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

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

$$\frac{\Gamma \vdash t_1 : T_{11} \times T_{12}}{\Gamma \vdash t_1.1 : T_{11}} \quad \text{(T-Proj1)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \times T_{12}}{\Gamma \vdash t_1.2 : T_{12}} \quad \text{(T-Proj2)}$$

Records

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

Evaluation rules for records

$$\{l_i{=}v_i \ ^{i\in 1..n}\}.l_j \longrightarrow v_j \qquad \text{(E-ProjRcd)}$$

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

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

Typing rules for records

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

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

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*

$\texttt{T}_1\texttt{+T}_2$ is a disjoint union of $\texttt{T}_1$ and $\texttt{T}_2$ (the tags `inl` and `inr` ensure disjointness)

New evaluation rules

$\boxed{t \longrightarrow t'}$

$$\texttt{case (inl } v_0\texttt{)}\ \texttt{of inl } x_1 \Rightarrow t_1 \ \texttt{|}\ \texttt{inr } x_2 \Rightarrow t_2 \longrightarrow [x_1 \mapsto v_0]t_1 \qquad \text{(E-CaseInl)}$$

$$\texttt{case (inr } v_0\texttt{)}\ \texttt{of inl } x_1 \Rightarrow t_1 \ \texttt{|}\ \texttt{inr } x_2 \Rightarrow t_2 \longrightarrow [x_2 \mapsto v_0]t_2 \qquad \text{(E-CaseInr)}$$

$$\frac{t_0 \longrightarrow t_0'}{\texttt{case } t_0 \texttt{ of inl } x_1 \Rightarrow t_1 \ \texttt{|}\ \texttt{inr } x_2 \Rightarrow t_2 \longrightarrow \texttt{case } t_0' \texttt{ of inl } x_1 \Rightarrow t_1 \ \texttt{|}\ \texttt{inr } x_2 \Rightarrow t_2} \qquad \text{(E-Case)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{inl } t_1 \longrightarrow \texttt{inl } t_1'} \qquad \text{(E-Inl)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{inr } t_1 \longrightarrow \texttt{inr } t_1'} \qquad \text{(E-Inr)}$$

New typing rules

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

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

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

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

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

`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

$\Gamma \vdash t : T$

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

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

Evaluation rules ignore annotations:

$\boxed{t \longrightarrow t'}$

$$\texttt{case (inl } v_0 \texttt{ as } T_0\texttt{)}\ \texttt{of inl } x_1 \Rightarrow t_1 \texttt{ | inr } x_2 \Rightarrow t_2 \longrightarrow [x_1 \mapsto v_0]t_1 \qquad \text{(E-CaseInl)}$$

$$\texttt{case (inr } v_0 \texttt{ as } T_0\texttt{)}\ \texttt{of inl } x_1 \Rightarrow t_1 \texttt{ | inr } x_2 \Rightarrow t_2 \longrightarrow [x_2 \mapsto v_0]t_2 \qquad \text{(E-CaseInr)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{inl } t_1 \texttt{ as } T_2 \longrightarrow \texttt{inl } t_1' \texttt{ as } T_2} \qquad \text{(E-Inl)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{inr } t_1 \texttt{ as } T_2 \longrightarrow \texttt{inr } t_1' \texttt{ as } T_2} \qquad \text{(E-Inr)}$$

Variants

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

New syntactic forms

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

New evaluation rules

$\boxed{t \longrightarrow t'}$

$$\texttt{case (<}l_j\texttt{=}v_j\texttt{> as T) of <}l_i\texttt{=}x_i\texttt{>}\Rightarrow t_i \;^{i\in 1..n} \longrightarrow [x_j \mapsto v_j]t_j \qquad \text{(E-CaseVariant)}$$

$$\frac{t_0 \longrightarrow t_0'}{\texttt{case } t_0 \texttt{ of <}l_i\texttt{=}x_i\texttt{>}\Rightarrow t_i \;^{i\in 1..n} \longrightarrow \texttt{case } t_0' \texttt{ of <}l_i\texttt{=}x_i\texttt{>}\Rightarrow t_i \;^{i\in 1..n}} \qquad \text{(E-Case)}$$

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

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

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

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

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

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

$\mathtt{fix\ t}$ *fixed point of* $\mathtt{t}$

New evaluation rules

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

$$\mathtt{fix\ (\lambda x{:}T_1 . t_2)} \longrightarrow [\mathtt{x} \mapsto \mathtt{(fix\ (\lambda x{:}T_1 . t_2))}]\mathtt{t}_2 \qquad \text{(E-FixBeta)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{fix\ t}_1 \longrightarrow \mathtt{fix\ t}_1'} \qquad \text{(E-Fix)}$$

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

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

A more convenient form

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

```
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;
```