On to real programming languages...

The Unit type

unit is like "void"

$$\begin{array}{lll} \texttt{t} & ::= & \ldots \\ & & \texttt{unit} \end{array}$$

terms
constant unit

$$\begin{array}{lll} \texttt{v} & ::= & \ldots \\ & & \texttt{unit} \end{array}$$

values
constant *unit*

$$\begin{array}{lll} \texttt{T} & ::= & \ldots \\ & & \texttt{Unit} \end{array}$$

types
unit type

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

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

Sequencing

$\texttt{t} \quad ::= \quad \ldots$ *terms*

$\texttt{t}_1 \texttt{;} \texttt{t}_2$

Sequencing

unit is like "void"

$$\texttt{t} ::= \ldots \qquad \textit{terms}$$
$$\texttt{t}_1\texttt{;}\texttt{t}_2$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{t}_1\texttt{;}\texttt{t}_2 \longrightarrow \texttt{t}_1'\texttt{;}\texttt{t}_2} \qquad \text{(E-Seq)}$$

$$\texttt{unit;}\texttt{t}_2 \longrightarrow \texttt{t}_2 \qquad \text{(E-SeqNext)}$$

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{Unit} \qquad \Gamma \vdash \texttt{t}_2 : \texttt{T}_2}{\Gamma \vdash \texttt{t}_1\texttt{;}\texttt{t}_2 : \texttt{T}_2} \qquad \text{(T-Seq)}$$

Derived forms

- Syntatic sugar
- Internal language vs. external (surface) language

Sequencing as a derived form

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

$$\text{where } \mathtt{x} \notin \mathit{FV}(\mathtt{t_2})$$

Ascription

documentation, enforcing types, catching potential bugs

New syntactic forms

$\texttt{t} ::= \ldots$ — *terms*

$\texttt{t as T}$ — *ascription*

New evaluation rules $\boxed{\texttt{t} \longrightarrow \texttt{t}'}$

$$\texttt{v}_1 \texttt{ as T} \longrightarrow \texttt{v}_1 \qquad \text{(E-Ascribe)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{t}_1 \texttt{ as T} \longrightarrow \texttt{t}_1' \texttt{ as T}} \qquad \text{(E-Ascribe1)}$$

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

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{T}}{\Gamma \vdash \texttt{t}_1 \texttt{ as T} : \texttt{T}} \qquad \text{(T-Ascribe)}$$

Haskell type annotation

```
plus :: a->a->a
plus x y = x + y
```

Ascription as a derived form

$$\texttt{t as T} \stackrel{\text{def}}{=} (\lambda \texttt{x:T. x) t}$$

Let-bindings

Prelude> let y = let x=6 in x+2
Prelude> y
8

New syntactic forms

$\texttt{t} ::= \ldots$ *terms*

$\quad \texttt{let x=t in t}$ *let binding*

New evaluation rules $\boxed{\texttt{t} \longrightarrow \texttt{t}'}$

$$\texttt{let x=v}_1 \texttt{ in t}_2 \longrightarrow [\texttt{x} \mapsto \texttt{v}_1]\texttt{t}_2 \qquad \text{(E-LetV)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{let x=t}_1 \texttt{ in t}_2 \longrightarrow \texttt{let x=t}_1' \texttt{ in t}_2} \qquad \text{(E-Let)}$$

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

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{T}_1 \qquad \Gamma, \texttt{x}:\texttt{T}_1 \vdash \texttt{t}_2 : \texttt{T}_2}{\Gamma \vdash \texttt{let x=t}_1 \texttt{ in t}_2 : \texttt{T}_2} \qquad \text{(T-Let)}$$

Derived Form for let binding?

$$\texttt{let}\ x=t_1\ \texttt{in}\ t_2 \quad \stackrel{\text{def}}{=} \quad (\lambda x{:}T_1.t_2)\ t_1$$

New typing rules

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

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \texttt{let}\ x=t_1\ \texttt{in}\ t_2 : T_2} \quad \text{(T-Let)}$$

Pairs, tuples, and records

Pairs

t	::=	...	*terms*
		{t,t}	*pair*
		t.1	*first projection*
		t.2	*second projection*
v	::=	...	*values*
		{v,v}	*pair value*
T	::=	...	*types*
		$T_1 \times T_2$	*product type*

Evaluation rules for pairs

how about call-by-name?

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

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

$$\frac{t_1 \longrightarrow t_1'}{t_1.1 \longrightarrow t_1'.1} \qquad \text{(E-Proj1)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1.2 \longrightarrow t_1'.2} \qquad \text{(E-Proj2)}$$

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

$$\frac{t_2 \longrightarrow t_2'}{\{v_1, t_2\} \longrightarrow \{v_1, t_2'\}} \qquad \text{(E-Pair2)}$$

Typing rules for pairs

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \{\mathtt{t}_1,\mathtt{t}_2\} : \mathtt{T}_1 \times \mathtt{T}_2} \qquad \text{(T-Pair)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \times \mathtt{T}_{12}}{\Gamma \vdash \mathtt{t}_1.\mathtt{1} : \mathtt{T}_{11}} \qquad \text{(T-Proj1)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \times \mathtt{T}_{12}}{\Gamma \vdash \mathtt{t}_1.\mathtt{2} : \mathtt{T}_{12}} \qquad \text{(T-Proj2)}$$

Tuples

t	::=	...	*terms*
		$\{t_i\ ^{i \in 1..n}\}$	*tuple*
		t.i	*projection*
v	::=	...	*values*
		$\{v_i\ ^{i \in 1..n}\}$	*tuple value*
T	::=	...	*types*
		$\{T_i\ ^{i \in 1..n}\}$	*tuple type*

Evaluation rules for tuples

$$\{v_i\ ^{i\in 1..n}\}.j \longrightarrow v_j \qquad \text{(E-ProjTuple)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1.i \longrightarrow t_1'.i} \qquad \text{(E-Proj)}$$

$$\frac{t_j \longrightarrow t_j'}{\begin{array}{l}\{v_i\ ^{i\in 1..j-1}, t_j, t_k\ ^{k\in j+1..n}\}\\ \longrightarrow \{v_i\ ^{i\in 1..j-1}, t_j', t_k\ ^{k\in j+1..n}\}\end{array}} \qquad \text{(E-Tuple)}$$

how about big-step eval?

Typing rules for tuples

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

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

Records

tuples with "labelled" components

$\mathtt{t}$	::=	...	*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*

C: "struct" type; PASCAL: "record" type

Evaluation rules for records

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

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

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

Typing rules for records

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

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

Sums and variants

```
data Ast = TRUE
         | FALSE
         | IFTHENELSE Ast Ast Ast
         | PAIR Ast Ast
         | FST Ast
         | SND Ast
```

C: “union” type

Sums – motivating example

```
PhysicalAddr = {firstlast:String, addr:String}
VirtualAddr  = {name:String, email:String}
Addr         = PhysicalAddr + VirtualAddr
inl   :   "PhysicalAddr → PhysicalAddr+VirtualAddr"
inr   :   "VirtualAddr → PhysicalAddr+VirtualAddr"
```

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

inject left (tagging)

inject right (tagging)

```
data Ast = FST Ast
         | SND Ast
```

```
type2 t = case t of
     FST (PAIR t1 t2) -> type2 t1
     SND (PAIR t1 t2) -> type2 t2
```

Variants

Just as we generalized binary products to labeled records, we can generalize binary sums to labeled *variants*.

```
data Ast = TRUE
         | FALSE
         | IFTHENELSE Ast Ast Ast
         | PAIR Ast Ast
         | FST Ast
         | SND Ast
```

New syntactic forms

t	::=	...	*terms*	
		$\texttt{<l=t> as T}$	*tagging*	variant
		$\texttt{case t of <l}_i\texttt{=x}_i\texttt{>}\Rightarrow \texttt{t}_i\ ^{i \in 1..n}$	*case*	using variant
T	::=	...	*types*	
		$\texttt{<l}_i\texttt{:T}_i\ ^{i \in 1..n}\texttt{>}$	*type of variants*	

```
t = <l1={3,True}.1> as <l1:int,l1:Bool>
```

variant

```
case t of <l1=x1>=>(iszro x1)
          <l2=x2>=>(not x2)
```

using variant

New evaluation rules

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

$$\texttt{case } (\texttt{<}\mathtt{l}_j\texttt{=}\mathtt{v}_j\texttt{>} \texttt{ as T}) \texttt{ of } \texttt{<}\mathtt{l}_i\texttt{=}\mathtt{x}_i\texttt{>}\Rightarrow\mathtt{t}_i \ ^{i\in 1..n} \longrightarrow [\mathtt{x}_j \mapsto \mathtt{v}_j]\mathtt{t}_j \quad \text{(E-CaseVariant)}$$

$$\frac{\mathtt{t}_0 \longrightarrow \mathtt{t}_0'}{\texttt{case } \mathtt{t}_0 \texttt{ of } \texttt{<}\mathtt{l}_i\texttt{=}\mathtt{x}_i\texttt{>}\Rightarrow\mathtt{t}_i \ ^{i\in 1..n} \longrightarrow \texttt{case } \mathtt{t}_0' \texttt{ of } \texttt{<}\mathtt{l}_i\texttt{=}\mathtt{x}_i\texttt{>}\Rightarrow\mathtt{t}_i \ ^{i\in 1..n}} \quad \text{(E-Case)}$$

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

```
   case <l1={3,True}.1> as <l1:int,l2:Bool> of <l1=x1> => (iszro x1)
                                               <l2=x2> => (not x2)
-> case <l1=3> as <l1:int,l2:Bool> of <l1=x1> => (iszro x1)
                                      <l2=x2> => (not x2)
-> FALSE
```

the first step uses E-Case, E-Variant, and E-PairBeta1 (2 congruence and 1 computation rule); the second step uses E-CaseVariant (1 computation rule). also note that $\texttt{<l}_1\texttt{=3> as <l}_1\texttt{:int,l}_1\texttt{:Bool>}$ *is not a value but a normal form.*

each single-step eval uses exactly 1 computation rule and 0+ congruence rules.

New typing rules

$\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\Gamma \vdash \mathtt{t}_j : \mathtt{T}_j}{\Gamma \vdash \mathtt{<l}_j\mathtt{=t}_j\mathtt{>}\ \mathtt{as}\ \mathtt{<l}_i\mathtt{:T}_i\ ^{i \in 1..n}\mathtt{>} : \mathtt{<l}_i\mathtt{:T}_i\ ^{i \in 1..n}\mathtt{>}} \quad \text{(T-Variant)}$$

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{<l}_i\mathtt{:T}_i\ ^{i \in 1..n}\mathtt{>} \qquad \text{for each } i \quad \Gamma, \mathtt{x}_i\mathtt{:T}_i \vdash \mathtt{t}_i : \mathtt{T}}{\Gamma \vdash \mathtt{case}\ \mathtt{t}_0\ \mathtt{of}\ \mathtt{<l}_i\mathtt{=x}_i\mathtt{>} \Rightarrow \mathtt{t}_i\ ^{i \in 1..n} : \mathtt{T}} \quad \text{(T-Case)}$$

```
|- <l1={3,True}.1> as <l1:int,l2:Bool>    :    <l1:int,l2:Bool>

|- case <l1={3,True}.1> as <l1:int,l2:Bool> of <l1=x1> => (iszro x1)
                                               <l2=x2> => (not x2)     : Bool
```

a variant has to annotate the full type (i.e., other possibilities).

this is different from the Haskell/Ocaml solution where constructors (labels) have different names and each name only occur in one variant type.

```
data Ast = TRUE
         | FALSE
         | IFTHENELSE Ast Ast Ast
         | PAIR Ast Ast
         | FST Ast
         | SND Ast
```

Example

```
Addr = <physical:PhysicalAddr, virtual:VirtualAddr>;

a = <physical=pa> as Addr;

getName = λa:Addr.
  case a of
    <physical=x> ⇒ x.firstlast
  | <virtual=y> ⇒ y.name;
```