Integrating Java and Prolog using Java 5.0 generics and annotations

Maurizio Cimadamore and Mirko Viroli

DEIS, Cesena
Università degli Studi di Bologna
via Venezia 52, 47023 Cesena, Italy
[maurizio.cimadamore|mirko.viroli]@unibo.it

Abstract. Although object-oriented languages are nowadays the mainstream for application development, several research contexts suggest that a multi-paradigm approach is worth pursuing. In particular, a declarative, logic-based paradigm could fruitfully add functionalities related to intelligence, adaptivity, and conciseness in expressing algorithms. In this paper we present a framework for enhancing interoperability between Java and Prolog, based on the tuProlog open-source Prolog engine for Java. Smoother language-interoperability is achieved through two stacked layers: *(i)* an API layer for automated mapping of Java types into Prolog types (and viceversa) and seamless exploitation of the Generic Collections Framework; and *(ii)* an annotation layer, that aims at truly extending Java programming with the ability of specifying Prolog-based declarative implementations of Java methods, relying on Java annotations.

1 Introduction

The growing complexity in modern software systems comes in two main ways: *(i) interaction*—since systems can be thought of as sub-systems that interact so as to achieve a global system *goal*; *(ii) intelligence*—since systems must generally embed adaptiveness and reconfiguration capabilities in order to achieve their task in open and unpredictable environments. Object-oriented languages typically provide valuable tools for decomposing a system in terms of interacting sub-systems (like objects, processes, components, agents); unfortunately, featuring intelligence, adaptiveness, and reconfigurability in those sub-systems through mainstream object-oriented languages (such as Java) is still very difficult. On the other hand, declarative logic-based programming provides valuable constructs for building software components based on logic theories, where the concepts of reasoning, inference, and declarative specification of algorithms and behavior are more naturally understood and implemented.

Accordingly, integrating object-oriented and logic-based programming has been the subject of several researches and corresponding technologies [3–7]. We focus on one such system which strives for a tight integration between the Java programming language and Prolog, namely, the tuProlog open-source project:

a lightweight Prolog engine written in Java and fully-integrated in the Java framework [8, 6]. This project focusses on solving at the library level the problem of writing applications using two most representative programming frameworks of object-orientation and logic programming, Java and Prolog: suitable Java libraries are used to access Prolog theories, and vice-versa a Prolog library of predicates is used to access Java APIs.

Developed on top of tuProlog, in this paper we push the Java-Prolog language integration a step further by studying a framework for extending Java programming with Prolog-based abilities leveraging some recent features of Java [1, 2]: *(i)* Java generics, by which a more expressive, compile-time-checked type system can be used; *(ii)* the Java Collections Framework, supporting powerful creation and access of Java data structures; and *(iii)* Java Annotations, allowing to flexibly extend the language with sort of metadata. In this context, a programming framework can be extended in basically three ways: by a library, introducing new concepts and idioms in terms of a system of types and an API; by an annotations system [1], decorating the source code and retrieving such information through *Reflection*; and finally by a true language extension, generally requiring a new compiler (and sometimes even an extended Virtual Machine). With this framework we decided to work on the first two levels only, for the third one has serious issues concerning maintainability, deployment, and compatibility. In particular, our work is structured in two stacked layers, each one further reducing the semantic gap between Java and Prolog:

- *Generic API layer*: a new hierarchy of Java classes modeling Prolog terms, so that user code interfacing with the tuProlog engine can take advantage of *generic types* and *enhanced for loops* [1].
- *Annotation layer*: a true Prolog-based extension of Java programming, providing custom Java *annotations* to be used for embedding Prolog theories within Java classes, and specifying Prolog-code as a possible implementation of given Java methods.

The remainder of the paper is organized as follows. Section 2 briefly describes the tuProlog engine, its shortcomings, and the main goals of the framework presented in this paper, Section 3 focuses on the generic API layer, Section 4 describes the annotation layer, Section 5 gives further explanation about advanced features, and finally Section 6 concludes providing final remarks.

2 Improving tuProlog

The *tuProlog* engine is a lightweight, full-fledged Prolog engine entirely written in Java. Some of the most compelling features of the tuProlog engine include:

- *Minimality.* The tuProlog core is a tiny Java object that contains only the most essential properties of a Prolog engine. Only the required Prolog features (like ISO compliance, I/O predicates, DCG operators) are then to be added to or removed from a tuProlog engine according to the contingent application needs.

- *Integration with Java*. The components of a tuProlog application can be developed by choosing at any step the most suitable paradigm—either declarative/logic or imperative/object-oriented. From the tuProlog side, thanks to the *JavaLibrary* library, any Java entity (object, class, package) can be represented as a Prolog term, and thus exploited from Prolog. From the Java side, the tuProlog engine can be invoked and used as a simple Java object according to the application needs.

In the remainder of this paper, we will focus on "Prolog from Java" interoperability: we are interested in investigating those patterns that simplify the usage of Prolog-like features within Java mainstream programming.

2.1 Prolog into Java: the standard tuProlog recipe

Since the tuProlog engine is 100% pure Java, it is possible to query the engine for a solution directly within a Java program; the tuProlog engine is in fact seen as any other Java object. We consider as reference example in this paper the case of a space-search-like algorithm for manipulating data structures, namely, generating all permutations of a list. In spite of its simplicity this example allows us to discuss all the main aspects of our framework, as well as the improvement with respect to the existing tuProlog library.

In Figure 1 we report Java class `PermutationUtility`, interfacing with the tuProlog engine to find all the permutations of a list. This example shows some details of the tuProlog library, as well as the typical programming pattern to exploit it:

1. Java data structures are typically different from logic terms, and it is then necessary in general to apply a transformation step, turning Java objects into proper tuProlog `Term` objects. In the example code an object of type `LinkedList<Integer>` is translated into an array of `Struct` elements—standing for compound terms.
2. Now that all the input elements are ready to be processed by the tuProlog engine, a goal should be created: this is basically another `Struct` for the term `permutation([1,2,3],X)` acting as input to the tuProlog engine.
3. The tuProlog engine has to be created; once it has been set up, it must be initialized with a `Theory` object—here encoded as the array of strings `theory`, though also an external file stream could be used.
4. The engine is ready to find a solution for the above goal. All solutions returned by the engine are `SolveInfo` objects that must be accessed in order to browse the entire solution space. Once all the solutions have been explored, the variable `solutions` here stores a `Vector`, keeping all the terms corresponding to the permutations of the given input list.
5. The results in `solutions` might be converted to a more usable representation; since it is known that `solutions` is indeed holding Prolog lists, it could be interesting to convert those lists back to plain Java Collection objects.

Note that in this paper we are not concerned with those applications requiring specific performance factors, for which the above wrapping and translation approach might introduce too much overhead. Our concern is most on linguistic aspects—the above issues are meant to be tackled "under the hood" by some orthogonal optimization technique.

2.2 Towards a better Java-Prolog integration

Although the tuProlog library allows for fully exploiting Prolog engine from Java, we seek in this paper for a tighter integration, one in which the code in Figure 1 could be simplified and made more coherent. Some steps that could be automated include:

- *Marshalling/Unmarshalling.* A simple mechanism to switch between the user and the Prolog representations of a given data could be provided. Recalling the example, this means that it should be easier to convert the `LinkedList<Integer>` object to a suitable tuProlog `Struct` object (and viceversa).
- *Result browsing.* The process of finding all the solutions of a given Prolog query should be simplified. The pattern for iterating results (shown in Figure 1) could e.g. be replaced by a single JDK5.0's enhanced for-loop statement through a `java.util.Iterator`.
- *Abstraction.* All the engine-dependent details should not be exposed to the user; in the example, the user code should be transparent to processes like *(i)* the initialization of the tuProlog engine, *(ii)* the construction of both the goal `Struct` and *(iii)* the `Theory` object. Instead, it is more interesting to see Prolog code as a true "constituent" of Java classes and methods, e.g. as a possible implementation for a Java method.

In this paper we present a framework for enhancing the language interoperability between Java and Prolog, that is to make it easier for the user to integrate Prolog-like functionalities within Java programs. High level integration between Prolog and Java is achieved through two layers:

- *Generic API layer.* A brand new hierarchy of Java classes modeling Prolog terms avoiding the need for explicit marshaling/unmarshaling operations from/to the user data; these classes are *generic* so as to enhance compile-time checking and to avoid redundant type conversions. Such terms can be converted to (and from) tuProlog terms in order to preserve interoperability with the tuProlog engine. This layer also includes a thin wrapper for the tuProlog engine with embedded support for these terms library along with capabilities for browsing all results of a given Prolog query within a single `for-each` loop.
- *Annotation layer.* The framework comes with a set of custom Java annotations that can be used to embed Prolog theories within Java classes (henceforth *Prolog classes*) and to define mappings between Prolog queries and Java methods (henceforth *Prolog methods*); from the user perspective, the

```
public class PermutationUtility {
   String theory =  "remove([X|Xs],X,Xs).\n" +
                    "remove([X|Xs],E,[X|Ys]):-remove(Xs,E,Ys).\n"+
                    "permutation([],[]).\n"+
                    "permutation(Xs,[X|Ys]):-remove(Xs,X,Zs),permutation(Zs, Ys).\n";

   public static void main(String[] args) throws TuPrologException {
      LinkedList<Integer> l=new LinkedList<Integer>();
      l.add(1);
      l.add(2);
      l.add(3);

      /**** 1. convert 'l' to a tuProlog list ****/

      Term[] term_array = new Term[l.size()];
      for (int i = 0;i<l.size();i++) {
         term_array[i]=new Int(l.get(i));
      }

      /**** 2. build the goal ****/

      Struct list_struct = new Struct(term_array);
      Var x = new Var("X");
      Struct goal = new Struct("permutation",list_struct,x); // permutation([1,2,3],X).

      /**** 3. setup tuProlog engine ****/

      Prolog engine = new Prolog();
      Theory t = new Theory(theory);
      engine.setTheory(t);

      /**** 4. browse all solutions ****/

      Vector<Struct> solutions=new java.util.Vector<Struct>();
      SolveInfo solution = engine.solve(goal);
      if (solution.isSuccess()) {
         solutions.add((Struct)solution.getTerm("X"));
      }
      solution = engine.solveNext();
      while (engine.hasOpenAlternatives()) {
         System.out.println(solution);
         solutions.add((Struct)solution.getTerm("X"));
         solution = engine.solveNext();
      }

      /**** 5. convert back the solution items into plain Java objects ****/

      Vector<LinkedList<Integer>> results = new Vector<LinkedList<Integer>>();
      for (Struct s : solutions) {
         Iterator<Object> struct_iterator = s.listIterator();
         LinkedList<Integer> temp = new LinkedList<Integer>();
         while (struct_iterator.hasNext()) {
            Int i = (Int)struct_iterator.next();
            temp.add(i.intValue());
         }
         results.add(temp);
      }

      // Next performing some operation on results
      ...
   }
}
```

Fig. 1. The `PermutationUtility` class

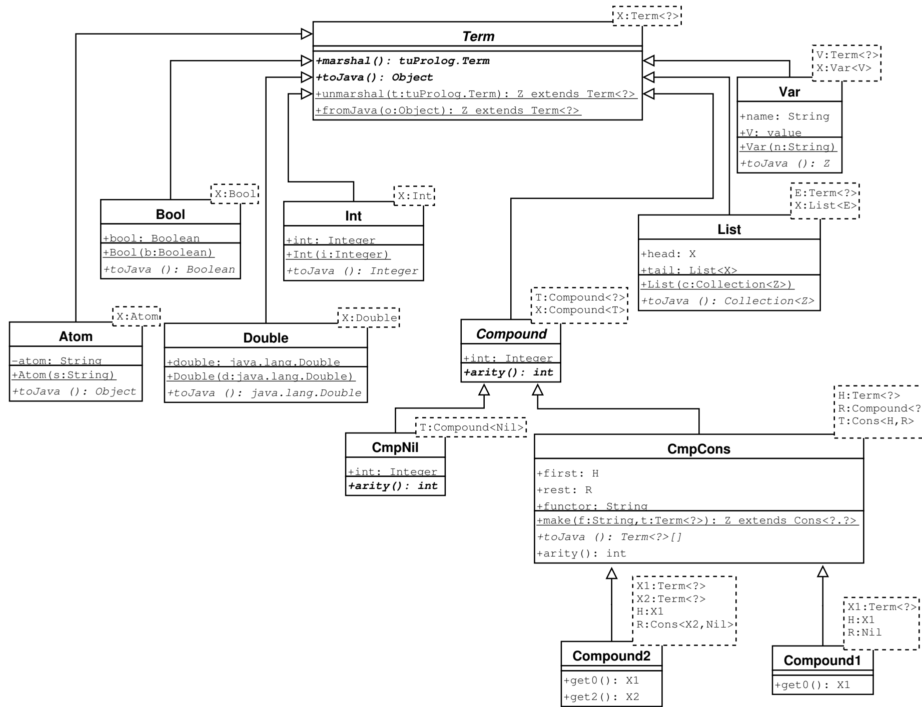

Fig. 2. The `Term` library

computation of a given Prolog query is achieved within a single method call. Among the features that are enabled by the annotation layer we have:

- high degree of customization, since the user can choose which style should be adopted when interacting with the framework (e.g. functional, procedural, relational);
- smart browsing of Prolog solutions, since Prolog methods that model multiple result goals make use of the `Iterable` class to heavily simplify the process of finding all the solutions of a given Prolog query;
- language extension, for Prolog classes can be checked using `javac` (no external tool is required); in this sense our work can be seen as an extension to the Java programing language although the user is not required to change its `javac` compiler in order to take advantage of this layer.

3 Generic API Layer

In this section we describe the generic API layer, devoted at defining an enhanced library for using Prolog in Java.

Every data value in Prolog is a term—and even clauses and goals can be represented as such. We map Prolog terms into a strongly-typed hierarchy of generic classes—see Figure 2. The advantage of using generics for expressing

Prolog terms is in enhancing the static type checking carried out by the `javac` compiler, and in expressing interesting relationships between terms. The term hierarchy is rooted by the generic class `Term<X>`:

```
abstract class Term<X extends Term<?>> {
 tuProlog.Term marshal(){..}
 <Z extends Term<?>> static Z unmarshal(tuProlog.Term t){..}
 <Z> Object toJava(){..}
 <Z extends Term<?>> static Z fromJava(Object o){..}
}
```

Using a recursive pattern exploiting wildcard types [1, 9], the type variable `X` is declared as being itself a term (thus extending `Term<?>`): in this way, we are able to access through `X` the actual type of this term—as described in the following. The class `Term<X>` first defines two methods which are the key mechanism for switching to/from the tuProlog engine representation of a given Prolog term: `marshal()` and `unmarshal()`. They convert a `Term<X>` element into a legacy `tuProlog.Term` and vice-versa, and are used internally by the library. More importantly, the class `Term<X>` defines two additional methods used for creating an object `Term<X>` from a Java object and viceversa.

- `toJava()`; this method converts the receiver `Term<X>` object into a Java object. It should be invoked so as to convert this object into a representation that is more friendly to the user—e.g. to obtain a `java.lang.Integer` out of an integer term.
- `fromJava(Object o)`; this is a static method that converts a Java object into a `Term<X>` object, e.g. instances of `java.util.Collection<E>` are converted into a `Term<X>` representation of Prolog lists.

In Prolog there are three main kinds of terms, each represented by different library classes implementing the above methods: *(i)* constants (either boolean, numeric or literal, called *atoms*), *(ii)* compound terms, and *(iii)* variables. Moreover, we have a special class for lists for they require a special treatment—though they are actually a compound type.

3.1 Constants

A proper mapping is defined for constants into Prolog terms. For example, the class `Atom` is used to represent literal constants

```
class Atom extends Term<Atom> {
  String atom;
  Atom(String val){ ... }
  <Z> String toJava(){ ... }
  ...
}
```

and similarly for classes `Int`, `Double`, and `Bool`—wrapping corresponding values of classes `java.lang.Integer`, `java.lang.Double` and `java.lang.Boolean`. A

```
abstract class Compound<X extends Compound<?>> extends Term<Compound<X>>{
   int arity();
}
class CmpNil extends Compound<Nil> {
   CmpNil() { ... }
   int arity() { ... }
}
class CmpCons<H extends Term<?>,R extends Compound<?>> extends Compound<CmpCons<H,R>> {
   String functor;
   H head;
   R rest;
   CmpCons(String functor, H head, R rest){..}
   <X extends Term<?>,Y extends Compound<?>> static CmpCons<X,Y> make(String f,Term<?>[]){..}
   int arity(){ return 1+rest.arity(); }
}
class Compound1<X extends Term<?>> extends CmpCons<X,Nil> {
   ...
   Compound1(String f, X x) { ... }
   X get0() { ... }
}
class Compound2<X extends Term<?>, Y extends Term<?>> extends CmpCons<X,CmpCons<Y,CmpNil>> {
   ...
   Compound2(String f, X x, Y y) { ... }
   X get0() { ... }
   Y get1() { ... }
}
```

Fig. 3. Data structures for compound terms

generic class type corresponding to a constant Prolog term is a subtype of the root class `Term<X>`, where `X` is instantiated to the generic class type itself, so that e.g. `Atom` is a particular kind of `Term`, namely, a `Term<Atom>` (a `Term` storing an `Atom`). Constant terms can be easily created from standard Java classes either via constructor call or via the `Term.fromJava()` method as follows:

```
Atom a = new Atom("Hello Prolog!");
Int i = new Term.fromJava(4);
```

Similarly, it is possible to retrieve a Java object from a constant `Term` object by exploiting its `toJava()` generic method.

```
String js = a.toJava() // a is an Atom;
Integer ji = i.toJava() // i is an Int;
```

3.2 Lists

A specific mapping is provided for the widely-used Prolog lists, though in Prolog they are a particular case of compound term. Our system represents a Prolog list as an instance of the class `List<X>`:

```
class List<X extends Term<?>> extends Term<List<X>>{
   X head;
   List<X> tail;
   List(X head, List<X> tail){..}
   <Z> List(Collection<Z> c){..}
   <Z> Collection<Z> toJava(){..}
   ...
}
```

A list is represented by a type that is parameterized on the type of its elements (which is itself an instance of `Term<?>`). Since an explicit mapping is provided with Java collection classes, one can write:

```
Collection<Integer> ci = ...;
List<Int> li = new List<Int>(ci);
```

Conversely, every `List<X>` object can be converted into a plain Java collection object by invoking its `toJava()` method:

```
for (Integer i : li.toJava()) {
   System.out.println(i);
}
```

Since `List<X>` is a generic type, it is not possible to e.g. add to the list an element whose type does not match with the expected type (the actual type of the `X` type variable). This means that an object whose type is `List<Int>` must contain only those terms that are of type `Int`. If there are cases in which the content of a given list is not known *a priori*, the wildcard type `List<? extends Term<?>>` can be used instead (the type of a list whose element type is an unknown subtype of `Term<?>`).

3.3 Variables

Prolog variables are represented as instances of the class `Var<X>`:

```
class Var<X extends Term<?>> extends Term<X> {
   String name;
   X value;
   Var(String name) { ... }
   <Z> Z toJava() { ... }
   ...
}
```

Note that a variable is expected to be eventually bound to a term whose type is its type parameter `X`. Differently from previous cases, `Var<Atom>` is e.g. not a subtype of `Term<Var<Atom>>` but rather of `Term<Atom>`. Hence, when we need to specify an atom term that could be either ground or a variable (to be later bound to an atom), we can use type `Term<Atom>`, which is a supertype of both `Atom` and `Var<Atom>`. Since a variable's value is a particular kind of `Term<X>` value, it can be easily converted exploiting its `toJava()` method, as shown below:

```
Var<List<Int>> var = new Var<List<Int>>("X");
...
Collection<Integer> li = var.value.toJava();
```

This allows the user to get a more friendly representation from the result of a given Prolog query.

3.4 Compounds

The most general term in Prolog is the compound term, which is basically made up of a functor and a list of terms. For example, the Prolog query `permutation([1,2,3],X)` is itself a compound term, whose functor is `permutation` and whose list of terms is $\{[1, 2, 3], X\}$—a list followed by a variable. Even Prolog lists are a special case of compound terms whose functor is '.' (so that the list `[1,2,3]` can be represented by the compound `.(1,.(2,.(3,[])))`). Compound terms are represented as instances of the root class `Compound<X>` (whose definition and concrete classes are given in Figure 3). A `Compound` type can be built using a list-type construction, through classes `CmpCons` and `CmpNil`.

The advantages of using generics in expressing compound types is in the fact that the Java compiler can ensure that a given compound type is used in the proper way (accordingly to its declaration) so that e.g. the following code fragment issues a compile-time error:

```
CompCons<List<Int>,CmpCons<Atom,CmpNil>>> c= ... ;
//Accessing the second element...
List<Int> list = c.rest.head; //ASSIGNMENT ERROR!!
```

Classes `Compound1`, `Compound2`, ... are introduced as a syntactic facility for expressing compound types with 1, 2, ... arguments. For instance, type `Compound2<List<Int>,Atom>` is a subtype of `CompCons<List<Int>,CmpCons<Atom,CmpNil>>`, and can therefore be used in place of it.

3.5 Wrapping the tuProlog engine: the generic API layer in action

The generic API layer of our framework provides also a thin wrapper around the tuProlog engine, called `PJProlog`, that deals natively with term classes. This means that, before executing a given Prolog query, the `marshal()` method is invoked on the goal term; the result, which is a plain tuProlog term, is thus passed to the tuProlog engine that computes the solution for that Prolog query. Conversely, once the tuProlog engine has built an instance of the `SolveInfo` class representing the solution of the above query, that instance is wrapped by `PJProlog` with an instance of the generic class `Solution<G,S>` (`G` and `S` are instantiated to the types of the goal and the solution terms, respectively). The `Solution` class of the framework can be seen as the generic counterpart of the `SolveInfo` class of the tuProlog engine; each time a term is retrieved using the `Solution` class, the corresponding tuProlog object of the underlying

```
public class PermutationUtility {
   String theory = "remove([X|Xs],X,Xs).\n"+
                   "remove([X|Xs],E,[X|Ys]):-remove(Xs,E,Ys).\n"+
                   "permutation([],[]).\n"+
                   "permutation(Xs,[X|Ys]):-remove(Xs,X,Zs),permutation(Zs, Ys).\n";

   public static void main(String[] args) throws TuPrologException {
      LinkedList<Integer> l=new LinkedList<Integer>();
      l.add(1);
      l.add(2);
      l.add(3);

      /**** 1. build the goal ****/

      List<Int> list = new List<Int>(l);
      Var<List<Int>> x = new Var<List<Int>>("X");
      Compound2<List<Int>,Var<List<Int>>> goal;
      goal = Compound.make("permutation",new Term<?>[]{list,x});

      /**** 2. setup tuProlog engine ****/

      PJProlog engine = new PJProlog();
      Theory t = new Theory(theory);
      engine.setTheory(t);

      /**** 3. browse/convert all solutions ****/

      Vector<Collection<Integer>> solutions=new java.util.Vector<Collection<Integer>>();
      for (Solution<?,Compound2<List<Int>,Var<List<Int>>>> si : engine.solveAll(goal)) {
         List<Int> li = si.getSolution().get1(); //this is equivalent to si.getTerm("X");
         solutions.add(li.toJava());
      }

      /**** 4. perform some operation on the result items ****/
      ...
   }
}
```

Fig. 4. The `PermutationUtility` class exploiting the generic API layer

`SolveInfo` class is unmarshalled (via the `Term.unmarshal()` factory method) and then returned back to the user. Moreover the engine wrapper defines a method, `solveAll`, that dramatically simplifies the process of finding all solutions of a given Prolog query; this is possible since `solveAll` returns an object of type `Iterable<Solution<?,?>>` that can thus be used inside a `for-each` statement.

The code in Figure 5 shows how the `PermutationUtility` class could be rewritten taking advantage of the generic API layer of the framework. It can be seen how much simpler it is now to convert a `Collection` object into a `Term` object and how expressive the resulting generic code is with respect to the one showed in Figure 1. Moreover, the process of finding all the solutions of a given Prolog is realized by just a `for-each` loop, thanks to the `solveAll()` method exposed by the engine wrapper.

style	*signature*
R	Compound1<List<Int>> perms(@HIDE List<Int> l, Var<List<Int>> x)
F	List<Int> perms(List<Int> l)
B	boolean perms(List<Int> l, Var<List<Int>> x)

Table 1. Differences between (R)elational, (F)unctional and (B)oolean Prolog methods

@INPUT	@OUTPUT	@GROUND	*Type*
yes	no	yes	List<Int>
no	yes	yes	Var<List<Int>>
yes	yes	yes	Term<? extends List<Int>>
yes	no	no	List<? extends Term<Int>>
no	yes	no	Var<? extends List<? extends Term<Int>>>
yes	yes	no	Term<? extends List<? extends Term<Int>>>

Table 2. Mapping between term and Java types

4 Annotation Layer

The annotation layer defines some Java annotations for explicitly marking classes and methods as Prolog classes/methods. A Prolog class is an abstract class (or an interface) that is linked to a valid Prolog theory and that defines one or more Prolog methods. Conversely, a Prolog method is an abstract (or interface) method that is linked to a specific clause of the defining Prolog class. When a Prolog method is being invoked, the framework handles the method call, *(i)* instantiating a tuProlog engine, *(ii)* mapping method parameters to a given term goal, *(iii)* evaluating the Prolog query, and *(iv)* possibly returning a `Iterable` datastructure back to the user. The user can easily customize the way in which each of the above operations is performed simply by changing the attributes of the custom annotations. The most compelling feature of the annotation layer is that it hides all the details of the underlying Prolog engine, providing an extension to the Java programming language featuring an explicit mapping from object-oriented programming concepts (e.g. classes, methods) into logic programming concepts (e.g. theories, queries).

4.1 The `@PrologClass` annotation

The `@PrologClass` annotation is used to link a class to a Prolog theory. The `@PrologClass` annotation can be applied to both abstract class and interface types. The attribute `clauses` of the `@PrologClass` annotation is used to specify the contents of the Prolog theory referred by the Prolog class. When a class (or an interface) is given a Prolog theory via the `@PrologClass` attribute, all Prolog methods within that class will rely on it. In the example in Figure 5 a `@PrologClass` attribute has been used to specify the body of the Prolog theory exploited by all the Prolog methods of `PermutationUtility`.

4.2 The `@PrologMethod` annotation

The `@PrologMethod` annotation is used to link a method to a Prolog query. A Prolog method can be seen as an accessor, since it exposes (in a Java-like fashion) certain properties of the Prolog theory in the defining Prolog class. The `@PrologMethod` annotation defines several attributes that control the way in which the Prolog theory is exposed to the user.

- `String link`; this attribute is used for building the underlying Prolog query. A link string must be expressed in the form `clause(argList)` where:
  - `clause` is the name of a clause in the theory associated to the current class;
  - `argList` is a list of items in the form $\$K_0, \$K_1, ...\$K_n$ where a given item $\$K_i$ is an integer value defining which Prolog method argument maps into the ith argument of `clause`. Optionally, one special item `$0` can be used for denoting the argument of `clause` that will be mapped as return type of the method.

  As an example, if the `link` attribute of a given Prolog method `m` is given the value `permutation($1,$2)`, an invocation of `m` triggers a Prolog query that is built upon the `permutation` clause, and in which a mapping is defined so that the first (resp. the second) argument of `m` is used as the first (resp. the second) argument of the `permutation` clause.
- `boolean hasMultipleOutput`; indicates whether the Prolog query triggered by a call to this Prolog method could have multiple solutions or not. A multiple result is always exposed to the user via the `Iterable` interface, so that the user can easily browse all solutions to a given query in a single `for-each` statement. A single result is instead obtained directly with a `Term` object, and only the first solution of the Prolog query is returned.
- `boolean exceptionOnFailures`; when set, it causes an exception of type `NoSolutionException` to be thrown whenever the Prolog query triggered by a call to this Prolog method fails.
- `PrologInvocationKind style`; this attribute is used to choose the interaction style that should be used. Three values are possible for this attribute, which influence the signature's argument and return types as shown in Table 1:
  - `RELATIONAL`: a call to a relational Prolog method yields an object which is a `Compound` term: all the arguments (excepting those marked with the `@HIDE` annotation) are copied in the result. This style of invocation is useful whenever the user is interested in getting the whole solution term to a given Prolog query (e.g. because most arguments represent output information).
  - `FUNCTIONAL`: a call to a functional Prolog method yields the object representing the unifying term for the argument marked as `$0` in the `link` attribute. This style of invocation is useful whenever the `link` clause has only one term acting as output, and the user is only interested in getting the substitution for that term.

```
@PrologClass (
   clauses = {"remove([X|Xs],X,Xs).",
              "remove([X|Xs],E,[X|Ys]):-remove(Xs,E,Ys).",
              "permutation([],[]).",
              "permutation(Xs,[X|Ys]):-remove(Xs,X,Zs), permutation(Zs,Ys)."})
public abstract class PermutationUtility {

  @PrologMethod (link="remove($1,$2,$0)",
                 style=PrologInvocationKind.FUNCTIONAL)
  abstract @GROUND List<Int> remove(@INPUT @GROUND List<Int> c1,
                                    @INPUT @GROUND Int i);

  @PrologMethod (link="permutation(@1,@2)",
                 multipleOutput=true,
                 style=PrologInvocationKind.RELATIONAL)
  abstract @GROUND Iterable<Compound1<List<Int>>> perms(@HIDE @INPUT @GROUND List<Int> c1,
                                                         @OUTPUT @GROUND Var<List<Int>> c2);

  public static void main(String[] args) throws Exception{
    PermutationUtility pu = Java2Prolog.newInstance(PermutationUtility.class);
    java.util.Collection<Integer> l=new java.util.LinkedList<Integer>();
    l.add(1);
    l.add(2);
    l.add(3);
    Var<List<Int>> x = new Var<List<Int>>("X");
    List<Int> list = Term.fromJava(l);
    //perform some operations on the results
    for (Compound1<List<Int>> compound : pu.perms(list,x)) {
      Collection<Integer> ci = compound.get0().toJava();
      System.out.println(Collections.max(li));
    }
  }
}
```

Fig. 5. The `PermutationUtility` class exploiting the annotation layer

- `BOOLEAN`; a call to a boolean Prolog method yields a boolean result (`true` if the query has at least one result, `false` otherwise). This style of invocation is minimal and should be used whenever the user is only interested in the success/failure of a given Prolog query.

In the example of Figure 5 a `@PrologMethod` attribute has been used to link the method `perms()` to the predicate `permutation/3` of the theory associated to the class. Since the selected style for the above method is relational, there is a one-to-one mapping between the terms in the `permutation` clause and the arguments of the `perms` method. However, since the first argument is annotated with `@HIDE`, only the second argument should be copied in the result `Compound` object, whose type is hence `Compound1<List<Int>>` instead of `Compound2<List<Int>,List<Int>>`.

4.3 `@INPUT`, `@OUTPUT` and `@GROUND` arguments

We have seen how the `@PrologMethod` defines a mapping between a Java method and a given Prolog goal. Such a mapping is involved primarily because of the role played by terms in a Prolog goal, which is very different from the one played by arguments in a Java method. A term in a Prolog clause can act either as

input or as output without any distinction. This is a very different situation with respect to Java methods, in which arguments can act only as input. The framework expects an argument of a Prolog method to have a certain generic signature with respect to the set of annotations attached to that argument. Every argument of a Prolog method can in fact be marked with one of the following annotations (that can be combined together in any way):

- `@INPUT`: the marked argument acts as an input; this means that the actual type of the argument may be a subtype of `Term<?>`.
- `@OUTPUT`: the marked argument acts as an output; this means that the actual type of the argument may be a subtype of `Var<?>`.
- `@GROUND`: the marked argument is ground; this means that the actual type of the argument must be a subtype of `Term<X>`, where `X` is itself the type of a ground (non-variable) term.

Consider for example a Prolog clause taking as input a term like `p(X)`, where `X` is expected to hold a list of integer values; we can have many different Java mappings depending on the role played by the term in the clause, as depicted in Table 2. These mappings have been suggested by the subtyping rules induced by the generic `Term` hierarchy. As an example, consider the case of a Prolog method `m` declaring an input/output ground argument of type `List<Int>`: then the formal argument is declared to be of type `Term<? extends List<Int>>` (see Table 2), meaning that either an object of type `Var<List<Int>>` (output) or an object of type `List<Int>` (input) could be passed to `m`.

4.4 The annotation layer in action

The code in Figure 5 shows how the `PermutationUtility` class could be rewritten taking advantage of the framework proposed in this paper. The `PermutationUtility` class is now `abstract` and a Prolog theory has been attached to it via its `@PrologClass` annotation as a sequence of strings. Two Prolog methods have been added to the class: `remove()` and `perms()`, respectively. The Prolog method `remove()` adopts a functional style so that an object of type `List<Int>` is returned; this is possible since all the arguments in this Prolog method are declared to be `@INPUT`. In the `link` attribute, the third argument is assigned item `$0`, hence this is mapped to the return value of the method. On the other hand, the Prolog method `perms()` adopts a relational style so that an object of type `Iterable<Compound1<List<Int>>>` is returned (since the attribute `hasMultipleOutput` is set, the return type is a subtype of `Iterable`). Here, the select attribute is used to specify which arguments of the Prolog method should appear in the result `Compound` object.

Inside the method `main()`, an instance of the Prolog class is created exploiting the `newInstance()` factory method provided by the framework. This method dynamically wraps the Prolog class passed as argument and returns the proxy object `pu` to the user. The object `pu` can thus be used as any other Java object: the method call `pu.perms(list,x)` is captured by the framework which, in

turn, triggers the resolution of a Prolog query to a tuProlog engine (the query is `permutation([1,2,3],X)`); as it can be seen, all these details are hidden to the user. The code has been significantly simplified with respect to the one showed in Figure 1, thanks to the builtin marshaling/unmarshaling support and to the `for-each` loop for finding solutions.

5 Some Advanced Features

The framework can also be added with several advanced features to further improve usability: here we discuss automated marshalling/unmarshaling support for user-defined classes and compile-time verification of annotated classes.

5.1 Support for user-defined classes

In the previous sections it has been shown how automated capabilities can be provided for switching between the relational and the object-oriented view of some given data type. For example, an object of type `List<Int>` can be converted to an object of type `Collection<Integer>`, and viceversa. In addition to these basic marshaling/unmarshaling mappings, an extension mechanism is provided through the `@Termifiable` annotation. If a user-defined class is annotated with the `@Termifiable` annotation, the framework will be able to automatically build a Prolog representation from that class (and viceversa). Note that the user is not required to implement any method for instrumenting the framework in order to perform this kind of conversion.

In the first step of the conversion process, we rely on the *Java Introspection API* in order to find all suitable properties (those with getters/setters methods) in a given user-defined class. Then, each property is turned into the special compound term `property(Name,Value)`, where `Name` and `Value` are the Prolog terms representing the name and the value of the property to be marshalled, respectively; Finally, a compound term is created whose functor is the name of the class to be marshalled and whose term list is the list of properties for the arguments. Assuming the user defines the class below:

```
@Termifiable class Counter {
  Integer value;
  Integer getValue() {
    return value;
  }
  void setValue(Integer v) {
    value = v;
  }
}
```

The framework translates a `Counter` object whose `value` is `12` into the compound Prolog term `'Counter'(property('value',12))`. The above compound term is then represented as an object of type `CmpCons<CmpCons<Atom,CmpCons<Int,CmpNil>>,CmpNil>`. The conversion from Prolog compound to user-defined class is as follows:

1. When unmarshalling a Prolog compound term, the the system classloader is asked for a class whose name is the functor of the compound term to be unmarshalled.
2. If such a class is found and that class is marked with the `@Termifiable` annotation, a new instance of that class is created. Otherwise the framework assumes this is a plain Prolog compound term.
3. The property values are then set on the instance created at step 2. This is possible since a compound term corresponding to a termifiable class always hold a list of compound terms in the form `property(Name,Value)`.

For example, when unmarshalling the compound term `'Counter'(property('value',12))`, we first see that `Counter` is indeed a class annotated with the `@Termifiable` annotation; to perform the unmarshalling it will suffice to *(i)* create a brand new `Counter` instance and *(ii)* set its `value` property to `12` (again, this is achieved through Introspection).

5.2 Compile-time checking of Prolog classes

Java 6 introduced a key mechanism for extending the `javac` compiler with custom annotation processors. If the compiler sees a subclass of `javax.annotation.processing.Processor` in the classpath, it automatically instantiates that class and uses it for processing custom annotations on the Java source code, at compile-time. Our framework can exploit this feature in order to define a custom annotation processor that can be used, together with `javac`, as a compiler for classes annotated with Prolog code. The annotation processor verifies that the whole code of a class is compliant with the framework in many ways; in particular, an error is raised when one of the following situations occurs:

- the theory referred by a Prolog class (via its `@PrologClass` annotation) is not a valid Prolog theory;
- the `@PrologClass` annotation is applied to a type that is neither an abstract class nor an interface type;
- a Prolog method refers to a non-existing clause via the `link` attribute of its `@PrologMethod`;
- a mapping for the return value of a functional Prolog method is not specified (via the `link` attribute);
- the formal argument types of a given Prolog method do not match their annotations (accordingly to the mappings in Table 2);
- the return type of a given Prolog method does not match either the method parameters' annotations or the selected invocation style;
- the return type of a multiple-output Prolog method is not a subtype of `Iterable<?>`;
- a Prolog method whose `@PrologMethod` annotation features the `exceptionOnFailures` has not been declared to throw exceptions of type `NoSuchSolutionException`.

Assuming that the framework jarfile is in the classpath, the annotation processor will be automatically detected by `javac` and used whenever a source file containing custom annotations is encountered. Since, the annotation library is rather rich, the possibility of checking for the well-formedness of a class definition—and reporting meaningful error messages—is crucial for making our framework a usable tool.

6 Conclusions

In this paper we showed a framework that significantly improves the seamless integration of Prolog code into Java applications, exploiting tuProlog technology. We believe that our work will encourage those programmers who are familiar with Java mainstream programming to easily incorporate declarative features into their programs. The framework is structured in a compositional way, since Prolog integration is achieved through two layers; moreover, the high degree of customization provided by the annotation layer makes it possible for the user to choose among many different interoperability paradigms. It should be noted that these two layers are not separate implementations: the key benefits of our work are achieved by their joined exploitation! In fact, the possibility of expressing rich (generic) types for the terms interfacing Java and Prolog is crucial for relying on annotations to automatically generate suitable code for executing Prolog queries.

6.1 Related works

There are several non-Java Prolog engines providing a Java interface such as SICSTus Prolog [10, 11], SWI-Prolog [12, 13] and k-Prolog [14, 7]; moreover, several Prolog engines have been written entirely in Java such as JLog [3] and Minerva [15]. Though all the above solutions enable the execution of Prolog queries from a Java program, they suffer from similar problems of tuProlog: *(i)* poor integration with the Java language, *(ii)* the need of manually performing marshalling/unmarshalling operations for switching between user objects and Prolog terms, and so on. Perhaps, JLog can be regarded as the only exception since it provides support for automatic marshalling/unmarshalling between Prolog terms and Java objects (as a service provided by the JLog engine, namely `jTermTranslation`). We believe that an approach like the one we propose in this paper can be exploited fruitfully in conjunction with all the above Prolog engines/interfaces to take advantage of the integration support described in this paper.

Japlo [4] is an hybrid Java and Prolog language that has been built on top of the JLog engine. The aim of Japlo is to bring declarative features into the Java programming language; this is done at the language level: Japlo is an extension of the Java programming language providing some declarative features such as strongly-typed Prolog list declarations, theory definition, and so on. A Japlo code mimicking our `PermutationUtility` sample is as follows:

```
rule rem(Int{} _={x|xs},Int x,Int{} xs) {}
rule rem(Int{} _={x|xs},Int e,Int{} _={x|ys}) {
  rem(xs,e,ys);
}
rule permutation(Int{} _={}, Int{} _={}) {}
rule permutation(Int{} xs, Int{} _={x|ys}) {
  Int{} zs;
  rem(xs,x,zs);
  permutation(zs,ys);
}
...
Int{} list = {1,2,3};
Int{} res;
for (permutation(list,res)) {
  System.out.println(res);
}
```

Hence, Japlo allows the user to define *rules*—a sort of method-like abstraction that is used to model Prolog clauses. The body of a clause is specified in terms of the Japlo language as if it were the body of an ordinary Java method. The main differences between our work and Japlo are compatibility and deployment requirements: Japlo comes with its own compiler which translates a Japlo program into plain Java bytecode that can thus be executed on a legacy Java Virtual Machine. On the other hand, we provide a set of custom Java annotations and a library that together provide a language extension without the drawbacks of introducing language incompatibilities. Moreover, Japlo requires the Java programmer to adopt a paradigm-oriented view, since a Japlo program is mainly structured in rules; instead we do not truly mix together Java and Prolog features. In this sense our framework can be seen as a way of building Java-based views over Prolog theories without the burden of writing boiler-plate code dealing with a true Prolog engine.

The idea of using annotations for extending the Java language derived mostly from *AspectJ/AspectWerkz* [16][17], which are very popular aspect-oriented extensions of the Java programming language. AspectJ uses Java annotations for declaring aspects, pointcuts, and advices. For instance, in a way similar to our work, AspectJ associates aspects and pointcuts to classes and methods as follows:

```
@Aspect
public class Logger {
  @Pointcut("call(* java.util.List.*(..))")
  void traceListCalls() {
     System.out.println("java.util.List called");
  }
  ...
}
```

Similarly, in [18] a framework is described that supports *pluggable type systems* in the Java programming language. This framework heavily relies upon Java annotations in order to define custom constraints on Java types. These techniques

based on annotations, along with the work presented in this paper, show that the annotation mechanism of Java has a great potential in flexibly tuning the programming language according to needs of the application at hand.

6.2 Future works

Future works will be focused in several directions. On the one hand, we will improve the ISO compliance of the annotation layer, providing support for those features such as *DCG*, *modules*, *exceptions*, which are not available in the current implementation of our framework. On the other hand, we do think that our work could be used as a basis for further investigation on the relationships between object-oriented/declarative programming languages. Among the many aspects to be considered, we aim at: *(i)* modeling inheritance between Prolog classes, *(ii)* applying dynamic dispatching techniques for Prolog methods, and *(iii)* representing the state of a given Java object as a Prolog theory. Finally, we will evaluate the performance overhead introduced by our approach, possibly studying optimization techniques.

References

1. Joy, B., Gosling, J., Steele, G., Bracha, G.: The Java Language Specification (Third Edition). Addison-Wesley, New York (2005)
2. Sun Microsystem: J2SE 6.0. http://java.sun.com (2007)
3. JLog team: JLog – Prolog in Java. http://jlogic.sourceforge.net/ (2002)
4. Espk, M.: Japlo: Rule-based programming on java. **12**(9) (2006) 1177–1189
5. Omicini, A., Natali, A.: Object-oriented computations in logic programming. In Tokoro, M., Pareschi, R., eds.: Object-Oriented Programming. Volume 821 of LNCS., Springer-Verlag (1994) 194–212 8th European Conference (ECOOP'94), Bologna, Italy, 4–8 July 1994. Proceedings.
6. Denti, E., Omicini, A., Ricci, A.: Multi-paradigm java-prolog integration in tuprolog. Sci. Comput. Program. **57**(2) (2005) 217–250
7. Kino, N.: Jipl: Java interface to prolog. http://www.kprolog.com/jipl/ (2005)
8. tuProlog Team: tuProlog at SourceForge. http://sourceforge.net/projects/tuprolog/ (2002)
9. Igarashi, A., Viroli, M.: Variant parametric types: A flexible subtyping scheme for generics. ACM Transactions on Programming Languages and Systems **28**(5) (2006) 795–847
10. the SICStus Prolog Team: Sicstus Prolog User's manual http://www.sics.se/isl/sicstuswww/site/index.html.
11. the SICStus Prolog Team: Documentation for PrologBeans http://www.sics.se/isl/sicstuswww/site/index.html.
12. the SWI-Prolog Team: SWI-Prolog Official Website http://www.swi-prolog.org/.
13. the SWI-Prolog Team: JPL: A bidirectional Prolog/Java interface http://www.swi-prolog.org/.
14. the k Prolog Team: K-Prolog Official Website http://www.kprolog.com/.
15. the Minerva Team: Minerva Official Website http://www.ifcomputer.co.jp/MINERVA/.

16. the AspectJ Team: The aspectj 5 development kit developer's notebook. http://www.eclipse.org/aspectj (2005)
17. Bon, J.: Annotation-driven AOP for Java. In: Annual European Conference on Java and Object-Oriented Software Engineering. (2004) http://aspectwerkz.codehaus.org.
18. Andreae, C., Noble, J., Markstrum, S., Millstein, T.: A framework for implementing pluggable type systems. In: OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, New York, NY, USA, ACM Press (2006) 57–74