CIS 500

Software Foundations

Fall 2005

31 October, 2005

Simple Extensions

Plan

- In lecture we're going to cover a few simple extensions of the typed-lambda calculus, from TAPL Chapter 11.
  1. Products, records
  2. Sums, variants
  3. Recursion
- Homework 6 covers some extensions from Chapter 11 that we haven't talked about: ascription and lists.
- You should also read Chapter 10, and bring questions about it to the recitation.
- We're skipping Chapter 12.

Products

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

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

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

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

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

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

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

Typing rules for pairs

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

$$\frac{\Gamma \vdash t_1 : T_{11} \times T_{12}}{\Gamma \vdash t_1.1 : T_{11}} \qquad \text{(T-PROJ1)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \times T_{12}}{\Gamma \vdash t_1.2 : T_{12}} \qquad \text{(T-PROJ2)}$$

Records

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

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

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

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

Sums

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

`t`	::=	...	*terms*
		`inl t`	*tagging (left)*
		`inr t`	*tagging (right)*
		`case t of inl x⇒t | inr x⇒t`	*case*
`v`	::=	...	*values*
		`inl v`	*tagged value (left)*
		`inr v`	*tagged value (right)*
`T`	::=	...	*types*
		`T+T`	*sum type*

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

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

$$\begin{array}{ll}\texttt{case (inl } \texttt{v}_0\texttt{)} & \longrightarrow [\texttt{x}_1 \mapsto \texttt{v}_0]\texttt{t}_1 \\ \texttt{of inl } \texttt{x}_1 \Rightarrow \texttt{t}_1 \texttt{ | inr } \texttt{x}_2 \Rightarrow \texttt{t}_2 & \end{array} \quad \text{(E-CaseInl)}$$

$$\begin{array}{ll}\texttt{case (inr } \texttt{v}_0\texttt{)} & \longrightarrow [\texttt{x}_2 \mapsto \texttt{v}_0]\texttt{t}_2 \\ \texttt{of inl } \texttt{x}_1 \Rightarrow \texttt{t}_1 \texttt{ | inr } \texttt{x}_2 \Rightarrow \texttt{t}_2 & \end{array} \quad \text{(E-CaseInr)}$$

$$\frac{\texttt{t}_0 \longrightarrow \texttt{t}_0'}{\begin{array}{c}\texttt{case } \texttt{t}_0 \texttt{ of inl } \texttt{x}_1 \Rightarrow \texttt{t}_1 \texttt{ | inr } \texttt{x}_2 \Rightarrow \texttt{t}_2 \\ \longrightarrow \texttt{case } \texttt{t}_0' \texttt{ of inl } \texttt{x}_1 \Rightarrow \texttt{t}_1 \texttt{ | inr } \texttt{x}_2 \Rightarrow \texttt{t}_2\end{array}} \quad \text{(E-Case)}$$

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

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

New typing rules

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

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{T_1}}{\Gamma \vdash \mathtt{inl}\ \mathtt{t_1} : \mathtt{T_1}\mathtt{+}\mathtt{T_2}} \qquad \text{(T-Inl)}$$

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathtt{T_2}}{\Gamma \vdash \mathtt{inr}\ \mathtt{t_1} : \mathtt{T_1}\mathtt{+}\mathtt{T_2}} \qquad \text{(T-Inr)}$$

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

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

```
t ::= ...                   terms
      inl t as T            tagging (left)
      inr t as T            tagging (right)

v ::= ...                   values
      inl v as T            tagged value (left)
      inr v as T            tagged value (right)
```

New typing rules

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

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

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_2}{\Gamma \vdash \mathtt{inr}\ \mathtt{t}_1\ \mathtt{as}\ \mathtt{T}_1\mathtt{+}\mathtt{T}_2 : \mathtt{T}_1\mathtt{+}\mathtt{T}_2} \quad \text{(T-INR)}$$

Evaluation rules ignore annotations:

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

$$\begin{array}{l}\texttt{case (inl } \texttt{v}_0 \texttt{ as } \texttt{T}_0\texttt{)} \\ \texttt{of inl } \texttt{x}_1 \Rightarrow \texttt{t}_1 \texttt{ | inr } \texttt{x}_2 \Rightarrow \texttt{t}_2 \\ \qquad \longrightarrow [\texttt{x}_1 \mapsto \texttt{v}_0]\texttt{t}_1\end{array} \qquad \text{(E-CaseInl)}$$

$$\begin{array}{l}\texttt{case (inr } \texttt{v}_0 \texttt{ as } \texttt{T}_0\texttt{)} \\ \texttt{of inl } \texttt{x}_1 \Rightarrow \texttt{t}_1 \texttt{ | inr } \texttt{x}_2 \Rightarrow \texttt{t}_2 \\ \qquad \longrightarrow [\texttt{x}_2 \mapsto \texttt{v}_0]\texttt{t}_2\end{array} \qquad \text{(E-CaseInr)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{inl } \texttt{t}_1 \texttt{ as } \texttt{T}_2 \longrightarrow \texttt{inl } \texttt{t}_1' \texttt{ as } \texttt{T}_2} \qquad \text{(E-Inl)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{inr } \texttt{t}_1 \texttt{ as } \texttt{T}_2 \longrightarrow \texttt{inr } \texttt{t}_1' \texttt{ as } \texttt{T}_2} \qquad \text{(E-Inr)}$$

Variants

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

New syntactic forms

$\mathtt{t}$	::=	...	*terms*
		$\mathtt{<l=t>\ as\ T}$	*tagging*
		$\mathtt{case\ t\ of\ <l_i=x_i>} \Rightarrow \mathtt{t_i}\ ^{i\in 1..n}$	*case*
$\mathtt{T}$	::=	...	*types*
		$\mathtt{<l_i:T_i}\ ^{i\in 1..n}\mathtt{>}$	*type of variants*

New evaluation rules

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

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

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

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

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

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

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

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;
```

Recursion

Recursion in $\lambda_{\rightarrow}$

- In $\lambda_{\rightarrow}$, all programs terminate. (Cf. Chapter 12.)
- Hence, untyped terms like `omega` and `fix` are not typable.
- But we can extend the system with a (typed) fixed-point operator...

Example

```
ff = λie:Nat→Bool.
       λx:Nat.
         if iszero x then true
         else if iszero (pred x) then false
         else ie (pred (pred x));

iseven = fix ff;

iseven 7;
```

New syntactic forms

$\mathtt{t}$	::=	...	*terms*
		$\mathtt{fix\ t}$	*fixed point of* $\mathtt{t}$

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

$$\mathtt{fix\ (\lambda x{:}T_1.t_2)} \longrightarrow [\mathtt{x} \mapsto \mathtt{(fix\ (\lambda x{:}T_1.t_2))}]\mathtt{t_2} \qquad \text{(E-FixBeta)}$$

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{t_1'}}{\mathtt{fix\ t_1} \longrightarrow \mathtt{fix\ t_1'}} \qquad \text{(E-Fix)}$$

New typing rules

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

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \to \mathtt{T}_1}{\Gamma \vdash \mathtt{fix}\ \mathtt{t}_1 : \mathtt{T}_1} \quad \text{(T-Fix)}$$

A more convenient form

$$\texttt{letrec x:T}_1\texttt{=t}_1\texttt{ in t}_2 \stackrel{\text{def}}{=} \texttt{let x = fix (}\lambda\texttt{x:T}_1\texttt{.t}_1\texttt{) in t}_2$$

```
letrec iseven : Nat→Bool =
  λx:Nat.
    if iszero x then true
    else if iszero (pred x) then false
    else iseven (pred (pred x))
in
  iseven 7;
```