x \ : \ \mathtt{A} \to \mathtt{A}$
$\lambda f : \mathtt{A} \to \mathtt{A}\,.\lambda x : \mathtt{A}.f(f(x)) \ : \ (\mathtt{A} \to \mathtt{A}) \to \mathtt{A} \to \mathtt{A}$

The Unit type

New terms $t ::= \cdots \mid \mathit{unit}$

New values $v ::= \cdots \mid \mathit{unit}$

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

New typing rules $\textsc{T-Unit}\ \dfrac{}{\Gamma \vdash \mathit{unit} : \mathtt{Unit}}$

- 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:

$$t ::= \cdots \mid t_1 ; t_2$$

$$\text{E-SEQ}\ \frac{t_1 \longrightarrow t_1'}{t_1 ; t_2 \longrightarrow t_1' ; t_2} \qquad \text{E-SEQNEXT}\ \frac{}{unit ; t_2 \longrightarrow t_2}$$

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

2. Define it as **derived forms**:

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

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 $\lambda\_.t \stackrel{def}{=} \lambda x.t$ where $x \notin FV(t)$.

Ascription

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

New terms $\quad t ::= \cdots \mid t \texttt{ as } \mathtt{T}$

New evaluation rules

$$\textsc{E-Ascribe}\ \frac{}{v_1 \texttt{ as } \mathtt{T} \longrightarrow v_1} \qquad \textsc{E-Ascribe1}\ \frac{t_1 \longrightarrow t_1'}{t_1 \texttt{ as } \mathtt{T} \longrightarrow t_1' \texttt{ as } \mathtt{T}}$$

New typing rules $\quad \textsc{T-Ascribe}\ \dfrac{\Gamma \vdash t_1 : \mathtt{T}}{\Gamma \vdash t_1 \texttt{ as } \mathtt{T} : \mathtt{T}}$

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 $t ::= \cdots \mid \texttt{let}\ x = t\ \texttt{in}\ t$

`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 $t ::= \cdots \mid \texttt{let}\ x = t\ \texttt{in}\ t$

New evaluation rules

$$\textsc{E-LetV}\ \frac{}{\texttt{let}\ x = v_1\ \texttt{in}\ t_2 \longrightarrow [x \mapsto v_1]t_2}$$

$$\textsc{E-Let}\ \frac{t_1 \longrightarrow t_1'}{\texttt{let}\ x = t_1\ \texttt{in}\ t_2 \longrightarrow \texttt{let}\ x = t_1'\ \texttt{in}\ t_2}$$

New typing rules $$\textsc{T-Let}\ \frac{\Gamma \vdash t_1 : \mathrm{T}_1 \qquad \Gamma, x : \mathrm{T}_1 \vdash t_2 : \mathrm{T}_2}{\Gamma \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2\ : \mathrm{T}_2}$$

`let` bindings, as derived forms

`let` bindings, as derived forms

$$\texttt{let } x = t_1 \texttt{ in } t_2 \stackrel{def}{=} (\lambda x : \texttt{T}_1 . t_2)\, t_1$$

`let` bindings, as derived forms

$$\texttt{let}\ x = t_1\ \texttt{in}\ t_2 \stackrel{\mathit{def}}{=} (\lambda x : \mathtt{T}_1 . t_2)\ t_1$$

Where $\mathtt{T}_1$ comes from?

`let` bindings, as derived forms

$$\texttt{let}\ x = t_1\ \texttt{in}\ t_2 \stackrel{def}{=} (\lambda x : \mathtt{T}_1 . t_2)\ t_1$$

Where $\mathtt{T}_1$ 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 $t ::= \cdots \mid \{t, t\} \mid t.1 \mid t.2$

New values $t ::= \cdots \mid \{v, v\}$

New types $t ::= \cdots \mid \mathtt{T}_1 \times \mathtt{T}_2$ 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 $t ::= \cdots \mid \mathtt{T}_1 \times \mathtt{T}_2$ 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 $t ::= \cdots \mid \mathtt{T}_1 \times \mathtt{T}_2$ product type

New evaluation rules

$$\textsc{E-PairBeta1}\ \frac{}{\{v_1, v_2\}.1 \longrightarrow v_1} \qquad \textsc{E-PairBeta2}\ \frac{}{\{v_1, v_2\}.2 \longrightarrow v_2}$$

$$\textsc{E-Proj1}\ \frac{t_1 \longrightarrow t_1'}{t_1.1 \longrightarrow t_1'.1} \qquad \textsc{E-Proj2}\ \frac{t_1 \longrightarrow t_1'}{t_1.2 \longrightarrow t_1'.2}$$

$$\textsc{E-Pair1}\ \frac{t_1 \longrightarrow t_1'}{\{t_1, t_2\} \longrightarrow \{t_1', t_2\}} \qquad \textsc{E-Pair1}\ \frac{t_2 \longrightarrow t_2'}{\{v_1, t_2\} \longrightarrow \{v_1, t_2'\}}$$

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 $t ::= \cdots \mid \mathtt{T}_1 \times \mathtt{T}_2$ product type

New evaluation rules

$$\textsc{E-PairBeta1}\ \frac{}{\{v_1, v_2\}.1 \longrightarrow v_1} \qquad \textsc{E-PairBeta2}\ \frac{}{\{v_1, v_2\}.2 \longrightarrow v_2}$$

$$\textsc{E-Proj1}\ \frac{t_1 \longrightarrow t_1'}{t_1.1 \longrightarrow t_1'.1} \qquad \textsc{E-Proj2}\ \frac{t_1 \longrightarrow t_1'}{t_1.2 \longrightarrow t_1'.2}$$

$$\textsc{E-Pair1}\ \frac{t_1 \longrightarrow t_1'}{\{t_1, t_2\} \longrightarrow \{t_1', t_2\}} \qquad \textsc{E-Pair1}\ \frac{t_2 \longrightarrow t_2'}{\{v_1, t_2\} \longrightarrow \{v_1, t_2'\}}$$

New typing rules $\textsc{T-Pair}\ \dfrac{\Gamma \vdash t_1 : \mathtt{T}_1 \qquad \Gamma \vdash t_2 : \mathtt{T}_2}{\Gamma \vdash \{t_1, t_2\} : \mathtt{T}_1 \times \mathtt{T}_2}$

$$\textsc{T-Proj1}\ \frac{\Gamma \vdash t_1 : \mathtt{T}_1 \times \mathtt{T}_2}{\Gamma \vdash t_1.1 : \mathtt{T}_1} \qquad \textsc{T-Proj1}\ \frac{\Gamma \vdash t_1 : \mathtt{T}_1 \times \mathtt{T}_2}{\Gamma \vdash t_1.2 : \mathtt{T}_2}$$

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 $t ::= \cdots \mid \{\mathtt{T}_i^{i \in 1..n}\}$ product type

New evaluation rules

$$\textsc{E-TupleBeta}\ \frac{}{\{v_i^{i \in 1..n}\}.j \longrightarrow v_j} \qquad \textsc{E-Proj}\ \frac{t_1 \longrightarrow t_1'}{t_1.j \longrightarrow t_1'.j}$$

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

New typing rules

$$\textsc{T-Tuple}\ \frac{\forall i \quad \Gamma \vdash t_i : \mathtt{T_i}}{\Gamma \vdash \{t_i^{i \in 1..n}\} : \{\mathtt{T}_i^{i \in 1..n}\}} \qquad \textsc{T-Proj}\ \frac{\Gamma \vdash t_1 : \{\mathtt{T}_i^{i \in 1..n}\}}{\Gamma \vdash t_1.j : \mathtt{T_j}}$$

Records

$\rightarrow$ $\{\}$ *Extends* $\lambda_{\rightarrow}$ *(9-1)*

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 $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

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

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

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

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

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

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

Figure 11-7: Records

Records

Surface difference:

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

Records

Surface difference:

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

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 $1, 2, 3, \ldots$.

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 $1, 2, 3, \ldots$.

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.

Records

Surface difference:

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

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

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*

$$
\begin{array}{rcll}
t & ::= & \cdots & \\
 & & \mathtt{inl}\ t & \textit{tagging (left)} \\
 & & \mathtt{inr}\ t & \textit{tagging (right)} \\
 & & \mathtt{case}\ t\ \mathtt{of} & \textit{case} \\
 & & \qquad\quad \mathtt{inl}\ x \Rightarrow t & \\
 & & \qquad\ \ \mid \mathtt{inr}\ x \Rightarrow t & \\
 & & & \\
v & ::= & \cdots & \\
 & & \mathtt{inl}\ v & \textit{tagged value (left)} \\
 & & \mathtt{inr}\ v & \textit{tagged value (right)} \\
 & & & \\
\mathtt{T} & ::= & \cdots & \\
 & & \mathtt{T} + \mathtt{T} & \textit{sum type}
\end{array}
$$

Sums, example

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

$$\textit{Types}: \quad \texttt{Score} = \texttt{Int} + \texttt{Char}$$

$$\textit{Constructor}: \quad t_1 = \texttt{inl}\ 59 : \texttt{Score}, \quad t_2 = \texttt{inr}\ `F` : \texttt{Score}$$

$$\begin{aligned} \textit{Accessor}: \quad a\_good\_teacher = \quad & \lambda t.\texttt{case}\ t\ \texttt{of} \\ & \quad \texttt{inl}\ x \Rightarrow \texttt{inl}\ (max(x, 60)) \\ & \quad \mid \texttt{inr}\ x \Rightarrow \texttt{inr}\ `P` \end{aligned}$$

Sums, example

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

$$\textit{Types}: \quad \texttt{Score} = \texttt{Int} + \texttt{Char}$$

$$\textit{Constructor}: \quad t_1 = \texttt{inl}\ 59 : \texttt{Score}, \quad t_2 = \texttt{inr}\ `F` : \texttt{Score}$$

$$\begin{aligned} \textit{Accessor}: \quad a\_good\_teacher = \quad & \lambda t.\texttt{case}\ t\ \texttt{of} \\ & \quad \texttt{inl}\ x \Rightarrow \texttt{inl}\ (max(x, 60)) \\ & \mid \texttt{inr}\ x \Rightarrow \texttt{inr}\ `P` \end{aligned}$$

Quiz.

1. Give the evaluation rule and typing rule for `sum`.
2. Does the "uniqueness of types" still hold for languages with `sum`? Why?

Sums, semantics

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

$$\text{E-Case}\ \frac{t_0 \longrightarrow t_0'}{\begin{array}{l}\texttt{case}\ t_0\ \texttt{of}\ \texttt{inl}\ x_1 \Rightarrow t_1 \mid \texttt{inr}\ x_2 \Rightarrow t_2 \\ \longrightarrow \texttt{case}\ t_0'\ \texttt{of}\ \texttt{inl}\ x_1 \Rightarrow t_1 \mid \texttt{inr}\ x_2 \Rightarrow t_2\end{array}}$$

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

$$\text{T-Inl}\ \frac{\Gamma \vdash t_1 : \mathrm{T}_1}{\Gamma \vdash \texttt{inl}\ t_1 : \mathrm{T}_1 + \mathrm{T}_2}$$

$$\text{T-Case}\ \frac{\Gamma \vdash t_0 : \mathrm{T}_1 + \mathrm{T}_2 \quad \Gamma, x_1 : \mathrm{T}_1 \vdash t_1 : \mathrm{T} \quad \Gamma, x_2 : \mathrm{T}_2 \vdash t_2 : \mathrm{T}}{\texttt{case}\ t_0\ \texttt{of}\ \texttt{inl}\ x_1 \Rightarrow t_1 \mid \texttt{inr}\ x_2 \Rightarrow t_2\ :\ \mathrm{T}}$$

Symmetric rules for `inr` are omitted.

Sums and uniqueness of types

This rule breaks the uniqueness of types:

$$\textsc{T-Inl}\ \frac{\Gamma \vdash t_1 : \mathtt{T}_1}{\Gamma \vdash \mathtt{inl}\ t_1 : \mathtt{T}_1 + \mathtt{T}_2}$$

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:

$$\textsc{T-Inl}\ \frac{\Gamma \vdash t_1 : \mathtt{T}_1}{\Gamma \vdash \mathtt{inl}\ t_1 : \mathtt{T}_1 + \mathtt{T}_2}$$

Solutions include:

- Keep it as a "variable" which will be instantiated later. Mainly used in PLs with type inference.
- Allow any $\mathtt{T}_2$. We will explore this option when discussing subtyping.

Sums and uniqueness of types

This rule breaks the uniqueness of types:

$$\text{T-INL}\ \frac{\Gamma \vdash t_1 : \mathtt{T}_1}{\Gamma \vdash \mathtt{inl}\ t_1 : \mathtt{T}_1 + \mathtt{T}_2}$$

Solutions include:

- Keep it as a "variable" which will be instantiated later. Mainly used in PLs with type inference.
- Allow any $\mathtt{T}_2$. We will explore this option when discussing subtyping.
- Ascription: use explicit annotation to tell the compiler or type checker which type $\mathtt{T}_2$ is intended.

$$\mathtt{inl}\ t\ \mathsf{as}\ \mathtt{T}$$

$$\text{T-INL}\ \frac{\Gamma \vdash t_1 : \mathtt{T}_1}{\Gamma \vdash \mathtt{inl}\ t_1\ \mathsf{as}\ \mathtt{T}_1 + \mathtt{T}_2\ : \mathtt{T}_1 + \mathtt{T}_2}$$

Variants

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

$$t_1 = < \texttt{none} = \texttt{unit} > \ \texttt{as} \ < \texttt{none} : \texttt{Unit}, \texttt{some} : \texttt{Nat} >$$

$$t_2 = < \texttt{some} = 20 > \ \texttt{as} \ < \texttt{none} : \texttt{Unit}, \texttt{some} : \texttt{Nat} >$$

$$\begin{aligned} f = \ & \lambda x : \ < \texttt{none} : \texttt{Unit}, \texttt{some} : \texttt{Nat} >. \\ & \quad \texttt{case } x \texttt{ of} \\ & \qquad < \texttt{none} = \texttt{u} > \Rightarrow \ 999 \\ & \quad \ \mid \ < \texttt{some} = \texttt{v} > \Rightarrow \ v \end{aligned}$$

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.

  $$\texttt{Weekday} = < \texttt{monday} : \texttt{Unit}, \texttt{tuesday} : \texttt{Unit}, \cdots, \texttt{friday} : \texttt{Unit} >$$

  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.

  $$\texttt{DollarAmount} = < \texttt{dollars} : \texttt{Float} >$$

  $$\texttt{EuroAmount} = < \texttt{euros} : \texttt{Float} >$$

Discussion: variants v.s. Datatypes

Variant type is analogous to the ML datatype

$$\texttt{type T} = \texttt{l}_1 \texttt{ of T}_1 \mid \cdots \mid \texttt{l}_n \texttt{ of T}_n$$

But there are several differences worth noticing

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

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

- ML datatype has several additional important features:
  - Recursive datatype:
    `type NatList = nil | cons of Nat * NatList`
  - Parametric datatype:
    `type 'a List = nil | cons of 'a * 'a List`
    List is called a type operator.

General recursion

Another facility found in most programming languages is the ability to define *recursive functions*.
Here is one way to define a function `iseven`:

```
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;
```

Quiz.
What's the type of `ff`?

General recursion

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

New terms $t ::= \cdots \mid \texttt{fix}\ t$

New evaluation rules

General recursion

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

New terms $t ::= \cdots \mid \texttt{fix}\ t$

New evaluation rules

$$\textsc{E-FixBeta}\ \frac{}{\texttt{fix}\ (\lambda x : \mathtt{T}_1.t_2) \longrightarrow [x \mapsto \texttt{fix}\ (\lambda x : \mathtt{T}_1.t_2)]t_2}$$

$$\textsc{E-Fix}\ \frac{t_1 \longrightarrow t_1'}{\texttt{fix}\ t_1 \longrightarrow \texttt{fix}\ t_1'}$$

New typing rules

General recursion

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

New terms $t ::= \cdots \mid \texttt{fix}\ t$

New evaluation rules

$$\text{E-FIXBETA}\ \frac{}{\texttt{fix}\ (\lambda x : \mathrm{T}_1 . t_2) \longrightarrow [x \mapsto \texttt{fix}\ (\lambda x : \mathrm{T}_1 . t_2)] t_2}$$

$$\text{E-FIX}\ \frac{t_1 \longrightarrow t_1'}{\texttt{fix}\ t_1 \longrightarrow \texttt{fix}\ t_1'}$$

New typing rules $$\text{T-FIX}\ \frac{\Gamma \vdash t_1 : \mathrm{T}_1 \to \mathrm{T}_1}{\Gamma \vdash \texttt{fix}\ t_1 : \mathrm{T}_1}$$

General recursion

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

New terms $t ::= \cdots \mid \texttt{fix}\ t$

New evaluation rules

$$\textsc{E-FixBeta}\ \frac{}{\texttt{fix}\ (\lambda x:\mathrm{T}_1.t_2) \longrightarrow [x \mapsto \texttt{fix}\ (\lambda x:\mathrm{T}_1.t_2)]t_2}$$

$$\textsc{E-Fix}\ \frac{t_1 \longrightarrow t_1'}{\texttt{fix}\ t_1 \longrightarrow \texttt{fix}\ t_1'}$$

New typing rules $\textsc{T-Fix}\ \dfrac{\Gamma \vdash t_1 : \mathrm{T}_1 \to \mathrm{T}_1}{\Gamma \vdash \texttt{fix}\ t_1 : \mathrm{T}_1}$

New derived forms

$$\texttt{letrec}\ x:\mathrm{T}_1 = t_1\ \texttt{in}\ t_2 \stackrel{def}{=} \texttt{let}\ x = \texttt{fix}\ (\lambda x:\mathrm{T}_1.t_1)\ \texttt{in}\ t_2$$

More on `fix`

- Notice that the type $\mathtt{T}_1$ in rule $\mathrm{T\text{-}F{\scriptsize IX}}$ is not restricted to function types.
- `fix` implies that every type is inhabited by some term.

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

  $diverge_T(unit)$ 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 $\rightarrow$ and $\times$
- `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