PyPlC – Towards a Prolog Database Connectivity for Python

Stefan Bodenlos[1], Daniel Weidner[1], and Dietmar Seipel[2]

University of Würzburg, Department of Computer Science,
Am Hubland, D – 97074 Würzburg, Germany

[1]{stefan.bodenlos,daniel.weidner}@stud-mail.uni-wuerzburg.de
[2]dietmar.seipel@uni-wuerzburg.de

Abstract. Integrating languages of different programming paradigms introduces interesting questions and conflicts. In multi–paradigm programming, interfaces and data structures must be stated in a way that is reasonable for both worlds. In this respect, the open database connectivity ODBC is a success story, being today's standard tool of accessing relational databases with applications written in mostly any programming language.

In this work, we extend the concepts of ODBC to the integration of the logic programming language Prolog and present a unified approach for the intuitive integration of Prolog queries into Python, a popular imperative programming language. Our tool chain called PYPLC aims to provide a similar standard for accessing derived facts from the Prolog database without side effects. PYPLC contains the Python/Prolog query language PYPLQL. Together with a specification language based on XML Schema, this allows to specify and generate structures in Python corresponding to facts from Prolog.

Keywords: Prolog· Python· Multi–paradigm Programming · Integration.

1 Introduction

Integrating different technologies is a common issue in the process of software development. Often, the program's logic and data tiers are driven by different programming paradigms. Either two separate teams design and maintain both components, or a single team does: the first causes expenditure for coordinating the teams, the latter complicates the planning process instead of simplifying it.

Python is a universal, multi–paradigm programming language, which is well known for its simple but nevertheless powerful syntax [11]. Although it is multi–paradigm, object–orientation can be viewed as one of its characteristics. Python lacks a support for logic programming. Prolog provides powerful mechanisms and tools related to deductive databases, some applications are easier to implement with logic programming, e.g., natural language parsing [9], and in some areas, like artificial intelligence [3], many solutions have been developed. Because Python

is a widespread and easy–to–learn language, the integration of Prolog offers an access to a powerful technology for a large part of software developers.

One way to integrate Prolog in Python is to imitate a Prolog interface in Python. Lager and Wielemaker have proposed a library called Pengines [7], which is an infrastructure providing general mechanisms for converting Prolog data and handling Prolog non–determinism; it is included in the standard package for Swi–Prolog 7. Like in Odbc, a client can send a Prolog program to a thread, that provides the data exchange. After that, the client can send Prolog queries to the thread, which will respond with a set of answer tuples. Based on Pengines, the Python module PythonPengines has been proposed [1]. A user can construct Prolog programs and query terms in Python. Here, a Python user has to comprehend larger parts of the Prolog syntax and semantics. Moreover, the instructions are usually not compact, because they need some lines of code to declare a query step by step.

There is a broad field of applications for Prolog. Here, we focus on its strength for database management and provide a framework for this specific application. Many solutions have been proposed, that can establish a connection between Prolog and object–oriented programming languages like Python. Often they mainly aim to establish a connection at all, and the user must have a deep understanding of the language to integrate (in this case Prolog). Also Odbc requires some knowledge of Sql. Here, we want to go one step further and integrate Prolog into Python in greater depth. In this new approach, a user can understand and query a Prolog database using Python concepts only. Thus, knowledge about Prolog is no prerequisite, nevertheless Prolog can be employed.

The fundamental structure of an object–oriented programming language is an object. It contains data describing the object, called attributes, as well as operations that manipulate the data and serve as an interface. Each object belongs to exactly one class, that can be viewed as a description for objects. All objects of a class share the same properties, except the concrete values of the attributes. A class can be inherited by another class, meaning that the class extends the original description. In general, Python follows these principles in a way that ensures a high level of dynamicity. Especially, the set of attributes of an object can be dynamically extended even at runtime. Since the object–oriented and the relational paradigm differ, issues can occur when an object is to be saved in a relation. This is called object–relational (impedance) mismatch [5]. A solution for this is an object–relational mapping; that is, an object–oriented program can access a relational database in an object–oriented manner.

In the present paper we propose a framework, which is more intuitively usable in terms of the Python philosophy. We introduce a new approach of such database connectivities extending the Odbc approach, namely a tool chain called PyPlC (Python/Prolog Database Connectivity) [2]. It aims to convert Prolog structures into corresponding, object–oriented structures, which can then easily be processed in Python. A new intuitive object–oriented query language allows for a simple way of querying a Prolog database in a Python–like fashion. PyPlC translates such requests into Prolog syntax and maps the result back into

Python objects. Instead of realizing such procedures manually by a user, all of these processes are automatized and customizable by PyPlC . This simplifies Prolog database integration for Python. PyPlC is inspired by the architecture of the uniform and object–oriented integration framework CAPJa for Java and Prolog introduced in [8], but it is a new implementation from scratch with some new features. The Java tool supports most Prolog systems, and provides semi–automatized and completely customizable mechanisms for the integration and a fully automatized mapping from Prolog functors to Java objects; an object–oriented query language enables queries based on Java syntax and semantics.

The rest of this paper is structured as follows: Section 2 describes how Prolog facts should be mapped to corresponding Python structures. Section 3 presents the query language for posing queries to Prolog in Python. Our approach is discussed shortly in Section 4. Conclusions and future work are given in Section 5.

2 Corresponding Structures in Prolog and Python

As a core for the intuitive access to a Prolog database, PyPlC generates corresponding structures in Python, or more specifically, it maps Prolog functors into corresponding classes. Thus, PyPlC eases the work for programmers, because it automatizes this integration process, whereas customization is completely supported.

The University Database in Prolog. For illustration, we introduce a simple Prolog database representing a university in Listing 1.1. It contains a functor `student/6` for facts composed of constants and a compound term with the functor `address/4`. The object–oriented representation of this model is a class structure, in which every functor is represented by a class, whose attributes represent the arguments and whose name is derived by the functor's name (see Figure 1).

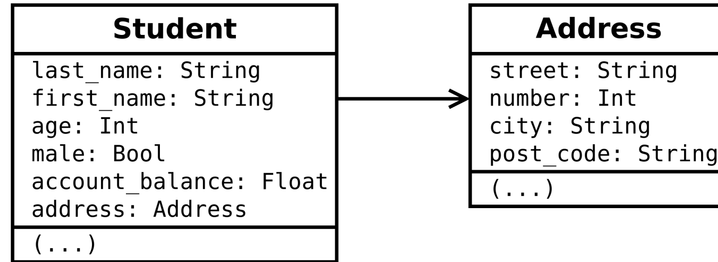
Listing 1.1. Part of a Prolog Database Representing a University.

```
% student( Last_Name, First_Name, Age, Sex,
%     Account_Balance,
%     address(Street, Number, City, Post_Code) ).

student( 'Doe', 'John', 20, male, 1000.00,
    address('Park Avenue', 42, 'Duckburg', 12345) )
```

The individual facts of the database are represented by objects of these classes. PyPlC 's mapping definition is bijective; therefore, mapping and remapping of facts and objects can be done any number of times without loss of information.

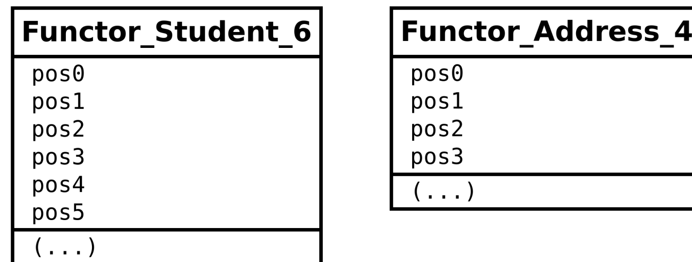
Fig. 1. Class Structure Representing the University Database.



In Python, attributes can be addressed by their names, and in Prolog, arguments can be addressed by their position in the compound term. Because of that, a mapper defines the association between both components. For each built–in type, a static mapping is implemented. A mapper transforms the components by using either such a mapping definition for built–in types or calls a mapper (possibly recursively). Thus, the type of the attribute *address* is the class *Address*.

Static and Dynamic Mapping. Our framework offers two types of mappings: static and dynamic mappings, where these names allude how they are intended to be used rather than how they operate. The dynamic mapping aims at providing a straightforward integration of Prolog into Python without any preparatory. It avoids major changes in PyPlC 's Python module and only stores necessary information needed for the mapping process. When using the dynamic mode, the mapping process is not customizable at all. Thus, every time a Prolog database is integrated into Python, the representation remains unchanged. This is helpful if the user is very familiar with the queried Prolog database.

Fig. 2. Dynamically Created Class Structure Representing the University Database.



PyPlC dynamically maps the university sample database of Listing 1.1 into the class structure given in Figure 2. Here, the name is extended by the arity of the predicate, because Prolog makes a distinction between predicates of different arities. In Python, the use of an attribute can easily be understood by its name, whereas in Prolog there are no such descriptions for components. Hence, PyPlC cannot extract coherent names for the attributes. Also the association between classes cannot be generated from a plain Prolog predicate, since the typing comes from a conversion rather than syntax. The created structure is not user–friendly, but it offers a simple access to a Prolog database in Python, of which the user knows all about.

The General Object–Oriented Mapping Notation. The static mapping resolves these issues, but it requires a description of the Prolog database. Here, we are planning to use the General Object–Oriented Mapping Notation (Goomn), an Xml–based notation. For each component of a Prolog structure, it provides a meaningful name and a generic description of the type. Once a Prolog database is described in Goomn, it can be mapped in principle to any object–oriented programming language due to its generic nature; see Listing 1.2 for the Goomn notation of the example database. The generated class structure in Python is identical with the opening Figure 1, and obviously it is much more user–friendly than the dynamically created class structure of Figure 2.

PyPlC creates a mapper class for each mapper and integrates it into the PyPlC module. By changing the Goomn notation, the mapping process is completely customizable: for instance, a user can decide whether a component should be mapped or not. Since one can compile the mapper classes, the processing speed can be significantly increased.

The Prolog Mapping Notation. Currently, we are still using an alternative to Goomn, the Prolog Mapping Notation (Pmn) inspired by [8]. Similar to Goomn, Pmn aims to provide a description for Prolog functors. Here, the information is saved into Prolog facts, instead of an external Xml file; see Listing 1.3 for the Pmn notation of the example database.

Listing 1.3. The Facts for the Predicate `student/6` in Pmn Notation.

```
functor(student, ['last_name', 'first_name',
    'age', 'male', 'account_balance', 'address'],
    [str, str, int, bool, float, functor(address)]).
functor(address,
    ['street', 'number', 'city', 'post_code'],
    [str, int, str, str]).
```

Listing 1.2. XML Schema Notation for the Predicate `student/6`.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace=
    "http://www.db.uni-passau.de/University">
  <xsd:element name="University" type="UniInfoType"/>
  <xsd:complexType name="Student"> ... </xsd:complexType>
</xsd:schema>

<xsd:complexType name="Student">
  <xsd:sequence>
    <xsd:element name="Last_Name" type="xsd:string"/>
    <xsd:element name="First_Name" type="xsd:string"/>
    <xsd:element name="Age" type="xsd:integer"/>
    <xsd:element name="Sex" type="xsd:string"/>
    <xsd:element name="Number" type="xsd:integer"/>
    <xsd:element name="Address">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="Street" type="xsd:string"/>
          <xsd:element name="Number" type="xsd:integer"/>
          <xsd:element name="City" type="xsd:string"/>
          <xsd:element name="Post_Code" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

In this case, the PMN structure of address has to be created first, because the predicate student references the predicate address. The generated functors consists of the predicate name, the arguments, and the related types. From the PMN notation, a Python programmer can immediately understand the structure of the original Prolog code.

3 The Python/Prolog Query Language

To keep the benefits of the underlying logic programming system, the query processing and data storage will be completely done by a Prolog interpreter.

We introduce the new Python/Prolog Query Language (PYPLQL), which is inspired by the query language JPQL of CAPJA. It is an easy–to–learn internal domain–specific language [4] based on corresponding structures mentioned above. In simple words, a PYPLQL query is a Python function, whose parameters declare the required objects and the function body describes the conditions

to be fulfilled by a Boolean expression. Such functions are not executed as they are; instead, they get transmitted to PyPlC translating, passing on to a Prolog instance and transferring the result into objects.

The syntax is defined in the following Listing 1.4. `queryName` is the name of the query and `[types]` represent the query types and the special query instructions. A query type is considered to be an object of a generated class according to the Python syntax, whereby a type hint has to be given for each single query type. The apparent objects are no actual objects (thus, they do not exist at running time), instead they declare which objects are required in that query. `constraints` defines the conditions the query types have to fulfill. Conditions can be made by an attribute of any query type or a hard–coded value as well as every built–in relational or Boolean operators of Python, used in the way they are permitted according to Python syntax. Additionally one can use brackets to define the order of evaluation.

Listing 1.4. Definition of a PyPlQL Query

```
def queryName [types]: constraints
```

A special query instruction is, for example, the `omit`–expression. If an attribute `a` of the query type `o` is declared by an `omit={o.a}`–expression, then the attribute `a` will be ignored during the query processing leading to a faster execution (see anonymous variable in Prolog). The value for such an attribute will be set to `None`. Queries, that omit an attribute, which is later accessed in the condition of a query, are invalid.

Other special query instructions are the aggregation expressions `min(o.a)`, `max(o.a)`, `avg(o.a)`, and `sum(o.a)`. They will retrieve the minimum, maximum, average, or summed values, respectively, of the attribute `a` of the query type `o`, for every group specified by `groupBy`.

PyPlC can delete all objects matching the constraint of a PyPlQL query, retrieve all of them, or retrieve just a single object and the other objects gradually, if requested (see backtracking in Prolog). Alternatively, PyPlC can be instructed to retrieve the number of matching objects. If an aggregation or `groupBy` expression is used in a PyPlQL query, then only all instances fulfilling the conditions can be retrieved or the number of matching objects. In a `groupBy` case, see, e.g., Listing 1.9, the return value is a (nested) Python dictionary, whose keys are the attributes mentioned in the `groupBy` clause and whose values are the results. Queries with joins can also be implemented in PyPlQL; see, e.g., Listing 1.8. This extends the Odbc approach.

3.1 Example Queries in PyPlQL

Assume that a user wants to determine every student of the Prolog university database whose surname is `'Doe'` and who is at least 18 years old. According to the created, corresponding class structure, this can be expressed as follows: determine every object of the class `Student`, whose attribute value for `last_name`

is equal to `'Doe'`, and whose attribute the value for `age` is greater than or equal to 18. This is a very natural way of expressing this regarding the object–oriented programming paradigm. Coming from that way of looking at the problem, it is easy to formulate the PyPlQL query, see Listing 1.5. In Python, atoms such as Student should start with capital characters. For a Python programmer, this query is easier to understand than the corresponding native Prolog code of Listing 1.6.

Listing 1.5. PyPlQL Query: Student over 18 with Last Name `'Doe'`

```
def studentQuery1 (s : Student, omit = {s.address}):
    s.last_name == 'Doe' and s.age >= 18
```

PyPlC translates the query `studentQuery1` from Listing 1.5 to the Prolog representation of Listing 1.6, which could also be send to a Prolog engine with Pengines.

Listing 1.6. Generated Prolog Representation of the Student Query

```
Name = 'Doe',
student(Name, Age, C, D, E, _),
Age >= 18.
```

The transformed return value is a set containing the single student of the university database, see Listing 1.7 (where Student and Address have been indented in Prolog style for readability, other than it would be in Python).

Listing 1.7. Result of the Example Query in Python

```
result = [
  Student('Doe', 'John', 20, male, 1000.00,
    Address('Park Avenue', 42, 'Duckburg', 12345))   ]
```

Another example query computes two students living in the same street. This can be expressed like in Listing 1.8.

Listing 1.8. PyPlQL Query: Two Students Living in the Same Street

```
def studentQuery2 (s1 : Student, s2 : Student):
    s1.address.street == s2.address.street
```

The number of students grouped by cities can be retrieved by a query like in Listing 1.9. I.e., PyPlQL includes aggregation and grouping.

Listing 1.9. PyPlQL Query: Number of Students Grouped by Cities

```
def studentQuery3
    (s : Student, groupBy = [s.address.city]): pass
```

3.2 Case Study: Querying and Result Processing in Python

In the case study of Listing 1.10, we want to demonstrate how a user can query the university data base step by step in order to print (result processing) information of students living in New York.

Listing 1.10. Case Study in Python: Querying Prolog

```
1  # Create an instance of Cappy
2  cappy = Cappy('swipl')
3
4  # Import the university Prolog database
5  # and its scheme
6  cappy.import_database('univeryity_db.pl',
7    'univeryity_db_goomn.xml')
8
9  # After examining the generated classes,
10 # the user can define the query
11 def student_query(s : Student):
12   s.address.city == 'Duckburg'
13 # and query the Prolog database
14 res = cappy.retrieveAll(student_query)
15
16 # Print all retrieved students
17 for s in res:
18   print(
19     'Name: '+ s.first_name + ' ' + s.last_name +
20     '; Address: ' + s.address.street + ', ' +
21     s.address.number + ' ' + s.address.city)
22 # Print output
23 # Name: John Doe; Address: Park Avenue, 42 ...
```

Although PyPlC aims to integrate Prolog from the Python point of view, it also supports some methods for Prolog experts. First, the user can use the dynamic mapping module of PyPlC . Second, he can formulate a native Prolog query and bypass the mapping component of PyPlC completely. In that case, the command lines 9–14 in the example above, can be replaced like in Listing 1.11 (where student and address have been indented in Prolog style for readability, other than it would be in Python). Since there are no explicit query types in a Prolog query, the result is not a student object, but a pre–processed Prolog result. Therefore, the user has to predict the structure of the result, in order to process it in Python. The command lines 16–21 in the previous example have to be changed like in Listing 1.12, since the result list will contain assignments for the Prolog variables given in the query, like `Last_name = 'Doe'`. Hence, the result processing becomes much more impractical. However, we consider this variant as an additional feature of PyPlC , if a user wants to have a direct access to Prolog; in this case, one might also prefer to use other tools like PythonPengines.

Listing 1.11. Case Study in Python: Querying Prolog

```
# After examining the generated classes,
# the user can define the query
student_query_prolog =
   'student( Last_name, First_name, _, _, _,
        address(Street, Number, City, _) )'
# and query the Prolog database
res = cappy.retrieveAll(
   student_query_prolog, bypassing = True)
```

Listing 1.12. Case Study in Python: Processing Prolog Result

```
# Print all retrieved students
for r in res:
  for a in r:
    if isinstance(a, PrologAssignment) and
      a.variable = 'Last_name':
      print('Last Name: ' + r.value + '; ')
    elif isinstance(a, PrologAssignment) and
      a.variable = 'First_name':
      print('First Name: ' + r.value + '; ')
    elif isinstance(a, PrologAssignment) and
      a.variable = 'Street':
      print('Street: ' + r.value + '; ')
    elif isinstance(a, PrologAssignment) and
      a.variable = 'Number':
      print('Number: ' + r.value + '; ')
    elif isinstance(a, PrologAssignment) and
      a.variable = 'City':
      print('City: ' + r.value + '; ')
# Print output
# Last Name: Doe; First Name: John; ...
```

4 Discussion

Instead of only offering a Prolog–like interface, PyPlC allows intuitive queries to a Prolog database using an object–oriented query language. This approach has a couple of advantages.

PyPlC does not only use Python structures, it also uses them in a natural way. Especially, no prerequisite knowledge of Prolog is needed. PyPlC is compatible with current versions of Swi–Prolog and Python. In principle there is a support for every Prolog implementation, since PyPlC only uses Iso–Prolog syntax [6] and includes an abstract layer offering an easy–to–extend interface

for a specific Prolog implementation. But so far, PyPlC only supports Swi–Prolog. Interchangeability of a specific implementation means a gain of quality in software development.

The new Xml–based markup language Goomn enables a standardized notation of Prolog functors. A Goomn–notated database can be integrated in any other programming language, since it only uses a very general syntax. Also non–notated functors can be available in Python through dynamic mapping, whereas a user has to use another tool to understand the Prolog database, for example the visualization module of the system Declare for data and knowledge engineering [10].

Another way to use PyPlC is to integrate not only deductive databases, but more complex Prolog libraries enabling a powerful way of calculating. For instance, the Prolog library ClioPatria [12] offers a convenient access to Rdf databases. In this way PyPlC also offers an interface to the semantic web.

Also in the Prolog library Declare, there exist extended methods for queries to relational and deductive databases, cf., e.g.,. the generic aggregation operator ddbase_aggregate, which extends the standard Prolog operator findall by grouping and user–defined aggregation functions.

5 Conclusions and Future Work

The proposed tool PyPlC offers a new unified and intuitive way for the integration of Prolog into Python. Derived Prolog facts can be retrieved to Python; all results for a Prolog query are derived like in deductive databases. Whereas in deductive databases, derivations usually are bottom–up, they are top–down here. But, bottom–up derivations can be implemented on the Prolog side, e.g. using Declare, to ensure termination for more logic programs, such as, e.g., the ones for cyclic transitive closure queries.

By automatically creating a corresponding Python class structure of a Prolog database and by providing a Python–like, object–oriented query language, PyPlC ensures an easy access to Prolog, where not much prior knowledge of Prolog is necessary.

This paper describes work in progress. We consider improvements of the new approach in our future work. It is impossible to create a Prolog database from Python. Thus, PyPlC is an integration of Prolog into Python, and not vice versa. Creating objects and mapping them into a relational scheme causes many problems, and Python users could face unexpected behaviour in such cases.

The PyPlQL syntax should be extended in a way permitting every built–in Python command to be used for formulating a query. For example, for retrieving whether a substring is contained in a string, the Python command **in** is suitable. A modified PyPlQL query of Listing 1.5 – asking only for surnames containing the substring `'man'` – can be seen in Listing 1.13.

Listing 1.13. PyPlQL Query with Substring Condition

```
def studentQuery1_substr (s : Student):
    'man' in s.last_name and s.age >= 18
```

In some cases, it is possible to translate a PyPlQL query during compilation, which would bring a significant speedup of the execution.

There are many fascinating Prolog applications. However, many programmers do not use these tools, because there are obstacles in terms of different paradigms compared to many wide–spread programming languages. PyPlC aims to spread interest in Prolog. Therefore, it provides a way for Prolog beginners, without getting into Prolog specifics. In order to overcome this entry barrier, PyPlC generates corresponding structures of a Prolog database in Python. By using an easy–to–learn query language, a beginner can use Prolog tools straightforwardly.

Acknowledgements. The authors would like to thank Mirco Lukas and Ludwig Ostermayer for their helpful suggestions during the master's thesis of Stefan Bodenlos, and also Falco Nogatz for his useful comments on an earlier draft of the paper.

References

1. Ian Andrich. PythonPengines. `https://github.com/ian-andrich/PythonPengines`.
2. Stefan Bodenlos. Integration von Prolog und ClioPatria in Python. Master's thesis, University of Würzburg, 2017.
3. Ivan Bratko. *Prolog programming for artificial intelligence*. Pearson Education, 2001.
4. Martin Fowler. *Domain–specific languages*. Pearson Education, 2010.
5. Christopher Ireland, David Bowers, Michael Newton, and Kevin Waugh. A classification of object–relational impedance mismatch. pages 36–43, 2009.
6. ISO. `https://www.iso.org/standard/21413.html`.
7. Torbjörn Lager and Jan Wielemaker. Pengines: Web logic programming made easy. *Theory and Practice of Logic Programming*, 14(4–5):539–552, 2014.
8. Ludwig Ostermayer. *Integration of Prolog and* Java *with the connector architecture* CAPJa. PhD thesis, University of Würzburg, 2017.
9. Fernando Pereira and Stuart Shieber. *Prolog and natural–language analysis*. Microtome Publishing, 2002.
10. Dietmar Seipel. The Declare developers' toolkit (Ddk). `http://www1.informatik.uni-wuerzburg.de/database/DisLog/`.
11. Guido van Rossum and Python Development Team. The Python language reference. `https://docs.python.org/3/archives/python-3.6.3rc1-docs-pdf-a4.zip`.
12. Jan Wielemaker, Wouter Beek, Michiel Hildebrand, and Jacco van Ossenbruggen. ClioPatria: A Swi–Prolog infrastructure for the semantic web. *Semantic Web*, 7(5):529–541, 2016.