CS520, Handout 09, 28 Sept 2006 (updated 3 Oct 2006)

Extensions of the Simply Typed Lambda Calculus (Continued)

Reading: [Pierce, Chapter 11]

Syntax of Types

The simple types are augmented (highlighted in red on the next page) in order to type extensions in the second group **[Handout 08, page 2]**.

- $A, B, C, \ldots$ are metavariables ranging over base types.
- Actual base types, in addition to `Bool` and `Nat`, are named $\mathtt{A}, \mathtt{B}, \mathtt{C}, \ldots$. Note the difference with the font of metavariables, which is ignored by Pierce.
- We collect all the type extensions discussed in [Pierce, Chapter 11] in one single definition. See next page.

Syntax of Types (continued)

All the types used in [Pierce, Chapter 11] in a single definition:

$T ::=$	*types*
A	*base type*
$\mathtt{Unit}$	***unit type***
$T \to T$	*type of functions*
$T \times T$	***type of products***
$T + T$	***type of sums***
$\{T_i{}^{i \in 1..n}\}$	***type of tuples***
$\{\ell_i : T_i{}^{i \in 1..n}\}$	***type of records***
$< \ell_i : T_i{}^{i \in 1..n} >$	***type of variants***
$\mathtt{List}\ T$	***type of lists***

Adding `unit` and `Unit` [Pierce, Section 11.2]

Syntactic form for `unit`

$t ::=$	$\ldots$	*terms*
	$\mathtt{unit}$	*constant* `unit`
$v ::=$	$\ldots$	*values*
	$\mathtt{unit}$	*constant* `unit`

Evaluation rules for `unit`

None

Typing rules for `unit`

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

The type `Unit` plays the same role as the type **void** in languages such as C and Java.
The term `unit` is a value and it is the only value of type `Unit`.

Adding sequencing [Pierce, Section 11.3]

In languages with side-effects (not yet considered), it is often useful to evaluate two or more expressions in sequence. The **sequencing notation** $t_1\,;\,t_2$ has the effect of evaluating t_1 first, throwing away its trivial result (because we are interested in its side-effect), and then going on to evaluate t_2.

Sequencing as derived form ("syntactic sugar")

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

The sequencing operator ";" associates to the left:
$t_1\,;\,t_2\,;\,t_3 \;=\; (t_1\,;\,t_2)\,;\,t_3$

Quick Exercise: We can take ";" as a primitive symbol rather than syntactic sugar. As a primitive symbol, it can be taken as a **constructor** [Pierce, Section 11.3] or as an **operator**. What is the type of ";" as an operator (not in [Pierce])?

Example borrowed from Chapter 13 in [Pierce]

```
let r = ref 5
in  (r := succ (!r); r := succ (!r); !r) ;        (* returns 7 *)
```

Term `r := succ (!r)` has type `Unit`.
Term `r := succ (!r); r := succ (!r)` has type `Unit`.
Term `r := succ (!r); r := succ (!r); !r` has type `Nat`.

Quick Exercise: What is the value returned by the above program fragment if ";" **associates to the right**?

Adding pairs [Pierce, Section 11.6]

Syntactic forms for pairs

t ::=	$\ldots$	*terms*
	$\{t, t\}$	*pair*
	$t.1$	*first projection*
	$t.2$	*second projection*
v ::=	$\ldots$	*values*
	$\{v, v\}$	*pair value*

Evaluation rules for pairs

$$\{v_1, v_2\}.1 \Rightarrow v_1 \qquad \text{(E-PairBeta1)}$$

$$\{v_1, v_2\}.2 \Rightarrow v_2 \qquad \text{(E-PairBeta2)}$$

$$\frac{t \Rightarrow t'}{t.1 \Rightarrow t'.1} \qquad \text{(E-Proj1)}$$

$$\frac{t \Rightarrow t'}{t.2 \Rightarrow t'.2} \qquad \text{(E-Proj2)}$$

$$\frac{t_1 \Rightarrow t_1'}{\{t_1, t_2\} \Rightarrow \{t_1', t_2\}} \qquad \text{(E-Pair1)}$$

$$\frac{t \Rightarrow t'}{\{v, t\} \Rightarrow \{v, t'\}} \qquad \text{(E-Pair2)}$$

Typing rules for pairs

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

$$\frac{\Gamma \vdash t : T_1 \times T_2}{\Gamma \vdash t.1 : T_1} \quad \text{(T-PROJ1)}$$

$$\frac{\Gamma \vdash t : T_1 \times T_2}{\Gamma \vdash t.2 : T_2} \quad \text{(T-PROJ2)}$$

Adding tuples [Pierce, Section 11.7]

Very similar to the treatment of pairs.

Example of a tuple which is not a pair: $\{1, 2, \mathtt{true}\}$

Adding records [Pierce, Section 11.8]

Very similar to the treatment of pairs.

Example of a record, a "labeled tuple": $\{\mathtt{a} = 1, \mathtt{b} = 2, \mathtt{c} = \mathtt{true}\}$

Adding sums [Pierce, Section 11.9]

Syntactic forms for sums

t ::=	...	*terms*
	$\mathtt{inl}\ t$	*tagging (left)*
	$\mathtt{inr}\ t$	*tagging (right)*
	$\mathtt{case}\ t\ \mathtt{of}\ \mathtt{inl}\ x \Rightarrow t \mid \mathtt{inr}\ x \Rightarrow t$	*case*
v ::=	...	*values*
	$\mathtt{inl}\ v$	*tagged value (left)*
	$\mathtt{inr}\ v$	*tagged value (right)*

Example using records and sums

```
(* two record types *)
PhysicalAddr = {firstlast: String, addr: String};
VirtualAddr  = {name: String, email: String};

(* a sum type to manipulate both records at the same time *)
Addr = PhysicalAddr + VirtualAddr;

(* if 'pa' has type PhysicalAddr then 'inl pa' has type Addr,
   if 'va' has type VirtualAddr then 'inr va' has type Addr. *)

(* extract a name from an object of type Addr *)
getName = lambda a: Addr.
   case a of
     inl x => x.firstlast
   | inr y => y.name;
```

Quick Exercise: What is the type of the function `getName`?

Another Example using records and sums

```
(* two record types *)
PhysicalAddr = {firstlast: String, addr: String, room: Nat};
VirtualAddr  = {name: String, email: String};

(* a sum type to manipulate both records at the same time *)
Addr = PhysicalAddr + VirtualAddr;

(* if 'pa' has type PhysicalAddr then 'inl pa' has type Addr,
   if 'va' has type VirtualAddr then 'inr va' has type Addr. *)

(* extract a room number or an email address from a: Addr *)
getRoomOrEmail = lambda a: Addr.
   case a of
     inl x => x.room
   | inr y => y.email;
```

Quick Exercise: What is the type of the function `getRoomOrEmail`?

Evaluation rules for sums

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

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

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

$$\frac{t \Rightarrow t'}{\texttt{inl}\ t \Rightarrow \texttt{inl}\ t'} \qquad \text{(E-Inl)}$$

$$\frac{t \Rightarrow t'}{\texttt{inr}\ t \Rightarrow \texttt{inl}\ t'} \qquad \text{(E-Inr)}$$

Typing rules for sums

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

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

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

Sums and uniqueness of types [Pierce, pages 134-135]

Can we derive both of the following typing judgments?

$$\vdash \texttt{inl}\ 3 : \texttt{Nat} + \texttt{Nat} \qquad \text{and} \qquad \vdash \texttt{inl}\ 3 : \texttt{Nat} + \texttt{Bool}$$

If yes, we have lost uniqueness of type assignments.

Adding variants [Pierce, Section 11.10]

(Labeled) variants are a generalization of **sums**,
just as **(Labeled) records** are a generalization of **tuples**.

The **syntactic forms** for variants, the **evaluation rules** for variants, and the **typing rules** for variants, are all shown in Fig. 11-11 in [Pierce, p. 136]. They are fairly involved, perhaps more than any of the others in this chapter. You should read them on your own, after reading the examples in the next few pages for motivation.

Example using records and variants (continuation of example on page 12)

```
(* two record types *)
PhysicalAddr = {firstlast: String, addr: String};
VirtualAddr  = {name: String, email: String};

(* a variant type to manipulate both records at the same time *)
Addr = <physical:PhysicalAddr, virtual:VirtualAddr>;

(* if 'pa' has type PhysicalAddr, '<physical=pa>' has type Addr,
   if 'va' has type VirtualAddr, '<virtual=va>' has type Addr. *)

(* extract a name from an object of type Addr *)
getName = lambda a: Addr.
   case a of
     <physical=x> => x.firstlast
   | <virtual=y>  => y.name;
```

Quick Exercise: What is the type of the function `getName`?

Special Cases of Variants

- example of an **enumeration** (or **enumerated type**)

```
Weekday = <mon:Unit, tue:Unit, wed:Unit, thu:Unit, fri:Unit>;

nextBusinessDay = lambda w:Weekday.
  case w of <mon=x>  =>  <tue=unit>
          | <tue=x>  =>  <wed=unit>
          | <wed=x>  =>  <thu=unit>
          | <thu=x>  =>  <fri=unit>
          | <fri=x>  =>  <mon=unit> ;
```

Does the preceding function type-check? No, according to Fig 11-11 in [Pierce, p. 136]. For "uniqueness of types" [Pierce, pp. 134-135], we insert type-ascriptions:

```
nextBusinessDay = lambda w:Weekday.
  case w of <mon=x>  =>  <tue=unit>  as Weekday
          | <tue=x>  =>  <wed=unit>  as Weekday
          | <wed=x>  =>  <thu=unit>  as Weekday
          | <thu=x>  =>  <fri=unit>  as Weekday
          | <fri=x>  =>  <mon=unit>  as Weekday ;
```


- example of a **ML datatype**

```
datatype Person = name  of String * String
                | age   of Nat
                | idnum of Nat ;
```

In the style of [Pierce] we write instead

```
Person = <name: String * String, age:Nat, idnum:Nat>;
```

See [Pierce, pp. 140-141] for a comparison between **variant types** as used in the book and **datatypes** as used in OCaml (and ML too). Pierce mentions and discusses 4 differences.

- example of a **ML option type**

In ML we can define the following type:

```
datatype OptionalNat = NONE | SOME of Nat;
```

Equivalently, in OCaml, we can also define:

```
type OptionalNat = NONE | SOME of Nat;;
```

In the style of [Pierce] we can write instead:

```
(* first define an option type *)
OptionalNat = <none:Unit, some:Nat>;

(* to model a ``table'' as a finite mapping from Nat to Nat *)
Table = Nat -> OptionalNat;

(* so that an empty table is defined by following function *)
emptyTable = lambda n:Nat. <none=unit> as OptionalNat;
```

Based on preceding definitions, how do we define a function "`extendTable`" to extend a table with an additional binding "$m \mapsto v$"?


```
(* a function to extend a table with a new binding ``(m,v)'' *)
extendTable =
  lambda t:Table. lambda m:Nat. lambda v:Nat.
    lambda n:Nat.
      if  (equal m n)
      then  <some=v> as OptionalNat
      else  t n;

(* type of ``extendTable'' *)
extendTable: Table -> Nat -> Nat -> Table

(* look up the binding of ``5'' in a table ``t'',
   return ``999'' if there is no binding for ``5'' *)
x = case t(5) of
       <none=u> => 999
     | <some=v> => v
```

Quick Exercise: Does the definition of a "`extendTable`" allow multiple bindings for the same number "m" in a table "t"? **Yes**. If there are multiple bindings for "m", which binding is returned by "$t(m)$"? **The last binding of "m" entered in "t".** Adjust the definition of "`extendTable`" so that multiple bindings are disallowed.

Adjustment of "`extendTable`" to avoid multiple bindings:

```
extendTable2 = lambda t:Table. lambda m:Nat. lambda v:Nat.
    case t(m) of
       <none=x> => lambda n:Nat.
                     if  (equal m n)
                     then  <some=v> as OptionalNat
                     else  t n
    |  <some=y> => t ;
```

Exercise: Although the preceding adjustment avoids multiple bindings, for the same "m", in a table "t", it does not allow to update the binding for "m". Write another adjustment of "`extendTable`" which avoids multiple bindings **and** allows updating.

Another function on tables – what does it do?

```
fooTable = lambda t:Table. lambda m:Nat.
    case t(m) of
       <none=x> => t
    |  <some=y> => lambda n:Nat.
                     if  (equal m n)
                     then  <none=unit> as OptionalNat
                     else  t n ;
```


Adding lists [Pierce, Section 11.12]

Syntactic forms for lists

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

Example using lists

```
letrec
   map : (Nat -> Bool) -> (List Nat) -> (List Bool) =
     lambda f : Nat -> Bool.
     lambda L : List Nat.
       if (isnil[Nat] L) then nil[Bool]
       else cons[Bool] (f (head[Nat] L)) (map f (tail[Nat] L))
in
   iseven (cons[Nat] 4 (cons[Nat] 5 (cons[Nat] 6 nil[Nat])));
```


Example using lists

```
letrec
   map : (Nat -> Bool) -> (List Nat) -> (List Bool) =
     lambda f : Nat -> Bool.
     lambda L : List Nat.
       if (isnil[Nat] L) then nil[Bool]
       else cons[Bool] (f (head[Nat] L)) (map f (tail[Nat] L))
in
   iseven (cons[Nat] 4 (cons[Nat] 5 (cons[Nat] 6 nil[Nat])));
```

Same example, without type annotations, easier to read

```
letrec
   map =
     lambda f .
     lambda L .
       if (isnil L) then nil
       else cons (f (head L)) (map f (tail L))
in
   iseven (cons 4 (cons 5 (cons 6 nil)));
```


Evaluation rules for lists

$$\frac{t_1 \Rightarrow t_1'}{\texttt{cons}[T]\; t_1\; t_2 \;\Rightarrow\; \texttt{cons}[T]\; t_1'\; t_2} \qquad \text{(E-Cons1)}$$

$$\frac{t_2 \Rightarrow t_2'}{\texttt{cons}[T]\; v_1\; t_2 \;\Rightarrow\; \texttt{cons}[T]\; v_1\; t_2'} \qquad \text{(E-Cons2)}$$

$$\texttt{isnil}[U]\,(\texttt{nil}[T]) \;\Rightarrow\; \texttt{true} \qquad \text{(E-IsnilNil)}$$

$$\texttt{isnil}[U]\,(\texttt{cons}[T]\; v_1\; v_2) \;\Rightarrow\; \texttt{false} \qquad \text{(E-IsnilCons)}$$

$$\frac{t \Rightarrow t'}{\texttt{isnil}[T]\; t \;\Rightarrow\; \texttt{isnil}[T]\; t'} \qquad \text{(E-Isnil)}$$

$$\vdots$$

(2 rules for `head`, 2 rules for `tail`)

Quick Exercise: Note the types written in **red**. Why is this not a mistake?

Typing rules for lists

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

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

$$\vdots$$

(1 rule for each of `isnil`, `head`, and `tail`)