On Bridging Prolog and Python to Enhance an Inductive Logic Programming System

Vítor Santos Costa[0000−0002−3344−8237] and Miguel Areias[0000−0003−1589−3174]

CRACS/INESC TEC
Dept. of Computer Science, Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
{vscosta,miguel.areias}@fc.up.pt

Abstract. Prolog is a programming language that provides a high-level approach to software development. Python is a versatile programming language that has a vast range of libraries including support for data analysis and machine learning tasks. We present a Prolog-Python interface that aims at exploiting Prolog deduction capabilities and Python's extensive libraries. Our novel interface was built using a divide and conquer methodology. In a first step, we implemented a set of `C++` classes that can be matched to Python classes; next, we used an interface generator to export the relevant classes. Finally, we use `C` code to actually convert between the two realms. In order to demonstrate the usefulness of the interface, we enhance an Inductive Logic Programming System with a visualization capabilities and show how to interface with a standard classifier.

Keywords: Prolog · Python · Inductive Logic Programming · Interoperability

1 Introduction

Prolog is a programming language that provides a high-level approach to programming through the use of a subset of First Order Logic [17]. Prolog relies on a very efficient querying mechanism, based on goal refutation for Horn clauses. Prolog systems complement this foundation with support for state capture and manipulation, and with mechanism for interaction. As Horn clause programs consist of a set of predicate definitions, this extra functionality is made available through "built-in" predicates; that is, through predicates that are defined as part of the Prolog implementation. Collections of these built-ins correspond to system libraries in traditional languages.

The design and implementation of these primitives is a large part of developing a Prolog system [11]. In fact, it is quite hard to support the different needs of the very diverse Prolog applications. One answer is to allow the user to build by herself some of these built-ins. To do so, the user will need to access the internal structure of the Prolog engine. The bidirectional protocol that defines how to access Prolog data-structures, on the one hand, and how logic programs may be

allowed to manipulate external data, on the other hand, is called the *Foreign Language Interface* (FLI).

Most often, the FLI is designed to interface with programs that were written in the Prolog engine's language. Arguably, most widely used systems are based on *C* or *Java*, and so are the FLIs. Note that a single Prolog system may have several FLIs, either to support different languages or for compatibility [31].

We introduce an interface that connects Python with a Prolog system. Python is a very popular language, and at the moment dominates in areas such as machine learning. It includes a large collection of tools that can be well used in Logic Programming Systems. Python programs are organized as modules, that group related classes, which may contain variables and functions. Arguably, the natural unit of sharing are instances of classes, and goals, the instances of predicates. Our goal is to provide a mapping such that Prolog goals can be Python objects, and Python goals can be Prolog calls.

Next, we describe our approach. Python is object oriented making it cumbersome to use a *C* based interface. Thus, the first step was to build a *C++* interface for the Prolog system. We then used the SWIG interface generator to export the classes to Python. The classes in the *C++* interface are thus translated to classes in a Python module. SWIG is very good at re-targeting the interface, but is not very good at writing all the nitty-grid of the translation. We found out it was sensible to use a separate library to do the actual translation.

The paper is organized as follows. First, we briefly review prior work on Prolog FLIs: there has been extensive work on these interfaces, but often this is not documented. Next, we describe the three components in our design: the *C++* interface, the SWIG translator, and the support libraries. We then give an example of how the interface can be applied to improve an existing application. Last, we conclude and discuss further work.

2 Prolog and the world

Most Prolog systems provide a foreign language interface (FLI), both to extend the language with user-defined built-ins, and to allow Prolog to be embedded as a component in a larger system. As most Prolog systems were traditionally written in `C`, FLIs supported `C`/Prolog. Quintus Prolog was one of the first systems to include a full fledged FLI allowing passing integers, floats, terms and pointers to the external code [21]. SICStus Prolog follows similar principles, but has a richer set of base types and more support for handling terms. Interface predicates are declared from types and modes [7]. A similar approach is implemented by GNU-Prolog [12].

Most other Prolog systems FLIs are based on reading and writing terms, built with *C* code. B-Prolog provides access to the engine internals [34]; YAP provides a wrapper but essentially exports the functional approach used in inner routines [10]; SWI-Prolog provides a handle-based abstraction of unification [33]; Ciao also implements term construction and access routines [15]; `ECL`$^{\mathtt{i}}$`PS`$^{\mathtt{e}}$ pro-

vides a more object-oriented flavor [19]; XSB provides two interfaces: one for direct access and the other for high-level access [24].

The advent of Java has generated interest in interfacing to other programming languages. There are two approaches: the client-server approach allows for distributed execution and is often more robust and cleaner. Examples include SICStus PrologBeans [1], InterProlog [6], and Prolog to R real [3]. Monolithic, or DLL based examples include Quintus Visual Prolog interface and the SWI-Prolog JPL [22]. The latter is a merge of a Prolog to Java and Prolog to Java interface and provides a very nice and complete API.

Python has also generated interest in the logic programming community. Bedevere was a SWIG [5] interface for GNU-Prolog, geared at Python [20]; similarly, pwig was developed for SWI-Prolog. Both rely on the C-interface [14]. py-xsb [4] exports the XSB FLI to a Python environment using ctypes. A more high-level approach is provided by PySWIP, that also uses Python ctypes to provides a module Prolog with most common operations [29].

The approaches presented so far rely on the system's FLI. Most FLIs provide only access to a system's functionality; these limitations are noticeable when trying to generate code that will run across systems [31]. A tighter integration with the external environment was proposed in Jinni [27] and Fluents [26]. External code embedding is also possible [32]. Finally, some authors would argue that object oriented logic languages provide a more natural integration with Object Oriented languages [8].

Janus is a Prolog-Python interface originally developed by Swift for XSB Prolog [25]. Janus was since adopted by SWI-Prolog and is the target of a joint effort between the XSB, SWI, and ciao communities [2]. ON the Prolog side, Janus provides a collection of built-in predicates, the `py_` family of predicates. These built-ins can execute arbitrary Python code and translate the results back to Prolog. Important examples are `py_call` that can call a Python method, and `py_iter` can be used to iterate over an object. In the specific case of SWI-Prolog one can also the quasi-quotation mechanism to inject Python code in the Prolog environment.

The Python to Prolog interface consists of five key classes: `query`, `apply`, `Term`, `Undefined`, and `PrologError`. The `query` and the `apply` classes serve the same goal, calling Prolog, but `query` is text based whereas `apply` is object based.

3 The C++ FLI

The *C++* FLI task is to wrap *C* data structures as classes. Next, we detail the main components that are concerned with term construction and manipulation, database management, and execution. Figure 1 describes graphically the class hierarchy of the interface.

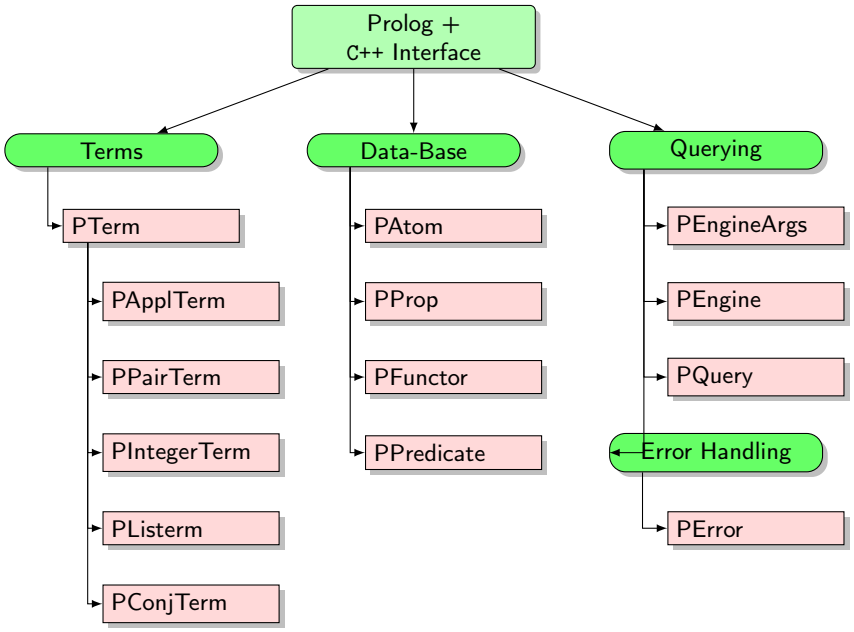


Fig. 1. Class structure of the C++ interface

Terms The `PTerm` or `T` class exports handles to Prolog terms (type `Term` or `PTerm`). The sub-classes are `PApplterm`, `PPairterm`, and sub-classes for the usual types. `PTerm` is a collection of methods plus a handler pointing to the Prolog term.

```
protected:
  yhandle_t hdl; /// handle to term, equivalent to term_t
```

The private methods `mk(Term t)` generates the handle and copies it to `this->hdl`; the getter `Term gt()` fetches the term. As an example, term type checking is implemented through virtual methods such as `isVar`, that is defined by `PTerm` as

```
{ virtual bool isVar() { return IsVarTerm(gt()); }
```

that is redefined by `PIntegerTerm` as:

```
 bool isVar() { return false; }
```

The `gt()` method gets the Prolog term from the handler.

`PTerm` also provides interfaces for most term operations, e.g., to verify whether our object is a variant of a term t1, we simply write:

```
virtual P_Term variant(PTerm t1) { return P_Variant(gt(), t1.term()); }
```

Notice that as `gt()` is a private method, we use `term()` to construct the the external object.

The sub-classes do not have to be disjoint. The interface provides both `PPairTerm` that refers to a pair of terms, and `PListTerm` that is used to access a true list. The latter class allows for constructors such as:

```
PListTerm(std::vector<Term>).
```

The constructor uses an array with objects of `Term`. Whenever possible, one should avoid creating intermediate `C++` objects. It is much more efficient to work with the engine `C` objects, and create a single `C++` object as the final result.

The Prolog Database Our approach assumes that the Prolog database is organised as a symbol table, where symbols are Prolog atoms. Atoms have a variable set of properties. One important property is `PFunctor`, that is special in the sense that functors themselves can have properties of type `PPredicate`. The classes `Patom`, `PProp`, `PFunctor`, and `PPredicate` wrap this functionality.

Run-Time The run-time consists of three classes: `PEngine`, `PEngineArgs`, and `PQuery`. To boot the Prolog system, one must first fill the execution parameters at `PEngineArgs` and create an `PEngine` object, whose main task is to create `PQuery` object. The latter's main task is to create an iterator for query execution, and to provide access to the query state. There are several ways to start a query, the example below is used by the interpreter to run a query from user input:

```
 PQuery(const char *s) :-
PPredicate(s, goal, names, (nts = a1_ptr()))
```

In this example, `PPredicate` is given the query string `s`, parses it, stores the goal's argument in the register array, uses call by reference to pass the map with the variable names, and finally returns itself. `PQuery` just needs to start the engine.

Error Handling The `PError` class takes care of error objects. Errors are generated by all the Prolog components. In our approach, we assume that the Prolog system ensures a fair treatment by storing the error as a dictionary, and storing all the entries as constants or as text strings. Terms are presented as text.

4 The SWIG Translator

SWIG is a tool that promises that, if given a specification for a *C* or *C++* library it will generate the corresponding stub in a target language, such as *Python*, or *Java*. In practice the specification details the data structures that are made visible in the library's header file. A simple example is:

```
%include "pt.hh"
```

This single line of code results in generating a set of Python classes, one per *C++* class. The classes are really just stubs: they wrap the arguments and pass them to the source library.

Unfortunately, the wrappers just insulate the objects, and often we need too access fields or call methods. One solution is to extend SWIG with libraries for common data-structures:

```
%include stdint.i
%include std_string.i
%include std_vector.i
%include "pt.hh"
```

We now can use the methods for classes `int_t`, `std::string`, and `std::vector<T>` from within Python. Although SWIG supports *namespaces* and templates, but not full `C++`.

We can construct and manipulate PTerms, but objects of type `Term` are opaque. This is a problem because Prolog programs only manipulate terms; as such, all calls from Prolog to Python will have arguments of type Term. SWIG addresses this problem through *typemaps*, such as the one in the followup code:

```
% typemap(out) Term {
  return $result =
      prolog_to_python($1, false, 0, true);
}
%
```

Unfortunately, writing SWIG extensions eventually becomes very hard. The interface relies instead on auxiliary functions, *p2py* () (`prolog_to_python`) and *py2p* () (`python_to_prolog`) interactions. Table 1 describes the main rules: notice that these rules are applied recursively over terms or Python expressions.

4.1 From Prolog to Python

The function *p2py*() is the key to having transparent execution. It maps a Prolog term to a Python object as follows:

- Integers and other numbers map to integers (integer objects). Prolog atoms are used for two purposes: as symbols (e.g., `fail` is likely to refer to a built-in), and as text, e.g., `'Where twinkling in the dewy light,'` is probably text. Unfortunately, these roles are in no way guaranteed: we can use the atom `fail` to search for students who `fail`ed a course, and we could use the text as the name of a predicate. This confusion stems from the origins of Prolog and the Edinburgh syntax.
  There is no perfect solution to this problem. Our approach is:
  1. if the atom text can be a legal Python symbol, map it as symbol: return *pyLookup*(*str*(*t*));

Table 1. Translation rules from Prolog to Python and from Python to Prolog; *vnames* is the set of valid names for Python symbols

Prolog → *p2py()* →	**Python** → *py2p()* →	**Prolog**
int	int	int
float	float	float
true	True	true
false	False	false
none	None	none
atom $\in$ *vnames*	symbol	atom
atom $\notin$ *vnames*	string	–
string	string	string
$[A, B, \ldots, C]$	Python List	$[A, B, \ldots, C]$
$t(\ldots)$	Python Tuple	$t(\ldots)$
$\{k_1 : v_1, \ldots, k_n : v_n\}$	Python Dictionary	$\{k_1 : v_1, \ldots, k_n : v_n\}$
$F(a_1, ..., k_n = a_n)$ (compound term)	$\phi = F(a_1, ..., k_n = a_n)$ or Python Named Tuple	py2p(ϕ) $F(a_1, ..., k_n = a_n)$
A.B.F(...)	Function call *obj*	*ptr*(&*obj*)

2. or alternatively, map it as a Python string (PyUnicode object): return $PyUnicode\_Str(str(t))$.

One alternative is to map atoms to symbols only if they are symbols in Python; the problem is that the atom may be translated in different ways as the program runs (in the worst case the translation will depend on the Python garbage collector).

– True lists are translated into Python lists (PyList objects). Prolog lists are linked lists, so the actual match for a Prolog list `[1,2,3]` should be an `X` such that:

```
t0 = [3,[]]
t1 = [2,t0]
X  = [1,t1]
```

In practice, users expect list to match list, and that is what we support when atom `[]` matches the empty list: while list(t) .

– Terms of the form `t(...)` are translated to Python tuples (PyTuple). Python tuples are similar to Python lists, but whereas Python lists can expand, contract and be updated, tuples have fixed size and fixed arguments.

– Dictionaries: we assume that the Prolog system does not support dictionaries at the engine level, instead, it will translate a Python dictionary to a term of the form:

```
{ a1:t1, ... , am:tm, ... . an:tn }.
```

This representation allow us to send dictionaries to Python and back.

– we still have to translate partial lists and compound terms. By default, Python does not construct compound terms; it evaluates functions (or methods). The interface thus tries T with functor named f, and arity a. It builds and execute the function call as follows:

   1. set $i \leftarrow a, args \leftarrow None, dict \leftarrow$
   2. search for an object of name f; proceed if the object is callable, otherwise certify it matches a named tuple: if it does, return the tuple otherwise an error;
   3. while $T[i] = (k = v)$, where k is an atom or a string $dict[k] = p2py(v)$ do $i \leftarrow i - 1$;
   4. create a tuple for the remaining arguments that must be represented: $args = (p2py(T[1]), \ldots, p2py(T[i]))$.
   5. execute the code and return $code(f)(args, dict)$
- Python uses $A.B.C$ to represent module/class/method hierarchy. We use the same syntax, taking advantage of the fact that early Prolog systems used the dot operator to represent a pair, so $A.B.C = [A|[B, C]$. This makes the text close to Python. Drawbacks include overloading even more the dot character, and diverging from other packages.

Our approach goes a little bit further and defines ./2 as an existing predicate, so that we can write $A.B.C$ as a goal:

4.2 From Python to Prolog

The reverse function *py2p* `python_to_prolog` follows the same guidelines as *p2py*:

- in Python there is no ambiguity about whether an object should be treated as a string or not. It is natural to map strings to strings, but it would be nice to translate to atoms, or list of codes or characters.
- If an object does not fit in the above mentioned rules, we assume that the Prolog system will pass the address of the object.

4.3 Assignment

The library supports two different forms of assignment:

- If the target is a variable, that is $V \leftarrow Exp$, we should bind the variable to the outcome of the *p2py* call.
- If the target is an atom and the atom is an attribute or a key already existing in the symbol table, $A \leftarrow Exp$ should set the attribute to $p2py(T)$.
- If the target is a new atom create it in a system table, and then proceed as before.
- if the target is indexed, $A[I] \leftarrow Exp$, call *py2p* to obtain either a variable or an address, and then proceed as before.

4.4 An Example

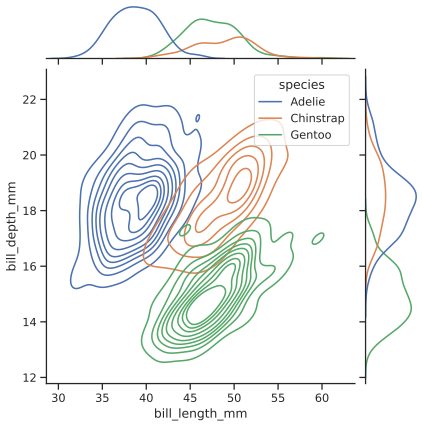
Next, we show a self-contained example of using the interface. The example was adapted from the the seaborn package [30], based on matplotlib [16].

```
:- use_module(library(python)).
:- python_import(seaborn as sns).
:- python_import(matplotlib.pyplot as plt).

main :-
     penguins := sns.load_dataset( "penguins" ),
     sns.histplot(penguins, x="flipper_length_mm"),
     plt.show().
```

After loading the Prolog library, we import seaborn and bind the package to two symbols, `seaborn` and `sns`. The actual program starts by looking up `sns` in the system table, and then creates a function call with the function `load_dataset()` obtained from the module `seaborn`, a tuple with a string "penguins", and an empty dictionary. The result of the function is assigned to a new symbol, `penguins`. To call the procedure we lookup `sns` and build a function call by looking up `histplot` in `seaborn`, constructing a tuple with a single entry `penguins`, and a dictionary with a single entry with key "x" and value "flipper_length_mm". This call generates the plot. Finally, `matplotlib` is called to show the result plot in a physical device.

A more complex example is shown next, with corresponding output presented in Figure 2.




```
:- use_module(library(python)).
:- python_import(seaborn as sns).
:- python_import(matplotlib.pyplot
                 as plt).

main :-
  sns.set_theme(
  style="ticks"),
  penguins :=
    sns.load_dataset(
        "penguins"),
  sns.jointplot(
    data=penguins,
    x="bill_length_mm",
    y="bill_depth_mm",
    hue="species",
    kind="kde"
  ),
  plt.show().
```

Fig. 2. Visualization of the Penguin dataset - showing the difference between three penguin species

4.5 Interface Libraries

The previous examples focused on the Prolog side. SWIG provides an extensive set of classes for the Python side, but as the classes are based on `C++`, it cannot take full advantage of the language. Thus, the Prolog system must include libraries to facilitate programming. We briefly discuss two of the most interesting techniques.

Iterators

```
class Query (PQuery):
    def __init__(self, engine, g):
        ...
    def __iter__(self):
        return self
   def done(self):
        gate = self.gate
        completed = gate == ...
        return completed

    def __next__(self):
        if self.done() or
           not self.next():
            raise StopIteration()
        return self
```

Iterators are classes that implement a sequence generation protocol. In this example, the class `Query` is a refinement of the `C++` class that can be used it to enumerate solutions in a `for` or `while` loop.

The two key methods are `done` and `__next__`. They rely on PQuery to provide the last port or gate crossed. A call to `self.done()` checks whether a query us still active. `self.__next__()` either gets the next answer, or sends a signal to stop iterating.

Named Tuples Named tuples are syntactically similar to Prolog terms, and we use them to give a Prolog flavor to Python code. By adding an iterator to a named tuple, we can have a Prolog goal:

```
class LoadLibrary(Predicate):
    def __init__(self, eng):
        self.engine = eng
        self.goal = namedtuple('load_library', 'name' )

    def run(self, c):
        self.engine.run(self.goal(library(c)))

    def __str__(self):
```

```
        return self.goal.__str__()

load_library = LoadLibrary(PEngine).run
```

This code allows to call `load_library("lists")`.

5 Pythonic Aleph

To conclude we show how the interface can be used to improve an existing application. Aleph [23] is an Inductive Logic Programming learning system, based on Progol [18]. Aleph implements relational machine-learning algorithms; the reference algorithm, `induce` generates a theory by following these steps:

- Choose an example, and collect literals that are connected to the example. Swap different constants by different variables and call the result *bottom clause*;
- select subsets from the example and pick the clause that best separate positives from negatives.

The `induce` algorithm implements greedy coverage removal, that is, examples covered by the chosen clause will be discarded from the step.

The bottom-clause dominates the search space. Its construction depends on the user-provided predicates plus mode declarations that structure the clause. Mode declarations are related, but quite different, from the usual mode declarations in logic programming [11]. As an example, consider the following two mode declarations for a chemical structure-activity dataset [13,9]:

```
:- modeb(*,atm(+drug,-atomid,#element,#integer,-charge)).
:- modeb(*,symbond(+drug,+atomid,-atomid,#integer)).
```

The first argument can be largely ignored: it just says we should look for all the solutions. The main functors of the axioms argument, `atm` and `symbond` declare that we were going to use atoms and bounding in a molecule. The symbols `drug`, `atomid`, `element`, `integer` and `charge` name a set of disjoint concepts, the types, that will be associated with clause variables. Finally, the mode declarations are as follows, assuming we want to place a variable V at an argument A_i whose type is $\mathcal{T}$:

- `+`: V must have been used in a previous call C'. Moreover, if $A'_j = V$ then $type(A'_j) == \mathcal{T}$;
- `-`: V may be a new variable of type $\mathcal{T}$;
- `#`: A_i must be set to a constant of type $\mathcal{T}$.

The following example tries to clarify the application of modes:

```
active(Drug) :- atm(Drug, Atom, c, 4, Charge)).
active(Drug) :- symbond(Drug, Atom1, Atom2, 1)).
active(Drug) :- atm(Drug, Atom, c, 4, Charge)),
                symbond(Drug, Atom, Atom0, 1).
```

The first example receives the input variable *Drug* from the head, hence it obeys the modes. The second clause is illegal, because the second argument is input, and it is the first occurrence of *Atom*. The third clause calls *atm* and then *symbond*, making it legal.

The first step in the `induce` algorithm is to create a maximal conjunction of goals that **(i)** include the example and **(ii)** goals satisfy the input and constant declarations. The saturated clause can be seen as clause but also as a graph (or hypergraph) where the edges are nodes and the edges are mode-induced dependencies: generating rules is enumerating sub-graphs.

Aleph can display the bottom-clause as a text clause, but as the bottom clause can easily reach hundreds or thousands of nodes, it is difficult to extract any useful insights. An alternative is to use graph visualization. Next we show results from D3blocks [28], an interface between Python and the D3.js library.

```
firstgraph2d3(Edges,Nodes,_Groups,Colors,Weights,Names,Preds) :-
    maplist(split_edge, Edges, Sources, Targets),
    maplist(edge_weight, Edges, EWeights),
    d3 := d3blocks.'D3Blocks'(),
    df := pd.'DataFrame'.from_dict({"source":Sources,
      "target":Targets,
      "weight":EWeights}),
     d3.elasticgraph(df, filepath="./SatClause.html",
                     figsize=[3000,2000]),
     maplist(set_node(d3), Nodes, Preds, Colors, Names, Weights),
     d3.'Elasticgraph'.'D3graph'.show().

set_node(Whom, Node, Pred, Color, Name, Size) :-
    n := Whom.'Elasticgraph'.'D3graph'.node_properties[Node],
   n["tooltip"] := Pred,
   n["color"] := Color,
   n["label"] := Name,
   n["size"] := Size.
```

First, the edges are converted into a Pandas data-frame. This data-frame is the main structure for the search object `d3`. This object also stores nodes as the dictionary `node_properties`. Dictionaries are not guaranteed to always maintain key order. The Prolog code uses `map_list` to iterates over the nodes. The `set_node` predicate fetches this dictionary by using unification to combine the D3 object, the path to the properties dictionary, and the code key. Finally, we set the properties.

We use color to distinguish five type of predicates:

1. the seed is the concept we want to learn (dark green);
2. the attributes are properties of the compound (light blue).
3. entities are the atoms in the compound (red);
4. relations provide nearest of the structure of the graph, (light green)

5. constraints are relations between attributes, such as arithmetic comparisons between numeric attributes, like $A = 8$ or $B \leq 7.3$. (yellow).

The picture is centered in the example. There is a ring with the atom ls and an external ring with the boundings. Looking closely one can notice clusters of atoms, quite often hydrogens bonded to a larger atom. One can also observe that the boundings are always duplicates: for all $A_1 \rightarrow A_2$ there is a $A_2 \rightarrow A_1$. The bonds form a nice half-ring with a cluster of carbons on top. The attributes cluster to the left; they are totally independent from the rest of the network. There are many other opportunities for visualization in this application, namely within the search process. Figure 3 shows a snapshot of plot of a drug discovery application's bottom clause using the elastic graph algorithm.

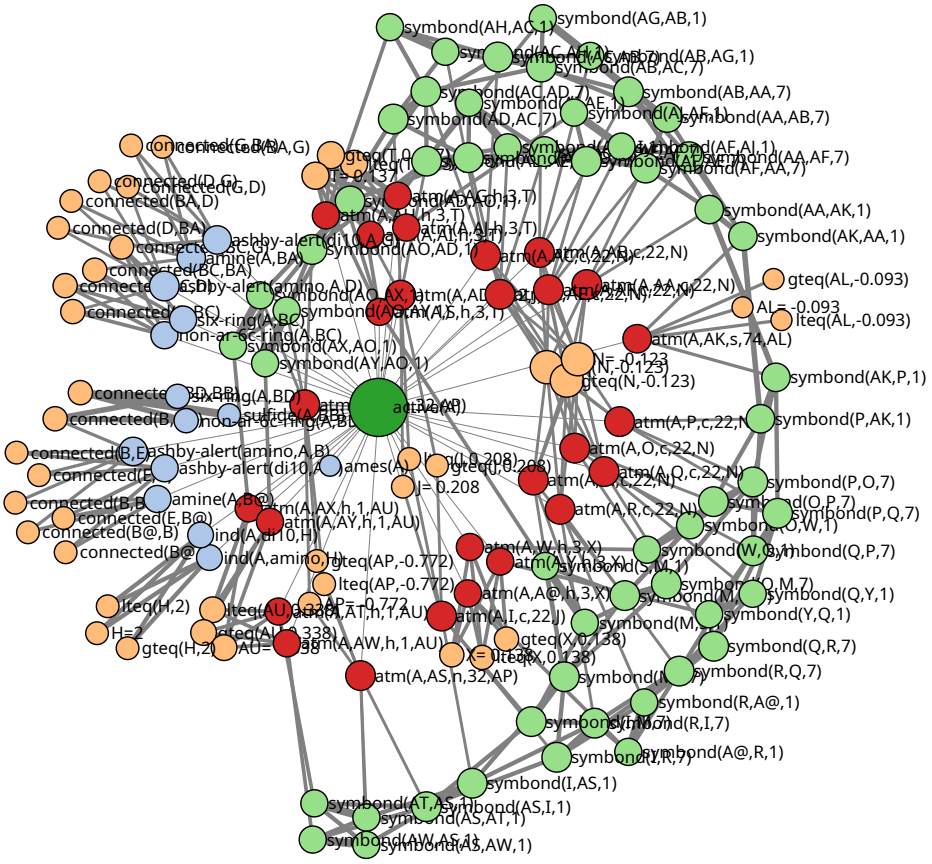


Fig. 3. A Bottom-Clause according to D3.js

Arguably, the relations described in the snapshot are too complex to be analyzed in a static visualization. However, the reader should keep in mind that D3.js is interactive, which for for molecular structure visualization offers significant advantages. D3.js enables precise control over the rendering of molecular diagrams, allowing for customizable layouts that accurately depict atom positions and bond types, including visual distinctions like bond thickness or color coding. Its interactivity supports also dynamic exploration, such as zooming, panning, and tool-tips for displaying the atomic properties, enhancing the user's engagement and understanding.

5.1 Other Applications

Often one tries to improve the leaner performance by seeing each clause in the theory as an attribute, so that the set of clauses become the attributes of a classifier. The next procedure learns an SVM classifier from a set of clauses plus a set of examples, and evaluates its performance on the training data:

```
learn :-
   findall(t(Train,Label),
        (example(_,Pol,Ex),
         ( Pol=pos -> Label = 1 ; Label = 0),
         findall(V,(clause(Ex,B),
                    (once(B)->V=1;V=0)),
                 Train),
        ), TrainingData),
        maplist(zip,TrainingData,Data,Labels),
        clf := sklearn.svm.'SVC'(kernel= "linear".
                                 class_weight="balanced").
        clf.fit(Data,  Labels),
         Scores := clf.predict_proba(Data).transpose()[1].tolist(),
        scipy.metrics.'RocCurveDisplay'.from_predictions(Labels,
                                                         Scores).
```

The algorithm constructs a list of labels, and a list of lists that represent the data (or examples) as bitmaps. The classifier is initialized, and trained according to the data and labels; the data is converted by the interface from Prolog list of lists to Python list with lists, and then by the NumPy library to an array. The reverse process is more complex because YAP does not convert NumPy matrices; we first transpose to extract the second column, next convert the NumPy vector to a list, and then pass the result to provide a first evaluation on the data set.

6 Conclusions

As the amount of reusable existing libraries keeps on growing, programming becomes more about connecting. This interface tries to take advantage of this trend, by making it as natural as possible to use both languages together.

The main challenge was the complexity of both environments. We used `C++` to obtain object orientation, and `SWIG` to automatically cover the libraries. The "piano lifting" is still in `C`. Reference handling and documentation need work. Altogether, we hope that this work will be a step to exploit the overall advantages of Prolog systems, making Prolog more helpful, but also more fun.

As future work, we plan to compare the performance of our approach against Janus, the Prolog-Python interface originally developed by Swift for XSB Prolog, adopted by SWI-Prolog and the target of a joint effort between the XSB, SWI, and ciao communities.

Acknowledgments. This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within projects UIDB/04434/2020, UIDP/04434/2020 and UIDB/50014/2020. DOI 10.54499/UIDB/50014/2020.

References

1. et al., M.C.: SICStus Prolog Users Manual (December 2022), `https://sicstus.sics.se/sicstus/docs/latest4/html/prologbeans/`
2. Andersen, C., Swift, T.: The janus system: A bridge to new prolog applications. In: Warren, D.S., Dahl, V., Eiter, T., Hermenegildo, M.V., Kowalski, R.A., Rossi, F. (eds.) Prolog: The Next 50 Years, Lecture Notes in Computer Science, vol. 13900, pp. 93–104. Springer (2023). `https://doi.org/10.1007/978-3-031-35254-6_8`, `https://doi.org/10.1007/978-3-031-35254-6_8`
3. Angelopoulos, N., Costa, V.S., Azevedo, J., Wielemaker, J., Camacho, R., Wessels, L.F.A.: Integrative functional statistics in logic programming. In: Sagonas, K. (ed.) Practical Aspects of Declarative Languages - 15th International Symposium, PADL 2013, Rome, Italy, January 21-22, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7752, pp. 190–205. Springer (2013), `https://doi.org/10.1007/978-3-642-45284-0_13`
4. Bartsch, G.: py-xsb (12 2004), collected at `https://github.com/gooofy/py-xsb`
5. Beazley, D.M.: SWIG: an easy to use tool for integrating scripting languages with C and C++. In: Diekhans, M., Roseman, M. (eds.) Fourth Annual USENIX Tcl/Tk Workshop 1996, Monterey, California, USA, July 10-13, 1996. USENIX Association (1996)
6. Calejo, M.: Interprolog: Towards a declarative embedding of logic programming in java. In: Alferes, J.J., Leite, J.A. (eds.) Logics in Artificial Intelligence, 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3229, pp. 714–717. Springer (2004), `https://doi.org/10.1007/978-3-540-30227-8_64`
7. Carlsson, M., Mildner, P.: Sicstus prolog - the first 25 years. Theory Pract. Log. Program. **12**(1-2), 35–66 (2012), `https://doi.org/10.1017/S1471068411000482`
8. Castro, S., Mens, K., Moura, P.: JPC: A library for categorising and applying inter-language conversions between java and prolog. Sci. Comput. Program. **134**, 75–99 (2017), `https://doi.org/10.1016/j.scico.2015.11.008`
9. Chen, J., Muggleton, S.H., Santos, J.C.A.: Learning probabilistic logic models from probabilistic examples. Mach. Learn. **73**(1), 55–85 (2008), `https://doi.org/10.1007/s10994-008-5076-4`

10. Costa, V.S., Rocha, R., Damas, L.: The YAP prolog system. Theory Pract. Log. Program. **12**(1-2), 5–34 (2012), https://doi.org/10.1017/S1471068411000512
11. Deransart, P., Ed-Dbali, A., Cervoni, L.: Prolog - the standard: reference manual. Springer (1996)
12. Diaz, D., Abreu, S., Codognet, P.: On the implementation of GNU prolog. Theory Pract. Log. Program. **12**(1-2), 253–282 (2012), https://doi.org/10.1017/S1471068411000470
13. Finn, P., Muggleton, S., Page, D., Srinivasan, A.: Pharmacophore discovery using the inductive logic programming system progol. Machine Learning **30**, 241–270 (1998)
14. no García, S.F.: pwig wrapper and interface generator (12 2004), collected at https://pwig.sourceforge.net/
15. Hermenegildo, M.V., Bueno, F., Carro, M., López-García, P., Mera, E., Morales, J.F., Puebla, G.: An overview of ciao and its design philosophy. Theory Pract. Log. Program. **12**(1-2), 219–252 (2012), https://doi.org/10.1017/S1471068411000457
16. Hunter, J.D.: Matplotlib: A 2d graphics environment. Computing in Science Engineering **9**(3), 90–95 (2007)
17. Körner, P., Leuschel, M., Barbosa, J., Costa, V.S., Dahl, V., Hermenegildo, M.V., Morales, J.F., Wielemaker, J., Diaz, D., Abreu, S.: Fifty years of prolog and beyond. Theory Pract. Log. Program. **22**(6), 776–858 (2022), https://doi.org/10.1017/S1471068422000102
18. Muggleton, S.H.: Inverting entailment and progol. In: Furukawa, K., Michie, D., Muggleton, S.H. (eds.) Machine Intelligence 14, Proceedings of the Fourteenth Machine Intelligence Workshop, held at Hitachi Advanced Research Laboratories, Tokyo, Japan, November 1993. pp. 135–190. Oxford University Press (1993)
19. Schimpf, J., Shen, K.: Ecl$^{\text{i}}$ps$^{\text{e}}$ - from LP to CLP. Theory Pract. Log. Program. **12**(1-2), 127–156 (2012), https://doi.org/10.1017/S1471068411000469
20. Seward, A.J.: bedevere (04 2002), collected at https://bedevere.sourceforge.net/
21. SICS Swedish ICT AB: Quintus Prolog Manual (2015), collected at https://quintus.sics.se/isl/quintuswww/site/index.html
22. Singleton, P., Dushin, F.: JPL: A bidirectional Prolog/Java interface (2018), collected at https://jpl7.org/
23. Srinivasan, A.: The Aleph Manual (2001)
24. Swift, T., Warren, D.S.: XSB: extending prolog with tabled logic programming. Theory Pract. Log. Program. **12**(1-2), 157–187 (2012), https://doi.org/10.1017/S1471068411000500
25. Swift, T., Andersen, C.: The janus system: Multi-paradigm programming in prolog and python. In: Pontelli, E., Costantini, S., Dodaro, C., Gaggl, S.A., Calegari, R., d'Avila Garcez, A.S., Fabiano, F., Mileo, A., Russo, A., Toni, F. (eds.) Proceedings 39th International Conference on Logic Programming, ICLP 2023, Imperial College London, UK, 9th July 2023 - 15th July 2023. EPTCS, vol. 385, pp. 241–255 (2023). https://doi.org/10.4204/EPTCS.385.24, https://doi.org/10.4204/EPTCS.385.24
26. Tarau, P.: Fluents: A refactoring of prolog for uniform reflection an interoperation with external objects. In: Lloyd, J.W., Dahl, V., Furbach, U., Kerber, M., Lau, K., Palamidessi, C., Pereira, L.M., Sagiv, Y., Stuckey, P.J. (eds.) Computational Logic - CL 2000, First International Conference, London, UK, 24-28 July, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1861, pp. 1225–1239. Springer (2000), https://doi.org/10.1007/3-540-44957-4_82

27. Tarau, P.: Agent oriented logic programming in jinni 2004. In: Haddad, H., Liebrock, L.M., Omicini, A., Wainwright, R.L. (eds.) Proceedings of the 2005 ACM Symposium on Applied Computing (SAC), Santa Fe, New Mexico, USA, March 13-17, 2005. pp. 1427–1428. ACM (2005), https://doi.org/10.1145/1066677.1067000
28. Taskesen, E.: D3blocks: The python library to create interactive and standalone d3js charts (Sep 2022), collected at https://towardsdatascience.com/d3blocks-the-python-library-to-create-interactive-and-standalone-d3js-charts-3dda98ce97d4/
29. Tekol, Y.: Pyswip (2023), collected at https://github.com/yuce/pyswip
30. Waskom, M.L.: Seaborn: Statistical data visualization. Journal of Open Source Software **6**(60), 3021 (2021)
31. Wielemaker, J., Costa, V.S.: On the portability of prolog applications. In: Rocha, R., Launchbury, J. (eds.) Practical Aspects of Declarative Languages - 13th International Symposium, PADL 2011, Austin, TX, USA, January 24-25, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6539, pp. 69–83. Springer (2011), https://doi.org/10.1007/978-3-642-18378-2_8
32. Wielemaker, J., Hendricks, M.: Why it's nice to be quoted: Quasiquoting for prolog. CoRR **abs/1308.3941** (2013), http://arxiv.org/abs/1308.3941
33. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: Swi-prolog. Theory Pract. Log. Program. **12**(1-2), 67–96 (2012), https://doi.org/10.1017/S1471068411000494
34. Zhou, N.: The language features and architecture of b-prolog. Theory Pract. Log. Program. **12**(1-2), 189–218 (2012), https://doi.org/10.1017/S1471068411000445