Today, we will learn about

- Type constructors
  - Lists, Options

- Algebraic structures
  - Monoids
  - Functors
  - Monads

- Algebraic structures in Haskell
Type Constructors

- A type constructor creates new types from existing types
Type Constructors

- A *type constructor* creates new types from existing types
  - E.g., product types, sum types, reference types, function types, ...
Assume CBV $\lambda$-calc with booleans, fixpoint operator $\mu x : \tau. \ e$

Expressions
\[
e ::= \cdots \ | \ [] \\
\quad \ | \ e_1 :: e_2 \ | \ \text{isempty}? \ e \ | \ \text{head} \ e \\
\quad \ | \ \text{tail} \ e
\]

Values
\[
v ::= \cdots \ | \ [] \ | \ v_1 :: v_2
\]

Types
\[
\tau ::= \cdots \ | \ \tau \ \text{list}
\]

Eval contexts
\[
E ::= \cdots \ | \ E :: e \ | \ v :: E \\
\quad \ | \ \text{isempty}? \ E \ | \ \text{head} \ E \ | \ \text{tail} \ E
\]
List inference rules

\[\text{isempty? } [ ] \rightarrow \text{true} \quad \text{isempty? } \nu_1 :: \nu_2 \rightarrow \text{false}\]

\[\text{head } \nu_1 :: \nu_2 \rightarrow \nu_1 \quad \text{tail } \nu_1 :: \nu_2 \rightarrow \nu_2\]

\[\Gamma \vdash [ ] : \tau \text{ list} \quad \Gamma \vdash e_1 : \tau \text{ list} \quad \Gamma \vdash e_2 : \tau \text{ list} \]

\[\Gamma \vdash e_1 :: e_2 : \tau \text{ list} \quad \Gamma \vdash \text{isempty? } e : \text{bool} \quad \Gamma \vdash \text{head } e : \tau \text{ list} \quad \Gamma \vdash \text{tail } e : \tau \text{ list}\]

append \(\triangleq \mu f : \tau \text{ list} \rightarrow \tau \text{ list} \rightarrow \tau \text{ list} \cdot \lambda a : \tau \text{ list} \cdot \lambda b : \tau \text{ list} \cdot \text{if } \text{isempty? } a \text{ then } b \text{ else } (\text{head } a) :: (f (\text{tail } a) b)\)
Options

Expressions
\[ e ::= \cdots \mid \text{none} \mid \text{some } e \]
\[ \quad \mid \text{case } e_1 \text{ of } e_2 \mid e_3 \]

Values
\[ v ::= \cdots \mid \text{none} \mid \text{some } v \]

Types
\[ \tau ::= \cdots \mid \tau \text{ option} \]

Eval contexts
\[ E ::= \cdots \mid \text{some } E \mid \text{case } E \text{ of } e_2 \mid e_3 \]
Option as syntactic sugar

- Option as syntactic sugar for the sum type `unit + τ`
- `none` as syntactic sugar for `inl unit + τ`
- `some e` as syntactic sugar for `inr unit + τ e`
Option as syntactic sugar

Ethernet MAC

the type \( \tau \text{ option} \) as syntactic sugar for the sum type \( \text{unit} + \tau \)
Option as syntactic sugar

- the type $\tau$ `option` as syntactic sugar for the sum type `unit + \tau`
- none as syntactic sugar for `inl_{unit+\tau}()`
Option as syntactic sugar

- the type $\tau$ **option** as syntactic sugar for the sum type $\text{unit} + \tau$
- none as syntactic sugar for $\text{inl}_{\text{unit} + \tau} ()$
- some $e$ as syntactic sugar for $\text{inr}_{\text{unit} + \tau} e$
Monoids
A *monoid* is a set $T$ with a distinguished element called the *unit* (which we will denote $u$) and a single operation $\text{multiply} : T \to T \to T$ that satisfies the following laws.

\[
\forall x \in T. \text{multiply} \ x \ u = x \quad \text{Left id.}
\]
\[
\forall x \in T. \text{multiply} \ u \ x = x \quad \text{Right id.}
\]
\[
\forall x, y, z \in T. \text{multiply} \ x \ (\text{multiply} \ y \ z) = \text{multiply} \ (\text{multiply} \ x \ y) \ z \quad \text{Assoc.}
\]
Monoid examples

- Integers with multiplication.
- Integers with addition.
- Strings with concatenation.
- Lists with append.
Functors
Functors

A functor associates with each set $A$ a set $T_A$; has a single operation $\text{map} : (A \to B) \to T_A \to T_B$ that takes a function from $A$ to $B$ and an element of $T_A$ and returns an element of $T_B$

\[
\forall f \in A \to B, \ g \in B \to C. \\
(map \ f); (map \ g) = map \ (f; g) \\
\text{Distributivity}
\]
\[
\text{map} \ (\lambda a : A. \ a) = (\lambda a : T_A. \ a) \\
\text{Identity}
\]
Functor examples

- Options.
- Lists.
A monad associates each set $A$ with a set $M_A$. Two operations:

- $\text{return} : A \rightarrow M_A$
- $\text{bind} : M_A \rightarrow (A \rightarrow M_B) \rightarrow M_B$
Monad laws

\( \forall x \in A, f \in A \rightarrow M_B. \)
\[
bind \ (return \ x) \ f = f \ x \quad \text{Left id.}
\]
\( \forall am \in M_A. \ bind \ am \ return = am \quad \text{Right id.}
\]
\( \forall am \in M_A, f \in A \rightarrow M_B, f \in B \rightarrow M_C. \)
\[
bind \ (bind \ am \ f) \ g =
bind \ am \ (\lambda a : A. \ bind \ (f \ a) \ g) \quad \text{Assoc.}
\]
Option monad

return: \( \tau \rightarrow \tau \text{ option} \)

bind: \( \tau_1 \text{ option} \rightarrow (\tau_1 \rightarrow \tau_2 \text{ option}) \rightarrow \tau_2 \text{ option} \)
Algebraic structures in Haskell

- https://www.haskell.org/
- Pure functional language
- Call-by-need evaluation (aka lazy evaluation)
- Type classes: mechanism for ad hoc polymorphism
  - Declares common functions that all types within class have
  - We will use them to express algebraic structures in Haskell
Why Monads?

- Monads are very useful in Haskell
- Haskell is pure: no side effects
- But side effects useful!
- **Monadic types cleanly and clearly express side effects computation may have**
- Monads force computation into sequence
- Monads as type classes capture underlying structure of computation
  - Reusable readable code that works for any monad
Further Reading

- Monadic Parsing in Haskell (Functional Pearl)
  https://www.cs.nott.ac.uk/~pszgmh/pearl.pdf

- Free Monads