

Chapter 11: Simply Extensions

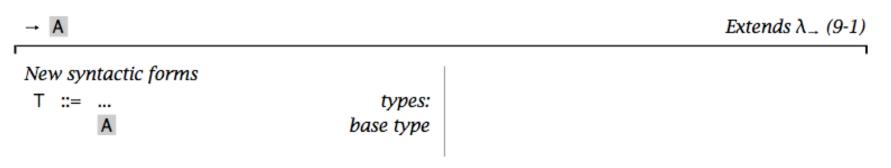
Basic Types / The Unit Type
Derived Forms: Sequencing and Wildcard
Ascription / Let Binding
Pairs/Tuples/Records
Sums/Variants
General Recursion / Lists



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.







λx:A. x;

<fun>: A→A

λx:B. x;

<fun $>: B \rightarrow B$

 $\lambda f: A \rightarrow A. \lambda x: A. f(f(x));$

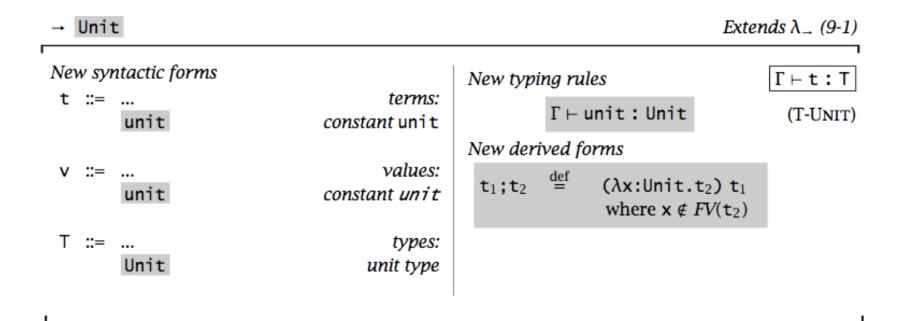
<fun $>: (A<math>\rightarrow$ A) \rightarrow A \rightarrow A



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 t₁; t₂



A direct extension λ^E

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 t₁; t₂



Derived form (λ^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

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

be the elaboration function (desugaring) that translates from the external to the internal language by replacing every occurrence of t1;t2 with $(\lambda x:Unit.t2)$ t1. 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.



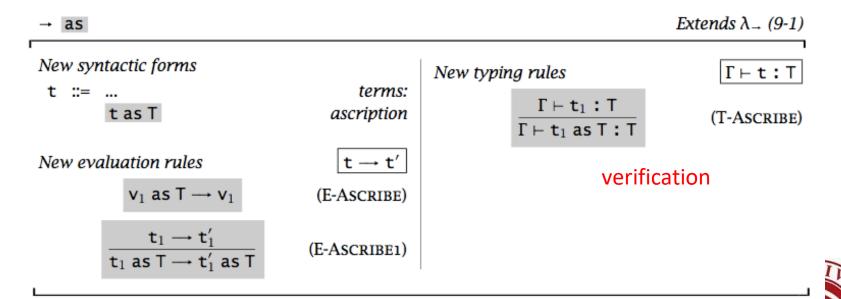
Ascription: t as T



t as T

meaning for the term t, we ascribe the type T

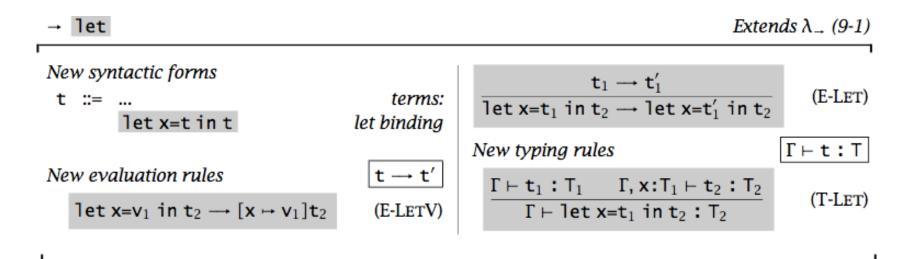
- Useful for documentation and pinpointing error sources
- Useful for controlling type printing
- Useful for specializing types



Let Bindings



To give names to some of its subexpressions.





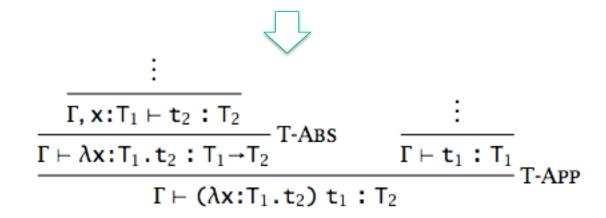


• Is "let binding" a derived form?

Yes, 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.

→ X		Ex	tends λ_{\rightarrow} (9-1)
New syntactic forms t ::= {t,t}	terms: pair	$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1.2 \longrightarrow \mathtt{t}_1'.2}$	(E-Proj2)
t.1 t.2	first projection second projection	$\frac{\mathtt{t}_1 \rightarrow \mathtt{t}_1'}{\{\mathtt{t}_1,\mathtt{t}_2\} \rightarrow \{\mathtt{t}_1',\mathtt{t}_2\}}$	(E-PAIR1)
v ∷= {v,v}	values: pair value	$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\{\mathtt{v}_1,\mathtt{t}_2\} \longrightarrow \{\mathtt{v}_1,\mathtt{t}_2'\}}$	(E-PAIR2)
T ::=	types:	New typing rules	$\Gamma \vdash t : T$
$T_1 \times T_2$	product type	$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash \{t_1, t_2\} : T_1 \times T_2}$	(T-PAIR)
New evaluation rules	t → t′	Γ . +. • T ∨ T	
$\{v_1, v_2\}.1 \rightarrow v_1$	(E-PAIRBETA1)	$\frac{\Gamma \vdash t_1 : T_{11} \times T_{12}}{\Gamma \vdash t_1 . 1 : T_{11}}$	(T-Proji)
$\{v_1, v_2\}.2 \longrightarrow v_2$	(E-PAIRBETA2)		
$\frac{\mathtt{t}_1 \to \mathtt{t}_1'}{\mathtt{t}_1.1 \to \mathtt{t}_1'.1}$	(E-PROJ1)	$\frac{\Gamma \vdash t_1 : T_{11} \times T_{12}}{\Gamma \vdash t_1 . 2 : T_{12}}$	(T-Proj2)

Tuples



Generalization: binary → n-ary products

→ {} Extends λ_{\rightarrow} (9-1)

New syntactic forms

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

$$:= ... value$$

$$\{ v_i^{i \in 1..n} \} tuple value$$

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

New evaluation rules

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

terms: tuple projection

values: tuple value

types: tuple type

$$t \rightarrow t'$$

(E-PROJTUPLE)

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

(E-Proj)

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

New typing rules

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

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

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

(E-TUPLE)

(T-TUPLE)

(T-Proj)



Records



Extends λ_{\rightarrow} (9-1)

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

Generalization: n-ary products → labeled records

New syntactic forms

t ::= ...
$$\{ \mathbf{1}_i = \mathbf{t}_i^{i \in I..n} \}$$
 t.1

$$V ::= ...$$
$$\{ \exists_{i} = \forall_{i} \ ^{i \in I..n} \}$$

$$\mathsf{T} ::= \dots \\ \{\mathsf{I}_i : \mathsf{T}_i \in I..n\}$$

New evaluation rules

$$\{ \exists_i = \mathsf{v}_i^{i \in 1..n} \} . \exists_j \longrightarrow \mathsf{v}_j$$

terms: record projection

values: record value

types: type of records

$$t \rightarrow t'$$

(E-ProjRcd)

$$\frac{\mathsf{t}_1 \to \mathsf{t}_1'}{\mathsf{t}_1.1 \to \mathsf{t}_1'.1} \tag{E-Proj}$$

$$\frac{\mathsf{t}_{j} \longrightarrow \mathsf{t}'_{j}}{\{\mathsf{l}_{i} = \mathsf{v}_{i}^{i \in I...j-1}, \mathsf{l}_{j} = \mathsf{t}_{j}, \mathsf{l}_{k} = \mathsf{t}_{k}^{k \in j+I..n}\}} \longrightarrow \{\mathsf{l}_{i} = \mathsf{v}_{i}^{i \in I...j-1}, \mathsf{l}_{j} = \mathsf{t}'_{j}, \mathsf{l}_{k} = \mathsf{t}_{k}^{k \in j+I..n}\}$$
(E-RCD)

New typing rules

$$\frac{\text{for each } i \quad \Gamma \vdash \mathsf{t}_i : \mathsf{T}_i}{\Gamma \vdash \{\mathsf{l}_i = \mathsf{t}_i \stackrel{i \in I..n}{}\} : \{\mathsf{l}_i : \mathsf{T}_i \stackrel{i \in I..n}{}\}}$$
 (T-RCD)

$$\frac{\Gamma \vdash \mathsf{t}_1 : \{\mathsf{I}_i : \mathsf{T}_i \stackrel{i \in I..n}{\}}}{\Gamma \vdash \mathsf{t}_1 . \mathsf{I}_j : \mathsf{T}_j} \tag{T-Proj}$$

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

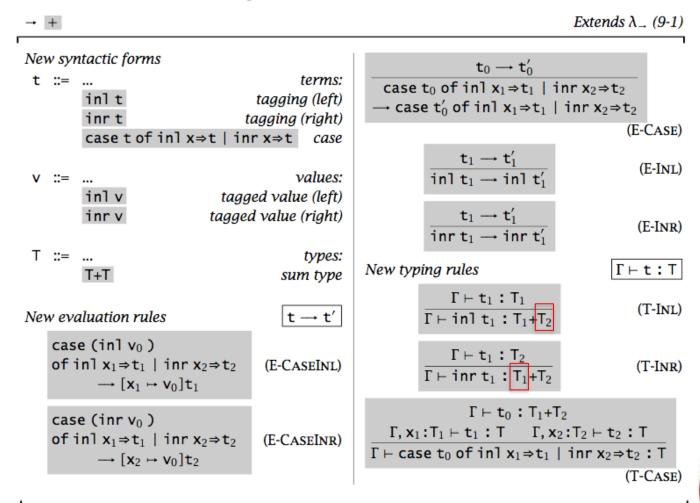
```
getName = λa:Addr.
  case a of
   inl x ⇒ x.firstlast
  | inr y ⇒ y.name;
```



Sums



To deal with heterogeneous collections of values.





Sums (with Unique Typing)



 \rightarrow +

Extends λ_{\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)
 $\rightarrow [x_2 \mapsto v_0]t_2$

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

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

New typing rules

 $t \rightarrow t'$

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

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



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

Variant



- Generalization: Sums → Labeled variants
 - T1 + T2 → <|1:T1, |2:T2>
 - inl t as T1+T2 → <l1=t> as <l1:T1, l2:T2>
- 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 λ_{\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:
$$< l_i : T_i \stackrel{i \in 1..n}{>}$$
 type of variants

New evaluation rules

$$t \rightarrow t'$$

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

(E-CASEVARIANT)

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

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

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

New typing rules

$$\Gamma \vdash t : T$$

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

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

$$V = < I:T >$$

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

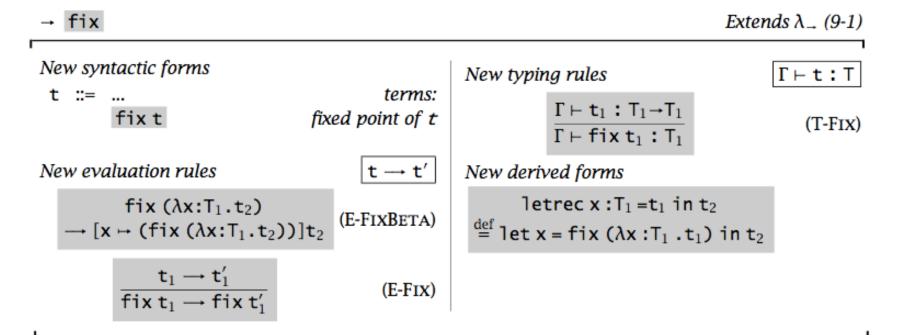


General Recursions



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

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







• Example 1:

```
ff = \lambdaie:Nat\rightarrowBool.
         λx:Nat.
            if iszero x then true
            else if iszero (pred x) then false
            else ie (pred (pred x));
▶ ff : (Nat→Bool) → Nat → Bool
  iseven = fix ff;
▶ iseven : Nat → Bool
  iseven 7;
► false : Bool
```





• 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 as T
fix (λx:T. x)

diverge<sub>T</sub> = λ_:Unit. fix (λx:T.x);

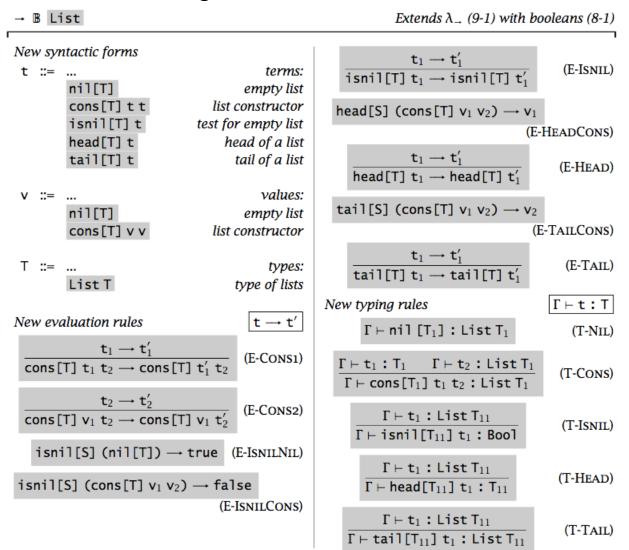
diverge<sub>T</sub> : Unit → T
```



Lists



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





Homework



- Read Chapter 11.
- Do Exercise 11.11.1.

11.11.1 EXERCISE [★★]: Define equal, plus, times, and factorial using fix.

Next class is programming exercise. Please download arith and fullsimple. Bring your laptop.

