Algebraic Structures

CS 152 (Spring 2021)

Harvard University

Tuesday, March 30, 2021

Today, we will learn about

- Type constructors
  - Lists, Options
- Alegebraic structures
  - Monoids
  - Functors
  - Monads
- Alegebraic 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, ...

Lists

- Assume CBV λ-calc with booleans, fixpoint operator $\mu x{:}\tau.\ e$

$$
\begin{array}{lrl}
\text{Expressions} & e ::= & \cdots \mid [\,] \\
& & \mid e_1 :: e_2 \mid \text{isempty? } e \mid \text{head } e \\
& & \mid \text{tail } e \\
\text{Values} & v ::= & \cdots \mid [\,] \mid v_1 :: v_2 \\
\text{Types} & \tau ::= & \cdots \mid \tau\ \textbf{list} \\
\text{Eval contexts} & E ::= & \cdots \mid E :: e \mid v :: E \\
& & \mid \text{isempty? } E \mid \text{head } E \mid \text{tail } E
\end{array}
$$

List inference rules

$$\frac{}{\text{isempty? } [\,] \longrightarrow \mathbf{true}} \qquad \frac{}{\text{isempty? } v_1 :: v_2 \longrightarrow \mathbf{false}}$$

$$\frac{}{\text{head } v_1 :: v_2 \longrightarrow v_1} \qquad \frac{}{\text{tail } v_1 :: v_2 \longrightarrow v_2}$$

$$\frac{}{\Gamma \vdash [\,] : \tau \textbf{ list}} \qquad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \textbf{ list}}{\Gamma \vdash e_1 :: e_2 : \tau \textbf{ list}}$$

$$\frac{\Gamma \vdash e : \tau \textbf{ list}}{\Gamma \vdash \text{isempty? } e : \mathbf{bool}} \qquad \frac{\Gamma \vdash e : \tau \textbf{ list}}{\Gamma \vdash \text{head } e : \tau} \qquad \frac{\Gamma \vdash e : \tau \textbf{ list}}{\Gamma \vdash \text{tail } e : \tau \textbf{ list}}$$

$$\begin{aligned}\text{append} \triangleq\ & \mu f : \tau \textbf{ list} \rightarrow \tau \textbf{ list} \rightarrow \tau \textbf{ list}.\ \lambda a : \tau \textbf{ list}.\ \lambda b : \tau \textbf{ list}. \\ & \textbf{if } \text{isempty? } a \textbf{ then } b \textbf{ else } (\text{head } a) :: (f\ (\text{tail } a)\ b)\end{aligned}$$

Options

$$
\begin{array}{lrl}
\text{Expressions} & e ::= & \cdots \mid \mathsf{none} \mid \mathsf{some}\ e \\
 & & \mid \mathsf{case}\ e_1\ \mathsf{of}\ e_2 \mid e_3 \\
\text{Values} & v ::= & \cdots \mid \mathsf{none} \mid \mathsf{some}\ v \\
\text{Types} & \tau ::= & \cdots \mid \tau\ \textbf{option} \\
\text{Eval contexts} & E ::= & \cdots \mid \mathsf{some}\ E \mid \mathsf{case}\ E\ \mathsf{of}\ e_2 \mid e_3
\end{array}
$$

Option as syntactic sugar

Option as syntactic sugar

- the type τ **option** as syntactic sugar for the sum type **unit** $+ \tau$

Option as syntactic sugar

- the type τ **option** as syntactic sugar for the sum type **unit** $+ \tau$
- none as syntactic sugar for $\mathsf{inl}_{\mathbf{unit}+\tau}\,()$

Option as syntactic sugar

- the type τ **option** as syntactic sugar for the sum type **unit** $+ \tau$
- none as syntactic sugar for $\mathsf{inl}_{\mathbf{unit}+\tau}\,()$
- some *e* as syntactic sugar for $\mathsf{inr}_{\mathbf{unit}+\tau}\,e$

Monoids

Monoids

A *monoid* is a set T with a distinguished element called the *unit* (which we will denote u) and a single operation $multiply : T \to T \to T$ that satisfies the following laws.

$$\begin{aligned}
\forall x \in T.\ multiply\ x\ u = x & \quad \text{Left id.} \\
\forall x \in T.\ multiply\ u\ x = x & \quad \text{Right id.} \\
\forall x, y, z \in T.\ multiply\ x\ (multiply\ y\ z) = & \\
multiply\ (multiply\ x\ y)\ z & \quad \text{Assoc.}
\end{aligned}$$

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 $map: (A \to B) \to T_A \to T_B$ that takes a function from A to B and an element of T_A and returns an element of T_B

$$\forall f \in A \to B, g \in B \to C.$$

$$\begin{aligned}(map\ f);(map\ g) &= map\ (f;g) && \text{Distributivity}\\ map\ (\lambda a{:}A.\,a) &= (\lambda a{:}T_A.\,a) && \text{Identity}\end{aligned}$$

Functor examples

- Options.
- Lists.

Monads

Monads

A monad associate each set A with a set M_A. Two operations:

- $return : A \to M_A$
- $bind : M_A \to (A \to M_B) \to M_B$

Monad laws

$$\forall x \in A, f \in A \to M_B.$$

$$bind\ (return\ x)\ f = f\ x \qquad \text{Left id.}$$

$$\forall am \in M_A.\ bind\ am\ return = am \qquad \text{Right id.}$$

$$\forall am \in M_A, f \in A \to M_B, f \in B \to M_C.$$

$$bind\ (bind\ am\ f)\ g =$$

$$bind\ am\ (\lambda a{:}A.\ bind\ (f\ a)\ g) \qquad \text{Assoc.}$$

Option monad

$$\text{return} : \tau \to \tau\ \textbf{option}$$
$$\text{bind} : \tau_1\ \textbf{option} \to (\tau_1 \to \tau_2\ \textbf{option}) \to \tau_2\ \textbf{option}$$

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 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