JPC: A Library for Categorising and Applying Inter-Language Conversions Between Java and Prolog

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

[a]*Université catholique de Louvain, ICTEAM, 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 combine the strengths of both languages. Most of these approaches provide limited support to allow programmers to customise how Prolog artefacts should be reified in the Java world, or how to reason about Java objects on the Prolog side. This is an error-prone task since often a considerable amount of mappings must be developed and organised. Furthermore, appropriate mappings may depend on the particular context in which a conversion is accomplished. Although some libraries alleviate this problem by providing higher-level abstractions to deal with the complexity of custom conversions between artefacts of the two languages, these libraries themselves are difficult to implement and evolve. We claim that this is caused by their lack of appropriate underlying building blocks for encapsulating, categorising and applying Java-Prolog conversion routines. We therefore introduce a new library, *JPC*, serving as a development tool for both programmers willing to categorise 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 applications to call Prolog predicates and Prolog applications to access Java's extensive set of libraries. In Castro et al. (2012, 2013) we presented a linguistic symbiosis approach to support the implementation of Java methods in terms of logic predicates. We focused on attaining a transparent and semi-automatic integration of Prolog within the Java language by providing a simple mechanism for inferring the appropriate mappings between Java and

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

Prolog artefacts. In addition to predefined mappings for typical scenarios, a customisation mechanism, based on the use of annotations, was provided.

Although we accomplished our objectives, our framework was difficult to extend to customisation mechanisms not based on annotations. In addition, our implementation was hard to maintain and evolve, mainly because methods dealing with artefact conversions (i.e., the mechanism defining how an artefact in one language should be reified into the other) had to analyse a considerable amount of conditions to infer the appropriate conversion. This resulted in a high cyclomatic complexity in the implementation, aggravated by the fact that, as discussed in section 3, selecting the most appropriate conversion often depends on a specific usage context. Furthermore, conversion algorithms were often tangled with the main concern of the program.

In hindsight, we understood these problems resulted from the lack of a structured mechanism for encapsulating, categorising and applying context-dependent conversions between Java objects and Prolog terms. The present work builds upon the difficulties we faced and the lessons we learned. Its outcome is a framework, inspired to a certain extent on the Gson library (Google Inc. (2012)), for dealing with context-dependent conversion concerns between Java-Prolog inter-language artefacts.

This paper is structured as follows. In section 2 we discuss the intrinsic complexity of Java-Prolog interaction, revisiting an example from previous work. Section 3 provides a conceptual analysis of the interaction problem, together with the outline of a solution. Section 4 describes the architecture and main components of our JPC[1] library. Section 5 presents, using small examples, the core of our technique for categorising and applying context-dependent conversions between Java objects and Prolog terms. Section 6 discusses the limitations of our approach. Section 7 overviews approaches that inspired our technique and related work. Section 8 presents our conclusions from a tool builder perspective and outlines our future work plans.

2. Java–Prolog Interaction Complexity

In Castro et al. (2012, 2013), 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 explain 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. It allowed to answer queries such as which stations are transitively connected, or which are the stations traversed by a particular line.

To facilitate the mapping between Java and Prolog artefacts, we used Logtalk (Moura (2003)), an object-oriented logic programming language implemented using Prolog as a backend compiler. Logtalk is highly portable, allowing us to work with most Prolog implementations by using Logtalk as a intermediate layer. An outline of the Logtalk `station(_Name)` parametric object (Moura (2011)) is shown in listing 1. The implementation of the public

[1] *JPC* stands from Java-Prolog-Connectivity.

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 (not shown here for brevity).

```
1  :- object(station(_Name)).
2
3      :- public(connected/2).
4      connected(Station, Line) :-
5          self(Self),
6          metro::connected(Self, Station, Line).
7
8      % other methods
9
10 :- end_object.
```

Listing 1: The `station(Name)` parametric object in Logtalk

```
public class Station {
    String name;
    ...
    // mapping an instance of Station to a logic Term
    public Term asTerm() {
        return new Compound("station", new Term[] {new Atom(name)});
    }
    // mapping a logic Term to an instance of Station
    public static Station create(Term stationTerm) {
        String lineName = ((Compound)stationTerm).arg(1).name();
        return new Station(lineName);
    }
    // mapping a Java method to a Logtalk method
    public Station connected(Line line) {
        Station connectedStation = null;
        String stationVarName = "ConnectedStation";
        Term[] arguments = new Term[]{new Variable(stationVarName),  line.asTerm()};
        Term message = new Compound("connected", arguments);
        Term objectMessage = new Compound("::", new Term[] {asTerm(), message});
        Query query = new Query(objectMessage);
        Hashtable<String, Term> solution = query.oneSolution();
        if(solution != null) {
            Term connectedStationTerm = solution.get(stationVarName);
            connectedStation = create(connectedStationTerm);
        }
        return connectedStation;
    }
    // other methods mapped to logic routines
    ...
}
```

Listing 2: A Java class interacting with Prolog by means of JPL

Listing 2 shows an outline of the corresponding `Station` Java 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 Logtalk as the message `station(station_name)::connected(ConnectedStation, line(line_name))`.

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.

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).

Larger applications often require writing a large number of (possibly context-dependent) conversion routines. This is an error-prone activity that often produces ad-hoc conversion code tangled with other aspects of the code. In the next section we discuss, at a conceptual level, a strategy for alleviating this problem.

3. Analysis

The small example in the previous section illustrated the complexity of writing programs that require conversions between artefacts of different languages. At first glance, adding conversion routines directly into the classes whose instances must be converted may be a straightforward solution. This also has the advantage that such routines are polymorphic (i.e., the conversion to accomplish depends on the actual receiver of the conversion message). However, tangling the main concern of these classes with inter-language conversion routines is a violation of the separation of concerns principle (Dijkstra (1997); Parnas (1972)) and hinders the overall maintenance and evolution of a system. Furthermore, this approach is only suitable if the programmer has access to, and can modify, the source code of such classes. Also, it does not provide an appropriate solution for dealing with inter-language conversions that depend on a particular usage context.

There are techniques that alleviate the separation of concerns problem without relying on having access to the source code of the class. A simple approach may be moving the conversion routines into a separate class. While this might work fine for reasonably small systems, a disadvantage is that a programmer cannot execute polymorphic conversion methods, because such methods are no longer part of the class hierarchy of the object to be converted. Therefore, an explicit binding between an object and its conversion routine, located in a different class, should be established (e.g., by means of the double dispatch pattern (Gamma et al. (1995))). Although an ad-hoc infrastructure could be developed to facilitate this mapping (e.g., based on dedicated design patterns or reflection), such infrastructure may be not that trivial to implement and maintain in larger systems.

Aspect-Oriented Programming (Kiczales (1996)) provides an alternative that allows to encapsulate conversion routines into separate modules (aspects) and weave them into appropriate classes using a quantification mechanism (e.g., by means of inter-type declarations (Masuhara and Kiczales (2003))). This approach achieves a better separation of concerns, but still does not provide a structured solution to the problem of context-dependent inter-language conversions. With this alternative, that problem is just deferred to the aspect. We argue that a suitable solution should provide context-specific constructs for:

- Defining inter-language conversion functions.
- Inferring the best target type of a conversion operation.
- (Optionally) instantiating conversion target types.
- Categorising conversion artefacts.
- Applying conversions.

In addition, an ideal solution should not rely on having access to the source code of classes whose instances participate in conversion operations. In the rest of this section we discuss, at an abstract level, these desirable properties.

3.1. Defining Inter-Language Conversion Functions

A programmer should be able to define conversion functions that act as gateways between representations of an object in another language. In Java, a conversion function can be specified as a method receiving, among other parameters, the source object to convert and returning the (reified) representation of this object in the target conversion language. Listing 3 shows a `Converter` interface declaring such a method.

```java
public interface Converter<SourceType,TargetType> {
    public TargetType apply(SourceType source, Type concreteTargetType, ...);
}
```

Listing 3: The `Converter` interface

Part of the definition of a conversion function involves the quantification of its domain and range. Such quantification should be accomplished taking into consideration the natural quantification mechanisms of the two languages taking part in the conversion.

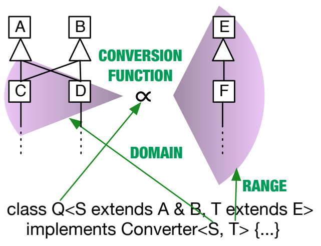

A
B
E
CONVERSION FUNCTION
∝
C
D
F
DOMAIN
RANGE
class Q<S extends A & B, T extends E> implements Converter<S, T> {...}

Figure 1: Quantifying over variable types

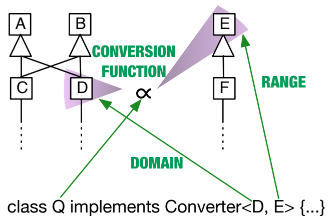


Figure 2: Quantifying over concrete types

3.1.1. Type-Quantified Converters

Types are a natural Java mechanism for quantifying over objects. Therefore, the domain and range of a converter function can be specified by means of types parameterising such function. For example, the type parameters of the `Converter` interface define the domain of the function (`SourceType`) and its range (`TargetType`). Note that these type parameters may reference a concrete type (parameterised or raw) or a variable type with (explicit or implicit) bounds quantifying over a set of types. For example, at the bottom of figure 1 we can see a converter defining its domain and range by means of type variables. Its domain are the classes `C` and `D`, since they inherit from both `A` and `B` (which are the upper bounds of the type variable `S`). Its range is `E` and `F`, according to the upper bounds of the type variable `T`.

As illustrated by figure 2, a converter domain and range can also refer to concrete types. In this case, the converter declares as its domain the class `D` and as its range the class `E`.

3.1.2. Term-Quantified Converters

Conversion functions can define Prolog terms as their domain (i.e., term to object conversions) or range (i.e., object to term conversions). Although terms can also be quantified using (reified Java) types, this approach is not ideal since the number of types in Prolog is quite limited (e.g., specifying the domain of a conversion function as instances of compound terms is, for most cases, too broad). An alternative is to declare what term instances belong to a particular conversion function domain or range, by means of a more general term that subsumes such instances.

More formally, Sterling and Shapiro (1994) define a term A as being an instance of term B if there exists a variable substitution θ such that θ applied to B (denoted as $B\theta$) gives A, so $A = B\theta$. A *substitution* is defined as "*a finite set (possibly empty) of pairs of the form $X_i = t_i$, where X_i is a variable and t_i is a term, and $X_i \neq X_j$ for every $i \neq j$, and X_i does not occur in t_j, for any i and j*".

Based on these definitions, we could, for example, define the domain of a certain converter as all the instances of the term `hello(X)`, where `X` is an unbound Prolog variable. The terms `hello(mary)` and `hello(peter)` will be included in this domain, while the terms `chello(mary)` or `hello(mary, peter)` will not.

3.2. Conversion-Type Inference

As discussed in section 3.1.1, the range of a converter may comprehend more than one type (e.g., if the range is given by a variable type as illustrated by figure 1). In those cases, a programmer should be able to select from such range a specific target type for a given conversion operation (hence the `concreteTargetType` parameter in the converter method in listing 3). It may also be the case that the converter declares a single generic type as its range, but that such type makes use of variable types (or wildcards) for some of its type parameters (e.g., `List<?>`, where the unbounded wildcard `?` stands for the family of all types) or it is not parameterised (e.g., the raw type `List`). In this case, the programmer may specify a concrete conversion target type (e.g., `List<String>`). Depending on the converter implementation, type parameters of the conversion target type may influence the result of the conversion. For example, a converter declaring as its range the type `List<?>` may convert a Prolog list to the parameterised type `List<String>` in a different way than it would convert the same term to the parameterised type `List<Boolean>`.

In case the target type of a conversion is not specified, or is overly general, it should be possible to infer it by inspecting the source object properties. For example, a Prolog list of compounds with functor `:/2` (e.g., `[a:x,b:y]`) may be typed as a Java `Map`. Furthermore, a programmer should be able to customise this (inter-language) type inference mechanism by means of registering new type inference functions. Such functions can be specified as a method receiving an object representing an artefact in one of the two languages (e.g., Java or Prolog), and returning the best type for the corresponding object or term in the other language. Listing 4 shows a `TypeSolver` interface declaring such a method. Similar to converter functions, the domain of type inference functions is quantified either by types (e.g., the type parameter of the `TypeSolver` interface) or by terms. The range of these functions is implicitly specified as the set of foreign language types (e.g., for Java to Prolog conversions, the range would be the set of classes reifying Prolog types). Concrete examples are presented in section 4.2.2.

```java
public interface TypeSolver<T> {
    public Type getType(T object);
}
```

Listing 4: The `TypeSolver` interface

3.3. Factories

Although a converter has access to the (explicit or inferred) target type of a conversion, it may not know how to instantiate this type or how to obtain an existing reference. A simple ad-hoc solution may be to initialise certain converters with a factory (or provider). Then, the converter can delegate to such factory the instantiation of the object to return. However, this is a repetitive problem that may spawn a certain amount of boilerplate code, since converters could become polluted with factory-related code.

A better alternative consists in allowing the programmer to register factory functions thus providing a transparent mechanism for instantiating required objects using registered factories (similar approaches are provided by dependency injection frameworks such as Google's

Guice library (Vanbrabant (2008))). Factory functions can be specified as a method receiving a type and returning an instantiation of this type. Listing 5 shows a `Factory` interface declaring such a method. Similar to converter functions, the factory domain is specified by means of a type parameter of the `Factory` interface. Also like converter functions, a factory domain can comprehend more than one type. Therefore, a programmer should be able to specify the concrete type to instantiate (hence the `concreteFactoryType` parameter).

The range of a factory is implicitly specified as the possible set of object instances of the types declared as its domain. We discuss in section 4.2.3 our solution for the management of user-defined factories.

```java
public interface Factory<T> {
    public <T> T instantiate(Type concreteFactoryType);
}
```

Listing 5: The `Factory` interface

3.4. Categorising Conversion Artefacts

The conversion artefacts described before are functions (converter functions, type inference functions, and factory functions) that should be categorised according to their domains and ranges. As discussed in section 3.1.1, in certain cases such domains and ranges may be inferred from the declaration of the function. In other cases, as shown in section 3.1.2, they are not implicit in the function declaration and the programmer should explicitly provide them. JPC's categorisation approach is presented in detail in sections 4.3 and 4.4.

3.5. Applying Conversions

From a high level point of view, the process of applying a conversion consists of the orchestration of the distinct conversion artefacts presented in this section. This process has as input an object to convert, optionally a hint guiding the conversion (e.g., the expected result type) and having as outcome the converted object in a (reified) foreign language representation (figure 3).

Given that conversion artefacts (e.g., converter functions) may overlap their domains or ranges, the implementation of this process should take into consideration delegation and conflict resolution mechanisms. A high level view of our approach for the selection of conversions is shown in section 4.4.5.

3.6. Context as Converter Discriminator

In Castro et al. (2013) we showed an example of how the best conversion of a Java object to a Prolog term may depend on the objective of such conversion. For example, a Java `List` is better translated to the Logtalk object `list` if the intention is to invoke a Logtalk method on that list object. Conversely, if the Java list would be used as the argument of a predicate, its best term representation is often a Prolog list.

This is not the only case where the usage context may determine the best conversion. As mentioned before in this section, different converter functions (and other conversion

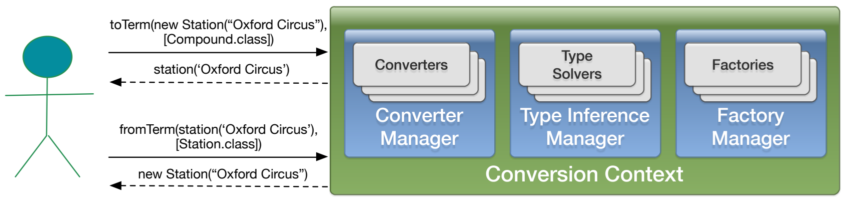

toTerm(new Station("Oxford Circus"), [Compound.class])
station('Oxford Circus')
fromTerm(station('Oxford Circus'), [Station.class])
new Station("Oxford Circus")
Converters
Converter Manager
Type Solvers
Type Inference Manager
Factories
Factory Manager
Conversion Context

Figure 3: The conversion context

artefacts) may overlap their domains or ranges. Unfortunately, for those situations it is hard or impossible to conceive a conflict resolution policy that would best suit all possible cases. However, distinct contexts may discriminate between converters using different policies, thus alleviating this problem.

Hirschfeld et al. (2008) defines context as "*any information which is computationally accessible*". Therefore, any computationally-accessible value should be usable to discriminate converters.

At an abstract level, we define a conversion context as a triple `ctx = <CM,TIM,FM>` where `CM` is a *Converter Manager*, `TIM` is a *Type Inference Manager*, and `FM` is a *Factory Manager*. A high level view of these components is illustrated by figure 3. As their names suggest, the converter manager administers registered converters, the type inference manager administers registered type solvers and the factory manager administers registered factories. A concrete implementation of these managers is introduced in section 4.2.

4. JPC Design

In this section we describe the design of the main components of our library. This discussion sets the ground for introducing its features in section 5.

4.1. An Abstract Prolog Virtual Machine

This component defines an abstract Prolog engine that is able to answer queries employing our term representation. This abstract engine is vendor-agnostic and thus requires a driver to connect to a specific Prolog implementation. Drivers can be built on top of existing Java-Prolog libraries for the target Prolog engine. These libraries are often made available by third parties. JPL, PDT Connector (Rho et al. (2004)) and InterProlog (Calejo (2004)) are three prominent examples.

As part of our Prolog engine abstraction, we provide a set of classes reifying Prolog data types, inspired by similar JPL library classes: For completeness, we list the main classes below:

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.

Method	Description
`fromTerm(Term)`	`Converts a term to a Java object`
`fromTerm(Term,Type)`	`Converts a term to a Java object instance of a given type`
`toTerm(Object)`	`Converts a Java object to a term`
`toTerm(Object,Class)`	`Converts a Java object to a term instance of a given term class`
`getType(Object)`	`Returns the inferred type of the object in an inter-language conversion`
`instantiate(Type)`	`Returns a new instantiation of the given type`

Table 1: Main `Jpc` methods

Var : A Prolog variable.
JRef : A Java reference term. A special kind of term wrapping a Java reference.

As in the JPL library, these term classes can be considered as a structured concrete syntax for Prolog terms. In addition to term classes, other important classes are:

Query : The reification of a Prolog query.
Solution : One query solution. A map binding variable names to terms.

A more detailed description of our abstraction of a Prolog virtual machine is outside the scope of this paper.

4.2. The JPC Context

Our primary library class is a conversion context, modelled by the `Jpc` class. This context encapsulates a bidirectional conversion strategy for a set of Java objects and Prolog terms. We will make use of this context to show, in section 5.6, an improved implementation of the example discussed in section 2. The main API of the `Jpc` class is summarised in table 1.

In order to facilitate the configuration of a `Jpc` instance, we provide a `JpcBuilder` class with a straightforward fluent API for configuring its properties. As outlined in section 3.6, this configuration involves the registration of converters, type solvers, and factories. For example, listing 6 shows how to configure a builder to create a `Jpc` context that knows how to convert objects from the example discussed in section 2. The main API of the `JpcBuilder` class is summarised in table 2.

```java
public static final Jpc jpc = JpcBuilder.create()
    .register(new MetroConverter())
    .register(new LineConverter())
    .register(new StationConverter()).build();
```

Listing 6: Building a `Jpc` context with the `JpcBuilder` class

4.2.1. Implementing Converters

Our current implementation provides two separate interfaces for defining conversions from Java to Prolog (`ToTermConverter`) and vice versa (`FromTermConverter`). Both interfaces inherit from the common interface `JpcConverter` so they both can be registered into a context

Method	Description
`register(JpcConverter)`	Registers a converter
`register(JpcConverter,Term)`	Registers a converter associated to a term
`register(TypeSolver)`	Registers a type solver
`register(TypeSolver,Term)`	Registers a type solver associated to a term
`register(Factory)`	Registers a factory

Table 2: Main `JpcBuilder` methods

with a single call to the method `register(JpcConverter)` introduced before in this section. Behind the curtains, implementors of these interfaces are adapted by our framework, at registration time, as instances of the `Converter` class showed in listing 3.

The `StationConverter` class in listing 7 implements both interfaces. It implements a method defining the conversion of `Station` instances to (lines 6–8) and from (lines 10–13) a Prolog compound term.

```
 1 public class StationConverter implements
 2         ToTermConverter<Station, Compound>,
 3         FromTermConverter<Compound, Station>  {
 4     public static final String STATION_FUNCTOR = "station";
 5 
 6     @Override public Compound toTerm(Station station, Class<Compound> termClass, Jpc
           context) {
 7         return new Compound(STATION_FUNCTOR, asList(new Atom(station.getName())));
 8     }
 9 
10     @Override public Station fromTerm(Compound term, Type type, Jpc context) {
11         String stationName = ((Atom)term.arg(1)).getName();
12         return new StationJpc(stationName);
13     }
14 }
```

Listing 7: The `StationConverter` class

4.2.2. *Implementing Type Solvers*

As discussed in section 3.2, when no type information is provided in a conversion, our library will attempt to infer the best target type based on the actual source object to convert.

As an example, we mentioned that a Prolog list term with a certain structure may be reified, by convention, as a map in Java. Listing 8 shows an extract of this type solver, that implements the `TypeSolver` interface showed in listing 4. It returns the `Map` class on line 12 if it can conclude that the term looks like a map. If it is unable to assign a type to the term it signals it throwing a `UnrecognizedObjectException` exception (line 14).

```
 1 public class MapTypeSolver implements TypeSolver<Compound> {
 2     @Override public Type getType(Compound term) {
 3         if(term.isList()) {
```

```
            ListTerm list = term.asList();
            Predicate<Term> isMapEntry = new Predicate<Term>() {
                @Override
                public boolean apply(Term term) {
                    return isMapEntry(term);
                }
            };
            if(!list.isEmpty() && Iterables.all(list, isMapEntry))
                return Map.class;
        }
        throw new UnrecognizedObjectException();
    }

    private boolean isMapEntry(Term term) {
        ...
    }
}
```

Listing 8: A type solver for a Prolog term representing a map

4.2.3. Implementing Factories

If a converter does not know how to instantiate a conversion target type (e.g., it is abstract), it can ask the `Jpc` context for an instance of such type. For example, in listing 8 we show that a Prolog list with a certain structure will be identified by the type solver as a `Map`. But 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 conversion type. Assuming that a registered factory can instantiate Java maps (listing 9), a converter only needs to invoke the `instantiate(Map.class)` in a `Jpc` context to obtain an instance of the desired type.

```
public class MapFactory implements Factory<Map<?,?>>() {
    @Override
    public Map<?,?> instantiate(Type type) {
        return new HashMap<>();
    }
};
```

Listing 9: A `Map` factory

4.3. An 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. This component is currently not intended to be used directly by the programmer as it lacks many of the features of a full Prolog system. However, several JPC interoperability features rely on it.

Particularly, in section 3.1.2 we discussed that certain conversion artefacts are functions whose domain and range can be specified by means of terms. For those cases, this term has to be explicitly provided by the programmer since it cannot be inferred from the definition of the conversion artefact. For example, our library allows a programmer to

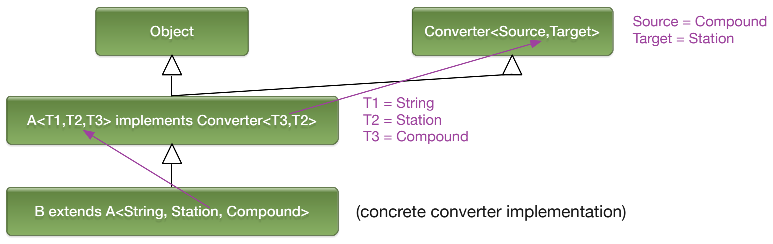

Object
Converter<Source,Target>
Source = Compound
Target = Station
A<T1,T2,T3> implements Converter<T3,T2>
T1 = String
T2 = Station
T3 = Compound
B extends A<String, Station, Compound>
(concrete converter implementation)

Figure 4: Inference of the Source and Target Type of a Converter

specify such terms by means of artefact registration methods in the `JpcBuilder` class (e.g., `register(JpcConverter,Term)` and `register(TypeSolver,Term)`). Behind the curtains, these artefacts are stored in the JPC embedded Prolog database. For example, when a converter is registered as associated to a term, a new fact `converter(Term, Converter)` is asserted in this database. The first argument of the `converter` predicates subsumes the logic terms that should be converted to Java objects using the converter specified as the second argument.

4.4. A Type-Based Categorisation Framework

In section 3.1.1 we discussed that certain conversion artefacts are functions whose domain and range are specified by means of types. This section overviews how our library accomplishes a type-based categorisation of these conversion artefacts. Most examples in this section refer to conversion functions. Type solver and factory functions are subject to a similar categorisation mechanism so we do not repeat the explanation for them.

4.4.1. Source and Target Type Inference

An advantage of a type-based categorisation is that the types defining the domain and range of a conversion function (or other conversion artefacts) can be inferred from its declaration. In listing 3 we showed that a conversion function is modelled in our framework by means of an interface declaring a method accomplishing the conversion. The first step of the categorisation of a converter consists of inferring the types defining its domain and range (`SourceType` and `TargetType` in listing 3). This inference is not trivial, since a programmer can provide the actual implementation of this interface (and the concrete type parameters) in a class arbitrarily far from the `Converter` interface in the class hierarchy (figure 4). Java does not provide a straightforward mechanism for finding the parameter types of a generic class (e.g., the `Converter` interface) given a descendant that defines such types (class `B` in figure 4). Therefore, at registration time we need to traverse the converter class hierarchy until the bindings of its variable types (`Source` and `Target` in figure 4) are resolved.

4.4.2. Categorising Converters

Once the domain and range of registered converter functions (or other conversion artefacts) have been inferred, these converters need to be categorised according to these types. This section discusses our categorisation approach after discarding simpler alternatives that we took into consideration at the beginning of our design.

A straightforward approach is to consider all registered converters as a single chain of responsibility (Gamma et al. (1995)). Such a chain avoids coupling the sender of a conversion request to a specific converter and allows more than one converter to handle the request. Thus, a converter whose domain and range match the source object and the expected conversion type, can either manage the conversion request themselves or may delegate to the next converter in the chain (e.g., by throwing an exception). A simple conflict resolution policy may be based on assigning weights to converters according to some criteria (e.g., their registration order). However, this approach may result in serious performance penalties in systems with a considerable number of converters.

To solve this performance problem, another approach could be to make use of a map associating classes (the converters domains) to a list of suitable registered converters. Unfortunately, this approach does not provide a solution for managing inheritable converters. A subclass of a class associated with a converter (by means of the map) will not implicitly inherit such converter.

Instead, JPC features a mechanism to associate inheritable dynamic properties (e.g., converters and other conversion artefacts) to classes (e.g., the converter domain types). This allows the categorisation of properties according to a type hierarchy, where these properties are found in a similar way as static class properties (e.g., methods or fields) are resolved. In addition, dynamic class properties are not global, but scoped to a specific JPC context.

In order to illustrate our approach, let's first consider a simplified view of our problem, purposely ignoring conflicts raised by converters with crosscutting domains. In this simplified problem, no Java class can be the domain of more than one converter. The more general case is discussed in section 4.4.4.

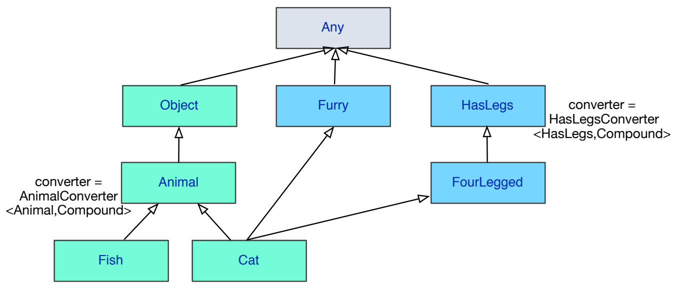
Figure 5 shows an object-oriented hierarchy of animals (example loosely based on an example from Odersky et al. (2011)). In this hierarchy, `Object`, `Animal`, `Fish` and `Cat` are Java classes. `Furry`, `HasLegs` and `FourLegged` are Java interfaces. The auxiliary node `Any` denotes the root of the hierarchy.

The figure also illustrates a type-based categorisation of converters according to this class hierarchy (note the *converter* properties associated to certain classes). Observe that our library considers, in many aspects, both classes and interfaces as same-level categorisation units. For example, one of the two converters shown in the figure (`AnimalConverter`) has the class `Animal` as its domain and `Compound` as its range. The other (`HasLegsConverter`) has the interface `HasLegs` as its domain and `Compound` as its range. This categorisation is automatic, the programmer only has to register a converter as shown in section 4.2 and our library will categorise it behind the curtains.

To keep our example simple, the illustrated converters only define one class as their domain. Converters specifying multiple domain classes (as show in figure 1) would be associated to all such classes.

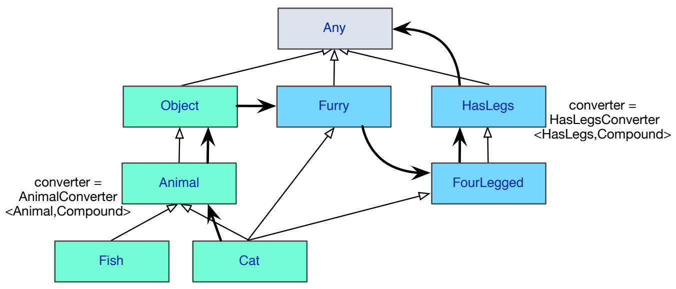
4.4.3. Applying Conversions

The previous converter categorisation supports both non-conflictive and conflictive property resolution scenarios. In the simplest case, assume we would like to find the right converter for an instance of `Fish`. Since there is only single inheritance in the type hierarchy



Any
Object
Furry
HasLegs
converter = HasLegsConverter <HasLegs,Compound>
converter = AnimalConverter <Animal,Compound>
Animal
FourLegged
Fish
Cat

Figure 5: A taxonomy of animals and their converters



Any
Object
Furry
HasLegs
converter = HasLegsConverter <HasLegs,Compound>
converter = AnimalConverter <Animal,Compound>
Animal
FourLegged
Fish
Cat

Figure 6: Linearization: Left to right (classes first)

of `Fish`, its converter is trivially found from the first ancestor defining such a property (i.e., `Animal`). However, if we are interested in the appropriate converter for instances of `Cat`, distinct property resolution strategies may result in different values. For example, if by convention super classes are looked up first, the appropriate converter is `AnimalConverter`. Conversely, if interfaces are looked up first, the converter would be `HasLegsConverter`.

Although different alternatives exist to solve conflicts for ambiguous properties in multi-inheritance hierarchies, unfortunately there is not a perfect one-fit-all solution (Knudsen (1988)). Linearisation helps to solve this conflict unambiguously by defining a linear order in which (super-)categories should be visited when querying a property (Ducournau et al. (1994)). In JPC, linearisation is defined as a function mapping a type to a list of types specifying a search path.

The default JPC linearisation function is inspired by the linearisation function used by Scala (Odersky et al. (2011)) to solve conflicts between classes and traits. JPC uses a left-first depth-first search, before eliminating all but the last occurrence of each category in the resulting list (the main difference with the Scala algorithm is that the later uses a right-first depth-first search). The bold lines in Figure 6 illustrate the resolution order of a property starting from the category `Cat`.

The linearisation function first finds this resolution order (left-first depth-first search): `[Cat, Animal, Object, Any, Furry, Any, FourLegged, HasLegs, Any]` which is reduced down to: `[Cat, Animal, Object, Furry, FourLegged, HasLegs, Any]` (eliminating all but the last occurrence of a redundant category). As the figure shows, this algorithm has the property that an an-

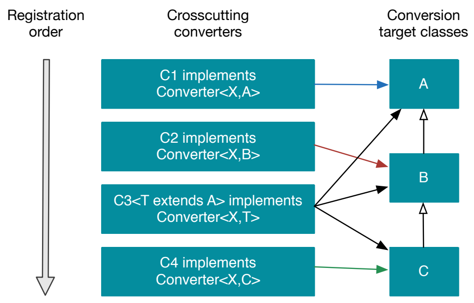

Registration order
Crosscutting converters
Conversion target classes
C1 implements Converter<X,A>
C2 implements Converter<X,B>
C3<T extends A> implements Converter<X,T>
C4 implements Converter<X,C>
A
B
C

Figure 7: Converters with crosscutting domains

cestor category will not be reached until all the descendants leading to it have been explored.

Alternative linearisation functions can be defined on a per context basis (e.g., right to left search, looking at interfaces first). However, this discussion has been left out of this paper due to space constraints.

Once a converter is found, its range is verified against the conversion target type provided by the programmer. If the target type is in range, the converter is requested to accomplish the conversion. Otherwise, the conversion is delegated to the next converter in the linearisation. A converter may also decide to delegate explicitly to the next converter by throwing a `ConversionException` exception. For example, when attempting to convert an instance of the `Cat` class shown in figure 6, the first converter found is the `AnimalConverter` associated to the `Animal` class. If this converter throws a `ConversionException` exception, the conversion will be delegated to the `HasLegsConverter` converter associated with the `HasLegs` class. If this converter also throws a `ConversionException` exception, it would then be propagated to the caller of the conversion routine, since there are no more converters available in the class hierarchy.

4.4.4. Dealing with Crosscutting Converters

This section refines our problem, assuming that converters may have crosscutting domains. This implies that instead of the `converter` property being a single converter associated to a type (figure 6), its value is a composition of (potentially) multiple converters. We consider a composition of converters as a simple converter. Hence, it follows the same rules as our previous example assuming a single converter per class. For example, if a composition of converters cannot perform a conversion operation, it will be delegated to the next composition of converters in the hierarchy (according to the defined linearisation function).

However, we still need to define a mechanism to resolve conflicts between converters in a composition. Figure 7 illustrates such problem. We have defined four converters having the same domain (the class `X`) and as range different classes in the class hierarchy on the right. We illustrate our conflict resolution strategy on the scenario of a programmer wanting to convert an instance of `X` to an instance of `B`. Although the actual implemented algorithm may vary a bit for performance reasons, it is reduced to these steps:

- Remove converters whose range is not compatible with the target type of the conversion. This removes the converter `C1`. Although it can convert to instances of `A` (the super class of `B`), we cannot infer from its declaration that it is able to convert to an instance of `B`.

- Order the remaining converters according to the 'distance' of their range to the target type. This means that a converter declaring as its range the exact target type of the conversion is evaluated before any other declaring as its range a subclass of the target type. In the same way, a converter declaring as its range a direct subclass of the target type is evaluated before anyone declaring as its range a subclass deeper in the hierarchy. According to this criterion, `C2` and `C3` should have priority over `C4`.
- In case of remaining collisions (e.g., converters with the same range distance to the target type), order them according to inverse registration order (i.e., converters registered later have more priority than earlier converters). This implies the order `C3`, `C2` and `C4` (since `C3` is registered after `C2`). Note that this policy allows programmers to register converters that may override default converters.
- Iterate over the ordered converters attempting to execute the conversion operation. If all available converters have been exhausted, throw a `ConversionException` exception signalling a delegation to the next converter (also a converters composition) in the hierarchy.

4.4.5. Context Dependent Conversions

This section puts together all the concepts previously described. We illustrate how a context dependent conversion is accomplished by means of the orchestration of all the conversion artefacts registered into a context.

Figure 8 shows a high level overview of this process. Note that this is a simplified view of the actual implementation. Many details (e.g., performance optimisations) have been left out for the sake of conciseness. The figure depicts a conversion from a Prolog term to its Java object representation (the inverse conversion process is similar). First a client of our library invokes the method `fromTerm(Term,Type)` on a JPC context instance (1). The context requests the type inference manager to infer the best conversion type for the term (2). The inference manager iterates over suitable type solvers (3) until one can infer the conversion type of such term (4).

Afterwards, the most specific type between the inferred type (if any) and the type provided by the programmer (if any) is determined (5). Once the best conversion target type has been found, the conversion request is delegated to the converter manager (6). The converter manager attempts to find a suitable converter (7). Once it is found, it is requested to perform the actual conversion (8). If the chosen converter cannot instantiate the conversion target type, it will delegate to the context the instantiation of such type (9). The context will delegate the request to the factory manager (10). The factory manager will attempt to find a suitable factory (11) and in case it is found the instantiation request will be delegated to it. Once a converter succeeds in completing the requested conversion, it returns the resulting object to the converter manager. The converter manager returns it to the context instance and the latter to the original invoker.

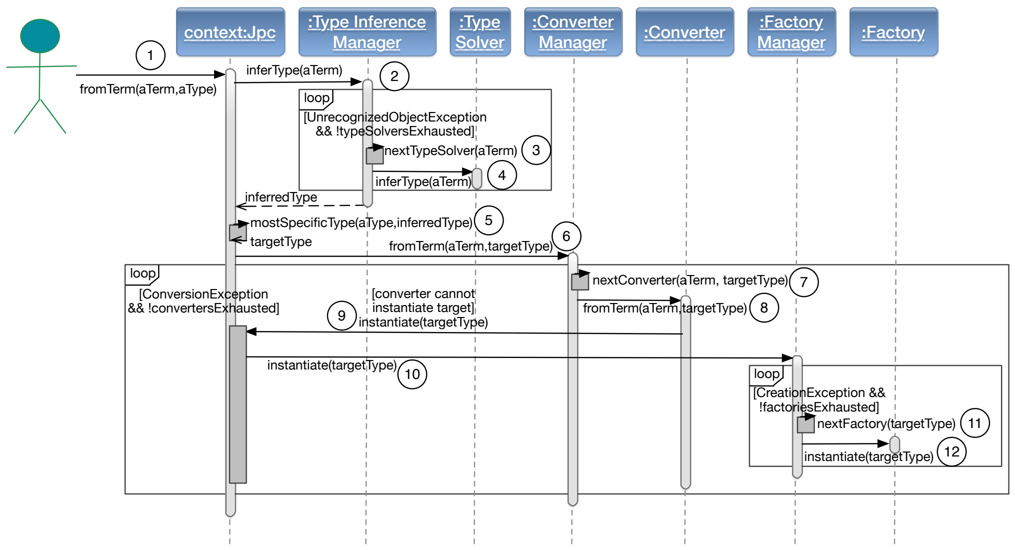


Figure 8: A context dependent conversion

In the next section we show the features of our library in action by means of concrete examples.

5. Inter-Language Conversions

JPC comes with a predefined catalog of converters that support a considerable number of common conversions, to minimise the amount of code to be written when defining new conversions. In the rest of this section we overview both pre-defined and custom conversions.

5.1. Primitives Conversions

In this section we illustrate how to convert between Java and Prolog primitives. In order to facilitate the discussion, we consider a Java `String` also as a primitive, since it is the natural equivalent of the `atom` primitive Prolog type.

The simplest way to use our library is by means of the `toTerm(Object)` and `fromTerm(Term)` methods in the `Jpc` class (introduced in section 4.2). Listing 10 shows a list of successful assertions that illustrates some pre-defined conversions of Java types 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 10: Primitives conversions from Java to Prolog

Pre-defined conversions of Prolog terms to Java types are shown in listing 11.

```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 11: Primitives conversions from Prolog to Java

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 than in Prolog. Thus, distinct Java objects may be mapped to the same Prolog term. For example, line 2 of listing 10 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"`. Unfortunately, this is not necessarily always what the programmer expects. The next section describes how to give hints to our library regarding the appropriate conversion that should be applied.

5.2. Typed Conversions

The `Jpc` class conversion methods can receive as a second parameter the expected type of the converted object. Listing 12 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 10, line 4). This is because we specify the `Atom` class as the target conversion type. In line 2, the `String` `"1"` is converted to an `IntegerTerm`.

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

Listing 12: Typed primitives conversions from Java to Prolog

In a similar way, listing 13 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 13: Typed primitives conversions from Prolog to Java

5.3. Multi-Valued Conversions

The default Jpc catalog of converters also provides conversions for multi-valued data types such as arrays, collections, and maps. Listing 14 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));`

```
1 Term term = jpc.toTerm(new Object[]{"apple", 10});
2 assertEquals(
3 new Compound(".", asList(new Atom("apple"),  // equivalent to .(apple, .(10, []))
4     new Compound(".", asList(new IntegerTerm(10),
5     new Atom("[]"))))),
6 term);
```

Listing 14: Conversion of an array to a Prolog term

A slightly more complex example is illustrated in listing 15. First, a Java map is instantiated (lines 1–5). The default term conversion is applied on line 6, generating a Prolog list with two key-value pairs: `[apple:10, orange:20]`. This result is tested on lines 8–9.

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

Listing 15: Conversion of a map to a Prolog term

5.4. Generic Types Support

Our library provides extensive support for generic types. Consider the example in listing 16. 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 16: 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 17. 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.

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

Listing 17: Changing the behaviour of the converter with a parameterised type

5.5. Inference of Conversion Target Types

Listing 8 showed the implementation of a default type solver responsible of determining if the best conversion type of a term is an instance of `Map`. Listing 18 shows a conversion example that relies on such type solver. 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 values on lines 5 and 6. As expected, our library infers that the best Java type of the term should be a `Map`. This is because the type solver finds that all the elements in the Prolog list (`[apple-10, orange,20]`) 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).

```java
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(10L, map.get("apple"));
6 assertEquals(20L, map.get("orange"));
```

Listing 18: Conversion of a Prolog term to a map

Alternatively, line 4 could be replaced by `List list = jpc.fromTerm(listTerm, List.class);` This type hint explicitly given by the programmer has higher priority that the one inferred by the type solver. In this case, the result would therefore be a list of map entries since the Prolog list would be converted to a Java list (i.e., an instance of a class implementing `List`), but the default conversion of each term in the list would still be a map entry object.

Note that we leave to the programmer the responsibility of providing enough information (i.e., a target type) in case where ambiguities are possible. For example, the previous type solver may answer false negatives if it cannot conclude something from the structure of members in the list (i.e., the list term is empty). If the programmer always specify the intended conversion type the possible ambiguity disappears.

5.6. Custom Conversions

The previous examples employed only predefined converters. Using a custom conversion context as defined in listing 6, we revisit the `Station` class shown in listing 2. Listing 19 shows a new implementation of such class. 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 into a converter class. 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 2 (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 may (optionally) receive a context. The advantage of making a query instance aware of a conversion context becomes clear on line 8. To better understand this, recall from section 4.1 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 whose 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 transparently accomplished 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 Var(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 19: A Java class interacting with Prolog by means of our library

The previous example makes use of a deterministic query. The `Query` class (and query adapters) interface also supports non-deterministic queries (i.e., backtracking over solutions). Internally, this relies on the concrete implementation of the chosen JPC driver. As already mentioned in section 4.1, this discussion has been left out of the scope of this work.

5.7. Term-Quantified Converters

In section 3.1.2 we discussed that the domain of Prolog to Java artefact converters is, in some cases, better quantified using terms. Listing 20 shows a straightforward implementation of the `HelloConverter` mentioned in that section: it returns a Java `String` containing the name of the compound to convert, a white space, and the compound first argument (line 3).

```
1 class HelloConverter implements FromTermConverter<Compound, String> {
2     @Override public String fromTerm(Compound term, Type targetType, Jpc context) {
3         return ((Atom)term.getName()).getName() + " " + ((Atom)term.arg(1)).getName();
4     }
5 }
```

Listing 20: The *Hello World* Converter

Listing 21 shows a concrete example of the registration and usage of this converter. In line 3, we make use of the `JpcBuilder` class to register the `HelloConverter` converter. Note that we pass both an instance of the converter and the term quantifying its domain (`hello(_)`). This is internally translated to an assert in the embedded JPC Prolog database, where the `HelloConverter` converter is associated to a domain term with functor `hello/1`. In line 6 we verify that the result of converting the compound term `hello(world)` is the Java `String` "`hello world`", as specified by the domain quantified converter.

```
1 JpcBuilder builder = JpcBuilder.create();
2 Compound helloCompound = new Compound("hello",  asList(Var.ANONYMOUS_VAR));
3 builder.register(new HelloConverter(), helloCompound);
4 Jpc jpc = builder.build();
5 Compound helloWorldCompound = new Compound("hello",  asList(new Atom("world")));
6 assertEquals("hello world", jpc.fromTerm(helloWorldCompound));
```

Listing 21: Applying a Term-Quantified Converter

6. Discussion and Limitations

We are aware that our approach may not always provide the best solution to all scenarios requiring Java–Prolog interaction. In particular, systems with high performance requirements may prefer to not use the inter-language conversion facilities we have made available. This is often the case of frameworks attempting to provide higher level abstractions that simplify the developer's work. But our library does not impose a particular abstraction level: JPC may be used exactly like the JPL library without further additions, since our conversion constructs are entirely optional. In any case, JPC's high level constructs can often be employed to quickly prototype a system requiring Java–Prolog interoperability. Afterwards, these constructs may be replaced, at performance critical spots, by the lower level API.

Another limitation concerns some difficulty regarding the traceability of the converters employed in a given conversion operation. This may hinder the debugging and maintenance of programs in certain scenarios. Particularly, the frequent usage of converters with crosscutting domains and ranges in multi-inheritance hierarchies may just shift the complexity from writing conversions easily to understanding whether the program is choosing the proper conversion. These are advanced features that should be managed with care. In most cases, the programmer should favour non-crosscutting converters and limiting as much as possible the categorisation of converters in multi-inheritance hierarchies. More inherently complex problems can profit of the flexibility and customisability of JPC advanced features.

As in the previous discussion, supporting multiple conversion contexts is a powerful feature that should be accompanied with a thoughtful design. Dealing with a great number of such contexts would increase the complexity for a programmer to foresee all their possible interactions, and ensure that programs use the right or expected converters in all possible contexts. Note that the same potential problem is present in popular libraries also employing the notion of a conversion context (e.g., the GSON library (Google Inc. (2012))). As mentioned in section 3.6, in certain cases additional tool support alleviating this problem can be developed. Particularly, our LogicObjects library (Castro et al. (2013)) automatically considers as part of the conversion context the semantics of a conversion (e.g., if the result of a conversion is intended to be a predicate argument or the receiver of a Logtalk method call).

7. 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 context were a source of inspiration for accomplishing our two-way conversions between Java and Prolog artefacts. Gson is a high-level tool for bidirectional conversions between Java objects and their JSON representation. Certain JPC components can be regarded as a re-implementation of Gson in the domain of Java-Prolog artefact conversions.

In addition, although our abstraction of a 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 applications interacting with databases (e.g., JDBC Fisher et al. (2003)). Some design aspects of this Prolog engine abstraction were inspired on InterProlog and the PDT library. Concerning our classes reifying Prolog terms, most of them were inspired by the JPL library.

The ATerm library (van den Brand et al. (2000)) defines a mechanism for maximal sub-term sharing for representing tree-like structures. We may adopt a similar approach in the future to reduce the memory footprint of our term structures if required.

The idea of storing references from the OO language in a Prolog database (from section 3.1.2) can be found in some Java implementations of Prolog (e.g., Jinni (Tarau (2004)), Prolog Cafe (Banbara et al. (2006)), LeanProlog (Tarau (2011))) and embedded logic engines (e.g., Soul (Wuyts (2001))).

The Soul language allows the programmer to customise not the term representation of an object, but rather its unification mechanism (Roover et al. (2011)). 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. A limitation of the Soul approach is that it requires a logic engine embedded in the object-oriented language and supporting open-unification. These requirements limit its portability to other Prolog engines with different architectures. In addition, Soul's approach does not provide a mechanism for context-dependent unification, equivalent to our context-dependent inter-language mappings.

There are several techniques for automating the bi-directional conversions between tree-like structures (e.g., XML documents) and artefacts from an object-oriented language. For example, the JAXB library (Kawaguchi et al. (2009)) allows custom conversions to be inferred using annotations present in the object-oriented code. However, although pre-defined JPC converters provide some degree of transparency in simple conversion operations, our library is not focussed on achieving a high level of transparency regarding such operations.

Nevertheless, one of the motivations behind JPC was to provide appropriate building blocks to higher level libraries (e.g., LogicObjects) that do support such advanced features.

8. 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 categorise conversion concerns in Java-Prolog programs, given them a fine-grained control about how a Java object should be represented and 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.

This work profited from cross-fertilisation of ideas from several 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 seen as an exercise of assembling a puzzle of abstractions, where most (if not all) pieces can be adapted from, or inspired by, well-proven existing solutions.

Although the problems discussed in this paper are often found in programs dealing with inter-language conversion routines, many of the techniques introduced here can be applied to other scenarios. In fact, we have factored out as separate open source projects the modules in charge of (1) creating and managing (type) categorisations[2] and (2) providing a general mechanism for categorising and applying conversions between arbitrary objects.[3]

In section 6 we overview the potential performance limitations of our high level API. As part of our future work, we intend to measure such performance overhead to provide an idea of which scenarios are better suited for such an API.

We plan to improve the interoperability from the Prolog side by developing a portable Logtalk library for simplifying the interaction from the perspective of this language, as we have done for 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)) systems.

We would also like to continue the development of our embedded Prolog database described in section 4.3 towards a more complete embedded Prolog system. In that way, it could be used in the future as a stand-alone component.

Acknowledgements. This work is partially financed by the ERDF (European Regional Development Fund) through the COMPETE Programme and by FCT (Portuguese Foundation for Science and Technology) within project LEAP (FCOMP-01-0124-FEDER-015008).

[2]JGum: `https://github.com/jgum`

[3]JConverter: `https://github.com/jconverter`

References

Banbara, M., Tamura, N., Inoue, K., 2006. Prolog Cafe: A Prolog to Java Translator System. In: Umeda, M., Wolf, A., Bartenstein, O., Geske, U., Seipel, D., Takata, O. (Eds.), Declarative Programming for Knowledge Management. Vol. 4369 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 1–11.

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., June 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). RAM-SE'12. ACM, pp. 11–16.

Castro, S., Mens, K., Moura, P., January 2013. LogicObjects: Enabling Logic Programming in Java Through Linguistic Symbiosis. In: Sagonas, K. (Ed.), Proceedings of the 15th International Symposium on Practical Aspects of Declarative Languages (PADL). Vol. 7752 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, Rome, Italy, pp. 26–42.

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

Dijkstra, E. W., 1997. A Discipline of Programming, 1st Edition. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Ducournau, R., Habib, M., Huchard, M., Mugnier, M.-L., 1994. Proposal for a Monotonic Multiple Inheritance Linearization. In: OOPSLA. pp. 164–175.

Fisher, M., Ellis, J., Bruce, J., 2003. JDBC API Tutorial and Reference. Addison-Wesley, Boston, MA.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

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/`.

Hirschfeld, R., Costanza, P., Nierstrasz, O., 2008. Context-Oriented Programming. Journal of Object Technology, March-April 2008, ETH Zurich 7 (3), 125–151.

Kawaguchi, K., Fialli, J., Vajjhala, S., 2009. The Java Architecture for XML Binding (JAXB) 2.2. URL `http://jcp.org/en/jsr/detail?id=222`

Kiczales, G., 1996. Aspect-Oriented Programming. ACM Comput. Surv. 28 (4es), 154.

Knudsen, J. L., 1988. Name Collision in Multiple Classification Hierarchies. In: Gjessing, S., Nygaard, K. (Eds.), ECOOP 88 European Conference on Object-Oriented Programming, Oslo, Norway, August 15-17, 1988, Proceedings. Vol. 322 of Lecture Notes in Computer Science. Springer, pp. 93–109.

Masuhara, H., Kiczales, G., 2003. Modeling Crosscutting in Aspect-Oriented Mechanisms. In: Cardelli, L. (Ed.), ECOOP 2003 – Object-Oriented Programming. Vol. 2743 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 2–28.

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.

Odersky, M., Spoon, L., Venners, B., 2011. Programming in Scala: A Comprehensive Step-by-Step Guide, 2Nd Edition, 2nd Edition. Artima Incorporation, USA.

Parnas, D. L., Dec. 1972. On the Criteria to be Used in Decomposing Systems into Modules. Commun. ACM 15 (12), 1053–1058.

Rho, T., Degener, L., Günter Kniesel, Frank Mühlschlegel, Eva Stöwe, Noth, F., Becker, A., 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., Jonckers, V., 2011. The SOUL Tool Suite for Querying Programs in Symbiosis with Eclipse. In: PPPJ. pp. 71–80.

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/.

Sterling, L., Shapiro, E., 1994. The Art of Prolog (2nd ed.): Advanced Programming Techniques. MIT Press, Cambridge, MA, USA.

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

Tarau, P., 2004. Agent Oriented Logic Programming Constructs in Jinni 2004. In: Demoen, B., Lifschitz, V. (Eds.), ICLP. Vol. 3132 of Lecture Notes in Computer Science. Springer, pp. 477–478.
URL http://dblp.uni-trier.de/db/conf/iclp/iclp2004.html#Tarau04

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, pp. 129–138.

van den Brand, M., de Jong, H. A., Klint, P., Olivier, P. A., 2000. Efficient Annotated Terms. Softw., Pract. Exper. 30 (3), 259–291.

Vanbrabant, R., 2008. Google Guice: Agile Lightweight Dependency Injection Framework (Firstpress). APress.

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

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.