**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%)
    - ≤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**

```plaintext
PhysicalAddr = {first,last: String, addr: String}
VirtualAddr = {name: String, email: String}
Addr = PhysicalAddr + VirtualAddr

inl : "PhysicalAddr → PhysicalAddr+VirtualAddr "
inr : "VirtualAddr → PhysicalAddr+VirtualAddr "

getName = A:Addr.
  case a of
  inl x ⇒ x.first last
  | inr y ⇒ y.name;
```

---

**New syntactic forms**

```plaintext
T ::= ... terms
    | inl t
    | inr t
    | case t of inl x ⇒ t | inr x ⇒ t

v ::= ... values
    | inl v
    | inr v

v ::= ... tagged values
    | tagged value (left)
    | tagged value (right)

T ::= ... types
    | sum type
```
New typing rules

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

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

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

New evaluation rules

\[ t \rightarrow t' \]

\[ \text{case } (\text{inl} \ v_0) \]
\[ \text{of } \text{inl} \ x_1 \Rightarrow t_1 \mid \text{inr} \ x_2 \Rightarrow t_2 \]
\[ \rightarrow [x_1 \mapsto v_0 | t_1] \]  (E-CASEINL)

\[ \text{case } (\text{inr} \ v_0) \]
\[ \text{of } \text{inl} \ x_1 \Rightarrow t_1 \mid \text{inr} \ x_2 \Rightarrow t_2 \]
\[ \rightarrow [x_2 \mapsto v_0 | t_2] \]  (E-CASEINR)

\[ t_0 \rightarrow t'_0 \]
\[ \text{case } t_0 \text{ of } \text{inl} \ x_1 \Rightarrow t_1 \mid \text{inr} \ x_2 \Rightarrow t_2 \]
\[ \rightarrow \text{case } t'_0 \text{ of } \text{inl} \ x_1 \Rightarrow t_1 \mid \text{inr} \ x_2 \Rightarrow t_2 \]  (E-CASE)

Sums and Uniqueness of Types

Problem:

If \( t \) has type \( T \), then \( \text{inl} \ t \) 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 \( \text{inl} \) and \( \text{inr} \) with the intended sum type.

For simplicity, let’s choose the third.
New syntactic forms

\[ t ::= \ldots \]
\[ \text{terms} \]
\[ \text{inl } t \text{ as } T \]
\[ \text{inr } t \text{ as } T \]

\[ v ::= \ldots \]
\[ \text{values} \]
\[ \text{inl } v \text{ as } T \]
\[ \text{inr } v \text{ as } T \]

New typing rules

\[ \Gamma \vdash t : T \]
\[ \Gamma \vdash t_1 : T_1 \]
\[ \Gamma \vdash \text{inl } t_1 \text{ as } T_1 + T_2 : T_1 + T_2 \]

\[ \text{tagging (left)} \]
\[ \text{tagging (right)} \]

\[ \Gamma \vdash \text{inl } t_1 \text{ as } T_1 + T_2 : T_1 + T_2 \]

\[ \Gamma \vdash t_1 : T_2 \]

\[ \Gamma \vdash \text{inr } t_1 \text{ as } T_1 + T_2 : T_1 + T_2 \]

\[ \text{tagged value (left)} \]
\[ \text{tagged value (right)} \]

Evaluation rules ignore annotations:

\[ t \rightarrow t' \]

\[ \text{case (inl } v_0 \text{ as } T_0) \]
\[ \text{of inl } x_1 \mapsto t_1 \mid \text{inr } x_2 \mapsto t_2 \]
\[ \rightarrow [x_1 \mapsto v_0 \& t_1] \]  
\[ \text{(E-CASEINL)} \]

\[ \text{case (inr } v_0 \text{ as } T_0) \]
\[ \text{of inl } x_1 \mapsto t_1 \mid \text{inr } x_2 \mapsto t_2 \]
\[ \rightarrow [x_2 \mapsto v_0 \& t_2] \]
\[ t_1 \rightarrow t'_1 \]  
\[ \text{(E-INL)} \]

\[ \text{inl } t_1 \text{ as } T_2 \rightarrow \text{inl } t_1' \text{ as } T_2 \]
\[ t_1 \rightarrow t_1' \]  
\[ \text{(E-INR)} \]

\[ \text{inr } t_1 \text{ as } T_2 \rightarrow \text{inr } t_1' \text{ as } T_2 \]

Variants

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

\[ t ::= \ldots \]

- \( \langle l \triangleright \rangle \rightarrow T \)
- \( \text{case } t \text{ of } \langle l \triangleright x \rangle \rightarrow t \mid l \\in L \)'s

\[ T ::= \ldots \]

- \( \langle l_1 : T_1 \mid L \rangle \)
- \( \text{type of variants} \)

New evaluation rules

For \( \Gamma \vdash t : T \):

- \( \text{New evaluation rules} \)
- \( t \leftrightarrow t' \)
- \( \text{case } \langle l_1 = v_1 \rangle \rightarrow \text{as } \langle l_1 : T_1 \mid L \rangle \rightarrow t_1 \mid l \in L \)
  \[ \rightarrow [x_1 \mapsto v_1] t_1 \]

(E-CASEVARIANT)

- \( \text{case } t \rightarrow t' \)

(E-CASE)

- \( \text{case } t \rightarrow t' \)
  \[ \rightarrow \text{case } t' \rightarrow \langle l_1 = x_1 \rangle \rightarrow t_1 \mid l \in L \]

(T-VARIANT)

\[ t_1 \rightarrow t' \]

Examples

\[ \text{Addr} = \langle \text{physical:PhysicalAddr, virtual:VirtualAddr} \rangle; \]

\[ a = \langle \text{physical=pa} \rangle \rightarrow \text{as } \text{Addr}; \]

\[ \text{getName} = \lambda a: \text{Addr}. \]

- \( \text{case } a \rightarrow \langle \text{physical=x} \rightarrow x.\text{firstlast} \mid \langle \text{virtual=y} \rightarrow y.\text{name} \rangle \rightarrow \text{null}; \)

\( \text{getName a; \null} \)
Options

Just like in OCaml...

```ocaml
OptionsNat = <none:Unit, some:Nat>;
Table = Nat -> OptionsNat;
emptyTable = \n: Nat. <none=unit> as OptionsNat;
extendTable = 
  At:Table. \n: Nat. \n: Nat.
  \n: Nat.
  if equal n m then <some=n> as OptionsNat
  else t n;
x = case t(5) of
  <none=n> => 999
  | <some=n> => n;
```

Enumerations

Weekday = <monday:Unit, tuesday:Unit, wednesday:Unit, thursday:Unit, friday:Unit>;
nextBusinessDay = \w: 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 $T_1 \lor T_2$, but its properties are more complex because it induces an interesting subtype relation...

General Recursion

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

\[
\begin{align*}
ff & = \lambda e : \text{Nat} \rightarrow \text{Bool}. \\
& \quad \lambda x : \text{Nat}. \\
& \quad \text{if iszero } x \text{ then true} \\
& \quad \text{else if iszero (pred } x) \text{ then false} \\
& \quad \text{else is (pred (pred } x)). \\
\text{isEven} & = \text{fix } ff; \\
\text{isEven } 7;
\end{align*}
\]

New syntactic forms

\[
t \ ::= \ldots \\
\text{fix } t \quad \text{fixed point of } t
\]

New evaluation rules

\[
\begin{align*}
\text{fix } (\lambda x : T_1 . t_2) & \rightarrow [x \mapsto \text{fix } (\lambda x : T_1 . t_2)] t_2 \\
(t_1 \rightarrow t'_1) \quad (E\text{-FIXBETA}) \\
\text{fix } t_1 & \rightarrow \text{fix } t'_1 \\
(t_1 \rightarrow t'_1) \quad (E\text{-FIX})
\end{align*}
\]

New typing rules

\[
\Gamma \vdash t_1 : T_1 \\
\Gamma \vdash \text{fix } t_1 : T_1 \\
\quad (T\text{-FIX})
\]

A more convenient form

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

letrec isEven : Nat \rightarrow \text{Bool} = \\
\quad \lambda x : \text{Nat}. \\
\quad \text{if iszero } x \text{ then true} \\
\quad \text{else if iszero (pred } x) \text{ then false} \\
\quad \text{else isEven (pred (pred } x)) \\
\quad \text{in} \\
\quad \text{isEven } 7;
Lists

[See book.]

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

\[ ... \]