MetaR: simple, high-level languages for data analysis with the R ecosystem

Fabien Campagne[1,2,3,*], William ER Digan[1,2], and Manuele Simi[1,2]

[1]The HRH Prince Alwaleed Bin Talal Bin Abdulaziz Alsaud Institute for Computational Biomedicine, Weill Cornell Medicine, New York, NY, United States of America
[2]Clinical Translational Science Center, Weill Cornell Medicine, New York, NY, United States of America
[3]Department of Physiology and Biophysics, Weill Cornell Medicine, New York, NY, United States of America
[*]To whom correspondence should be addressed: fac2003@campagnelab.org

ABSTRACT

Data analysis tools have become essential to the study of biology. Tools available today were constructed with layers of technology developed over decades. Here, we explain how some of the principles used to develop this technology are sub-optimal for the construction of data analysis tools for biologists. In contrast, we applied language workbench technology (LWT) to create a data analysis language, called MetaR, tailored for biologists with no programming experience, as well as expert bioinformaticians and statisticians. A key novelty of this approach is its ability to blend user interface with scripting in such a way that beginners and experts alike can analyze data productively in the same analysis platform. While presenting MetaR, we explain how a judicious use of LWT eliminates problems that have historically contributed to data analysis bottlenecks. These results show that language design with LWT can be a compelling approach for developing intelligent data analysis tools.

Keywords: Data analysis, Bioinformatics, Language Workbench Technology, Jetbrains MPS, Composable R, R language

The modern tools of biology often require biologists to rely on software tools for data analysis. For instance, software tools are required for analysis of high-throughput data, for the study of genome-wide gene expression, genetic or epigenetic. Similarly, most fields of biology require specialized software tools for analysis of microscopy, crystallography or other data. Most analysis software is constructed in a very similar manner: writing a program as a collection of text source code that is compiled into one or more executable analysis tools. Despite the evolution of programming languages, encoding programs as text has been a constant since the invention of the first high-level programming language (FORTRAN Backus (1958, 1978)).

In this manuscript, we discuss several drawbacks of encoding programs as text that we believe contribute to common challenges encountered by data analysts. Language Workbenches (LWs) with projectional editors offer an alternative to storing source code as text. These approaches were conceived in the 90s Simonyi (1995) and have since led to the development of robust software development environments Dmitriev (2004); Erdweg et al. (2013). For this study, we used the Meta-Programming System (MPS, `http://jetbrains.com/mps`), a robust and open-source LW to explore whether LW technology (WLT) can help develop improved data analysis tools.

One question we were particularly interested in testing was whether we could create an analysis tool that would blend the boundary between programming/scripting languages and graphical user interfaces. Programming languages such as the R language Ihaka and Gentleman (1996) are frequently preferred for data analysis by experts. They have so far been the most flexible and powerful tools for data analysis, but require a steep learning curve. In contrast, beginners tend to prefer data analysis software with a graphical user interface, which are easier to learn, but eventually are found to lack flexibility and extensibility. We reasoned that blending these two types of interfaces into one tool could provide a simpler way for beginners to learn elements of scripting, improve repeatability and reproducibility of their analyses, and increase their productivity.

We found that LWT made it straightforward to develop a data analysis tool that blends the distinction between graphical user interface and scripting. If implementation was straightforward, our design of a novel type of analysis tool was an iterative process that benefited from frequent feedback from users of the tool. In this manuscript, we describe the goals of the language, explain how the tool can be used, and highlight the most innovative aspects of the language compared to other tools used for data analysis, such as the R language Ihaka and Gentleman (1996) or electronic notebooks.

The initial focus of MetaR was on analysis of RNA-Seq data and the creation of heatmaps, but the tool is general and can be readily extended to support a broad range of data analyses. For instance, we have used MetaR to analyze data in a study of association between the allogenomics score and kidney graft function Mesnard et al. (2015). We chose to focus on the construction of heatmaps as a use case and illustration for this study because this activity is of interest to many biologists who obtain high-throughput data.

Interestingly, we found that both beginners and experts can benefit from blending user interfaces and scripting. Beginners benefit because the MetaR user interface is much simpler to learn than the full R programming language. Expert users benefit because they can develop high-level language elements to simplify repetitive aspects of data analysis in ways that text-based programming languages cannot achieve.

LANGUAGE WORKBENCH TECHNOLOGY PRIMER

Since many readers may not be familiar with LWT, this section briefly describes how this technology differs from traditional text-based technology.

Text-based programming languages are implemented with compilers that internally convert the text representation of the source code into an *abstract syntax tree* (AST), a data structure used when analyzing and transforming programming languages into machine code.

In the MPS LW, the AST is also a central data structure, but the parsing elements of the compilers are replaced with a graphical user interface (called a *projectional editor*) that enables users to directly edit the data structure. Where text-based languages are restricted to programs written as text, a projectional editor can support both textual and graphical user interfaces (such as images, buttons, tables or diagrams) Voelter and Solomatov (2010). Projectional editors can also offer distinct views of the same AST, implemented as alternative editors. Projectional editors keep an AST in memory until the user saves the program. Saving an AST to disk is done using serialization (loading is conversely done via deserialization to memory AST data structures).

The choice of serialization rather than encoding with text has a profound consequence. Serialization uniquely identifies the concept for each node in an AST. This method makes it possible to combine AST fragments expressed with different languages, when the concept hierarchy of the languages supports composition. We have presented examples of language composition in Simi and Campagne (2014); Benson and Campagne (2015). In this manuscript, we extensively use language composition to extend the R language and provide the ability to embed user interfaces into R programs.

Abstract Syntax Tree (AST)

An AST is a data structure traditionally used by compilers as a step towards generating machine code. In the MPS Language Workbench, an AST is a tree data structure, where nodes of the tree are instances of concepts (in the object-oriented sense). Figure 1 illustrates the notion of AST nodes, concepts and projectional editor.

AST concepts may have properties (values of primitive types), children (lists of other nodes they contain), references (links to other nodes defined elsewhere in the AST). An AST has always a root node, which is used to start traversing the tree. In the MPS LW, AST root nodes are stored in models.

Languages

In the MPS LW, languages are defined as collection of concepts, concept editors (which together implement the user interface for the language), and other language aspects Campagne (2014). Each language has a name which is used to import, or activate, the language inside a model. After importing a language into a model, it becomes possible to create ASTs with this language in the model. Creating an AST starts with the creation of a root node. Children of the root node are added using the projectional editor. Children of root nodes, properties and references can be edited interactively in the editor.

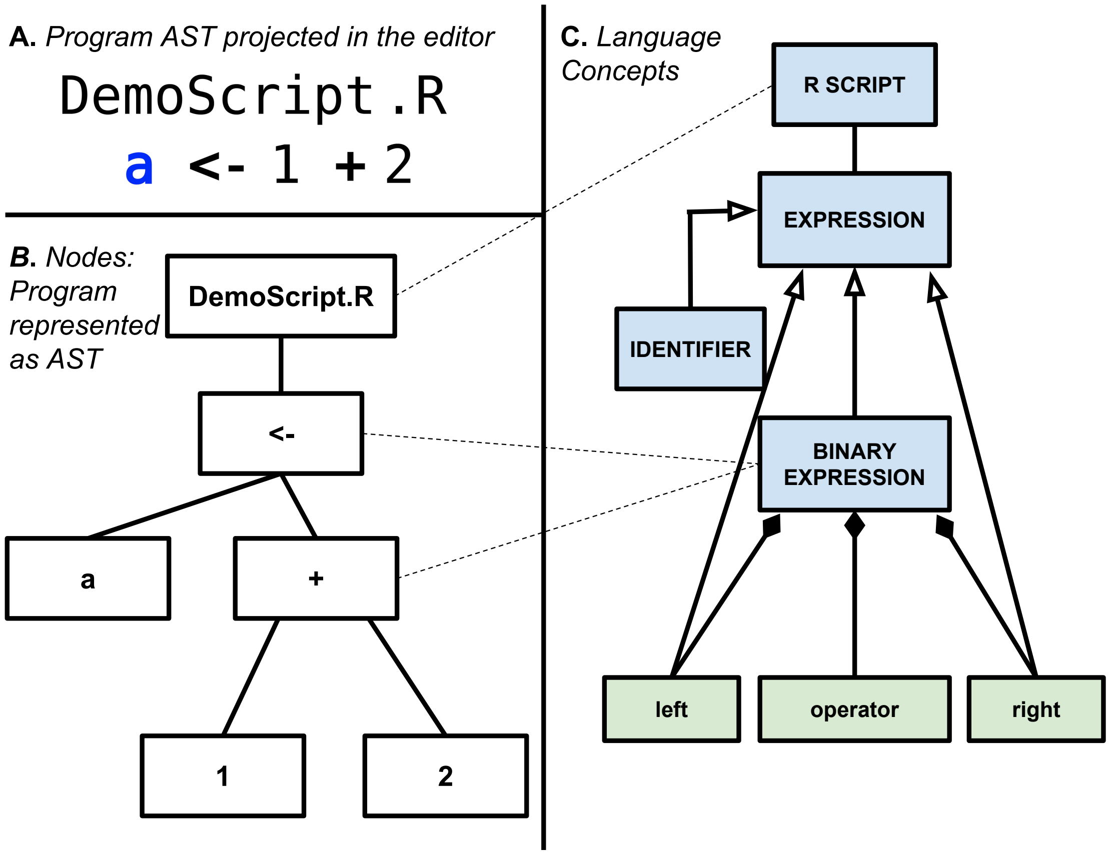


Figure 1. Concepts, Nodes and Projectional Editor. Panel **A**: Projectional editor showing a simple R script with one assignment expression. Panel **B**: An abstract syntax tree is shown with nodes that correspond to the program in panel A. Panel **C**: Language Concepts for the nodes in Panel (B) (shown as blue boxes). Each concept is connected to other concepts with an open-ended arrow to indicate inheritance (e.g., `A <- B` indicates that B is a sub-concept of A). Green boxes indicate fields of a concept and are connected to the concept that has these fields by a line with a black diamond on the concept that owns the field. This shows that `BinaryExpression` is a concept that is an `Expression` and has three fields: `left`, `operator` and `right`. Dotted lines connect nodes to their concept. For instance, the <- and + nodes are instances of `BinaryExpression`.

DESIGN OF A HIGH-LEVEL DATA ANALYSIS LANGUAGE

Several decisions must be made when designing a new computational language. Most decisions are driven by design goals. We have designed the MetaR language to address the following goals:

1. The language should help users who have no knowledge of programming. The goal is to offer a smooth learning curve for beginners used to GUIs. We favor declarative language constructs over flexibility in parts of the language aimed at beginners.

2. Since a table of data is a frequent input when working with high-throughput data, make Table a first class element of the design. Leverage this element to simplify the annotation of the columns of a table. We rely on the idea that a little formalism (e.g., annotation of table columns) goes a long way to simplify analysis scripts.

3. Eliminate the need to know the language syntax to help beginners get started quickly. We leverage the MPS LW and its projectional editor to this end (Voelter and Solomatov (2010)). The MPS projectional editor provides interactive features, such as auto-completion, that provide guidance to beginners and experts alike when using the language to develop analyses.

4. Provide the ability to blend a scripting language with a graphical user interface. We use language composition and the ability of the MPS LW to render nodes with a mix of text and graphical user interface components.

5. Offer essential data transformations (e.g., joining two tables, taking subsets of rows of a table) via simple, yet composable language elements.

6. Provide means for experts to use their knowledge of the R language to work-around cases when the MetaR language is not sufficiently expressive to perform a specific analysis. We offer the ability to embed R code inside a MetaR analysis, as well as the ability to write scripts in the R language. In both instances, this variant of the R language supports language composition and enables embedding graphical user interfaces inside script fragments.

High-level Design Choices

In addition to these goals, the design of MetaR included several strategic choices. We now present these choices and their rationales:

Choice of a Target Language and Runtime System A language needs a runtime system to execute the code of programs written in the language. A possible choice for a runtime is to target another high-level language (such as Java, or C) but this would require implementing all aspects of data manipulation in the target language. Since the R language (Ihaka and Gentleman (1996)) is widely used for data analysis in biology, we considered using it as a runtime system. Experts biostatisticians and bioinformaticians have developed many R packages that implement advanced analysis for biological high-throughput data. These packages can be used to simplify the implementation of a runtime system for a new data analysis language. We therefore decided that the MetaR language would generate R code in order to take advantage of the packages developed in this language. This decision greatly simplified the implementation of the MetaR language because it removed the need to develop a custom language runtime system.

Data Object Surrogates MetaR makes extensive use of Data Object Surrogates (DOS, our terminology). A data object surrogate is an object that represents other data (the source data). The surrogate often contains only limited information from the original data source. The DOS contains just enough to facilitate referring to the source data in another context for the purpose of data analysis, but not as much as to represent the entire content of the data source in memory. A good example of DOS is the Table object, which stores information about the columns of a data file. The Table DOS describes the columns of the table, but does not store the data contained in the table. A DOS typically has a name which can be used to refer to the DOS and its source data inside a MetaR model. References to table DOS help users refer to the table as they develop an analysis. Our use of the MPS LW facilitates the creation of DOS. In the MPS LW, we model DOS as concepts of the language. For instance, the Table DOS is represented by a Table concept, whose instances can be created in a model as root nodes. DOS are also used in MetaR to represent plots.

Immutable Data Objects Many programming languages (of which C, C++, Java, Perl, Python and R are members) make it possible to define variables or objects whose values can be changed (so called mutable variables). While this provides flexibility, it is a frequent source of confusion for beginners until they have developed their own mental model of how program steps modify variable values. During the design of MetaR, we chose to offer immutable objects rather than mutable variables when possible. This makes MetaR analyses easier to reason about because the value of objects cannot be changed after the object is created. Note that design decision does not prevent adding mutable variables to the MetaR language, but simplifies initial learning of the language by complete beginners.

Organization into Languages

We designed MetaR as a collection of MPS languages. The main language, *org.campagnelab.metar* is aimed at beginners with limited computational experience.

In the next section, we explain how the MetaR language can be used from the point of view of an end-user. This section also includes highlights of features that differ from the state of the art in data analysis. Please note that exhaustive reference documentation is available elsewhere (see Campagne and Simi (2015)) and the goal of the following paragraphs is to provide a sufficient introduction to data analysis with MetaR that readers can understand the impact of the innovations we tested in developing this tool.

The MetaR Language

Tables

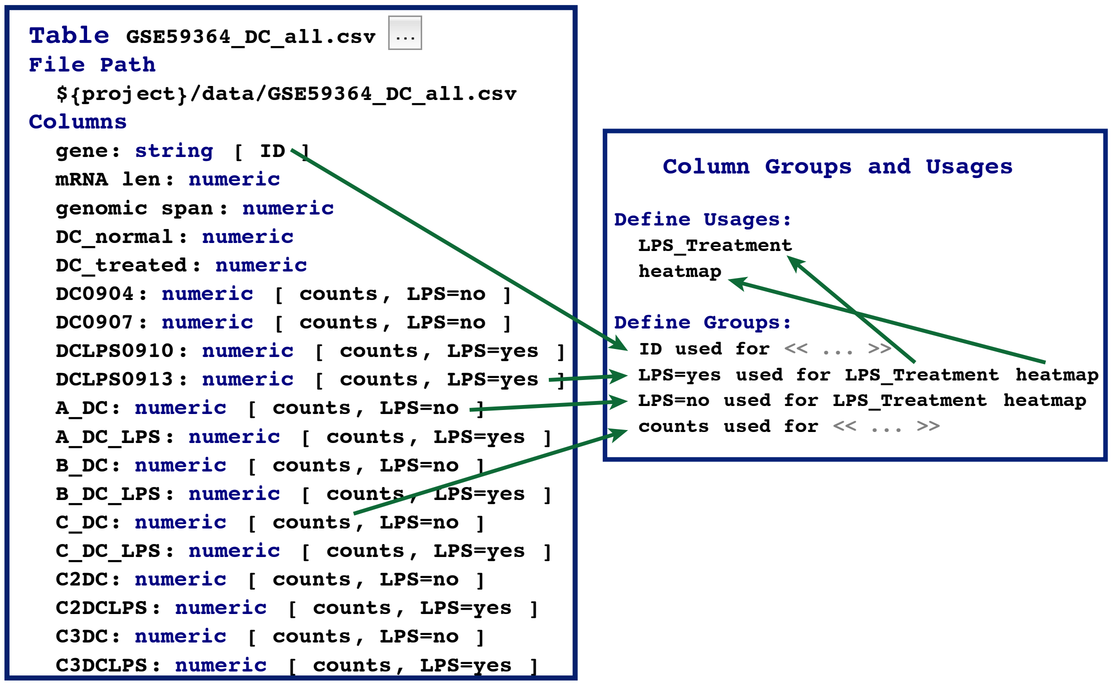
An example of an immutable DOS is the MetaR Table object. In MetaR, objects of type Table represent tabular data with columns and rows of data. An example of a MetaR Table is shown in Figure 2. A MetaR Table is associated to a data file that contains the actual data of the table in a Tab-Separated Value (TSV). The location of the data file can be specified using Variables (i.e., `${project}`), which offer independence from the local file system structure, and are particularly useful when keeping analyses under source control). A table has columns. Columns have names and types, which determine how data in each column is used. Types of data include string, numeric, boolean and enumeration (a small number of pre-defined categories, such as Male and Female). Figure 2 presents a table of RNA-Seq read counts which was obtained from the Gene Expression Omnibus Seguin-Estevez et al. (2014) and annotated to enable analysis with MetaR.

Annotating a Table consists of two steps: (1) browsing to the file that contains the data. This can be accomplished by clicking on the file dialog button (the little square with `...`) to locate the file. Upon selection of a valid file, the MetaR table node inspects the file and determines column names and types. Names and types are then shown in the Table node (under the `Columns` heading). (2) Specific columns can be annotated with one or more Column Groups.

Users can define arbitrary Column Groups in a different node called "Column Groups and Usages" (shown on the right of Figure 2). If two columns are related, user can define a Group Usage to explicitly document the relation. For instance, in Figure 2, the usage `LPS_Treatment` is defined to indicate that the Column Groups `LPS=no` and `LPS=yes` are two kinds of LPS treatments.

Tables and their annotations help users formalize information about data in a table. We find that asking the user to provide such information early on is beneficial because the structure of annotations can be leveraged in other parts of the language to provide intelligent auto-completion, customized for each table of data (for instance, to provide auto-completion for column names when writing expressions, or to select columns to use when joining two tables, examples of intelligent auto-completion is provided in the following sections, see Figure 3).

For instance, in the dataset of Seguin-Estevez et al. (2014), users can indicate which columns contain data for samples that were treated (`LPS=yes`) with lipopolysaccharide (LPS) or not (`LPS=no`). MetaR facilitates the data curation steps of a data analysis project by offering an interactive user interface to help users keep track of annotations. The interface is interactive in several ways: group names can be auto-completed to the groups defined in the "Column Group and Usage" object. Menus are available to add column group annotations to a set of columns that the user has selected. In addition to LPS treatment, Figure 2 shows the `count` annotation, used in an RNA-Seq differential expression analysis to identify which columns contain read counts, the `ID` column group, which uniquely identifies specific rows of the data table and the `heatmap` column group, used to choose which columns groups should be heatmap.



```
Table GSE59364_DC_all.csv ...
File Path
  ${project}/data/GSE59364_DC_all.csv
Columns
  gene: string [ ID ]
  mRNA len: numeric
  genomic span: numeric
  DC_normal: numeric
  DC_treated: numeric
  DC0904: numeric [ counts, LPS=no ]
  DC0907: numeric [ counts, LPS=no ]
  DCLPS0910: numeric [ counts, LPS=yes ]
  DCLPS0913: numeric [ counts, LPS=yes ]
  A_DC: numeric [ counts, LPS=no ]
  A_DC_LPS: numeric [ counts, LPS=yes ]
  B_DC: numeric [ counts, LPS=no ]
  B_DC_LPS: numeric [ counts, LPS=yes ]
  C_DC: numeric [ counts, LPS=no ]
  C_DC_LPS: numeric [ counts, LPS=yes ]
  C2DC: numeric [ counts, LPS=no ]
  C2DCLPS: numeric [ counts, LPS=yes ]
  C3DC: numeric [ counts, LPS=no ]
  C3DCLPS: numeric [ counts, LPS=yes ]
```

```
  Column Groups and Usages

Define Usages:
  LPS_Treatment
  heatmap

Define Groups:
  ID used for << ... >>
  LPS=yes used for LPS_Treatment heatmap
  LPS=no used for LPS_Treatment heatmap
  counts used for << ... >>
```

Figure 2. Table and Column Group objects. This figure presents the Table and Column Group objects. Green arrows show some cross-references among nodes of Tables and Column Groups. For instance, the ID group used to annotate the gene column is a reference to the ID group defined under the Column Group and Usage Container.

This illustrates that the table annotation mechanism is flexible and can be leveraged by specific statements of the language, in order to indicate that the statement needs data annotated in a certain way.

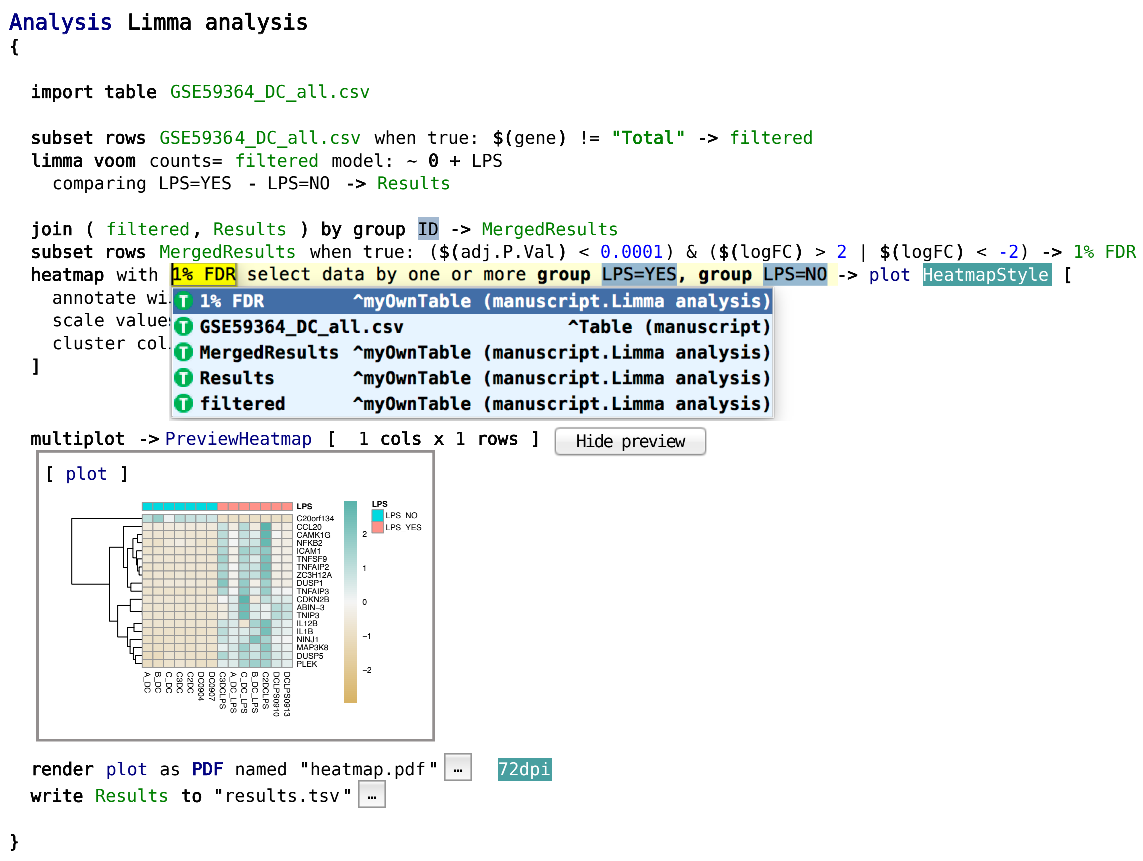
Analyses

Analyses make it possible for users to express how data is to be analyzed. Figure 3 presents a MetaR Analysis node. This analysis is the one we use as a worked example during training sessions we offer at our institution. The editor of an analysis node offers an interface similar to that of a script in a traditional editor, but provides a more interactive and intelligent user interface. For instance, auto-completion is available at every point inside an analysis and suggests possible elements of the language that are compatible with the context at the cursor position.

The user may accept a suggestion and this results in the insertion of the language element at the position of the cursor. When the context calls for referencing a column of a table, for instance, only columns of Tables available at this point of the analysis are shown. While it is still possible to make mistakes when using this interface, mistakes created as a result of typos are less common than in programs encoded as text, for two reasons:

- Auto-completion offers a convenient way to set references between objects. Accepting an auto-completion suggestion helps users avoid typos.
- Some users choose not to use auto-completion to set references and instead type a referenced node name. In this case, mis-typed names that cannot be resolved to a valid node are highlighted in red and in the right margin of the editor (this feature of the MPS LW is available for all languages developed with the MPS platform). This highlighting draws the attention of the user to the error or typo. This feature is also important when merging different versions of an analysis placed under source control or when combining analyses from parts of other analyses (e.g., errors will be clearly marked after a code fragment is pasted into a new analysis).

Auto-completion help is available for the various types of references supported by the MetaR language. Examples of these can be seen on Figure 3 for tables (whose names are in green), plots (whose names are



```
Analysis Limma analysis
{

  import table GSE59364_DC_all.csv

  subset rows GSE59364_DC_all.csv when true: $(gene) != "Total" -> filtered
  limma voom counts= filtered model: ~ 0 + LPS
    comparing LPS=YES  - LPS=NO  -> Results

  join ( filtered, Results ) by group ID -> MergedResults
  subset rows MergedResults when true: ($(adj.P.Val) < 0.0001) & ($(logFC) > 2 | $(logFC) < -2) -> 1% FDR
  heatmap with 1% FDR select data by one or more group LPS=YES, group LPS=NO -> plot HeatmapStyle [
    annotate wi
    scale value
    cluster col
  ]

  multiplot -> PreviewHeatmap [  1 cols x 1 rows ]  Hide preview
  [ plot ]

  render plot as PDF named "heatmap.pdf" …   72dpi
  write Results to "results.tsv" …

}
```

1% FDR ^myOwnTable (manuscript.Limma analysis)
GSE59364_DC_all.csv ^Table (manuscript)
MergedResults ^myOwnTable (manuscript.Limma analysis)
Results ^myOwnTable (manuscript.Limma analysis)
filtered ^myOwnTable (manuscript.Limma analysis)
LPS
LPS_NO
LPS_YES
C20orf134
CCL20
CAMK1G
NFKB2
ICAM1
TNFSF9
TNFAIP2
ZC3H12A
DUSP1
TNFAIP3
CDKN2B
ABIN-3
TNIP3
IL12B
IL1B
NINJ1
MAP3K8
DUSP5
PLEK
A_DC
B_DC
C_DC
C3DC
C2DC
DC0904
DC0907
C3DCLPS
A_DC_LPS
C_DC_LPS
B_DC_LPS
C2DCLPS
DCLPS0910
DCLPS0913
2
1
0
−1
−2

Figure 3. MetaR Analysis. The Analysis node is composed of a list of statements. This analysis works with the table of data presented in Figure 2, removes the row of data where the value `Total` appears in the gene column, performs statistical modeling with Limma Voom to identify genes differentially expressed between LPS treated and control samples, constructs a heatmap and displays the plot as a preview. Finally, the analysis converts the plot to PDF format and writes the joined table (statistics and counts) in the `results.tsv` file.

in blue), styles (names shown with a green background and white foreground, such as HeatmapStyle), or Column Group names (shown with a blue grey background and black foreground). Pressing control-B (or command-B on Mac) with the cursor on these nodes navigates to the destination of the reference (a menu is also available to help novice users discover this navigation mechanism). References may point to children nodes defined inside an analysis (e.g., plots), or nodes defined outside the analysis (e.g., tables and column groups).

Importantly, the MetaR user interface can also display buttons and images directly as part of the language. This feature takes advantage of the ability of the MPS LW to embed arbitrary graphical elements in the projectional editor. This capability is illustrated in Figure 3 by the "Hide Preview" button and by the heatmap image shown immediately below the `multiplot` keyword (pressing this button hides the plot preview).

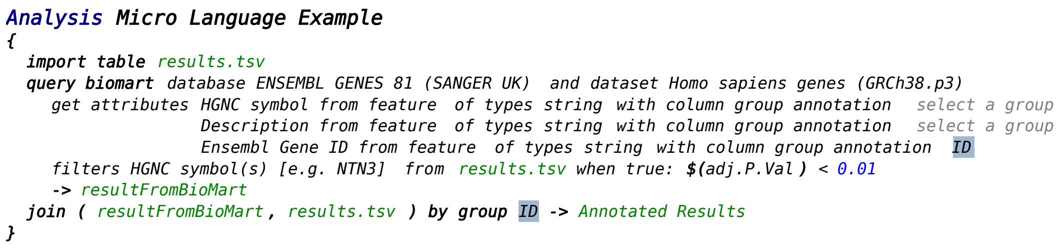
The level of interactivity provided by the MetaR user interface is best conveyed by watching video recordings of its use. We provide training videos at `http://metaR.campagnelab.org` that help illustrate how much more interactive the MetaR language is compared to other languages commonly used for data analysis.

Language Composition and Micro-Languages

Since MetaR is implemented as a set of MPS languages, it fully supports language composition (Voelter and Solomatov (2010)). Language composition has no equivalent in text-based programming languages and many readers may be therefore unfamiliar with this technique. We will use an example to explain the advantage of this technique for data analysis.

Consider the table of results produced by the analysis shown in Figure 3. Users are likely to need to annotate the subset of genes found differentially expressed with gene names and gene descriptions. Information such as this is available in the Biomart system Haider et al. (2009).

To illustrate language composition, we created a new kind of MetaR statement called `query biomart`, which we defined in a micro-language. A micro-language is a language which provides only a few concepts meant to extend a host language. In this case, the MetaR language is the host language and `query biomart` is a concept contributed by the the micro-language. The purpose of this concept is to connect to Biomart and retrieve data. In the R language, this functionality is provided as a BioConductor package (called "biomaRt", Durinck et al. (2005))



```
Analysis Micro Language Example
{
  import table results.tsv
  query biomart database ENSEMBL GENES 81 (SANGER UK)  and dataset Homo sapiens genes (GRCh38.p3)
    get attributes HGNC symbol from feature  of types string with column group annotation   select a group
                   Description from feature  of types string with column group annotation   select a group
                   Ensembl Gene ID from feature  of types string with column group annotation  ID
    filters HGNC symbol(s) [e.g. NTN3]  from results.tsv when true: $(adj.P.Val ) < 0.01
    -> resultFromBioMart
  join ( resultFromBioMart , results.tsv ) by group ID -> Annotated Results
}
```

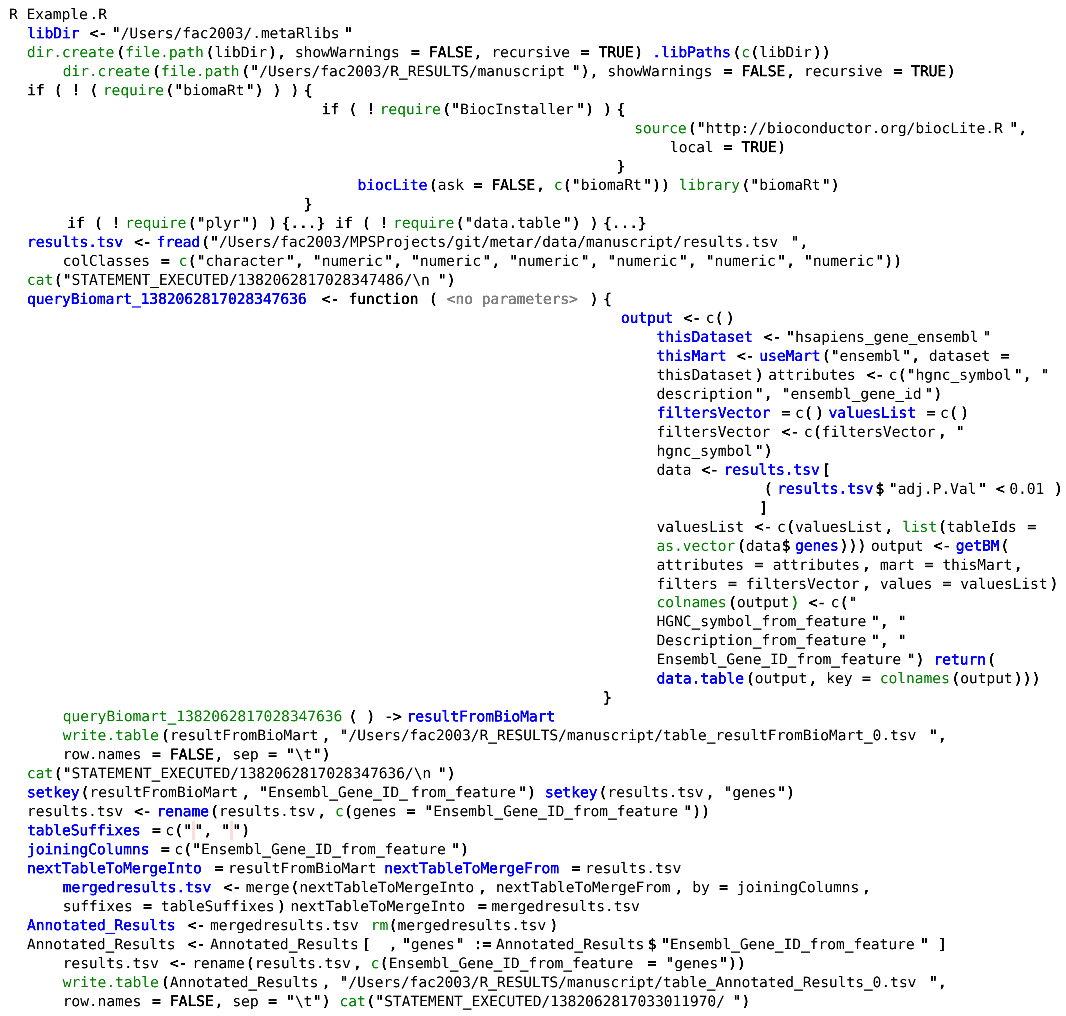
Figure 4. Example of Micro-language Composition. The query biomart stament is defined in a micro-language called *org.campagnelab.metar.biomart*, which extends the host language *org.campagnelab.metar.tables*. The biomart language provides one statement that offers an interactive user interface to help users retrieve data from biomart. This language reuses expressions and tables from the host language. Micro-languages can be enabled or disabled dynamically by the end-user at the level of a model. This example retrieves Human ENSEMBL identifiers and gene descriptions using the HGNC gene symbols used as identifiers in the Results table (see Figure 3 for the analysis that produced Results).

Querying Biomart in R consists in calling one of the functions defined in the package with specific parameters. The statement is very specialized, and for this reason would not typically be part of the core statements of a text-based programming language. Leveraging language composition, we can offer a dedicated statement that supports auto-completion in a remote Biomart instance. The statement acts as a specialized user interface designed to help users retrieve data from Biomart (in very much the same way that the web-based interface to Biomart helps users query this resource, but here completely integrated with the MetaR host language).

Figure 4 illustrates how the `query biomart` statement can be used to obtain gene annotations. In order to use these statements, end-users of MetaR would declare using both the *org.campagnelab.metar.tables* (the host language) and *org.campagnelab.metar.biomart* (the micro-language). In this specific case, the micro-language is provided with the MetaR distribution, but end-users can also implement other micro-languages to seamlessly combine them with the host language (the process for doing so is described in the MetaR documentation booklet Campagne and Simi (2015), Chapter 10). This capability makes it possible to customize the data analysis process for specific problems in much more flexible ways than would be possible with text-based programming languages: with the `query biomart` statement, we demonstrated that it is possible to remotely query databases to support auto-completion directly in the language. In contrast, text-based languages can only be extended in ways compatible with the syntax of the programming language, and are not able to support such levels of interactivity.

Composable R language

In addition to the MetaR language illustrated in Figure 2-4, we have developed a composable R language. This language models the traditional R language Ihaka and Gentleman (1996), but supports language composition. Composable R is implemented in the language *org.campagnelab.metar.R* distributed with MetaR. R programs can be pasted in text form into an RScript root node and the text is parsed and converted to nodes of the composable R language. In Figure 5, we show the R code equivalent to the analysis shown in Figure 4. This R script was pasted from the text generated automatically from the MetaR analysis shown in Figure 4. Executing this script is supported in the MPS LW and yields the same result that of the simpler MetaR script shown in Figure 4.



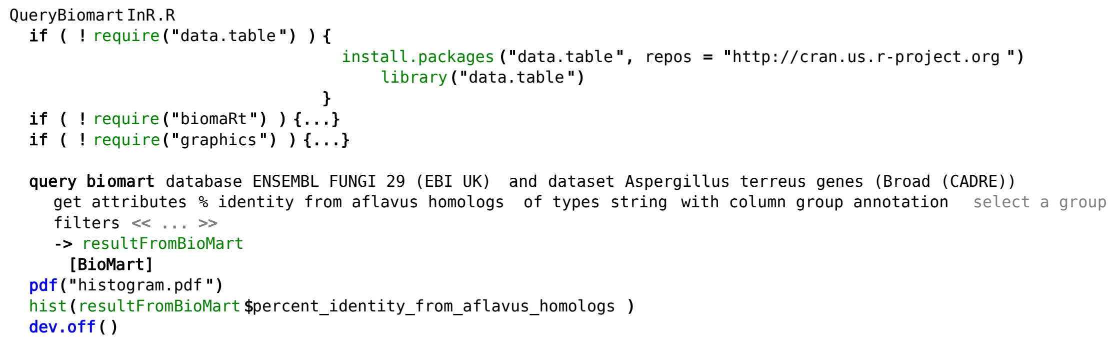
```
R Example.R
  libDir <- "/Users/fac2003/.metaRlibs"
  dir.create(file.path(libDir), showWarnings = FALSE, recursive = TRUE) .libPaths(c(libDir))
      dir.create(file.path("/Users/fac2003/R_RESULTS/manuscript"), showWarnings = FALSE, recursive = TRUE)
  if ( ! ( require("biomaRt") ) ) {
                                   if ( ! require("BiocInstaller") ) {
                                                                        source("http://bioconductor.org/biocLite.R",
                                                                            local = TRUE)
                                                                      }
                                       biocLite(ask = FALSE, c("biomaRt")) library("biomaRt")
                                 }
      if ( ! require("plyr") ) {...} if ( ! require("data.table") ) {...}
  results.tsv <- fread("/Users/fac2003/MPSProjects/git/metar/data/manuscript/results.tsv",
      colClasses = c("character", "numeric", "numeric", "numeric", "numeric", "numeric"))
  cat("STATEMENT_EXECUTED/1382062817028347486/\n")
  queryBiomart_1382062817028347636 <- function ( <no parameters> ) {
                                                                    output <- c()
                                                                        thisDataset <- "hsapiens_gene_ensembl"
                                                                        thisMart <- useMart("ensembl", dataset =
                                                                        thisDataset) attributes <- c("hgnc_symbol", "
                                                                        description", "ensembl_gene_id")
                                                                        filtersVector = c() valuesList = c()
                                                                        filtersVector <- c(filtersVector, "
                                                                        hgnc_symbol")
                                                                        data <- results.tsv[
                                                                                    ( results.tsv$"adj.P.Val" < 0.01 )
                                                                                   ]
                                                                        valuesList <- c(valuesList, list(tableIds =
                                                                        as.vector(data$genes))) output <- getBM(
                                                                        attributes = attributes, mart = thisMart,
                                                                        filters = filtersVector, values = valuesList)
                                                                        colnames(output) <- c("
                                                                        HGNC_symbol_from_feature", "
                                                                        Description_from_feature", "
                                                                        Ensembl_Gene_ID_from_feature") return(
                                                                        data.table(output, key = colnames(output)))
                                                                  }
      queryBiomart_1382062817028347636 ( ) -> resultFromBioMart
      write.table(resultFromBioMart, "/Users/fac2003/R_RESULTS/manuscript/table_resultFromBioMart_0.tsv",
      row.names = FALSE, sep = "\t")
  cat("STATEMENT_EXECUTED/1382062817028347636/\n")
  setkey(resultFromBioMart, "Ensembl_Gene_ID_from_feature") setkey(results.tsv, "genes")
  results.tsv <- rename(results.tsv, c(genes = "Ensembl_Gene_ID_from_feature"))
  tableSuffixes = c("", "")
  joiningColumns = c("Ensembl_Gene_ID_from_feature")
  nextTableToMergeInto = resultFromBioMart nextTableToMergeFrom = results.tsv
      mergedresults.tsv <- merge(nextTableToMergeInto, nextTableToMergeFrom, by = joiningColumns,
      suffixes = tableSuffixes) nextTableToMergeInto = mergedresults.tsv
  Annotated_Results <- mergedresults.tsv rm(mergedresults.tsv)
  Annotated_Results <- Annotated_Results[ , "genes" := Annotated_Results$"Ensembl_Gene_ID_from_feature" ]
      results.tsv <- rename(results.tsv, c(Ensembl_Gene_ID_from_feature = "genes"))
      write.table(Annotated_Results, "/Users/fac2003/R_RESULTS/manuscript/table_Annotated_Results_0.tsv",
      row.names = FALSE, sep = "\t") cat("STATEMENT_EXECUTED/1382062817033011970/")
```

Figure 5. R language equivalent of the Analysis shown in Figure 4. To produce this figure, the analysis shown in Figure 4 was generated to the R language and the text was pasted in a RScript node of the composable R language. Automatic parsing of the R code into composable R objects yields a composable R version of the biomart example. Notice that boiler plate code needed to import R packages is shown only for the biomaRt package. Subsequent package import statements have been folded {...} to save space in the Figure. Folding is directly supported by the MPS LW. Function calls are highlighted in green and are linked to the function declaration in the package stub (end-user can navigate to each function to review its list of arguments, for instance). Comparing the complexity of this code with the equivalent MetaR code shown in Figure 4 makes a strong case for the need for simplified languages for data analysis.

Micro-Languages for the R Language

A composable R language makes it possible to create micro-languages that compose directly with R as the host language. We demonstrate this capability by adapting the `query biomart` statement shown in Figure 4 to the R language. Adaptation is simple because both MetaR and R generate to the same target language (R). In this case, we create a sub-concept of `Expr` (this type represents any R expression), and define a field of type `Biomart` (the concept that implements `query biomart`). This simple adapter is sufficient to make it possible to use the `query biomart` user interface inside an R script and is defined in the language *org.campagnelab.metar.biomartToR*. The result of composing the adapter language with composable R is shown in Figure 6.

This example illustrates that a composable R language makes it possible to mix regular R code with new types of language constructs that can include user interfaces elements. This opens up new possibilities to facilitate repetitive analyses in R, for instance for specific data science domains (e.g., the Biomart example is useful for bioinformatic data analyses), but also for more general activities where simpler ways to perform a task would be beneficial. An example of this would be a micro-language to facilitate the use of packages to replace the boiler-plate package import code found at the beginning of most R scripts.



```
QueryBiomartInR.R
  if ( ! require("data.table") ) {
                                  install.packages("data.table", repos = "http://cran.us.r-project.org ")
                                      library("data.table")
                                }
  if ( ! require("biomaRt") ) {...}
  if ( ! require("graphics") ) {...}

  query biomart database ENSEMBL FUNGI 29 (EBI UK)  and dataset Aspergillus terreus genes (Broad (CADRE))
    get attributes % identity from aflavus homologs  of types string  with column group annotation   select a group
    filters << ... >>
    -> resultFromBioMart
      [BioMart]
  pdf("histogram.pdf")
  hist(resultFromBioMart$percent_identity_from_aflavus_homologs )
  dev.off()
```

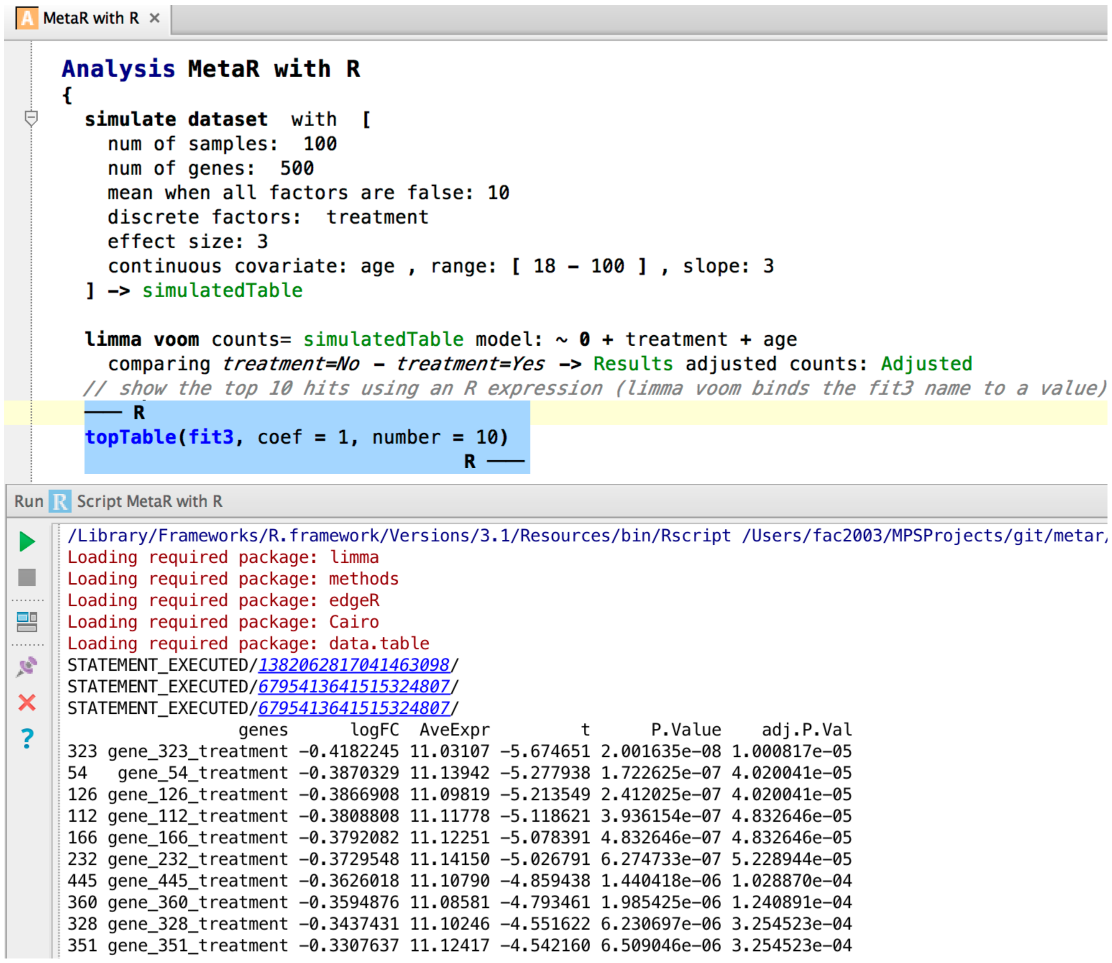
Figure 6. Composing Query Biomart with the composable R language. We developed an adapter that makes it possible to use the MetaR `query biomart` statement directly inside a composable R Script. This figure shows how the `query biomart` Expression adapter appears when used inside an R script. Notice how the table and column adapters are used inside a regular `hist()` function call `resultFromBioMart$percent_identity_from_aflavus_homologs`. These adapters make it possible to refer to the table produced by the statement as an R expression and provide auto-completion for column names in the table (determined dynamically based on the query expressed in the `query biomart` statement).

Using R Expressions in the MetaR Language

Figure 7 illustrates that language composition can also be used to embed R expressions inside a MetaR analysis. This extension is possible because both analyses and R expressions generate code compatible with the syntax of the R programming language. Providing a way to embed the full language in a simpler analysis language offers a guarantee that the end-user will not be overly limited by restrictions of the simpler language.

SOFTWARE

MetaR is distributed as a plugin of the MPS LW. Instructions for installing the software are available online at `http://metaR.campagnelab.org`. Briefly, after installing MPS, users can download and activate plugins with the Preferences/Plugins (Mac) or Settings/Plugins (Windows/Linux) menu. Plugins are stored as Zip files on the Jetbrains Plugin repository `https://plugins.jetbrains.com/category/index?pr=mps&category_id=92` and can also be downloaded and installed manually from the zip file. Source code (technically, MPS languages serialized to files) are distributed on GitHub at `https://github.com/campagnelaboratory/MetaR`. MetaR (and the MPS LW) are distributed under the open-source Apache 2.0 license.



```
Analysis MetaR with R
{
  simulate dataset  with  [
    num of samples:  100
    num of genes:  500
    mean when all factors are false: 10
    discrete factors:  treatment
    effect size: 3
    continuous covariate: age , range: [ 18 - 100 ] , slope: 3
  ] -> simulatedTable

  limma voom counts= simulatedTable model: ~ 0 + treatment + age
    comparing treatment=No - treatment=Yes -> Results adjusted counts: Adjusted
  // show the top 10 hits using an R expression (limma voom binds the fit3 name to a value)
  —— R
  topTable(fit3, coef = 1, number = 10)
                                     R ——
```

```
/Library/Frameworks/R.framework/Versions/3.1/Resources/bin/Rscript /Users/fac2003/MPSProjects/git/metar,
Loading required package: limma
Loading required package: methods
Loading required package: edgeR
Loading required package: Cairo
Loading required package: data.table
STATEMENT_EXECUTED/1382062817041463098/
STATEMENT_EXECUTED/6795413641515324807/
STATEMENT_EXECUTED/6795413641515324807/
                 genes      logFC  AveExpr         t      P.Value    adj.P.Val
323 gene_323_treatment -0.4182245 11.03107 -5.674651 2.001635e-08 1.000817e-05
54   gene_54_treatment -0.3870329 11.13942 -5.277938 1.722625e-07 4.020041e-05
126 gene_126_treatment -0.3866908 11.09819 -5.213549 2.412025e-07 4.020041e-05
112 gene_112_treatment -0.3808808 11.11778 -5.118621 3.936154e-07 4.832646e-05
166 gene_166_treatment -0.3792082 11.12251 -5.078391 4.832646e-07 4.832646e-05
232 gene_232_treatment -0.3729548 11.14150 -5.026791 6.274733e-07 5.228944e-05
445 gene_445_treatment -0.3626018 11.10790 -4.859438 1.440418e-06 1.028870e-04
360 gene_360_treatment -0.3594876 11.08581 -4.793461 1.985425e-06 1.240891e-04
328 gene_328_treatment -0.3437431 11.10246 -4.551622 6.230697e-06 3.254523e-04
351 gene_351_treatment -0.3307637 11.12417 -4.542160 6.509046e-06 3.254523e-04
```

Figure 7. Composing R Expressions with the MetaR Language. Top panel: this example illustrates that it is possible to use R code inside a MetaR analysis. In this snapshot, R code is delimited by the — R and R — markers and shown with a blue background. Embedding R code in MetaR provides flexibility to perform operations for which MetaR statements have not yet been developed. The analysis shown simulates a dataset using simple parameters and tests the ability of Limma voom, as integrated with MetaR, to call differentially expressed genes. Bottom panel: shows the result of executing the analysis inside the MPS LW. As part of execution, the analysis is converted to R code, this code is run and standard output displayed inside the LW. The `STATEMENT_EXECUTED//` lines hyperlink the progress of the execution with each specific analysis statement that has been executed.

DISCUSSION

Data Object Surrogates and Relation to Meta Data

DOS are related, but different from metadata. For instance, the Table DOS provides metadata about the file that contains the tabular data represented by Table nodes. It lists columns, associates columns to groups and defines group usages. This type of information can be thought of as metadata about the file that contains the tabular data. However, there is an important difference between DOS and metadata. For instance, a MetaR Table only provides metadata relevant to the analysis that the user needs to perform. It makes no effort to provide information that would have a meaning outside of the user's analysis. This simplification maximizes the benefit of annotation while keeping the effort needed to produce it minimal and local to the user who actually needs the annotation.

Graphical User Interfaces for Data Analysis

Programs with graphical user interfaces (GUIs) (also called direct manipulation interfaces Galitz (2007)) are often popular among beginners who are starting with data analysis and have no programming or scripting experience. GUIs are popular in part because they facilitate discovery of software functionality directly when using the software. They do not require prior-knowledge of syntax.

Data Analysis software with GUIs constrains how analysis is to be performed and provides clear menus and buttons that make it obvious what the program can do. A user can often discover new ways to perform analysis with these tools simply by browsing the user interface and looking at choices offered in menus and dialogs of the program. While such programs are favored by beginners (because they are relatively easy to learn), more advanced users who need to perform similar analyses across several datasets tend to strongly prefer analysis software that does not require repeating interactions with a GUI for every new dataset that must be studied. The novel approaches we have used to develop MetaR share these advantages with GUIs.

A minority of analysis software with GUIs also supports writing and running scripts in their user interface. For instance, JMP from SAS Inc. is an example of a statistical analysis software with GUI that also offers a scripting language. However, when scripting is offered, it is often only loosely integrated with the rest of the interface. Furthermore, users who are familiar with the GUI often need to learn scripting from scratch and do not benefit much from their prior experience using the GUI.

Scripting and Programming Languages for Data Analysis

Scripting and programming languages are popular options for data analysis because analyses encoded in scripts or programs can be reused with different datasets. This makes these options popular among researchers who have programming skills and engage frequently in data analysis. The popularity and power of scripting for data analysis is epytomized by the development of the R language Ihaka and Gentleman (1996), which has become a defacto workhorse of open data science in biology. The versatility of the R language is its strength, but mastering the language requires elements of programming. Learning the R programming language is not as simple as learning how to use a GUI analysis tool and many users who would benefit from data analysis experience difficulties with the steep learning curve involved in learning programming and the R language.

In contrast to R, the MetaR language offers a much simpler alternative for users who have no prior programming background. At the same time, the Composable R language offers the means for expert R users to extend the R language with micro-languages in order to provide custom user interfaces. Such interfaces could be used to flatten the learning curve for novice data analysts or to empower expert data analysts with expressive means to encode solution to specific problems. Since both these options are available in the same platform (the MPS LW), users who become skilled with one language acquire transferable skills that help them learn other languages available on the platform.

Relation to Electronic Notebooks

MetaR shares some similarities to electronic notebooks such as IPython Pérez and Granger (2007), Jupyter (`https://jupyter.org/`) and Beaker (`http://beakernotebook.com/`) notebooks, but also has some important differences.

Regarding analogies, both MetaR and notebooks can be used to present analysis results alongside the code necessary to reproduce the results. For instance, the MetaR multi-view plot can be used to show a plot at the location where the statement is introduced in an analysis.

MetaR was developed approximately over the course of one year (2015). As such the software cannot be expected to be as feature-rich as software developed for many years. Beside this obvious difference, MetaR has the advantage to support language composition. In contrast, current data analysis notebooks support conventional programming languages constructed using text-based technology. Therefore, the closest that notebooks can approach language composition is to support multiple languages in one notebook, a so-called polyglot feature, available for instance in the Beaker notebook. Polyglot notebooks are useful, but cannot be extended by data analysts to customize languages for the requirements of a specific analysis project or domain. For instance, supporting a simple analysis language like MetaR would not be possible without developing a MetaR compiler and an associated execution kernel for the notebook. Developing and using micro-languages together with the traditional languages supported by the notebooks is also not possible.

Hence, the approach taken with MetaR is different from notebooks in two major ways. First, MetaR provides flexibility in designing new languages or micro-languages. It is not constrained by the syntax of a full programming language. Extending MetaR often consists in adding just one statement to an existing language. This promotes collaborative language design and development since many users can acquire sufficient skills to create one or two statements, reusing the building blocks provided by the host language (the steps needed to extend MetaR with a new language statement are described in the user manual Campagne and Simi (2015)). As long as a new statement generates valid R code, a MetaR Analysis that contains this statement will be executable.

Second, the syntax of the MetaR languages is not limited to text scripts or programs. Language Workbench technology used to implement MetaR supports graphical notations and diagrams as well as text. These differences combine to make it easier to design and implement custom data analysis abstractions with the LWT approach than it is possible with current electronic notebooks. Interestingly, the R IPython kernel could be used to execute scripts generated from MetaR analyses, which would provide an interactive console similar to that offered in the IPython notebook inside the MPS LW.

Reproducible Research and Education

MetaR analysis and Composable R scripts can be executed seamlessly with an R environment installed inside a docker image (see Methods). Users can enable this feature by providing a few details about the installation of docker on their computer and checking the "Run with Docker" option in the MPS LW. This feature is particularly useful to facilitate reproducible research because docker images can be tagged with version numbers and always result in the same execution environment at the start of an analysis. This makes it possible to pre-install specific versions of R, CRAN and Bioconductor packages in a container and distribute this image with the MetaR analyses or R scripts that implement the analysis inside the container. While this is possible also with R, using docker on the command line, the customization of the MPS LW makes it seamless to run analyses with docker. We are not aware of a similar feature being supported by current R IDEs.

We found this feature also particularly useful for training sessions where installation of a working R environment can be challenging on trainees' laptops. Using docker, we simply request that trainees pre-install Kitematic (available on Mac and Windows), or run docker natively on Linux and download the image we prepared with the packages used in the MetaR training sessions. The ability to run MetaR analysis in docker container results in a predictable installation of dependencies for training session and frees more of the instructor's time to present data analysis techniques.

METHODS

We have used the MPS Language Workbench (`http://jetbrains.com/mps`), as also described in Campagne (2014) and Campagne (2015). For an introduction to Language Workbench Technology (LWT) in the context of bioinformatics see Simi and Campagne (2014) and Benson and Campagne (2015) in the context of predictive biomarker model development.

Language Design

We designed the MetaR MPS languages through an iterative process, releasing the languages at least weekly to end-users at the beginning of the project and adjusting designs and implementations according to user feedback. Full language developments logs are available on the GitHub code repository (`https://github.com/CampagneLaboratory/MetaR`). Briefly, we designed abstractions to facilitate specific analyses and implemented these abstractions with the structure, editor, constraints and typesystem aspects of MPS languages. Generated R code is produced from nodes of the languages using the *org.campagnelab.TextOutput* plugin. An illustration of the steps required to develop one language statement is available in Chapter 10 of the MetaR documentation booklet (see Campagne and Simi (2015)).

Table Viewer

We implemented a Table viewer as an MPS Tabbed Tool, using the MPS LW mechanisms for user interface extension (see Campagne (2015)). The table viewer provides the ability to inspect the data content of any table produced during an analysis, or any input table. When the cursor is positioned over a node that represent a FutureTable (created when running the R script generated from the MetaR Analysis), and the viewer is opened, it tries to load the data file that the analysis would create for this table. If the file is found, the content is displayed using a Java Swing Component in the MPS user interface of the Table Viewer tool.

Language Execution

MetaR analyses can be executed directly from within the MPS LW. This capability was implemented with Run Configurations (see Campagne (2015), Chapter 5).

Execution in a Docker Container

In order to facilitate reproducible execution, we implemented optional execution within a Docker container. A docker image was created to contain a Linux operating system and a recent distribution of the R language (provided in the rocker-base image), as well as several R packages needed when executing the MetaR statements. The Run Configuration was modified to enable execution inside a docker container when the user selects a checkbox ”execute inside docker container”. Information necessary to run with docker (i.e., location of the docker executable, docker server connection settings and image name and tag) is collected under a tab in the MPS Preferences (Other Settings/Docker).

ACKNOWLEDGMENTS

The authors thank Carl Boettiger and Dirk Eddelbuettel, who developed the `rocker-base` image, used in the MetaR project to facilitate the creation of docker images for training sessions and reproducible research. This investigation was supported by the National Institutes of Health NIAID award 5R01AI107762-02 to Fabien Campagne and by grant UL1 RR024996 (National Institutes of Health (NIH)/National Center for Research Resources) of the Clinical and Translation Science Center at Weill Cornell Medical College.

REFERENCES

Backus, J. (1978). The history of fortran i, ii, and iii. In *History of programming languages I*, pages 25–74. ACM.

Backus, J. W. (1958). Automatic programming: properties and performance of fortran systems i and ii. In *Proceedings of the Symposium on the Mechanisation of Thought Processes*, pages 165–180.

Benson, V. M. and Campagne, F. (2015). Language workbench user interfaces for data analysis. *PeerJ*, 3:e800.

Campagne, F. (2014). *The MPS Language Workbench*, volume I. Fabien Campagne.

Campagne, F. (2015). *The MPS Language Workbench*, volume II. Fabien Campagne.

Campagne, F. and Simi, M. (2015). *MetaR Documentation Booklet*. Fabien Campagne.

Dmitriev, S. (2004). Language oriented programming: The next programming paradigm.

Durinck, S., Moreau, Y., Kasprzyk, A., Davis, S., Moor, B. D., Brazma, A., and Huber, W. (2005). BioMart and Bioconductor: a powerful link between biological databases and microarray data analysis. *Bioinformatics*, 21:3439–3440.

Erdweg, S., van der Storm, T., Völter, M., Boersma, M., Bosman, R., Cook, W. R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., et al. (2013). The state of the art in language workbenches. In *Software language engineering*, pages 197–217. Springer.

Galitz, W. O. (2007). *The Essential Guide to User Interface Design: An Introduction to GUI Design Principles and Techniques*. John Wiley & Sons.

Haider, S., Ballester, B., Smedley, D., Zhang, J., Rice, P., and Kasprzyk, A. (2009). BioMart Central Portal–unified access to biological data. *Nucleic Acids Res.*

Ihaka, R. and Gentleman, R. (1996). R: a language for data analysis and graphics. *Journal of computational and graphical statistics*, 5(3):299–314.

Mesnard, L., Muthukumar, T., Burbach, M., Li, C., Shang, H., Dadhania, D., Lee, J. R., Sharma, V. K., Xiang, J., Suberbielle, C., Carmagnat, M., Ouali, N., Rondeau, E., Friedewald, J. J., Abecassis, M. M., Suthanthiran, M., and Campagne, F. (2015). Exome sequencing and prediction of long-term kidney allograft function.

Pérez, F. and Granger, B. E. (2007). IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29.

Seguin-Estevez, Q., Dunand-Sauthier, I., Lemeille, S., Iseli, C., Ibberson, M., Ioannidis, V., Schmid, C. D., Rousseau, P., Barras, E., Geinoz, A., Xenarios, I., Acha-Orbea, H., and Reith, W. (2014). Extensive remodeling of DC function by rapid maturation-induced transcriptional silencing. *Nucleic Acids Research*, 42(15):9641–9655.

Simi, M. and Campagne, F. (2014). Composable languages for bioinformatics: the nyosh experiment. *PeerJ*.

Simonyi, C. (1995). The death of computer languages, the birth of intentional programming. Technical report.

Voelter, M. and Solomatov, K. (2010). Language modularization and composition with projectional language workbenches illustrated with MPS. *Software Language Engineering, SLE*.