Exercise sheet 06— Solutions

1 Typeclass Instances for Binary Trees

a) SemigroupK for binary trees

Assuming we had a `Monoid` instance for our binary trees, what would the zero element look like? We don't have a representation for an empty tree. Our zero would be either a Branch or a Leaf, but neither is suitable as a zero element:

- If we used leafs as zero, we could no longer store values in our tree.
- If we used branches as zero, there would no longer be a tree.

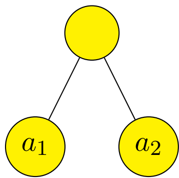
So we cannot define a monoid for our binary trees. But we can define a semigroup, which has an associative combine operation like monoid, but no neutral element for that operation. As we want to join trees independently from the contained type, we use `SemigroupK`.

A naive solution would be to simply combine two trees with a `Branch`:

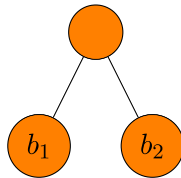
```
given SemigroupK[Tree] with
  override def combineK[A](x: Tree[A], y: Tree[A]): Tree[A] = Branch(x, y)
```

But this operation is not associative! We can see that in the following example:

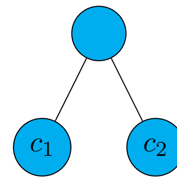
Tree A:



Tree B:



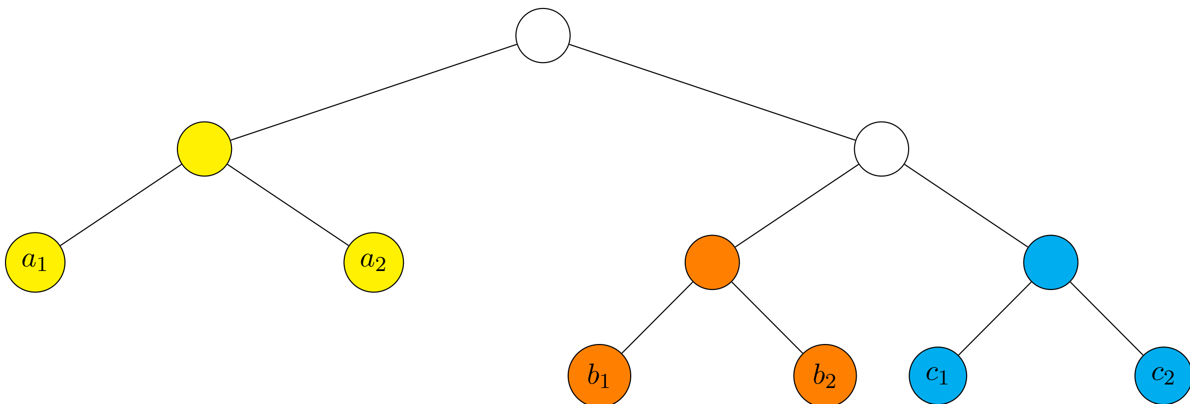
Tree C:



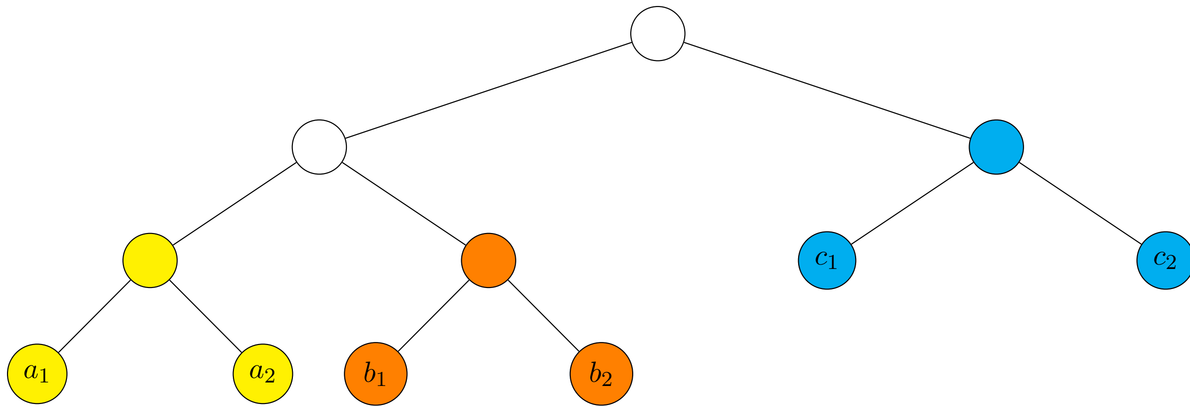
Here we have three trees. For an associative operator `<+>` (an alias for combineK in cats) the following must hold:

$$A \texttt{<+>} (B \texttt{<+>} C) = (A \texttt{<+>} B) \texttt{<+>} C$$

Our naive solution does not fulfill this law. The result of $A \texttt{<+>} (B \texttt{<+>} C)$ is:

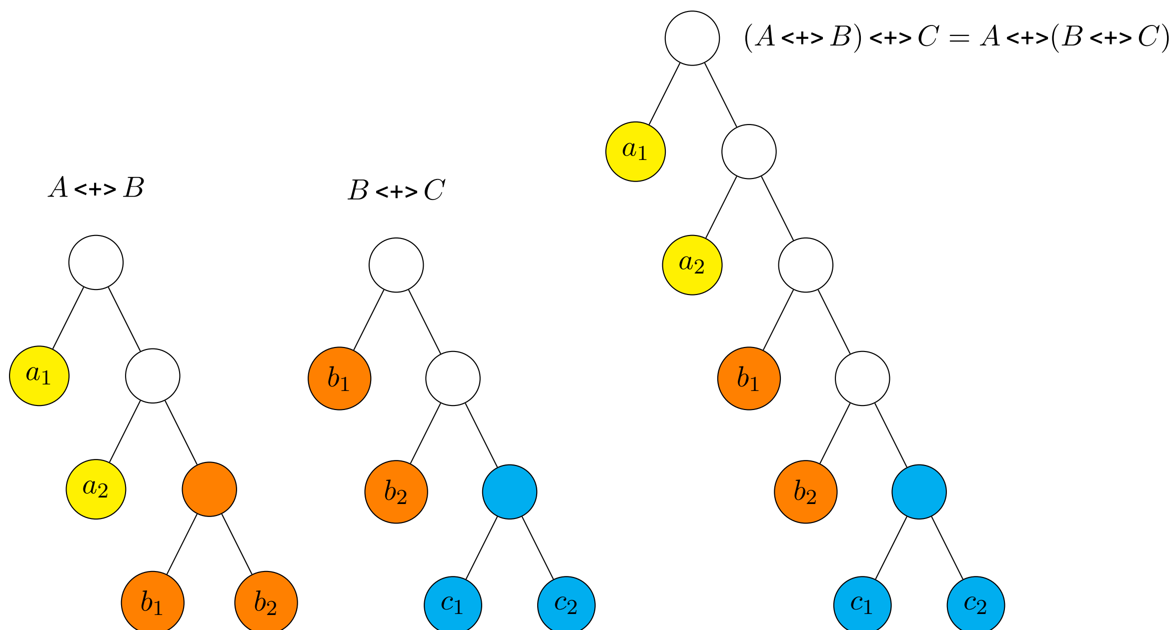


If we put the parentheses like this $(A \texttt{<+>} B) \texttt{<+>} C$ the result differs:



One way to get a better implementation fulfilling associativity is to always append the second tree at the last `Leaf` element of the first tree. To do this `combineK` has to walk the tree recursively: if the tree to which we are appending is a branch, we recursively append to the right subtree. When we reach a leaf we insert a new branch.

```
given SemigroupK[Tree] with
  def combineK[A](x: Tree[A], y: Tree[A]): Tree[A] =
    x match
      case Branch(l, r) => Branch(l, r <+> y)
      case leaf         => Branch(leaf, y)
```



b) Functor for binary trees

The map function for a binary tree has to apply the given `f` to all leaf nodes. To do that, we call `map` recursively if we encounter a branch:

```
given Functor[Tree] with
  def map[A, B](fa: Tree[A])(f: A => B): Tree[B] =
    fa match
      case Leaf(value)         => Leaf(f(value))
      case Branch(left, right) => Branch(left.map(f), right.map(f))
```

c) Foldable for binary trees

When folding binary trees, we also have two instead of one recursive call, similar to `map`. When encountering a branch in `foldLeft`, we first fold the left subtree and pass the result as starting value for the right subtree's fold. When we reach a leaf, we apply the function. In `foldRight` we do the same, but with swapped left and right.

```
given Foldable[Tree] with
  def foldLeft[A, B](fa: Tree[A], b: B)(f: (B, A) => B): B =
    fa match
      case Leaf(a) => f(b, a)
      case Branch(left, right) => right.foldLeft(left.foldLeft(b)(f))(f)

  def foldRight[A, B](fa: Tree[A], lb: Eval[B])(f: (A, Eval[B]) => Eval[B]): Eval[B] =
    fa match
      case Leaf(a) => f(a, lb)
      case Branch(left, right) => left.foldRight(right.foldRight(lb)(f))(f)
```

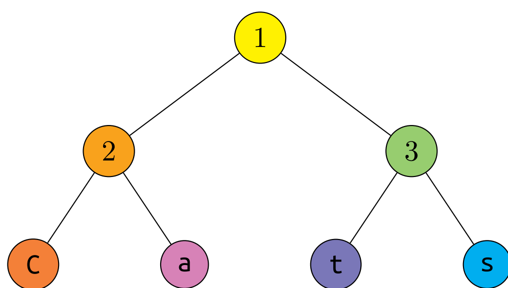
Like described in the exercise text, the presence of the `Eval` wrapper can be ignored in the solutoion, it behaves transparently. But when calling `foldRight`, we have to provide it.

You can see an example of using `foldRight` in the `toList` method, which we already saw in the lecture. Here the version adapted for Cats, using Eval:

```
import Eval.*
def toList[F[_],A](fa: F[A])(using Foldable[F]): List[A] =
  fa.foldRight[List[A]](now(Nil))((h, t) => now(h :: t.value)).value
```

The method `now` is required to wrap the values in Eval (and causes strict evaluation).

The single recursive calls then are as follows (only new starting values shown, $\varnothing$ = Nil):




```
(1).foldRight(∅)
(2).foldRight((3).fR(∅))
(2).foldRight((t).fR((s) .foldRight(∅)))
(2).foldRight((t).fR({s}))
(2).foldRight({t,s})
(C).foldRight((a).fR({t,s}))
(C).foldRight({a,t,s})
{C,a,t,s}
```

2 Monoid Isomorphisms

We are given two Monoids $(A, \oplus, z_A)$ and $(B, \odot, z_B)$, as well as an invertable function $f : A \to B$. We also know, that applying f to the result of A's operator has the same result as applying it to both operands and then using B's operator to combine them. Also, f maps the neutral element of A to the neutral element of B. In mathematical notation:

- $\forall x, y \in A : f(x \oplus y) = f(x) \odot f(y)$,
- $f(z_A) = z_B$.

We now want to generate z_B and $\odot$, i.e. the monoid for B, from the monoid of A and the function f.

Based on the above instructions, the solution for z_B is simple, it is $f(z_A)$.

For deriving $\odot$ we use the fact that f is invertable. Therefore $x = f(f^{-1}(x))$. Let $x' = f(x)$. Then from our requirements for f we get:

$$x' \odot y' = f(f^{-1}(x') \oplus f^{-1}(y'))$$

The `imap` function receives a monoid for A, as well as the function f and its inverse as g, this is enough to implement it following the above equation:

```
def imap[A, B](f: A => B, g: B => A)(using Monoid[A]): Monoid[B] = new Monoid[B]:
  def empty: B = f(Monoid[A].empty)
  def combine(x: B, y: B): B = f(g(x) |+| g(y))
```

Our g converts a B into A, so we can use A's monoid for combining the values. Then we use f to convert the result back into a B.

We now want to use this to create a monoid for our simple Box class, if the contained type has a monoid. Therefore `Box[A]` corresponds to `B` from before. As function for converting an A to a `Box[A]`, we pass Box's constructor to `imap`, for the other direction we pass a function that returns the `value` attribute of Box.

```
given boxMonoid[A](using Monoid[A]): Monoid[Box[A]] = imap(Box[A], _.value)
```

Cats includes an implementation of `imap` already, it can be found on the typeclass `Invariant`. So we also could simply delegate to that:

```
def imap[A, B](f: A => B, g: B => A)(using Monoid[A]): Monoid[B] =
  cats.Invariant[Monoid].imap(Monoid[A])(f)(g)
```