

MiniFrame

MiniFrame provides a simple and user-friendly interface for working with datasets in a tabular format. The idea of a miniframe comes from programming language R's `data.frame` object. MiniFrame is just a very stripped down version of R's `data.frame`. It provides just enough power and flexibility to conveniently explore the data. The function names as well as their functionalities are so simple that even those who are not familiar with Haskell can easily use this package for their convenience.

For the sake of simplicity, everything in a miniframe is of the type `String` (the same as `[Char]`). Yet, this does not make interacting with miniframes very inconvenient, nor does it limit its flexibility. A separate section in the documentation will be dedicated to this issue. MiniFrame heavily utilizes Haskell's `List` data type meaning that everything within miniframes including the fundamental data types can be manipulated directly using built-in list functions such as `map`, `concatMap`, `foldl`, `foldr`, `scanl`, `scanr`, and so forth.

NOTE: The PDF version of this documentation is available [here](#).

Documentation

- [Data types](#)
- [Construction](#)
- [Accessing the data](#)
- [Counting the dimensions](#)
- [Modifications](#)
- [Removal](#)
- [Pretty-printing](#)
- [Additional operations](#)
- [Type conversion and numeric computation](#)
- [Relational Algebra](#)
- [Leveraging Haskell's built-in goodness](#)
- [Installation](#)

Those who want to take a look at Haddock-generated documentation, jump to this [link](#). Note that the library is not well-documented on Haddock.

Data types

MiniFrame has one fundamental data type, it is called `MiniFrame`. Its definition is shown below.

```
data MiniFrame = MiniFrame
  { name    :: !Name
  , header :: !Header
```

```
, rows    :: ![Row]
} deriving (Eq, Show)
```

Most of the functions operate on this data type. As it can be seen above, there are auxiliary types, which are defined as follows:

```
type Index  = Int
type Name   = String
type Header = [Name]
type Row    = [String]
type Column = [String]
```

Note that the user does not need to be familiar with these types other than knowing the fact that types `Header`, `Row`, and `Column` are just the lists of the type `[String]`. These facts make it super simple to navigate through and manipulate the dataset as well as to perform numeric computations.

Construction

Function	Description	Signature
<code>fromSample</code>	construct out of a sample	<code>MiniFrame</code>
<code>fromNull</code>	construct out of nothing	<code>MiniFrame</code>
<code>fromRows</code>	construct out of rows	<code>Name -> Header -> [Row] -> MiniFrame</code>
<code>fromColumns</code>	construct out of columns	<code>Name -> Header -> [Column] -> MiniFrame</code>
<code>fromCSV</code>	construct out of CSV file	<code>String -> IO MiniFrame</code>

NOTE #1: Do not let names `fromSample` and `fromNull` deceive you. The only thing these two functions do is a construction of a miniframe from a sample name, header, and rows and from nothing (resulting in an empty miniframe). Just for consistency, all these functions have a prefix `from`.

NOTE #2: These are recommended ways (A.K.A. *smart* constructors) to build a `MiniFrame`. Though, one could also use the `MiniFrame` data type constructor to build it, but in this case, the error-handling will be left for the user (`∴`).

Example usage:

```
import MiniFrame.Frames

main :: IO ()
```

```

main = do
  -- A sample miniframe
  let smf = fromSample

  -- A null miniframe
  let nmf = fromNull

  -- Constructing a miniframe from rows
  let rs = [ ["Bianca" , "21", "Apple" ]
            , ["Brian"  , "20", "Orange"]
            , ["Bethany", "19", "Banana"]
            ]

  let frmf = fromRows "Favorite fruits" ["Name", "Age", "Favorite Fruit"] rs

  -- Constructing a miniframe from columns
  let cs = [ ["Walter", "John", "Eric"]
            , ["500"    , "700" , "600" ]
            , ["18"    , "20"  , "19"  ]
            ]

  let fcmf = fromColumns "Game scores" ["Player", "Score", "Age"] cs

  -- Constructing a miniframe from CSV file
  mf <- fromCSV "schools.csv"

  return ()

```

Accessing the data

Function	Description	Signature
<code>nameOf</code>	get the name	<code>MiniFrame -> Name</code>
<code>headerOf</code>	get the header	<code>MiniFrame -> Header</code>
<code>rowsOf</code>	get the rows	<code>MiniFrame -> [Row]</code>
<code>columnsOf</code>	get the columns	<code>MiniFrame -> [Column]</code>
<code>headOf</code>	get the head	<code>MiniFrame -> Row</code>
<code>tailOf</code>	get the tail	<code>MiniFrame -> [Row]</code>

Example usage:

```

import MiniFrame.Frames

main :: IO ()
main = do
    let n  = nameOf    fromSample -- Get the name
    let h  = headerOf  fromSample -- Get the header
    let rs = rowsOf    fromSample -- Get the rows
    let cs = columnsOf fromSample -- Get the columns
    let ho = headOf    fromSample -- Get the head
    let to = tailOf    fromSample -- Get the tail

    return ()

```

Counting the dimensions

Function	Description	Signature
rowsNum	get the number of rows	MiniFrame -> Int
columnsNum	get the number of columns	MiniFrame -> Int
entriesNum	get the number of cells	MiniFrame -> Int

Example usage:

```

import MiniFrame.Frames

main :: IO ()
main = do
    let rsn = rowsNum    fromSample -- Number of rows
    let csn = columnsNum fromSample -- Number of columns
    let esn = entriesNum fromSample -- Number of cells

    return ()

```

Modifications

Function	Description	Signature
prependRow	add a row to the beginning	Row -> MiniFrame -> MiniFrame
prependColumn	add a column to the beginning	Name -> Column -> MiniFrame -> MiniFrame
appendRow	add a row to the end	Row -> MiniFrame -> MiniFrame

Function	Description	Signature
<code>appendColumn</code>	add a column to the end	<code>Name -> Column -> MiniFrame -> MiniFrame</code>
<code>insertRow</code>	add a row by given index	<code>Index -> Row -> MiniFrame -> MiniFrame</code>
<code>insertColumn</code>	add a column by given index	<code>Index -> Name -> Column -> MiniFrame -> MiniFrame</code>
<code>renameMf</code>	rename a miniframe	<code>Name -> MiniFrame -> MiniFrame</code>

Example usage:

```
import MiniFrame.Frames

main :: IO ()
main = do
    let newRow      = map show [1..4]    -- New row
    let newColumn   = map show [1..8]    -- New column

    let prs = prependRow      newRow      fromSample -- Prepend a row
    let ars = prependColumn "Nums" newColumn fromSample -- Prepend a column

    let ars = appendRow      newRow      fromSample -- Appending a row
    let acs = appendColumn "Nums" newColumn fromSample -- Appending a column

    let irs = insertRow      1          newRow      fromSample -- Inserting a row
    let ics = insertColumn 3 "Nums" newColumn fromSample -- Inserting a column

    let rmf = renameMf "New Name" fromSample -- Rename miniframe

    return ()
```

Removal

Function	Description	Signature
<code>removeRowByIndex</code>	remove a row by index	<code>Index -> MiniFrame -> MiniFrame</code>
<code>removeColumnByName</code>	remove a column by name	<code>Name -> MiniFrame -> MiniFrame</code>

Example usage:

```
import MiniFrame.Frames

main :: IO ()
main = do
    let rrs = removeRowByIndex 2 fromSample -- Remove a row by index
    let rcs = removeColumnByName "C4" fromSample -- Remove a column by name
```

Pretty-printing

Function	Description	Signature
printName	print the name	MiniFrame -> IO ()
printHeader	print the header	MiniFrame -> IO ()
printRow	print the row by index	Index -> MiniFrame -> IO ()
printRows	print the rows	MiniFrame -> IO ()
printColumn	print the column by name	Name -> MiniFrame -> IO ()
printColumns	print the columns	MiniFrame -> IO ()
printMF	print the miniframe	MiniFrame -> IO ()

Example usage:

```
import MiniFrame.Frames

main :: IO ()
main = do
    printName      fromSample -- Pretty-print the name
    printHeader    fromSample -- Pretty-print the header
    printRow 1     fromSample -- Pretty-print the row by index
    printRows      fromSample -- Pretty-print all rows
    printColumn "C4" fromSample -- Pretty-print the column C4
    printColumns   fromSample -- Pretty-print all columns
    printMF        fromSample -- Pretty-print the miniframe

    return ()
```

Additional operations

Function	Description	Signature
rowByIndex	get the row by index	Index -> MiniFrame -> Row
columnByName	get the column by name	Name -> MiniFrame -> Column

Function	Description	Signature
columnByIndex	get the column by index	Index -> MiniFrame -> Column

Example usage:

```
import MiniFrame.Frames

main :: IO ()
main = do
    let rbi = rowByIndex    5    fromSample -- Row by index
    let cbn = columnByname  "C3" fromSample -- Column by name
    let cbi = columnByIndex 1    fromSample -- Column by index

    return ()
```

Type conversion and numeric computation

Function	Description	Signature
toInt	Convert a column of string to a column of fixed-precision integers	Name -> MiniFrame -> [Int]
toDecimal	Convert a column of stings to a column of fixed-precision decimals	Name -> MiniFrame -> [Float]
toBigInt	Convert a column of string to a column of arbitrary precision integers	Name -> MiniFrame -> [Integer]
toBigDecimal	Convert a column of stings to a column of arbitrary decimals	Name -> MiniFrame -> [Double]

NOTE: Word “arbitrary” here refers to a size that can be handled by the machine.

Example usage:

```
import MiniFrame.Frames

main :: IO ()
main = do
    let mf = fromColumns

        -- Name
```

```

"MiniFrame"

-- Header
["Name", "Quantity", "Total Spending"]

-- Columns
[ ["Paul" , "Ryan", "Kim"  ],
  ["10"   , "20"  , "30"   ],
  ["100.0", "200" , "300.0"]
]

-- Calculating the total quantity
let tq = sum $ toInt "Quantity" miniframe

-- Calculating the average number of dollars spent per person
let ad = sum (toDecimal "Total Spending" miniframe) / 3

-- Calculating the total quantity using arbitrary precision integers
let tqa = sum $ toBigInt "Quantity" miniframe

-- Calculating the average number of dollars spent per person
-- using arbitrary precision decimals
let ada = sum (toBigDecimal "Total Spending" miniframe) / 3

return ()

```

Relational algebra

Function	Description	Signature
union	union of two miniframes	MiniFrame -> MiniFrame -> MiniFrame
diff	difference of two miniframes	MiniFrame -> MiniFrame -> MiniFrame
intersect	intersection of two miniframes	MiniFrame -> MiniFrame -> MiniFrame
project	project a miniframe	[Name] -> MiniFrame -> MiniFrame
rename	rename a column	Name -> Name -> MiniFrame -> MiniFrame

Example usage:


```

import MiniFrame.Frames
import MiniFrame.Relational

main :: IO ()
main = do
    let umf = fromSample `union`      fromSample -- Union
    let dmf = fromSample `diff`        fromSample -- Difference
    let imf = fromSample `intersect`    fromSample -- Intersection
    let pmf = project      ["C2","C4"] fromSample -- Projection
    let cmf = fromColumns "R1" ["C1","C2"] [ ["A","B","C"], ["1","2","3"] ]
              `cartprod`
              fromColumns "R2" ["C3","C4"] [ ["4","5","6"], ["D","E","F"] ]
    let rmf = rename      "C1" "FC"  fromSample -- Rename

    return ()

```

Leveraging Haskell's built-in goodness

Recall that miniframe is built on top of Haskell's built-in list data type which is arguably the most powerful data type in Haskell. This means that we can use the built-in list manipulation functions directly.

```

import MiniFrame.Frames

main :: IO ()
main = do
    let mf = fromRows

        -- Name
        "MiniFrame with numeric data"

        -- Header
        ["Product","Company","Value"]

        -- Rows
        [ ["FP toolkit" , "Haskell Enterprises", "10000000000000000000.00"]
        , ["OOP toolkit", "C++ Enterprises"      , "10000000000000000000.00" ]
        , ["PP toolkit" , "C Enterprises"         , "10000000000000000000.00" ]
        , ["LP toolkit" , "Prolog Enterprises"    , "10000000000000000000.00" ]
        ]

        -- Print out the average of all prices (notice the built-in sum function)
    print $ sum $ toBigDecimal "Value" mf

```

```
-- Get the particular entry ("Haskell Enterprises" in this case)
-- notice Haskell's built-in head function
print $ head $ columnByName "Company" mf
```

Using the built-in operations, however, does have its drawbacks such as no error messages if one messes up the structure of a miniframe. In other words, one is on its own once the borders of MiniFrame are crossed.

Installation

The package can be installed via [Cabal](#). Run the commands shown below to install the package.

```
git clone https://github.com/oniani/miniframe.git
cabal install
```

License

MIT License