Automatic Integration of Hybrid Java-Prolog Entities with LogicObjects

Sergio Castro [1], Kim Mens [1] and Paulo Moura [2]
(1) Université catholique de Louvain, ICTEAM, INGI.
(2) Center for Research in Advanced Computing Systems, INESC-TEC.

Contents

Introduction

Logic languages excel for problems that can defined declaratively, e.g. applications that require symbolic computation such as expert and planning systems. [1] [2]. However, it is often difficult to develop complete applications that require e.g. GUIs, heavy numerical computations, or low-level operating system and network access [3] [4]. On the other hand, object-oriented languages have demonstrated their usefulness for modelling a wide range of concepts (e.g., GUIs) found in many business scenarios [5]. The availability of continuously growing software ecosystems around widely used modern object-oriented languages, including advanced IDEs and rich sets of libraries, has significantly contributed to their success.

Non-trivial applications can profit from implementing their components, or even distinct routines of the same entity, in the language that is most appropriate for expressing them [3] [6] [7]. However, the integration of programs or routines written in different languages is not trivial when such languages belong to different paradigms [8] [9] [10].

Although numerous approaches allow to integrate Java and Prolog programs, few of them put an emphasis on the transparency and automation of the integration concern at the entity level. In this article, we describe an approach based on the notion of automatic inter-language conversions that does tackle this problem.

Preparing the Ground: Inter-Language Conversions with JPC

The core of our approach is built on top of an extensible and fine-grained mechanism for mapping artefacts between the Prolog and Java worlds: the JPC (Java-Prolog Connectivity) library. Conversely to other Java-Prolog integration libraries, JPC leaves open the mapping between concrete Prolog terms and Java objects.

JPC's design has been inspired by a well-known library for accomplishing context-dependent conversions between Java and JSON artefacts: Google's Gson library. The goal of Gson is to provide an intuitive, minimalistic interface for facilitating the specification and execution of Java-JSON bidirectional conversions. As part of our design philosophy we extrapolate several design goals of Gson to our domain (i.e., Java-Prolog conversions), amongst others:

- Allowing arbitrary objects to be converted to and from Prolog terms.
- Working with pre-existing objects that cannot or should not be modified (e.g., no sources available).
- Including a considerable catalog of pre-defined bi-directional converters.
- Allowing to extend the default conversion strategy by means of custom converters.
- Encapsulating conversion strategies as reified conversion contexts.
- Supporting objects with deep hierarchies that may contain generic classes.
- Enabling type-guided conversions. For example, a Prolog list may be converted to a Java Array, List or Map, according to the expected conversion type, if any.

In the rest of this section we overview how JPC accomplishes some of these goals.

Conversion Contexts

A conversion context is a scoped bidirectional conversion strategy reified by means of the `Jpc` class. Being encapsulated into an entity, multiple conversion contexts can be employed in the same program, facilitating the co-existence of different mappings when required. New conversion contexts can be easily created by means of a `context builder`. The instantiation of a default conversion context (i.e., a context including only pre-defined conversions) is shown in the code snippet below:

```
1  JpcBuilder builder = JpcBuilder.create(); // a default conversion context builder
2  Jpc jpc = builder.build(); // a default conversion context
```

Pre-Defined Conversions

Once a conversion context has been instantiated, it can be employed to convert between Java and Prolog artefacts. By default, a conversion context can accomplish common conversions (e.g., conversions between primitives) without further configuration. Some of these pre-defined conversions are shown below:

```
1  Jpc jpc = JpcBuilder.create().build(); // a default conversion context
2
3  //Java to Prolog
4  jpc.toTerm(true);                  // ==> new Atom("true")
5  jpc.toTerm("1");                   // ==> new Atom("1")
6  jpc.toTerm(1);                     // ==> new IntegerTerm(1)
7  jpc.toTerm(1D);                    // ==> new FloatTerm(1)
8
9  //Prolog to Java
10 jpc.fromTerm(new Atom("true"));    // ==> true
11 jpc.fromTerm(new Atom("1"));       // ==> "1"
12 jpc.fromTerm(new IntegerTerm(1));  // ==> 1L
13 jpc.fromTerm(new FloatTerm(1));    // ==> 1D
```

Type-Guided Conversions

A conversion operation can also be parameterised with the expected result type of a conversion. Some examples of type-guided conversions are shown in the following code snippet:

```
1  Term listTerm = listTerm(new Atom("1"), new Atom("2")); //Prolog list -> ['1', '2']
2  String[] array = jpc.fromTerm(listTerm, String[].class); //array of strings
3  List<?> list = jpc.fromTerm(listTerm, List.class); //list of strings
4  Type type = new TypeToken<List<Integer>>(){}.getType(); //reification of the List<Integer> type
5  list = jpc.fromTerm(listTerm, type); //list of integers
```

Depending on the target conversion type, the Prolog atoms list `['1', '2']` (line 1) is converted into a a string array (line 2), a list of strings (line 3) or a list of integers (line 5).

Custom Conversions

A conversion context can be extended with custom converters. Such a converter is just a class implementing the `FromTermConverter` and/or `ToTermConverter` interfaces.

When registered into a conversion context, a converter adds to the pre-defined conversions a new mapping of artefacts. As an example, consider a converter that knows how to convert a Prolog compound term to an instance of the `Person` class:

```
1  public class PersonConverter implements FromTermConverter<Compound, Person> {
2
3      @Override public Person fromTerm(Compound term, Type type, Jpc context) {
4          String personName = ((Atom)term.arg(1)).getName();
5          return new Person(personName);
6      }
7
8  }
```

The following code snippet illustrates how a custom conversion context extended with the `PersonConverter` can be employed to facilitate the interpretation of a query result as a Java object with a minimum amount of code:

```
1  JpcBuilder builder = JpcBuilder.create().register(new PersonConverter(),
2                                                    new Functor("person"), 1).asTerm()); //person(_)
3  Jpc jpc = builder.build(); //custom conversion context
4  //the term bound to P is automatically converted to an instance of the Person class
5  Person person = engine.query("cool_person(P)", jpc).<Person>selectObject("P").oneSolutionOrThrow();
```

Note that at the moment of registering the converter, it is associated with the compound term `person(_)` (line 2). Therefore, the converter is applicable only to terms subsumed by such a compound.

Automatic Integration of Hybrid Entities with LogicObjects

The low-level inter-language conversion framework provided by JPC is a convenient building block for facilitating the implementation of more advanced integration libraries. LogicObjects is a high-level integration framework for Java and Prolog built on top of JPC. It allows to define hybrid entities with partial implementations in Java and Prolog. Although to a certain extent LogicObjects is compatible with plain Prolog, to get the best out of it, it is recommended to install on the Prolog side Logtalk, an object-oriented layer for Prolog.

Figure 1 shows the definition of a `Person` entity. This entity has a dual nature. In the Java world, it takes the form of an instance of the `Person` class. In the Prolog world, it is reified as the Logtalk parametric object `person/1`. In this example, the implementation is scattered over the two worlds (the method `experience()` is defined in Java and the predicate `salary/1` in Prolog). However, it is also possible to create entities that are completely defined either in Java or Prolog, but still transparently accessible from the foreign language.

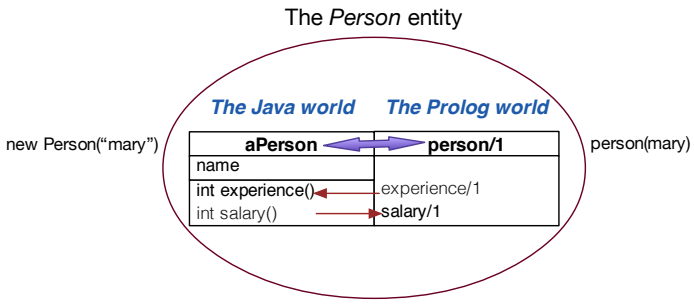

Fig 1. - A dual representation of the Person entity.

In the rest of this section we review the automatic integration mechanisms provided by LogicObjects from both language perspectives.

Integration from the Java Perspective

The following code snippet shows the Java side of the implementation of the `Person` entity. Abstract methods correspond to routines implemented in Prolog/Logtalk.

```
1  @LObject(name="person", args={"name"})
2  abstract class Person {
3     private final String name;
4
5     public Person(String name) {this.name = name;}
6
7     @LMethod(args = {"LSolution"})
8     public abstract int salary();
9
10    public int experience() {
11        ...
12    }
13
14 }
```

The `@LObject` annotation on line 1 provides mapping information to LogicObjects. The `name` attribute indicates that an instance of the `Person` class is reified as a compound with name `person`. The `args` attribute signals that the argument of such a compound is the term representation of the instance variable `name`.

The `@LMethod` annotation on line 7 provides mapping information regarding a specific routine. In this case, the `salary()` Java method is mapped to a predicate having the same name. As arguments, it will have the unbound logic variable `LSolution`. Therefore, an invocation of the `salary()` method on a person with name `"mary"` will be interpreted as querying the Prolog goal:

```
person(mary)::salary(LSolution)
```

where `person(mary)` corresponds to the conversion to a term of the receiver of the message (an instance of `Person`); `salary(LSolution)` is the conversion to a predicate of the method `salary()` and `::` is the Logtalk message sending operator.

LogicObjects provides several heuristics for determining the return value in Java of a routine implemented on the Prolog side. One of them consists in inspecting the names of the unbound logic variables. In case it encounters an occurrence of a variable named `LSolution` (line 7), it will consider as the return value of the method the conversion to a Java object of the term bound to that variable. Also note that by default a query is interpreted as deterministic. Hence, only its first solution is considered. However, this can be customised by means of another Java annotation (e.g., to compose all the solutions in a container object such as an instance of `java.util.List`).

In addition to the `@LObject` and `@LMethod` annotations illustrated in the `Person` class, other useful annotations are:

- `@LComposition`: If present in a method, all solutions will be composed into one single object.
- `@LSolution`: Allows to specify the term representation of an object corresponding to a single solution.
- `@LQuery`: Allows to map a Java method to an arbitrary Prolog query.

We have left, however, a detailed explanation of these annotations outside the scope of this short article.

An instance of a class with a partial implementation in Prolog can be obtained by means of the static method `newLogicObject`, as shown in the following code snippet:

```
import static org.logicobjects.newLogicObject;
...

Person person = newLogicObject(Person.class, "mary");
System.out.println("Salary: " + person.salary()); //automatic delegation to Prolog
```

The first argument is the class to instantiate. Remaining arguments correspond to the class constructor arguments.

Integration from the Prolog Perspective

The logic counterpart of the `Person` class is the Logtalk parametric object `person/1` defined as in the following code snippet:

```
:- object(person(_Name) imports jobject).

    :- public(salary/1).
    salary(S) :- ...

:- end_object.
```

Any Logtalk object importing the `jobject` category will delegate automatically to the Java side any message that it does not understand. For example, a Logtalk method call `person(mary)::experience` will be interpreted in Java as:

```
new Person("mary").experience()
```

where `new Person("mary")` corresponds to the conversion to a Java object of the term `person(mary)` and the method `experience()` the looked-up method when receiving a message with the same name on the logic side.

Java methods returning values impose a defiance to the transparency of our approach. While returning values is a natural concept in Java, it does not exist in Prolog. Therefore, an important problem to solve is how a programmer can specify that a Java method return value is required on the Prolog side. We provide three different alternative mechanisms for this. They are illustrated by means of the following equivalent message calls on the `person(mary)` Logtalk object:

```
1  person(mary)::experience(return(ReturnSpecifier)). %return value specified as an argument of the method.
2  jobject(person(mary), ReturnSpecifier)::experience. %return value specified as an argument of the jobject/2 Logtalk object.
3  person(mary)::experience return ReturnSpecifier. %return value specified by means of the return operator.
```

On line 1, the return value is part of the message term. Arguments matching `return(_)` are ignored since they correspond not to a Java method parameter, but to the Java method return value. On line 2, the return value is an argument of the auxiliary Logtalk object `jobject/2`, which has as first argument the receiver of the message, and the second argument corresponds to the return value. This alternative has the advantage that the message term does not require to be polluted with a return value. As the last alternative on line 3, the return value is captured by means of the `return` operator. This has the advantage that the return value is not present as an argument neither of the receiver nor the message. None of these approaches is perfect, since they make explicit the foreign concept of a returned value on the Prolog side. We believe, however, that providing several alternatives alleviate this problem to a certain extent.

Finally, note that the object returned from a Java method can be reified as a term in many different ways (e.g., a term containing the object serialisation data, or a simple numeric identifier pointing to a Java reference, etc). Therefore, an integration approach should not impose a single reification strategy. We provide a solution to this problem by means of introducing the notion of *return specifiers*.

By employing return specifiers, a programmer can provide a term describing the meaning of an object returned from the Java world. For example, if we invoke the Java method `experience()` from the Prolog side, we should write `term(R)` as the return specifier, which means that the object returned from the Java method should be converted into a term according to the default JPC conversion context. This is illustrated by the following example:

```
person(mary)::experience(return(term(R))).
```

Several possible ways of interpreting a return value are possible by means of choosing distinct return specifiers. The following list describes some of them:

- `term(R)`: `R` is the default conversion to a term of the returned object.
- `jserialized(R)`: `R` is a term representation of the serialisation of the returned object.
- `jref(R)`: `R` is an opaque term representation of a Java reference. The reference remains valid as long as the term exists somewhere in the Prolog database.
- `weak(jref(R))`: `R` is an opaque term representation of a Java reference. The reference remains valid as long as the reference is not garbage-collected in the Java world.
- `strong(jref(R))`: `R` is an opaque term representation of a Java reference. The reference remains valid as long as the reference symbol is not explicitly forgotten.

We have left outside the scope of this article a complete description of all the possible return specifiers.

Conclusions

LogicObjects enables a fine-grained, (semi-)automatic integration (i.e., at the entity level) of routines written in Java and Prolog. It is an example of the advanced integration frameworks that can be built on top of the notion of inter-language conversions implemented by the JPC library.

There are other powerful JPC and LogicObjects features in addition to the ones described here. Those include a comprehensive set of heuristics for interpreting the solution to a Prolog query as a Java object, autoloading of Prolog artefacts, and the automatic propagation and handling of inter-language exceptions. A description of these and other advanced features can be found in the JPC and LogicObjects documentation.

JPC and LogicObjects are currently compatible with SWI, YAP and XSB Prolog by means of drivers developed on top of the JPL, PDT Connector and InterProlog libraries. We envision the development of native drivers and support for other Prolog engines in the short term.

We hope that both JPC and LogicObjects will be useful to the Prolog and Java community. In this spirit, we are releasing our work as open-source software.

Bibliography

1. Artificial Intelligence, A Modern Approach. Russel, S. and Norvig, P. Prentice Hall. 1995.
2. The Art of Prolog (2nd ed.): Advanced Programming Techniques. Sterling, L. and Shapiro, E. MIT Press. Cambridge, MA, USA. 1994.
3. InterProlog: Towards a Declarative Embedding of Logic Programming in Java. Calejo, M. In proceedings of Logics in Artificial Intelligence, 9th European Conference, JELIA. V. 3229. P. 714-717. Lisbon, Portugal. September 27-30, 2004.
4. An Architecture for Making Object-Oriented Systems Available from Prolog. Wielemaker, J. and Anjewierden, A. Workshop on Logic-based methods in Programming Environments (WLPE). 2002.
5. Object-Oriented Software Construction. Meyer, B. Prentice-Hall, Inc. 1988.
6. When and How to Develop Domain-specific Languages. Mernik, M, Heering, J. and Sloane, A. ACM Comput. Surv. V. 37, N. 4, P. 316-344. New York, USA. December 2005.
7. Seamless integration of rule-based knowledge and object-oriented functionality with linguistic symbiosis. D'Hondt, M., Gybels, K. and Jonckers, V. Proceedings of the 2004 ACM symposium on Applied computing. New York, USA. 2004.
8. Object-oriented computations in logic programming. Omicini, A. and Natali, A. Object-Oriented Programming. Lecture Notes in Computer Science. V. 821, P. 194-212. 1994.
9. Towards Linguistic Symbiosis of an Object-Oriented and a Logic Programming Language. Brichau, J. Gybels, K and Wuyts, R. In Jörg Striegnitz, Kei Davis, and Yannis Smaragdakis, editors, Proceedings of the Workshop on Multiparadigm Programming with Object-Oriented Languages. 2002.
10. SOUL and Smalltalk - Just Married: Evolution of the Interaction Between a Logic and an Object-Oriented Language Towards Symbiosis. Gybels, K. Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages. 2003.