JPC: A Library for Modularising Inter-Language Conversion Concerns between Java and Prolog

Sergio Castro[a], Kim Mens[a], Paulo Moura[b]

[a] *Université catholique de Louvain, Belgium*
[b] *Center for Research in Advanced Computing Systems, INESC-TEC, Portugal*

Abstract

The number of approaches existing to enable a smooth interaction between Java and Prolog programs testifies the growing interest in solutions that take advantage of the strengths of both languages. Most of these approaches provide limited support to allow programmers to customise how a Prolog artefact should be reified in the Java world, or how to reason about Java objects on the Prolog side. The burden of creating a convenient mapping between inter-language artefacts is left, however, to the user of the library. This is an error-prone task since the appropriate mapping often depends on the particular context of usage. Although some libraries alleviate this problem by providing higher-level abstractions to deal with the complexity of converting programming artefacts between the two languages, these libraries themselves are difficult to implement and evolve. This is caused by the lack of appropriate underlying building blocks for modularising and managing Java-Prolog conversion constructs. We therefore introduce a library intending to serve as a portable development tool both for programmers willing to modularise context-dependent conversion constructs in their Java-Prolog systems, and for architects implementing frameworks providing higher-level abstractions for better interoperability between these two languages.

Keywords: Object-Oriented Programming, Logic Programming, Multi-Paradigm Programming, Programming Language Interoperability, Separation of Concerns

1. Introduction

Several libraries and frameworks have been proposed in the past to enable both Java programmers to call Prolog predicates and Prolog programmers to have access to Java's extensive set of libraries. In Castro et al. (2013, 2012) we presented a linguistic symbiosis approach to facilitate the implementation of certain methods in a Java class using Prolog. Our focus was on attaining a transparent and semi-automatic integration of Prolog within the Java language by providing a simple mechanism for inferring the appropriate mappings

Email addresses: `sergio.castro@uclouvain.be` (Sergio Castro), `kim.mens@uclouvain.be` (Kim Mens), `pmoura@inescporto.pt` (Paulo Moura)

between Java and Prolog artefacts. Default mappings were defined for typical scenarios and a mechanism of customising these mappings, by means of annotations, was provided.

Although we accomplished our objectives, our framework was difficult to extend to other customisation mechanisms (different from the built-in one based on annotations) and not portable to Prolog engines other than the ones we targeted. In addition, our implementation was hard to maintain. This was mainly because methods dealing with artefact conversions (i.e., the mechanism defining how an artefact in one language should be represented into the other) had to analyse a considerable amount of conditions to infer the appropriate conversion to apply. This caused a high cyclomatic complexity in the implementation, aggravated by the fact that the right conversion, as we demonstrated, often depends on a specific usage context. Furthermore, conversion algorithms were often tangled with other unrelated concerns.

The present work builds upon the difficulties we faced and the lessons we learned. Its outcome is a tool, inspired on the Gson library (Google Inc. (2012)) for modularising context-dependent conversion concerns between inter-language artefacts.

This paper is structured as follows. In section 2 we discuss the intrinsic complexity of Java–Prolog interaction, revisiting an example presented in a previous work. The architecture and main components of our library are discussed in section 3. In section 4 we present, using small examples, the core of our technique for modularising context-dependent conversions between Java objects and Prolog terms. Section 5 discusses approaches that inspired our technique. In section 6 we summarise our conclusions from the perspective of a tool builder and outline our future work plans.

2. Java–Prolog Interaction Complexity

In Castro et al. (2013, 2012), we illustrated with an example the inherent complexity of mapping between Java artefacts and Prolog terms. We revisit in this section some details of this example to help explaining the nature of our problem. Our case study modelled a city underground system, with *stations*, *lines* connecting stations, and the *underground* as the objects of discourse. To facilitate the mapping between Java and Prolog artefacts, we use Logtalk (Moura (2003)), an object-oriented logic programming language that is implemented in Prolog. Logtalk is highly portable, allowing us to work with most Prolog implementations, using LogTalk as middle-layer. An outline of the Logtalk `station` parametric object (Moura (2011)) is shown in listing 1. The implementation of the public method `connected/2` is shown on line 4. This method answers stations directly connected to the receiver by means of the `line` object received as second argument. The method delegates to the `connected/3` method of the `metro` object (that is not shown here for brevity).

```
1 :- object(station(_Name)).
2   :- public([connected/2, ...]).
3
4   connected(S, L) :- self(Self), metro::connected(Self, S, L).
5
6   ...%other Logtalk methods
7 :- end_object.
```

Listing 1: The `station` object in Logtalk

On the Java side, listing 3 shows an outline of a corresponding `Station` class. This class has been implemented using JPL (Singleton et al. (2004)), a well-known library for Java–Prolog interaction. This class defines how a `Station` instance should be represented on the Prolog side by means of the method `asTerm()` (lines 5–7). In our example, a station is represented in Prolog as a compound term using the functor `station` and an atom as single argument representing the name of the station (line 6). The static method `create(Term)` (lines 9–12) does the opposite: it defines how a station term defined on the Prolog side has to be represented as a Java object. It takes the first argument of the term representing the station's name (line 10) and uses it to instantiate the `station` class (line 11). The method `connected(Line)` (lines 14–27) returns a station connected to the receiver by means of the line received as a parameter. First a term representing a Logtalk message is built on line 18. The arguments of this message are an unbound variable and the term representation of the line object received as parameter (line 17). Then an object message is built on line 19. Logtalk uses the `::/2` infix operator for sending a message to an object. Its left operand is the receiver (in this case the result of invoking the `asTerm()` method) and the right operand is a Logtalk message with its arguments. Thus, the query created on line 20 is interpreted in Prolog as:

```
station(station_name)::connected(ConnectedStation, line(line_name))
```

Listing 2: A Logtalk query

Once the query has been constructed, we request its first solution on line 21. The binding for the variable sent as first argument to the Logtalk method is collected from the solution on line 23. This variable has been bound to a Prolog term representing a station. On line 24 we create the Java representation of this station object and we return it on line 26.

```
1  public class Station {
2      String name;
3      ...
4      //mapping an instance of Station to a logic Term
5      public Term asTerm() {
6          return new Compound("station", new Term[] {new Atom(name)});
7      }
8      //mapping a logic Term to an instance of Station
9      public static Station create(Term stationTerm) {
10         String lineName = ((Compound)stationTerm).arg(1).name();
11         return new Station(lineName);
12     }
13     //mapping a Java method to a Logtalk method
14     public Station connected(Line line) {
15         Station connectedStation = null;
16         String stationVarName = "ConnectedStation";
17         Term[] arguments = new Term[]{new Variable(stationVarName), line.asTerm()};
18         Term message = new Compound("connected", arguments);
19         Term objectMessage = new Compound("::", new Term[] {asTerm(), message});
20         Query query = new Query(objectMessage);
21         Hashtable<String, Term> solution = query.oneSolution();
22         if(solution != null) {
23             Term connectedStationTerm = solution.get(stationVarName);
24             connectedStation = create(connectedStationTerm);
25         }
26         return connectedStation;
27     }
28     ...//other methods mapped to logic routines
29 }
```

Listing 3: A Java class interacting with Prolog by means of JPL

As illustrated by this example, for each query, the programmer must write the necessary code to convert multiple Java objects to a convenient Prolog term representation (e.g., lines 17 and 19) and to convert Prolog terms back to Java objects (e.g., line 24). As the correct conversion may depend on the usage context, this is an error-prone activity, often difficult to separate from the core logic of the problem the programmer is trying to solve. In the next section we discuss our library that alleviates most of these problems.

3. Architecture of the library

In this section we describe the main components of our library. This discussion sets the ground for introducing its features in section 4.

3.1. Reification of Prolog Data Types

Our library provides a set of classes reifying Prolog data types. These classes are inspired by (and to certain extent equivalent to) term classes in the JPL library [1]. For completeness, we list the main classes here:

Term : An abstract Prolog term.
Atom : A sequence of characters representing a Prolog Atom.
Compound : A Compound term consisting of a name and a list of arguments.
IntegerTerm : A Prolog Integer term.
FloatTerm : A Prolog Float term.
Variable : A Prolog Variable term.
Query : The reification of a Prolog query with convenient methods for obtaining its solutions, where each solution is a map of Prolog variable names to terms.

As in the JPL library, these term classes can be considered as a structured concrete syntax for Prolog terms.

3.2. An Abstraction of a Prolog Virtual Machine

These classes implement an abstract Prolog Engine that is able to accept queries and answer solutions employing our term representation. It is vendor-agnostic and requires a driver to connect to a specific Prolog engine. Drivers can be built on top of existing Java–Prolog libraries for the target Prolog engine. These libraries can be made available by the Prolog engine provider (e.g., JPL) or by third parties (e.g., InterProlog: Calejo (2004)). A complete detailed description of our abstraction of a Prolog virtual machine is outside the scope of this paper.

[1]However, the semantics of certain equally named methods have been modified.

3.3. The Conversion Context

The core of our library is inspired by Google's Gson library, which aims to provide a high-level tool for conversions between Java objects and their JSon representation. In fact, many aspects of our library can be regarded as a re-implementation of Gson in the domain of Java–Prolog artefact conversions.

The primary class in our library is a conversion context, encapsulated by the `Jpc` class [2]. This context can be considered as a two–way conversion strategy for a set of Java Objects and Prolog terms. In addition, it is immutable so it can safely be reused across multiple conversion operations. Its main components are:

The converter manager : Responsible to convert a Java object to a Prolog term and vice-versa.

The type solver : Resolves the Java type of a Prolog term.

The instantiation manager : Allows to customise the instantiations of abstract classes or interfaces.

In order to facilitate the setup of these components in a `Jpc` instance, we provide a `JpcBuilder` class with convenient configuration methods. For example, listing 4 shows how to configure a builder to create a `Jpc` context that knows how to convert objects from the example discussed in section 2. We will comeback to this example in the next section.

```
public static final Jpc jpcContext = JpcBuilder.create()
    .registerConverter(new MetroConverter())
    .registerConverter(new LineConverter())
    .registerConverter(new StationConverter()).build();
```

Listing 4: Building a `Jpc` context with the `JpcBuilder` class

4. Modularizing Inter-Language Conversion Concerns

The `Jpc` class provides convenient methods to convert between Java and Prolog artefacts. In the rest of this section we overview these techniques.

4.1. Primitives Conversions

In this section we illustrate how to map primitive types between Java and Prolog. The simplest way to use our framework is by means of the `toTerm(Object)` and `fromTerm(Term)` methods in the `Jpc` class. These methods require as a parameter the object to convert.

Listing 5 shows a list of successful assertions that illustrates some pre-defined conversions of Java primitive types and strings to Prolog terms.

```
assertEquals(new Atom("true"), jpc.toTerm(true));  //Boolean to Atom
assertEquals(new Atom("c"), jpc.toTerm('c'));  //Character to Atom
assertEquals(new Atom("1"), jpc.toTerm("1"));  //String to Atom
assertEquals(new IntegerTerm(1), jpc.toTerm(1));  //Integer to IntegerTerm
assertEquals(new FloatTerm(1), jpc.toTerm(1D));  //Double to FloatTerm
```

Listing 5: Conversions of primitive Java objects to Prolog terms

[2] *Jpc* stands from Java-Prolog-Connectivity.

Pre-defined conversions of Prolog terms to Java primitive types are shown in listing 6.

```java
assertEquals(true, jpc.fromTerm(new Atom("true")));  //Atom to Boolean
assertEquals("c", jpc.fromTerm(new Atom("c")));  //Atom to String
assertEquals("1", jpc.fromTerm(new Atom("1")));   //Atom to String
assertEquals(1L, jpc.fromTerm(new IntegerTerm(1)));   //IntegerTerm to Long
assertEquals(1D, jpc.fromTerm(new FloatTerm(1)));   //FloatTerm to Double
```

Listing 6: Conversion of Prolog terms to primitive Java objects

Note that, f being our default conversion function from a Java object to a Prolog term, and g our default reverse conversion function, it is not always the case that $g(f(x)) = x$, where x is a Java primitive object. This is because there are more primitive types in Java that in Prolog. Thus, distinct Java objects may be mapped by default to the same Prolog term. For example, line 2 of listing 5 shows that the default conversion of the Java character `c` is the Prolog atom `c`. However, the default conversion of the atom `c` is the String `"c"`, but this is not necessarily always what the programmer expects. The next section describes how to give a hint to our library on the appropriate conversion that should be applied.

4.2. Typed Conversions

The conversion methods in the `Jpc` class can receive as a second parameter the expected type of the converted object. Listing 7 shows examples of Java–Prolog conversions that specify the expected Prolog term type. In line 1, the `Integer` 1 is converted to an `Atom` instead of an `IntegerTerm` (as in listing 5, line 4). This is because we send the `Atom` class as the second parameter of the conversion method. In line 2, the `String "1"` is converted to an `IntegerTerm`.

```java
1 assertEquals(new Atom("1"), jpc.toTerm(1, Atom.class));  //Ingeter to Atom
2 assertEquals(new IntegerTerm(1), jpc.toTerm("1", IntegerTerm.class));  //String to IntegerTerm
```

Listing 7: Typed conversions of primitive Java objects to Prolog terms

In a similar way, listing 8 shows examples of Prolog–Java conversions that specify the expected Java type.

```java
assertEquals(1, jpc.fromTerm(new Atom("1"), Integer.class));  //Atom to Integer
assertEquals("1", jpc.fromTerm(new IntegerTerm(1), String.class));  //IntegerTerm to String
assertEquals("true", jpc.fromTerm(new Atom("true"), String.class));  //Atom to String
assertEquals('c', jpc.fromTerm(new Atom("c"), Character.class));  //Atom to Character
```

Listing 8: Typed conversions of Prolog terms to primitive Java objects

4.3. Arrays, Collections, and Maps Conversions

Our library provides default conversions for multi-valued data types such as arrays, collections, and maps.

```java
Term term = jpc.toTerm(new Object[]{"apple", 10});
assertEquals(
new Compound(".", asList(new Atom("apple"),  //equivalent to .('apple', .(10, []))
    new Compound(".", asList(new IntegerTerm(10),
    new Atom("[]")))))
, term);
```

Listing 9: Conversion of an array to a Prolog term

Listing 9 shows a conversion of an array object with a string and an integer element: `["apple", 10]`. Its result is a Prolog term list having as elements an atom and an integer term: `['apple', 10]`.

Alternatively, we could have used a list instead of an array. We would have obtained exactly the same result by replacing line 1 by: `Term term = jpc.toTerm(asList("apple", 10));`

A slightly more complex example is illustrated in listing 10. First, a Java map is instantiated (lines 1–4). The default term conversion is applied on line 5, generating a Prolog list with two key-value pairs: `['apple'-10,'orange'-20]`. This result is tested on lines 7–8.

```
1 Map<String, Integer> map = new LinkedHashMap<String, Integer>() {{ //LinkedHashMap preserves insertion
      order
2     put("apple", 10);
3     put("orange", 20);
4 }};
5 Term term = jpc.toTerm(map);
6 List<Term> listTerm = term.asList();  //transforms a Prolog list term into a list of terms
7 assertEquals(new Compound("-", asList(new Atom("apple"), new IntegerTerm(10))), listTerm.get(0));
8 assertEquals(new Compound("-", asList(new Atom("orange"), new IntegerTerm(20))), listTerm.get(1));
```

Listing 10: Conversion of a map to a Prolog term

4.4. Typing Prolog Terms

When no type information is provided in a term-to-object conversion, our library will attempt to give as hint to the converter manager an inferred type (if any) based on structural properties of the term. For instance, by convention we assume (but this could be configured) that a Prolog list of terms with certain properties should be reified as a map in Java. Listing 11 illustrates this by means of an example. On line 3 we create a list term from two previously created compound terms. We convert it to a Java map on line 4 and test its structure on lines 5 and 6. In this case, our library will infer that the type of the converted object should be a Java `Map`, since the (default) type solver will analyse the structure of the term to convert: `[apple-10, orange,20]` and will find that all of its elements are compounds with an arity of 2 and with functor '-', which are mapped by default to map entries (i.e., instances of the `Map.Entry` class).

```
1 Compound c1 = new Compound("-", asList(new Atom("apple"), new IntegerTerm(10)));
2 Compound c2 = new Compound("-", asList(new Atom("orange"), new IntegerTerm(20)));
3 Term listTerm = listTerm(c1, c2); //creates a list term from a list of terms
4 Map map = jpc.fromTerm(listTerm);
5 assertEquals(map.get("apple"), 10L);
6 assertEquals(map.get("orange"), 20L);
```

Listing 11: Conversion of a Prolog term to a map

Alternatively, we could have replaced line 4 by
`List list = jpc.fromTerm(listTerm, List.class);`
The type explicitly given by the programmer as a hint has higher priority that the one inferred by the type solver. In this case, the result would have been a list of map entries since the Prolog list would have been mapped to a Java list (i.e., an instance of a class implementing `List`), but the default conversion of each term in the list (a compound with arity 2 and functor '–') is still a map entry object.

Listing 12 shows an extract of this type solver. It will return the `Map` class on line 13 if it can conclude that the term looks like a map. If it is unable to assign a type to the term it will return `null` on line 15. Note that this type solver may answer false negatives if it does not have enough information (e.g., if the list term is empty). The programmer should explicitly supply a type in case of ambiguities.

```java
1  public class MapTypeSolver implements TermTypeSolver {
2      @Override
3      public Type getType(Term term) {
4          if(term.isList()) {
5              ListTerm list = term.asList();
6              Predicate<Term> isMapEntry = new Predicate<Term>() {
7                  @Override
8                  public boolean apply(Term term) {
9                      return isMapEntry(term);
10                 }
11             };
12             if(!list.isEmpty() && Iterables.all(list, isMapEntry))
13                 return Map.class;
14         }
15         return null;
16     }
17
18     private boolean isMapEntry(Term term) {
19         ...
20     }
21 }
```

Listing 12: A type solver for a Prolog term representing a map

Additional type solvers can be added to a conversion context by means of the `registerTypeSolver(TermTypeSolver)` method of the `JpcBuilder` class.

4.5. Instantiation Managers

When a type hint is given for a Prolog–Java conversion (either explicitly by the programmer or implicitly by the type solver) and a converter does not know the right way to instantiate it (e.g., it is abstract or an interface), it can ask the instantiation manager for an instance of the type. For example, in listing 12 we showed that a Prolog list with a certain structure will be identified by the type solver as a `Map`. However, the type solver does not provide any mechanism to instantiate such an interface, since its only responsibility is to give a hint on the appropriate Java type. Converters may use the instantiation manager to instantiate abstract types. An instantiation manager administers a collection of instance creators, which are classes implementing the interface shown in listing 13. When required to instantiate a type, it will iterate on the registered instance creators querying their `canInstantiate(Type)` method. If one instance creator returns `true`, the manager will return the result of invoking `instantiate(Type)` on the instance creator.

```java
public interface InstanceCreator {
    public <T> T instantiate(Type type);

    public boolean canInstantiate(Type type);
}
```

Listing 13: The `InstanceCreator` interface

Instance creators can be added to a conversion context by means of the `registerInstanceCreator(InstanceCreator)` method of the `JpcBuilder` class.

4.6. Conversion of Generic Types

Our library provides support for Java Generics. Consider the example in listing 14. A Prolog list term is created on line 1. We use a utility class (from Google's Guava library) to obtain an instance of the parameterised type `List<String>` (line 2). Then we give this type as a hint to the converter (line 3) and we verify on lines 4 and 5 that the elements of the Java `List` are indeed instances of `String`, as it was specified on line 3.

```
1 Term listTerm = listTerm(new Atom("1"), new Atom("2"));
2 Type type = new TypeToken<List<String>>(){}.getType();
3 List<String> list = jpc.fromTerm(listTerm, type);
4 assertEquals("1", list.get(0));
5 assertEquals("2", list.get(1));
```

Listing 14: Specifying redundantly the target parameterised type in a conversion

In the previous example, the type passed to the converter was redundant, since elements in the Prolog list are atoms, which are converted by default to instances of `String` in Java. Consider, however, listing 15. The main change w.r.t. the previous example is that the type we send as a hint is now `List<Integer>` (line 3). This instructs the converter to instantiate a list where all its elements are integers, as demonstrated on lines 4 and 5.

```
1 Term listTerm = listTerm(new Atom("1"), new Atom("2"));
2 Type type = new TypeToken<List<Integer>>(){}.getType();
3 List<Integer> list = jpc.fromTerm(listTerm, type);
4 assertEquals(1, list.get(0));
5 assertEquals(2, list.get(1));
```

Listing 15: Changing the behaviour of the converter with a parameterised type

4.7. Custom Conversions

The previous examples have employed only pre-defined converters. Listing 16 shows the `StationConverter` class. This class has two main methods to convert to (lines 4–6) or from (lines 8–11) a Prolog term.

```
 1 public class StationConverter extends JpcConverter<Station, Compound> {
 2     public static final String STATION_FUNCTOR = "station";
 3 
 4     @Override public Compound toTerm(Station station, Jpc context) {
 5         return new Compound(STATION_FUNCTOR, asList(new Atom(station.getName())));
 6     }
 7 
 8     @Override public Station fromTerm(Compound term, Jpc context) {
 9         String stationName = ((Atom)term.arg(1)).getName();
10         return new StationJpc(stationName);
11     }
12 }
```

Listing 16: The `StationConverter` class

Using this converter, and the others shown in listing 4 to create a custom conversion context, we re–visit the `Station` class shown in listing 3.

Listing 17 shows a new version of the `Station` class originally shown in listing 3. Using our library, the `connected(Line)` method was reduced from 14 to 7 lines of code. In addition, the methods `asTerm()` and `create(Term)` are not in the `Station` class anymore since they have been encapsulated in a converter. Note that terms are easily created according to a conversion context. In line 5, the last argument of the compound is an instance of `Line`. The conversion of this object to a term is done automatically by our framework. Conversely, in listing 3 (line 17), we were forced to invoke an explicit conversion when we requested the term representation of the line object. The same applies in line 6, where the `Station` instance denoted by the `this` keyword is automatically transformed to its term representation.

A `Query` object is instantiated on line 7 from an object abstracting a Prolog engine. Note that this object can (optionally) receive a context. The advantage of making a query instance aware of a conversion context becomes clear on line 8. To understand this, let's recall from section 3 that a Prolog solution is represented as a map binding variable names to terms. On line 8, the invocation of the `selectObject(String)` method encapsulates the original query in an adapter, where each solution of this query adapter is an object which term representation is given in the argument of `selectObject`, taking into account the bindings of any variables in the solution. In our example, the solution object is expressed as the Prolog variable `Station`, which has been bound to a term representing an instance of `Station`. The conversion of this term to a `Station` object is transparent and accomplished behind the curtains by our library.

```
 1 public class Station {
 2     ...
 3     public Station connected(Line line) {
 4         String stationVarName = "Station";
 5         Term message = jpcContext.compound("connected", asList(new Variable(stationVarName), line));
 6         Term objectMessage = jpcContext.compound("::", asList(this, message));
 7         Query query = getPrologEngine().query(objectMessage, jpcContext);
 8         return query.<Station>selectObject(stationVarName).oneSolution();
 9     }
10 }
```

Listing 17: A Java class interacting with Prolog by means of our library

5. Related Work

Most techniques and abstractions employed by our library were not invented from scratch but taken from existing tools in other domains. In particular, Google's Gson library and its notion of a context were an important source of inspiration for accomplishing our two-way conversions between Java and Prolog artefacts. In addition, although our abstract Prolog engine was only briefly discussed in this paper, its core idea of decoupling a driver into a common API and an engine-specific component has been extensively proven in scenarios involving programs interacting with databases (e.g., JDBC).

In the domain of Java–Prolog connectivity, most classes reifying Prolog terms were inspired on the JPL library. Certain aspects of InterProlog were also inspiring for the design of our class modelling a Prolog engine.

6. Conclusion and Future Work

The tool presented in this paper was originally designed as a portable library providing convenient functionality for a Java–Prolog linguistic symbiosis problem. We believe, however, that it can be useful to programmers wanting to modularise and encapsulate conversion concerns in Java–Prolog programs, given them a fine-grained control about how a Java object can be reasoned about on the Prolog side, and which is the best representation of a Prolog artefact in Java. It can also serve as a convenient building block for helping architects in building efficiently more sophisticated frameworks (e.g., different to the one presented in our previous work) for integrating Java and Prolog.

While building this tool we have profited from cross fertilisation of ideas from different domains. We believe that tool design and implementation requires an attentive observation of different domains where a similar problem may exist, perhaps with subtle variations. As we have demonstrated, tool building can be an exercise of assembling a puzzle of abstractions, where many pieces (possibly all) can be adapted from, or inspired on, well-proven existing solutions.

Regarding future work, we plan to port the framework for linguistic symbiosis introduced in Castro et al. (2013, 2012) to our library, thus simplifying its implementation. To improve the Prolog–Java interoperability, we plan to develop a portable library on top of Logtalk for simplifying the interaction from the Prolog perspective, as we have done on the Java side. We also plan to extend support to Prolog compilers other than the currently supported SWI–Prolog (Wielemaker et al. (2012)), YAP (Costa et al. (2012)) and XSB (Swift and Warren (2012)).

Acknowledgements. This work is partially supported by the LEAP project (PTDC/EIA-CCO/112158/2009), the ERDF/COMPETE Program and by FCT project FCOMP-01-0124-FEDER-022701.

References

Calejo, M., 2004. InterProlog: Towards a Declarative Embedding of Logic Programming in Java. In: José Júlio Alferes and João Alexandre Leite (Ed.), Logics in Artificial Intelligence, 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004, Proceedings. Vol. 3229 of Lecture Notes in Computer Science. Springer, pp. 714–717.

Castro, S., Mens, K., Moura, P., 2012. LogicObjects: A Linguistic Symbiosis Approach to Bring the Declarative Power of Prolog to Java. In: Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'12).

Castro, S., Mens, K., Moura, P., January 2013. LogicObjects: Enabling Logic Programming in Java Through Linguistic Symbiosis. In: Proceedings of the 15th International Symposium on Practical Aspects of Declarative Languages (PADL). Rome, Italy.

Costa, V. S., Rocha, R., Damas, L., 2012. The YAP Prolog System. Theory and Practice of Logic Programming 12 (1-2), 5–34.

Google Inc., Jul. 2012. Gson 2.2.2: A Java library to convert JSON strings to Java objects and vice-versa. `http://code.google.com/p/google-gson/`.

Moura, P., Sep. 2003. Logtalk – Design of an Object-Oriented Logic Programming Language. Ph.D. thesis, Department of Computer Science, University of Beira Interior, Portugal.

Moura, P., Apr. 2011. Programming Patterns for Logtalk Parametric Objects. In: Applications of Declarative Programming and Knowledge Management. Vol. 6547 of Lecture Notes in Artificial Intelligence. Springer-Verlag, pp. 52–69.

Singleton, P., Dushin, F., Wielemaker, J., Feb. 2004. JPL 3.0: A bidirectional interface between Prolog and Java. `http://www.swi-prolog.org/packages/jpl/java_api/`.

Swift, T., Warren, D., 2012. XSB: Extending the power of Prolog using tabling. Theory and Practice of Logic Programming 12 (1-2), 157–187.

Wielemaker, J., Schrijvers, T., Triska, M., Lager, T., 2012. SWI-Prolog. Theory and Practice of Logic Programming 12 (1-2), 67–96.