

# Chapter 11: Simply Extensions

Basic Types / The Unit Type

Derived Forms: Sequencing and Wildcard

Ascription / Let Binding

Pairs/Tuples/Records

Sums/Variants

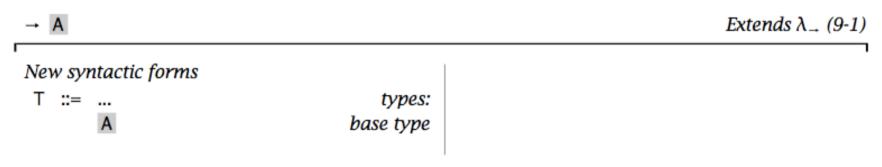
General Recursion / Lists



# Base Types



- 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, we may consider our language is equipped with some uninterpreted base types.







```
\lambda x:A. x;

<fun>: A\rightarrowA
\lambda x:B. x;

<fun>: B\rightarrowB
\lambda f:A\rightarrowA. \lambda x:A. f(f(x));

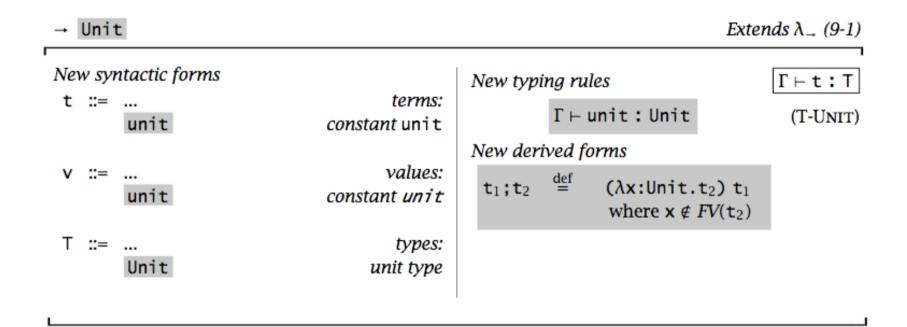
<fun>: (A\rightarrowA)\rightarrowA
```



# The Unit Type



• It is the singleton type (like void in C).



Application: Unit-type expressions care more about "side effects" rather than "results".

# Derived Form: Sequencing t<sub>1</sub>; t<sub>2</sub>



- A direct extension ( $\lambda^{E}$ )
  - † ::= ... †1 ; †2
  - New valuation relation rules

$$\frac{\mathsf{t}_1 \to \mathsf{t}_1'}{\mathsf{t}_1; \mathsf{t}_2 \to \mathsf{t}_1'; \mathsf{t}_2} \tag{E-SEQNEXT}$$
 unit;  $\mathsf{t}_2 \to \mathsf{t}_2$ 

- New typing rules

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



# Derived Form: Sequencing t<sub>1</sub>; t<sub>2</sub>



• Derived form ( $\lambda^{I}$ ): syntactic sugar

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

• Theorem [Sequencing is a derived form]: Let

$$e \in \lambda^E \rightarrow \lambda^I$$

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:Unit. $t_2$ )  $t_1$ . Then

- $t \longrightarrow_E t'$  iff  $e(t) \longrightarrow_I e(t')$
- $\Gamma \vdash^E \mathsf{t} : \mathsf{T} \text{ iff } \Gamma \vdash^I e(\mathsf{t}) : \mathsf{T}$



## Derived Form: Wildcard



• A derived form

$$\lambda$$
\_:S.t  $\rightarrow \lambda$  x:S.t

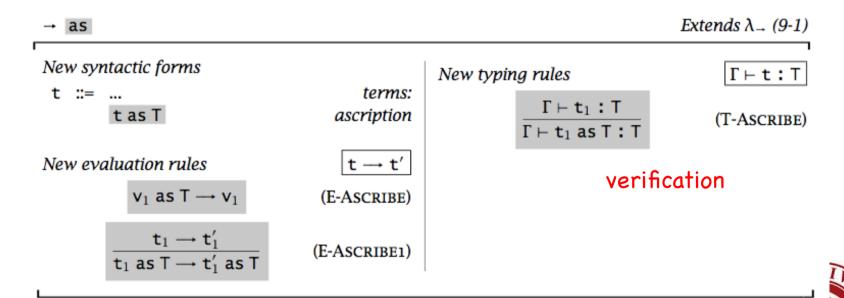
where x is some variable not occurring in t.



# Ascription: t as T



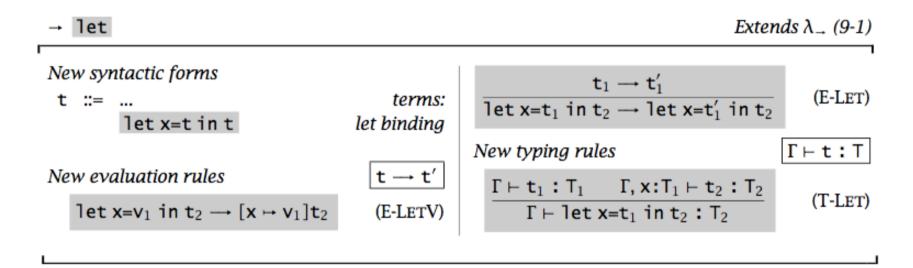
- t as T
   meaning for the term t, we ascribe the type T
  - Useful for documentation and pinpointing error sources
  - Useful for controlling type printing
  - Useful for specializing types



# Let Bindings



• To give names to some of its subexpressions.

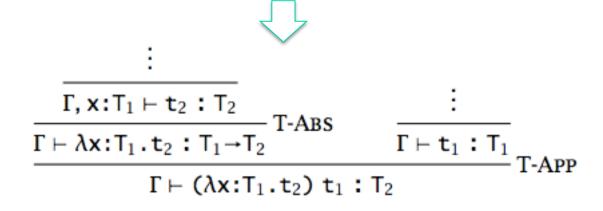






- Is "let binding" a derived form? let  $x=t_1$  in  $t_2 \rightarrow (\lambda x:T_1.t_2) t_1$
- Desugaring is not on terms but on typing derivations

$$\frac{\vdots}{\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{T}_1}{\Gamma \vdash \mathsf{let} \; \mathsf{x=t}_1 \; \mathsf{in} \; \mathsf{t}_2 : \mathsf{T}_2}} \frac{\vdots}{\Gamma, \mathsf{x} : \mathsf{T}_1 \vdash \mathsf{t}_2 : \mathsf{T}_2} \mathsf{T-LET}$$

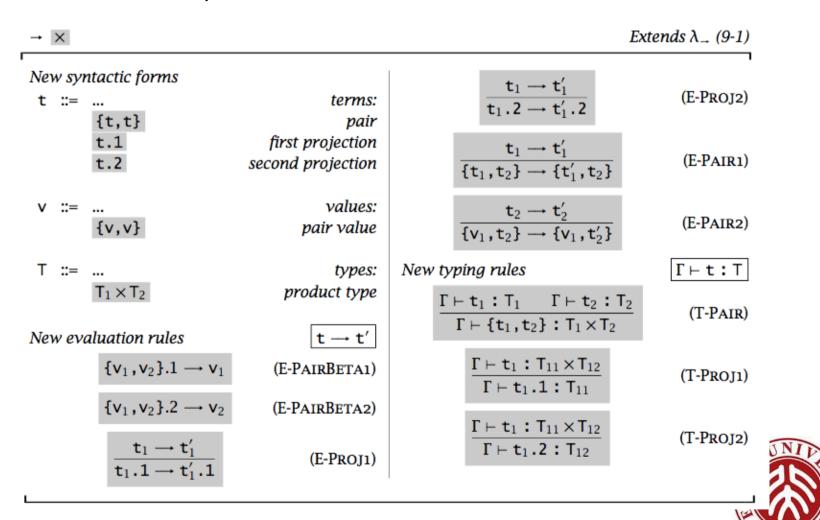




## **Pairs**



## • To build compound data structures.



# Tuples



### Generalization: binary → n-ary products

 $\rightarrow$  {}

New syntactic forms

$$\mathsf{t} ::= \dots \\ \{\mathsf{t}_i^{i \in l..n}\} \\ \mathsf{t.i}$$

tuple projection

terms:

$$\mathbf{v} ::= \dots \\ \{\mathbf{v}_i \mid i \in 1..n\}$$

values: tuple value

$$\mathsf{T} ::= \dots \\ \{\mathsf{T}_i^{\ i \in 1..n}\}$$

types: tuple type

$$\{\mathsf{v}_i^{\ i\in 1..n}\}$$
.j $\longrightarrow \mathsf{v}_j$ 

\_\_\_\_\_t → t′

$$\frac{\mathsf{t}_1 \to \mathsf{t}_1'}{\mathsf{t}_1.\mathsf{i} \to \mathsf{t}_1'.\mathsf{i}}$$

(E-Proj)

$$\frac{\mathsf{t}_j \longrightarrow \mathsf{t}_j'}{\{\mathsf{v}_i^{\ i \in 1...j-1}, \mathsf{t}_j, \mathsf{t}_k^{\ k \in j+1..n}\}} \\ \longrightarrow \{\mathsf{v}_i^{\ i \in 1...j-1}, \mathsf{t}_j', \mathsf{t}_k^{\ k \in j+1..n}\}$$

(E-TUPLE)

New typing rules

$$\frac{\text{for each } i \quad \Gamma \vdash \mathsf{t}_i : \mathsf{T}_i}{\Gamma \vdash \{\mathsf{t}_i^{i \in I..n}\} : \{\mathsf{T}_i^{i \in I..n}\}}$$

 $\Gamma \vdash \mathsf{t} : \mathsf{T}$ 

$$\frac{\Gamma \vdash \mathsf{t}_1 : \{\mathsf{T}_i^{\ i \in I..n}\}}{\Gamma \vdash \mathsf{t}_1.\,\mathsf{j} : \mathsf{T}_j}$$

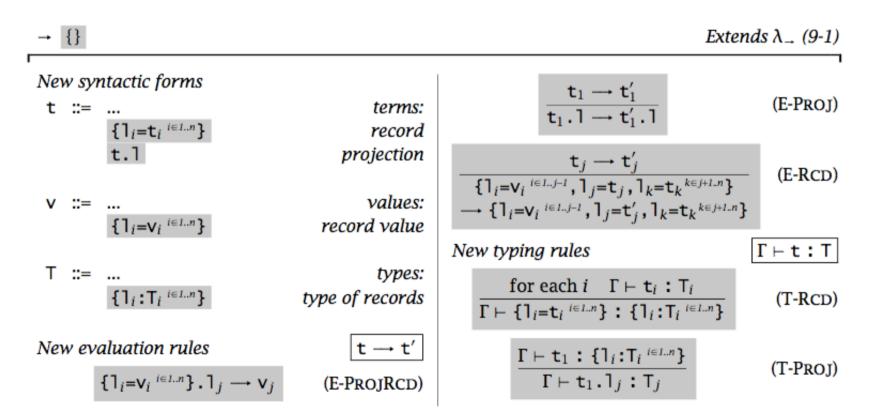
(T-Proj)



### Records



### Generalization: n-ary products → labeled records



Question: {partno=5524, cost=30.27} = {cost=30.27,partno=5524}?



### Sums



- To deal with heterogeneous collections of values.
- An Example: Address books

```
PhysicalAddr = {firstlast:String, addr:String};
VirtualAddr = {name:String, email:String};
Addr = PhysicalAddr + VirtualAddr;
```

- Injection by tagging (disjoint unions)

```
inl : PhysicalAddr → PhysicalAddr+VirtualAddr
inr : VirtualAddr → PhysicalAddr+VirtualAddr
```

- Processing by case analysis

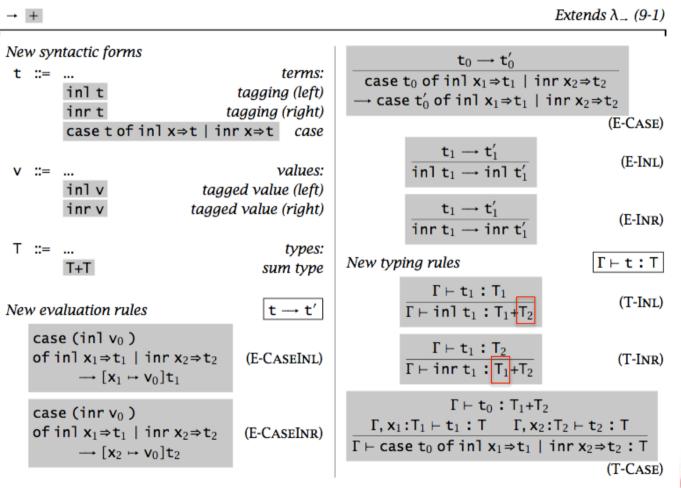
```
getName = λa:Addr.
  case a of
   inl x ⇒ x.firstlast
  | inr y ⇒ y.name;
```



## Sums



• To deal with heterogeneous collections of values.





# Sums (with Unique Typing)



 $\rightarrow$  +

Extends  $\lambda_{\rightarrow}$  (11-9)

#### New syntactic forms

#### New evaluation rules

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

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

$$\frac{\texttt{t}_1 \to \texttt{t}_1'}{\texttt{inl} \; \texttt{t}_1 \; \texttt{as} \; \texttt{T}_2 \; \to \; \texttt{inl} \; \texttt{t}_1' \; \texttt{as} \; \texttt{T}_2} \tag{E-INL}$$

$$\frac{\mathsf{t}_1 \to \mathsf{t}_1'}{\mathsf{inr}\,\mathsf{t}_1 \;\mathsf{as}\,\mathsf{T}_2 \;\to\; \mathsf{inr}\,\mathsf{t}_1' \;\mathsf{as}\,\mathsf{T}_2} \tag{E-INR}$$

#### New typing rules

 $t \rightarrow t'$ 

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{T}_1}{\Gamma \vdash \mathsf{inl} \; \mathsf{t}_1 \; \mathsf{as} \; \mathsf{T}_1 + \mathsf{T}_2} \; : \; \mathsf{T}_1 + \mathsf{T}_2} \tag{T-INL}$$

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{T}_2}{\Gamma \vdash \mathsf{inr} \; \mathsf{t}_1 \; \mathsf{as} \; \mathsf{T}_1 + \mathsf{T}_2 \; : \; \mathsf{T}_1 + \mathsf{T}_2} \tag{T-INR}$$



 $\Gamma \vdash \mathsf{t} : \mathsf{T}$ 

### Variant



Generalization: Sums → Labeled variants

```
    T1 + T2 → <l1:T1, l2:Te>
    inl † as T1+T2 → <l1=t> as <l1:T1, l2:Te>
```

• 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
```







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

New syntactic forms

t ::= ... terms: 
$$<1=t>$$
 as T tagging case t of  $<1_i=x_i>\Rightarrow t_i$   $i\in I...n$  case

T ::= ... types: 
$$< 1_i : T_i \stackrel{i \in 1..n}{>}$$
 type of variants

New evaluation rules

$$t \rightarrow t'$$

case (
$$\langle 1_j = v_j \rangle$$
 as T) of  $\langle 1_i = x_i \rangle \Rightarrow t_i \stackrel{i \in I...n}{\longrightarrow} [x_j \mapsto v_j]t_j$ 

(E-CASEVARIANT)

$$\frac{\mathsf{t}_0 \to \mathsf{t}_0'}{\mathsf{case} \; \mathsf{t}_0 \; \mathsf{of} \; \mathsf{ \Rightarrow \mathsf{t}_i \;^{i \in I..n}} \qquad \text{(E-CASE)}$$

$$\to \; \mathsf{case} \; \mathsf{t}_0' \; \mathsf{of} \; \mathsf{ \Rightarrow \mathsf{t}_i \;^{i \in I..n}$$

$$\frac{\mathtt{t}_i \longrightarrow \mathtt{t}_i'}{<\mathtt{l}_i = \mathtt{t}_i > \text{ as } \mathsf{T} \longrightarrow <\mathtt{l}_i = \mathtt{t}_i' > \text{ as } \mathsf{T}} \quad \text{(E-VARIANT)}$$

New typing rules

$$\Gamma \vdash \mathsf{t} : \mathsf{T}$$

$$\frac{\Gamma \vdash \mathsf{t}_j : \mathsf{T}_j}{\Gamma \vdash \langle \mathsf{l}_j = \mathsf{t}_j \rangle \text{ as } \langle \mathsf{l}_i : \mathsf{T}_i \stackrel{i \in I..n}{\rangle} : \langle \mathsf{l}_i : \mathsf{T}_i \stackrel{i \in I..n}{\rangle}}{(\text{T-VARIANT})}$$

$$\begin{array}{c} \Gamma \vdash \mathsf{t}_0 : < \mathsf{l}_i : \mathsf{T}_i \stackrel{i \in I..n}{>} \\ \frac{\text{for each } i \quad \Gamma, \mathsf{x}_i : \mathsf{T}_i \vdash \mathsf{t}_i : \mathsf{T}}{\Gamma \vdash \mathsf{case} \ \mathsf{t}_0 \ \mathsf{of} < \mathsf{l}_i = \mathsf{x}_i > \Rightarrow \mathsf{t}_i \stackrel{i \in I..n}{:} \mathsf{T}} \end{array} \tag{T-CASE}$$



# 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

```
V = \langle I:T \rangle
```

Operations on T cannot be applied to elements of V without first unpackaging them: a V cannot be accidentally mistaken for a T.

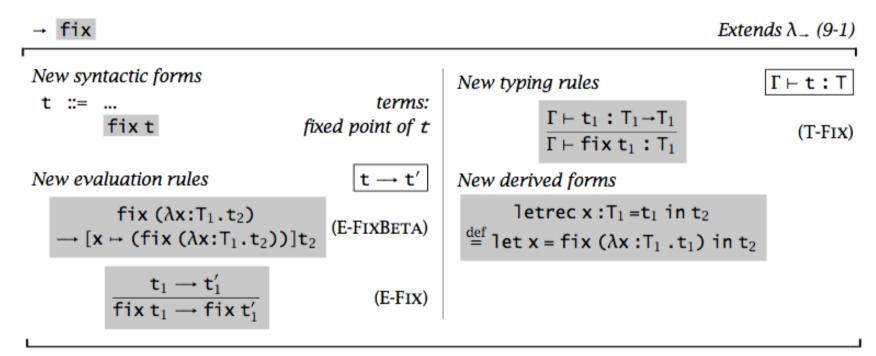


### General Recursions



• Introduce "fix" operator: fix f = f (fix f)

(It cannot be defined as a derived form in simply typed lambda calculus)







## • Example 1:





## • Example 2:

```
ff = λieio:{iseven:Nat→Bool, isodd:Nat→Bool}.
         {iseven = \lambda x:Nat.
                     if iszero x then true
                     else ieio.isodd (pred x),
          isodd = \lambda x: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
```





 Example 3: Given any type T, can you define a term that has type T?

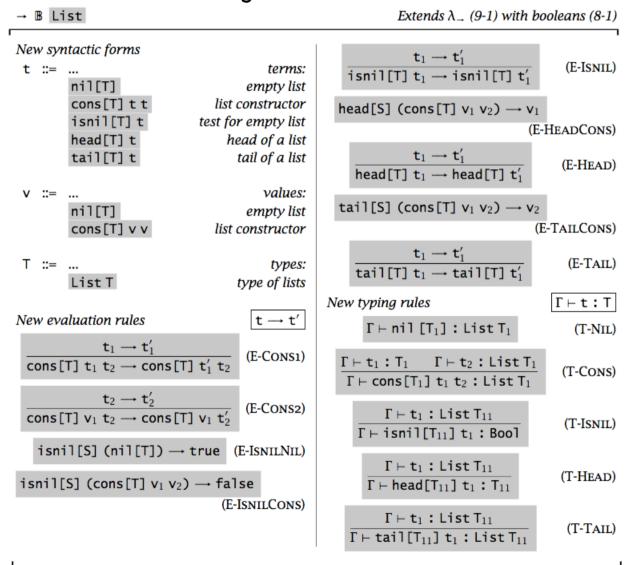
```
x \text{ as } T
fix (\lambda x:T. x)
diverge_T = \lambda_-: Unit. \ fix (\lambda x:T.x);
\bullet \ diverge_T : Unit \rightarrow T
```



## Lists



List T describes finite-length lists whose elements are drawn from T.





## Homework



- Read Chapter 11.
- Do Exercise 11.11.2.

11.11.2 EXERCISE [★]: Rewrite your definitions of plus, times, and factorial from Exercise 11.11.1 using letrec instead of fix. □

