Heinrich Heine
Universität
Düsseldorf

INSTITUT FÜR INFORMATIK
Softwaretechnik und Programmiersprachen
Universitätsstr. 1 D–40225 Düsseldorf

A Jupyter Kernel for Prolog

Masterarbeit

im Studiengang Informatik
zur Erlangung des akademischen Grades

Master of Science (M.Sc.)

vorgelegt von
Anne Brecklinghaus

Beginn der Arbeit: 24. März 2022
Abgabe der Arbeit: 26. September 2022

Erstgutachter: Prof. Dr. Michael Leuschel
Zweitgutachter: Prof. Dr. Michael Schöttner

Selbstständigkeitserklärung

Hiermit versichere ich, die vorliegende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, den 26. September 2022 Anne Brecklinghaus

Abstract

Benefits of literate programming are well-known. The combination of source code and its documentation can improve the comprehensibility of a program considerably. *Jupyter notebooks* are documents corresponding to this idea by consisting of multiple cells that can either contain rich text or code. Code can be executed with any programming language for which a so-called *kernel* exists as an interface between a Jupyter application and the corresponding language. Support for a new programming language can easily be added by implementing such a kernel.

To create notebooks with Prolog, the Jupyter kernel *Herculog* was implemented. It is based on the default Jupyter kernel *IPython* for Python and code is executed by communicating with a Prolog server following a JSON-RPC protocol. It was developed for SICStus Prolog only at first and later, support was added for SWI-Prolog. Additionally, by providing configuration options, the kernel was made extensible for further Prolog implementations. In this thesis, the development of that kernel is presented as well as its application and extensibility.

Herculog provides almost all basic Prolog functionality. Additionally, some convenience features are implemented such as defining predicates on the fly and outputting results in a table. Moreover, it supports the Jupyter features of syntax highlighting and code completion as well as inspection for some predicates. Therefore, it offers advantages over the usual Prolog usage and can further facilitate teaching Prolog by documenting source code and creating student assignments.

Contents

1 Introduction

1.1 Motivation

In 1984, Donald Knuth, the creator of TEX, developed the notion of *literate programming* [28]. The idea is that programs and their documentation should be seen as *works of literature*. A good explanation should facilitate understanding the program and therefore result in less time spent on debugging. For this, the programming language WEB was developed with the aim of combining code in Pascal and documentation in TEX in one file.

Jupyter notebook documents are a more recent implementation of this idea. They are convenient for executing code interactively as well as for documenting it, which is why they have become popular in scientific contexts. The documents can be edited, viewed, and converted into other formats with Jupyter web applications such as Jupyter Notebook and JupyterLab. Each of these frontends communicates with a so-called *kernel* that is responsible for code execution. Therefore, the kernel determines the programming language in which code can be executed.

Even though a multitude of kernels for different languages exist, to the best of the author's knowledge, there is no proper Jupyter kernel for Prolog. Moreover, Jupyter applications are best suited for imperative programming languages. Since Prolog is a declarative language with its control flow differing considerably, several issues have to be faced when implementing such a kernel. Nevertheless, to use the Jupyter functionality with Prolog, the kernel **Herculog**[1] was developed, as is presented in this thesis. Note that parts of the thesis have been accepted as a paper for the 36th Workshop on (Constraint) Logic Programming (WLP 2022) [5].

While the kernel was intended for SICStus Prolog [8] only at first, it was later extended for SWI-Prolog [58], and with some adjustment, it can support further Prolog implementations. In addition to core Jupyter functionality such as code completion and inspection and basic Prolog clause definition and query execution, the kernel also provides more advanced features. These include printing all possible results of a query in a table and reusing the value of a variable that was assigned to it by a previous query.

Following an introduction to the main Jupyter and Prolog concepts, this thesis presents the features that Herculog provides for programming in Prolog with a Jupyter notebook. Additionally, alternative approaches for creating notebook documents with Prolog are introduced. After explaining the kernel's general architecture and the design choices made, details about the implementation are given. Finally, before listing some possible use cases and future improvements, the extensibility for further Prolog implementations and more functionality in general is explained while also reporting on issues during the porting to SWI-Prolog.

[1] The name alludes to Hercules, who, according to Roman mythology, is the son of Jupiter and known for his extraordinary strength.

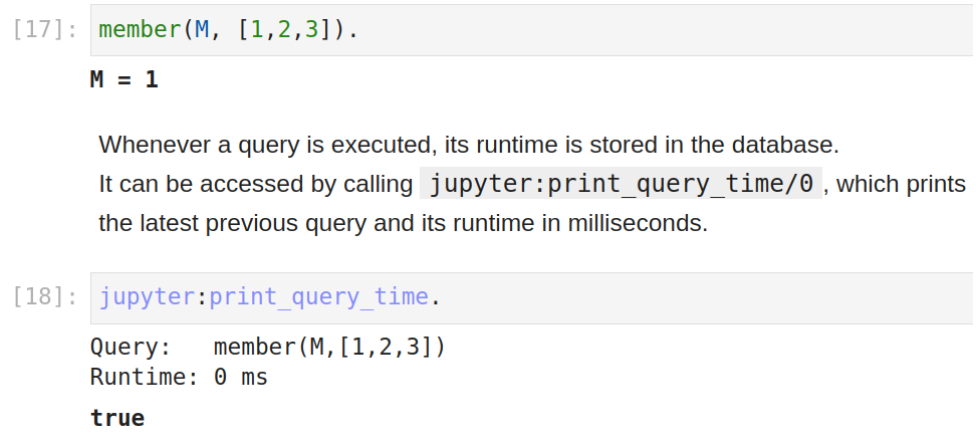


Figure 1: Cells in a notebook document connected to Herculog. The text in the middle was written as Markdown. The gray boxes correspond to code cells followed by their corresponding output.

1.2 From IPython to Project Jupyter

Recently, the usage of systems providing interactive code execution and visualization of the results accompanying the code has increased. Without having to recompile code for every change, it can be tested and refactored considerably faster. Furthermore, a good visualization of the results facilitates the distribution of those, especially in scientific contexts. That is why the **IPython** project [37] was developed. The main goal was to facilitate interactive Python development, which is what the interactive Python shell was implemented for.

Since the first release of IPython, multiple frontends were developed, some of which support notebook documents. Additionally, a new *two-processes model* was created. According to this, there is a client process responsible for user interaction while a kernel process handles code execution. In 2014, Project Jupyter [27, 43] evolved from the IPython project. It is an open-source project with the aim of making the IPython applications accessible for other programming languages. Today, IPython still provides the interactive Python shell and a Python kernel. Most other features such as the frontends were moved to Project Jupyter.

1.3 Jupyter Notebooks

Jupyter notebook documents consist of *cells* which can contain either executable code and its output or accompanying (rich) text which is not meant for execution (see Figure 1). Therefore, they can be used to execute code interactively and document it with text and visualizations. This can be helpful for both code development and documentation. Each notebook cell can be run and the behavior of the execution depends on the cell's type. A cell contains either code, Markdown, or raw text.

When running a code cell, its content is executed by sending it to the Jupyter kernel the notebook is connected to. The response is formatted and shown as the cell's output, which can be plain text as well as visualizations or other rich output. Each code cell is separate from the others in that its execution only influences its own output (aside from possible global program state modifications). Additionally, the content of a code cell can be syntax highlighted and code completion and inspection can be used for some elements of the code.

Executing a Markdown cell does not produce any output. Instead, the cell's possibly marked up content is converted to formatted rich text. These types of notebook cells support any HTML as well as a subset of LaTeX code, which is handled by MathJax [9].

The content of a raw cell does not change when being run, neither does it produce any output. These cells can be useful when converting the notebook into another format such as LaTeX. The content of a raw cell is included in the created file without being converted. For instance, this way, LaTeX code can easily be inserted in a LaTeX file created from a Jupyter notebook document. If a format is defined for a raw cell, the cell's content is not included when converting to any other format. If no format is specified, the content is always included.

1.3.1 Jupyter Notebook Applications

Internally, Jupyter notebook documents are stored as JSON files with the extension `.ipynb` in a specific format [22]. In 2011, the web application **Jupyter Notebook** [25] (formerly called **IPython Notebook**) was released, which provides a convenient way of using Jupyter notebooks. Multiple notebooks and other text documents can be worked on at the same time and notebook cells can easily be created, viewed, edited, and run. Additionally, the documents can be exported to other formats such as PDF, LaTeX, or HTML with **nbconvert** [20]. This way, a notebooks's content can be made accessible for others without the need to rely on Jupyter Notebook.

In 2018, the web application **JupyterLab** [40] was released, which is planned to replace Jupyter Notebook eventually. One of the main advantages of JupyterLab is that it is highly customizable since basically all functionality is provided by extensions. In addition to the core ones, users can install community-developed extensions or develop their own. With extensions, new themes, file editors, and viewers can be added as well as menu items, shortcuts, advanced settings options, and various other features.

In order to be able to distribute the results of a notebook even to a wider audience, **Voilà** [55] was developed. One reason for this was that interactive notebooks with code cells which have to be run are not ideal for non-technical users. Another reason was the security issue of executing arbitrary code. That is why Jupyter notebooks can be turned into standalone web applications containing interactive widgets with Voilà since 2019. Just like JupyterLab, it can be extended and it is language-agnostic so that it works with any Jupyter kernel. However, a main difference is that it does not allow code execution. Instead,

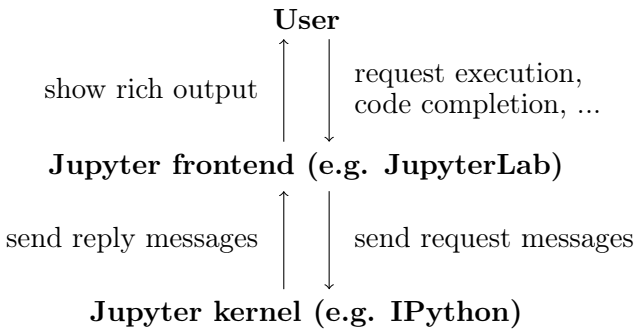


Figure 2: Diagram showing the handling of user interaction with a Jupyter frontend.

when rendering a notebook with Voilà, all cells are run and the output is collected so that it can be converted to HTML. However, code cells are hidden by default. On the created page, interaction with widgets is supported by accessing the corresponding Jupyter kernel.

Another Jupyter web application facilitating the distribution of notebooks is **nbviewer** [42]. When inserting the URL to a Jupyter notebook document, it is rendered as an HTML web page. The link to that page can easily be shared with others so that they can view notebooks without the need of having any Jupyter application installed.

1.4 Jupyter Kernels

Each Jupyter notebook is associated with a kernel, which is a process with the main task of executing code in a particular programming language. Furthermore, it is also responsible for code completion and inspection. As depicted in Figure 2, when a user interacts with a Jupyter frontend, a corresponding request is sent to the connected kernel. After handling the request, the kernel sends a reply to the frontend, which needs to be handled and displayed for the user. Because of this decoupling, one kernel can be connected to several frontends at the same time.

All the Jupyter applications can be used with any programming language for which a kernel exists. Besides the default IPython one and other popular kernels such as **IRkernel** for R [26] and **IJulia** for Julia [51], there are many community-maintained kernels for various languages [24].

In addition to connecting kernels to a Jupyter Notebook application, they can be used with a console frontend. At first, only **Jupyter Console** [52] existed, which is a terminal application for interactive computing with a programming language provided by a Jupyter kernel. Later, the terminal-like frontend **QtConsole** [53] was developed. On top of the Jupyter Console functionality, it supports features like rich output and syntax highlighting.

1.5 SICStus Prolog

Prolog is a logic programming language. According to Colmerauer and Roussel, at first it was not developed with the aim of creating a new programming language [10]. Instead, it was implemented to process natural languages. The usual workflow of Prolog starts with writing a source code file which defines *clauses*. These correspond to first-order logic formulas. A clause is a *fact* or *rule* and represents properties and relations of objects, which are called *atoms* in Prolog. All clauses with the same name and *arity* (i.e. number of arguments) define a *predicate*. To execute code, an interactive Prolog session can be started in a console. As this can be seen as a so-called *Read-Eval-Print Loop* (meaning that a user can continuously type in code that is evaluated and the result of which is output), it will be referred to as a REPL in the remainder of this thesis. This way, a user can load source files and query the defined data by calling predicates.

By now, there are multiple Prolog implementations which may differ considerably in terms of syntax and features. One of them is SICStus Prolog, which is a commercial one complying with the ISO standard. As described in a paper discussing the evolution of Prolog implementations, their portability, and future developments, SICStus Prolog started off as an open-source project originating from the idea of parallel execution [29]. It was developed to support research in or-parallelization, which means that alternative paths of an execution are explored in parallel with the aim of considerable speedups. At the time the development was started, Quintus Prolog was the *de facto* standard (e.g. for syntax and libraries). While SICStus was based on and compatible with Quintus Prolog from the beginning, numerous of its features became part of later SICStus versions. As of today, SICStus Prolog is still actively maintained, has facilitated further research, and is used for multiple commercial applications. Reasons for its popularity include its high performance and advanced support for constraint solving.

2 Application

This section describes the application of Herculog. Following an explanation of the installation process, its features are presented along with usage examples. Some features such as defining predicates and executing queries correspond to basic Prolog functionality. Additionally, there are special predicates, mostly implemented for user convenience. Almost all of them are provided with the module name expansion `jupyter`. Note that most of those need to be executed as the only goal in a term so that they can be recognized correctly.

To support programming in Prolog with a Jupyter application, one of the main requirements for the implemented kernel was to replicate the Prolog REPL as closely as possible. Therefore, any code to be executed needs to be valid Prolog code and the output mostly resembles console output. Additionally, some Jupyter specific messages are printed. These include information about the Prolog server (a process responsible for all code execution) being halted or restarted.

The code cells in notebooks can contain multiple Prolog terms. As another requirement for the kernel was to enable the definition of predicates on the fly, the terms can be clause definitions as well as directives or queries. Any variable bindings or other output produced by the terms is displayed in the cell's output. To increase ease of use, code can be executed even if the last term of a cell is missing the terminating full-stop. Thereby, a common cause for a query not to be run is eliminated. However, it should be noted that if a term does not terminate, the server gets stuck and no result at all can be displayed. In that case, a server restart is required by interrupting or restarting the kernel.

Herculog was implemented to create notebooks. Since JupyterLab is the most recent notebook application, the features are mostly described for that application. During the development of the kernel for SICStus Prolog, it was extended to support SWI-Prolog as well. While most functionality is the same for both implementations, there are some differences which are pointed out. Unless stated otherwise, all screenshots in the remainder of this section are taken from JupyterLab for SICStus Prolog. Furthermore, most of them are taken from a notebook explaining the provided functionality and some peculiarities, which is available with the source code at:

https://github.com/anbre/prolog-jupyter-kernel

Note that the notebook can be accessed with nbviewer without having to install Jupyter and Herculog:

https://nbviewer.org/github/anbre/prolog-jupyter-kernel/blob/master/notebooks/feature_introduction/sicstus/using_jupyter_notebooks_with_sicstus_prolog.ipynb

2.1 Installation

Herculog is provided as a Python package on the Python Package Index [54] and can be installed with `pip`. In order to be able to install and use it, Python and a Jupyter application such as JupyterLab needs to be installed. Additionally, a Prolog implementation is expected on the PATH environment variable. In general, all further required Python packages are installed during the installation of the kernel. However, for Windows, installing Graphviz [13, 50] with `pip` does not suffice. Instead, a manual installation is necessary.

When all requirements are met, the kernel can be installed with the following two commands:

```
python -m pip install prolog_kernel
python -m prolog_kernel.install
```

For the second command, installation options can be specified. These can be listed by running the command with the `-help` option.

As stated above, Herculog supports both SICStus and SWI-Prolog. Since in contrast to SICStus Prolog, SWI-Prolog is not commercial and is therefore used for teaching more often, the kernel executes code with SWI-Prolog by default. However, as described in Section 6.1, this behavior can be configured. Furthermore, the kernel can also be configured to be used with any other Prolog implementation for which a corresponding server exists.

2.2 Basic Prolog Functionality

When programming in a Prolog REPL, clauses of a predicate are usually defined in a source file which is loaded. They can also be added to the database by consulting the pseudo file `user` (e.g. with `[user]`) or by calling a predicate such as `assert/1`. In a Jupyter notebook, programs can be defined on the fly in code cells, which is a major advantage. However, allowing more types of terms than just queries also brings up the issue of having to differentiate between them.

While terms like directives and clauses with bodies can easily be distinguished from queries, it is more difficult for clauses without bodies. The expected main application of using Prolog interactively in a Jupyter notebook is to execute queries, possibly in multiple separate cells. Therefore, if a cell contains a single potential query, it is interpreted as such instead of a clause definition. Note that by writing a term of the form `foo :- true.`, a user can still assert a single term.

A Prolog source file usually contains predicate clauses. However, it is also possible to define goals which are executed when the file is loaded by extending them with a prefix `?-` or `:-` for a directive. To mimic this, terms starting with `?-` or `:-` are evaluated as queries even if the cell contains further terms. If instead, a potential query without prefix or body is encountered with multiple other terms, it is handled as a clause definition. Further, terms with bodies are always seen as clause definitions. In each case, as described below, the cell output lets the user infer how a term was interpreted.

```
[1]: app([], Res, Res).
     app([H|T], List, [H|Res]) :-
       app(T, List, Res).

     % Asserting clauses for user:app/3

[2]: user:app([], Res, Res).
     user:app([Head|Tail], List, [Head|Res]) :-
       app(Tail, List, Res).

     ▼Previously defined clauses of user:app/3 were retracted (click to expand)
     app([], A, A).
     app([A|B], C, [A|D]) :-
             app(B, C, D).

     % Asserting clauses for user:app/3
```

Figure 3: Predicate (re-)definition.

2.2.1 Predicate (Re-)Definition

All terms of a code cell interpreted as clause definitions (except for PlUnit tests) are added as dynamic facts to the database. However, this is not the case for predicates loaded from a file. Further, in the form `Head --> Body.`, a clause can also be defined as a DCG rule. In all cases, the clauses can be module name expanded (i.e. `Module:Head`). If no module name is defined, the module `user` is chosen by default. To let the user know the exact predicate that was defined, its specification is output (see Figure 3).

As described previously, the use of Prolog in a Jupyter application is meant for interactive programming. This involves writing, testing and rewriting clauses rather than adding new clauses to the fact database. Therefore, by default, when clauses are defined for a dynamic predicate for which there are existing ones, these are retracted first. This implies that all clauses of a predicate need to be defined in one cell. If previous clauses are retracted, the user is informed about the exact clauses.

Adding Further Clauses

Sometimes a user might want to add clauses for a predicate instead of redefining it. This can be achieved by declaring the corresponding predicate as discontiguous and dynamic. When removing a predicate with `abolish`, its properties are removed as well. Therefore, afterwards, previous clauses are retracted again when new ones are defined. It should be noted that in case of SICStus Prolog, declarations need to be handled seperately. That is why all declarations which are to be valid at the same time have to be defined with a single request. Furthermore, a single cell should not contain both, declarations for predicate properties and clauses for the same predicate. In that case, the clauses cannot be added to the database.

```
[7]: X = [1,2,3], append(X, [4,5,6], Z).

X = [1,2,3],
Z = [1,2,3,4,5,6]

[ ]: ?- member(2, [1, 2, 3]).
     ?- member(4, [1, 2, 3])

yes
no
```

Figure 4: Query execution.

```
[12]: ?- member(M, [1, 2]).
      ?- member(M, [a, b, c]).

M = 1
M = a

[13]: jupyter:retry.

% Retrying goal: member(M,[a,b,c])
M = b
```

Figure 5: Computing the next solution with `jupyter:retry/0`.

2.2.2 Query Execution

If a query succeeds and binds any variables, the bindings are shown in the output of the cell like they would be displayed in a console. Analogously, if there are no bindings or the query fails, the corresponding output for success or failure is printed (see Figure 4). It should be noted that, like in a REPL, usually no variable bindings are shown for directives. In every case, if the execution produces any output, it is displayed preceding the other information. Additionally, if the query causes an exception, the corresponding error message is printed.

2.2.3 Handling Multiple Solutions

Usually, when a Prolog query succeeds with a choicepoint, further solutions can be requested via backtracking. Since that is not possible in the same way in a Jupyter application, the predicate `jupyter:retry/0` is implemented to mimic this instead. Whenever a query is executed, it is seen as the active one as long as there might be further solutions for it. If there is an active query, by calling the `retry` predicate in a following query (which is possible in the same cell), backtracking may be triggered to compute the next solution. As can be seen in Figure 5, to let the user know which query was retried, a corresponding message is output.

Additionally, there is the predicate `jupyter:cut/0` to cut off choicepoints of the active execution. A previous query which might have open choicepoints is set as active, and the user is informed about the new active goal.

Since backtracking is a Prolog feature which is used frequently, these two predicates can also be called without the module name expansion and cannot be redefined in the `user` module. However, note that they do not work unless they are the only goal in a term.

As a user may wish to see the stack of queries which can be retried, the predicate `jupyter:print_stack/0` can be called to output them. The currently active query is printed at the top and marked by a preceding arrow.

```
[22]: use_module(library(clpfd)).
      true

[23]: X #< 10, X #> 5.
      X in 6..9
```

Figure 6: Loading and using `library(clpfd)` with SWI-Prolog.

```
[24]: :- begin_tests(list).

      test(list) :-
        lists:is_list([]).

      :- end_tests(list).
      % Defined test unit list

[25]: run_tests.
      % PL-Unit: list . done
      % test passed
      true
```

Figure 7: Defining and running tests for SWI-Prolog.

2.2.4 Loading Source Files and Libraries

In principle, when running Jupyter locally, source files and libraries can be loaded in the same way as in a REPL (see Figure 6). However, since user interaction is not possible, predicates are always re-defined instead of leaving the decision to the user.

A cell can contain both, code for loading a library and terms using predicates from that library, at the same time. The only special case that does not work is loading a library which defines new operators and using those operators in a single cell. In that case, when the server tries to read all terms of the cell, a syntax error is caused by the undefined operators.

2.2.5 Running Automated Tests

Automated tests can be defined and run with `library(plunit)`. Tests can either be defined in a file which is loaded or in a cell (see Figure 7). In the latter case, any definition of a `test/1` or `test/2` clause needs to be preceded by a `begin_tests` directive. Additionally, if there is an optional `end_tests` directive, it has to follow the test clauses. Otherwise, they are not recognized as such.

In case of SICStus Prolog, after defining new test clauses in a different cell, tests units of previous ones still exist, but not the tests themselves. Therefore, all tests which are to be run at the same time need to be defined in one cell. This is not the case for SWI-Prolog. Instead, test units defined in separate cells can be run at the same time.

```
[34]: jupyter:trace(app([1], [2], R)).
         1      1 Call: app([1],[2],_382553)
         2      2 Call: app([],[2],_386363)
         2      2 Exit: app([],[2],[2])
         1      1 Exit: app([1],[2],[1,2])
R = [1,2]
```

Figure 8: Printing the call stack with `jupyter:trace/1`.

2.2.6 Debugging

With Herculog, debugging cannot be performed interactively as user input is not supported. Thus, switching on trace mode with `trace/0` would cause the server to stop at an invocation and expect interaction, after which a kernel restart would be required. However, the call stack can be accessed by defining breakpoints which cause debugging messages to be printed. As this mechanism might be difficult to use, especially when newly learning Prolog, the predicate `jupyter:trace(Goal)` was implemented. While the goal `Goal` is executed, its trace is computed. To be recognized as a special goal, it needs to be the only one in a term.

By default, all ports are unleashed and included in the output (see Figure 8), which means that no user interaction is requested when a breakpoint is activated. However, the leashing mode does not apply to breakpoints. If a breakpoint is created which requires user interaction, the Prolog server has to be restarted after that breakpoint's activation.

2.3 Jupyter-Specific Features

2.3.1 Code Completion

In JupyterLab as well as Jupyter Notebook, code completion for the token at the current cursor position can be requested by pressing the *Tab* key. If there is a single possible match, the code preceding the cursor is replaced directly. Otherwise, a list of options is shown from which the user can choose one (see Figure 9).

Completion can be used for predicates which are built-in or exported by a currently loaded module. After loading another module, the completion data needs to be updated with `jupyter:update_completion_data/0`. Otherwise, completion does not work for predicates from that module.

The predicate terms with which the current token is compared and replaced are module name expanded. Therefore, it is possible to see all predicates of a module. This is especially useful for retrieving all special predicates defined by the module `jupyter`.

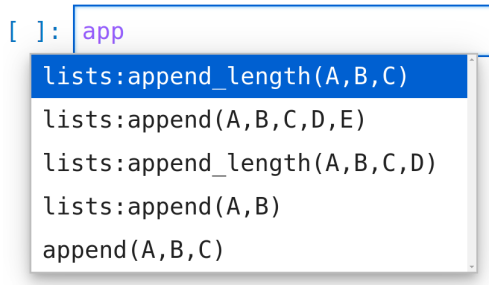


Figure 9: Completion for token `app`.

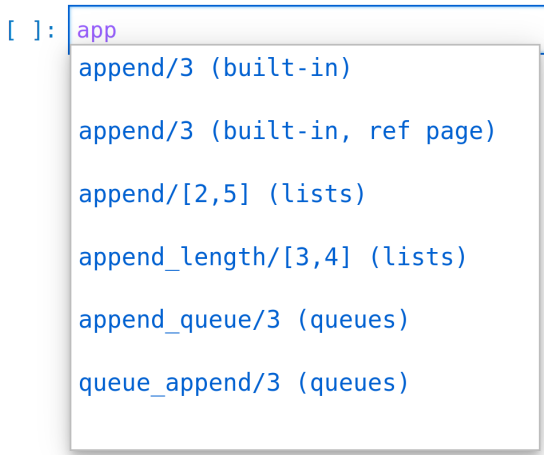


Figure 10: Inspection for token `app`.

2.3.2 Predicate Inspection

By pressing *Shift+Tab* in one of the Jupyter notebook applications, inspection for the token preceding the current cursor position is requested. For SWI-Prolog, documentation for the token is retrieved with `help/1` and shown right away. However, this is not possible for SICStus Prolog. Instead, for all predicates which the Predicate Index page [38] lists, a link to the documentation of the corresponding predicate is displayed if the predicate's name contains the current token. The data shown about the predicate is the same as given on the Predicate Index page. In most cases, it includes the predicate's name, arity, and information such as if it is built-in or the library's name. For JupyterLab, clickable hyperlinks can be displayed (see Figure 10). Since this is not possible for Jupyter Notebook, the links are only given to be copied.

In contrast to code completion, most predicate names are not module name expanded as they are not listed that way on the Predicate Index page. Therefore, normally, the documentation cannot be shown for all predicates from a module. An exception to this are the predicates from the special module `jupyter`. The documentation for those is provided by the module itself and is displayed at the bottom of the text for either Prolog implementation (see Figure 11).

Additionally, some documentation can be accessed with the predicate `jupyter:help/0`. When called, the documentation of all predicates defined in module `jupyter` is output. In case of SICStus Prolog, if the predicate is called without the module name expansion and it is not defined in the current module `user`, the corresponding error message is displayed followed by a note that there is `jupyter:help/0`. However, if `user:help/0` is defined, calling it works as expected without any special output for the `jupyter` predicate.

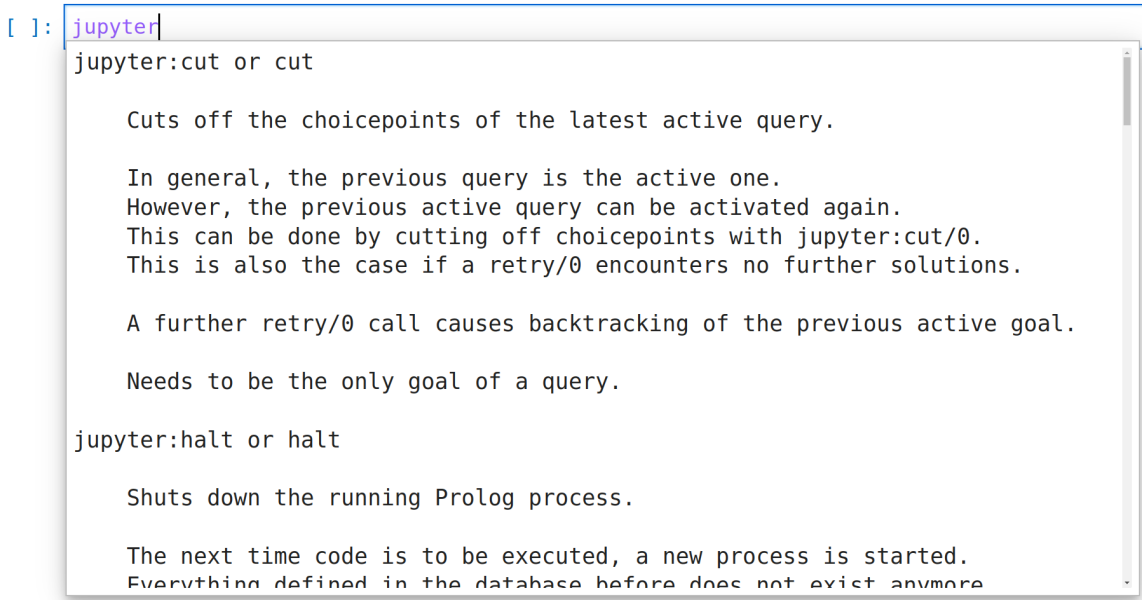


Figure 11: Inspection for token `jupyter`.

2.3.3 Resetting the Prolog State

As explained in more detail later, all code is executed by a Prolog server (see Section 4.4). As long as the server is running, the database state with all potential side effects such as clause assertions is active. However, the user might want to undo all those and reset everything to a clean state. This is especially useful when all cells of a notebook are to be run, but not all of them are supposed to work with the same Prolog database. There are several ways to achieve this.

The most Prolog-like way is to call the built-in predicate `halt/0` or `jupyter:halt/0`, both of which are handled specially by the server. However, in order to be identified as special predicates and processed correctly, they need to be the only goal of a query. Moreover, if `halt/0` is called in a different way, the server stops and an error message is output informing the user that something went wrong and the server needs to be restarted. If the handling does work as expected, a success message is displayed instead. Additionally, the Jupyter notebook applications provide buttons for interrupting or restarting the kernel. The same can also be achieved by pressing the keys *I+I* or *0+0* (zero) respectively.

All of these options cause the Prolog server to be stopped. The next time code is executed, it needs to be restarted. In that case, a message is output to let the user know that everything defined so far has been undone (see Figure 12).

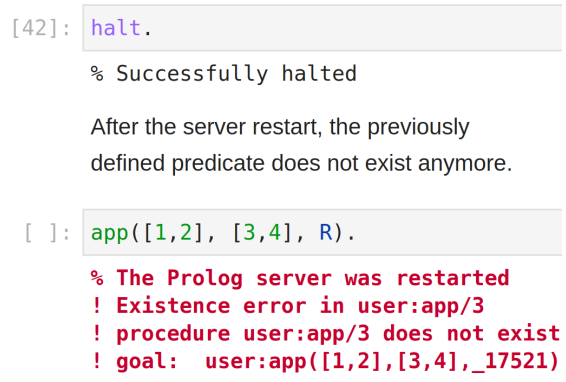


Figure 12: Resetting the Prolog state.

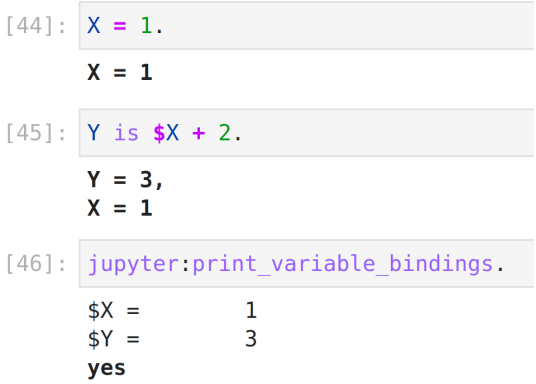


Figure 13: Accessing previous bindings.

2.3.4 Accessing Previous Query Data

When a query is executed, certain data about it is collected, which can be accessed by following queries. The data includes an atom representing the query, its absolute runtime in milliseconds, and its variable bindings. Additionally, the number of the execution which is displayed next to a notebook cell in case of success is collected. This corresponds to the ID of the request sent to the Prolog server.

Reusing Bindings

SWI-Prolog provides functionality of reusing top-level bindings [45]. When a top-level goal succeeds, its bindings are asserted in a database. By using a `$Var` term in a top-level query, the latest binding for the variable `Var` can be accessed.

This functionality is also provided for SICStus Prolog by Herculog. Additionally, there is the predicate `jupyter:print_variable_bindings/0` that outputs all stored variable bindings (see Figure 13).

Accessing the Previous Execution Time

Herculog provides a benchmarking feature. Whenever a query is executed, its runtime is computed with `statistics(walltime, Value)`. It can then be accessed with the predicate `jupyter:print_query_time/0`, which outputs the latest previous query and its runtime in milliseconds (see Figure 1).

```
[47]: member(Member, [10, 20, 30]).
Member = 10

[48]: Square is $Member * $Member.
Square = 100,
Member = 10

[49]: jupyter:print_queries([47, 48]).
member(Member,[10,20,30]),
Square is Member*Member.
yes
```

Figure 14: Collecting previous queries.

```
[52]: findall([M, S],
          (member(M, [10,20]), S is M*M),
          ResultLists).
ResultLists = [[10,100],[20,400]]

[53]: jupyter:print_table($ResultLists,
          ['Member', 'Square']).
```

Member	Square
10	100
20	400

```
yes
```

Figure 15: Reusing bindings to print a table with `jupyter:print_table/2`.

Collecting Previous Queries

When writing a new predicate, a user might test its subgoals gradually in separate cells, potentially using `$Var` terms to reuse previous results. Once all the parts are written, the predicate `jupyter:print_queries(Ids)` can be called to access the previous queries from cells with IDs in the list `Ids`. They are printed in a way that they can easily be copied to a cell and executed right away or expanded with a head to define a predicate (see Figure 14). If a query contains a `$Var` term and one of the previously printed queries contains the variable `Var`, the term is replaced by the variable name.

2.3.5 Structured Output

Herculog defines two special predicates to display data in a table. With the predicate `jupyter:print_table(Goal)`, all results of the goal `Goal` are computed with `findall/3`. The corresponding table contains a column for each variable occurring in the goal and a line for each result.

In order to fill a table with other data than results of `findall/3`, there is the predicate `jupyter:print_table(ValuesLists, VariableNames)`. `ValuesLists` is expected to be a list of lists where each of these lists corresponds to one row of the table. Therefore, all lists are required to be of the same length. The argument `VariableNames` is used to provide the column headers and needs to be a list of ground terms of the same length as well unless it equals `[]`. In the latter case, the headers contain capital letters starting from `A`.

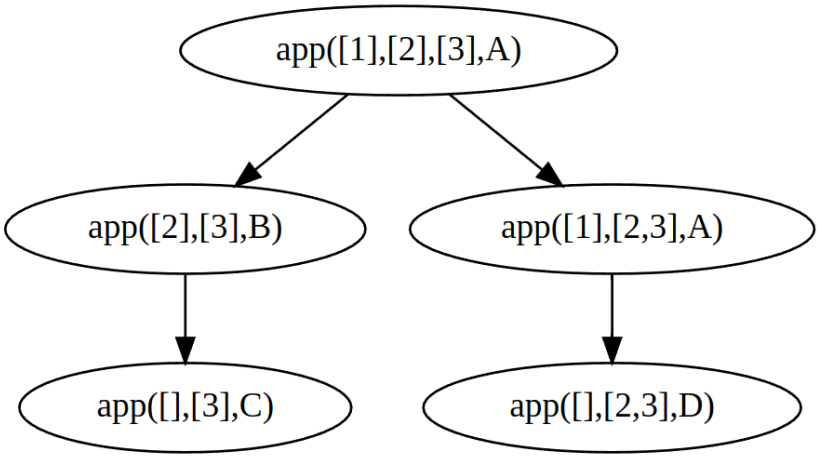
Note that both predicates have to be handled specially and cannot be recognized correctly unless they are the only goal in a query. Thus, the value for `ValuesLists` cannot be computed in the same query. Instead, previous bindings can be reused (see Figure 15).

```
[54]: app([], Res, Res).
      app([Head|Tail], List, [Head|Res]) :-
        app(Tail, List, Res).

      app(L1, L2, L3, Res) :-
        app(L2, L3, R1),
        app(L1, R1, Res).

      % Asserting clauses for user:app/3
      % Asserting clauses for user:app/4

[55]: jupyter:print_sld_tree(app([1], [2], [3], R)).
```




```
R = [1,2,3]
```

Figure 16: Printing a graph resembling an SLD tree.

2.3.6 Printing SLD Trees

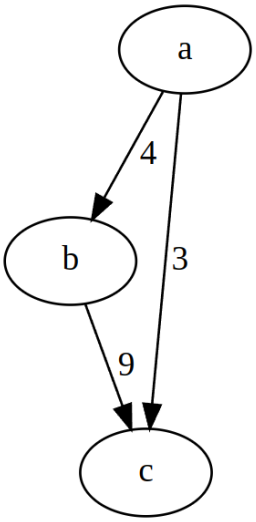
Usually, the execution of Prolog queries is based on so-called SLD resolution standing for *selection rule*, *linear resolution*, and *definite clauses*. The goals called during an execution can be visualized with an SLD tree. Herculog provides the special predicate `jupyter:print_sld_tree(Goal)` with which a graph resembling such a tree can be output (see Figure 16). In order to be recognized as a special predicate, it needs to be the only goal of a query.

Note that so far, nodes are only output for invocations at `call` ports. Therefore, successful branches cannot be distinguished from failing ones yet. However, visualizing the called goals in a tree can still facilitate debugging.

```
[56]: edge(a, 4, b).
      edge(a, 3, c).
      edge(b, 9, c).

      % Asserting clauses for user:edge/3

[57]: jupyter:print_transition_graph(edge/3, 1, 3, 2).
```



yes

Figure 17: Printing a transition graph.

2.3.7 Creating Transition Graphs

By calling `jupyter:print_transition_graph(PredSpec, FromIdx, ToIdx, LabelIdx)`, a transition graph can be created. Again, it needs to be the only goal of a query so that it can be treated specially. In that case, a graph representing the possible transitions between clauses of the predicate with specification `PredSpec` is created by computing all solutions (see Figure 17).

`PredSpec` needs to be of the form `PredName/PredArity` or `Module:PredName/PredArity`. The index arguments `FromIdx` and `ToIdx` point to predicate arguments used as nodes. `LabelIdx` points to the argument providing a label for an edge. If `LabelIdx` equals 0, no label is shown.

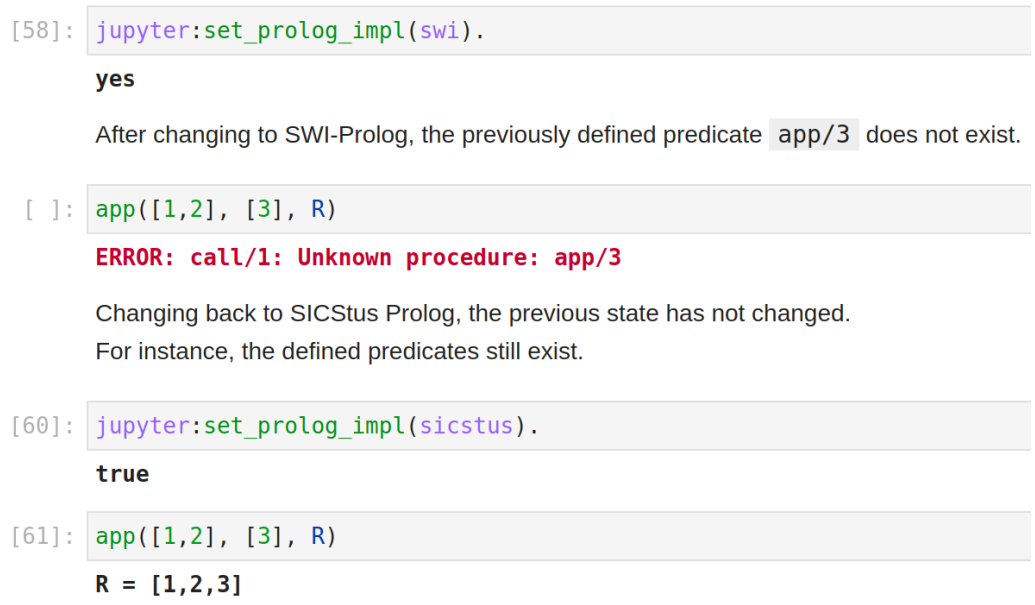


Figure 18: Changing the active Prolog implementation.

2.3.8 Changing the Prolog Implementation

If implementation-specific data is configured for more than one Prolog implementation (see Section 6.1), the active Prolog implementation used for code execution can be changed on the fly with `jupyter:set_prolog_impl(+PrologImplementationID)`. The corresponding goal needs to be the only one of a query, as otherwise the predicate cannot be recognized as a special one.

Note that the implementation is changed after all code of the cell has been executed. Therefore, any code following a `jupyter:set_prolog_impl/1` query in the same cell is still executed with the currently active implementation. Furthermore, the server for the previously used implementation is kept running so that when changing back, the state has not changed. For instance, the previous variable bindings and defined predicates still exist (see Figure 18). However, when the Jupyter kernel is interrupted, all running Prolog server processes are terminated and need to be restarted the next time code is executed. If a user wants to restart a single process, `jupyter:halt/0` can be called.

3 Related Work & Alternatives

In order to use a Jupyter notebook for executing code in a target programming language for which no kernel exists, there are two options. A simple way is to select a kernel for a language from which the target language can be called. Another approach is to write a new kernel for the corresponding language. This enables using that language for all Jupyter applications.

For the sake of completeness, note that instead of reusing the existing notebook implementation Jupyter, it is also possible to implement a new one which corresponds to the idea of *literate programming*. However, in the scope of this thesis, a new application could not have been implemented with significantly more features than basic ones. That is why using Jupyter and all its functionality was the preferable option.

The following subsections present examples of implementations of the different approaches for some Prolog versions while pointing out their features as well as limitations.

3.1 Calling Prolog from a Different Kernel

Magic Command

The most straightforward way of calling Prolog from a kernel for a different language is by using a magic command starting a Prolog session [46]. This can be achieved by defining an alias for a command using the IPython magic command `%%script`. With that IPython command, a program such as Prolog can be defined with which the rest of the cell is run. For SICStus Prolog as well as SWI-Prolog, the pseudo file user can be consulted. This way, facts and rules can be defined in a REPL or, in this case, from a notebook cell. By default, everything is interpreted as a clause and the code that is to be executed needs to be a directive or a query starting with `?-`.

An example of the command definition and SICStus Prolog code execution is shown in Figure 19a. In every cell, Prolog code needs to be preceded by the magic command starting a new session. Thus, all cells are independent of each other, which implies that defining clauses and querying them cannot be split into separate cells. Therefore, this approach is useful for executing small Prolog programs or showing simple examples. However, it is not suitable for writing and documenting large programs.

[1]:
```
%alias_magic prolog script -p "sicstus --goal '[user].' --nologo --noinfo"
```

```
Created `%%prolog` as an alias for `%%script sicstus --goal '[user].' --no
logo --noinfo`.
```

[2]:
```
%%prolog
parent(peter, chris).
parent(peter, caroline).

sibling(X, Y) :- parent(P, X), parent(P, Y), X \= Y.

:- parent(P, C), print(P), nl, print(C), nl, nl.

?- sibling(X, Y), print(X), nl, print(Y).
```

```
peter
chris

chris
caroline
```

(a) Using a magic command.

[1]:
```
from pyswip import Prolog
prolog = Prolog()
```

[2]:
```
prolog.assertz("parent(peter, chris)")
prolog.assertz("parent(peter, caroline)")

prolog.assertz("sibling(X, Y) :- parent(P, X), parent(P, Y), X \= Y")
```

[3]:
```
for sol in prolog.query("parent(P, C)"):
    print(sol)
```

```
{'P': 'peter', 'C': 'chris'}
{'P': 'peter', 'C': 'caroline'}
```

[4]:
```
for sol in prolog.query("sibling(X, Y)"):
    print(sol["X"], "is a sibling of", sol["Y"])
```

```
chris is a sibling of caroline
caroline is a sibling of chris
```

(b) Using PySwip.

Figure 19: Calling Prolog code from a Jupyter notebook associated with a Python kernel.

Calling Prolog Code from Python

Another way of executing Prolog code from a Jupyter notebook is by using a kernel for a programming language from which Prolog can be called. For instance, for a Python kernel this is possible with a Prolog interpreter written in Python such as **Pyrolog** [3], **pytholog** [14], or the **Simple Prolog Interpreter in Python** [47]. Another option is to use an interface between Python and Prolog such as **Picstus** [12] for SICStus Prolog or **PySwip** [49] for SWI-Prolog. However, these implementations do not claim to provide a complete representation of all available Prolog features or, in some cases, to work properly.

Nevertheless, they can be used for calling small Prolog programs from within other programs. This can be convenient for a computation for which Prolog is more suitable than the embedding programming language. However, since these implementations mix Prolog and Python code, they can neither be used without having any Prolog experience nor do they facilitate learning Prolog and its syntax properly.

The **PySwip** interface is used and referenced by a couple of other implementations and publications. For code execution in SWI-Prolog, its foreign language interface to C is used. PySwip provides methods such as `asserta`, `assertz`, `retract`, and `query`, which can be called to define and query Prolog facts and rules. In contrast to the magic command approach, each of them exists as long as the kernel is running. That is why cells containing facts and rules should normally not be executed more than once. When querying data, all possible solutions are returned as a generator. As can be seen in Figure 19b, each of them is a dictionary with the variable names as keys and their corresponding values as values.

Additionally, with PySwip, source files can be loaded with the method `consult`. However, there is no method for loading a library directly. Instead, a source file loading the library needs to be consulted. Even though this way a library can be used, there are further drawbacks. For instance, in the case of `library(clpfd)`, variables which are bound by constraints are output correctly, but for the ones which have been assigned a domain instead, no information is provided. The other implementations for calling Prolog code from Python listed above share similar limitations.

3.2 Prolog Kernels

As mentioned before, there are numerous community-written Jupyter kernels for various programming languages including Prolog implementations. Some of these are presented in the following.

One of them is the **Calysto Prolog** kernel [6, 35], which is a wrapper kernel extending the Metakernel (for further information about wrapper kernels, see Section 4.2). Code execution is based on a Prolog interpreter written in Python [33]. Each term ending with a `?` is interpreted as a query, and other terms are used to define new Prolog facts and rules. The kernel is suitable for defining and querying simple facts and even computing alternative

```
[1]: parent(peter, chris).
     parent(peter, caroline).

     sibling(X, Y) :- parent(P, X), parent(P, Y), X \= Y.

     Rule added to database.
     Rule added to database.
     Rule added to database.

[2]: parent(P, C)?

     Use '%continue' for more results.
[2]: {'P': peter, 'C': chris}

[3]: sibling(X, Y)?

     Unable to process query. Please restate.
```

(a) Calysto Prolog kernel.

```
[1]: parent(peter, chris).
     parent(peter, caroline).

     sibling(X, Y) :- parent(P, X), parent(P, Y), X \= Y.

[2]: ?- parent(P, C).

     P = peter, X = chris ;
     P = peter, X = caroline .

[3]: ?- sibling(X, Y) {1}.

     X = chris, Y = caroline .
```

(b) jswipl kernel.

Figure 20: Jupyter notebooks associated with kernels for Prolog.

solutions with a magic command. However, more complex but still basic functionality such as querying rules and producing output does not seem to work. Furthermore, no helpful error messages are printed (see Figure 20a).

Another Prolog kernel is **jswipl** [11], which was inspired by **SWI-Prolog-Kernel** [32]. Both of them are Jupyter kernels for SWI-Prolog. However, for the SWI-Prolog-Kernel, no proper installation instructions exist. It seems to work by writing Prolog code to a file which is executed in a subshell. This means that like in the case of using a magic command, all cells are independent of each other. It is therefore not possible to define clauses in one cell and query them in another. Queries are marked as such by enclosing them between lines containing `QUERYSTART` and `QUERYEND`.

The other SWI-Prolog kernel jswipl uses the SWI-Prolog and Python interface PySwip introduced above for code execution. As can be seen in Figure 20b, Prolog facts and rules can be defined and queried. Contrary to PySwip, each term added to the database exists as long as the kernel is running. Therefore, cells containing facts and rules should not be executed more than once. A query needs to start with `?-` and if there is more than one solution, by default, up to 10 answers are printed. This limit can be adjusted with special syntax.

The jswipl kernel does not claim to be complete or tested properly. In contrast to the Calysto Prolog kernel, there are some cases in which the jswipl kernel prints error messages. However, neither kernel provides more advanced features such as producing output or supporting DCG rules.

It is important to note that the kernels need to be able to differentiate between facts and queries. When using Prolog in the standard way, predicates are defined in a file and they can be queried by starting a Prolog REPL and loading the source file. When using a Jupyter kernel for Prolog instead, the executed code can either be a query or a clause definition. Therefore, the kernels introduced above expect queries to be somehow marked. While in the case of jswipl using `?-` preceding a query, this corresponds to valid Prolog code, the other kernels expect syntax which does not comply with the one that could be used in a Prolog REPL.

The need for marking queries can lead to unexpected or incorrect execution results and cause overhead. If a user forgets to mark a query and types in code that would be handled as a query in a Prolog REPL, it is added as a fact instead. Therefore, the kernel first needs to be restarted and some code cells may have to be rerun before the query can be executed correctly.

All of this shows that these approaches have major drawbacks, especially when it comes to advanced functionality. Furthermore, they do not seem to be maintained any more. Nevertheless, they can facilitate learning Prolog including its syntax better than calling Prolog from a Jupyter kernel for a different language.

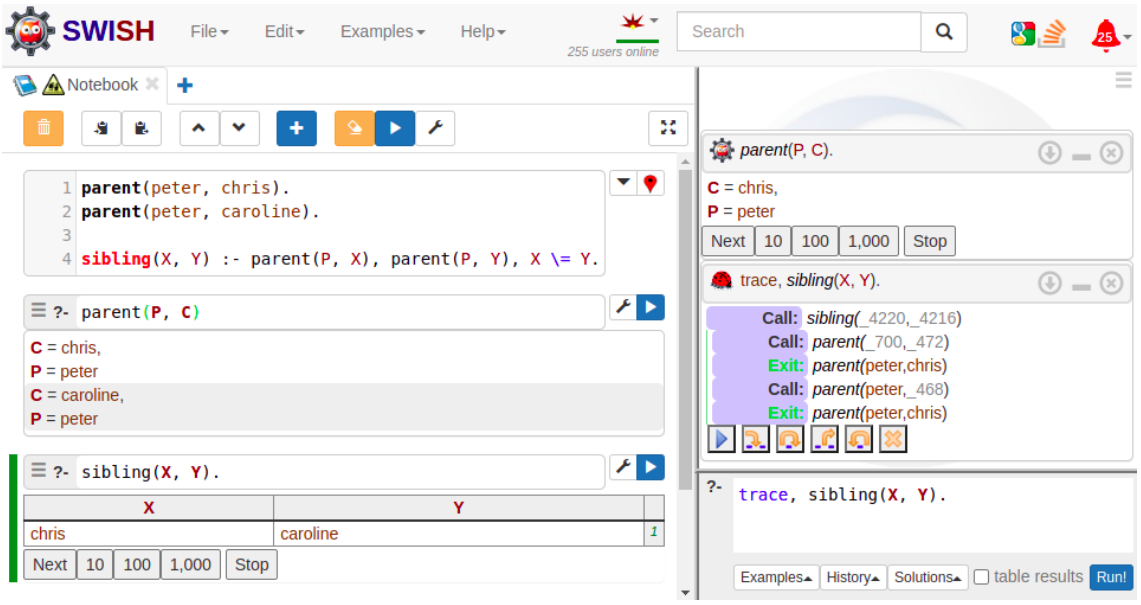


Figure 21: Example of a notebook and debugging in SWISH.

3.3 Implementing a Notebook Application

Instead of extending an existing notebook application, a new one can be implemented for a given programming language. Such a specific implementation has advantages such as being tailored to the corresponding language. Thus, it can provide all desired functionality, which might improve the ease of use and therefore facilitate learning and using the language. However, a major drawback is the amount of work that needs to be invested in implementing features which might already exist for other applications. Furthermore, it is basically not possible to test a new application as extensively as an existing one which might have been used for years and therefore has been tested by its users implicitly.

SWISH [2, 57, 59] is a notebook application especially developed for SWI-Prolog, and it is meant to facilitate sharing Prolog code. When saving a file, a user can decide if it is public, in which case other users can search for it. While Torbjörn Lager was the one who originally started developing it, the current version was implemented by Jan Wielemaker, the main developer and maintainer of SWI-Prolog.

Figure 21 is a depiction of the SWISH environment, where the left-hand side shows an example of a notebook. Each of the notebook cells can contain a `Program`, `Query`, `Markdown`, or `HTML` text. Code defined in a `Program` cell can be configured to be either callable from queries below the cell only or from all queries in the notebook. `Query` cells can be run and the number of desired solutions can be defined. In addition to that, queries can also be executed in a separate Prolog REPL as shown on the right-hand side of Figure 21. If a query succeeded with a choicepoint, additional solutions can be requested. Furthermore,

there is the option of displaying results as a table as well as download them as a CSV file. Moreover, by printing the whole notebook, it can be converted to a PDF file.

When running a query in the aforementioned REPL, information about some elements can be accessed by hovering over them. For example, for own predicates, the line of the first defined clause is shown and a description can be seen for SWI-Prolog predicates. For these queries, some interactive debugging features are available. The execution can be traced and there are buttons for continuing the execution, stepping into, out of, or over a goal, and retrying or aborting the execution. An example of this and a query without debugging is shown on the right-hand side of Figure 21. For each compound term in a result or trace, its arguments can be listed horizontally, vertically, or be omitted.

In contrast to the other approaches of using Prolog in a notebook document presented above, SWISH supports most functionality of SWI-Prolog including producing output. Furthermore, especially its debugging functionality is a considerable advantage over a normal Prolog REPL. However, it was implemented solely for SWI-Prolog and, as opposed to Jupyter, it is not meant for being extended for any other programming language.

4 Architecture and Communication

As can be seen from the approaches of using notebook documents with Prolog so far, creating a new Jupyter kernel is the best option for supporting the execution of valid Prolog code without needing to spend too much work on implementing basic notebook features.

There is more than one approach for writing a Jupyter kernel. As described above, the kernel is responsible for handling requests from a frontend such as JupyterLab and sending responses. This communication (see Section 4.1) needs to be taken into account when deciding for an implementation method.

The following subsections list some options for writing a Jupyter kernel (Section 4.2) and for interfacing SICStus Prolog with a different language (Section 4.3). Finally, the choices which were taken for Herculog are summarized and an overview of its architecture is given (Section 4.4).

4.1 Communication between a Jupyter Frontend and a Kernel

A Jupyter frontend communicates with a kernel by sending JSON messages complying with a specific message protocol over ZeroMQ [19, 56]. There are several types of messages for the available actions. In general, the frontend sends a request message of a specific type and expects a reply of the same type. However, it does not get a reply if the kernel does not support that type of message.

Among others, there are the following message types:

- **Kernel info**:

  Used to access core kernel information such as the programming language, its version number and the CodeMirror mode used for syntax highlighting.

- **Kernel status**:

  The kernel status is `starting` when the kernel is starting, `busy` in case a message is currently being handled, and `idle` otherwise. Whenever the status changes, the kernel publishes a status message.

- **Execution**:

  When the user wants to execute code, the frontend sends an execution request to the kernel. If the execution was successful, the reply contains the result and the status `ok`. Otherwise, the status is `error` or `aborted`.

- **Introspection**:

  Introspection can be used to access additional information about some elements of the code. The kernel is responsible for the information being displayed and how it is formatted.

- **Completion**:

  Upon request, the kernel returns all available matches for the code that is to be completed.

- **Interrupt**:

  Used to interrupt the kernel.

- **Shutdown**:

  The client can either request a final shutdown or a restart, which is preceded by a shutdown.

While all kernels are required to support kernel info, kernel status and execute messages, the other types of messages are optional.

4.2 Writing a Jupyter Kernel

When developing a new Jupyter kernel for a particular target language, the communication with the frontend described above needs to be implemented. This can either be done by writing a so-called *wrapper* kernel which extends another kernel handling the communication or by implementing everything from scratch.

While the latter option takes more work initially, it also has benefits. In such a case, the kernel can be written in the target language, making it a so-called *native* kernel. Thus, the execution of code received from the frontend is straightforward. Furthermore, the users of the kernel know the language it is written in, which means that it is more likely for them to contribute to it.

Writing a *wrapper* kernel is usually easier and quicker. This is the case if there are good wrappers for the target language so that it can be driven from the language of the extended kernel. This way, the messaging protocol does not need to be implemented again. Instead, only the language-specific part has to be written, which includes the execution of code, code completion, and inspection. However, a disadvantage is that the code execution is not as straightforward as for *native* kernels, since an interface between the languages is required.

There are several kernels which are a good base for being extended by a *wrapper* kernel. The following are examples for which good documentation for possible extensions exist:

- **IPython kernel** [16]:

  The default Python kernel.

- **Metakernel** [7, 35]:

  Another Python kernel providing additional magic functions.

- **xeus** [30]:

  A C++ version convenient for target languages that can be driven from C or C++.

- **Jupyter JVM BaseKernel** [36]:

  A Java implementation of the communication protocol which should be used for target languages running on the JVM only.

4.3 Interfacing SICStus Prolog with other Languages

As described in its user manual, there are several ways of interfacing SICStus Prolog with other programming languages [31]. While the interface for C or C++ and Prolog is built-in, there are libraries for other languages. The libraries `library(jasper)` and `library(prologbeans)` can be used for mixing Prolog and Java and the latter also serves as an interface with .NET. Additionally, `library(jsonrpc)` provides examples of calling Prolog from a client process in a language such as Python, C#, C and Java by using JSON objects for communication. This way, Prolog code can be called from basically any programming language with which JSON messages can be sent.

Using Java or .NET requires the extra dependencies of a JVM or .NET SDK respectively which a potential user might have to install first. Because of this and other reasons described in the following section, these languages were not used for Herculog. Those reasons are also arguments against mixing Prolog with C or C++ with the built-in mechanism. Furthermore, the C interface is rather complex, especially when it comes to computing more than one solution. In that case, it is important to understand the underlying structure of the Prolog term and query handling. This would make the maintenance of anything using this interface more complicated.

Therefore, no additional information about mixing Prolog with Java, .NET, C or C++ with the specific interfaces is provided here. Moreover, since the implemented kernel was intended for SICStus Prolog only at first, no interfaces of SWI-Prolog were taken into account. Thus, only `library(jsonrpc)` is introduced in more detail in the following.

library(jsonrpc)

The examples of `library(jsonrpc)` work by starting a Prolog server subprocess from a client process, which can basically be written in any language. Among others, examples are given for Prolog, Java, and Python. The processes communicate by sending JSON messages over the standard streams according to the JSON-RPC 2.0 protocol [34]. The client sends request messages, which are JSON objects. The objects need to contain the members `jsonrpc` and `method`. These provide the version of the JSON-RPC protocol, which is required to be `"2.0"` in this case, and a method name respectively. Additional parameters for the method can be provided with `params`.

```
{
   "jsonrpc":  "2.0",
   "id":  1,
   "method":  "call",
   "params":  {
      "goal":  "X is a."
   }
}
```

(a) Request message.

```
{
   "jsonrpc":  "2.0",
   "id":  1,
   "error":  {
      "code":  -4712,
      "message":  "Exception",
      "data":  "type_error(evaluable,a/0)"
   }
}
```

(b) Response message.

Figure 22: Messages sent between a Prolog JSON-RPC 2.0 server and a client for an execution throwing an exception.

If no `id` is included, the request is assumed to be a notification. Otherwise, the server is required to reply with a response object where, in general, the value of `id` is expected to be the same as the value of the request object. Additionally, the JSON-RPC version `"2.0"` has to be provided as the member `jsonrpc`. If the invocation of the requested method was successful, the response object will contain a member `result`. Otherwise, the value of the member `error` needs to be an error object containing values for `code`, `message`, and `data`. The response object for the request shown in Figure 22a is depicted in Figure 22b.

4.4 Herculog

As can be seen from the sections above, there were several options for developing a Jupyter kernel for Prolog. This section lists those options and gives an explanation of why the corresponding approach was chosen before presenting the architecture of Herculog.

The following were the main options for writing the kernel:

- Create a native kernel and implement the communication with a Jupyter frontend via ZeroMQ in Prolog.
- Write a wrapper kernel in Java based on the Jupyter JVM BaseKernel and communicate with Prolog via one of `library(prologbeans)`, `library(jasper)`, and `library(jsonrpc)`.
- Implement a wrapper kernel in C based on xeus and interface with Prolog directly or by using `library(jsonrpc)`.
- Use any other language for which a wrapper kernel can be written and which can send JSON requests to a Prolog server like the one provided by `library(jsonrpc)`.

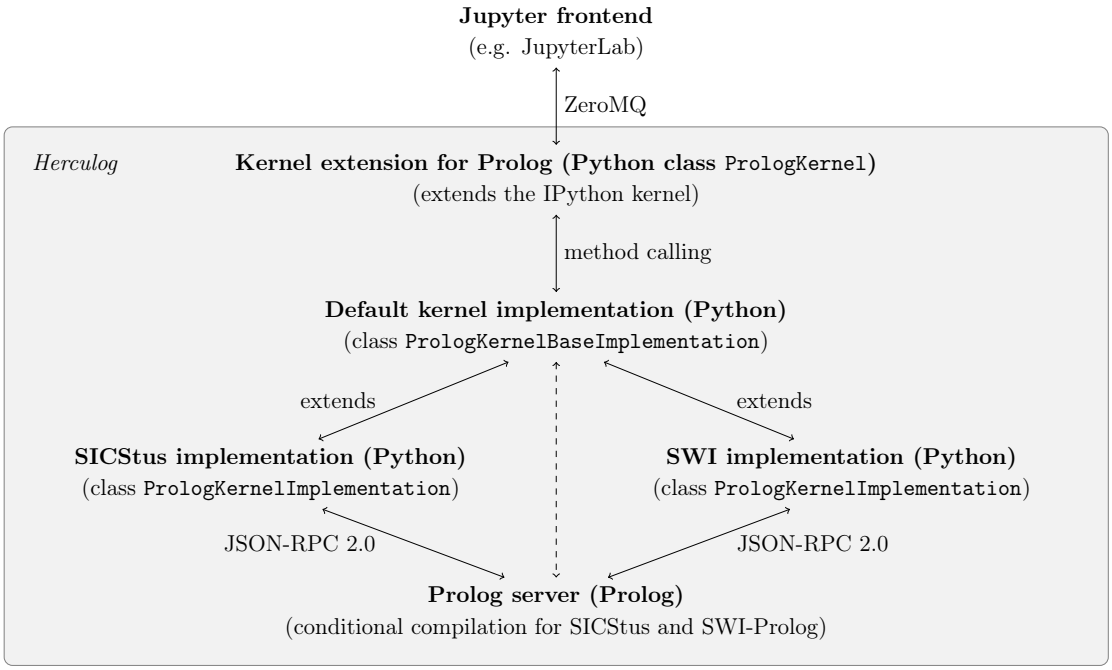


Figure 23: Diagram showing the architectural components and their communication methods.

Since the amount of work that could be invested for this thesis was limited, implementing Herculog as a wrapper kernel instead of writing the communication protocol in Prolog was preferable. This way, more effort could be concentrated on developing additional functionality.

Extending the Jupyter JVM BaseKernel requires a JVM, which is not needed for Prolog. Because this dependency should not be added unless necessary, the Prolog kernel was not implemented in Java.

Instead of using the interface to C, which is built-in, the decision was made in favor of `library(jsonrpc)`. One reason for this is that the C interface is more complicated, which would make the kernel more error-prone and difficult to maintain. Furthermore, the interface is SICStus-specific. By basing the kernel on the examples of `library(jsonrpc)` instead, the kernel can communicate with an independent Prolog server. This way, it can be made extensible for other Prolog implementations, which corresponds to the Jupyter notion of keeping things language-agnostic.

When writing a Jupyter kernel communicating with an independent Prolog server, the kernel can basically be written in any language. As Jupyter originates from a Python context and IPython is the default kernel, extending this kernel seemed most natural.

```
{
    "jsonrpc":  "2.0",
    "id":  0,
    "method":  "dialect"
}
```

(a) Request message.

```
{
    "jsonrpc":  "2.0",
    "id":  0,
    "result":  "sicstus"
}
```

(b) Response message.

Figure 24: Examples of messages sent between the Prolog server and the Python client for the method `dialect`.

Architecture and Communication

All of the above were reasons for deciding on the architecture and communication methods as depicted in Figure 23. Herculog is a wrapper kernel extending the IPython kernel. It communicates with a frontend by sending messages over ZeroMQ, which is handled by the IPython kernel.

The extension is written in Python and it does not interpret Prolog code itself. Instead, it starts a Prolog session as a subprocess and communicates with it over the standard input and output streams according to the JSON-RPC 2.0 protocol. For any code execution request the kernel receives from the frontend, a request message is sent to the Prolog server containing the code. The server then handles the execution and sends a response. Depending on the type of that response, a reply is sent to the frontend and displayed for the user.

To make Herculog extensible for basically any Prolog interpreter or customize its behavior, there is the additional layer of a *kernel implementation* in between. When the kernel is started, it loads a configuration file which can contain paths to interpreter-specific Python class files (see Section 6.1 for an explanation of the configuration options). The classes need to be extensions of the default implementation `PrologKernelBaseImplementation` called `PrologKernelImplementation`. They are responsible for basically all functionality including starting and communicating with the Prolog server. For almost all requests the kernel receives from the frontend, a `PrologKernelBaseImplementation` method is called, which sends a response after processing the request. For SICStus and SWI-Prolog, corresponding implementation classes are provided by Herculog. Note that instead of writing such a class for a new interpreter, it is also possible to use the default implementation. Additionally, the kernel can be configured to start a different Prolog server.

```
{
   "jsonrpc":  "2.0",
   "id":  1,
   "method":  "call",
   "params":  {
      "code":  "member(M, [1,2,3])."
   }
}
```

(a) Request message.

```
{
   "jsonrpc":  "2.0",
   "id":  1,
   "result":  {
      "1":  {
         "status":  "success",
         "type":  "query",
         "bindings":  {
            "M":  "1"
         },
         "output":  ""
      }
   }
}
```

(b) Response message.

Figure 25: Examples of messages sent between the Prolog server and the Python client for a successful execution.

Messages

As mentioned before, for each of the messages, a method is specified. In addition to `call`, `dialect`, `enable_logging`, and `jupyter_predicate_docs`, there is the method `version` for SICStus Prolog. While `call` messages are sent for code execution requests, the other ones are used to retrieve the current dialect, create a log file, access the documentation of the special `jupyter` predicates, and get the SICStus Prolog version respectively. For all methods except `call`, the corresponding message does not contain any additional parameters (see Figure 24a). As mentioned before, if the handling of the request is successful, the JSON response object sent by the server will contain the member `result`. In case of a method other than `call`, the corresponding value is a string representing the result (see Figure 24b).

The messages sent for `call` requests are more complex. The additional parameters given as `params` correspond to an object providing the code to be executed as `code` (see Figure 25a). Since a notebook cell can contain multiple terms, the `result` value of the response represents those terms' results. They are given as an object where the members are numbers starting from 1. Each of the results is an object with a `status` which is either `halt`, `success`, or `error`. Depending on the status, a corresponding reply is sent to the Jupyter frontend.

In case a term was executed successfully, its result object will additionally contain values for `type`, `bindings`, and `output`. The member `type` corresponds to the type of the term and can be one of `directive`, `clause_definition`, and `query`. If the execution caused variables to be bound, the value of the `bindings` member will be an object corresponding to those bindings. It contains the variables' names and values. Any output produced by the execution is given as `output`.

```
{
   "jsonrpc":  "2.0",
   "id":  1,
   "result":  {
      "1":  {
         "status":  "error",
         "error":  {
            "code":  -4711
            "message":  "Failure"
            "data":  {
               "prolog_message":  "",
               "output":  "test"
            }
         }
      }
   }
}
```

Figure 26: Response message as sent by the Prolog server for a failing term producing output.

For some terms handled separately, additional information is required. To provide this to the kernel implementation, the result objects can contain further members. Among others, these are `retracted_clauses` so that the potentially retracted clauses can be shown for clause definitions and `print_table` providing data to be output in a table.

If a term causes an exception or fails, the corresponding term result object will be of the type `error`. In that case, the result contains an `error` object. This corresponds to an error object as defined by the JSON-RPC 2.0 protocol. Thus, it contains values for `code`, `message`, and `data`. In this case, the `data` value is an object providing a Prolog message as `prolog_message`. Usually, this is the exception message as it would normally be output in a REPL. Figure 26 shows the response object for code printing output before failing.

5 Implementation

This section presents some implementation details of Herculog. Since one of the requirements was to replicate the Prolog REPL, the input needs to be valid Prolog code and the output should mostly resemble console output. This is supposed to facilitate learning Prolog and being able to transfer the knowledge to a console with minimal need for adjustment. Since none of the approaches introduced in Section 3 meet these requirements sufficiently and could easily be extended, the implementation is not based on any of them.

Instead, Jupyter's default IPython kernel is extended (Section 5.1). Further, the specific implementation details are handled by a class which can be extended for Prolog implementations different from the supported SICStus and SWI-Prolog (Section 5.2). For most of its features, the kernel needs to communicate with a Prolog server which executes code (Section 5.3). In addition to standard kernel functionality, the server provides some more specific features, mostly for convenience (Section 5.3.7).

For Jupyter notebooks, a single cell type is available for code, which aligns with the fundamentals of a typical imperative programming language. Prolog being a declarative language and its control flow differing significantly causes issues for various aspects of the implemented kernel. For instance, in addition to executing queries, it should be possible to define and modify a Prolog program in a Jupyter notebook. Distinguishing between program code and queries would be most straightforward with different types of cells. However, in order to keep compatibility with all Jupyter frontend applications, adding such cell types would be infeasible. For the same reason, no buttons can be implemented for cycling through several solutions or for providing debugging functionality as is the case for the notebook application SWISH. Instead, workarounds needed to be implemented (e.g. in the form of special `jupyter` predicates).

5.1 IPython Kernel Extension

Herculog is a wrapper kernel based on the IPython kernel. The latter provides the class `Kernel`, which handles the communication with a Jupyter frontend over ZeroMQ. Whenever the kernel receives a request message (e.g. for execution, inspection, or completion), a corresponding method is called. These methods must or may be overridden when writing a wrapper kernel [18].

In addition to those methods, there are attributes containing metadata about the kernel which a wrapper kernel needs to implement. One of them is `language_info`, which is a dictionary containing information about the target language of the kernel. This information is included in the JSON file of notebooks associated with the corresponding kernel. The attributes `implementation` and `implementation_version` refer to the kernel and its version. When used in console UIs, the value of the attribute `banner` is shown before the first prompt.

The implemented Prolog kernel provides the class `PrologKernel` in the file `kernel.py`. By extending the IPython class `Kernel`, the communication mechanism can be reused so that only code execution and some additional features a kernel may have need to be implemented.

The actual kernel code is not provided by this class itself. Instead, it mostly handles the loading of potential configuration files and creating an implementation object defining the actual kernel behavior accordingly. There is the file `prolog_kernel_base_implementation.py` which defines an implementation class named `PrologKernelBaseImplementation`. When Herculog is started, a (sub)object of this class is created. The implementation object handles the starting of and communication with the Prolog server. For all execution, shutdown, completion, and inspection requests the kernel receives, a `PrologKernelBaseImplementation` method is called.

Among other options, the kernel can be configured to use a specific Prolog implementation. In case the provided implementation ID equals `swi` or `sicstus`, by default, the subclass which needs to be called `PrologKernelImplementation` is loaded from one of the files `swi_kernel_implementation.py` or `sicstus_kernel_implementation.py` respectively. Otherwise, if no path to a file containing a valid subclass is configured, the default implementation class `PrologKernelBaseImplementation` is used. For more information about how to configure Herculog, see Section 6.1.

If configured accordingly, the Prolog implementation can be changed while the kernel is running. In that case, the previously active Prolog server is kept running so that it can be reactivated when the implementation is changed back to the previous one. Therefore, there can be multiple Prolog server processes running at the same time for a single Herculog instance.

Installation Files

A Jupyter frontend works with any of the kernels available on the system. As described in the Jupyter Client documentation, to retrieve all available kernels, specific locations are searched for directories containing kernel spec information [17]. For each kernel, one of these directories exists. It is a common practice to create a kernel spec directory which is copied to one of the kernel locations on installation. This directory can contain the following installation files:

- **kernel.json**:

  Should contain a JSON dictionary providing information about the kernel such as its name, the programming language, and how it can be started.

- **kernel.js**:

  Can be created to define custom syntax highlighting.

- **css files**:

  Can accompany the kernel.js file to further customize syntax highlighting.

- **logo image files**:

  Shown when selecting a kernel in a user interface.

The **kernel.json** file is the main file, and it is the only one mandatory in a minimal setup. It provides kernel information in form of a JSON dictionary. Among others, this includes the name of the corresponding programming language and the kernel's name, which may be displayed when selecting a kernel for a Jupyter frontend. Furthermore, command line arguments with which the kernel can be started need to be specified. These normally include the text `{connection_file}`. This text will be replaced with the path to a connection file a kernel is given when it is started by a frontend. That file contains all information necessary to establish a connection with the corresponding frontend such as the ports of the sockets used for communication.

A kernel can support syntax highlighting by specifying a CodeMirror mode [15] for its programming language. If there is no such mode for the corresponding language yet, a new one can be defined in the file **kernel.js**. While this enables syntax highlighting for the application Jupyter Notebook, for JupyterLab, an extension can be written instead. The file is expected to contain JavaScript code for defining syntax highlighting by parsing the source code. In order to further customize the highlighting (e.g. by adjusting colors), **css files** can be created.

If the installation files contain **logo image files**, these are shown in a corresponding frontend when a kernel is to be selected. In case no such files exist, a default image of the first letter of the kernel's name is shown.

For Herculog, the mandatory `kernel.json` file is provided as well as a `kernel.js` file defining a simple CodeMirror mode for Prolog.

5.2 Kernel Implementation

As stated above, the code for the actual kernel implementation is provided by the class `PrologKernelBaseImplementation`. It handles the starting of a Prolog server and communicates with it. Additionally, code completion and inspection are implemented. Thus, it is responsible for almost all features Herculog provides. These are explained in detail in the following subsections. All of them except interruption and syntax highlighting are implemented by overriding methods from the extended IPython kernel.

5.2.1 Execution

In order for the kernel to be able to execute code, the method `do_execute` needs to be overridden. This method is called whenever the kernel receives an execution request, and it is the only one which is mandatory for a wrapper kernel. One of its parameters is the code that the user inputs.

When Herculog is started, it starts a Prolog server subprocess which is responsible for all code execution. The code to be executed is sent to it without any modification. This is done by writing a corresponding JSON message to the standard input stream of the subprocess. For each request sent to the server, the client expects a corresponding response representing a valid JSON being written to the processes' standard output stream in a single line. If no response is received, the client cannot be stopped from waiting unless it is interrupted, shutdown, or restarted. If a response is received which is no valid JSON object or if any other error occurs while processing the response, an error message is sent to the Jupyter frontend and the server process is terminated. This is necessary since in that case, the server process might be in a state from which it cannot recover to handle further requests from the client.

A request can contain multiple terms, which are executed successively. For each execution, the server computes a result object of a specific type. If the execution was successful, it creates a `success` result. If it failed or caused an exception, an `error` result is created instead. Additionally, there is the type `halt` which needs to be treated separately. Once all terms have been processed, a response is sent containing all the term results. The remainder of this section describes how the results are handled by the kernel implementation.

Halt

By executing a term `halt` or `jupyter:halt`, a user can request to stop the Prolog server. In that case, the active process is terminated and the user is informed about it. The next time code is to be executed, a new process is started first.

Success

In case of a successful execution, the client receives a response containing the term type of the input, variable bindings, and any output that was produced. Further, for terms that need to be treated specially, the server can send additional information. If there is such data, it is handled first and any potential output is displayed in the frontend. Afterwards, if the term execution produced output, it is shown followed by variable bindings or a term representing success, which is `yes` in case of SICStus and `true` for SWI-Prolog. However, for directives, no bindings or success message is output.

There are six optional members a term result object can contain for additional data. They are processed in the following way:

- `predicate_atoms`:

  The user requested to update the predicate data available for code completion. The given value is stored internally so that it can be accessed when the kernel receives a completion request.

- `print_sld_tree`:

  The provided data is a string corresponding to the content of a file defining a graph. It is used to render an SVG file with dot, the content of which is then read in and sent to the frontend so that the graph is displayed.

- `print_table`:

  To display the results of a query in a table, a dictionary is provided with the members `ValuesLists` and `VariableNames`. The first one is a list of lists, for each of which a row of the table is computed. The latter is a list of strings from which the header of the table is created.

- `print_transition_graph`:

  The given data is processed in the same way as for `print_sld_tree`.

- `retracted_clauses`:

  When defining new clauses for a predicate, previous ones might be retracted first. In that case, a string representing them is computed and sent by the server so that it can be displayed to the user. Since this might not always be of interest, a message informing about retractions is shown which can be expanded to show the exact clauses.

- `set_prolog_impl_id`:

  The switching between Prolog implementations needs to be handled by the class extending the IPython kernel. If there is a running server for the implementation with the given ID, it is activated. Otherwise, the implementation-specific data is loaded (which starts a new server) and set as the active one. The previous server is kept running so that it can be re-activated again later.

The variable bindings are provided as a dictionary, where the keys are strings representing the variable names and in most cases the values correspond to the variable values. For each of them, a term of the form `Name = Value` is displayed, which resembles the Prolog REPL output. However, if `library(clpfd)` is used, there might be domain variables which have been assigned a domain instead of a single value. In that case, the variable value in the dictionary corresponds to another dictionary where the value `Dom` of `dom` is a string representing the domain. For these variables, results are printed in the form `Name in Dom`.

Error

In case the execution did not succeed, the server sends a message containing an error code. Among others, this code might stand for failure or an exception. Like in the case of success, if output was produced, it is displayed. If the code execution failed, `no` or `false` is output. In case an exception occurred, the result object contains the error message which would normally be printed to a console. This message is sent to the frontend.

5.2.2 Completion

Code completion is implemented by overriding the IPython method `do_complete`. When the Herculog kernel is started, a request is sent to the server for terms of all predicates exported from currently loaded modules and built-in ones. To find matches for code completion, the current token is retrieved and compared to the available predicates by checking if it is contained in the corresponding term.

5.2.3 Introspection

For an introspection request for the token preceding the current cursor position, the method `do_inspect` is called. The Prolog server provides the `jupyter` module which defines some predicates available to the user. Additionally, documentation is defined for them. The default kernel implementation loads these on kernel startup and receives a dictionary where the keys are module name expanded predicate specifications and the values are the corresponding documentation strings. After computing the current token, it is compared to the predicates. The documentations of all predicates for which the specification contains the token is collected to be displayed by the frontend.

For the remaining predicates, inspection needs to be handled differently for SICStus and SWI-Prolog. That is why it is implemented in the files `swi_kernel_implementation.py` or `sicstus_kernel_implementation.py`. In case of SWI-Prolog, when inspection is requested, predicate help is requested by calling `help/1` with the current token as argument. Its output is prepended to potential `jupyter` documentation.

For SICStus Prolog, supporting predicate inspection is more complex. Documentation can be retrieved from the Predicate Index page [38], which lists predicates and provides links to documentation pages. On kernel startup, all links are read for the current SICStus version, which is requested from the Prolog server. Most of the link texts consist of the predicate specification followed by additional information (e.g. `append/3 (built-in, ref page):`). The predicate names are compared to the current token. For all names containing the token, the corresponding links are collected and prepended to potential `jupyter` documentation. Since most predicate names are given without a module name, the module name the current token might have is not taken into account for the comparison.

5.2.4 Interrupt

When the interruption signal is sent, the active Prolog server receives this signal as well. Since it cannot recover from that to handle further requests, it needs to be shut down and restarted the next time code is to be executed.

5.2.5 Shutdown

When the kernel is shut down or restarted, the method `do_shutdown` is called. As a wrapper kernel, Herculog only needs to take care of its specific cleanup and the IPython kernel handles everything else. Since a single kernel might have started multiple Prolog server processes, all of them need to be terminated on shutdown. Otherwise, numerous stray processes might be caused.

5.2.6 Syntax Highlighting

Syntax Highlighting in Jupyter notebook applications can be activated by specifying a CodeMirror mode. Since there was none for Prolog, a simple mode was written for Herculog. For Jupyter Notebook, the corresponding code can be provided in the file `kernel.js`. For JupyterLab, however, an extension based on a Cookiecutter template [41] was implemented, which is available at:

`https://github.com/anbre/jupyterlab-prolog-codemirror-extension`

With this extension, it is also possible to activate syntax highlighting for basic Prolog files in JupyterLab. The corresponding language can be chosen by selecting *View* > *Text Editor Syntax Highlighting* > *Prolog*. Note that by default, the language Perl is selected for files ending with `pl`.

5.3 Prolog JSON-RPC Server

The source code of the Prolog JSON-RPC server is split into several modules. The diagram in Figure 27 illustrates those modules and their dependencies. The top section of each box contains the module name which equals the file name without the `.pl` extension. The bottom one contains all exported predicates of a module. If a module uses another one, this is indicated by an arrow pointing to the latter. Note that all names start with `jupyter_` to avoid conflicts with other modules loaded by the user.

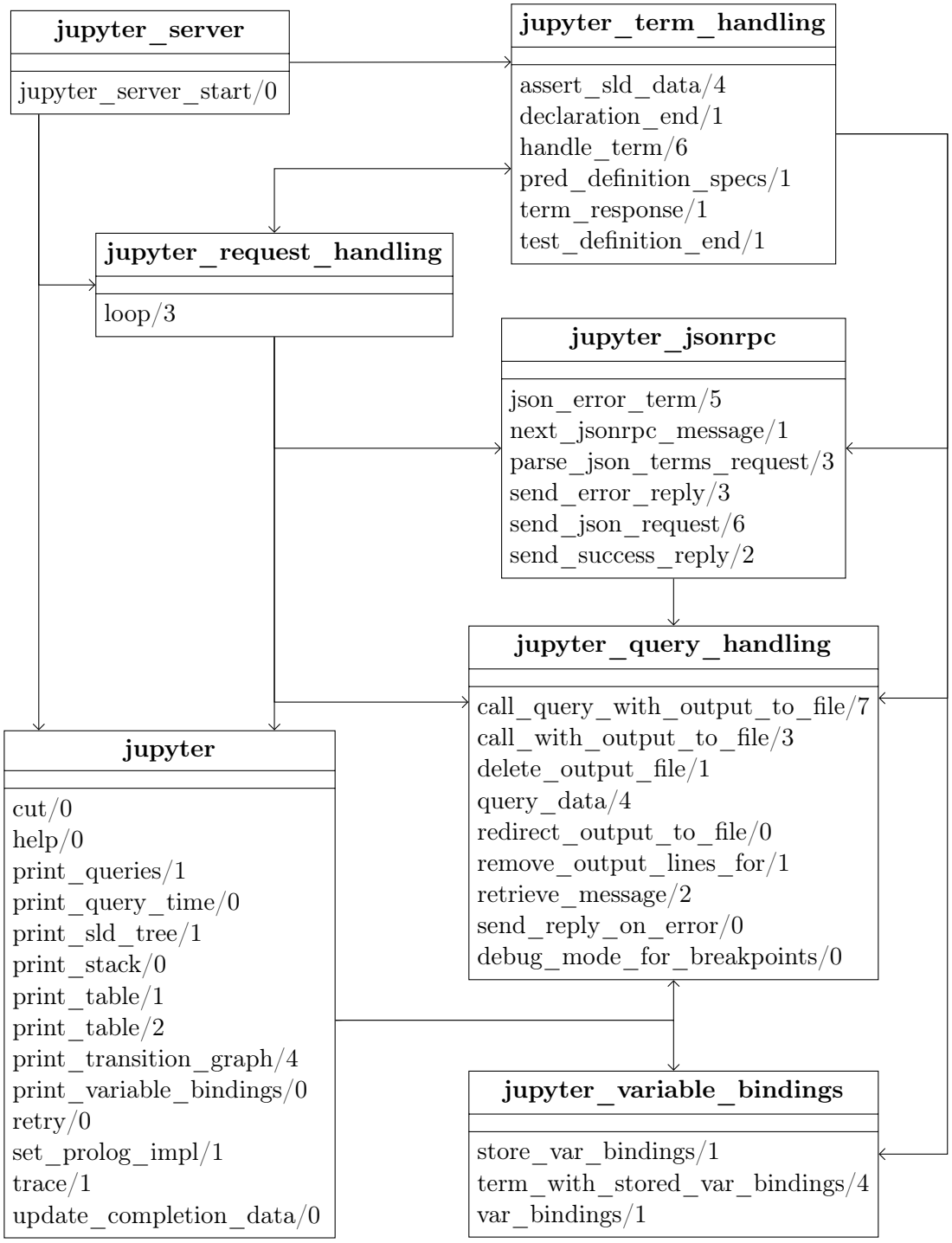


Figure 27: Diagram representing the modules of the Prolog server and their dependencies.

The only source code files belonging to the Prolog server which are not shown in the diagram are `jupyter_server_tests.pl` and `jupyter_logging.pl`. The former file contains tests for different types of JSON request messages sent to the server. When running the kernel, the Prolog execution part cannot simply be debugged and writing to the standard output stream would send messages to the Python client. Therefore, the file `jupyter_logging.pl` defines a module which can be used to write logging messages to a file.

The following are the remaining modules implementing the features provided by the server:

- `jupyter_server`:
  The main module of the server. It is the one with which the process is started.
- `jupyter_request_handling`:
  Used for starting a loop which reads requests, handles them and sends a response.
- `jupyter_jsonrpc`:
  Provides predicates for creating JSON terms and for reading from and writing to the standard input and output streams.
- `jupyter_term_handling`:
  Implements the handling of different term types.
- `jupyter_query_handling`:
  Handles the actual execution of queries and their output while collecting query data.
- `jupyter`:
  Provides some special predicates available to the user (e.g. `jupyter:trace/1`).
- `jupyter_variable_bindings`:
  Handles bindings of previous queries.

In the following subsections, each module is explained in more detail. The source code files contain code for both SICStus and SWI-Prolog by making use of conditional compilation with the directives `if(...)`, `else`, and `endif`. Most code is compatible with both Prolog implementations. However, there are major differences, as pointed out in the description.

5.3.1 jupyter_server

The `jupyter_server` module is the main one of the Prolog server. It exports the predicate `jupyter_server_start/0` with which the server can be started. This is done by calling the predicate `loop/3` from the module `jupyter_request_handling`, which starts a loop handling requests from a client. Additionally, it contains implementation-specific code needed for the setup, especially for enabling tracing. Further, the printing of some special messages is defined.

5.3.2 jupyter_request_handling

The module `jupyter_request_handling` exports the predicate `loop/3` starting a loop to read and process JSON requests and Prolog terms. It reads a message from the standard input stream and parses it with `jupyter_jsonrpc:next_jsonrpc_message/1`. If the message represents a valid request, it is handled according to the provided method.

The method `call` is used for calling Prolog code, of which the result is returned to the client. In that case, the request is expected to contain code which is parsed with `jupyter_jsonrpc:parse_json_terms_request/3`. If it contains a syntax error and therefore cannot be parsed correctly, an error reply is sent to the client. In case the code does not contain any term (i.e. it only contains comments), the server replies with a success message. Otherwise, the request contains one or more Prolog terms. Before handling the first term with `jupyter_term_handling:handle_term/6`, the remaining terms are asserted so that they can be retrieved by the next call of the `loop/3` predicate. Once all terms of a request have been processed, a response containing the terms' results is sent to the client. Afterwards, the next message is tried to be read.

With the method `enable_logging`, a log file is created to which log messages can be written. Additionally, the Herculog kernel needs some further information which can be accessed with the remaining request methods. The method `dialect` is provided to retrieve the current dialect. The documentation of the special predicates defined in module `jupyter` can be accessed with the method `jupyter_predicate_docs`. Additionally, for SICStus Prolog, the current version is requested from the server with `version`.

5.3.3 jupyter_jsonrpc

All reading, writing, and parsing of JSON messages is handled by the `jupyter_jsonrpc` module based on the files `jupyter_server.pl` and `jsonrpc_client.pl` from SICStus (version 4.5.1). The predicate `next_jsonrpc_message/1` is used to read the next message from the standard input stream. If it corresponds to a `call` request, `parse_json_terms_request/3` is called to parse the given code. This is done by reading Prolog terms from the code with `read_term/3` or `read_term_from_atom/3` depending on the Prolog implementation. By reading the terms with the option `variable_names(Variables)`, a list of `Name = Var` terms is retrieved. `Name` is an atom representing the name of a variable in the term and `Var` is the corresponding variable. In case the term is a query which is called, the variables might be bound. This way, the bindings can be output as the result of the term.

If the execution of a query results in an error, a JSON error term is created with the exported predicate `json_error_term/5`. Further, success or error replies are sent with `send_success_reply/2` or `send_error_reply/3` respectively by writing them to the standard output stream. Additionally, `send_json_request/6` is defined for testing the server by writing messages to the input stream and reading their responses from the output stream.

5.3.4 jupyter_term_handling

The module `jupyter_term_handling` is responsible for the actual handling of terms from a `call` request. The main predicate it exports is `handle_term/6`. It processes a term according to its type, which is either a directive, clause definition, or query. Furthermore, for each of the types there are terms which need to be processed separately. If a term has a result, it is asserted in the database so that it can be sent to the client when all terms of a request have been processed.

As mentioned before, distinguishing clause definitions from queries is not always straightforward. One option for differentiation would have been to expect all queries to be somehow marked. However, this would likely cause frustration for users forgetting about it because they are used to a Prolog REPL where no such marking is required. Since most single terms appearing in a cell are likely to be queries, they are interpreted as such. Additionally, terms starting with `?-` and most directives starting with `:-` are handled as queries even in cells with multiple other terms. In all other cases, the terms are treated as clause definitions.

Directives

The directives `begin_tests/1`, `begin_tests/2`, and `end_tests/1` provided by the library `library(plunit)` must be handled seperately. To use PlUnit tests with Herculog, the tests need to be written to a file which is loaded. In general, such a test definition file is loaded after processing all terms of a request or when a `run_tests` query is to be called. When the first `begin_tests` directive of a request is encountered, a file is created and opened for writing. The corresponding stream is asserted so that it can be accessed to write to the file. The following test definitions and `end_tests` (in case of SICStus Prolog also `begin_tests`) directives are written to the file.

With SWI-Prolog, reloading a file which does not define a test unit that has been loaded from it before, causes an error. Furthermore, when a test unit was loaded from a file, a unit with the same name cannot be loaded from a different file afterwards. Therefore, every test unit is written to a file of which the name contains the unit name. Every time a `begin_tests` directive is encountered, the previous file is loaded and a new one is opened.

In principle, the same could be done for SICStus Prolog. However, each time a file defining tests is loaded, a redefinition warning for `unit_body/4` is displayed if the file name differs from the previous test definition file. This would be the case for every test unit. Instead, for SICStus Prolog, all test definitions of a cell can be written to the same file. Moreover, the file is always called the same and loaded once all terms of a request have been handled. Note that all tests which are to be run at the same time have to be defined by the same request. Otherwise, a previously defined test unit still exists, but not the tests in it. This is different in the case of SWI-Prolog, for which all test units defined in the current server session can be run at the same time, even if they were defined by separate requests.

For SICStus Prolog, declarations need to be treated specially as they must not appear in a query. Therefore, like in the case of test definitions, all declarations of a request are written to a file which is loaded. This implies that all declarations which are to be valid at the same time, need to be defined with a single request. The declaration file is loaded when all terms of a request have been handled. When declaring a property of a predicate for which clauses had been asserted before, these do not exist any more after the declaration. Thus, a cell declaring predicate properties cannot define clauses for the same predicate.

In general, the execution of any other directive starting with `:-` is the same as for a query, which is described below. The only differences are that for directives, backtracking is not possible and variable bindings are not displayed, which corresponds to the REPL behavior.

Clause Definitions

When it comes to clause definitions, there are two predicates which need to be treated specially. These are `test/1` and `test/2`. If they are preceded by a `begin_tests` directive, the corresponding clause is written to a test definition file as described above. Otherwise, they are not interpreted as test definitions, but handled as regular clause definitions instead.

In general, for any other clause definition, the clause is added to the database with `assertz/1`, making it a dynamic predicate. Optionally, clauses can be module name expanded and if they are not, module `user` is chosen by default. If the term is a DCG rule, it is expanded with `expand_term/2` (or `dcg_translate_rule/2` in case of SWI-Prolog) before further processing.

To be able to redefine a predicate without having to remove its clauses first, previous clauses are retracted automatically by default. Whenever a clause is to be defined for a predicate for which there already are clauses, these are retrieved with `listing/1` before retracting them. To inform the user about the retraction, the term result contains the corresponding atom as data for the additional member `retracted_clauses`. Note that this mechanism implies that all clauses of a predicate usually need to be defined by one request.

However, it is also possible to define a predicate in separate cells by declaring it as discontiguous. In that case, new clauses are added to the database without retracting the previous ones first. For SWI-Prolog, there is the predicate property `discontiguous` which can be checked. Since such a property does not exist for SICStus Prolog, for each clause declared as discontiguous during the current server session, a `jupyter_discontiguous(PredSpec)` fact is asserted. When a user removes a predicate with `abolish`, the corresponding clause is retracted. However, the predicate cannot be recognized unless the call to `abolish` is the only one in a term.

Queries

The Prolog server provides the functionality of using a term of the form `$Var` in a query to access the latest value the variable `Var` has been bound to by a previous query. In order for this to work, before a query is processed, any of its subterms of the form `$Var` are replaced accordingly. In case of SWI-Prolog, this mechanism is built-in. Instead, for SICStus Prolog, the handling of `$Var` terms is provided by the module `jupyter_variable_bindings`. In either case, the original term needs to be asserted so that previous queries can be accessed in a state before unification assigned values to any of their variables. This is needed for the special predicate `jupyter:print_queries/1` described below (see Section 5.3.7).

Some predicates that need to be treated separately are listed at the end of this section. For any other query, `jupyter_query_handling:call_query_with_output_to_file/7` is called, which is introduced in the next section.

To be able to reuse bindings with `$Var` terms, in case a query was executed successfully, previous bindings are updated with the ones from the current query. Additionally, before asserting the term result, it needs to be made sure that the variable values do not cause an exception when sent to the client. For instance, this would be the case for uninstantiated variables or compound terms, as these cannot be parsed to JSON terms with the corresponding Prolog library. Therefore, in most cases, the values are converted to atoms. However, with `library(clpfd)`, there might be domain variables which have been assigned a domain instead of a single value. The term representing that domain of a variable `X` is retrieved with `clpfd:fd_dom(X, Range)`. Usually, the variable bindings list sent to the client contains elements of the form `Name=Var`, where `Var` is the variable which might have been bound by the query. Instead, for domain variables, `Var` corresponds to a JSON object where the member `dom` provides an atom representation of the domain `Range`. This way, the client is informed about the special case and can display the result accordingly.

As described above, all requests are processed in a loop. In general, after asserting a term result, the current predicate is exited and the loop continued. However, if a query execution is successful, instead of continuing the current loop, a new recursive loop is started with `jupyter_request_handling:loop/3`. In that case, the current goal is seen as the active goal which can be retried or of which choicepoints can be cut off if there are any. This is possible because of `call_query_with_output_to_file/7` leaving a choicepoint. When a `retry` request is encountered, the current loop iteration fails, causing the active goal to backtrack. One argument of the `loop/3` predicate is a list of atoms representing the queries which were called. This way, it is possible to keep track of the called queries (resembling a call stack), which is needed for some special predicates.

The predicates treated specially are listed in the following. Note that all of them are required to be the only query in a term to be recognized correctly. Otherwise, an error message might be output or the query might produce unexpected results. Additionally, for some of the predicates the result object contains additional data. Others are treated separately because they require data such as the call stack or variable bindings.

- `retry` or `jupyter:retry`:

  The user requested to retry the active query. The current predicate fails into the caller `jupyter_query_handling:call_query_with_output_to_file/7` so that backtracking is caused. Before failing, a clause is asserted to inform the caller about the retry. In that case, the actual goal which is called is output. Usually, before calling a query, all output is redirected to a file and `statistics/2` is called to compute its runtime. This also needs to be done before backtracking.

- `cut` or `jupyter:cut`:

  The predicate `jupyter_query_handling:call_query_with_output_to_file/7` is the one which actually executes queries. By cutting off all its possible choicepoints, a potentially active goal cannot be retried. Afterwards, a previous query is seen as the active one and the user is informed about the new active goal.

- `jupyter:print_stack`:

  While looping over terms and requests, a stack of previously called queries is built. These are the goals which can be retried. All elements of this stack can be printed with the currently active query displayed at the top and indicated by a preceding `->`.

- `halt` or `jupyter:halt`:

  When the halting of the Prolog server is requested, the loop reading and processing messages is stopped so that the server process is stopped as well. In order to inform the client about this, a success reply of type `halt` is sent. This way, the next time code is to be executed, a new server is started.

- `jupyter:print_table(+Goal)` or
  `jupyter:print_table(+ValuesLists, +VariableNames)`:

  When the server sends a response object containing data for the additional member `print_table`, it is displayed in a table. The value needs to be an object itself with the members `ValuesLists` and `VariableNames`. The first one is a list of lists where each of them corresponds to one row of the table. The latter is a list of names for computing the header of the table.

  In case of `jupyter:print_table(Goal)`, the table will contain the results of the goal `Goal` computed with `findall/3`. The header will consist of the names of the variables occurring in the goal (which were extracted when reading in the term). Note that like in any other case, if the goal does not terminate, no data at all can be sent to the client and therefore displayed. Instead, the server needs to be restarted.

  The predicate `jupyter:print_table(ValuesLists, VariableNames)` can be used to print given data in a table. The arguments mostly correspond to the object members of the same names described above. However, before sending the response, it has to be ensured that all terms are JSON parsable. `VariableNames` is either of `[]` or a list of ground terms which are required to be of the same length as the values lists. If no names are provided, capital letters are used instead.

```
digraph {
   "1" [label="app([1,2],[3],A)"]
   "2" [label="app([2],[3],B)"]
   "3" [label="app([],[3],C)"]
   "1" -> "2"
   "2" -> "3"
}
```

Figure 28: Content for a file representing a graph resembling an SLD tree.

- `jupyter:print_sld_tree(+Goal)`:

  To output a graph resembling the SLD tree of the goal `Goal`, during its execution, required data about the invocations at `call` ports is collected with debugging features. The collection of the data differs between SICStus and SWI-Prolog. In case of SICStus Prolog, a breakpoint is added before and removed after the execution. For SWI-Prolog, a dynamic fact `collect_sld_data` is asserted before calling the goal and retracted afterwards. Its existence is checked by the dynamic predicate `prolog_trace_interception/4`, which is called before the debugger displays a goal.

  After collecting the data, content for a file representing the graph is created which can be rendered with Graphviz. For the file content, nodes are defined by an ID and labelled with the string representing the corresponding goal. Directed edges are added from a parent invocation to the child invocation (see Figure 28). A string representing the file content is sent to the client as additional data for `print_sld_tree`.

  Since so far, data is collected for call ports only, no leaves are shown marking a successful or failing branch. In order to add such leaves, invocation data needs to be collected for other ports as well. Then, to add a failure or success leaf, the first fail or exit port for a call needs to be determined.

- `jupyter:print_transition_graph(+PredSpec, +FromIdx, +ToIdx, +LabelIdx)`:

  Like in the case of SLD trees, in order to display a transition graph, content for a file representing that graph is computed. The data is provided to the client as `print_transition_graph`. The transitions between the clauses of the predicate with specification `PredSpec` are computed with `findall/3`. For each of them, a line representing an edge is added to the computed file content.

- `jupyter:set_prolog_impl(+PrologImplementationID)`:

  In case the user requested to change the active Prolog implementation, all the server needs to do is to return the corresponding `PrologImplementationID` back to the client as `set_prolog_impl_id`. This way, the client is informed about the request and can handle it accordingly.

- `jupyter:update_completion_data`:

  The user requested to reload the data used for code completion. All built-in and exported predicates of currently loaded modules are computed and sent to the client as additional data `predicate_atoms`.

- `run_tests/0`, `run_tests/1`, or `run_tests/2`:

  Before running tests, it needs to be checked if any tests were defined by previous terms of the current request. In that case, the created test definition file is loaded. Afterwards, a `run_tests` query is handled like any other query. This way, any output representing the test results is displayed for the user.

- `trace/0`, `trace/1`, or `trace/2`:

  When running the implemented Prolog server, trace mode should not be turned on by a user. This is the case because debugging messages are not printed in a way that they can be read in and sent to the client. That is why rather than calling anything, an error message is output for calls of a `trace` predicate. Instead, `jupyter:trace/1` can be used to print the trace of a goal.

- `leash/1`:

  By default, leashing is turned off for all ports so that no user interaction is expected when tracing a call for `jupyter:trace/1`. As mentioned above, trace mode should not be turned on by a user. Therefore, changing the leashing would not have an effect anyway. Thus, an error message is output when it is tried to be changed.

- `abolish/1`, or `abolish/2`:

  In case of SICStus Prolog, predicates declared as discontiguous need to be treated specially. As explained before, for all predicates that were declared as such, a `jupyter_discontiguous/1` clause is added to the database. When removing a predicate with `abolish`, the corresponding clause needs to be removed as well.

5.3.5 jupyter_query_handling

As mentioned above, the output of a query is contained in the term result sent to the client. To access it, the module `jupyter_query_handling` provides predicates to redirect all output to a file and read it from that file. This is done when calling one of the predicates `call_with_output_to_file/3` and `call_query_with_output_to_file/7`. Before calling the goal provided as an argument, a file is opened. The corresponding stream is set as the current output, standard output, and error stream so that all output and debugging messages are written to the file. In case of SWI-Prolog, this is done with `set_stream/2`. For SICStus Prolog, the current output stream needs to be set with `set_output/1` and for the other ones, `set_prolog_flag/2` is called.

The given query is executed with the built-in predicate `catch/3` to catch any exception that might be thrown. Note that if the execution does not terminate (e.g. because something goes wrong unexpectedly), the server is stuck and cannot send a response to the client. In case of an exception, an error result with the corresponding error message is created. When asserting that term result, the message is retrieved from the given error term. This is done with `jupyter_query_handling:retrieve_message/2` by redirecting the error stream to a new file, printing the message with the built-in `print_message/2`, and reading the content from the file.

If an exception was thrown by the execution of a `jupyter:trace(TraceGoal)` goal, trace mode needs to be switched off. However, in case of SICStus Prolog, if breakpoints exist, debug mode is switched back on so that these breakpoints can be activated in following queries. Furthermore, for SICStus Prolog, some message lines might be output which should not be shown to the user. Therefore, these are removed after reading the output from the file. For instance, this is the case when producing the trace for `jupyter:trace/1` and switching on or off trace mode outputs corresponding messages.

Additionally, when calling the predicate `call_query_with_output_to_file/7` for a query, its runtime is computed with `statistics(walltime, Value)`. This and some more information about the query (including the ID of the request and an atom representing the query term) is asserted with the dynamic predicate `query_data/4`. The data is needed for some of the `jupyter` predicates. In contrast to that, the predicate `call_with_output_to_file/3` can be called when the output of a goal is needed, but no query data should be collected. For instance, this is the case when loading a test definition file. Note that for most of the terms handled specially by the module `jupyter_term_handling`, neither of these predicates is called. Therefore, no query data that could be accessed later is collected for them.

5.3.6 jupyter_variable_bindings

The `jupyter_variable_bindings` module provides predicates to enable reusing previous variable bindings in a query. It is based on the module `toplevel_variables` from version 8.4.2 of SWI-Prolog. Since the mechanism is already built-in for SWI-Prolog, predicates from this module are called for SICStus Prolog only.

When a query is executed which contains a term of the form `$Var`, it is tried to be replaced by the latest value that has been assigned to the variable `Var` by a previous query. In order for this to be valid syntax, `$` is defined as an operator. Before a query is called, the term can be expanded with `term_with_stored_var_bindings/4` by replacing all terms of the form `$Var` with the corresponding value. If there is no previous value for one of the variables, an exception is thrown.

The dynamic predicate `var_bindings/1` is used to remember the values of variables from previous queries. Its argument is a list containing `Name=Var` pairs, where `Name` is the name of a variable `Var` of the latest query in which a variable of this name was bound to a value.

After a successful query execution, the predicate `store_var_bindings/1` from this module is called with a list containing the names and values of all the variables occurring in the query. The only variables excluded from this are singleton variables starting with an underscore. The list of variables stored as `var_bindings/1` is updated with the variables that were instantiated by the current query.

5.3.7 jupyter

The Prolog server provides some special predicates which can be called by a user. To not block any predicate names so that they cannot be used by a user, most of them are defined in the module `jupyter` and need to be called with the module name expansion. The only exceptions are `retry`, `cut`, and `halt`, which can also be called without the module name.

As described above, some of these predicates need to be treated separately. They have to be the only goal of a query to be recognized correctly. Further, for those, the module defines clauses throwing an error when called so that the user is informed about this.

The following are the exported predicates which do not require special handling:

- `jupyter:help/0`

  For each predicate defined by the module there is a clause providing its predicate specification and the corresponding documentation as an atom. When calling `jupyter:help/0`, these documentation atoms are collected and output. The documentation is the same as used for predicate inspection.

- `jupyter:print_queries(+Ids)`

  This predicate causes the previous queries from requests with IDs in `Ids` to be output. Each query is printed on a separate line followed by a comma or full-stop in case of the last query. This way, they can be copied to a notebook cell and executed right away or expanded with a head to define a clause.

  As mentioned before, query data is collected by the module `jupyter_query_handling`. This is done by asserting a clause of the following predicate:

  ```
  query_data(CallRequestId, Runtime, TermData, OriginalTermData)
  ```

  The arguments `TermData` and `OriginalTermData` are terms of the following form:

  ```
  term_data(TermAtom, NameVarPairs)
  ```

  `TermAtom` is the representation of a term as an atom and `NameVarPairs` is a list of `Name=Var` pairs, where `Name` is the name of a variable `Var` from that term. This list is needed for the replacement of `$Var` terms. If the query contained any of those terms, `TermData` contains data about the term after the replacement. In that case, `OriginalTermData` is the data of the term before replacement. Otherwise, it equals the atom `same`.

For each of the queries to be printed, it is checked if it contains any `$Var` terms. If not, the query can be printed right away. Otherwise, the original term is expanded. If any of the previously printed queries contains the variable `Var`, it is replaced by the variable's name. Otherwise, the `$Var` term is not replaced.

- `jupyter:print_query_time/0`

  By calling `statistics(walltime, Value)` right before and after calling a query, its runtime is computed. In order for this information to be accessible later, it is asserted with the predicate `query_data/4` from the module `jupyter_query_handling`. Thereby, `jupyter:print_query_time/0` can output the latest previous goal followed by the time in milliseconds it took the query to complete.

- `jupyter:print_variable_bindings/0`

  In case of SWI-Prolog, variable bindings from previous queries are output with `print_toplevel_variables/0`. For SICStus Prolog, the variable bindings stored by the module `jupyter_variable_bindings` are displayed in the same format.

- `jupyter:trace(+Goal)`

  When it comes to debugging, there are major differences between SICStus and SWI-Prolog. Therefore, the trace of a goal needs to be computed differently.

  In case of SWI-Prolog, the tracer is switched on before calling the goal `Goal` and switched off afterwards. Further, debug mode is switched on again so that any breakpoints which might exist can still be activated after the query execution. Because of a `user:prolog_trace_interception/4` clause defined in `jupyter_server.pl`, debugging messages are printed to the current output without requesting user interaction.

  For SICStus Prolog, trace mode is also switched on before the call of the goal and switched off afterwards. Additionally, before activating trace mode, it needs to be checked if the debugger was already switched on. In that case, messages are output which should not be shown to the user. By asserting a clause that is checked when retrieving the output of the goal, these can be removed. Furthermore, the message printed when switching on trace mode is removed from the output. Afterwards, if any breakpoints exist, debug mode is switched back on again so that the debugger can stop at a breakpoint for following queries. Since all ports are unleashed, the debugger usually does not stop at an invocation to wait for user input. However, breakpoints are not affected by this.

6 Extensibility

The Herculog kernel was built to be highly extensible. For instance, it can be configured to run with a different Prolog server or Python kernel implementation. Thereby, it can run with another Prolog interpreter altogether, which might not be supported by default. The available configuration options are further explained in Section 6.1. In addition to supporting further Prolog implementations, which is described in more detail in Section 6.2, it might also be desirable to provide more convenience features, e.g. by adding `jupyter` predicates (see Section 6.3).

6.1 Configuration

The implemented Prolog Jupyter kernel can be configured by defining a Python configuration file. The kernel will look for files named `prolog_kernel_config.py` in the current working directory and the Jupyter config path, which can be retrieved with `jupyter --paths`. Note that if a config file exists in the current working directory, it overrides values from previously loaded configuration files. An example of such a file with an explanation of the options and their default values commented out can be found in the GitHub repository of the kernel [4]. The following options can be configured:

- `jupyter_logging`:

  If set to `True`, the logging level is set to `DEBUG` by the kernel so that Python debugging messages are logged. The messages are printed to the console from which the Jupyter application was started. Additionally, they can be accessed in JupyterLab with the menu *View* > *Show Log Console* or by right-clicking in a notebook and selecting *Show Log Console*. Then, in a drop down-menu, the log level needs to be set to level *Debug*.

  Note that this way, logging debugging messages can only be enabled after reading a configuration file. Thus, the user cannot be informed that no configuration file was loaded if none was defined at one of the expected locations. To switch on debugging messages by default, the development installation described in the GitHub repository can be followed and the logging level set to `DEBUG` in the file `kernel.py` (which contains a corresponding comment). However, note that this causes messages to be printed in the Jupyter console applications, which interferes with the other output.

- `server_logging`:

  If set to `True`, a log file is created by the Prolog server. The name of the file consists of the implementation ID preceded by `.prolog_server_log_`.

- `implementation_id`:

  The ID of the Prolog interpreter for which the server is started. To use the default SICStus or SWI-Prolog implementation, the ID `sicstus` or `swi` is expected.

- `implementation_data`:

  The implementation-specific data needed to run the Prolog server for code execution. This is required to be a dictionary containing at least an entry for the configured `implementation_id`. Each entry needs to define values for the following:

  - `failure_response`: The output which is displayed if a query fails.
  - `success_response`: Output if a query succeeds without any variable bindings.
  - `error_prefix`: The prefix printed for error messages.
  - `informational_prefix`: The prefix for informational messages.
  - `program_arguments`: Command line arguments to start the Prolog server. For SICStus and SWI-Prolog, the default server provided by Herculog can be used by configuring the string `"default"`.

  Additionally, a `kernel_implementation_path` can be provided, which needs to be an absolute path to a Python file. The corresponding module needs to define a subclass of `PrologKernelBaseImplementation` named `PrologKernelImplementation`. As described in Section 6.2.3, this can be used to override some basic kernel behavior.

Note that if the `program_arguments` for SICStus Prolog are invalid (e.g. the source code file does not exist), the kernel waits for a response from the server which it will never receive. In that state, it cannot log any exception and instead, nothing happens. To facilitate finding the cause of the error, before trying to start the Prolog server, the arguments and the directory from which they are tried to be executed are logged (if logging is configured).

6.2 Supporting Further Prolog Implementations

For code execution, the Herculog kernel communicates with a Prolog server process over JSON-RPC 2.0. Basically, by replacing this server with another one, the kernel can easily be extended for a different Prolog implementation. As described above, this can be done by configuring the kernel and specifying `program_arguments` to start a differing server process. In the following, some details about the server are given which might serve as guidance for creating a new one.

Most of the server code is compatible with both SICStus and SWI-Prolog. Since this is expected to be similar for other Prolog implementations as well, extending the existing server should usually suffice (see Section 6.2.1). However, in rare cases it might be required to write a new server from scratch (see Section 6.2.2). For instance, when trying to support Prolog derivatives like Mercury [48]. Additionally, supporting another Prolog interpreter could require adjustment of basic kernel behavior by overriding the Python base implementation class `PrologKernelBaseImplementation` (see Section 6.2.3).

6.2.1 Extending the Existing Prolog Server

Originally, Herculog was built for SICStus Prolog only. Later, the server code was extended to support SWI-Prolog as well by making use of conditional compilation. Since most code is compatible with both implementations, no major adjustments were required to support basic functionality such as query execution. However, this was different for some other features, especially the ones provided by the module `jupyter`.

When extending the server code for further Prolog interpreters, similar portability obstacles as for SWI-Prolog might occur. Since the issues or general differences might serve as guidance for potential required adjustments, some of them are listed in the following. Additionally, the tests from the file `jupyter_server_tests.pl` can help to detect problems. However, note that since test definitions may also differ considerably depending on the implementation, running them might not be possible without major adjustments.

The differences between SICStus and SWI-Prolog encountered during the server extension include the following:

- Some of the predicates which are built-in for one implementation need to be loaded from a library for the other one. Furthermore, some libraries are named differently and predicates with the same functionality are provided under different names. Examples for the latter case include the following:
  - For SWI-Prolog, new message types are defined with `prolog_exception_hook/4` instead of `user:portray_message/2`.
  - Instead of setting standard output and error streams with `set_prolog_flag/3`, `set_stream/2` needs to be called for SWI-Prolog.
  - In case of SWI-Prolog, DCG rule terms need to be expanded with the predicate `dcg_translate_rule/2` instead of `expand_term/2`.
- The arguments for starting the server process differ.
- Reading in terms from an atom is more complex in SWI-Prolog when a helpful error message is required in case of syntax errors.
- To write a message to a single line with `json_write/3`, the option `width(0)` needs to be specified instead of `compact(true)` for SICStus.
- In SWI-Prolog, using `$Var` terms to access previous variable bindings is built-in.
- The handling of breakpoints and debugging in general differs. In SICStus Prolog, by turning leashing off for all ports, the debugger does not stop at any invocation without a breakpoint to wait for user interaction. For SWI-Prolog, this needs to be handled by defining a clause for `user:prolog_trace_interception/4`.
- In case of SICStus Prolog, declarations must not appear in a query. Instead, they need to be loaded from a file.

- Even though the SICStus `library(plunit)` is based on the one from SWI-Prolog, there are major differences between them. These include the following:
  - The options available for `begin_tests/2`, `test/2`, and `run_tests/2` differ.
  - With SWI-Prolog, a `true` option cannot be contained multiple times in the options specified for a `test/2` clause definition, which is possible for SICStus.
  - The loading of files defining test units differs considerably. For SICStus Prolog, the best option for the server is to write all test units to the same file, which is named the same for every request. For SWI-Prolog, this would cause an error if the file did not define a test unit any more which was loaded from it previously. Instead, for each test unit, a separate file is created which contains the unit name so that a unit can be reloaded from the same file again.
- In case of SICStus Prolog, a website listing links to documentation of all predicates is accessed for providing inspection information. For SWI-Prolog, there is the predicate `help/1`, with which help for predicates can be output.

Debugging

Usually, if the execution of a goal causes an exception, the corresponding Prolog error message is computed and displayed in the Jupyter frontend. However, in case something goes wrong unexpectedly or the query does not terminate, the Prolog server might not be able to send a response to the client. In that case, the user can only see that the execution does not terminate without any information about the error or output that might have been produced. However, it is still possible to write logging messages and access any potential output, which might facilitate finding the cause of the error.

Debugging the server code is not possible in the usual way by tracing invocations. Furthermore, all messages exchanged with the client are written to the standard streams. Therefore, printing helpful debugging messages does not work either. Instead, if `server_logging` is configured, messages can be written to a log file by calling `log/1` or `log/2` from the module `jupyter_logging`. By default, only the responses sent to the client are logged.

When a goal is executed, all its output is written to a file named `.server_output`, which is deleted afterwards by `jupyter_query_handling:delete_output_file`. If an error occurs during the actual execution, the file cannot be deleted and thus, the output can be accessed. Otherwise, the deletion can be prevented.

Furthermore, the server might send a response which the client cannot handle. In that case, logging for the Python code can be enabled by configuring `jupyter_logging`. For instance, by default, the client logs the responses received from the server.

6.2.2 Writing a New Prolog Server

Instead of basing a new Prolog server on the existing one, it can be created from scratch. In that case, it is required to know the types of messages which are sent. As mentioned before, the Python kernel implementation communicates with the Prolog server by sending JSON messages over the standard streams according to the JSON-RPC 2.0 protocol. Section 4.3 gives an introduction to the general structures of such messages by providing examples. The format of the actual messages that are sent by the kernel's Prolog server is explained in Section 4.4.

Furthermore, to provide some more functionality than the basic one, for some requests, additional data needs to be sent by the server. This can be done by sending a value for one of the members `predicate_atoms`, `print_sld_tree`, `print_table`, `print_transition_graph`, `retracted_clauses`, and `set_prolog_impl_id`. Most of them are sent for special `jupyter` predicates and the values which are expected are explained in Section 5.3. Further, the description of the server implementation in that section gives a good overview of what behavior is expected. Additionally, the tests in `jupyter_server_tests.pl` can serve as help for getting to know the format of responses that is expected by the client for specific request messages.

Note that once the Python kernel implementation tries to read a response message, it cannot be stopped from waiting without shutting down the kernel. Thus, for every request sent to the Prolog server, a single response messages is expected on a single line.

6.2.3 Extending the Kernel Implementation

The actual kernel code determining the handling of requests is not implemented by the class `PrologKernel` itself. Instead, for every request, a `PrologKernelBaseImplementation` method is called. By creating a subclass named `PrologKernelImplementation` and defining the path to the corresponding Python file in a config file as `kernel_implementation_path`, the actual implementation code can be replaced. If no file containing a valid class is found, a default implementation is used instead. Besides `PrologKernelBaseImplementation`, there are default classes for SICStus and SWI-Prolog. The only basic kernel code that differs between the implementations is for predicate inspection.

The `PrologKernelImplementation` needs to handle the starting of and communication with the server. The following main methods are defined for that, which can be overriden:

- `start_prolog_server`:

  Tries to (re)start the Prolog server process with the configured arguments.

- `kill_prolog_server`:

  Terminates the Prolog server process if it is still running.

- `retrieve_predicate_information`:

  Requests data from the Prolog server needed for code completion and inspection.

- Request methods which are called by the `PrologKernel` for a corresponding request received from the frontend:
  - `do_shutdown`: Executes kernel-specific shutdown actions.
  - `do_execute`: Executes code and computes its output.
  - `do_complete`: Finds possible predicate completions.
  - `do_inspect`: Computes introspection results for predicates.

- `server_request`:

  Sends a request to the Prolog server and reads the JSON response before deserializing and returning it.

- `handle_success_response`:

  Handles a success response by computing output for each term result and sending it to the frontend.

- `handle_error_response`:

  Handles an error response by sending an error message to the frontend.

- `send_response_display_data`:

  Sends a response to the frontend as plain text.

- `handle_additional_data`:

  Handles additional data which might be provided by a term result. Checks if the given dictionary contains any `predicate_atoms`, `print_sld_tree`, `print_table`, `print_transition_graph`, `retracted_clauses`, or `set_prolog_impl_id` key. If so, one of the following methods is called, the purpose of which can easily be inferred from their names:
  - `handle_completion_data_update`
  - `handle_print_graph`
  - `handle_print_table`
  - `handle_retracted_clauses`
  - `handle_set_prolog_impl`

6.3 Adding Convenience Predicates

All special predicates of the Herculog kernel are provided with the module name expansion `jupyter`. To add more such convenience features, further `jupyter` predicates can be implemented. However, one needs to differentiate between predicates requiring additional data such as the variable name and variable pairs read from the corresponding term and ones that can be processed without further information.

The first kind of predicates is not actually defined in the `jupyter` module's source file. Instead, the execution is handled by the module `jupyter_term_handling`. Before processing a query, it is checked if it corresponds to a special one with `handle_query_term_/8`, where the first argument is the query term. Therefore, by adding a clause, a new predicate can be supported. The additional data accessible at that point is the stack of queries that can be retried, the variable name and variable pairs of the term, data about the query before potential `$Var` terms were replaced, and information about if the query was called as a directive. Note that in this case, the predicate can only be determined as a special one if it is the only one in a term. In order to inform the user about this in case it is called differently, a clause throwing an error can be added to the `jupyter` source file.

If none of the listed additional data is required, a new predicate can be defined in the `jupyter` module instead. For both types of convenience features, there is the special case of data about previous queries which can be accessed. This data is asserted by the module `jupyter_query_handling` as `query_data/4` after calling a query. It includes the ID of the request, the query runtime, an atom representing the query before any `$Var` term replacements (if there were any), and an atom corresponding to the actual query term that was called.

In all cases, the specification of the newly defined predicate should be added to the predicate list of the module declaration so that it can be shown for code completion. Furthermore, in order to support inspection, a new clause for `predicate_doc/2` providing a documentation string needs to be defined.

7 Use Cases

One of the main reason for implementing a Jupyter kernel for Prolog was that using Jupyter notebooks could facilitate teaching Prolog in an educational context. This section presents some more specific use cases. Again, since Jupyter Notebook is to be replaced by JupyterLab, any Screenshots are taken from the latter application. However, some explanations are given for Jupyter Notebook as well.

Note that all the example notebook documents mentioned in the following are available in the GitHub repository of the Herculog kernel [4].

7.1 Prolog REPL Extension

As stated before, any Jupyter kernel can be used with the applications Jupyter Console and Qt Console. Since Herculog provides convenience features in addition to basic Prolog REPL functionality, it can be seen as an extension of the REPL. One of the major advantages of this extension is that predicates can be defined on the fly. Additionally, by not requiring a terminating full-stop, a main reason for queries not to be run right away is eliminated. Furthermore, code completion is supported for some predicates.

Executing Prolog in a Jupyter notebook provides even more functionality. In addition to producing special output such as tables and requesting inspection, the documents can be saved. Thus, a REPL session can be looked at and executed again later and even distributed to others so that they can execute it. However, all cells can be changed at any time. Therefore, it should be mentioned that the order of the cells might not correspond to the order in which they can actually be executed. Furthermore, the output might not be reproducible at all with the provided code.

7.2 Source Code Documentation

Since a Jupyter notebook document can contain code accompanied by text, it is ideal for source code or other documentation. In addition to code cells defining Prolog predicates or executing queries, text can be written explaining those predicates and their execution. Moreover, it can facilitate understanding the code even better since it can easily be executed as is or adjusted to experiment with it. The Herculog GitHub repository contains notebooks explaining its features for SICStus and SWI-Prolog. These can be seen as examples of documentation notebooks.

To distribute a notebook to others without them needing a Jupyter application, the document can be exported to other formats with nbconvert. Among others, it can be converted to a PDF, LaTeX, or HTML file. Additionally, the Jupyter applications Voilà and nbviwer introduced before facilitate the distribution of notebook documents.

Differentiating Term Types

- Each code cell can contain multiple terms: **clause definitions**, **directives** and **queries**

Query

- Single term without body in a cell

```
In [4]: X = [1,2,3], append(X, [4,5,6], Z).

        X = [1,2,3],
        Z = [1,2,3,4,5,6]
```

Figure 29: A slide created from a Jupyter notebook with RISE.

7.3 Lecture Slides

Since Jupyter notebooks can facilitate explaining source code, creating lecture notes is an obvious use case. While a notebook can be presented or distributed as is, it can also be converted to other formats, such as HTML slides. Furthermore, a live slideshow supporting cell execution can be created with the Jupyter Notebook extension RISE [1].

For every slide, arrows are shown in the bottom-right corner (see Figure 29). The right one of these is enabled if there is a next slide. If there is a sub-slide or fragment, the down arrow can be clicked. To configure this, for each cell, a slide type can be chosen from one of the following options in a drop-down menu:

- **Slide**: The content of the cell will appear on a new slide.
- **Sub-Slide**: The cell content will appear on a new sub-slide.
- **Fragment**: With the down arrow, the content is added to the current (sub-)slide.
- **Skip**: Not part of any slide.
- **Notes**: Not part of any slide.
- **-**: When the last part of the slide or sub-slide is shown (e.g. by clicking the down arrow), the cell's content is added to the current slide or sub-slide.

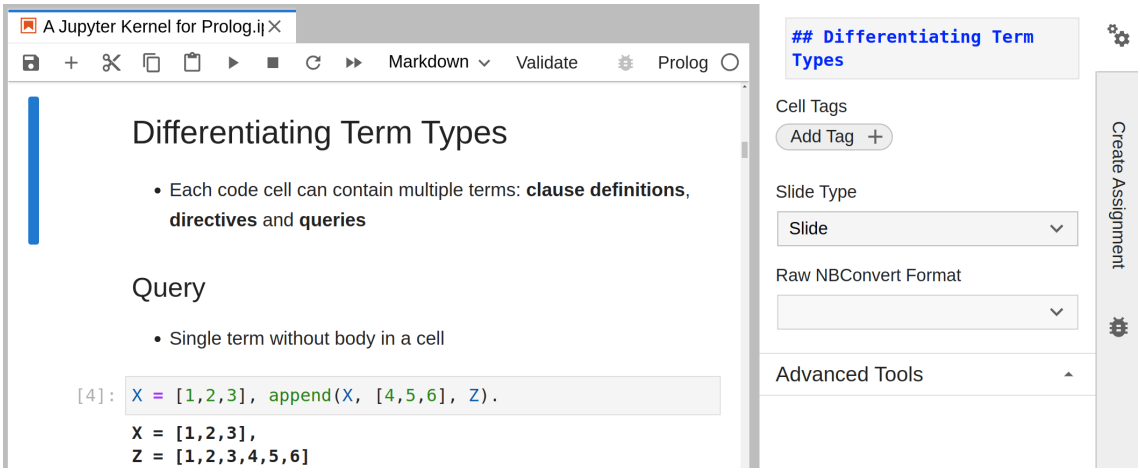


Figure 30: Defining the slide type in JupyterLab.

In JupyterLab, the menu can be accessed in the *Property Inspector* tab from the right side bar (see Figure 30). In Jupyter Notebook, a cell toolbar showing the menu in the upper right corner of a cell can be enabled by clicking *View* > *Cell Toolbar* > *Slideshow*.

7.4 Assignments

A further use case for Jupyter notebooks are student assignments. These can be created as documents containing task descriptions, sample or skeleton code, and cells for the students' solutions. In order to facilitate this, **nbgrader** [21, 44] was developed, of which the application with Herculog is presented in the following. Note that in most cases, the usage for JupyterLab is described only. However, most steps can also be performed from the command line.

With nbgrader, instructors can create assignments as Jupyter notebooks and generate student versions from it. Furthermore, it aids grading, especially by providing automatic grading for some types of notebook cells. Additionally, the tool can be integrated with **JupyterHub** [39], an application for serving notebooks for multiple users. This simplifies the process of distributing and collecting assignments for instructors as well as accessing and submitting them for students. To inform users about the extensive functionality, nbgrader comes with detailed documentation as well as examples explaining the workflow.

Setup

As described in the documentation, each nbgrader course directory has a specific structure [23]. Summarized briefly, the `source` and `release` directories contain the instructor and student assignment notebooks respectively. Notebooks handed in by students are collected in `submitted` with a separate directory for each student. The same is the case for the `autograded` and `feedback` directories, which contain executed notebooks after autograding and generated HTML feedback files to be returned to the students respectively. With the `nbgrader quickstart` command, an example source directory with some source files can be created as a quick starting point.

When installing nbgrader, any required extensions are installed as well. For instance, this includes an extension for JupyterLab adding the toolbar *Create Assignment* with which a type can be specified for each cell. For Jupyter Notebook, a corresponding cell toolbar can be enabled by clicking *View* > *Cell Toolbar* > *Create Assignment*. As described in more detail below, the type of a cell determines how its content may be changed when generating student notebooks.

Among other options, some specifics of the cell types can be configured by creating a `nbgrader_config.py` file, which should normally be located in the directory from which nbgrader commands are run (e.g. to generate the student versions of assignments). The Herculog GitHub repository provides a directory with example course files including such a configuration file. Note that in order for all nbgrader functionality to work properly, the Jupyter application needs to be started from the course directory.

During generation of the student version of an assignment, (parts of) the content of some cells are removed or replaced. For instance, if an answer cell contains the delimiters `BEGIN SOLUTION` and `END SOLUTION`, anything between them will be replaced with `YOUR ANSWER HERE` for Markdown cells or a code stub in case of code cells. If no solution delimiters are present, the entire cell content will be replaced. Otherwise, the cell can contain an answer stub providing some help for the students. By default, there are code stubs for programming languages such as Python and Java. In order to add one for Prolog, the config option `ClearSolutions.code_stub` needs to be set. In the example from the GitHub repository, the following stub is specified as replacement:

```
% YOUR CODE HERE
throw(jupyter(no_answer_given)).
```

Note that the way the stub is written and Prolog terms are handled, the markers should only be placed as the last goal of a clause or enclose all code of a cell (the student version cell can still contain comments). Otherwise, the generated content is no valid Prolog code.

Part A (2 points)

Write a predicate `my_append(List1, List2, List3)`, where `List3` is a list consisting of all the elements of the list `List1` followed by the elements of the list `List2`.

[ ]:
```
set_prolog_flag(informational, off).
```

[ ]:
```
my_append([], Res, Res).
my_append([Head|Tail], List, [Head|Res]) :-
    my_append(Tail, List, Res).
```

[ ]:
```
:- begin_tests(my_append).

test(append_two_lists, [true(List3 = [1, 2, 3, 4])]) :-
    my_append([1, 2], [3, 4], List3).
% BEGIN HIDDEN TESTS
test(split_list, [true((List1 = [], List2 = [1, 2])), nondet]) :-
    my_append(List1, List2, [1, 2]).
% END HIDDEN TESTS

:- end_tests(my_append).

?- run_tests(all, [failed(0)]).
```

(a) Instructor version.

Before turning the assignment in, make sure to replace any `YOUR CODE HERE` or "YOUR ANSWER HERE". Also check your solution by running all cells after restarting the kernel.

Part A (2 points)

Write a predicate `my_append(List1, List2, List3)`, where `List3` is a list consisting of all the elements of the list `List1` followed by the elements of the list `List2`.

[ ]:
```
set_prolog_flag(informational, off).
```

[ ]:
```
% YOUR CODE HERE
throw(jupyter(no_answer_given)).
```

[ ]:
```
:- begin_tests(my_append).

test(append_two_lists, [true(List3 = [1, 2, 3, 4])]) :-
    my_append([1, 2], [3, 4], List3).

:- end_tests(my_append).

?- run_tests(all, [failed(0)]).
```

(b) Student version.

Figure 31: An assignment created as a Jupyter notebook with nbgrader.

Cell Types

The type of a cell determines how its content may be adjusted for a student notebook. Examples of the instructor and student version of a notebook can be seen in Figure 31. Note that the path to the file containing the header for the student version is configured.

Due to the vast amount of functionality available for the cell types, not all details can be listed here. The following information is considered to be especially helpful:

- **Manually graded answer**:

  Defines a cell of which the content (usually a free-response answer) needs to be graded manually. For these cells, the number of points the answer is worth needs to be specified. Unless the cell contains `BEGIN SOLUTION` and `END SOLUTION` markers, the entire content will be replaced with a configurable text or code stub.

- **Manually graded task**:

  This type of cell is similar to a manually graded answer cell. However, the task is not to be performed in the cell itself, but with other cells instead. This can be useful if in order to solve a task, multiple cells need to be created. A special syntax allows the instructor to insert text which will be visible in the feedback and when grading, but not in the student version. Any text between the delimiters `BEGIN MARK SCHEME` and `END MARK SCHEME` will be removed for the student version.

- **Autograded answer**: (available for code cells only)

  If students are to write code, this type of cell can be selected. By using the `BEGIN SOLUTION` and `END SOLUTION` delimiters, a method stub can be provided for students. Otherwise, the entire cell content will be replaced with the configured code stub. The points the answer is worth are not specified for the cell itself. Instead, they need to be given for the tests used to grade the given answer (see the next bullet point).

- **Autograder tests**: (available for code cells only)

  These cells contain tests which are run during autograding. They can also be run with the *Validate* button to validate the solution. This can be useful for both, instructors before generating the student version and students before submitting the assignment. If the tests succeed, students will receive the number of specified points. Note that if no points are set, a failure of this kind of cell does not affect the validation result.

  The content of these cells cannot be modified by students. By default, the tests will be visible in the student version. However, there is special syntax to hide tests. Any text between the `BEGIN HIDDEN TESTS` and `END HIDDEN TESTS` markers is removed when generating the student version.

- **Read-only**:

  As the name suggests, a cell of this type cannot be modified by a student.

Figure 32: Validation error window.

Validation and PlUnit Tests

Once assignments have been created, the student versions can be generated either with the command `generate_assignment` or by using the formgrader extension. The latter can be accessed with the JupyterLab menu *Nbgrader* > *Formgrader*. In Jupyter Notebook, the corresponding nbgrader menus including *Formgrader* are shown in the top bar instead. Under *Manage Assignments*, all assignments of the course are listed (see Figure 33a). By clicking on the *Generate* button for one of them, the corresponding student version is created, which can then be accessed with the *Preview* button.

Before submitting the solution as well as before releasing the assignment to students, a user can validate the given answers with the *Validate* button. Basically, this causes the kernel to be restarted and all cells to be run. However, validation can succeed even if there are erroneous cells. The only cells affecting the validation result are autograder tests for which more than zero points are specified. If any of those cells fails or causes an exception, the validation fails. In that case, the content of the failing cell is displayed as well as an error trace. For Prolog, this is the first failing term's result (i.e. the output and error message or failure text). When validating the example notebook from Figure 31 without any adjustments, the window shown in Figure 32 pops up, which displays the result.

Name	Due Date	Status	Edit	Generate	Preview	Release	Collect	# Submissions	Generate Feedback	Release Feedback
ps1	None	draft						2		

Add new assignment...

(a) List of assignments.

Student Name	Student ID	Timestamp	Status	Score	Autograde	Generate Feedback	Release Feedback
Meier, Tina	timei100	None	graded	0 / 2			
Schneider, Martin	masch100	None	graded	2 / 2			

(b) List of submissions.

Figure 33: Data available in the nbgrader *Formgrader* view.

When it comes to test results, again, SICStus and SWI-Prolog differ significantly. In case of SICStus Prolog, when running tests, by default, the user is informed about the number of tests that passed or failed. However, even if there are failing tests, the corresponding `run_tests` goal can succeed anyway. Thus, in order to cause the validation to fail for failing tests, `run_tests(all, [failed(0)]).` can be called. For SWI-Prolog, calling `run_tests/0` instead is sufficient as the goal fails for any failing test.

Furthermore, it might be advisable to turn off informational messages for SICStus Prolog. Otherwise, long messages are printed for the loading of `library(plunit)` and the file in which the tests are defined. This can make the error message that might be shown for a validation unnecessarily complex. Note that this also causes the number of successful and failed tests not to be printed. However, in case of a test failure, a corresponding message is output anyway.

As mentioned above, any cell containing a failing term or resulting in an exception causes the validation to fail. Therefore, it is not mandatory to use PlUnit tests for autograder tests. Instead, it suffices to call corresponding queries right away.

Grading

When collecting the submissions in the `submitted` directory, it is expected to contain a subdirectory for each student with the corresponding notebooks. The structure should look like the following:

```
submitted/{student_id}/{assignment_id}/{notebook_id}.ipynb
```

The list of assignments as shown in Figure 33a displays the number of submissions. By clicking on it, all of them can be accessed (see Figure 33b). With the *Autograde* button, individual submissions can be autograded. From the command line, the autograder can be run for all students at once with the `nbgrader autograde` command. In either case, the autograded versions of the notebooks will be written to the `autograded` directory.

Afterwards, the submissions are available for manual grading via the *Manual Grading* button. Now, instructors can award points and comment on the students' answers. By clicking on the *Generate Feedback* button, HTML files containing feedback are created in a `feedback` directory. These show the notebook cells with output and additional nbgrader information such as scores and comments.

8 Future Work

The Herculog kernel provides most basic features as well as some more specific functionality for programming with Prolog. However, it could still be improved by implementing even more features as discussed in the following. While some of them correspond to rather minor improvements of convenience features, the implementation of others would imply more extensive changes.

The kernel already supports creating and displaying a graph resembling an SLD tree. However, it contains nodes for call ports only. By adding leaves representing failure or success (by showing the empty clause), it can be turned into an actual SLD tree. Further, to allow customization of the visualization, the corresponding predicate could be extended with an additional argument containing options for the graph creation. This would also be beneficial for the other `jupyter` predicate creating a transition graph. Moreover, those graphs cannot contain nodes without any edges. Therefore, supporting such nodes would be a practical improvement.

Another feature that would benefit from further implementation is introspection. So far, the documentation of Prolog predicates can be displayed. Since Herculog supports reusing bindings from previous queries with `$Var` terms, it would be convenient to see the latest binding when inspecting a variable. This way, the user could see that value without the need of executing a query to retrieve it. Furthermore, code completion can be used for predicates which are built-in or exported by a loaded module. As predicates defined in a Jupyter notebook are added as dynamic clauses, completion does not work for them. Thus, adding support for this would make for another helpful adjustment.

As pointed out by the analysis of available Prolog implementations and their portability, there are major differences between them [29]. More precisely, the available features differ considerably. Thus, it would be an immense gain to combine the strengths of several interpreters by computing data with one and then reusing it with the other implementation. The Herculog kernel can already be connected with multiple Prolog instances and it is possible to switch between them on the fly. Once a Prolog server was started for one implementation, it is kept running until the server is interrupted, restarted, or shutdown. Since reusing results from one Prolog server for another one is likely to be relatively easy, implementing this idea should not take too much effort.

Moreover, executing queries with multiple Prolog instances *at the same time* could be interesting. The benchmarking mechanism can then aid in comparing the performance. This could be simplified by implementing a predicate switching on or off printing the execution time for every query by default. In order for this to be even more helpful, an obvious improvement for the kernel is the extension for further Prolog implementations or even different versions of the same implementation. By facilitating detecting differences in their behavior, this could especially be of interest for Prolog implementors and (ISO) standard maintainers.

9 Conclusion

While there were multiple options for supporting literate programming for SICStus Prolog, to the best of the author's knowledge, writing a Jupyter kernel was the preferable one for this thesis. Even though this means that the notebook application is not tailored for Prolog and thus cannot provide some desirable functionality (such as dedicated cell types for predicate definitions or queries and interactive debugging support), this shortcoming is outweighed by the benefits that come with Jupyter.

The implemented kernel is based on the default IPython kernel and at first, it was created for SICStus Prolog only. During the development process, it was extended for SWI-Prolog as well. Additionally, it was made extensible for further implementations by configuration. With some adjustment, the kernel could even be used to combine strengths of different interpreters or facilitate the comparison of them. Code is executed by communicating with a Prolog server, which is implemented with conditional compilation to be compatible with both SICStus and SWI-Prolog. Messages are written to the standard input and output streams and follow a JSON-RPC protocol.

The extension for SWI-Prolog exposed the difficulties that result from the lack of portability for some Prolog features. The code for basic functionality was mostly compatible with both implementations. However, more complex features such as retrieving the stack trace required significant adjustments. Standards for libraries (e.g. for tests) and debugging would have facilitated this work considerably.

In addition to basic functionality for SICStus and SWI-Prolog, the implemented kernel provides some more advanced convenience features. For instance, functionality such as code completion and inspection or the ability to define programs on the fly and execute queries without a terminating full-stop offers advantages over programming in a Prolog REPL. Further, as pointed out, Jupyter does not only allow the handling of notebooks for interactive programming and code documentation. Since the documents can additionally be turned into lecture slides and aid in creating and grading student assignments, Herculog might be a good tool for teaching Prolog.

List of Figures

References

[1] Damián Avila. *RISE 5.7.1*. URL: https://rise.readthedocs.io/en/stable/index.html (visited on 09/26/2022).

[2] Tessel Bogaard et al. "SWISH DataLab: A Web Interface for Data Exploration and Analysis". In: *Proceedings BNAIC*. Ed. by Tibor Bosse and Bert Bredeweg. Vol. 765. Communications in Computer and Information Science. Springer, 2016, pp. 181–187. DOI: 10.1007/978-3-319-67468-1_13.

[3] Carl Friedrich Bolz. "A Prolog Interpreter in Python". In: *Bachelorsthesis. HHU Düsseldorf* (2007).

[4] Anne Brecklinghaus. *Prolog Jupyter Kernel*. URL: https://github.com/anbre/prolog-jupyter-kernel (visited on 09/26/2022).

[5] Anne Brecklinghaus and Philipp Körner. *A Jupyter Kernel for Prolog*. Tech. rep. Proceedings WLP'22. DFKI/GI. URL: https://wlp2022.dfki.de/data/papers/005.pdf.

[6] Calysto. *Calysto Prolog*. URL: https://github.com/Calysto/calysto_prolog (visited on 09/26/2022).

[7] Calysto. *Metakernel*. URL: https://github.com/Calysto/metakernel (visited on 09/26/2022).

[8] Mats Carlsson and Per Mildner. "SICStus Prolog – The first 25 years". In: *Theory and Practice of Logic Programming* 12.1-2 (2012), pp. 35–66.

[9] Davide Cervone. "MathJax: A Platform for Mathematics on the Web". In: *Notices of the AMS* 59.2 (2012), pp. 312–316. DOI: 10.1090/noti794.

[10] Alain Colmerauer and Philippe Roussel. "The Birth of Prolog". In: *History of Programming languages—II*. Ed. by Thomas J. Bergin Jr. and Richard G. Gibson Jr. ACM, 1996, pp. 331–367. DOI: 10.1145/234286.1057820. URL: http://doi.acm.org/10.1145/234286.1057820.

[11] Luca Corbatto. *jswipl*. URL: https://github.com/targodan/jupyter-swi-prolog (visited on 09/26/2022).

[12] Janez Demšar. *Picstus*. URL: https://github.com/janezd/picstus (visited on 09/26/2022).

[13] John Ellson et al. "Graphviz— Open Source Graph Drawing Tools". In: *Graph Drawing*. Ed. by Petra Mutzel, Michael Jünger, and Sebastian Leipert. Springer, 2002, pp. 483–484. DOI: 10.1007/3-540-45848-4_57.

[14] Muhammad Fawi. *pytholog*. URL: https://github.com/MNoorFawi/pytholog (visited on 09/26/2022).

[15] Marijn Haverbeke. *CodeMirror*. URL: https://codemirror.net/ (visited on 09/26/2022).

[16] IPython Development Team. *IPython Kernel for Jupyter*. URL: https://github.com/ipython/ipykernel (visited on 09/26/2022).

[17] Jupyter Development Team. *Making kernels for Jupyter*. URL: https://jupyter-client.readthedocs.io/en/stable/kernels.html (visited on 09/26/2022).

[18] Jupyter Development Team. *Making simple Python wrapper kernels*. URL: https://jupyter-protocol.readthedocs.io/en/latest/wrapperkernels.html (visited on 09/26/2022).

[19] Jupyter Development Team. *Messaging in Jupyter*. URL: https://jupyter-client.readthedocs.io/en/latest/messaging.html (visited on 09/26/2022).

[20] Jupyter Development Team. *nbconvert Documentation*. URL: https://nbconvert.readthedocs.io/en/latest/ (visited on 09/26/2022).

[21] Jupyter Development Team. *nbgrader*. URL: https://nbgrader.readthedocs.io/en/stable/ (visited on 09/26/2022).

[22] Jupyter Development Team. *The Notebook file format*. URL: https://nbformat.readthedocs.io/en/stable/format_description.html (visited on 09/26/2022).

[23] Jupyter Development Team. *What is nbgrader?* URL: https://nbgrader.readthedocs.io/en/stable/user_guide/what_is_nbgrader.html#course-directory (visited on 09/26/2022).

[24] *Jupyter kernels*. URL: https://github.com/jupyter/jupyter/wiki/Jupyter-kernels (visited on 09/26/2022).

[25] Jupyter Team. *Jupyter Notebook Documentation*. URL: https://jupyter-notebook.readthedocs.io/en/stable/notebook.html (visited on 09/26/2022).

[26] Thomas Kluyver and Philipp A. *IRkernel*. URL: https://irkernel.github.io/ (visited on 09/26/2022).

[27] Thomas Kluyver et al. "Jupyter Notebooks — a publishing format for reproducible computational workflows". In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas* (2016), pp. 87–90.

[28] Donald E. Knuth. "Literate Programming". In: *The Computer Journal* 27.2 (Jan. 1984), pp. 97–111. DOI: 10.1093/comjnl/27.2.97. URL: https://doi.org/10.1093/comjnl/27.2.97.

[29] Philipp Körner et al. "Fifty Years of Prolog and Beyond". In: *Theory and Practice of Logic Programming* (2022), pp. 1–83. DOI: 10.1017/S1471068422000102.

[30] Johan Mabille and Sylvain Corlay. *xeus Documentation*. URL: https://xeus.readthedocs.io/en/latest/ (visited on 09/26/2022).

[31] Mats Carlsson et al. *SICStus Prolog User's Manual*. 4.7.1. RISE Research Institutes of Sweden AB, Jan. 2022, pp. 289–348. URL: https://sicstus.sics.se/sicstus/docs/4.7.1/pdf/sicstus.pdf.

[32] Max Mensing. *SWI-Prolog-Kernel*. URL: https://github.com/madmax2012/SWI-Prolog-Kernel (visited on 09/26/2022).

[33] Chris Meyers and Fred Obermann. *Prolog in Python - Part 3*. URL: http://openbookproject.net/py4fun/prolog/prolog3.html (visited on 09/26/2022).

[34] Matt Morley. *JSON-RPC 2.0 Specification*. URL: https://www.jsonrpc.org/specification (visited on 09/26/2022).

[35] Keith O'Hara, Douglas Blank, and James Marshall. "Computational notebooks for AI education". In: *The Twenty-Eighth International Flairs Conference*. 2015.

[36] Spencer Park. *Jupyter JVM BaseKernel*. URL: https://github.com/SpencerPark/jupyter-jvm-basekernel (visited on 09/26/2022).

[37] Fernando Pérez and Brian E. Granger. "IPython: a System for Interactive Scientific Computing". In: *Computing in Science and Engineering* 9.3 (May 2007), pp. 21–29. DOI: 10.1109/MCSE.2007.53. URL: https://ipython.org.

[38] *Predicate Index (SICStus Prolog)*. URL: https://sicstus.sics.se/sicstus/docs/latest/html/sicstus.html/Predicate-Index.html (visited on 09/26/2022).

[39] Project Jupyter. *JupyterHub*. 2016. URL: https://jupyterhub.readthedocs.io/en/stable/ (visited on 09/26/2022).

[40] Project Jupyter. *JupyterLab Documentation*. 2018. URL: https://jupyterlab.readthedocs.io/en/stable/ (visited on 09/26/2022).

[41] Project Jupyter. *JupyterLab extension-cookiecutter-ts*. URL: https://github.com/jupyterlab/extension-cookiecutter-ts (visited on 09/26/2022).

[42] Project Jupyter. *nbviewer*. URL: https://nbviewer.org/ (visited on 09/26/2022).

[43] Project Jupyter. *Project Jupyter*. URL: https://jupyter.org/ (visited on 09/26/2022).

[44] Project Jupyter et al. "nbgrader: A Tool for Creating and Grading Assignments in the Jupyter Notebook". In: *Journal of Open Source Education* 2.16 (2019), p. 32. DOI: 10.21105/jose.00032. URL: https://doi.org/10.21105/jose.00032.

[45] *Reuse of top-level bindings*. URL: https://www.swi-prolog.org/pldoc/man?section=topvars (visited on 09/26/2022).

[46] Alex Riina. *Extending IPython to run Prolog from a Jupyter Notebook*. URL: http://alexriina.com/2019/02/11/ipython-prolog/ (visited on 09/26/2022).

[47] Rob Śliwa. *Simple Prolog Interpreter in Python*. URL: https://github.com/robjsliwa/pyprolog (visited on 09/26/2022).

[48] Zoltan Somogyi, Fergus Henderson, and Thomas C. Conway. "The Execution Algorithm of Mercury, an Efficient Purely Declarative Logic Programming Language". In: *Journal of Logic Programming* 29.1-3 (1996), pp. 17–64. DOI: 10.1016/S0743-1066(96)00068-4.

[49] Yüce Tekol and contributors. *PySwip v0.2.10*. 2020. URL: https://github.com/yuce/pyswip (visited on 09/26/2022).

[50] The Graphviz Authors. *Graphviz*. URL: https://graphviz.org/ (visited on 09/26/2022).

[51] The Julia Programming Language. *IJulia*. URL: https://github.com/JuliaLang/IJulia.jl (visited on 09/26/2022).

[52] The Jupyter Development Team. *Jupyter console 6.0*. URL: https://jupyter-console.readthedocs.io/en/latest/ (visited on 09/26/2022).

[53] The Jupyter Development Team. *The Qt Console for Jupyter*. URL: https://qtconsole.readthedocs.io/en/stable/ (visited on 09/26/2022).

[54] *The Python Package Index*. URL: https://pypi.org/ (visited on 09/26/2022).

[55] The Voilà Development Team. *Voilà Documentation*. URL: https://voila.readthedocs.io/en/stable/ (visited on 09/26/2022).

[56] The ZeroMQ authors. *ZeroMQ*. URL: http://zeromq.org/ (visited on 09/26/2022).

[57] Jan Wielemaker. *SWISH*. URL: https://swish.swi-prolog.org/ (visited on 09/26/2022).

[58] Jan Wielemaker et al. "SWI-Prolog". In: *Theory and Practice of Logic Programming* 12.1-2 (2012), pp. 67–96.

[59] Jan Wielemaker et al. "Using SWISH to Realize Interactive Web-based Tutorials for Logic-based Languages". In: *Theory and Practice of Logic Programming* 19.2 (2019), pp. 229–261. DOI: 10.1017/S1471068418000522.