Functional Programming 2014/2015
Assignment 3: Type classes

Ruud Koot

September 19, 2014

In this assignment we'll present a few type classes and ask you to implement some instances of them.

1 Containers

1.1 Functors

Recall the rose tree data structure from the previous assignment:

$$\textbf{data}\ \mathit{Rose}\ \alpha = \alpha \rhd [\mathit{Rose}\ \alpha]$$

Similarly to how we might want to apply a function uniformly to all elements in a list, we might also want to apply a function uniformly to all the elements in a rose tree, or any other container-like data structure for that matter. For this purpose Haskell has a *Functor* type class, exposing a single function *fmap* that generalizes the *map* function:

$$\begin{array}{l} \textbf{class}\ \mathit{Functor}\ f\ \textbf{where} \\ \quad \mathit{fmap} :: (\alpha \to \beta) \to f\ \alpha \to f\ \beta \end{array}$$

We see that *fmap* generalizes *map* by giving a *Functor* instance for lists:

$$\begin{array}{l} \textbf{instance}\ \mathit{Functor}\ [\,]\ \textbf{where} \\ \quad \mathit{fmap} = \mathit{map} \end{array}$$

Verify that *fmap* and *map* have the same type if we instantiate f to $[\,]$.

Exercise 1. *Write a Functor instance for the Rose data type.*

1.2 Monoids

A *monoid* is an algebraic structure over a type m with a single associative binary operation $(\diamond) :: m \to m \to m$ and an identity element $\mathit{mempty} :: m$.

$$\begin{array}{lll} \multicolumn{3}{l}{\textbf{class}\ \mathit{Monoid}\ m\ \textbf{where}} \\ \quad \mathit{mempty} & :: & m \\ \quad (\diamond) & :: & m \to m \to m \end{array}$$

Lists are monoids:

$$
\begin{aligned}
&\textbf{instance } \mathit{Monoid}\ [\,]\ \textbf{where} \\
&\quad \mathit{mempty} = [\,] \\
&\quad (\diamond) \quad\;\; = (+\!\!+)
\end{aligned}
$$

Verify that $(+\!\!+)$ is an associative operation (i.e., that $\forall xs\ ys\ zs.(xs +\!\!+ ys) +\!\!+ zs \equiv xs +\!\!+ (ys +\!\!+ zs)$) and that the empty list $[\,]$ is indeed an identity element with respect to list concatenation $(+\!\!+)$ (i.e., that $\forall ls.[\,] +\!\!+ ls \equiv ls$ and $\forall ls.ls +\!\!+ [\,] \equiv ls$).

Numbers also form a monoid, both under addition with 0 as the identity element, and under multiplication with 1 as the identity element (verify this). However, we are only allowed to give one instance per combination of type and type class. To overcome this limitation we create some **newtype** wrappers:

$$
\begin{aligned}
&\textbf{newtype } \mathit{Sum}\ \alpha = \mathit{Sum}\ \{\mathit{unSum} :: \alpha\} \\
&\textbf{newtype } \mathit{Product}\ \alpha = \mathit{Product}\ \{\mathit{unProduct} :: \alpha\}
\end{aligned}
$$

Now we can give two instances:

$$
\begin{aligned}
&\textbf{instance } \mathit{Num}\ \alpha \Rightarrow \mathit{Monoid}\ (\mathit{Sum}\ \alpha)\ \textbf{where} \\
&\quad \mathit{mempty} \qquad\qquad\quad = \mathit{Sum}\ 0 \\
&\quad \mathit{Sum}\ n1 \diamond \mathit{Sum}\ n2 = \mathit{Sum}\ (n1 + n2)
\end{aligned}
$$

Exercise 2. *Complete the second instance by writing a Monoid instance for numbers under multiplication.*

1.3 Foldable

If f is some container-like data structure storing elements of type m that form a monoid, then there is a way of folding all the elements in the data structure into a single element of the monoid m.

$$
\begin{aligned}
&\textbf{class } \mathit{Functor}\ f \Rightarrow \mathit{Foldable}\ f\ \textbf{where} \\
&\quad \mathit{fold} :: \mathit{Monoid}\ m \Rightarrow f\ m \rightarrow m
\end{aligned}
$$

In the case of lists:

$$
\begin{aligned}
&\textbf{instance } \mathit{Foldable}\ [\,]\ \textbf{where} \\
&\quad \mathit{fold} = \mathit{foldr}\ (\diamond)\ \mathit{mempty}
\end{aligned}
$$

Exercise 3. *Write a Foldable instance for Rose.*

It might be the case that we have a container-like data structure storing elements of type α that do not yet form a monoid, but where we do have a function of type $\alpha \rightarrow m$ that makes them into one. In such situation it would be convenient to have a function $\mathit{foldMap} :: \mathit{Monoid}\ m \Rightarrow (\alpha \rightarrow m) \rightarrow f\ \alpha \rightarrow m$ that first injects all the elements of the container into a monoid and then folds them into a single monoidal value.

Exercise 4. *Add a default implementation of foldMap to the Foldable type class, expressed in terms of fold and fmap.*

Exercise 5. *Write functions* $\mathit{fsum}, \mathit{fproduct} :: (\mathit{Foldable}\ f, \mathit{Num}\ \alpha) \Rightarrow f\ \alpha \rightarrow \alpha$ *that compute the sum, respectively product, of all numbers in a container-like data structure.*

2 Poker

If we want to implement a poker game, we need to represent playing cards, hands and have way of ranking hands:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Rank} = \mathit{R2} \mid \mathit{R3} \mid \mathit{R4} \mid \mathit{R5} \mid \mathit{R6} \mid \mathit{R7} \mid \mathit{R8} \mid \mathit{R9} \mid \mathit{R10} \mid J \mid Q \mid K \mid A \\
\quad \textbf{deriving}\ (\mathit{Bounded}, \mathit{Enum}, \mathit{Eq}, \mathit{Ord}) \\
\textbf{data}\ \mathit{Suit} = S \mid H \mid D \mid C \\
\quad \textbf{deriving}\ (\mathit{Bounded}, \mathit{Enum}, \mathit{Eq}, \mathit{Ord}, \mathit{Show}) \\
\textbf{data}\ \mathit{Card} = \mathit{Card}\ \{\mathit{rank} :: \mathit{Rank}, \mathit{suit} :: \mathit{Suit}\} \\
\textbf{type}\ \mathit{Deck} = [\mathit{Card}]
\end{array}
$$

2.1 Show

Exercise 6. *Define a Show instance for Rank that shows the ranks as* "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K", "A" *respectively.*

Exercise 7. *Give a Show instance for Card showing* $\mathit{Card}\ \{\mathit{rank} = \mathit{R2}, \mathit{suit} = H\}$ *as* "2H".

Exercise 8. *Define constants* $\mathit{fullDeck}, \mathit{piquetDeck} :: \mathit{Deck}$ *that give a full 52-card deck and a 32-card Piquet deck (with cards of ranks from 7 up to and including the Ace).*

2.2 Ord

A poker hand can be represented as:

$$\textbf{newtype}\ \mathit{Hand} = \mathit{Hand}\ \{\mathit{unHand} :: [\mathit{Card}]\}$$

and the various hand categories as:

$$
\begin{array}{l}
\textbf{data}\ \mathit{HandCategory} \\
\quad \begin{array}{llllll}
= & \mathit{HighCard} & [\mathit{Rank}] & & \\
\mid & \mathit{OnePair} & \mathit{Rank} & [\mathit{Rank}] & \\
\mid & \mathit{TwoPair} & \mathit{Rank} & \mathit{Rank} & \mathit{Rank} \\
\mid & \mathit{ThreeOfAKind} & \mathit{Rank} & \mathit{Rank} & \mathit{Rank} \\
\mid & \mathit{Straight} & \mathit{Rank} & & \\
\mid & \mathit{Flush} & [\mathit{Rank}] & & \\
\mid & \mathit{FullHouse} & \mathit{Rank} & \mathit{Rank} & \\
\mid & \mathit{FourOfAKind} & \mathit{Rank} & \mathit{Rank} & \\
\mid & \mathit{StraightFlush} & \mathit{Rank} & &
\end{array} \\
\quad \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord}, \mathit{Show})
\end{array}
$$

If you are not familiar with the ranking of poker hands then https://en.wikipedia.org/w/index.php?title=List_of_poker_hands&oldid=574969062 would be a good place to refer to.

The fields stored together with each category are used to distinguish between hands of the same category. For example, for a *HighCard* the field of type $[\mathit{Rank}]$ contains all five cards in the hand, sorted from high to low. In the case of a *TwoPair* the first *Rank* is that of the high pair, the second *Rank* that of the low pair, and the third *Rank* that of the kicker (the card that isn't part of any of the two pairs).

Convince yourself that the derived *Ord* instance correctly ranks poker hands represented as a *HandCategory*.

We are now going to write a function that converts hands of type *Hand* into hands of type *HandCategory*. First we'll need a few helper functions:

Exercise 9. *Write a function sameSuits* $::$ *Hand* $\rightarrow$ *Bool that returns True if all cards in a Hand are of the same suit.*

Exercise 10. *Write a function isStraight* $:: [Rank] \rightarrow$ *Maybe Rank that return a Just with the highest ranked card, if the Hand is a straight (or a straight flush). Note that the Ace can count both as the highest and as the lowest ranked card in a straight!*

Exercise 11. *Write a function ranks* $:: Hand \rightarrow [Rank]$ *that converts a Hand into a list of Ranks, ordered from high to low.*

Exercise 12. *Write a function order* $:: Hand \rightarrow [(Int, Rank)]$ *that converts a Hand into a list of Ranks paired with their multiplicity, order from high to low using a lexicographical ordering. For example, the hand* ["7H","7D","QS","7S","QH"] *should be ordered as* $[(3, R7), (2, Q)]$.

Exercise 13. *Finally, write a function handCategory* $:: Hand \rightarrow HandCategory$ *that converts a Hand into a HandCategory.*

Exercise 14. *Using handCategory, write an Ord instance for Hand.*

2.3 Combinatorics

Exercise 15. *Write a function combs* $:: Int \rightarrow [\alpha] \rightarrow [[\alpha]]$ *that returns all the combinations that can be formed by taking n elements from a list.*

Exercise 16. *Write a function allHands* $:: Deck \rightarrow [Hand]$ *that returns all combinations of 5-card hands than can be taken from a given deck of cards*

2.4 Data.Set

The *Ord* class on *Hand* induces an equivalence relation on poker hands. This can be useful when using functions or data structures that require an *Ord* instance on the data they are working with.

One example of such a data structure is *Data.Set*: a data structure that can only store unique elements. Uniqueness is determined by the equivalence relation induced by an *Ord* instance.

Exercise 17. *Write a function distinctHands* $:: Deck \rightarrow$ *Set Hand that constructs a maximal set of distinct hands from deck. (Hints: You will need the empty and insert functions from Data.Set. Use* $foldl'$ *instead of foldr to avoid a stack overflow when applying this function to large decks.)*

3 Questions

Question 1. *Do numbers*[1] *form a monoid under subtraction? If so, give the identity element. Do numbers form a monoid under division?*

Does Bool form a monoid under conjunction $(\wedge)$*? Does Bool form a monoid under the biconditional* $(\equiv)$*?*

Question 2. *Sheldon wants to implement Rock-paper-scissors-lizard-Spock in Haskell. He defined a data type:*

$$\mathbf{data}\ \mathit{Gesture} = \mathit{Rock} \mid \mathit{Paper} \mid \mathit{Scissors} \mid \mathit{Lizard} \mid \mathit{Spock}$$

and now wants to define an Ord instance for this data type that specifies which of two gestures beats the other.

Explain why this is not a good idea. The answer can be found by carefully reading the documentation of Data.Ord or imagining what happens if you sort a list of gestures using such an ordering.

[1] If you're now asking yourself: "But what kind of numbers do you mean exactly, Sir?" then please consider both various Haskell types having a *Num* instance (*Integer*, *Rational*, *Float*, ...) as well as various mathematical classes of numbers ($\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}\backslash\{0\}$, ...).