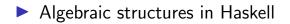
Algebraic Structures CS 152 (Spring 2022)

Harvard University

Thursday, March 31, 2022

Today, we will learn about

- Type constructors
 Lists, Options
- Algebraic structures
 - Monoids
 - Functors
 - Monads



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 μx:τ. e
- Expressions $e ::= \cdots | []$ $| e_1 :: e_2 | \text{ isempty}? e | \text{ head } e$ | tail eValues $v ::= \cdots | [] | v_1 :: v_2$ Types $\tau ::= \cdots | \tau \text{ list}$ Eval contexts $E ::= \cdots | E :: e | v :: E$ | isempty? E | head E | tail E

List inference rules

isempty? [] \longrightarrow true isempty? $v_1 :: v_2 \longrightarrow$ false head $v_1 :: v_2 \longrightarrow v_1$ tail $v_1 :: v_2 \longrightarrow v_2$ $\Gamma \vdash e_1 : \tau$ $\Gamma \vdash e_2 : \tau$ list $\Gamma \vdash []: \tau$ list $\Gamma \vdash e_1 :: e_2 : \tau$ list $\Gamma \vdash e:\tau$ list $\Gamma \vdash e:\tau$ list $\Gamma \vdash e: \tau$ list $\Gamma \vdash \text{isempty}$? e: bool $\Gamma \vdash \text{head} e: \tau$ $\Gamma \vdash \text{tail} e: \tau$ list

append $\triangleq \mu f : \tau$ list $\rightarrow \tau$ list $\rightarrow \tau$ list. $\lambda a : \tau$ list. $\lambda b : \tau$ list. if isompty? *a* then *b* else (head *a*) :: (*f* (tail *a*) *b*)

Options

Expressions $e ::= \cdots \mid$ none \mid some e \mid case e_1 of $e_2 \mid e_3$ Values $v ::= \cdots \mid$ none \mid some vTypes $\tau ::= \cdots \mid \tau$ optionEval contexts $E ::= \cdots \mid$ some $E \mid$ case E of $e_2 \mid e_3$

the type τ option as syntactic sugar for the sum type unit + τ

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

- the type τ option as syntactic sugar for the sum type unit + τ
- none as syntactic sugar for $\operatorname{inl}_{\operatorname{unit}+\tau}()$
- some e as syntactic sugar for inr_{unit+τ} 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* : $T \rightarrow T \rightarrow T$ that satisfies the following laws.

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

Monoid examples

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

Functors

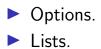
Functors

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

$$orall f \in A
ightarrow B, g \in B
ightarrow C.$$

 $(map \ f); (map \ g) = map \ (f;g)$ Distributivity
 $map \ (\lambda a: A. a) = (\lambda a: T_A. a)$ Identity

Functor examples



Monads

Monads

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

$$\blacktriangleright$$
 return : $A \rightarrow M_A$

▶ bind :
$$M_A \rightarrow (A \rightarrow M_B) \rightarrow M_B$$

Monad laws

$$\begin{aligned} \forall x \in A, f \in A \to M_B. \\ bind (return x) f = f x & \text{Left id.} \\ \forall am \in M_A. bind am return = am & \text{Right id.} \\ \forall am \in M_A, f \in A \to M_B, f \in B \to M_C. \\ bind (bind am f) g = \\ bind am (\lambda a: A. bind (f a) g) & \text{Assoc.} \end{aligned}$$

Option monad

return: $\tau \rightarrow \tau$ option bind: τ_1 option $\rightarrow (\tau_1 \rightarrow \tau_2 \text{ option}) \rightarrow \tau_2$ 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 https://okmij.org/ftp/Computation/ free-monad.html