Customisable Handling of Java References in Prolog Programs

SERGIO CASTRO, KIM MENS and PAULO MOURA

ICTEAM Institute, Université catholique de Louvain, Belgium
CRACS & INESC TEC, Faculty of Sciences, University of Porto
(*e-mail:* {sergio.castro,kim.mens}@uclouvain.be,pmoura@inescporto.pt)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Integration techniques for combining programs written in distinct language paradigms facilitate the implementation of specialised modules in the best language for their task. In the case of Java-Prolog integration, a known problem is the proper representation of references to Java objects on the Prolog side. To solve it adequately, multiple dimensions should be considered, including reference representation, opacity of the representation, identity preservation, reference life span, and scope of the inter-language conversion policies. This paper presents an approach that addresses all these dimensions, generalising and building on existing representation patterns of foreign references in Prolog, and taking inspiration from similar inter-language representation techniques found in other domains. Our approach maximises portability by making few assumptions about the Prolog engine interacting with Java (e.g., embedded or executed as an external process). We validate our work by extending JPC, an open-source integration library, with features supporting our approach. Our JPC library is currently compatible with three different open source Prolog engines (SWI, YAP and XSB) by means of drivers.

KEYWORDS: Multi-Paradigm Programming, Language Interoperability, Logic Programming, Object-Oriented Programming, Prolog, Java

1 Introduction

Writing program modules in the language best suited for their task can greatly facilitate their implementation (Mernik et al. 2005; Omicini and Natali 1994; D'Hondt 2004). However, integrating modules written in different languages is not trivial when such languages belong to different paradigms (Gybels 2003). This is especially the case for Prolog programs integrated with an object-oriented language such as Java (Denti et al. 2005; Hermenegildo et al. 2012; Calejo 2004). One of the main problems of this integration is the proper representation of foreign language artefacts in the logic language, such that they can be conveniently manipulated and interpreted (Wuyts and Ducasse 2001).

The scope of this work concerns a portable approach to simplify the management and representation of Java object references in Prolog. Studying existing solutions to this problem in Prolog, similar logic languages (e.g., Soul (Roover et al. 2011)) and even inter-language conversion libraries in other domains (e.g., Google's Gson library (Google Inc. 2012)), we have identified the following dimensions to be tackled: 1) reference representation; 2) opacity of the representation; 3) identity preservation; 4) reference life span and 5) scope of the inter-language conversion policies. To maximise portability, our approach does not make any simplifying assumption regarding the architecture of the Prolog engine (e.g., such as it being embedded in the JVM). We validate our work by extending our Java Prolog Connectivity (JPC) [1] integration library (Castro et al. 2013) with customisable support for managing Java references in Prolog.

This paper is structured as follows. The next section discusses the different dimensions to be considered when representing Java references in Prolog programs. A short overview of JPC's architecture is given in section 3. JPC's approach for custom management of Java references in Prolog is discussed in section 4. Section 5 discusses related work. Section 6 presents our conclusion and future work plans.

2 The Problem of Representing Java References in Prolog

In this section, we identify the different dimensions to be taken into consideration when looking at the problem of representing Java references in Prolog (figure 1). These dimensions have been extracted and generalised from existing solutions to this problem both in Prolog and other inter-language representation domains.

2.1 Reference Representation

A first important dimension is how Java objects are represented on the logic side. Several integration libraries allow to reify Java objects in Prolog using a symbolic term representation (e.g., JPL (Singleton et al. 2004), InterProlog (Calejo 2004), Jasper (Carlsson et al. 1995) or the Ciao Java interface (Bueno et al. 2011)). As show in figure 1, such approach has the advantage of not relying on any specific Prolog engine architecture.

Alternatively, Prolog implementations running in the JVM may support the storage of direct object references (e.g., Jinni (Tarau 2004) and LeanProlog (Tarau 2011)). An advantage of this representation scheme is that there are no performance penalties associated to the marshalling/unmarshalling of Java objects to/from the Prolog engine.

2.2 Opacity of the Representation

A second important dimension is the degree of opacity (i.e., the degree of data exposed) when reifying a Java object reference in Prolog. For the scenario of symbolic term representations *(A)*, frequently a fine-grained reification of the internal object structure (i.e., a white box representation) is desired. For example, JTransformer (Kniesel et al. 2007) allows to reason over the structure of terms reifying objects modelling a Java abstract syntax tree. However, when inspecting or reasoning about the object's structure on the Prolog side is not required, having an opaque reference (i.e., a black box representation) to the corresponding Java object is preferable (e.g., an opaque reference to a GUI component on the Java side). In those cases, an automatic mechanism to generate (opaque) term representations of Java objects is desirable. Otherwise, the programmer would have the burden to ensure the uniqueness of an arbitrarily chosen black box representation.

When the Prolog engine is embedded/implemented inside a JVM a more direct kind of

[1] `https://github.com/java-prolog-connectivity`

reference to Java objects can be established *(B)*. In the simplest case, the object reference can be considered (and unified) as a special constant term. In spite of the more direct mapping (no automated mapping to generate the reference is needed; the term wraps the object 'as is'), conceptually this case is equivalent to mapping the object reference to an opaque (black box) term representation.

However, we may want to combine the best of both worlds and have direct references to the actual Java objects, while still allowing Prolog programs to reason over the internal structure of such objects. Approaches such as SOUL (Roover et al. 2011) and EKEKO (De Roover and Stevens 2014) have achieved this through the mechanism of *open unification* (Brichau et al. 2007; Wuyts 2001). This approach consists in allowing the programmer to customise not the term representation of an object, but rather its unification mechanism. In a nutshell, the unification mechanism is opened up so that Java objects are not regarded as constants but can be unified with structured logic terms of the right form. Although conceptually different, from several practical aspects this is similar to mapping the object reference to a white box term representation, with the important notable difference that an actual reference to the original Java object is kept.

2.3 Object Identity Preservation

For logic engines running in the JVM *(D)* there is no problem with preserving the identity of the Java objects referred to on the logic side. Object references are preserved automatically since the term wraps the object 'as is'. For engines not embedded in the JVM, a programmer needs to decide if an object reified as a term should preserve its identity when the term is translated back to a Java object *(C)*. In many situations, it is not relevant or important to preserve such identity (e.g., instances of `String`) and a different reference, considered equivalent to the original object (e.g., by means of the `equals` method), is an acceptable outcome. However, in certain cases, keeping track of the original reference is required to guarantee the expected behaviour of the program (e.g., if the reference points to a GUI component). Furthermore, passing around terms symbolically representing object references is often more efficient than marshalling and unmarshalling large Java objects. Note that the need for preserving the original object identity is orthogonal to the required opacity of the representation. I.e., independently if the reference should be preserved or not, the programmer should still be able to decide on the best representation of the object on the Prolog side.

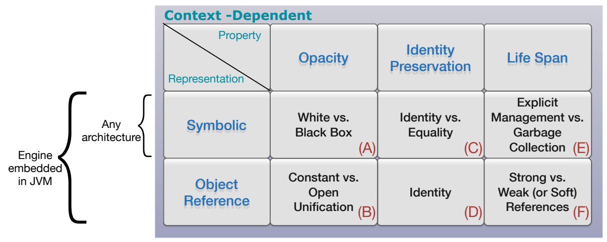

Fig. 1. Reference Management Dimensions

2.4 Reference Life Span

A fourth important dimension is the life span of Prolog references to Java objects. For a symbolic term representation, a programmer should decide on a mechanism for delimiting the life span of a mapping between a Java reference and a Prolog term *(E)*. This mechanism can be explicit (e.g., by means of an API allowing to request to 'forget' a mapping) or rely on the garbage collection mechanisms of the JVM. An explicit mechanism enables a fine-grained control over the life span of a reference. For example, a symbolic term representation of an object that is not explicitly referenced in a program (i.e., normally to be scheduled for garbage collection) can still remain valid until explicitly discarded. Alternatively, a reference life span may be automatically delimited by the Java garbage-collection mechanism (e.g., a reference to the application main window).

For an object reference representation *(F)*, often the programmer may want to keep the reference alive as long as it is present in the Prolog database (i.e., a strong reference). However, in certain scenarios a Java reference stored in Prolog should not prevent it from being garbage collected (e.g., the reference points to a disposed GUI component). In that case, the reference should be invalidated when it is reclaimed by the garbage collector.

In addition, a programmer may want to define customisable cleaning tasks to be automatically executed when a reference is garbage collected. For example, clauses containing dead references may be automatically retracted from the Prolog database in order to avoid unexpected behaviours (e.g., null pointer exceptions). Furthermore, references that may be reclaimed by the garbage collector should be classified according to the natural Java (garbage-collected) reference types: *Weak* for eagerly collected references (i.e., discarded at the next garbage collection cycle) and *Soft* for references not aggressively reclaimed (i.e., only collected when the memory is tight).[2]

2.5 Scope of the Inter-Language Conversion Policies

In addition to deciding which of the above choices are most appropriate for a particular programming task at hand, we claim that it would be useful for a programmer to be able to choose different policies in different parts of the program. To achieve that, the programmer needs a simple mechanism for scoping and encapsulating the best reference handling policy for certain objects. In addition to greater flexibility, this facilitates performance tuning and testing (e.g., generating mocking representations of references).

In the next section, we will introduce the architecture of a library that supports a customisable management of all these dimensions.

3 Architecture

JPC is an integration library supporting the development of hybrid Java-Prolog programs. It provides different level of abstractions that simplify the implementation of common inter-operability tasks. To set the ground for a discussion of the JPC features related to Java reference management in Prolog, this section first overviews its main components (figure 2).

[2] `http://docs.oracle.com/javase/7/docs/api/java/lang/ref/Reference.html`

3.1 Prolog VM Abstraction

Several integration libraries rely on the notion of a Prolog engine as a convenient abstraction for interacting with a Prolog virtual machine from Java (Tarau 2012; Tarau 2004; Rho et al. 2004; Calejo 2004). In JPC, a programmer interacts with a Prolog engine abstraction that communicates with concrete Prolog engines using drivers. With portability in mind, when modelling such an abstract Prolog engine we tried to find a compromise between (1) offering convenient features facilitating the interaction from Java programs and (2) not assuming a specific implementation architecture of the underlying Prolog engine. Our Prolog engine abstraction provides a Java programmer with a general purpose API for interacting with Prolog. However, as illustrated in section 4, JPC also supplies a higher level API that simplifies such tasks (e.g., inter-language conversions). JPC defines a set of classes reifying Prolog data types, inspired by similar JPL library classes:

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.
Var : A Prolog variable.
JRef : A Java reference term. A special kind of term wrapping a Java reference.
Query : The reification of a Prolog query.

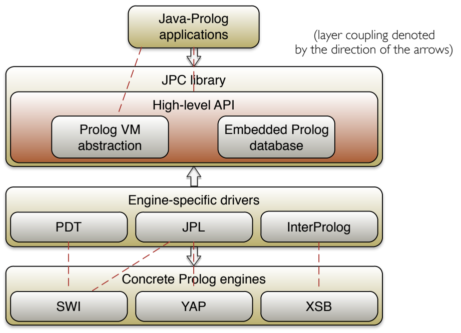

Fig. 2. The JPC architecture

3.2 Embedded Prolog Database

JPC uses an embedded Prolog database running on the JVM and supporting the storage of Java object references in addition to standard Prolog terms. Several JPC interoperability features rely on this component, which maintains mappings between Prolog terms and arbitrary Java objects (represented as `JRef` terms). Currently, this database is not intended to be used directly by the programmer as it lacks many of the features of a full Prolog system.

4 Reference Management with JPC

This section describes JPC's support for the different dimensions related to the management of Java references in Prolog (figure 1).

4.1 Symbolic Representation

To illustrate the properties of symbolic references (identified by the first row of figure 1), we start by defining a `Person` class (listing 1) declaring `name` as its only instance variable.

```java
public  class Person implements Serializable {
    private final String name;
    public Person(String name) {this.name = name;}
    ...
    @Override
    public boolean equals(Object obj) {
        ... return ((Person)obj).name.equals(name); //simplified implementation
    }
}
```

Listing 1. The `Person` class

The `PersonConverter` class (listing 2) defines how instances of class `Person` are translated to a Prolog (compound) term (lines 4–6) and back (lines 8–10). According to our classification in section 2.2, the term reification of a person, according to this converter, corresponds to a white box representation since it exposes its internal data.

```java
1  public class PersonConverter implements FromTermConverter<Compound, Person>, ToTermConverter<Person,
       Compound> {
2      public static final String PERSON_FUNCTOR_NAME = "person";
3
4      @Override public Compound toTerm(Person person, Class<Compound> termClass, Jpc context) {
5          return new Compound(PERSON_FUNCTOR_NAME, asList(new Atom(person.getName())));
6      }
7
8      @Override public Person fromTerm(Compound personTerm, Type targetType, Jpc context) {
9          return new Person(((Atom)((Compound)personTerm).arg(1)).getName());
10     }
11 }
```

Listing 2. The `PersonConverter` class

An example of a white box term representation of a Java object, without object identity preservation, is shown in listing 3 (the first three lines are common to most examples so we will not repeat them). A central artefact in our approach is a *conversion context*, instantiated in line 4 using a builder class and configured with the `PersonConverter` converter. We use this context for obtaining the conversion of a person in line 5 (`person(mary)`). Next, we assert the fact `student(person(mary))` (line 6). A `student(A)` goal is instantiated in line 7 passing the context defined before. A person is queried in line 8 using a deterministic query. The `selectObject()` method adapts each solution to the query as an object whose term reification is given as a string. This adaptation corresponds to the conversion as a Java object of the term that has been bound to the `Person` variable in the solution. We verify in line 9 that the queried person is equal to the original person. However, their identities are different (line 10).

```java
1 final String STUDENT_FUNCTOR_NAME = "student";
2 PrologEngine prologEngine = getPrologEngine();
3 Person mary = new Person("Mary");
4 Jpc ctx = JpcBuilder.create().register(new PersonConverter()).build();
5 Term personTerm = ctx.toTerm(mary);
6 prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(personTerm)));
```

```
7  Query query = prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME, asList(new Var("Person"))), ctx);
8  Person queriedPerson = query.<Person>selectObject("Person").oneSolutionOrThrow();
9  assertEquals(mary, queriedPerson);
10 assertFalse(mary == queriedPerson);
```

Listing 3. White Box without Identity Preservation

Listing 4 illustrates the mapping of a reference to a term representation (line 2) in the scope of a context. The `newJTerm()` method associates a person reference (first argument) to an arbitrary (compound) term representation (second argument). In this example, the term corresponds to the term conversion of the reference according to a given conversion context (obtained by the `toTerm()` method of the context instance). We verify that this time the queried person corresponds to the original person reference in line 6.

```
1 Jpc ctx = JpcBuilder.create().register(new PersonConverter()).build();
2 Term personTerm = ctx.newJTerm(person, ctx.<Compound>toTerm(mary));
3 prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(personTerm)));
4 Query query = prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME, asList(new Var("Person"))), ctx);
5 Person queriedPerson = query.<Person>selectObject("Person").oneSolutionOrThrow();
6 assertTrue(mary == queriedPerson);
```

Listing 4. White Box and Identity Preservation

An example of a black box representation is shown in listing 5. Here, we assert a term of the form `student(serialisation)`, where the compound argument corresponds to the term representation of the serialisation of a `Person` instance. No converter is passed to the query in line 2. This is because the default conversion context (employed by the query if no context is explicitly passed) includes a converter able to deserialize a Java object from the term representation of its serialisation. Finally, we verify that our queried person is equal to the original person (line 4) although having different identities (line 5).

Although in the context of this example we have presented this term reification as a black box representation, note that in other contexts this may be considered as a white box. This would be the case if the Prolog side is intended to interpret such representation (e.g., if it reasons over the serialised bytes of the object (Calejo 2004)).

```
1 prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(SerializedTerm.serialize(mary))));
2 Query query = prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME, asList(new Var("Person"))));
3 Person queriedPerson = query.<Person>selectObject("Person").oneSolutionOrThrow();
4 assertEquals(mary, queriedPerson);
5 assertFalse(mary == queriedPerson);
```

Listing 5. Black Box without Identity Preservation

A programmer can also associate an automatically generated term to a reference. An example is given in listing 6. This time we invoke the method `newJTerm()` passing as only argument the reference to reify as a term (line 2). A (black box) term representation is generated behind the curtains. Our library guarantees that such generated term representations are identical for the same object even across different contexts.

```
1 Jpc ctx = JpcBuilder.create().build();
2 Term personTerm = ctx.newJTerm(mary);
3 prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(personTerm)));
4 Query query = prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME, asList(new Var("Person"))), ctx);
```

```
5 Person queriedPerson = query.<Person>selectObject("Person").oneSolutionOrThrow();
6 assertTrue(mary == queriedPerson);
```

Listing 6. Black Box and Identity Preservation

As discussed in section 2.4, a programmer should also be able to control the life span of term–reference mappings. Listing 7 shows an example. As in listing 4, we use the `newJTerm()` method (line 2) to associate a reference to its (context dependent) term reification. But afterwards we delete this association using the `forgetJTerm()` method (line 5). Thus, although the queried person is equal to the original person (line 7) since the term is translated according to the conversion context (line 1), they do not have the same identity (line 8) as the association between the term and the original reference was eliminated.

```
1 Jpc ctx = JpcBuilder.create().register(new PersonConverter()).build();
2 Term personTerm = ctx.newJTerm(person, ctx.<Compound>toTerm(mary));
3 prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(personTerm)));
4 assertTrue(mary == prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME, asList(new Var("Person"))),
      ctx).selectObject("Person").oneSolutionOrThrow());
5 ctx.forgetJTerm((Compound)personTerm);
6 Person queriedPerson = prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME, asList(new
      Var("Person"))), ctx).<Person>selectObject("Person").oneSolutionOrThrow();
7 assertEquals(mary, queriedPerson);
8 assertFalse(mary == queriedPerson);
```

Listing 7. Explicit Management of Associations Life Span

A programmer can also rely on the natural garbage collection mechanisms of Java for delimiting the life span of an association. This is shown in listing 8. The `newWeakJTerm()` method (line 2) is equivalent to the `newJTerm()` method discussed in previous examples. However, in this case the association between a term and a reference persists as long as the reference is not reclaimed in the next garbage collection cycle. To prove it, we assign `null` to the only variable keeping a reference to the person (line 4) and give a hint to the garbage collector to start a cycle (line 5). Note that the query is not instantiated with a conversion context (line 7). Therefore, an exception is raised when we try to convert the term (bound to the variable `Person`) to an object, since no converter is found and no reference is associated to such term. Our framework also provides the `newSoftJTerm()` method with similar semantics than `newWeakJTerm()`, with the only difference that an association between a term and a reference may persist some time after a garbage collection cycle, and will be deleted only if the memory gets tight.

```
1 Jpc ctx = JpcBuilder.create().register(new PersonConverter()).build();
2 Term personTerm = ctx.newWeakJTerm(mary, ctx.<Compound>toTerm(mary));
3 prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(personTerm)));
4 mary = null;
5 System.gc();
6 try {
7    prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME, asList(new
         Var("Person")))).<Person>selectObject("Person").oneSolutionOrThrow();
8    fail();
9 } catch(ConversionException e) {}
```

Listing 8. Garbage Collection Management of Associations Life Span

4.2 Object Reference Representation

This section focuses on the properties of object references (identified by the second row of figure 1). Although our library currently only has drivers for non-embedded Prolog engines, as a proof of concept we implement the examples in this section using the JPC embedded Prolog database described in section 3.1. With the exception of open unification, all the other properties are supported by our implementation.

We start with an example of constant unification of references. As mentioned in section 3.1, a `JRef` term is a special JPC term wrapping an object reference. In our current version, they are unified as constants (i.e., a unification between `JRef` terms succeeds if their referred objects are equal). This is illustrated in listing 9. In line 1 we assert that *mary* (wrapped in a `JRef` term) is a student. In line 2 we query if a different person object with the same name is a student, which succeeds.

```
1 prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(JRef.jRef(mary))));
2 assertTrue(prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME, asList(JRef.jRef(new
      Person("mary"))))).hasSolution());
3 Solution solution = prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME, asList(new
      Var("X")))).oneSolutionOrThrow();
4 JRef<Person> jRef = (JRef<Person>) solution.get("X");
5 assertTrue(mary == jRef.getReferent());
```

Listing 9. Constant Unification of `JRef` terms

Thanks to our embedded Prolog database, the identity of a reference is trivially preserved. To illustrate this, we execute a deterministic query (line 3) with goal `student(X)`. The query solution, an instance of the `Solution` class, exposes a map interface where keys correspond to unbound variables in the goal and the values to the bound terms in the solution. We verify that the obtained referent has the same identity as *mary* in line 5.

Listing 10 shows how to create `JRef` instances that may be garbage collected. We first create two objects equal to *mary* and assert them, using different kind of references for each object: *strong* (line 3), *weak* (line 4) and *soft* (line 5). When we query for students unifying with *mary* (line 6) using a strong reference, we get three results instead of one. This is because the unification semantics of `JRef` terms evaluates the referents, not the actual `JRef` term wrapper. Listing 10 also shows an example of an invalidated reference. We assign to `null` the variable `person2` (line 7) and give a hint to the garbage collector to execute a cycle (line 8). Since the referent of the `JRef` term asserted in line 4 has been invalidated, the number of students unifying with *mary* is now only 2 (line 9). Note that weak (and soft) references should be used with care: they may require non-monotonic reasoning as the referent of a `JRef` term may be invalidated during the query execution.

```
1 Person person2 = new Person("Mary");
2 Person person3 = new Person("Mary");
3 prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(JRef.jRef(mary))));
4 prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(JRef.weakJRef(person2))));
5 prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(JRef.softJRef(person3))));
6 assertEquals(3, prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME,
      asList(JRef.jRef(mary)))).allSolutions().size());
7 person2 = null;
8 System.gc();
9 assertEquals(2, prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME,
      asList(JRef.jRef(mary)))).allSolutions().size());
```

Listing 10. Life Span of `JRef` terms

The previous example motivates the need of a cleaning mechanism. Listing 11 illustrates such mechanism using user-defined cleaning tasks. To keep our example simple, this cleaning task retracts all the asserted students (lines 1–5) when a reference is invalidated (a more sophisticated example would retract only the invalidated reference). Our cleaning task is associated with a weak reference in line 6. In line 9 we verify that no students are in the database after the reference to *mary* has been invalidated (lines 7–8).

```
1 Runnable cleaningTask = new Runnable() {
2     @Override public void run() {
3         prologEngine.retractAll(new Compound(STUDENT_FUNCTOR_NAME, asList(Var.ANONYMOUS_VAR)));
4     }
5 };
6 prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(JRef.weakJRef(mary, cleaningTask))));
7 mary = null;
8 System.gc();
9 assertFalse(prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME,
      asList(Var.ANONYMOUS_VAR))).hasSolution());
```

Listing 11. Cleaning Tasks

5 Related Work

Most related work has already been overviewed in sections 2 and 3.1 so we do not repeat it here. InterProlog inspired the serialisation mechanism illustrated in listing 5. It provides a more structured representation of a serialised object on the Prolog side using a definite clause grammar (currently we only support representing serialised bytes as an atom using a raw base-64 encoding (Josefsson 2006)). InterProlog has limited support, however, for customising the reification as a term of arbitrary Java objects (even not serialisable ones) as in our approach. Concerning our mechanisms for custom two-way conversions between inter-language artefacts, this was mainly 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, certain aspects of our library can be regarded as a re-implementation of Gson for the domain of Java-Prolog artefact conversions.

6 Conclusions and Future Work

This work discusses different dimensions that should be taken into consideration when dealing with Java references in Prolog programs. These dimensions have been extracted from many sources, including our own experience, a study of existing approaches, and even existing solutions in other domains. At the moment, JPC does not implement a mechanism for interacting with Java from the Prolog side. In line with our portability goal, we plan to implement our Prolog side API using Logtalk (Moura 2003), a portable object-oriented layer for Prolog. As in the current Java side API, we expect to prototype a first version by reusing existing bridge libraries. We will also continue improving our embedded Prolog database so that it can be released as a stand-alone embedded Prolog engine. We hope that our work will benefit not only implementors of Java-Prolog integration libraries, but also integrators of similar object-oriented and logic languages.

Acknowledgments

This work is partially funded by the ERDF (European Regional Development Fund) through the COMPETE Programme and by FCT (Portuguese Foundation for Science and Technology) within project SIBILA (NORTE-07-0124-FEDER-000059).

References

Brichau, J., De Roover, C., and Mens, K. 2007. Open Unification for Program Query Languages. In *Proceedings of the XXVI International Conference of the Chilean Computer Science Society (SCCC 2007)*.

Bueno, F., Carro, M., Hermenegildo, M., Haemmerlé, R., López-García, P., Mera, E., , Morales, J., and Puebla-(Eds.), G. 2011. The Ciao System. Ref. Manual (v1.14). Tech. rep. July. Available at http://ciao-lang.org.

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

Carlsson, M. et al. 1995. *SICStus Prolog User's Manual*, Release 3 ed. Swedish Institute of Computer Science. ISBN 91-630-3648-7.

Castro, S., Mens, K., and Moura, P. 2013. JPC: A Library for Modularising Inter-Language Conversion Concerns between Java and Prolog. In *International Workshop on Advanced Software Development Tools and Techniques (WASDeTT). Co-located with the European Conference in Object-Oriented Programming (ECOOP)*.

De Roover, C. and Stevens, R. 2014. Building Development Tools Interactively using the Ekeko Meta-Programming Library. In *Proceedings of the IEEE CSMR-WCRE 2014 Software Evolution Week (CSMR-WCRE14)*.

Denti, E., Omicini, A., and Ricci, A. 2005. Multi-paradigm Java–Prolog Integration in tuProlog. *Science of Computer Programming 57,* 2, 217 – 250.

D'Hondt, M. 2004. Hybrid Aspects for Integrating Rule-Based Knowledge and Object-Oriented Functionality. Ph.D. thesis, Vrije Universiteit Brussel.

Google Inc. 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/.

Gybels, K. 2003. SOUL and Smalltalk — Just Married: Evolution of the Interaction Between a Logic and an Object-Oriented Language Towards Symbiosis. In *Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages*.

Hermenegildo, M. V., Bueno, F., Carro, M., López, P., Mera, E., Morales, J., and Puebla, G. 2012. An Overview of Ciao and its Design Philosophy. *TPLP 12,* 1–2, 219–252.

Josefsson, S. 2006. The Base16, Base32, and Base64 Data Encodings. Internet RFC 4648.

Kniesel, G., Hannemann, J., and Rho, T. 2007. A Comparison of Logic-Based Infrastructures for Concern Detection and Extraction. In *Proceedings of the 3rd workshop on Linking aspect technology and evolution*. LATE '07. ACM, New York, NY, USA.

Mernik, M., Heering, J., and Sloane, A. M. 2005. When and How to Develop Domain-specific Languages. *ACM Comput. Surv. 37,* 4 (Dec.), 316–344.

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

Omicini, A. and Natali, A. 1994. Object-Oriented Computations in Logic Programming. In *Object-Oriented Programming*, M. Tokoro and R. Pareschi, Eds. Lecture Notes in Computer Science, vol. 821. Springer Berlin Heidelberg, 194–212.

Rho, T., Degener, L., Günter Kniesel, Frank Mühlschlegel, Eva Stöwe, Noth, F., Becker, A., and Alyiev, I. 2004. The Prolog Development Tool — A Prolog IDE for Eclipse. http://sewiki.iai.uni-bonn.de/research/pdt/.

Roover, C. D., Noguera, C., Kellens, A., and Jonckers, V. 2011. The SOUL Tool Suite for Querying Programs in Symbiosis with Eclipse. In *PPPJ*. 71–80.

Singleton, P., Dushin, F., and Wielemaker, J. 2004. JPL 3.0: A Bidirectional Interface Between Prolog and Java. `http://www.swi-prolog.org/packages/jpl/java_api/`.

Tarau, P. 2004. Agent Oriented Logic Programming Constructs in Jinni 2004. In *ICLP*, B. Demoen and V. Lifschitz, Eds. Lecture Notes in Computer Science, vol. 3132. Springer, 477–478.

Tarau, P. 2011. Integrated Symbol Table, Engine and Heap Memory Management in Multi-engine Prolog. In *Proceedings of the 10th International Symposium on Memory Management*. ACM, 129–138.

Tarau, P. 2012. The BinProlog Experience: Architecture and Implementation Choices for Continuation Passing Prolog and First-Class Logic Engines. *Theory Pract. Log. Program. 12,* 1-2 (Jan.), 97–126.

Wuyts, R. 2001. A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation. Ph.D. thesis, Vrije Universiteit Brussel.

Wuyts, R. and Ducasse, S. 2001. Symbiotic Reflection between an Object-Oriented and a Logic Programming Language. *International Workshop on MultiParadigm Programming with Object-Oriented Languages*.