CIS 500

Software Foundations

Fall 2004

20/27 October

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*

New typing rules

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

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1}{\Gamma \vdash \mathtt{inl}\ \mathtt{t}_1 : \mathtt{T}_1\mathtt{+}\mathtt{T}_2} \qquad \text{(T-Inl)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_2}{\Gamma \vdash \mathtt{inr}\ \mathtt{t}_1 : \mathtt{T}_1\mathtt{+}\mathtt{T}_2} \qquad \text{(T-Inr)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{t}_0 : \mathtt{T}_1\mathtt{+}\mathtt{T}_2 \\ \Gamma, \mathtt{x}_1\mathtt{:}\mathtt{T}_1 \vdash \mathtt{t}_1 : \mathtt{T} \qquad \Gamma, \mathtt{x}_2\mathtt{:}\mathtt{T}_2 \vdash \mathtt{t}_2 : \mathtt{T}\end{array}}{\Gamma \vdash \mathtt{case}\ \mathtt{t}_0\ \mathtt{of}\ \mathtt{inl}\ \mathtt{x}_1 \Rightarrow \mathtt{t}_1\ \mathtt{|}\ \mathtt{inr}\ \mathtt{x}_2 \Rightarrow \mathtt{t}_2 : \mathtt{T}} \qquad \text{(T-Case)}$$

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

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

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

$$\frac{\mathtt{t_0} \longrightarrow \mathtt{t_0'}}{\begin{array}{l}\mathtt{case\ t_0\ of\ inl\ x_1 \Rightarrow t_1\ |\ inr\ x_2 \Rightarrow t_2} \\ \longrightarrow \mathtt{case\ t_0'\ of\ inl\ x_1 \Rightarrow t_1\ |\ inr\ x_2 \Rightarrow t_2}\end{array}} \quad \text{(E-Case)}$$

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

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

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

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

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{T}_1}{\Gamma \vdash \texttt{inl } \texttt{t}_1 \texttt{ as } \texttt{T}_1\texttt{+}\texttt{T}_2 : \texttt{T}_1\texttt{+}\texttt{T}_2} \qquad \text{(T-Inl)}$$

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{T}_2}{\Gamma \vdash \texttt{inr } \texttt{t}_1 \texttt{ as } \texttt{T}_1\texttt{+}\texttt{T}_2 : \texttt{T}_1\texttt{+}\texttt{T}_2} \qquad \text{(T-Inr)}$$

Evaluation rules ignore annotations:

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

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

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

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

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

Variants

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

New syntactic forms

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

New evaluation rules

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

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

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

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

New typing rules

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

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

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

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;
```

Terminology: "Union Types"

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

(We could also consider a non-disjoint union $T_1 \vee T_2$, but its properties are substantially more complex, because it induces an interesting subtype relation. We'll come back to subtyping later.)

Recursion

Recursion in $\lambda_{\to}$

- In $\lambda_{\to}$, 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

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

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

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

$$\texttt{fix } (\lambda \texttt{x:T}_1\texttt{.t}_2) \longrightarrow [\texttt{x} \mapsto (\texttt{fix } (\lambda \texttt{x:T}_1\texttt{.t}_2))]\texttt{t}_2 \qquad \text{(E-FixBeta)}$$

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

New typing rules

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

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

A more convenient form

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

```
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;
```

Lists

Lists — syntax

`t`	::=	...	*terms*
		`nil[T]`	*empty list*
		`cons[T] t t`	*list constructor*
		`isnil[T] t`	*test for empty list*
		`head[T] t`	*head of a list*
		`tail[T] t`	*tail of a list*
`v`	::=	...	*values*
		`nil[T]`	*empty list*
		`cons[T] v v`	*list constructor*
`T`	::=	...	*types*
		`List T`	*type of lists*

Lists — evaluation

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{cons[T]}\ \texttt{t}_1\ \texttt{t}_2 \longrightarrow \texttt{cons[T]}\ \texttt{t}_1'\ \texttt{t}_2} \quad \text{(E-Cons1)}$$

$$\frac{\texttt{t}_2 \longrightarrow \texttt{t}_2'}{\texttt{cons[T]}\ \texttt{v}_1\ \texttt{t}_2 \longrightarrow \texttt{cons[T]}\ \texttt{v}_1\ \texttt{t}_2'} \quad \text{(E-Cons2)}$$

$$\texttt{isnil[S]}\ \texttt{(nil[T])} \longrightarrow \texttt{true} \quad \text{(E-IsnilNil)}$$

$$\texttt{isnil[S]}\ \texttt{(cons[T]}\ \texttt{v}_1\ \texttt{v}_2\texttt{)} \longrightarrow \texttt{false} \quad \text{(E-IsnilCons)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{isnil[T]}\ \texttt{t}_1 \longrightarrow \texttt{isnil[T]}\ \texttt{t}_1'} \quad \text{(E-Isnil)}$$

$$\texttt{head[S] (cons[T] v}_1\ \texttt{v}_2\texttt{)} \longrightarrow \texttt{v}_1 \qquad \text{(E-HeadCons)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{head[T] t}_1 \longrightarrow \texttt{head[T] t}_1'} \qquad \text{(E-Head)}$$

$$\texttt{tail[S] (cons[T] v}_1\ \texttt{v}_2\texttt{)} \longrightarrow \texttt{v}_2 \qquad \text{(E-TailCons)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{tail[T] t}_1 \longrightarrow \texttt{tail[T] t}_1'} \qquad \text{(E-Tail)}$$

Note that evaluation rules do not look at type annotations!

Lists — typing

$$\Gamma \vdash \texttt{nil}[\texttt{T}_1] : \texttt{List}\ \texttt{T}_1 \qquad \text{(T-Nil)}$$

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{T}_1 \qquad \Gamma \vdash \texttt{t}_2 : \texttt{List}\ \texttt{T}_1}{\Gamma \vdash \texttt{cons}[\texttt{T}_1]\ \texttt{t}_1\ \texttt{t}_2 : \texttt{List}\ \texttt{T}_1} \qquad \text{(T-Cons)}$$

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{List}\ \texttt{T}_{11}}{\Gamma \vdash \texttt{isnil}[\texttt{T}_{11}]\ \texttt{t}_1 : \texttt{Bool}} \qquad \text{(T-IsNil)}$$

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{List}\ \texttt{T}_{11}}{\Gamma \vdash \texttt{head}[\texttt{T}_{11}]\ \texttt{t}_1 : \texttt{T}_{11}} \qquad \text{(T-Head)}$$

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{List}\ \texttt{T}_{11}}{\Gamma \vdash \texttt{tail}[\texttt{T}_{11}]\ \texttt{t}_1 : \texttt{List}\ \texttt{T}_{11}} \qquad \text{(T-Tail)}$$