CIS 500

Software Foundations

Fall 2002

21 October

Administrivia

- Missing HW5s have been found
- Notes on exam
  - Graded exams and answer key available from Christine (in 556)
  - Rough grade breakdown:
    - 65-80 points: A (32%)
    - 50-64 points: B (35%)
    - 35-49 points: C (19%)
    - $\leq$34 points: D/F (14%)

    60+ points is on-target for WPE-I
  - This exam mostly focused on the more "mechanical" aspects of the material we have seen. Future exams will be more focused on concepts (i.e., there will be more questions like 8, 9, 10, and 12).
- Grading questions? See your TA.

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

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

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

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

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

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

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

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

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

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

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

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

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

Evaluation rules ignore annotations: $\boxed{t \longrightarrow t'}$

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

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

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

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

Variants

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

New syntactic forms

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

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

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

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

Examples

```
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 more complex because it induces an interesting subtype relation...

General Recursion

- 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

t ::= ...		terms
	fix t	fixed point of t

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

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

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

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

$$\frac{\Gamma \vdash t_1 : T_1 \to T_1}{\Gamma \vdash \texttt{fix } t_1 : T_1} \qquad \text{(T-Fix)}$$

A more convenient form

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

```
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;
```

Lists

[See book.]

References

Mutability

- In most programming languages, variables are mutable. I.e., a variable provides both
  - a name that refers to a previously calculated value
  - the possibility of overwriting this value with another (which will be referred to by the same name)
- In some languages (e.g., OCaml), these two features are kept separate
  - variables are only for naming — the binding between a variable and its value is immutable
  - introduce a new class of mutable cells or references
  - at any given moment, a reference holds a value (and can be dereferenced to obtain this value)
  - a new value may be assigned to a reference

We choose OCaml's style, which is easier to work with formally.

So a variable of type `T` in most languages (except OCaml) will correspond to a `Ref T` (actually, a `Ref(Option T)`) here.

Examples

[...]

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

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{Ref } \texttt{T}_1}{\Gamma \vdash \texttt{!}\texttt{t}_1 : \texttt{T}_1} \qquad \text{(T-Deref)}$$

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{Ref } \texttt{T}_1 \qquad \Gamma \vdash \texttt{t}_2 : \texttt{T}_1}{\Gamma \vdash \texttt{t}_1\texttt{:=}\texttt{t}_2 : \texttt{Unit}} \qquad \text{(T-Assign)}$$