TableToLongForm

Literate Program

Jimmy Oh

Department of Statistics
University of Auckland

Corresponds to R Package Version 1.3.1 (release)

Abstract

TableToLongForm automatically converts hierarchical Tables intended for a human reader into a simple LongForm Dataframe that is machine readable. It does this by recognising positional cues present in the hierarchical Table (which would normally be interpreted visually by the human brain) to decompose, then reconstruct the data into a LongForm Dataframe. This is the Literate Program for TableToLongForm and contains the entirety of the code with accompanying documentation.

Contents

On Literate Programs

This software is presented as a *literate program* written in the *noweb* format (Ramsey 1994). It serves as both the documentation and container of the literate program. The `noweb` file can be used to produce both the *literate document* and the executable code.

The literate document is separated into *documentation chunks* and named *code chunks*. Each *code chunk* can contain code directly, or contain references to other *code chunks* which act as placeholders for the contents of the respective *code chunk*. The name of each *code chunk* should serve as a short description of the code it contains. Thus each *code chunk* provides an overview of its purpose by either directly containing code, or by containing the names of other *code chunks*. The reader is then free to delve deeper into the respective *code chunks* if desired.

1 Introduction

This Literate Document delves deeply into the source code for TableToLongForm. Most users will probably find the Home Page for TableToLongForm[1] more informative.

The Literate Program is a constant work-in-progress, and some of the sections may have out of date documentation, or be lacking in documentation completely.

[1] `https://www.stat.auckland.ac.nz/~joh024/Research/TableToLongForm/`

2 Code Overview

Unless the Table is horrible beyond mortal imagination, it should have some kind of pattern, such that a human will be able to discern the structure and hence understand the data it represents. This code attempts to algorithmically search for such patterns, discern the structure, then reconstruct the data into a LongForm Dataframe.

The task can be seen to consist of three phases:

- Phase One is Identification, which involves identifying the rows and columns where the labels and the data can be found.
- Phase Two is Discerning the Parentage, which involves identifying the hierarchical structure of the data, based on the row and column labels.
- Phase Three is Reconstruction, where we use what we've found in the first two phases to reconstruct the data into a LongForm Dataframe.

3a ⟨*TableToLongForm.R* 3a⟩≡
  ⟨*document header* 3b⟩
  ⟨*Front End* 4a⟩
  ⟨*Back End* 5b⟩
  ⟨*Identification* 9a⟩
  ⟨*Discern Parentage* 15a⟩
  ⟨*Reconstruction* 25a⟩

This code is written to file `TableToLongForm.R`.

We place a document header at the top of the extracted code to encourage people to read the literate description rather than attempting to study the code alone.

3b ⟨*document header* 3b⟩≡
```
##----------------------------------------------------------------------
## The code in this .R file is machine generated from the literate
##  program, TableToLongForm.Rnw
## Documentation can be found in the literate description for this
##  program, TableToLongForm.pdf
##----------------------------------------------------------------------
```

2.1 Front End

The main function `TableToLongForm` is defined here. For most users this is the only function they will call. The arguments are as follows:

Table the Table to convert, given as a `character matrix`. Also accepts a `data.frame`, which is coerced to a `matrix` with a warning.

IdentResult an optional `list` specifying the locations of the various elements of the `Table`. By default this is automatically generated but it can be specified manually where the automatic detection fails.

IdentPrimary, IdentAuxiliary, ParePreRow, ParePreCol specify the algorithms TableToLongForm should use. Refer to the respective sections for more details.

fulloutput if `TRUE`, returns a `list` containing additional information primarily useful for diagnostic purposes. Otherwise, and by default, the function only returns the converted `data.frame` object.

diagnostics a `character vector` specifying the name of the file diagnostic output will be written to. Can also be `TRUE`, in which case the file name will be the name of the object specified in `Table`.

diagnostics.trim a `logical` indicating whether the diagnostics output should be trimmed. A good idea to keep `TRUE` (default) as trimmed output is generally more useful.

This function handles some busy-work, such as coercing the `Table` to a `matrix` (with a warning) and setting up the diagnostics output file. It then calls `ReconsMain` which handles the real meat of the conversion.

In the package version of TableToLongForm, this, and some back-end functions, are the only functions that are exported, the rest are hidden in the package namespace (which is still accessible, just not as easily). If sourcing in the raw .R file, the majority of the supporting functions are not hidden and can be accessed directly from the Global Environment.

4a ⟨*Front End* 4a⟩≡
```
TableToLongForm =
  function(Table, IdentResult = NULL,
           IdentPrimary = "combound",
           IdentAuxiliary = "sequence",
           ParePreRow = NULL,
           ParePreCol = c("mismatch", "misalign", "multirow"),
           fulloutput = FALSE,
           diagnostics = FALSE, diagnostics.trim = TRUE){
    ⟨Check Table arg 4b⟩
    ⟨Setup diagnostics file 5a⟩
    fullout = ReconsMain(matFull = Table, IdentResult,
      IdentPrimary, IdentAuxiliary, ParePreRow, ParePreCol)
    if(fulloutput) fullout else fullout$datafr
  }
```
Uses `IdentResult` 26a and `ReconsMain` 25b.

4b ⟨*Check Table arg* 4b⟩≡
```
if(is.data.frame(Table)){
  warning("Table supplied is a data.frame.\n",
          "TableToLongForm is designed for a character matrix.\n",
          "The data.frame is being coerced to a matrix but this\n",
          "may lead to unexpected results.",
          immediate. = TRUE)
  Table = as.matrix(Table)
}
if(!is.matrix(Table))
  stop("Table argument must be a matrix or a data.frame")
```

5a ⟨*Setup diagnostics file* 5a⟩≡
```
if(diagnostics != FALSE){
  if(!is.character(diagnostics))
    diagnostics = deparse(substitute(Table))
  assign("TCRunout", file(paste0(diagnostics, ".TCRunout"), "w"),
         envir = TTLFBaseEnv)
  assign("TCtrim", diagnostics.trim, envir = TTLFBaseEnv)
  on.exit({
    with(TTLFBaseEnv, {
      close(TCRunout)
      rm(TCRunout)
      rm(TCtrim)
    })
  })
}
```
Uses `TTLFBaseEnv` 7.

2.2 Back End

Various code, mainly to help produce diagnostic output, can be ignored by most users.

5b ⟨*Back End* 5b⟩≡
```
⟨BErbinddf 5c⟩
⟨BEprintplist 6a⟩
⟨BEattrLoc 6b⟩
⟨BETCRsink 7⟩
⟨BETTLFalias 8⟩
```

rbinddf An `rbind` method to handle data.frames with differing column names. Does not check if arguments are actually data.frames, so can break easily.

5c ⟨*BErbinddf* 5c⟩≡
```
rbinddf =
  function(..., deparse.level = 0){
    bindlist = list(...)
    nameunion = NULL
    for(j in 1:length(bindlist))
      nameunion = union(nameunion, colnames(bindlist[[j]]))
    for(j in 1:length(bindlist)){
      curdf = bindlist[[j]]
      namediff = setdiff(nameunion, colnames(curdf))
      matdummy = matrix(NA, nrow = nrow(curdf), ncol = length(namediff),
        dimnames = list(NULL, namediff))
      bindlist[[j]] = cbind(curdf, matdummy)
    }
    outdf = do.call(rbind,
      c(bindlist, list(deparse.level = deparse.level)))
    for(j in 1:ncol(outdf))
      if(mode(outdf[,j]) == "character") outdf[,j] = factor(outdf[,j])
    outdf
  }
```
Defines:
  `rbinddf`, used in chunk 30a.

print.plist A print method for class `plist`, which are nested lists with a numeric vector at the lowest level; `print.default` is rather inefficient in displaying such nested lists.

6a ⟨*BEprintplist* 6a⟩≡

```
print.plist =
  function(x, ...){
    plistC = function(plist){
      pLoc = attr(plist, "Loc")
      if(is.list(plist)){
        namevec = names(plist)
        if(!is.null(pLoc))
          namevec = paste0(names(plist),
            " (", pLoc[,"rows"], ", ", pLoc[,"cols"], ")")
        namelist = as.list(namevec)
        for(i in 1:length(namelist))
          namelist[[i]] =
            c(paste("+", namelist[[i]]),
              paste("-", plistC(plist[[i]])))
        do.call(c, namelist)
      } else{
        if(!is.null(names(plist))){
          namevec = names(plist)
          if(!is.null(pLoc))
            namevec = paste0(names(plist),
              " (", plist, ", ", pLoc[,"cols"], ")")
          paste("+", namevec)
        } else paste(plist, collapse = " ")
      }
    }

    cat(plistC(x), sep = "\n")
  }
```

attrLoc A function for creating a `plist` object and binding location information (rows and cols) to it.

6b ⟨*BEattrLoc* 6b⟩≡

```
attrLoc =
  function(plist, rows = NULL, cols = NULL){
    attr(plist, "Loc") = cbind(rows, cols)
    class(plist) = "plist"
    plist
  }
```

Defines:
`attrLoc`, used in chunks 20 and 22–24.

TCRsink Sinks the output to `TCRunout` for diagnostic output. Requires the existence of `TCRunout` which is created by the main function `TableToLongForm` when `diagnostics = TRUE`.

Spaces may be introduced by `match.call`, thus any spaces in the args of *variables to sink* (that is, the arguments supplied via `...`) are removed without warning.

We also create the `TTLFBaseEnv` here, which is currently only used to temporarily store `TCRunout`.

7 ⟨*BETCRsink* 7⟩≡

```
TCRsink =
  function(ID, ...)
  if(exists("TCRunout", envir = TTLFBaseEnv)){
    varlist = list(...)
    names(varlist) = gsub(" ", "", as.character(match.call()[-(1:2)]))
    TCtrim = get("TCtrim", envir = TTLFBaseEnv)
    with(TTLFBaseEnv, sink(TCRunout))
    for(i in 1:length(varlist)){
      cat("###TCR", ID, names(varlist)[i], "\n")
      curvar = varlist[[i]]
      if(TCtrim == TRUE){
        curvar = head(curvar)
        if(is.matrix(curvar) || is.matrix(curvar))
          if(ncol(curvar) > 6)
            curvar = curvar[,1:6]
      }
      print(curvar)
    }
    sink()
  }
TTLFBaseEnv = new.env()
```

Defines:

`TCRsink`, used in chunks 10, 16–18, 20, 22–24, 26b, 27, and 30b.
`TTLFBaseEnv`, used in chunks 5a and 8.

TTLFalias Used for the new Modular System. Check "Working with Modules" documentation available from the main website.

Should add a check to aliasAdd for existing rows with same alias (and same Type, probably ok to allow same alias for different Types).

8 ⟨*BETTLFalias* 8⟩≡

```
with(TTLFBaseEnv, {aliasmat = NULL})
TTLFaliasAdd =
  function(Type, Fname, Falias, Author = "", Description = "")
  assign("aliasmat",
         rbind(get("aliasmat", envir = TTLFBaseEnv),
               c(Type = Type, Name = Fname, Alias = Falias,
                 Author = Author, Description = Description)),
         envir = TTLFBaseEnv)

TTLFaliasGet =
  function(Type, Falias){
    aliasmat = get("aliasmat", envir = TTLFBaseEnv)
    matchRow = which(aliasmat[,"Type"] == Type &
      aliasmat[,"Alias"] == Falias)
    if(length(matchRow) == 1)
      aliasmat[matchRow,"Name"]
    else stop("Invalid algorithm specified for ", Type)
  }

TTLFaliasList =
  function(){
    aliasmat = get("aliasmat", envir = TTLFBaseEnv)
    Types = unique(aliasmat[,"Type"])
    for(Type in Types){
      cat("==Type: ", Type, "==\n", sep = "")
      Algos = aliasmat[aliasmat[,"Type"] == Type,,drop=FALSE]
      for(i in 1:nrow(Algos))
        cat("Name: ", Algos[i, "Name"], "\n",
            "Alias: ", Algos[i, "Alias"], "\n",
            "Author: ", Algos[i, "Author"], "\n",
            "Description: ", Algos[i, "Description"], "\n\n",
            sep = "")
    }
  }
```

Defines:
`TTLFaliasAdd`, used in chunks 9c, 12a, 16, and 18.
`TTLFaliasGet`, used in chunks 26 and 27.
Uses `TTLFBaseEnv` 7.

3 Identification

The purpose of **Identification** is to identify where in the Table the data is found and where the accompanying labels are, while ignoring any extraneous information we do not want. The output is the `IdentResult`, a list containing two elements, `rows` and `cols`, each of which is a list containing these two elements:

label - a vector of the rows or columns where the labels are found.

data - a vector of the rows or columns where the data are found.

It is intended for this procedure to involve a number of Identification algorithms that are used for a high degree of reliability and flexibility, but at this stage there is only a single Primary algorithm, supplemented by a single Auxiliary algorithm.

We separate the Identification functions into three groups.

Ident Primary contain Primary Ident algorithms, of which one is chosen when calling `TableToLongForm`.

Ident Auxiliary contain Auxiliary Ident algorithms, of which any combination, in any order, can be chosen when calling `TableToLongForm`. They are called after the Primary algorithm, to refine the `IdentResult`.

Ident Support contains supporting functions called by the Primary and Auxiliary functions.

9a ⟨*Identification* 9a⟩≡
```
⟨Ident Primary 9b⟩
⟨Ident Auxiliary 11b⟩
⟨Ident Support 12d⟩
```

3.1 Identification - Primary

The Primary Ident algorithms should take a single argument, `matFull`. They should return an `IdentResult`.

9b ⟨*Ident Primary* 9b⟩≡
```
⟨Ident by Most Common Boundary 9c⟩
```

3.1.1 Ident by Most Common Boundary

Search for the most common start and end rows and columns (the boundary) to find a block (rectangular region) of numbers, which is assumed to be our table of data.

9c ⟨*Ident by Most Common Boundary* 9c⟩≡
```
IdentbyMostCommonBoundary =
  function(matFull){
    ⟨Get Non empty rows and cols 9d⟩
    ⟨Call Ident MostCommonBoundary 10a⟩
    ⟨Construct rowslist and colslist 10b⟩
    ⟨Cleanup MostCommonBoundary Results 11a⟩
    list(rows = rowslist, cols = colslist)
  }
TTLFaliasAdd("IdentPrimary", "IdentbyMostCommonBoundary", "combound",
             "Base Algorithm", "Default IdentPrimary algorithm")
```
Uses `TTLFaliasAdd` 8.

9d ⟨*Get Non empty rows and cols* 9d⟩≡
```
rowNonempty = (1:nrow(matFull))[IdentNonEmpty(matFull, 1)]
colNonempty = (1:ncol(matFull))[IdentNonEmpty(matFull, 2)]
```
Uses `IdentNonEmpty` 13a.

10a ⟨*Call Ident MostCommonBoundary* 10a⟩≡

```
rowData = IdentMostCommonBoundary(matFull, 2)
colData = IdentMostCommonBoundary(matFull, 1)
TCRsink("CIMCB", rowData, colData)
```

Uses `IdentMostCommonBoundary` 14b and `TCRsink` 7.

Example values for **ToyExComplete.csv** `(ID: CIMCB)`

```
> rowData
[1]  5 14

> colData
[1]  4 11
```

We construct the interim `rowslist` taking every non-empty row before the most common start of the numbers block (`rowData[1]`) and assigning these to the `label` region. The numbers block (which is bounded by `rowData[1]` and `rowData[2]`) is assigned to the `data` region. The interim `colslist` is constructed in the same manner.

10b ⟨*Construct rowslist and colslist* 10b⟩≡

```
rowslist = list(label = rowNonempty[rowNonempty < rowData[1]],
                data = rowNonempty[(rowNonempty >= rowData[1]) &
                                   (rowNonempty <= rowData[2])])
colslist = list(label = colNonempty[colNonempty < colData[1]],
                data = colNonempty[(colNonempty >= colData[1]) &
                                   (colNonempty <= colData[2])])
TCRsink("CRAC", rowslist, colslist)
```

Uses `TCRsink` 7.

Example values for **ToyExComplete.csv** `(ID: CRAC)`

```
> rowslist
$label
[1] 1 2 3 4

$data
 [1]  5  6  7  8  9 10 11 12 13 14


> colslist
$label
[1] 1 2

$data
[1]  4  5  6  7  8  9 10 11
```

As the `MostCommonBoundary` algorithm searches for the data region, it can be conservative with respect to the rows and columns assigned to data. Under most circumstances this causes no problems, but in certain rare cases of mismatched column labels, there are column labels that are outside the data region (that is, the column label is not over the data it is the label of, hence mismatched). To correct for this, we do the following:

1. If `matRowLabel` isn't all empty
2. Shift any fully empty columns on the right to `cols$data`

11a ⟨*Cleanup MostCommonBoundary Results* 11a⟩≡

```
matRowLabel = matFull[rowslist$data, colslist$label,drop=FALSE]
if(!all(is.na(matRowLabel)) && ncol(matRowLabel) > 1){
  RowLabelNonempty = IdentNonEmpty(matRowLabel, 2)
  if(max(RowLabelNonempty) < ncol(matRowLabel)){
    toshift = (max(RowLabelNonempty) + 1):ncol(matRowLabel)
    colslist$data = c(colslist$label[toshift], colslist$data)
    colslist$label = colslist$label[-toshift]
  }
}
```

Uses `IdentNonEmpty` 13a.

3.2 Identification - Auxiliary

The Auxiliary Ident algorithms should take two arguments, `matFull` and `IdentResult`. They should return an `IdentResult`.

11b ⟨*Ident Auxiliary* 11b⟩≡

  ⟨*Ident by Sequence* 12a⟩

3.2.1 Ident by Sequence

Search for fully numeric row labels (e.g. Years) that were misidentified as data, by checking if the numbers follow some fixed sequence. If such a situation is found (result is not `NA`), we update `IdentResult`. This is intended to be used in conjunction with the *Ident by Most Common Boundary* Primary algorithm, which assumes numbers to be data, and not labels.

Currently the algorithm is conservative, only making the check if the current `matRowLabel` is empty (`ncol = 0`, or all `NA`s), and only accepting a sequence of fixed difference, with no gaps or jumps, e.g.

- 1 2 3 4, then a sequence
- 1 2 4 5, then not a sequence

12a ⟨*Ident by Sequence* 12a⟩≡

```
IdentbySequence =
  function(matFull, IdentResult)
  with(IdentResult, {
    matRowLabel = matFull[rows$data, cols$label]
    ⟨If empty take next column 12b⟩
    ⟨Check if sequence 12c⟩
  })
TTLFaliasAdd("IdentAuxiliary", "IdentbySequence", "sequence",
             "Base Algorithm", paste("Search for fully numeric row",
             "labels (e.g. Years) that were misidentified as data"))
```

Uses `IdentResult` 26a and `TTLFaliasAdd` 8.

12b ⟨*If empty take next column* 12b⟩≡

```
if(all(is.na(matRowLabel))){
  cols$label = cols$data[1]
  cols$data = cols$data[-1]
  IdentbySequence(matFull, list(rows = rows, cols = cols))
}
```

Check to see if all diffs are equal, but original values are not. If it is, we have a sequence and we return an updated `IdentResult`.

12c ⟨*Check if sequence* 12c⟩≡

```
else{
  matRowLabel = suppressWarnings(as.numeric(matRowLabel))
  if(length(unique(matRowLabel)) > 1 &&
     length(unique(diff(matRowLabel))) == 1)
    list(rows = rows, cols = cols)
  else IdentResult
}
```

Uses `IdentResult` 26a.

3.3 Identification - Support

Here we discuss the supporting functions called by the Primary and Auxiliary functions. Each chunk corresponds to a separate supporting function.

12d ⟨*Ident Support* 12d⟩≡

```
⟨Ident Non Empty 13a⟩
⟨Ident Pattern 13b⟩
⟨Ident Most Common Boundary 14b⟩
```

3.3.1 IdentNonEmpty

Given a matrix (`mat`) and a margin (1 for rows, 2 for columns), return a vector giving the indices of non-empty rows or columns. Can specify a different empty identifying function (default `is.na`). Procedure:

1. Compute `isnonempty`, a logical vector about whether the rows or cols are not empty.

2. Use `which` on `isnonempty` to get indices.

13a ⟨*Ident Non Empty* 13a⟩≡

```
IdentNonEmpty =
  function(mat, margin, emptyident = is.na){
    isnonempty = apply(mat, margin, function(x) !all(emptyident(x)))
    which(isnonempty)
  }
```

Defines:

`IdentNonEmpty`, used in chunks 9d, 11a, and 16a.

3.3.2 IdentPattern

Attempt to discern a repeating pattern in `vec`, which can be a vector of any type (which is coerced to `character`). The returned value is the grouping number for the repeating pattern, or the length of `vec` if there is no repeating pattern, e.g.

- `vec` = 1 1 1 1, then return 1
- `vec` = 3 4 3 4, then return 2
- `vec` = 1 2 3 4, then return 4
- `vec` = 1 2 3 1, then return 4

`IdentPattern` does this fairly efficiently by use of regular expressions and `match`.

13b ⟨*Ident Pattern* 13b⟩≡

```
IdentPattern =
  function(vec){
    ⟨Look for potential repeat 13c⟩
    ⟨Check if pattern repeats 14a⟩
  }
```

Defines:

`IdentPattern`, used in chunk 17a.

Look for when unique values of `vec` repeat, and see if the distance (`diff`) between these are equal (hence the `unique` of the `diff` result will be of length 1). If it is, we take this as our potential repeating point and move on.

If the value does not repeat at all, `diff` will return a `vector` of length 0, which is adjusted to the length of `vec`.

13c ⟨*Look for potential repeat* 13c⟩≡

```
matchvec = match(vec, unique(vec))
for(i in 1:length(unique(matchvec))){
  repind = unique(diff(which(matchvec == i)))
  if(length(repind) == 0)
    repind = length(vec)
  if(length(repind) == 1)
    break
}
```

We combine the first `repind` elements of `vec` and collapse this into a single string. A `grep` is then called on the entire `vec` that has also been collapsed into a single string, checking to see if the entire string can be matched to some repeat of the aforementioned collapsed string of the first `repind` elements. If it can, we have a repeating pattern and thus return `repind`. Otherwise, we return the length of `vec`.

14a ⟨*Check if pattern repeats* 14a⟩≡

```
curseg = paste0("^(", paste(vec[1:repind], collapse = ""), ")+$")
if(length(grep(curseg, paste(vec, collapse = ""))) > 0)
  repind else length(vec)
```

3.3.3 IdentMostCommonBoundary

Search for the most common first and last rows/cols to identify a block (rectangular region) of numbers. Procedure:

1. Suppose `margin = 2`, then loop through each column and search for cells containing numbers.
2. Compute the first row with a number for each column (`nstarts`), and do the same for the last row (`nends`).
3. Return the most common first and last rows.

14b ⟨*Ident Most Common Boundary* 14b⟩≡

```
IdentMostCommonBoundary =
  function(matFull, margin){
    isnumber = suppressWarnings(apply(matFull, margin,
      function(x) which(!is.na(as.numeric(x)))))
    nstarts = table(sapply(isnumber,
      function(x) if(length(x) > 0) min(x) else NA))
    nends = table(sapply(isnumber,
      function(x) if(length(x) > 0) max(x) else NA))
    as.numeric(names(c(which.max(nstarts), which.max(rev(nends)))))
  }
```

Defines:
`IdentMostCommonBoundary`, used in chunk 10a.

4 Discern Parentage

The purpose of **Discern Parentage** is to understand the hierarchical structure (the *parentage*) of the row and column labels. The output will be the `rowplist` and `colplist`, the row and column parentage lists. TO DO explanation of plist.

We separate the Parentage functions into five groups.

Pare Pre Row contain pre-requisite algorithms that tidy up the Row Labels for correct operation of the Main Parentage algorithm. Any combination of these algorithms, in any order, can be chosen when calling `TableToLongForm`. The current implementation of TableToLongForm has no Pre Row algorithms.

Pare Pre Col contain pre-requisite algorithms that tidy up the Column Labels for correct operation of the Main Parentage algorithm. Any combination of these algorithms, in any order, can be chosen when calling `TableToLongForm`.

Pare Front is a simple 'front-end' function that makes the appropriate first call to `PareMain`.

Pare Main contains the Main algorithm that will recursively call itself until the all parentage is discerned.

Pare Low Level contains low-level functions called by the Main function.

15a ⟨*Discern Parentage* 15a⟩≡
    ⟨*Pare Pre Row* 15b⟩
    ⟨*Pare Pre Col* 15c⟩
    ⟨*Pare Front* 19a⟩
    ⟨*Pare Main* 19b⟩
    ⟨*Pare Low Level* 21c⟩

4.1 Parentage - Pre Row

Parentage Pre Row algorithms should take two arguments, `matData` and `matRowLabel`. They should return a named list containing two elements, `matData` and `matRowLabel`.

The current implementation of TableToLongForm has no Pre Row algorithms, but has support for external modules that add Pre Row algorithms.

15b ⟨*Pare Pre Row* 15b⟩≡
```
## Empty
```

4.2 Parentage - Pre Col

Parentage Pre Col algorithms should take two arguments, `matData` and `matColLabel`. They should return a named list containing two elements, `matData` and `matColLabel`.

15c ⟨*Pare Pre Col* 15c⟩≡
    ⟨*Mismatched Col Labels* 16a⟩
    ⟨*Misaligned Col Parent* 16b⟩
    ⟨*Multirow Col Labels* 18⟩

Column Label	
	1
	2
	3

Column Label	
	1
	2
	3

Table 1: An example of mismatched column labels. The label is in a different column to the data it belongs to. The algorithm can detect this as mismatched as they have the same number of non-empty columns (1), and have empty columns in each subset (seen easily in the left table as the 2 empty cells). Such cases can occur due to some misguided attempts to visually align the label to the data (e.g. table on the right).

4.2.1 Case Mismatched Column Labels

We check for any mismatched column labels by checking if there are the same number of non-empty columns for the two subsets, and that there are empty columns in the subsets, which together imply mismatched column labels. If that is the case, we update our mat Subsets as required.

16a ⟨*Mismatched Col Labels* 16a⟩≡

```
ParePreColMismatch =
  function(matData, matColLabel){
    colsData = IdentNonEmpty(matData, 2)
    colsLabels = IdentNonEmpty(matColLabel, 2)
    if(length(colsData) == length(colsLabels))
      if(ncol(matData) != length(colsData)){
        matColLabel = matColLabel[,colsLabels,drop=FALSE]
        matData = matData[,colsData,drop=FALSE]
      }
    list(matData = matData, matColLabel = matColLabel)
  }
TTLFaliasAdd("ParePreCol", "ParePreColMismatch", "mismatch",
             "Base Algorithm", paste("Correct for column labels",
             "not matched correctly over data (label in a",
             "different column to data)"))
```

Uses `IdentNonEmpty` 13a and `TTLFaliasAdd` 8.

4.2.2 Case Misaligned Column Parents

We correct for any misaligned column parents by using pattern matching to detect parent-groupings, and then realigning the parents.

16b ⟨*Misaligned Col Parent* 16b⟩≡

```
ParePreColMisaligned =
  function(matData, matColLabel){
    TCRsink("MCPBefore", matColLabel)
    for(i in 1:nrow(matColLabel)){
      currow = matColLabel[i,]
      ⟨Search for Pattern 17a⟩
      ⟨Align Column Parents 17b⟩
    }
    TCRsink("MCPAfter", matColLabel)
    list(matData = matData, matColLabel = matColLabel)
  }
TTLFaliasAdd("ParePreCol", "ParePreColMisaligned", "misalign",
             "Base Algorithm", paste("Correct for column labels",
             "not aligned correctly over data (parents not",
             "positioned on the far-left, relative to their",
             "children in the row below)"))
```

Uses `TCRsink` 7 and `TTLFaliasAdd` 8.

Example values for **ToyExComplete.csv** (ID: MCPBefore)

```
> matColLabel
     V4       V5             V6       V7       V8       V9
[1,] NA       NA             NA       NA       NA       NA
[2,] NA       "Col Parent1"  NA       NA       NA       "Col Parent2"
[3,] "Col"    "Col"          "Col"    "Col"    "Col"    "Col"
[4,] "Child1" "Child2"       "Child3" "Child4" "Child1" "Child2"
```

Example values for **ToyExComplete.csv** (ID: MCPAfter)

```
> matColLabel
     V4            V5       V6       V7       V8            V9
[1,] NA            NA       NA       NA       NA            NA
[2,] "Col Parent1" NA       NA       NA       "Col Parent2" NA
[3,] "Col"         "Col"    "Col"    "Col"    "Col"         "Col"
[4,] "Child1"      "Child2" "Child3" "Child4" "Child1"      "Child2"
```

	Column Parent1			Column Parent2	
Child1	Child2	Child3	Child1	Child2	Child3

Table 2: An example of misaligned column parents. For our low-level Parentage algorithm to work, we want the Column Parents to be in the left-most cell of their parent-grouping.

The value of `curPattern` will be the following:

- If completely empty (all `NA`), return `NA`.
- If any empty, check pattern of emptiness. In the above Table row 1, this will find the pattern: NonEmpty-Empty-NonEmpty which occurs twice. Hence return 2.
- Else, all cells are non-empty, check pattern of contents. In the above Table row 2, this will find the pattern: Child1-Child2-Child3 which occurs twice. Hence return 2.

17a ⟨*Search for Pattern* 17a⟩≡

```
curPattern =
  if(all(is.na(currow))) NA
  else if(any(is.na(currow))) IdentPattern(is.na(currow))
  else IdentPattern(currow)
```

Uses `IdentPattern` 13b.

For each subset of the row (based on pattern), move any empty cells (`NA`) to the end, hence aligning the non-empty cell (the parent) to the left.

17b ⟨*Align Column Parents* 17b⟩≡

```
if(!is.na(curPattern)){
  nParents = length(currow)/curPattern
  for(j in 1:nParents){
    curcols = 1:curPattern + curPattern * (j - 1)
    cursub = currow[curcols]
    currow[curcols] = c(cursub[!is.na(cursub)], cursub[is.na(cursub)])
    TCRsink("ACP", cursub, currow[curcols])
  }
  matColLabel[i,] = currow
}
```

Uses `TCRsink` 7.

Example values for **ToyExComplete.csv** (ID: ACP)

```
> cursub
          V4            V5            V6            V7
          NA "Col Parent1"            NA            NA

> currow[curcols]
          V4            V5            V6            V7
"Col Parent1"           NA            NA            NA
```

4.2.3 Case Multi-row Column Labels

It is also quite common for Col Labels that are too wide to be physically split over multiple rows to manage the width of the labels. For now, we simply assume that any rows that are not full (and hence not parents) should all really be a single row of children, and collapse these.

18 ⟨*Multirow Col Labels* 18⟩≡

```
ParePreColMultirow =
  function(matData, matColLabel){
    fullrows = apply(matColLabel, 1, function(x) all(!is.na(x)))
    if(any(diff(fullrows) > 1))
      warning("full rows followed by not full rows!")
    if(any(fullrows)){
      pastestring = ""
      pasterows = which(fullrows)
      for(i in 1:length(pasterows))
        pastestring[i] = paste0("matColLabel[", pasterows[i],
                     ",,drop=FALSE]")
      collapsedlabels =
        eval(parse(text = paste0("paste(",
                     paste(pastestring, collapse = ", "), ")")))

      TCRsink("MCLBefore", matColLabel)
      matColLabel = rbind(matColLabel[!fullrows,,drop=FALSE],
        collapsedlabels, deparse.level = 0)
      TCRsink("MCLAfter", matColLabel)
    }
    list(matData = matData, matColLabel = matColLabel)
  }
TTLFaliasAdd("ParePreCol", "ParePreColMultirow", "multirow",
             "Base Algorithm", paste("Merge long column labels",
             "that were physically split over multiple rows",
             "back into a single label"))
```

Uses `TCRsink` 7 and `TTLFaliasAdd` 8.

Example values for **ToyExComplete.csv** (ID: MCLBefore)

```
> matColLabel
     V4             V5       V6       V7       V8             V9
[1,] NA             NA       NA       NA       NA             NA
[2,] "Col Parent1" NA       NA       NA       "Col Parent2" NA
[3,] "Col"          "Col"    "Col"    "Col"    "Col"          "Col"
[4,] "Child1"       "Child2" "Child3" "Child4" "Child1"       "Child2"
```

Example values for **ToyExComplete.csv** (ID: MCLAfter)

```
> matColLabel
     V4            V5           V6           V7           V8
[1,] NA            NA           NA           NA           NA
[2,] "Col Parent1" NA           NA           NA           "Col Parent2"
[3,] "Col Child1"  "Col Child2" "Col Child3" "Col Child4" "Col Child1"
     V9
[1,] NA
[2,] NA
[3,] "Col Child2"
```

4.3 Parentage - Front

This front end function takes the `matLabel`, which can be the `matRowLabel` or the transpose of the `matColLabel`, and constructs an initialising `plist`, which is used to make the first call to the Main function.

19a ⟨*Pare Front* 19a⟩≡

```
ParePront =
  function(matLabel)
  PareMain(matSub = matLabel, plist =
           list(rows = 1:nrow(matLabel), cols = 1:ncol(matLabel)))
```

Defines:
`PareFront`, used in chunks 26b and 27.
Uses `PareMain` 19b.

4.4 Parentage - Main

The purpose of the `PareMain` function is to identify (or *Discern*, to better differentiate this stage from the *Identification* stage) hierarchical relationships (the *Parentage*) in the data.

It first makes various checks for fringe cases, then calls various detection algorithms (`Pare Low Levels`) to discern the parentage.

19b ⟨*Pare Main* 19b⟩≡

```
PareMain =
  function(matSub, plist){
    ⟨If only one column 20a⟩
    ⟨If first column empty 20b⟩
    ⟨If only one row 20c⟩
    ⟨If first cell empty 21a⟩
    ⟨Otherwise call Pare Low Levels 21b⟩
    class(res) = "plist"
    res
  }
```

Defines:
`PareMain`, used in chunks 19–21.

If only one column is found then this means we are in the right-most column (or there was only one column to begin with), and hence the currently examined cells cannot be parents. We return the rows of these children as a vector, with names that correspond to their labels.

20a ⟨*If only one column* 20a⟩≡
```
if(length(plist$cols) == 1){
  res = structure(plist$rows, .Names = matSub[plist$rows, plist$cols])
  res = attrLoc(res, cols = plist$col)
  TCRsink("IOOC", plist, res)
}
```
Uses `attrLoc` 6b and `TCRsink` 7.

Example values for **ToyExComplete.csv** `(ID: IOOC)`

```
> plist
$rows
[1] 3 4

$cols
[1] 2


> res
Row Child-Child1 Row Child-Child2
               3                4
```

If the first column is found to be empty, then we will shift to the next column (which we know exists because we passed the check for only one column).

20b ⟨*If first column empty* 20b⟩≡
```
else if(all(is.na(matSub[plist$rows, plist$cols[1]]))){
  plist$cols = plist$cols[-1]
  res = PareMain(matSub, plist)
}
```
Uses `PareMain` 19b.

If only one row is found then our row is a parent to itself (we know there are children in the row as we passed the check for only one column). We return the row as a numeric vector, nested in a list using correct parentage and names of the parentage within the row.

20c ⟨*If only one row* 20c⟩≡
```
else if(length(plist$rows) == 1){
  res = structure(plist$rows,
    .Names = matSub[plist$rows, plist$cols[length(plist$cols)]])
  res = attrLoc(res, cols = plist$cols[length(plist$cols)])
  for(i in (length(plist$cols) - 1):1){
    res = list(res)
    names(res) = matSub[plist$rows, plist$cols[i]]
    res = attrLoc(res, rows = plist$rows, cols = plist$cols[i])
  }
  TCRsink("IOOR", plist, res)
}
```
Uses `attrLoc` 6b and `TCRsink` 7.

Example values for **ToyExComplete.csv** `(ID: IOOR)`

```
> res
Never occurs
```

If the first cell is empty, after all previous checks, then this is an unrecognised format and we return a warning message.

21a ⟨*If first cell empty* 21a⟩≡
```
else if(is.na(matSub[plist$rows[1], plist$cols[1]])){
  warning("cell[1, 1] is empty")
  print(plist)
  res = NA
}
```

If we have passed all the checks, we can then call the Low Level `Pare` functions. We first call `ByEmptyRight` to check for *empty right* situations. If none are found, it returns `NA`, in which case we try `ByEmptyBelow` instead.

We then loop through each element of the returned list and call the main function, as per the recursive nature of the function.

21b ⟨*Otherwise call Pare Low Levels* 21b⟩≡
```
else{
  res = PareByEmptyRight(matSub, plist)
  if(any(is.na(res)))
    res = PareByEmptyBelow(matSub, plist)
  for(i in 1:length(res))
    res[[i]] = PareMain(matSub, res[[i]])
  res
}
```
Uses `ParePyEmptyBelow` 24, `PareByEmptyRight` 21d, and `PareMain` 19b.

4.5 Parentage - Low Level Functions

The Low Level Parentage functions are called by the Main Parentage function. In particular, `ByEmptyRight` is always called first. Then `ByEmptyBelow` is called on the results of the above.

21c ⟨*Pare Low Level* 21c⟩≡
  ⟨*Pare By Empty Right* 21d⟩
  ⟨*Pare By Empty Below* 24⟩

4.5.1 Pare By Empty Right

We check to see if we have an *empty right* situation. If we do not, we return `NA`.

21d ⟨*Pare By Empty Right* 21d⟩≡
```
ParePyEmptyRight =
  function(matSub, plist)
  with(plist,
       if(all(is.na(matSub[rows[1], cols[-1]]))){
         ⟨Check for Other Empty Rights 21e⟩
         ⟨Case Single Empty Right 22⟩
         ⟨Case Multiple Empty Rights 23⟩
         res
       } else NA)
```
Defines:
  `PareByEmptyRight`, used in chunk 21b.

21e ⟨*Check for Other Empty Rights* 21e⟩≡
```
emptyrights = apply(matSub[rows, cols[-1],drop=FALSE], 1,
  function(x) all(is.na(x)))
rowemptyright = rows[emptyrights]
```

1	***New Zealand***	
2	**Auckland**	
3	Accounting	Male
4		Female
5	Economics	Male
6		Female
7	Statistics	Male
8		Female
9	**Wellington**	
10	Economics	Male
11		Female
12	Statistics	Male
13		Female
14	***Australia***	
15	**Sydney**	
16	Accounting	Male
17		Female
18	Economics	Male
19		Female

Consider the toy example on the left.
In this case we do not have a simple `ByEmptyRight` structure. We have *super-parents* in the form of countries (New Zealand and Australia), and also *parents* in the form of cities (Auckland, Wellington and Sydney). To handle situations such as this, we must **Check for Other Empty Rights**.
If only a **Single Empty Right** is found, the situation is simple and we simply pass on the children of the single parent for the next iteration of `PareMain`.
However, if **Multiple Empty Rights** are found, we must identify the super-parents, and pass on the *children* of these super-parents (which would, in turn, contain parents and their children) as a list, to be handled in the next iteration of `PareMain`. In this example, we would have a list of length 2. The first element of the list would contain the `plist` with `rows` 2 to 13 (corresponding to the children of the New Zealand super-parent). The second element would have `rows` 15 to 19.

In the case of only a single empty right, we know there is only a single parent, which is the first line. Thus we take everything except the first line (which will be the rows of the children of this parent) and pass this through with correct naming.

22 ⟨*Case Single Empty Right* 22⟩≡

```
if(length(rowemptyright) == 1){
  res = list(list(rows = rows[-1], cols = cols))
  names(res) = matSub[rows[1], cols[1]]
  res = attrLoc(res, rows = rows[1], cols = cols[1])
  TCRsink("CSER", res)
}
```

Uses `attrLoc` 6b and `TCRsink` 7.

Example values for **ToyExComplete.csv** `(ID: CSER)`

```
> res
Never occurs
```

In the case of multiple empty rights, we first call `diff` to compute the gap in rows between the empty rights. If the value of `rowdiff[i]` is 1, this means there is no gap between the i^{th} `rowemptyright` and the $(i + 1)$ `rowemptyright`. This happens with *super-parents* as described in the example above. In this case, we gather these super-parents and ignore all other `rowemptyright` (the parents inside the super-parents will be handled at the next iteration of `PareMain`). Note, we assume there are never any super-super-parents (i.e. we can only handle a maximum of 2-levels of parentage in the same column).

Whether or not super-parents were identified, we compute the rows for the children of each parent (or super-parent) identified by `rowemptyright` and pass this through as a list, with correct naming.

23 ⟨*Case Multiple Empty Rights* 23⟩≡

```
else{
  rowdiff = diff(rowemptyright)
  if(any(rowdiff == 1))
    rowemptyright = rowemptyright[c(rowdiff == 1, FALSE)]

  rowstart = pmin(rowemptyright + 1, max(rows))
  rowend = c(pmax(rowemptyright[-1] - 1, min(rows)), max(rows))

  res = list()
  for(i in 1:length(rowstart))
    res[i] = list(list(rows = rowstart[i]:rowend[i], cols = cols))
  names(res) = matSub[rowemptyright, cols[1]]
  res = attrLoc(res, rows = rowemptyright, cols = cols[1])
  TCRsink("CMER", res)
}
```

Uses `attrLoc` 6b and `TCRsink` 7.

Example values for **ToyExComplete.csv** `(ID: CMER)`

```
> res
$‘Row Super-Parent‘
$‘Row Super-Parent‘$rows
[1]  2  3  4  5  6  7  8  9 10

$‘Row Super-Parent‘$cols
[1] 1 2
```

4.5.2 Pare By Empty Below

We check which cells are empty below (there should be at least 1 based on previous checks). Based on this, we compute the rows for the children of each parent and pass this through as a list, with correct naming.

24 ⟨*Pare By Empty Below* 24⟩≡

```
PareByEmptyBelow =
  function(matSub, plist)
  with(plist, {
    emptybelow = is.na(matSub[rows, cols[1]])
    rowstart = rows[!emptybelow]
    rowend = c(rowstart[-1] - 1, max(rows))
    res = list()
    for(i in 1:length(rowstart))
      res[i] = list(list(rows = rowstart[i]:rowend[i], cols = cols[-1]))
    names(res) = matSub[rowstart, cols[1]]
    res = attrLoc(res, rows = rowstart, cols = cols[1])
    TCRsink("PBEB", res)
    res
  })
```

Defines:
`PareByEmptyBelow`, used in chunk 21b.
Uses `attrLoc` 6b and `TCRsink` 7.

Example values for **ToyExComplete.csv** `(ID: PBEB)`

```
> res
$‘Row Child1‘
$‘Row Child1‘$rows
[1] 3 4

$‘Row Child1‘$cols
[1] 2


$‘Row Child2‘
$‘Row Child2‘$rows
[1] 5 6

$‘Row Child2‘$cols
[1] 2
```

---

5 Reconstruction

We separate the Reconstruction functions into two groups.

Recons Main contains the main function that is called by the *Front End* function.

Recons Low Level contains supporting functions called by the *Recons Main* function.

25a ⟨*Reconstruction* 25a⟩≡
    ⟨*Recons Main* 25b⟩
    ⟨*Recons Low Level* 28⟩

5.1 Reconstruction - Main Function

The `ReconsMain` function is, in a manner of speaking, the true `TableToLongForm` function, as it makes the calls to `IdentMain` and `PareFront`, in conjunction with its own `Recons Low Level` functions, to carry out the conversion.

25b ⟨*Recons Main* 25b⟩≡

```
ReconsMain =
  function(matFull, IdentResult,
           IdentPrimary, IdentAuxiliary,
           ParePreRow, ParePreCol){
    ⟨Call Ident Algos 26a⟩
    ⟨Reconstruct Row Labels 26b⟩
    ⟨Reconstruct Col Labels 27⟩
  }
```

Defines:
    `ReconsMain`, used in chunk 4a.
Uses `IdentResult` 26a.

If a custom `IdentResult` is given, we use that. Otherwise (`IdentResult == NULL`), we call the Ident algorithms as specified by the arguments, `IdentPrimary` and `IdentAuxiliary`. Only 1 `IdentPrimary` is accepted, while any number of `IdentAuxiliary` algorithms can be specified, which will be called in the order they are given.

26a ⟨*Call Ident Algos* 26a⟩≡

```
if(is.null(IdentResult)){
  IdentPrimary = TTLFaliasGet("IdentPrimary", IdentPrimary)
  IdentResult = do.call(IdentPrimary, list(matFull = matFull))
  if(!is.null(IdentAuxiliary))
    for(AuxAlgo in IdentAuxiliary){
      AuxAlgo = TTLFaliasGet("IdentAuxiliary", AuxAlgo)
      IdentResult = do.call(AuxAlgo,
        list(matFull = matFull, IdentResult = IdentResult))
    }
}
```

Defines:
`IdentResult`, used in chunks 4a, 12, and 25–27.
Uses `TTLFaliasGet` 8.

We create the subsets of `matFull` using `IdentResult`:

matData Which should contain just the Data.

matRowLabel Which should contain just the Row Labels.

We then call the `ParePreRow` algorithms in the order given (assuming there are any), to tidy up `matData` (rarely) and `matRowLabel` (primarily), before calling `PareFront` to discern the parentage of the Row Labels.

We then use this to reconstruct the portion of the LongForm Dataframe relating to the Row Labels and assign this to `rowvecs`.

26b ⟨*Reconstruct Row Labels* 26b⟩≡

```
matData = with(IdentResult,
  matFull[rows$data, cols$data,drop=FALSE])
matRowLabel = with(IdentResult,
  matFull[rows$data, cols$label,drop=FALSE])
if(!is.null(ParePreRow))
  for(PreAlgo in ParePreRow){
    PreAlgo = TTLFaliasGet("ParePreRow", PreAlgo)
    PreOut = do.call(PreAlgo,
      list(matData = matData, matRowLabel = matRowLabel))
    matData = PreOut$matData
    matRowLabel = PreOut$matRowLabel
  }
rowplist = PareFront(matRowLabel)
rowvecs = ReconsRowLabels(rowplist)
TCRsink("RRL", rowplist, rowvecs)
```

Defines:
`rowplist`, used in chunk 27.
`rowvecs`, used in chunks 27, 29, and 30.
Uses `IdentResult` 26a, `PareFront` 19a, `ReconsRowLabels` 29a, `TCRsink` 7, and `TTLFaliasGet` 8.

Example values for **ToyExComplete.csv** `(ID: RRL)`

```
> rowplist
$`Row Super-Parent`
+ Row Parent1 (2, 1)
- + Row Child1 (3, 1)
- - + Row Child-Child1 (3, 2)
- - + Row Child-Child2 (4, 2)
- + Row Child2 (5, 1)
- - + Row Child-Child1 (5, 2)
- - + Row Child-Child2 (6, 2)
+ Row Parent2 (7, 1)
- + Row Child1 (8, 1)
- - + Row Child-Child1 (8, 2)
- - + Row Child-Child2 (9, 2)
- + Row Child2 (10, 1)
- - + Row Child-Child2 (10, 2)


> rowvecs
 [,1]               [,2]          [,3]         [,4]
 "Row Super-Parent" "Row Parent1" "Row Child1" "Row Child-Child1"
 "Row Super-Parent" "Row Parent1" "Row Child1" "Row Child-Child2"
 "Row Super-Parent" "Row Parent1" "Row Child2" "Row Child-Child1"
 "Row Super-Parent" "Row Parent1" "Row Child2" "Row Child-Child2"
 "Row Super-Parent" "Row Parent2" "Row Child1" "Row Child-Child1"
 "Row Super-Parent" "Row Parent2" "Row Child1" "Row Child-Child2"
```

We create a further subset of `matFull` using `IdentResult`:

matColLabel Which should contain just the Column Labels.

We then call the `ParePreCol` algorithms in the order given (assuming there are any), to tidy up `matData` (rarely) and `matColLabel` (primarily), before calling `PareFront` on the transpose of `matColLabel` (as the Main Parentage algorithm is written to work for Row Labels) to discern the parentage of the Column Labels.

We then call `ReconsColLabels` which in truth reconstructs the entire LongForm Dataframe by making use of the `rowvecs` generated above.

We finally return the full output back to the main `TableToLongForm` function.

27 ⟨*Reconstruct Col Labels* 27⟩≡

```
matColLabel = with(IdentResult,
  matFull[rows$label, cols$data,drop=FALSE])
if(!is.null(ParePreCol))
  for(PreAlgo in ParePreCol){
    PreAlgo = TTLFaliasGet("ParePreCol", PreAlgo)
    PreOut = do.call(PreAlgo,
      list(matData = matData, matColLabel = matColLabel))
    matData = PreOut$matData
    matColLabel = PreOut$matColLabel
  }
colplist = PareFront(t(matColLabel))
matDataReduced = matData[unlist(rowplist),,drop=FALSE]
res = ReconsColLabels(colplist, matDataReduced, rowvecs)
TCRsink("RCL", colplist, res)
list(datafr = res, oriTable = matFull, IdentResult = IdentResult,
     rowplist = rowplist, colplist = colplist)
```

Uses `IdentResult` 26a, `PareFront` 19a, `ReconsColLabels` 29b, `rowplist` 26b, `rowvecs` 26b, `TCRsink` 7, and `TTLFaliasGet` 8.

Example values for **ToyExComplete.csv** `(ID: RCL)`

```
> colplist
$`Col Parent1`
+ Col Child1 (1, 3)
+ Col Child2 (2, 3)
+ Col Child3 (3, 3)
+ Col Child4 (4, 3)

$`Col Parent2`
+ Col Child1 (5, 3)
+ Col Child2 (6, 3)
+ Col Child3 (7, 3)
+ Col Child4 (8, 3)


> res
      UNKNOWN           UNKNOWN     UNKNOWN    UNKNOWN           UNKNOWN
1 Col Parent1 Row Super-Parent Row Parent1 Row Child1 Row Child-Child1
2 Col Parent1 Row Super-Parent Row Parent1 Row Child1 Row Child-Child2
3 Col Parent1 Row Super-Parent Row Parent1 Row Child2 Row Child-Child1
4 Col Parent1 Row Super-Parent Row Parent1 Row Child2 Row Child-Child2
5 Col Parent1 Row Super-Parent Row Parent2 Row Child1 Row Child-Child1
6 Col Parent1 Row Super-Parent Row Parent2 Row Child1 Row Child-Child2
  Col Child1 Col Child2 Col Child3 Col Child4
1         12         22         32         42
2         13         23         33         43
3         14         24         34         44
4         15         25         35         45
5         17         27         37         47
6         18         28         38         48
```

5.2 Reconstruction - Low Level Functions

The Low Level Reconstruction functions are called by the Main Reconstruction function. In particular, `ReconsRowLabels` is always called first and its results are one of the arguments for `ReconsColLabels`, which finishes the reconstruction of the entire LongForm Dataframe.

28 ⟨*Recons Low Level* 28⟩≡
  ⟨*Recons Row Labels* 29a⟩
  ⟨*Recons Column Labels* 29b⟩

5.2.1 Reconstruction - Row Labels

`ReconsRowLabels` iterates down the row parentage list (`plist`) recursively, extracting the names and using this to construct the columns of the finished LongForm Dataframe corresponding to the row labels. The final output is what was shown in the *Reconstruct Row Labels* chunk above as `rowvecs`.

29a ⟨*Recons Row Labels* 29a⟩≡

```
ReconsRowLabels =
  function(plist)
  if(is.list(plist)){
    rowvecs = as.list(names(plist))
    for(i in 1:length(rowvecs))
      rowvecs[[i]] = cbind(rowvecs[[i]], ReconsRowLabels(plist[[i]]))
    do.call(rbind, rowvecs)
  } else as.matrix(names(plist))
```

Defines:
`ReconsRowLabels`, used in chunk 26b.
Uses `rowvecs` 26b.

Example values for **ToyExComplete.csv** `(ID: RRL)`

```
> rowvecs
 [,1]               [,2]          [,3]         [,4]
 "Row Super-Parent" "Row Parent1" "Row Child1" "Row Child-Child1"
 "Row Super-Parent" "Row Parent1" "Row Child1" "Row Child-Child2"
 "Row Super-Parent" "Row Parent1" "Row Child2" "Row Child-Child1"
 "Row Super-Parent" "Row Parent1" "Row Child2" "Row Child-Child2"
 "Row Super-Parent" "Row Parent2" "Row Child1" "Row Child-Child1"
 "Row Super-Parent" "Row Parent2" "Row Child1" "Row Child-Child2"
```

5.2.2 Reconstruction - Column Labels

As with the row labels, `ReconsColLabels` iterates down the column parentage list (`plist`) recursively. We also need to handle the parents differently from the lowest level child. The final output is what was shown in the *Reconstruct Col Labels* chunk above as `res`.

29b ⟨*Recons Column Labels* 29b⟩≡

```
ReconsColLabels =
  function(plist, matData, rowvecs){
    ⟨Recons Col Parents 30a⟩
    ⟨Recons Col Children 30b⟩
    datfr
  }
```

Defines:
`ReconsColLabels`, used in chunks 27 and 30a.
Uses `rowvecs` 26b.

Example values for **ToyExComplete.csv** `(ID: RCL)`

```
> res
     UNKNOWN          UNKNOWN     UNKNOWN    UNKNOWN          UNKNOWN
1 Col Parent1 Row Super-Parent Row Parent1 Row Child1 Row Child-Child1
2 Col Parent1 Row Super-Parent Row Parent1 Row Child1 Row Child-Child2
3 Col Parent1 Row Super-Parent Row Parent1 Row Child2 Row Child-Child1
4 Col Parent1 Row Super-Parent Row Parent1 Row Child2 Row Child-Child2
5 Col Parent1 Row Super-Parent Row Parent2 Row Child1 Row Child-Child1
6 Col Parent1 Row Super-Parent Row Parent2 Row Child1 Row Child-Child2
  Col Child1 Col Child2 Col Child3 Col Child4
1         12         22         32         42
2         13         23         33         43
3         14         24         34         44
4         15         25         35         45
5         17         27         37         47
6         18         28         38         48
```

Any parents are used to construct additional columns of factors (the labels of the parents) for the LongForm Dataframe, which is attached to the portion previously constructed in `ReconsRowLabels`.

30a ⟨*Recons Col Parents* 30a⟩≡

```
if(is.list(plist)){
  colvecs = as.list(names(plist))
  for(i in 1:length(colvecs)){
    colvecs[[i]] = cbind(colvecs[[i]],
              ReconsColLabels(plist[[i]], matData, rowvecs))
    colnames(colvecs[[i]])[1] = "UNKNOWN"
  }
  datfr = do.call(rbinddf, colvecs)
}
```

Uses `rbinddf` 5c, `ReconsColLabels` 29b, and `rowvecs` 26b.

For the lowest level child, we extract the relevant ‘data bits’ from the original table and bind it to our Dataframe, using the lowest level child as the labels of these columns of data values.

30b ⟨*Recons Col Children* 30b⟩≡

```
else{
  datbit = matData[,plist,drop=FALSE]
  TCRsink("RCC", plist, matData, datbit)
  datlist = NULL
  for(j in 1:ncol(datbit)){
    asnumer = suppressWarnings(as.numeric(datbit[,j]))
    if(all(is.na(datbit[,j])) || !all(is.na(asnumer)))
      datlist[[j]] = asnumer
    else
      datlist[[j]] = datbit[,j]
  }
  datbit = do.call(cbind, datlist)
  ## Specify row.names to avoid annoying warnings
  datfr =
    cbind(as.data.frame(rowvecs, row.names = 1:nrow(rowvecs)), datbit)
  colnames(datfr) =
    c(rep("UNKNOWN", length = ncol(rowvecs)), names(plist))
}
```

Uses `rowvecs` 26b and `TCRsink` 7.

6 Chunk Index

7 Identifier Index

Numbers indicate the chunks in which the function appears. Underline indicates the chunk where the function is defined.

References

R Core Team, 2013. R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria.
URL `http://www.R-project.org/`

Ramsey, N., Sept 1994. Literate programming simplified. IEEE Software 11 (5), 97–105.
URL `http://www.cs.tufts.edu/~nr/noweb/`