

Foundations of Software Winter Semester 2007

Week 7

Plan

PREVIOUSLY: unit, sequencing, let, pairs, tuples

TODAY:

1. options, variants
2. recursion
3. state

NEXT: exceptions?

NEXT: polymorphic (not so simple) typing

Records

$t ::= \dots$	<i>terms</i>
$\{l_i = t_i \mid i \in 1..n\}$	<i>record</i>
$t.l$	<i>projection</i>
$v ::= \dots$	<i>values</i>
$\{l_i = v_i \mid i \in 1..n\}$	<i>record value</i>
$T ::= \dots$	<i>types</i>
$\{l_i : T_i \mid i \in 1..n\}$	<i>type of records</i>

Evaluation rules for records

$$\{l_i = v_i \mid i \in 1..n\}.l_j \longrightarrow v_j \quad (\text{E-PROJRCD})$$

$$\frac{t_1 \longrightarrow t'_1}{t_1.l \longrightarrow t'_1.l} \quad (\text{E-PROJ})$$

$$\frac{t_j \longrightarrow t'_j}{\{l_i = v_i \mid i \in 1..j-1, l_j = t_j, l_k = t_k \mid k \in j+1..n\} \longrightarrow \{l_i = v_i \mid i \in 1..j-1, l_j = t'_j, l_k = t_k \mid k \in j+1..n\}} \quad (\text{E-RCD})$$

Typing rules for records

$$\frac{\text{for each } i \quad \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i = t_i \mid i \in 1..n\} : \{l_i : T_i \mid i \in 1..n\}} \quad (\text{T-RCD})$$

$$\frac{\Gamma \vdash t_1 : \{l_i : T_i \mid i \in 1..n\}}{\Gamma \vdash t_1.l_j : T_j} \quad (\text{T-PROJ})$$

Sums and variants

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;
```

New syntactic forms

<code>t ::= ...</code>	<i>terms</i>
<code>inl t</code>	<i>tagging (left)</i>
<code>inr t</code>	<i>tagging (right)</i>
<code>case t of inl x⇒t inr x⇒t</code>	<i>case</i>
<code>v ::= ...</code>	<i>values</i>
<code>inl v</code>	<i>tagged value (left)</i>
<code>inr v</code>	<i>tagged value (right)</i>
<code>T ::= ...</code>	<i>types</i>
<code>T+T</code>	<i>sum type</i>

T_1+T_2 is a *disjoint union* of T_1 and T_2 (the tags `inl` and `inr` ensure disjointness)

New evaluation rules

$t \longrightarrow t'$

$$\begin{array}{l} \text{case (inl } v_0) \\ \text{of inl } x_1 \Rightarrow t_1 \mid \text{inr } x_2 \Rightarrow t_2 \end{array} \longrightarrow [x_1 \mapsto v_0]t_1 \quad (\text{E-CASEINL})$$

$$\begin{array}{l} \text{case (inr } v_0) \\ \text{of inl } x_1 \Rightarrow t_1 \mid \text{inr } x_2 \Rightarrow t_2 \end{array} \longrightarrow [x_2 \mapsto v_0]t_2 \quad (\text{E-CASEINR})$$

$$\frac{t_0 \longrightarrow t'_0}{\text{case } t_0 \text{ of inl } x_1 \Rightarrow t_1 \mid \text{inr } x_2 \Rightarrow t_2 \longrightarrow \text{case } t'_0 \text{ of inl } x_1 \Rightarrow t_1 \mid \text{inr } x_2 \Rightarrow t_2} \quad (\text{E-CASE})$$

$$\frac{t_1 \longrightarrow t'_1}{\text{inl } t_1 \longrightarrow \text{inl } t'_1} \quad (\text{E-INL})$$

$$\frac{t_1 \longrightarrow t'_1}{\text{inr } t_1 \longrightarrow \text{inr } t'_1} \quad (\text{E-INR})$$

New typing rules

$\Gamma \vdash t : T$

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \text{inl } t_1 : T_1 + T_2} \quad (\text{T-INL})$$

$$\frac{\Gamma \vdash t_1 : T_2}{\Gamma \vdash \text{inr } t_1 : T_1 + T_2} \quad (\text{T-INR})$$

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

Sums and Uniqueness of Types

Problem:

If t has type T , then `inl t` has type $T+U$ for every U .

I.e., we've lost uniqueness of types.

Possible solutions:

- ▶ “Infer” U as needed during typechecking
- ▶ Give constructors different names and only allow each name to appear in one sum type (requires generalization to “variants,” which we'll see next) — OCaml's solution
- ▶ Annotate each `inl` and `inr` with the intended sum type.

For simplicity, let's choose the third.

New syntactic forms

<code>t ::= ...</code>	<i>terms</i>
<code>inl t as T</code>	<i>tagging (left)</i>
<code>inr t as T</code>	<i>tagging (right)</i>
<code>v ::= ...</code>	<i>values</i>
<code>inl v as T</code>	<i>tagged value (left)</i>
<code>inr v as T</code>	<i>tagged value (right)</i>

Note that `as T` here is not the ascription operator that we saw before — i.e., not a separate syntactic form: in essence, there is an ascription “built into” every use of `inl` or `inr`.

New typing rules

$\Gamma \vdash t : T$

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \text{inl } t_1 \text{ as } T_1+T_2 : T_1+T_2} \quad (\text{T-INL})$$

$$\frac{\Gamma \vdash t_1 : T_2}{\Gamma \vdash \text{inr } t_1 \text{ as } T_1+T_2 : T_1+T_2} \quad (\text{T-INR})$$

Evaluation rules ignore annotations:

$t \longrightarrow t'$

$$\begin{array}{l} \text{case (inl } v_0 \text{ as } T_0) \\ \text{of inl } x_1 \Rightarrow t_1 \mid \text{inr } x_2 \Rightarrow t_2 \\ \longrightarrow [x_1 \mapsto v_0]t_1 \end{array} \quad (\text{E-CASEINL})$$

$$\begin{array}{l} \text{case (inr } v_0 \text{ as } T_0) \\ \text{of inl } x_1 \Rightarrow t_1 \mid \text{inr } x_2 \Rightarrow t_2 \\ \longrightarrow [x_2 \mapsto v_0]t_2 \end{array} \quad (\text{E-CASEINR})$$

$$\frac{t_1 \longrightarrow t'_1}{\text{inl } t_1 \text{ as } T_2 \longrightarrow \text{inl } t'_1 \text{ as } T_2} \quad (\text{E-INL})$$

$$\frac{t_1 \longrightarrow t'_1}{\text{inr } t_1 \text{ as } T_2 \longrightarrow \text{inr } t'_1 \text{ as } T_2} \quad (\text{E-INR})$$

Variants

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

New syntactic forms

<code>t ::= ...</code>	<i>terms</i>
<code> <l=t> as T</code>	<i>tagging</i>
<code> case t of <l_i=x_i>⇒t_i^{i∈1..n}</code>	<i>case</i>
<code>T ::= ...</code>	<i>types</i>
<code> <l_i:T_i^{i∈1..n}></code>	<i>type of variants</i>

New evaluation rules

$t \longrightarrow t'$

$\text{case } \langle l_j = v_j \rangle \text{ as } T \text{ of } \langle l_i = x_i \rangle \Rightarrow t_i^{i \in 1..n}$ (E-CASEVARIANT)
 $\longrightarrow [x_j \mapsto v_j] t_j$

$$\frac{t_0 \longrightarrow t'_0}{\text{case } t_0 \text{ of } \langle l_i = x_i \rangle \Rightarrow t_i^{i \in 1..n} \longrightarrow \text{case } t'_0 \text{ of } \langle l_i = x_i \rangle \Rightarrow t_i^{i \in 1..n}}$$
 (E-CASE)

$$\frac{t_i \longrightarrow t'_i}{\langle l_i = t_i \rangle \text{ as } T \longrightarrow \langle l_i = t'_i \rangle \text{ as } T}$$
 (E-VARIANT)

New typing rules

$\Gamma \vdash t : T$

$$\frac{\Gamma \vdash t_j : T_j}{\Gamma \vdash \langle l_j = t_j \rangle \text{ as } \langle l_i : T_i^{i \in 1..n} \rangle : \langle l_i : T_i^{i \in 1..n} \rangle}$$
 (T-VARIANT)

$$\frac{\Gamma \vdash t_0 : \langle l_i : T_i^{i \in 1..n} \rangle \text{ for each } i \quad \Gamma, x_i : T_i \vdash t_j : T}{\Gamma \vdash \text{case } t_0 \text{ of } \langle l_i = x_i \rangle \Rightarrow t_j^{i \in 1..n} : T}$$
 (T-CASE)

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;
```

Options

Just like in OCaml...

```
OptionalNat = <none:Unit, some:Nat>;  
  
Table = Nat → OptionalNat;  
  
emptyTable = λn:Nat. <none=unit> as OptionalNat;  
  
extendTable =  
  λt:Table. λm:Nat. λv:Nat.  
  λn:Nat.  
    if equal n m then <some=v> as OptionalNat  
  else t n;  
  
x = case t(5) of  
  <none=u> ⇒ 999  
  | <some=v> ⇒ v;
```

Enumerations

```
Weekday = <monday:Unit, tuesday:Unit, wednesday:Unit,  
          thursday:Unit, friday:Unit>;
```

```
nextBusinessDay = λw:Weekday.
```

```
  case w of <monday=x>    ⇒ <tuesday=unit> as Weekday  
           | <tuesday=x>   ⇒ <wednesday=unit> as Weekday  
           | <wednesday=x> ⇒ <thursday=unit> as Weekday  
           | <thursday=x> ⇒ <friday=unit> as Weekday  
           | <friday=x>   ⇒ <monday=unit> as Weekday;
```