Chapter 1

Implementation

This chapter describes low-level details of the `clerk` library implementation. First, the used types and their purpose are explained. Following that, the main helper functions are presented. Finally, a simple example is given to demonstrate the capabilities of `clerk`.

1.1 Types

The core type of the library is the `RowBuilder`. This is a monad that allows to construct a template for a row of data. It keeps track of the coordinates of the current cell via the `StateT Coords m a` transformer. It writes the new cells into a template via the `Writer (Template input output) a`.

```
newtype RowBuilder input output a = RowBuilder
  { unBuilder :: StateT Coords (Writer (Template input output)) a
  }
  deriving (
    Functor, Applicative, Monad, MonadState Coords,
    MonadWriter (Template input output)
  )
```

The row templates are applied to a list of inputs, producing a template for a

table. In this table, each cell has an address, data, and formatting. A special type is used to construct formulas to produce data.

1.1.1 Addresses

The `Coords` denote the address of a cell. This data type has a `Num` instance that provides cell arithmetics with them like shifts along sheet axes in the directions given as another `Coords`. Additionally, this type has a `Show` instance that translates it into valid spreadsheet addresses.

```
data Coords = Coords {_row :: Int, _col :: Int}
```

Typeclasses `ToCoords` and `FromCoords` allow to generalize working with data that is convertible to and from `Coords`. Based on these typeclasses, a pair of lenses is provided for convenient work with `Coords`-like data types.

```
row :: (ToCoords a, FromCoords a) => Lens' a Int
col :: (ToCoords a, FromCoords a) => Lens' a Int
```

Such data types include `Refs`, which are addresses of cells plus a phantom type to allow type-safe operations. For example, for a `Ref Int`, arithmetic operations are only allowed with another `Ref Int`. `Ref` inherits the `Num` instance of `Coords`.

```
newtype Ref a = Ref {unRef :: Coords} deriving newtype (Num)
```

The phantom type transformations are made possible via the `UnsafeChangeType` class.

```
class UnsafeChangeType (a :: Type -> Type) where
  unsafeChangeType :: forall c b. a b -> a c
```

1.1.2 Cell data

When building a template, all `inputs` are translated into `CellData`, which unites the data types from `xlsx`.

```
data CellData
  = CellFormula X.CellFormula
  | CellValue X.CellValue
  | CellComment X.Comment
  | CellEmpty
```

That is why, one can use a type synonym for building row templates.

```
type Row input a = RowBuilder input CellData a
```

There is a typeclass `ToCellData` that allows to work with arbitrary types convertible to `CellData`.

```
class ToCellData a where
  toCellData :: a -> CellData
```

1.1.3 Formatting

To store the additional information about a cell, a `CellTemplate` type is introduced.

```
data CellTemplate input output = CellTemplate
  { mkOutput :: input -> output
  , fmtCell :: FormatCell
  , columnsProperties :: Maybe X.ColumnsProperties
  }
```

The `FormatCell` type synonym denotes a function that produces a formatted cell based on that cell's address, index in the input list, and the data.

```
type FormatCell =
  forall a b. (ToCoords a, ToCellData b) =>
    a -> InputIndex -> CellData -> X.FormattedCell
```

1.1.4 Formulas

The spreadsheet formulas are modeled via recursive data types and have a phantom type to store the resulting type of a formula.

```
data Expr t
  = EBinOp BinaryOperator (Expr t) (Expr t)
  | EFunction String [Expr t]
  | ERef (Ref t)
  | ERange (Ref t) (Ref t)
```

To introduce the new functionality on top of `Expr`, the `Formula` is used.

```
newtype Formula t = Formula {unFormula :: Expr t}
  deriving newtype (UnsafeChangeType, Show)
```

It is accompanied by a type class that allows conversion to a `Formula`.

```
class ToFormula a where
  toFormula :: a -> Formula t
```

Formulas are constructed from values and addresses combined via operators and functions.

1.1.4.1 Operators

A number of operators are used to build formulas. These operators resemble the spreadsheet ones.

- For constructing ranges

  ```
  (.:) :: forall c a b. Ref a -> Ref b -> Formula c
  ```

- Arithmetic operators

```
type NumOperator a b c = (Num a, ToFormula (b a), ToFormula (c a))
(.+) :: NumOperator a b c
(.-) :: NumOperator a b c
(./) :: NumOperator a b c
(.*) :: NumOperator a b c
(.^) :: NumOperator a b c
```

- Operators that produce boolean values

```
type BoolOperator a b c = (Ord a, ToFormula (b a), ToFormula (c a))
(.<) :: BoolOperator a b c
(.>) :: BoolOperator a b c
(.<=) :: BoolOperator a b c
(.>=) :: BoolOperator a b c
(.=) :: BoolOperator a b c
(.<>) :: BoolOperator a b c
```

1.1.4.2 Functions

A user may want to construct custom functions. It is made possible via another typeclass and a helper method. To set the types of arguments, a user can specify the type `t`.

```
type FunName = String

class MakeFunction t where
  makeFunction :: FunName -> [Formula s] -> t

fun :: MakeFunction t => FunName -> t
fun n = makeFunction n []
```

Due to an instance of `IsString`, it is possible to use function names as `Haskell` functions.

```
instance {-# OVERLAPPABLE #-} MakeFunction t => IsString t where
  fromString :: MakeFunction t => String -> t
  fromString = fun
```