Lecture 26    Review

PLAN

- OCaml syntax
  symbols, expressions, terms, types, values

- OCaml computation rules
  $t \rightarrow t'$, rules for reduction, substitution model
  can we have $\text{eval} t \rightarrow n$?
  eager and lazy evaluation

- State and mutable variables
  while bare do body done
  state as a record
  tail recursion

- OCaml typing rules
  type checking is decidable

- OCaml SL types
  type checking is undecidable

- Research question example
  which recursive types make sense
  for OCaml SL?

- Recursion and Induction

- Real numbers and Modules

- If time, another Research Question.
OCaml Syntax

symbols $\Sigma = \{ a, b, c, \ldots, 0, 1, 2, \ldots, i, \to, \_ \}$

expressions recognized by the parser

terms expressions that have types
- true, let, $a \times b$, $a \times c$
  Is type a value?

types expressions recognized by types, also defined type $\alpha::\beta$
- polymorphic type expressions
  $\alpha \times \beta$, $\alpha \to \beta$, $\text{Id}_1 \circ \alpha | \text{Id}_2 \circ \beta$

in OCaml SL these polymorphic expressions give rise to polymorphic logic

$\alpha \times \beta \to \alpha \quad \alpha \to \alpha \mid \beta$

$\alpha :: \Rightarrow \beta (\alpha)$
\( \mathbb{E} \uparrow \mathbb{E} \) is our notation for evaluation (reduction rules).

\begin{align*}
\text{e.g.} & \quad \text{bexp}_1 \uparrow \text{true} \uparrow \text{bexp}_1 \uparrow \text{bexp}_2 \uparrow \text{true} \\
\text{Note how if bexp then exp, else exp\_2 evaluates, it is lazy! What does this mean?}
\end{align*}

Function evaluation \( f \alpha \)

What is the standard order?

What is lazy evaluation?

\( \text{fix} \) vs \( \text{efix} \)

\begin{align*}
\text{fix} f & \uparrow f (\text{fix} f) \\
\text{efix} f & \uparrow f (\text{efix} f) \uparrow 
\end{align*}

It would be nice to have

\( \text{eval} \uparrow \alpha \) that evaluates \( \mathbb{E} \) for \( n \) steps

Can this be eager in \( \mathbb{E} \)?

Recursive functions can be defined by \( \text{fix} \) and \( \text{efix} \).

What is tail recursion?

Consider the while bexp do exp od

\( \text{(in OCaml while bexp do exp done )} \)
Ocaml Computation Rules continued

```ocaml
while (expr do expr done) is used for mutable variables, but we can also think of state as a record

\[ \exists x_1 : t_1 \times x_2 : t_2 \cdots \times x_n : t_n \]

We can "simulate" mutable variables by operations on state that are functional.

This leads to state monads that we did not cover. It's how Haskell deals with state.

Exercise: WRITE THE SQUARE ROOT EXAMPLE FROM LECTURE 19 USING THE WHILE LOOP AS A RECURSIVE FUNCTION ON A FUNCTIONAL RECORD.

Specify the square root problem with a dependent type.
Ocaml Typing Rules

We provided some explicit typing rules in Lecture 2

\[ \text{exp}_1 : \text{int}, \text{exp}_2 : \text{int} \vdash \text{exp}_1 + \text{exp}_2 : \text{int} \]

\[ f : x \rightarrow \beta, a : x \vdash f a : \beta \]

Exercise Give typing rules for Fix and efix.

Note that recursive types are limited

\[ \text{type } \alpha = \alpha \rightarrow \alpha \quad \text{cyclic!} \]
\[ \text{type } \alpha = \alpha \times \text{int} \quad \text{cyclic} \]

But Ocaml allows

\[ \text{type } \alpha = \text{Left of int} \mid \text{Right of } \alpha \rightarrow \text{int} \]

This type "does not make sense" mathematically.

Research Question What Ocaml SL recursive types make sense?

We investigate this using the subtype relation, \( \alpha \subseteq \beta \)

that needs to be defined for a full Ocaml SL.
Recursion and Induction

We have studied in detail the idea that recursion and induction are closely related. Recursion is terminating induction.

If we use dependent types we can say that induction is specified by a dependent type inhabited (realized) by a recursive function—see Lecture 18, page 6.

```
let rec list_ind (l: α list) (base : P [l])
(step : m: α → n: α list → μ: P n → P m: n)
: P l =
match l with [l] → base
| h::t → step h t list_ind t base step
```

We noted that this is fold-right.

Note, we can type P as α list → type.
Real Numbers

If we define $e = \sum_{i=1}^{\infty} \frac{1}{i!}$, this provides a simple basis for writing $e$ as a real number. You should be able to define $e + e$ and $e \times e$. 
Research Question: Is the Fan Theorem valid in OCA and SL?

**Fan Specification**

\[ \bar{x}(n) = [x_0, ..., x_n] \]

\[ x: (\text{nat} \to \text{bool}) \to (n: \text{nat where } R \bar{x} n) \to \]

\[ z: (\text{nat} \times x: (\text{nat} 	o \text{bool})) \to (x: \text{nat where } x \leq z \land R \bar{x} x) \]

\[ x: \text{nat where } x \leq z \land R \bar{x} x \]

The \( z \) is a uniform bound on the height of the tree.