Learning Data Analysis with MetaR

Manuele Simi

Abstract The analysis of biological data is an important part of modern biological and medical research. Many statistical and computational tools are available for statisticians and data scientists, providing them with both computational power and flexibility. However, these tools are often not suitable for biomedical researchers performing data processing and simple analyses. Learning to code is indeed challenging and quite difficult when coming from a non-technical background. This chapter introduces MetaR, a tool designed to help biologists and clinicians to learn, and perform, the basis of data analysis. Originally created as an educational software, it eventually evolved into a mature and stable Domain Specific Language (DSL) used to support various aspects of a research project up to the creation of figures suitable for scientific publications. MetaR generates to R code, but users do not even see it. By providing high-level abstractions, MetaR hides most of the details of each step in the data analysis process. Because of its capability to blend scripting and graphical elements, it provides a novel approach to the practice of analyzing data. Simplified executions and syntax-completion are additional features that make MetaR easy to use for beginners. Language composition is available for advanced users that wish to contribute and extend the project. In our data-rich age, there is a great divide between biomedical researchers and data analysts. MetaR has proved to be an educational bridge between these two worlds.

1 Introduction

Bioinformatics is an interdisciplinary field of study that combines biology with computer science to understand biological data. The analysis of these data is

M. Simi (✉)
Englander Institute for Precision Medicine, Weill Cornell Medicine, New York, NY, USA

Institute for Computational Biomedicine, Weill Cornell Medicine, New York, NY, USA

Department of Physiology and Biophysics, Weill Cornell Medicine, New York, NY, USA
e-mail: mas2182@med.cornell.edu

A. Bucchiarone et al. (eds.), *Domain-Specific Languages in Practice*,
https://doi.org/10.1007/978-3-030-73758-0_9

an important part of most modern clinical and genomic studies. However, while statistical and computational tools are available for statisticians and data scientists, providing them with both computational power and flexibility, they are often not suitable for biomedical researchers looking to perform data processing and simple analyses.

Following a long track record of success in providing tools and services to support the conduct, management, and evaluation of research, the Informatics Core at the Clinical & Translational Science Center (CTSC)[1] at Weill Cornell Medicine (WCM) has put effort and funds to create new computational methods to facilitate data analysis.

For this purpose, the CTSC has developed MetaR [1, 2], a new kind of interactive tool providing high-level data abstraction, manipulation, and visualization. MetaR is composed of a set of data analysis languages built with the Language Workbench Technology [3] offered by JetBrains Meta Programming System (MPS) [4] to make data analysis easier for biologists and clinicians with minimal computational skills.

The goal of the MetaR project is to provide a tool for educating biomedical researchers in data analysis by keeping the learning curve as smooth and simple as possible.

2 Domain

2.1 What Is Data Analysis?

Data analysis commonly refers to a set of methods for inspecting, cleaning, transforming, modeling, interpreting, and visualizing data to discover useful information for downstream decision-making activities. However, data analysis is much more than a set of formal technical procedures: it involves goals, relationships, and ideas, in addition to working with the actual data itself. Simply put, data analysis includes ways of working with information (data) to support the goals and plans of a business or research project.

To be effective, the data analysis process must be based upon the domain and mission of the organization, as well as the skills of the team in charge of working with the data. Bioinformatics [5] is the field that creates and applies data analysis techniques to biological data. Such techniques mainly include the development of computational methods to analyze small and large collections of data to make new interpretations or predictions, come up with biological insights, and, ultimately, help advance biomedical research.

[1] https://ctscweb.weill.cornell.edu/.

In bioinformatics, there are several types of analysis techniques depending on the nature of the data to analyze, the domain, and the technology. The branch of data analysis in the scope of this chapter is called *Statistical Data Analysis*. In particular, MetaR focuses on procedures of performing various statistical operations over biological datasets (see Sect. 6).

2.2 Which Data?

The initial focus of MetaR was on analysis of RNA-Seq data [6] and the creation of heatmaps (a type of plot we encounter later in the chapter) for educational purposes. Next-generation sequencing is without doubt one of the most important technological advances in molecular biology of the past 15 years. Data coming from RNA sequencing experiments determine the identity and abundance of RNA molecules in biological samples, providing the investigator with a snapshot of the activity in cells. The analysis of these data produces new knowledge and insights from basic science, to drug discovery, to medical treatments.

Nevertheless, since the beginning, MetaR has been conceived to be readily extended to support a broad range of data.

2.3 The R Statistical Language

The most popular tool for statistical data analysis is probably R [7], a language and environment for statistical computing. Especially in academia, many researchers and scholars write programs in R for processing and manipulating data, analyzing the relationships and correlations among datasets, visualizing results, and more. R also helps to identify patterns and trends for interpretation of the data.

R provides many features, notably:

- Data cleaning and reduction
- Data visualization
- Statistical hypothesis testing
- Statistical modeling
- Data analysis report output (with R markdown)

Due to its expressive syntax and a great number of packages (see Sect. 2.5.2) that extend its capabilities, R has grown in popularity in recent years. However, even for an experienced programmer, learning R can be a frustrating challenge. It is frequently stated that R is a programming language built by data scientists for data scientists and this results in a mixed approach to the language. There is a lot of unusual coding syntax and daily practice problems to work through before a programmer can be productive with R. Plus, R is a command-line program, which means that the R command console interprets each line of code as it is entered and,

if it is valid, executes it and returns the result in the console. This sets R apart from other more friendly and visual languages and, because of this, biomedical researchers have even fewer chances to approach data analysis with R. It is not surprising that many beginners give up or drop off at points along their climb up this "learning hill."

With the MetaR DSL, we tried to eliminate (or reduce) the so-called cliff of boring (the lapse of time between starting to learn and seeing the results). In MetaR, most of the details of the R language are hidden or made optional; familiar keywords based on the domain help users to identify what to expect as input and output from each command. A shift in the programming paradigm (see Sect. 2.5.1) also simplifies the understanding of the program's logic.

2.4 MetaR Languages

The MetaR Languages[2] (we refer to them also as MetaR software or simply MetaR) have been under active development since late 2014. Initially, the CTSC created MetaR as an educational tool to introduce biologists and clinicians to data analysis. Despite retaining this original purpose, over the years, it has eventually evolved into a mature and stable Domain Specific Language (DSL) used to support various aspects of a research project up to the creation of figures suitable for scientific publications.

> DSLs are smaller and easier to understand, they allow nonprogrammers to see the code that drives important parts of their business. By exposing the real code to the people who understand the domain, you enable a much richer communication channel between programmers and their customers.
>
> Martin Fowler [8]

Learning to code is indeed a challenging endeavor and quite difficult when coming from a non-technical background. MetaR fills this gap within health care research projects by bringing biomedical researchers closer to data scientists so they can establish a more fruitful collaboration.

[2] https://metar-languages.github.io/.

2.5 Relation with R

MetaR generates R code in the form of R scripts.

R scripts are text files with lists of R instructions to execute from the command line. Later in this chapter, we will learn that the main concept in MetaR is called `Analysis` (Sect. 4.2.3) and is composed of `Statements` (Sect. 4.2.2). Each instance of the `Analysis` concept is transformed into an executable R script, while each instance of `Statement` generates a piece of R code fitting in the script.

2.5.1 Programming Paradigm

Even if it is largely classified as a functional language, R can be considered a multi-paradigm language. It mostly relies on functions, but also has statements, objects, states, and other elements belonging to other paradigms. Such versatility is one of the reasons why many programmers have difficulties with their approach to R.

The way MetaR composes analyses makes a big shift in the programming paradigm from R: *MetaR is a DSL that follows the imperative paradigm*. Each statement is executed and generates results available for the next statements. We found that this step-after-step method of proceeding, with a sequence of statements, is easier to understand and follow for our users.

2.5.2 External Packages

Many of the R capabilities come from packages, the fundamental units of which are created by the R community to share reusable R code. Recently, the official repository (CRAN[3]) has reached 10,000 packages published, and many more are publicly available online. Each package must be independently installed in the local system before being used by R commands.

The R code generated by a `Statement` typically relies on one or more packages. Each `Statement` is configured (by the language designer) with the list of packages it will require at runtime. R commands to install these packages are automatically generated when the `Analysis` is built. In addition, several packages used by MetaR for the analysis and comprehension of biological data are released by the Bioconductor project [9].[4] These packages require a sort of "non-standard" installation, also transparently handled by MetaR.

[3] https://cran.r-project.org/.

[4] https://www.bioconductor.org/.

3 Development and User Community

3.1 Development

Two senior language engineers designed and developed the core languages of MetaR. At different times, rotating students and summer interns joined the team to work on selected parts of the software.

3.2 Target Audience

MetaR's data analysis languages are tailored for users with a diverse range of experience. They can be used by:

- Biologists, clinicians, and other biomedical researchers with limited computational experience, who wish to understand how data analysis works. No programming skills are required to start analyzing data for these users. They represent the majority of users.
- Students, looking for a relaxed entry point to the field.
- Bioinformaticians, who need to perform repetitive analyses and find it beneficial to design and use specialized analyses micro-languages [10] to increase productivity and consistency of data analysis.
- Analysis experts, who wish to create and package state-of-the-art analysis methods into user-friendly MetaR languages.

MetaR is obviously not a replacement for R. It is much less flexible and, mainly, it covers a very small subset of what can be done with R. As stated, its goal is to provide an alternative and gentle introduction to the topics of data analysis. We do not expect that R programmers drop R and start using MetaR. However, they can package their reusable code as MetaR statements, as an additional way to distribute their work to a different audience.

3.3 User Community

Since January 2015, the CTSC has been offering periodical training sessions to introduce data analysis with MetaR to technicians, students, postdoctoral fellows, and faculty who hold an appointment in one of the CTSC institutions. Occasionally, sessions also include participants from other institutions in NYC or visiting fellows. Detailed training material has been developed to support these sessions.

We offer two different types of sessions:

- Hands-on sessions, where participants bring their laptop with MetaR installed to get a hands-on experience during the 2 h of the session

- Demo sessions, where the instructor demonstrates in 1 h how to use MetaR and participants try to determine whether MetaR can help with their projects

As of Summer 2020, more than 40 training events have been held, with a total participation of approximately 300 trainees. Many of them followed up with questions or requests of support and concretely tried to use the tool to analyze their data. These sessions have been (and still are) mutually beneficial: while participants learn about MetaR and get a gentle introduction to data analysis, instructors collect feedback and new requirements. In fact, many features added to MetaR over the years emerged from discussions that occurred during the training events.

4 Requirements, Design, and Architecture

> It is easy to lose sight of a program when looking at all its functions and classes.
>
> Bjarne Stroustrup [11]

Omitting non-essential elements is especially important when the user is a computational beginner. In MetaR, design and implementation choices described in this and the next sections were taken with the principle of keeping the attention of the user on the data analysis.

Examples of application of this principle are found:

- In keeping what the user needs visible within the analysis and taking out details that can be optionally included in other concepts and then imported, as with styles (see Sect. 6.3)
- In the static typing of data sources, in which the structure is anticipated with respect to the execution (see Sect. 4.2) and cannot be changed
- In creating a simplified and monitored execution of the analysis (see Sects. 7.3.1 and 7.3.2)
- In the virtualization of the execution environment, where, once the initial settings are activated, MetaR automatically executes all the analyses in the selected target environment (see Sect. 7.3.3)

4.1 Requirements of the Project

MetaR has been created with the intention of making available to biomedical researchers a *new instrument to learn and perform data analysis through high-level abstractions*. Such abstractions revolved around many stages of the data analysis process, particularly:

– Data sources, in which we model the data and their structure
– Data inspection, in which we check and validate the data

– Data manipulation, in which we change the structure of the data
– Handling of results, in which we hold and maintain the results
– Plotting, in which we generate graphical representations of the data
– Execution, in which we build and execute the analysis
– Virtualization, in which we virtualize the execution environment

Another strong requirement of the project was the creation of a tool blending the boundary between programming/scripting languages and graphical user interfaces. This was deemed critical to support the educational goals of MetaR.

4.2 DSL Design

At its highest level, the design focused on the following aspects of the *problem domain*:

– The input data
– What operates on the input data and how to support new methods
– What is executed
– Execution of the generated code

This section presents the solutions adopted to address these aspects. Each of these solutions was pondered keeping in mind the goal of smoothing the learning curve for beginners. Occasionally, this required favoring simplicity over flexibility.

Section 6 reports use cases that demonstrate how these abstractions work in practice, while Sect. 7.3 covers the execution environment.

4.2.1 Surrogating Data Sources

The data to analyze can be really big and we obviously do not want to represent their entire content in memory. MetaR introduced the term *Data Object Surrogate* (DOS) to describe the way it refers data sources.

A Data Object Surrogate is an object that models a source of data, such as a table or an image.

When a DOS is created, it contains only limited information from the original data source, just enough to facilitate referring the source for the purpose of data analysis.

Since working with RNA-Seq data often requires using tables of data as inputs, the most important DOS is the `Table` concept defined in the `metar.tables` language (Fig. 1).

```
Table MyTableWithData.tsv ...
File Path
  /Users/manuelesimi/MPSProjects/MetaRPaper/Figure1/MyTableWithData.tsv
Columns
  gene_name: string
  gene: string
  SAMPLE1: numeric
  SAMPLE2: numeric
  SAMPLE3: numeric
  SAMPLE4: numeric
  SAMPLE5: numeric
  SAMPLE6: numeric
  SAMPLE7: numeric
  SAMPLE8: numeric
```

Fig. 1 An instance of the `Table` concept as shown in the projectional editor. The user selects the file (a TSV in this case) containing the data by browsing the local file system. Following up the selection, MetaR creates a DOS for the table

Each `Table` instance holds the information to refer and use the table within a MetaR program: location, columns along with the type of their data, and name of the table. In Sect. 6, we will see how to enrich a `Table` with further information that helps analyze the data.

By design, DOSs are *immutable objects*. This means that the information of these objects cannot be changed after the object is created. When changes are required, a new DOS is created. While this does not allow the flexibility of other environments, it makes MetaR analyses easier to understand for unexperienced users.

4.2.2 Operating on the Data

Once we had the data modeled in MetaR, the design focused on the way the user interacts with the data coming from the data sources. We needed an abstraction for the building blocks of the data analysis process, something that takes some inputs (often including one or more DOSs) and generates a result to use in the downstream steps.

These were the premises upon which the abstract `Statement` concept and some supporting interfaces were created. The goal of `Statement` is to model how an inheriting concept delivers a specific functionality and how it will integrate with other statements.

A statement is an abstraction over one or more steps in the data analysis process.

When a user adds a node based on `Statement`, the projectional editor creates the syntax for the statement. To be consistent, we enforced having (most of) the

Fig. 2 Template for syntax of a `Statement` in the projectional editor

```
subset rows MyTableWithData.tsv when true: $(gene_name) != "" -> subsetTable
```

Fig. 3 A sample statement. This instance of the `subset rows` statement takes as input a reference to a `Table` instance and a Boolean condition. It thus creates an output DOS named `subsetTable` containing only the rows matching the condition (gene name not empty, in this example)

`Statements` following the same syntax's template. A typical syntax is shown in Fig. 2.

The keywords in the statement are expressed using a domain language as close as possible to a natural language. For those statements expected to generate an output, an arrow points to what will be the result(s) of the statement after the execution. Of course, the type and cardinality of the inputs/results vary from one statement to another, depending on what the statement does. This mirrors commands executed from the command line in any operating system.

MetaR also has expressions, but they are not first-level concepts. Rather, they can be created as language elements to input to statements wherever they are accepted. Figure 3 shows an instance of the `subset rows` statement operating on the table in Fig. 1 that makes use of the `Expression` concept.

Eventually, a `Statement` node generates a snippet of R code that, once executed, creates the expected results and makes them available to other statements' R code.

4.2.3 Analyses

The next step in the design was to decide how to hold together the `Statements` nodes in a cohesive way to then generate the R script to execute. This choice had the greatest impact because it would be the entity that a user would interact the most with, ultimately determining the level of interactivity and productivity of MetaR.

MetaR offers the `Analysis` concept as a means of collecting a set of `Statement` nodes that together express how data is to be analyzed. An instance of `Analysis` takes care of creating a consistent environment to build the data analysis steps and to control the subsequent code generation and execution phases.

An analysis often imports one or more DOSs, performs data transformations, and outputs some generated DOSs as tables or plots.

```
Analysis MakeUpperCase
{
  import table MyTableWithData.tsv
  transform table MyTableWithData.tsv -> upperCaseGenes {
      make uppercase values gene_name
  }
  write upperCaseGenes to "UpperCaseGenes.tsv" ...
}
```

Fig. 4 An `Analysis` instance as shown in the projectional editor. 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. This analysis works with the table of data presented in Fig. 1

Figure 4 shows an `Analysis` instance with three `Statements` nodes:

- `import table`, which adds a table to the analysis and makes it available for the next statements
- `transform table`, which applies a small change to the values in a column of the table and outputs a new table
- `write`, which serializes the transformed table on the file system

The `Analysis` interface can also display buttons, images, and other visual objects directly as part of the language. This feature takes advantage of the ability of JetBrains MPS to embed arbitrary graphical elements in the projectional editor. Figure 4 shows one of these elements: the button next to the `write` statement allowing for selection of the destination folder to save the serialized table. More of these elements are described in Sect. 7.1.

4.3 Other Relevant Design Aspects

As mentioned, MetaR puts great emphasis on the impact of the design choices on the user experience. In this regard, DOSs and statements have been designed with extra features to further simplify the writing of the analysis. These features are often supported by MPS built-in mechanisms.

4.3.1 Auto-completion and Scope

Auto-completion plays a crucial role in MetaR and is deeply exploited at several levels to improve the user experience. It is through auto-completion that MetaR achieves many of its educational and learning purposes, as it avoids almost any knowledge of the DSL syntax prior to its usage.

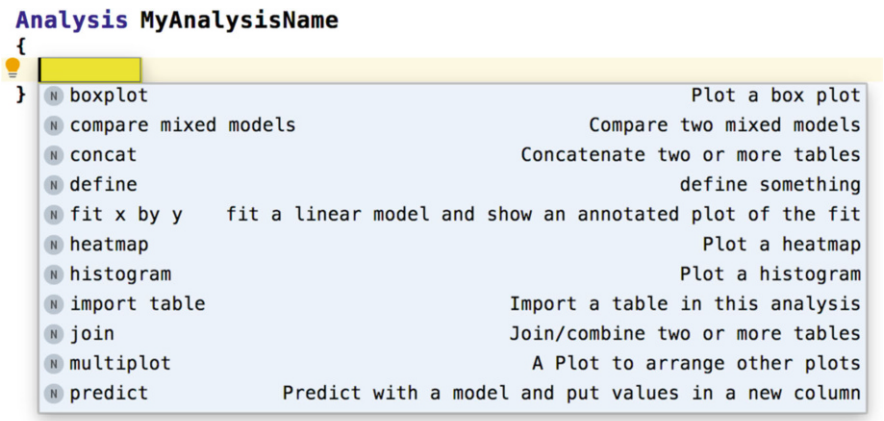


Fig. 5 Auto-completion inside the `Analysis` node. The figure shows what happens when auto-completion is triggered on an empty line, suggesting `Statement` elements that can be inserted at the cursor point

Auto-completion is strictly connected with the notion of *scope*, the list of language elements that can be referred in each statement of the analysis. This list depends on the *lexical environment* that coincides with the previous statements, the languages imported in the model, and the constraints defined for the reference.

For instance, a table may be suggested as input in a particular point if the table has been made available by a previous statement. In Fig. 4, the `preview table` statement can take the table `MyTableWithData.tsv` as input because `import table` adds the table to the lexical scope.

In a freshly (and empty) created instance of `Analysis`, auto-completion proposes the list of available `Statements` (Fig. 5).

Users can change the default lexical environment with custom-defined regions (or *blocks*) of `Statement` nodes. These regions are nested inside the main `Analysis` block and are especially useful in case of analyses with long sequences of statements or many DOSs. A block inherits the lexical environment of the parent block and its language elements are not visible outside the block itself.

This feature enables less experienced users to work with a smaller set of references. We found this option particularly useful during our training sessions (see Sect. 3) (Fig. 6).

4.3.2 Future DOSs

One of the advantages of MetaR compared to R is its capability to "predict" the structure of a DOS resulting from a statement *before* actually executing the statement (hence the entire analysis). This type of DOS is called *future* DOS. It allows for the tracing of all its possible transformations in advance and thus enables

(a)

```
{
  import table MyTableWithData.tsv
  import table DataInsideScope.tsv

  subset rows DataInsideScope.tsv when true: $(gene) != "" -> subset

}
```

(b)

```
{
  import table MyTableWithData.tsv
  // define a nested lexical scope
  {
    import table DataInsideScope.tsv
  }
  // end of the scope

  subset rows DataInsideScope.tsv when true: $(<no name>) != "" -> subset

}
```

Fig. 6 Lexical scope. The figure shows how a custom-defined scope impacts on visibility. In the two analyses, we import the same two tables. In the first analysis (**a**), the `subset rows` statement can reference both tables because they are both part of its lexical environment. In the second analysis (**b**), the second table is imported in a custom defined scope; outside of the nested block, this table is not visible and therefore cannot be used by statements

the creation of the entire data analysis at once. There is no need to write a statement, build, run, check the results, and then go back to the editor to write the next statement. Statements are simply created and added to the analysis one after the other. This results in saving a lot of time and, mainly, keeping users focused on the data analysis' logic with fewer distractions (Fig. 7).

4.4 Structure of the Project

From the implementation point of view, MetaR is a vast project composed of several modules that in turn relies on an ecosystem of plugins.

Table 1 shows how the modules of the project are classified.

The complexity of the languages fluctuates from one to another. Core languages, such as `metar.tables`, can have hundreds of concepts and several supporting interfaces. Others have a few concepts that provide a specific extension or feature to reuse in the core DSL. Solution modules usually collect Java code invoked from the *behavior*, *constraints, typesystem*, and *generator* aspects of concepts.

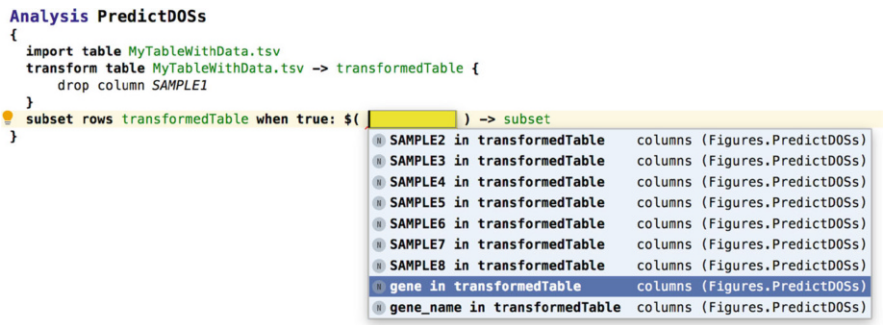


Fig. 7 Predicting the output of a statement. In this analysis, the original imported table is transformed and one of its columns is removed from the table with the `transform table` statement. The output table is then used by the `subset rows` statement. Here, when auto-completion is triggered to list the columns of the table, the dropped column (*SAMPLE1*) is not suggested. This is a consequence of the output prediction applied to the `transform table` statement

Table 1 MetaR modules

Module type	# of modules
Language	31
Solution	3
Plugin Solution	4
Build Solution	1
DevKit	1

The Plugin Solutions extend the IDE to improve the user experience. These are discussed in Sect. 7.

Finally, the remaining two modules are dedicated to the tool distribution. The Build Solution uses the DSL defined in the MPS Build Language[5] to automate the building and packaging of the MetaR plugin and make it compatible with the specifications of the JetBrains Plugins Repository (see Sect. 8). The DevKit module is a convenient way to group modules used during the training events (see Sect. 3).

5 Language Composition

MetaR has been developed in a research community, where collaboration prevails (or should) over competition. For this reason, MetaR offers a means to contribute to the project with new features that can be plugged in the DSL with minimal effort by advanced users. This is possible thanks to the very particular way JetBrains MPS maintains and manipulates the language definition.

[5] https://www.jetbrains.com/help/mps/build-language.html.

Differently from other language workbenches and development environments, MPS maintains the language definition in an Abstract Syntax Tree (AST) [12]. Working directly with the AST has the advantage of facilitating the creation of *composable languages* and making them seamlessly integrate.

Language composition has been a pillar of MetaR since its inception. Two languages compose when elements of language B can be used or referred by elements of language A. Since MPS works with ASTs, this translates to attaching nodes defined in B as children of a node defined in A. Favoring composition makes it possible to create micro-languages [10] that integrate directly with the core DSL.

The design of the elements to compose in MetaR builds on top of a previous work [13] where we experimented and assembled our experience with language composition and MPS.

Inheritance (or generalization) relationships across concepts of two languages are the primary mechanism used to compose these languages.

The structure of a language in MPS is defined by a set of concepts. Mirroring what is available in object-oriented programming, a concept can extend another concept and implement multiple concept interfaces. On the other hand, interfaces can extend several other interfaces. All these relations permit the modeling of complex contracts among concepts, achieving language composition.

What can be primarily composed in MetaR are statements inside the analysis. Or at least this is the type of composition most visible to the end user. Several concepts and interfaces in the `metar.tables` language define the rules for code editing and rendering of new statements. These rules mainly affect how statements are expected to behave, take their inputs, expose their outputs, and generate the R code to ultimately execute the analysis of the data.

Defining a new `Statement` concept compliant with these rules makes it immediately available to be used inside an analysis, no matter in which language it is defined: it just needs to be part of the lexical environment. New statements simply and seamlessly compose with MetaR through language composition without any intervention in the host language(s).

With these techniques, several *micro-languages* have been created and incorporated in MetaR. We call a language a micro-language when it has a very small structure (sometimes even just one concept) offering extensions to MetaR core DSL. Each of them provides abstractions designed for a specific purpose (typically, new statements). When combined through language composition, micro-languages provide complementary features and make it possible to write richer MetaR analyses.

One of the most prominent examples of a micro-language is the BioMart language (see Sect. 6.4).

6 Working with MetaR Languages

In this section, we walk through a selection of cases to demonstrate some usage scenarios for MetaR. Most of them use tables with the following structure as input DOSs:

1) Most of the columns represent biological samples.
2) One of the columns is the gene name or identifier (or both).
3) Each row includes information about a single gene.
4) Each value in the matrix is the number of reads mapped to each gene for the sample.

Tables with this type of information are called *tables of counts*.

6.1 Data Annotation and Manipulation

Managing the various input objects used in data analysis can be challenging. Each step of the process might need to refer to these objects and their structure several times. Occasionally, different steps access the data in completely different ways. Section 4.2.1 introduced Data Object Surrogates as a means of creating a uniform model for input data sources.

As a further step toward harmonizing the access to data, MetaR offers a mechanism to hand in data to statements in a simplified way. The mechanism is called *data annotation* and, as its name suggests, allows metadata to be added to a data source. There are two types of metadata:

- labels: assigned to columns and used to replace list of columns' names
- usages: assigned to labels to group them for special purposes

Labels are especially beneficial for those statements that manipulate the structure of data sources and require referring multiple columns. These statements change the structure of Data Object Surrogates to enable a more convenient consumption and further data analysis with other statements. To a certain extent, each data manipulation statement can be viewed as an implementation of the Adapter Pattern [14].

Usages are required by some statements to select labels. For instance, the `heatmap` statement (see **UC5**) requires the usage *heatmap* to be associated with the groups of samples to compare.

In this first set of cases, we demonstrate how annotations are exploited by some statements performing data manipulation on tables.

The main advantage of data annotation is accessing the data through metadata instead of the actual structure of the data source.

```
Column Labels and Usages

Define Usages:
  << ... >>

Define Labels:
  ToDelete used for << ... >>
```

Fig. 8 Defining labels for a table. This instance of `ColumnLabelContainer` defines a single label named *ToDelete*

```
Table MyTableWithData.tsv [...]
File Path
  /Users/manuelesimi/MPSProjects/MetaRPaper/Figure1/MyTableWithData.tsv
Columns
  gene_name: string
  gene: string
  SAMPLE1: numeric
  SAMPLE2: numeric
  SAMPLE3: numeric
  SAMPLE4: numeric [ ToDelete ]
  SAMPLE5: numeric
  SAMPLE6: numeric
  SAMPLE7: numeric
  SAMPLE8: numeric [ ToDelete ]
```

Fig. 9 Using labels to annotate a table. In this `Table` instance, two columns are labeled with *ToDelete*. Multiple labels can be associated with each column and auto-completion is available to insert labels next to the column's type

```
import table MyTableWithData.tsv
transform table MyTableWithData.tsv -> transformedTable {
    drop columns which have label ToDelete
}
```

Fig. 10 Transforming a table using labels. The `transform table` statement deletes two columns (SAMPLE4 and SAMPLE8) which have been labeled with *ToDelete* without using their name

UC1

Starting from the table in Fig. 1, we are going to enrich it with metadata and show how statements benefit from them. The `ColumnLabelContainer` concept in MetaR allows for the definition of labels in the model and, optionally, groups of labels for a specific purpose with `Usages`. Figure 8 shows an instance of the concept.

In Fig. 9, we again present the table from Fig. 1, this time annotated with the label defined in Fig. 8.

We are now going to demonstrate how to drop the columns from the table using the label *ToDelete*. The snippet in Fig. 10 shows an instance of the `transform table` statement that refers the columns to drop via the selected label instead of listing them one by one with their name.

```
Table MySecondTableWithData.tsv ...
File Path
  /Users/manuelesimi/MPSProjects/MetaRPaper/Tables/MySecondTableWithData.tsv
Columns
  gene_name: string
  gene_id: string [ ID ]
  SAMPLE9: numeric
  SAMPLE10: numeric
  SAMPLE11: numeric
  SAMPLE12: numeric
  SAMPLE13: numeric
  SAMPLE14: numeric
```

Fig. 11 A table with one label. This table has five columns with samples and one column (*gene_id*) with a single annotation

```
import table MyTableWithData.tsv
import table MySecondTableWithData.tsv
join ( MyTableWithData.tsv , MySecondTableWithData.tsv ) by label ID -> joinedTable
```

Fig. 12 Joining tables by label. The two imported tables have common information in columns with different names. The *ID* label assigned to both columns is used by the `join` statement to merge the two tables

In this example, we have only two columns to drop, and listing them by name would not be difficult. However, tables may have tens or hundreds of columns. Avoiding listing all their names and replacing such long lists with just a single label drastically reduces the time needed to write the analysis and greatly improves its readability.

UC2

We show another way to exploit data annotation, this time across tables. Given two tables with a different structure, we want to merge them. For this case, the first table is the one in Fig. 9, with an additional *ID* label on the *gene* column, while the second table is presented in Fig. 11.

Comparing this table with the one in Fig. 9, we notice that the columns with the gene identifiers are named differently in the two tables (*gene* vs *gene_id*). This is a very common situation where tables with experimental results come from different sources. With data annotation, MetaR can easily work around this inconsistency by assigning the same label to these columns. Figure 12 illustrates a statement joining the tables in Figs. 9 and 11 through the label *ID*.

UC3

In this last data manipulation use case, we demonstrate how to alter the order of the columns in a table. In Fig. 13, we have a table with two groups of columns, each of three samples, identified by labels.

Figure 14 shows an instance of the `reorder columns` statement changing the order of the columns in the table by moving them up or down.

```
Table MyCleanedTableWithData.txt ...
File Path
  /Users/manuelesimi/MPSProjects/MetaRPaper/Tables/MyCleanedTableWithData.txt
Columns
  gene_name: string
  gene: string [ ID ]
  SAMPLE1: numeric [ Sample1-3 ]
  SAMPLE2: numeric [ Sample1-3 ]
  SAMPLE3: numeric [ Sample1-3 ]
  SAMPLE5: numeric [ Sample5-7 ]
  SAMPLE6: numeric [ Sample5-7 ]
  SAMPLE7: numeric [ Sample5-7 ]
```

Fig. 13 Table with two groups of columns identified by labels

```
{
  import table MyCleanedTableWithData.txt

  reorder columns in table MyCleanedTableWithData.txt

  UP DOWN label Sample5-7
  UP DOWN label Sample1-3
  UP DOWN label ID
  UP DOWN label NoLabel
  -> reorderedTable

}
```

Fig. 14 Reordering columns by label. By clicking on the buttons, groups of columns from Fig. 13 matching the labels are moved up or down in the order. The columns in `reorderedTable` will have the following order: *SAMPLE5, SAMPLE6, SAMPLE7, SAMPLE1, SAMPLE2, SAMPLE3, gene, gene_name*

6.2 Data Processing and Visualization

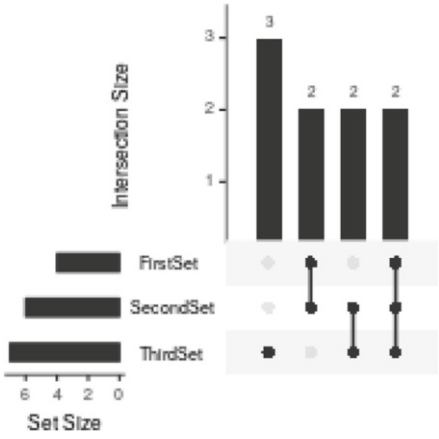
In this section, we present two usages of MetaR performing some typical steps of data analysis. They both process data from input sources and visualize the results in a graphical form. In data analysis, such visual representations of the data are called *plots* and allow them to be analyzed from angles that are not clear in unstructured data.

UC4
A very frequent task is to intersect multiple sets of identifiers to understand which ones are shared among them. Figure 15 demonstrates how to create an UpSet plot [15] that graphically shows the intersections across three sets.

```
{
  define Set of IDs FirstSet {
    GeneA GeneB GeneC GeneD
  }
  define Set of IDs SecondSet {
    GeneA GeneB GeneC GeneD GeneE GeneF
  }
  define Set of IDs ThirdSet {
    GeneA GeneB GeneE GeneF GeneG GeneH GeneI
  }
  UpSet { set FirstSet  } -> intersection
          set SecondSet
          set ThirdSet
  multiplot -> preview [ 1 cols x 1 rows ]
```

Hide preview

```
 [ intersection ]
```




```
}
```

Fig. 15 Intersecting sets. Given three sets of identifiers, manually defined with the `define set` statements, the `UpSet` statement intersects them and creates a plot showing how many IDs are shared across the sets

Sets of identifiers are usually loaded from DOSs (instead of being manually defined) and they can include up to hundreds of thousands of IDs. UpSet plots do not report which the shared identifiers are, only how many. Other statements (not shown) are available to list and work with them.

```
Column Labels and Usages

Define Usages:
  AllSamples
  heatmap

Define Labels:
  Sample1-3 used for AllSamples heatmap
  Sample5-7 used for AllSamples heatmap
  ToDelete used for << ... >>
  counts used for << ... >>
  ID used for << ... >>
```

Fig. 16 Labels and usages for differential expression tests. *Sample1-3* and *Sample5-7* identify the two groups of samples to compare, each composed of three samples. *counts* marks the columns with read counts and *ID* sets the column with the unique identifier for each gene

This use case also introduces a feature appreciated by many MetaR users: the `multiplot` statement. This statement is an example of the capabilities of MetaR to mix graphical elements within the text. `multiplot` creates a preview of one or more plots immediately visible to the user right after the statements that generate the plots. This adds significant assistance to the process of writing the analysis steps.

UC5

A gene is considered differentially expressed if an observed difference or change in read counts or expression levels between two experimental conditions is statistically significant. R has very good packages that implement strategies to calculate when genes are differentially expressed. In this use case, we demonstrate how to perform analysis of gene expression on a table of counts using a method called Limma Voom [16].

Figure 16 shows the labels and usages we need for our analysis.

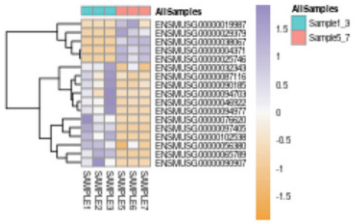
The analysis operates on the same table and annotations in Fig. 13, with an additional label *counts* on the columns marked with *Sample1-3* and *Sample5-7*.

Figure 17 shows how to invoke Limma Voom to perform a differential expression test on the two groups of samples in our table. An instance of the `limma voom` statement is configured with a model (with one factor: *AllSamples*) to call differences between columns labeled with *Sample1-3* and *Sample5-7*. Limma Voom uses this model to define the mean and variance of the data. It is possible to define complex linear models by typing + followed by the name of a usage attached to the counts table and repeat to add multiple factors to the model.

```
{
  import table MyCleanedTableWithData.txt
  // run a diff expression test
  limma voom counts= MyCleanedTableWithData.txtmodel: ~ 0 + AllSamples
    comparing Sample1-3 - Sample5-7 -> stats: resultsTable normalized: default
  // filter the results according to values in a column
  subset rows resultsTable when true: $(logFC) <= -4.5 | $(logFC) >= 4.5 -> subsetTable
  // join the results with the original table
  join ( subsetTable, MyCleanedTableWithData.txt) by label ID -> joinedTable

  heatmap with joinedTable select data by one or more label Sample1-3, label Sample5-7
    -> heatmap no style [
    show names using group ID
    annotate with these groups:AllSamples
    scale values: scale by row
    cluster columns: false cluster rows: true
  ]

  multiplot -> preview [  1 cols x 1 rows ]  Hide preview
   [ heatmap ]
```




```
  // save the heatmap to an external PDF file
  render heatmap as PDF named "heatmap.pdf"  ...  no style
}
```

Fig. 17 Analysis for differentially expressed genes. Given two groups of samples identified with the labels *Sample1-3* and *Sample5-7*, the `limma voom` statement finds differentially expressed genes by comparing the two groups and then stores the results in `resultsTable`. The `subset row` and `join` statements do some data processing by filtering the results and joining them with the original table of counts. Finally, a heatmap plot is created to visualize the counts of the genes that are differentially expressed; the plot is saved in a PDF format with the `render` statement

The sequence of statements in Fig. 17 shows a typical pattern in statistical data analysis:

- A statistical method is executed over the data.
- Data are manipulated (first to filter them and then to create more suitable structures for the next steps).
- The results are visualized in a plot.
- The plot is saved on an external file as image, ready to be included in a talk or scientific manuscript.

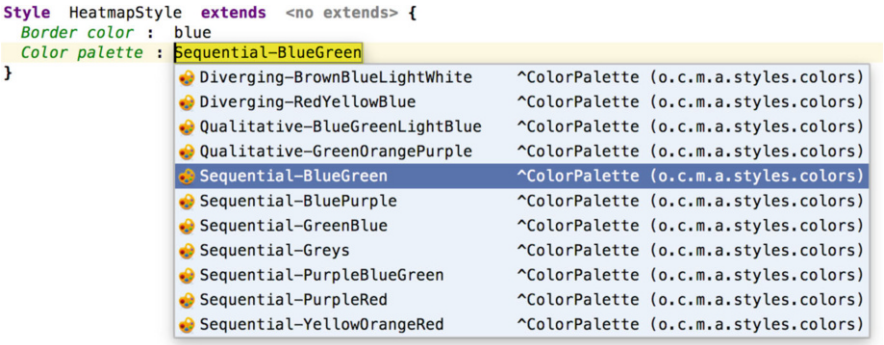


Fig. 18 A style for a heatmap. This `Style` instance defines the color of the border in the plot and the color palette to use

6.3 Styles

There are many aspects of a graphical representation that can be customized. In R, each instruction has its own way (i.e., input parameters) to create such customizations, and this can be very confusing for beginners. MetaR offers the `metar.styles` language to define *styles* to bind to plots. The goal of this language is to shape a uniform way to customize any type of plot.

UC6
The `Style` concept in the `metar.styles` language is the entry point for customizing plots. An instance of this concept can be optionally attached to a statement that outputs a plot and helps to define its visualization.

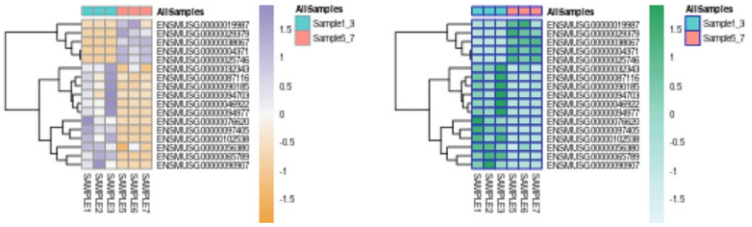
Figure 18 illustrates an example of a `Style` instance. Many pre-configured graphical settings are distributed with `metar.styles`. Color palettes listed in the auto-completion menu are just an example of these. Nevertheless, users can create their own and/or extend existing styles.

Figure 19 shows what happens when the style selected in Fig. 18 is applied to the heatmap plot created in **UC5**.

6.4 Interactive Statements

By combining the *editor*, *behavior*, and *constraints* aspects of a concept, it is possible to add *interactive statements* to MetaR. Interactive statements improve the user experience by adding a dynamic facet to the analysis. Their goal is to extend auto-completion to what is available in the lexical scope, by searching for language elements even outside it.

```
{
  heatmap with joinedTable select data by one or more label Sample1-3, label Sample5-7
    -> heatmapWithStyle HeatmapStyle [
    show names using group ID
    annotate with these groups:AllSamples
    scale values: scale by row
    cluster columns: false cluster rows: true
  ]
  multiplot -> preview [  2 cols x 1 rows ]  Hide preview
   [ heatmap ]                          [ heatmapWithStyle ]
```




```
}
```

Fig. 19 Heatmaps with different styles. The style in Fig. 18 is associated to a copy of the heatmap created in Fig. 17. When auto-completion is triggered next to the `heatmapWithStyle` plot, `Style` instances in the lexical scope are prompted for selection. The `multiplot` statement in the figure shows both heatmaps next to each other to appreciate the differences due to their different styles

UC7

BioMart is a web service accessed through the home page of Ensembl.[6] It can be used to query a collection of Ensembl databases for ID mapping and feature extraction. The `metar.biomart` language is dedicated to interfacing the BioMart services.

The language provides the interactive `query biomart` statement to assemble specific queries to retrieve data. As the user auto-completes the information in the projectional editor, the statement sends queries to the remote BioMart services to provide values (wrapped on the fly inside language elements) for the next input according to the selected information.

Each part of the statement auto-completes based on the previous selected inputs. In Fig. 20, the list of datasets in the drop-down menu is retrieved from the database *Ensembl Genes 102* as soon as it is selected in the first part of the statement. In the same way, the next input will propose the list of attributes retrieved for the selected dataset (Fig. 21).

The output of `query biomart` is a DOS (table) with the results of the queries. As for other future DOSs (see Sect. 4.3.2), this is just an anticipation of the structure

[6] https://www.ensembl.org/.

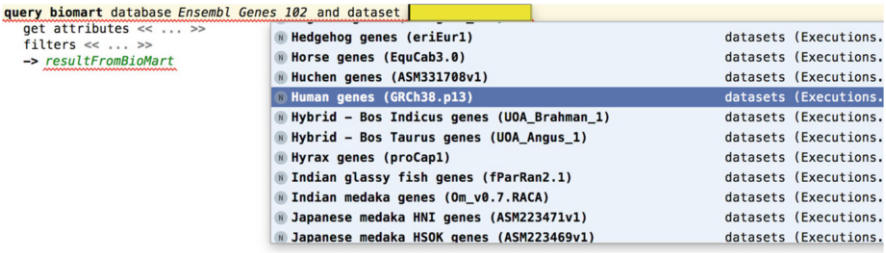


Fig. 20 Querying a remote service. When auto-completion is triggered, the `query biomart` statement interfaces with the remote BioMart services and creates the appropriate language elements to propose for selection

```
// get the gene names from BioMart
query biomart database Ensembl Genes 102 and dataset Human genes (GRCh38.p13)
   get attributes Gene name from feature of types string with column group annotation ID
   filters << ... >>
   -> resultFromBioMart
```

Fig. 21 A complete `query biomart` instance

of a table with data returned by the remote BioMart services when the analysis is executed.

7 Advanced Exploitation of MPS

JetBrains MPS is not only a language workbench to create DSLs, it also provides a wide range of additional instruments to build a comprehensive working environment around the DSL itself.

MetaR leverages some of these MPS features to make its delivered functionalities and the user experience unique. This section presents some notable capabilities of MetaR that would not have been achieved with any other language workbench than MPS.

7.1 Graphical Elements

In MetaR, graphical elements are mixed with statements in the `Analysis` editor. We have already encountered some of these elements in Sect. 6. These inclusions are possible thanks to the capabilities of MPS to embed Java Swing components inside the editor's cells.

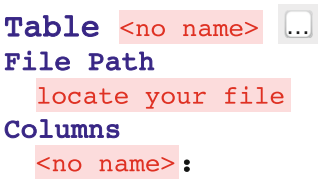


Fig. 22 Table editor with Browse button. The button, placed next the <no name> red label, opens a file selection dialog to locate the file to configure the `Table` instance

Graphical elements in MetaR have two different functions:

1) Triggering actions—these actions typically support the statement's purpose.
2) Displaying results—the results coming from one statement or multiple statements are shown in between the statements.

On a small scale, the second function can be compared to Jupyter Notebooks[7] where snippets of code generate results inside a notebook's cells.

7.1.1 Buttons

MetaR comes with a language (`metar.ui`) for adding cells with buttons to the projectional editor and associating actions with them. We have already seen some buttons next to statements to show/hide/move cells in the editor (see Figs. 14, 15, and 17).

The `Table` (Sect. 4) editor uses buttons differently. In this case, the action associated with the button is to open the File Open Dialog box on the local system and allow for locating and selecting the file with the table (Fig. 22).

7.1.2 HTML Tables

Sometimes it is useful to visualize a snapshot of a table's content directly inside the editor. The `preview table` statement allows for the creation of a small customized preview by embedding an HTML table inside a cell. Columns and rows of this preview can be resized as desired as shown in Fig. 23.

7.1.3 Auto-refreshable Images

In Figs. 15 and 17, we saw how the `multiplot` statement displays images inside one or more cells in the editor. The same images are also available as preview in the inspector of the concept.

[7] https://jupyter.org/.

```
{
  preview table MyTableWithData.tsv [  5 cols x 6 rows ]
```

Hide preview DefaultStyle

	gene_name	gene	SAMPLE1	SAMPLE2	SAMPLE3
1	0610012D04Rik	ENSMUSG00000089755	0.00	1.21	0.00
2	1010001N08Rik	ENSMUSG00000097222	1.00	0.00	0.00
3	1200007C13Rik	ENSMUSG00000087684	0.00	1.21	0.00
4	1500015O10Rik	ENSMUSG00000026051	221.96	1253.98	23.77
5	1500035N22Rik	ENSMUSG00000059631	9.95	6.07	5.78
6	1700001K19Rik	ENSMUSG00000056508	0.00	0.00	0.00

```
}
```

Fig. 23 Table with in-line HTML preview. A snapshot of the table's content is visualized by the `preview table` statement inside a Java Swing component

Images in MetaR are auto-refreshable. This means that whenever the image changes, it is automatically reloaded in the cell. To achieve this result, MetaR uses a Project Plugin. This type of MPS plugin provides a way to integrate Java code with the IDE functionalities. They are created with the `jetbrains.mps.lang.plugin.standalone` language and held inside solutions.

The concept in charge of displaying a specific image also registers a listener of the image file in the plugin. The plugin makes sure that the listener subscribes on the file exactly once and is triggered at the right time. When the analysis is executed and a change is detected, the listener invokes the Java code inside another solution and the image is automatically refreshed in all of the cells where it is currently loaded.

7.2 Table Viewer Tool

Tools are extensions of the original MPS IDE. They provide customized views to open with the context menu. The MPS plugin language (`jetbrains.mps.lang.plugin`) supports the creation of plugin solutions that define new Tools.

The MetaR Table Viewer is a Tool associated with the `Table` concept and its descendants. The viewer adds to the MPS interface the capabilities to load the table's content and show it in a graphical context inside a view. Wherever a table

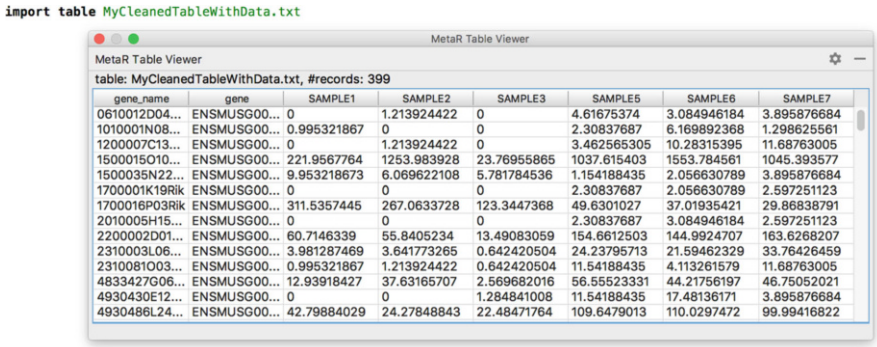


Fig. 24 Table Viewer Tool. The tool is immediately available for those tables directly loaded from the file system (e.g., `Table` nodes) and their references (like the `import table` statement in the figure). Other tables (future DOSs; see Sect. 4.3.2) become visible after the analysis has been run and the content of the table has been created. In this latter case, the tool is available after the first execution of the analysis

name appears (in a `Table` or an `Analysis` node), the tool can be opened to see the rows and columns of that table along with their values. Rows are dynamically loaded as the user scrolls down the content (Fig. 24).

7.3 Execution

MPS provides *Run configurations* (`jetbrains.mps.execution.configurations`) to define how to execute processes starting from selected nodes in the language. MetaR uses these configurations to run the R scripts that it generates from the `Analysis` node.

7.3.1 RunR Configuration

The RunR configuration is a plugin solution in MetaR for the execution of analyses. By right-clicking on any descendant node of the `Analysis` node, it is possible to trigger the RunR configuration from the context menu. A setting editor dialog for the configuration is opened to customize the execution.

When the configuration is started, it first builds the current model and then runs the generated R script for the analysis. The configuration includes *Commands* (defined with the `jetbrains.mps.execution.commands` language) to invoke the R runtime installed on the local system with the proper parameters (including, of course, the location of the R script).

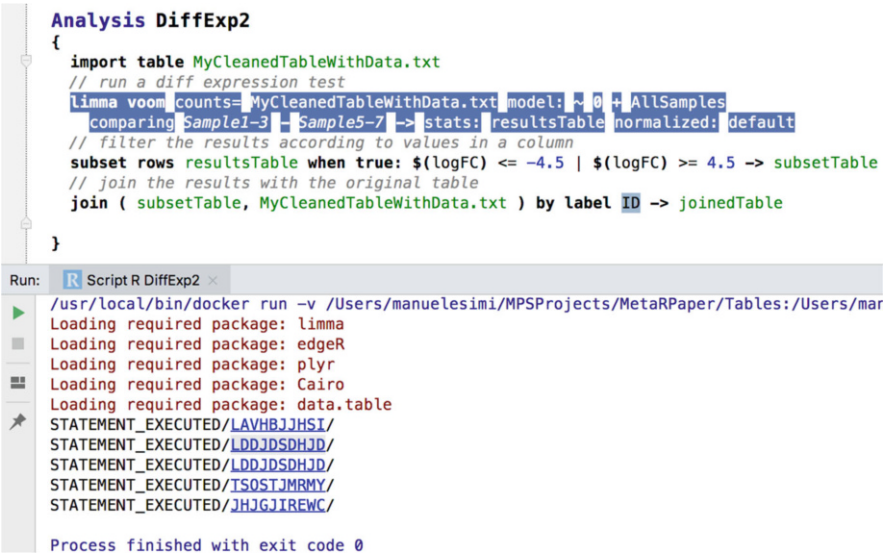


Fig. 25 Monitoring the Execution. Users can follow the execution as statement IDs are printed and linked to statements in the Run view. When the user clicks on the link, the target statement is highlighted in the editor

7.3.2 Monitored Execution

MetaR users are not experts at debugging problems in a program, however, and in any execution things can fail for several reasons. Since the R script is not directly visible, MetaR offers a way to monitor the progress of the execution and understand which statement generates an error, if one occurs.

Each statement has an ID assigned, which is basically the identifier of the node. As the execution advances, the statement IDs are sent to the Run view (see Fig. 25) and printed. Inside the view, the text is properly linked to the node in the AST with the same ID. Whenever an error occurs (also printed in the Run view), the user can click on the link and get to the root of the problem (the statement that generates the error messages).

This is obviously not a debugging tool, yet it gives users an idea of where to investigate an execution problem.

7.3.3 Integration with Container Technology

Reproducibility of results is a key point in research. The scientific community does not accept or consider valid results that cannot be reproduced. This also applies to data analysis: if the same analysis is executed with the same inputs several times, it must yield exactly the same results.

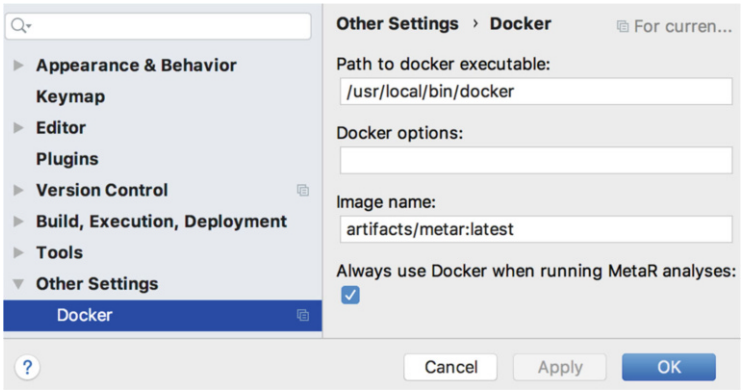


Fig. 26 MPS preferences with Docker. In the preferences window of MPS, it is possible to set in which Docker image MetaR analyses are executed

Any software application (including MetaR) needs some sort of runtime support from other software. This is critical in terms of reproducibility, because these external dependencies introduce a level of uncertainty that is difficult to control.

Several years ago, container technology was introduced to provide fully capable execution environments on any computer supporting the technology. There is no need for the user to deal with installation of libraries and dependencies, downloading packages, messing with configuration files, etc.; everything is made available in a single package called *image*. MetaR integrates with Docker [17], the most popular container technology. Extensions to default MPS configuration settings have been created in MetaR to set the required information.

When options in Fig. 26 are set, each MetaR analysis becomes a *containerized application*; it is automatically executed inside the virtual environment created starting from the selected Docker image. This small and lightweight environment, called *container*, guarantees that whenever and wherever the analysis runs, it will always use the same software packaged in the image. Typically, the R runtime and common R packages are deployed in the image. From the user point of view, once the checkbox in Fig. 26 is checked, it is handled transparently and seamlessly by MetaR.

8 Distribution

MetaR is an open source software distributed under the terms of the Apache License, Version 2.0, and available on GitHub.[8] Its installation is straightforward since

[8] https://github.com/MetaR-Languages/MetaR.

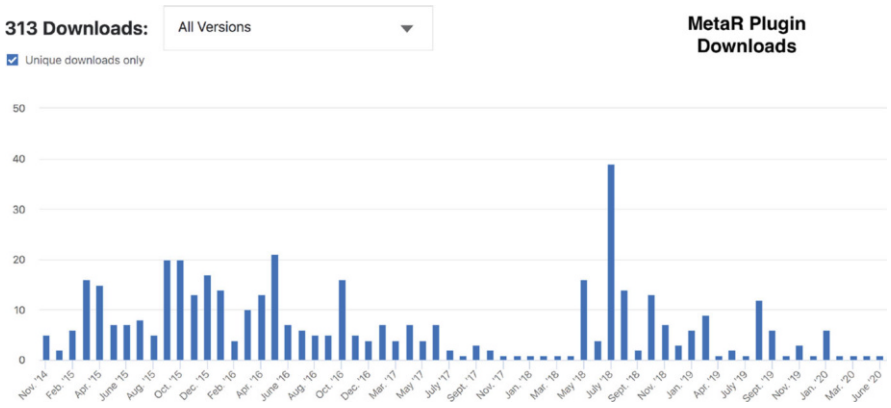


Fig. 27 MetaR Plugin Unique Downloads. The figure shows the total unique downloads of the MetaR plugin since its first release as recorded by the JetBrains Plugins Repository. Unique downloads do not include multiple downloads from the same user or host

MetaR is distributed as a plugin available in the JetBrains Plugins Repository.[9] All of its dependencies are automatically resolved and installed with the MetaR plugin.

Figure 27 shows the statistics of unique downloads of MetaR over the years. These statistics do not include the installations performed during our training events.

9 Conclusions

MetaR exploited JetBrains MPS in many ways. The generation of code in a target language (the R language) simplified the implementation by removing the need to develop our own language runtime system for the DSL. The possibility to extend the IDE with custom tools/plugins served the requirements of the project very well. The Build Language provided a convenient way to package the software as MPS plugin and to manage its dependencies. The MPS Plugins Repository, along with its automatic dependency resolution among plugins, saved us from creating our own distribution website and spared our users from numerous tedious installation instructions. Above all, by coordinating and putting together the features offered by each aspect of the language definition, we achieved a homogeneous approach for many steps of the data analysis process.

Graphical elements with scripting, auto-completion, high-level abstractions over data and instructions, language composition, automatic installation of dependencies for each individual statement, extensions to the environment, seamless integration with external technologies, and solutions to package and distribute the software all

[9] https://plugins.jetbrains.com/plugin/7621-org-campagnelab-metar.

smoothly combined in MetaR to serve the purpose of the project. We were able to achieve everything we planned and beyond, and to create languages with a runtime support somehow unique in the DSL landscape.

The main lesson learned from this experience is that biologists and clinicians can use the tools of bioinformatics and get closer to data scientists. And when different profiles can speak (almost) the same computational language, misunderstanding is reduced and the speed of a research project is greatly enhanced. In our data-rich age, MetaR has proved to be an educational bridge between these two worlds.

Acknowledgments Work reported in this chapter was supported by the National Center for Advancing Translational Science of the National Institute of Health under awards number UL1RR024996 and UL1TR002384. Additional support was provided by the National Institute of Health NIAID award number 5R01AI107762-02 to Fabien Campagne.

References

1. Campagne, F., Digan, W.E.R., Simi, M.: MetaR: simple, high-level languages for data analysis with the R ecosystem. (2015). [Online]. https://doi.org/10.1101/030254
2. Simi, M.: MetaR Case Study. (2020). [Online]. https://resources.jetbrains.com/storage/products/mps/docs/MPS_MetaR_Case_Study.pdf
3. Erdweg, S., van der Storm, T., Völter, M., et al.: The state of the art in language workbenches. In: Lecture Notes in Computer Science. (2013).
4. Dmitriev, S.: LOP – Language Oriented Programming: the next programming paradigm. (2004). [Online]. https://resources.jetbrains.com/storage/products/mps/docs/Language_Oriented_Programming.pdf
5. Di Gesù, V.: Data analysis and bioinformatics. In: Pattern Recognition and Machine Intelligence. PReMI 2007. Lecture Notes in Computer Science, vol. 4815. (2007).
6. Chu, Y., Corey, R.D.: RNA sequencing: platform selection, experimental design, and data interpretation. Nucleic Acid Therap. **22**(4), 271–274 (2012)
7. Ihaka, R., Gentleman, R.: R: a language for data analysis and graphics. J Comput Graphical Stat. **5**(3), 299–314 (1996)
8. Fowler, M.: Domain-specific languages. Addison-Wesley (2010)
9. Gentleman, R., Carey, V., Huber, W., Irizarry, R., Dudoit, S.: Bioinformatics and computational biology solutions using R and Bioconductor. Springer, Berlin (2005)
10. Bentley, J.: Programming pearls: little languages. Commun ACM. **29**(8), 711–721 (1986)
11. Stroustrup, B.: Programming: principles and practice using C++, 2nd edn. Addison-Wesley Professional (2014)
12. Aho, V.A., Lam, S.M., Sethi, R., Ullman, D.J.: Compilers: principles, techniques, and tools, 2nd edn. Addison-Wesley Longman, Boston, MA (2006)
13. Simi, M., Campagne, F.: Composable languages for bioinformatics: the NYoSh experiment. PeerJ. **2**(562), e241 (2014)
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software, pp. 139–142. Addison Wesley (1994)
15. Conway, R.J., Lex, A., Gehlenborg, N.: UpSetR: an R package for the visualization of intersecting sets and their properties. Bioinformatics. **33**(18), 2938–2940 (2017)
16. Ritchie, E.M., Phipson, B., Wu, W.D., Hu, Y., Law, W.C., Shi, W., Smyth, K.G.: Limma powers differential expression analyses for RNA-sequencing and microarray studies. Nucleic Acids Res. **43**(7), e47 (2015)
17. Merkel, D.: Docker: lightweight linux containers for consistent development and deployment. Linux J. **2014**(239), 2 (2014)