Solutions

Exercise 1.2-i

Determine the cardinality of `Either Bool (Bool, Maybe Bool) -> Bool`.

$$
\begin{aligned}
&|\texttt{Either Bool (Bool, Maybe Bool) -> Bool}| \\
&= |\texttt{Bool}|^{|\texttt{Either Bool (Bool, Maybe Bool)}|} \\
&= |\texttt{Bool}|^{|\texttt{Bool}|+|\texttt{Bool}|\times|\texttt{Maybe Bool}|} \\
&= |\texttt{Bool}|^{|\texttt{Bool}|+|\texttt{Bool}|\times(|\texttt{Bool}|+1)} \\
&= 2^{2+2\times(2+1)} \\
&= 2^{2+2\times 3} \\
&= 2^{2+6} \\
&= 2^{8} \\
&= 256
\end{aligned}
$$

Exercise 1.4-i

Use Curry–Howard to prove the exponent law that $a^b \times a^c = a^{b+c}$. That is, provide a function of the type `(b -> a) -> (c -> a) -> Either b c -> a` and one of `(Either b c -> a) -> (b -> a, c -> a)`.

```
productRule1To
    :: (b -> a)
    -> (c -> a)
    -> Either b c
    -> a
productRule1To f _ (Left b)  = f b
productRule1To _ g (Right c) = g c
```

```
productRule1From
    :: (Either b c -> a)
    -> (b -> a, c -> a)
productRule1From f = (f . Left, f . Right)
```

Notice that `productRule1To` is the familiar `either` function from `Prelude`.

Exercise 1.4-ii

Prove $(a \times b)^c = a^c \times b^c$.

```
productRule2To
    :: (c -> (a, b))
    -> (c -> a, c -> b)
productRule2To f = (fst . f, snd . f)
```

```
productRule2From
    :: (c -> a)
    -> (c -> b)
    -> c
    -> (a, b)
productRule2From f g c = (f c, g c)
```

Exercise 1.4-iii

Give a proof of $(a^b)^c = a^{b\times c}$. Does it remind you of anything from `Prelude`?

```
curry :: ((b, c) -> a) -> c -> b -> a
curry f c b = f (b, c)
```

```
uncurry :: (c -> b -> a) -> (b, c) -> a
uncurry f (b, c) = f c b
```

Both of these functions already exist in `Prelude`.

Exercise 2.1.3-i

If `Show Int` has kind CONSTRAINT, what's the kind of `Show`?

TYPE → CONSTRAINT

Exercise 2.1.3-ii

What is the kind of `Functor`?

(TYPE → TYPE) → CONSTRAINT

Exercise 2.1.3-iii

What is the kind of `Monad`?

(TYPE → TYPE) → CONSTRAINT

Exercise 2.1.3-iv

What is the kind of `MonadTrans`?

((TYPE → TYPE) → TYPE → TYPE) → CONSTRAINT

Exercise 2.4-i

Write a closed type family to compute `Not`.

```
type family Not (x :: Bool) :: Bool where
  Not 'True  = 'False
  Not 'False = 'True
```

Exercise 3-i

Which of these types are `Functor`s? Give instances for the ones that are.

Only `T1` and `T5` are `Functor`s.

```
instance Functor T1 where
  fmap f (T1 a) = T1 $ fmap f a
```

```
instance Functor T5 where
  fmap f (T5 aii) = T5 $ \bi -> aii $ bi . f
```

Exercise 5.3-i

Implement `Ord` for `HList`.

```
instance Ord (HList '[]) where
  compare HNil HNil = EQ
```

```
instance (Ord t, Ord (HList ts))
    => Ord (HList (t ': ts)) where
  compare (a :# as) (b :# bs) =
    compare a b <> compare as bs
```

Exercise 5.3-ii

Implement `Show` for `HList`.

```
instance Show (HList '[]) where
  show HNil = "HNil"
```

```haskell
instance (Show t, Show (HList ts))
    => Show (HList (t ': ts)) where
  show (a :# as) = show a <> " :# " show as
```

Exercise 5.3-iii

Rewrite the `Ord` and `Show` instances in terms of `All`.

```haskell
instance (All Eq ts, All Ord ts) => Ord (HList ts) where
  compare HNil HNil = EQ
  compare (a :# as) (b :# bs) =
    compare a b <> compare as bs
```

```haskell
instance (All Show ts) => Show (HList ts) where
  show HNil = "HNil"
  show (a :# as) = show a <> " :# " <> show as
```

Exercise 6.3-i

What is the rank of `Int -> forall a. a -> a`? Hint: try adding the explicit parentheses.

`Int -> forall a. a -> a` is rank-1.

Exercise 6.3-ii

What is the rank of `(a -> b) -> (forall c. c -> a) -> b`? Hint: recall that the function arrow is right-associative, so `a -> b -> c` is actually parsed as `a -> (b -> c)`.

`(a -> b) -> (forall c. c -> a) -> b` is rank-2.

Exercise 6.3-iii

What is the rank of `((forall x. m x -> b (z m x)) -> b (z m a)) -> m a`? Believe it or not, this is a real type signature we had to write back in the bad old days before `MonadUnliftIO`!

Rank-3.

Exercise 6.4-i

Provide a `Functor` instance for `Cont`. Hint: use lots of type holes, and an explicit lambda whenever looking for a function type. The implementation is sufficiently difficult that trying to write it point-free will be particularly mind-bending.

```haskell
instance Functor Cont where
  fmap f (Cont c) = Cont $ \c' ->
    c (c' . f)
```

Exercise 6.4-ii

Provide the `Applicative` instances for `Cont`.

```haskell
instance Applicative Cont where
  pure a = Cont $ \c -> c a
  Cont f <*> Cont a = Cont $ \br ->
    f $ \ab ->
      a $ br . ab
```

Exercise 6.4-iii

Provide the `Monad` instances for `Cont`.

```haskell
instance Monad Cont where
  return = pure
  Cont m >>= f = Cont $ \c ->
```

```
  m $ \a ->
    unCont (f a) c
```

Exercise 6.4-iv

There is also a monad transformer version of `Cont`. Implement it.

```
newtype ContT m a = ContT
  { unContT :: forall r. (a -> m r) -> m r
  }
```

The `Functor`, `Applicative` and `Monad` instances for `ContT` are identical to `Cont`.

Exercise 7.1-i

Are functions of type `forall a. a -> r` interesting? Why or why not?

These functions can only ever return constant values, as the polymorphism on their input doesn't allow any form of inspection.

Exercise 7.1-ii

What happens to this instance if you remove the `Show t =>` constraint from `HasShow`?

The `Show HasShow` instance can't be written without its `Show t =>` constraint—as the type inside `HasShow` is existential and Haskell doesn't know which instance of `Show` to use for the `show` call.

Exercise 7.1-iii

Write the `Show` instance for `HasShow` in terms of `elimHasShow`.

```
instance Show HasShow where
  show = elimHasShow show
```

Exercise 8.2-i

What is the role signature of `Either a b`?

```
type role Either representational representational
```

Exercise 8.2-ii

What is the role signature of `Proxy a`?

```
type role Proxy phantom
```

Exercise 10.1-i

Defunctionalize `listToMaybe :: [a] -> Maybe a`.

```
data ListToMaybe a = ListToMaybe [a]

instance Eval (ListToMaybe a) (Maybe a) where
  eval (ListToMaybe [])       = Nothing
  eval (ListToMaybe (a : _)) = Just a
```

Exercise 10.2-i

Defunctionalize `listToMaybe` at the type-level.

```
data ListToMaybe :: [a] -> Exp (Maybe a)
type instance Eval (ListToMaybe '[])        = 'Nothing
type instance Eval (ListToMaybe (a ': _1)) = 'Just a
```

Exercise 10.2-ii

Defunctionalize `foldr :: (a -> b -> b) -> b -> [a] -> b`.

```
data Foldr :: (a -> b -> Exp b) -> b -> [a] -> Exp b
type instance Eval (Foldr _1 b '[]) = b
type instance Eval (Foldr f b (a ': as)) =
  Eval (f a (Eval (Foldr f b as)))
```

Exercise 10.4-i

Write a promoted functor instance for tuples.

```
type instance Eval (Map f '(a, b))  = '(a, Eval (f b))
```

Exercise 11.2-i

Write `weaken :: OpenSum f ts -> OpenSum f (x ': ts)`

```
weaken :: OpenSum f ts -> OpenSum f (t ': ts)
weaken (UnsafeOpenSum n t) = UnsafeOpenSum (n + 1) t
```

Exercise 11.3-i

Implement `delete` for `OpenProduct`s.

```
delete
    :: forall key ts f
     . KnownNat (FindElem key ts)
    => Key key
    -> OpenProduct f ts
    -> OpenProduct f (Eval (DeleteElem key ts))
delete _ (OpenProduct v) =
  let (a, b) = V.splitAt (findElem @key @ts) v
   in OpenProduct $ a V.++ V.tail b
```

Exercise 11.3-ii

Implement `upsert` (update or insert) for `OpenProduct`s.

Hint: write a type family to compute a MAYBE NAT corresponding to the index of the key in the list of types, if it exists. Use class instances to lower this kind to the term-level, and then pattern match on it to implement `upsert`.

This is a particularly involved exercise. We begin by writing a FCF to compute the resultant type of the `upsert`:

```
type UpsertElem (key :: Symbol)
                (t :: k)
                (ts :: [(Symbol, k)]) =
  FromMaybe ('(key, t) ': ts)
    =<< Map (Placeholder1Of3 SetIndex '(key, t) ts)    .
    ↪  ❶
    =<< FindIndex (TyEq key <=< Fst) ts
```

Notice that at ❶ we refer to `Placeholder1Of3`—which is a little hack to get around the lack of type-level lambdas in FCFs. Its definition is this:

```
data Placeholder1Of3
    :: (a -> b -> c -> Exp r)
    -> b
    -> c
    -> a
    -> Exp r

type instance Eval (Placeholder1Of3 f b c a) =
  Eval (f a b c)
```

The actual implementation of `upsert` requires knowing whether we should insert or update. We will need to compute a MAYBE NAT for the type in question:

```
type UpsertLoc (key :: Symbol)
               (ts :: [(Symbol, k)]) =
  Eval (FindIndex (TyEq key <=< Fst) ts)
```

And we can use a typeclass to lower this the MAYBE NAT into a `Maybe Int`:

```
class FindUpsertElem (a :: Maybe Nat) where
  upsertElem :: Maybe Int

instance FindUpsertElem 'Nothing where
  upsertElem = Nothing

instance KnownNat n => FindUpsertElem ('Just n) where
  upsertElem =
    Just . fromIntegral . natVal $ Proxy @n
```

Finally, we're capable of writing `upsert`:

```
upsert
    :: forall key ts t f
     . FindUpsertElem (UpsertLoc key ts)
    => Key key
    -> f t
    -> OpenProduct f ts
    -> OpenProduct f (Eval (UpsertElem key t ts))
upsert k ft (OpenProduct v) =
  OpenProduct $ case upsertElem @(UpsertLoc key ts) of
```

```
    Nothing -> V.cons (Any ft) v
    Just n  -> v V.// [(n, Any ft)]
```

Exercise 12-i

Add helpful type errors to `OpenProduct`'s `update` and `delete` functions.

```
type family FriendlyFindElem (funcName :: Symbol)
                             (key :: Symbol)
                             (ts :: [(Symbol, k)]) where
  FriendlyFindElem funcName key ts =
    Eval (
      FromMaybe
           ( TypeError
           ( 'Text "Attempted to call `"
        ':<>: 'Text funcName
        ':<>: 'Text "' with key `"
        ':<>: 'Text key
        ':<>: 'Text "'."
        ':$$: 'Text "But the OpenProduct only has keys :"
        ':$$: 'Text "  "
        ':<>: 'ShowType (Eval (Map Fst ts))
           )) =<< FindIndex (TyEq key <=< Fst) ts)


update
    :: forall key ts t f
     . ( KnownNat (FriendlyFindElem "update" key ts)
       , KnownNat (FindElem key ts)
```

```
        )
    => Key key
    -> f t
    -> OpenProduct f ts
    -> OpenProduct f (Eval (UpdateElem key t ts))
update _ ft (OpenProduct v) =
  OpenProduct $ v V.// [(findElem @key @ts, Any ft)]


delete
    :: forall key ts f
     . ( KnownNat (FriendlyFindElem "delete" key ts)
       , KnownNat (FindElem key ts)
       )
    => Key key
    -> OpenProduct f ts
    -> OpenProduct f (Eval (DeleteElem key ts))
delete _ (OpenProduct v) =
  let (a, b) = V.splitAt (findElem @key @ts) v
   in OpenProduct $ a V.++ V.tail b
```

These functions could be cleaned up a little by moving the `FriendlyFindElem` constraint to `findElem`, which would remove the need for both constraints.

Exercise 12-ii

Write a closed type family of kind $[K] \to$ ERRORMESSAGE that pretty prints a list. Use it to improve the error message from `FriendlyFindElem`.

```
type family ShowList (ts :: [k]) where
  ShowList '[] = Text ""
  ShowList (a ': '[]) = ShowType a
  ShowList (a ': as)  =
    ShowType a ':<>: Text ", " ':<>: ShowList as
```

```
type family FriendlyFindElem2 (funcName :: Symbol)
                              (key :: Symbol)
                              (ts :: [(Symbol, k)]) where
  FriendlyFindElem2 funcName key ts =
    Eval (
      FromMaybe
            ( TypeError
            ( 'Text "Attempted to call `"
        ':<>: 'Text funcName
        ':<>: 'Text "' with key `"
        ':<>: 'Text key
        ':<>: 'Text "'."
        ':$$: 'Text "But the OpenProduct only has keys :"
        ':$$: 'Text "  "
        ':<>: ShowList (Eval (Map Fst ts))
            )) =<< FindIndex (TyEq key <=< Fst) ts)
```

Exercise 12-iii

See what happens when you directly add a `TypeError` to the context of a function (eg. `foo :: TypeError ... => a`). What happens? Do you know why?

GHC will throw the error message immediately upon attempting to compile the module.

The reason why is because the compiler will attempt to discharge any extraneous constraints (for example, `Show Int` is always in scope, and so it can automatically be discharged.) This machinery causes the type error to be seen, and thus thrown.

Exercise 13.2-i

Provide a generic instance for the `Ord` class.

```
class GOrd a where
  gord :: a x -> a x -> Ordering

instance GOrd U1 where
  gord U1 U1 = EQ

instance GOrd V1 where
  gord _ _ = EQ

instance Ord a => GOrd (K1 _1 a) where
  gord (K1 a) (K1 b) = compare a b

instance (GOrd a, GOrd b) => GOrd (a :*: b) where
  gord (a1 :*: b1) (a2 :*: b2) = gord a1 a2 <> gord b1
  ↪  b2
```

```haskell
instance (GOrd a, GOrd b) => GOrd (a :+: b) where
  gord (L1 a1) (L1 a2) = gord a1 a2
  gord (R1 b1) (R1 b2) = gord b1 b2
  gord (L1 _)  (R1 _)  = LT
  gord (R1 _)  (L1 _)  = GT
```

```haskell
instance GOrd a => GOrd (M1 _x _y a) where
  gord (M1 a1) (M1 a2) = gord a1 a2
```

```haskell
genericOrd :: (Generic a, GOrd (Rep a)) => a -> a ->
↪  Ordering
genericOrd a b = gord (from a) (from b)
```

Exercise 13.2-ii

Use `GHC.Generics` to implement the function `exNihilo :: Maybe a`. This function should give a value of `Just a` if `a` has exactly one data constructor which takes zero arguments. Otherwise, `exNihilo` should return `Nothing`.

```haskell
class GExNihilo a where
  gexNihilo :: Maybe (a x)
```

```haskell
instance GExNihilo U1 where
  gexNihilo = Just U1
```

```haskell
instance GExNihilo V1 where
```

```haskell
  gexNihilo = Nothing

instance GExNihilo (K1 _1 a) where
  gexNihilo = Nothing

instance GExNihilo (a :*: b) where
  gexNihilo = Nothing

instance GExNihilo (a :+: b) where
  gexNihilo = Nothing

instance GExNihilo a => GExNihilo (M1 _x _y a) where
  gexNihilo = fmap M1 gexNihilo
```

Exercise 15.3-i

Provide instances of `SingI` for lists.

```haskell
instance SingI '[] where
  sing = SNil

instance (SingI h, SingI t) => SingI (h ': t) where
  sing = SCons sing sing
```

Exercise 15.4-i

Give instances of `SDecide` for `Maybe`.

```
instance SDecide a => SDecide (Maybe a) where
  SJust a %~ SJust b =
    case a %~ b of
      Proved Refl -> Proved Refl
      Disproved _ -> Disproved $ const undefined
  SNothing %~ SNothing = Proved Refl
```

Exercise 15.5-i

Provide an instance of `Ord` for `Sigma` by comparing the `fs` if the singletons are equal, comparing the singletons at the term-level otherwise.

```
instance ( Dict1 Eq  (f :: k -> Type)
         , Dict1 Ord f
         , SDecide k
         , SingKind k
         , Ord (Demote k)
         ) => Ord (Sigma f) where
  compare (Sigma sa fa) (Sigma sb fb) =
    case sa %~ sb of
      Proved Refl ->
```

```
    case dict1 @Ord @f sa of
      Dict -> compare fa fb
  Disproved _ ->
    compare (fromSing sa) (fromSing sb)
```