Integrating Java and Prolog through Generic Methods and Type Inference

Maurizio Cimadamore, Mirko Viroli
DEIS, Cesena
Alma Mater Studiorum – Università di Bologna
via Venezia 52, 47023 Cesena, Italy
{maurizio.cimadamore,mirko.viroli}@unibo.it

ABSTRACT

P@J is a framework, based on the tuProlog open-source engine, allowing Prolog code to be used as possible implementation of a Java method: Java annotations are used for specifying all the necessary information to fill the Java-Prolog gap. This framework is useful to inject a declarative, logic-based paradigm into mainstream object-oriented programming, so as to easily code functionalities related to automatic reasoning, adaptivity, and conciseness in expressing algorithms.

In this paper, an extension of P@J is presented which improves the invocation technique for such Prolog-implemented methods. Java type inference of generic method calls is intensively used to automatically infer all the necessary paradigm mismatch information: this results in an elegant and concise invocation style, which further reduces the gap between Prolog goal satisfaction and Java method invocation. This new approach inspires some interesting applications: we show examples related to the implementation of abstract data types and parsers for context-free grammars.

Categories and Subject Descriptors: D.1.5 [Programming Techniques]: Object-oriented Programming; D.1.5 [Programming Techniques]: Logic Programming; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs

General Terms: Algorithms, Languages, Design

Keywords: Generics, Java, Multiparadigm, Prolog, Wildcards

1. INTRODUCTION

The growing complexity in modern software systems calls for developing applications that achieve their tasks in dynamic, open, and unpredictable environments. This can be achieved by embedding automatic and complex reasoning, adaptiveness, and reconfiguration capabilities into mainstream programming platforms, such as Java or .NET: however, handling these aspects through standard object-oriented languages is often difficult. Instead, 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. Such proposals either attempt at joining the two paradigms [6, 12], or simply provide an interface library for accessing a Prolog engine from a mainstream object-oriented language [9, 5, 11]. Both solutions have however drawbacks: in the first case, the conceptual integrity of the programming model is sacrificed since the new language is typically more complex, thus making application development an harder task; in the second case, there is no true language integration, and some "boilerplate code" has to be implemented each time to fix the paradigm mismatch. What we promote, hence, is an extension that better trades off language integration and programming integrity.

In [4] we presented P@J, a framework built on top of tuProlog open-source project [14] that exploits Java generics and annotations to promote seamless exploitation of Prolog programming in Java. P@J is structured in two stacked layers, each one further reducing the semantic gap between Java and Prolog. First, the Generic API Layer is introduced to model Prolog terms in Java, so that user code interfacing with the tuProlog engine can automatically switch between the object-oriented and logic-oriented representation of data. Second, the Annotation Layer is added acting as a true Prolog-based extension of Java programming: it provides custom Java *annotations* to be used for embedding Prolog theories within Java classes, so as to specify Prolog code as a possible implementation of given Java methods. This is done by annotating an abstract method with the corresponding Prolog code, and with further additional information to bridge the gap between logic-oriented and object-oriented paradigms: P@J automatically and transparently provides an implementation for the method by exploiting the Prolog engine. Although quite expressive, the burden for such annotation is still significant, and this might limit the code points where method invocations are redirected to Prolog.

To solve this problem, this paper evaluates an alternative Annotation Layer with a different invocation style, relying on generic methods and taking advantage of Java type inference at the call side. On the one hand, all the necessary

SAC'08, March 16-20, 2008, Fortaleza, Ceará, Brazil.

information to fill the gap between Prolog goal satisfaction and Java method invocation is extracted from the method declaration, which naturally embeds this information in the form of method type variables. On the other hand, type inference is used to check correctness of a method invocation and properly type the computation result: although the generic type system of Java 6.0 is extensively used, little knowledge of this is hence actually required to the method users. This approach makes the annotation technique required to bridge the two paradigms much more intuitive and elegant, promoting a more pervasive exploitation of this programming mechanism.

This framework inspires some example applications, which emphasize the ability of this extended version of `P@J` to act as a useful tool to inject declarative, logic-based specifications into Java programming. First, it can be used to automatically derive implementation of data structures and their algorithms out of abstract data type specifications expressed in a declarative style; second, it can be used to generate parsers for context-free grammars, paving the way towards a complete tool for programming interpreters and compilers, in the style of e.g. `JavaCC`[1].

The remainder of the paper is organized as follows. Section 2 briefly describes the existing Generic API Layer of `P@J`, Section 3 describes the new Annotation Layer based on type inference in method calls, Section 4 describes examples, and finally Section 5 concludes providing final remarks.

2. THE GENERIC API LAYER

Every data value in Prolog is a term—and even clauses and goals can be represented as such—with syntax:

$$t ::= a|p|V|f(t_1,\ldots,t_n)|[t_1,\ldots,t_n]$$

A term t is either an atom a (an unstructured literal), a primitive value p (an integer, a boolean, and so on), a logic variable V (a variable that can be bound to a value during computation, expressed as a literal starting with a capital letter), or a compound term where f is the functor name and each t_i is a term ($n > 0$). Terms can also be lists of the kind $[t_1,\ldots,t_n]$ (or $[t_h|t_t]$ where t_h is the head and t_t is the tail), which Prolog implementations handle as special cases of compound terms.

The main problem that `P@J` faces is to map these concepts in a very precise way into an object-oriented structure. To understand the requirements when doing so, consider e.g. the simple Prolog predicate `length(List,Size)`, which holds when `List` is a list of elements and `Size` is an integer equals to the size of the list. Once properly implemented, predicate `length/2` can be queried in different ways, e.g.:

```
?-length([1,2,3],4). -> no
?-length([1,2,3],X). -> X/3
?-length(X,3). -> X/[_,_,_]
?-length(X,Y). -> X/[],Y/0;X/[_],Y/1;X/[_,_],Y/2;...
```

In particular, the user can ask *(i)* whether the length of `[1,2,3]` is 4 (Prolog replies no), *(ii)* what is the length of `[1,2,3]` (Prolog replies with primitive value 3), *(iii)* for a list whose length is 3 (Prolog replies with the most general list, a list of three anonymous variables `[_,_,_]`), and finally *(iv)* for two terms in the `length` relation (Prolog iteratively provides all replies). Note that arguments to `length/2` can hence be either inputs or outputs, depending on whether a non-variable term or a variable is passed. The ability of supporting this peculiar feature is not a mere programming mechanism of Prolog, but it is a core difference between the relational and functional paradigms: it is a main reason why many computations are more easily and elegantly expressed in Prolog than in Java, and hence it is a crucial aspect of Java-Prolog integration.

Accordingly, `P@J` introduces a strongly-typed hierarchy of generic classes, which enhances the static type checking carried out by the `javac` compiler, and flexibly expresses key relationships between terms: this allows us to support a practical annotation layer as described in the following sections. Class `Term<X>` is the root of the hierarchy of terms.

```
abstract class Term<X extends Term<?>>{..}
```

Using a recursive pattern exploiting wildcard types [10, 8], type variable `X` is used to reify the type of the actual content of the term as shown in the following set of definitions:

```
class Atom extends Term<Atom>{..}
class Int extends Term<Int>{..}
class Double extends Term<Double>{..}
class List<X extends Term<?>> extends Term<List<X>>{..}
...
class Var<X extends Term<?>> extends Term<X>{..}
class Comp<X extends Term<?>> ..
```

hence, a `Term<Int>` will be a term keeping an `Int`, `Term<Double>` a `Double`, `Term<List<Int>>` a `List<Int>`, and so on. On the other hand, variables are handled differently: generic class `Var<X>` is a wrapper for a term with type `X` and is defined as a straight subtype of `Term<X>` (instead of `Term<Var<X>>`). As a result of this careful design choice, the type `Term<Atom>` is a common supertype of both `Atom` and `Var<Atom>`: namely, when an argument to a method needs to be either a logical input or output, it can be given type `Term<T>`, so that one can pass either an actual term `T` (input), or a variable `Var<T>` (output) which will hold the result term (of type `T`) after computation is over. Hence, a general solution for the signature of method `length()` is the following:

```
boolean length(Term<? extends List<?>> list,
               Term<Int> size)
```

This signatures expresses that *(i)* `list` is actually a term containing any list and `size` is an integer term, and *(ii)* both can be either inputs or outputs—wildcard "`? extends`" is used for it allows covariance of the argument type, so that e.g. a `Term<List<Int>>` could be passed [8].

The hierarchy of terms is completed by dealing with compound terms through class `Comp<X>`, `CmpNil` and `CmpCons<H,R>` as follows:

```
abstract class Comp<X extends Comp<?>>
         extends Term<Comp<X>>{..}
class CmpNil extends Comp<CmpNil>{..}
class CmpCons<H extends Term<?>,R extends Comp<?>>
        extends Comp<CmpCons<H,R>>{..}
class Comp1<X0 extends Term<?>>
        extends CmpCons<X0,CmpNil>{..}
class Comp2<X0 extends Term<?>, X1 extends Term<?>>
        extends CmpCons<X0,CmpCons<X1,CmpNil>>{..}
```

[1] `https://javacc.dev.java.net/`

```
public abstract class PermutationUtility {

  @PrologMethod (
     predicate="permutation(@X,-!Y)",
     signature="(X)->{Y}",
     types={"List<Int>","List<Int>"},
     clauses={"any([X|Xs],X,Xs).",
              "any([X|Xs],E,[X|Ys]):-any(Xs,E,Ys).",
              "permutation([],[]).",
              "permutation(Xs,[X|Ys]):-any(Xs,X,Zs), permutation(Zs,Ys)."})
  public abstract Iterable<List<Int>> permutations(List<Int> list);

  public static void main(String[] args) {
    PermutationUtility pu = PJ.newInstance(PermutationUtility.class);
    java.util.Collection<Integer> l=java.util.Arrays.<Integer>asList(new Integer[]{1,2,3});
    for (List<Int> p : pu.permutations(Term.fromJava(l))) {
      System.out.println(p.toJava());
} } }
```

Figure 1: The `PermutationUtility` class exploiting the existing `P@J` annotation layer

A compound term is basically a tuple of terms of any length, e.g., it can have arity two and orderly contain a `List<Int>` and an `Atom` as in term `p([1,2],'a')`. Hence, `P@J` provides a list-like construction mechanism for the type parameter `X`, through classes `CmpCons` and `CmpNil`, as in the following case:

```
CmpCons<List<Int>,CmpCons<Atom,CmpNil>> c= ... ;
List<Int> first = c.head;      //OK!!
Number   second = c.rest.head; //ASSIGNMENT ERROR!!
```

Variable `c` is declared to be a compound term with two arguments of type `List<Int>` and `Atom`, hence assignment to `second` can be checked as wrong. Classes `Comp1`, `Comp2` (and so on) are introduced as a syntactic facility for expressing compound types with 1 and 2 arguments. For instance, type `Comp2<List<Int>,Atom>` is a subtype of `CmpCons<List<Int>,CmpCons<Atom,CmpNil>>`, and can therefore be used in place of it.

3. AN ANNOTATION LAYER BASED ON GENERIC TYPE INFERENCE

The main idea of having an annotation layer in `P@J` is to define some Java annotations for explicitly marking methods as being associated to some Prolog code: basically, an abstract method can be annotated with information on a Prolog theory defining its behavior—*Prolog methods* henceforth. From an abstract level, this forms a specification for the intended behavior of the method: most importantly, the `P@J` framework exploits Java Dynamic Proxy Classes [1] (*proxies* in the following) to provide facilities for automatically implementing that class, so that invoking the method actually results in calling the Prolog engine. As an example, an abstract `permutation()` method taking a `List<Int>` and returning an iterator for all its permutations `Iterable<List<Int>>` can be annotated with this simple Prolog implementation:

```
any([X|Xs],X,Xs).
any([X|Xs],E,[X|Ys]):-any(Xs,E,Ys).
permutation([],[]).
permutation(Xs,[X|Ys]):-any(Xs,X,Zs),
                        permutation(Zs, Ys).
```

The `P@J` framework takes care of automatically creating the necessary bridge code, thus implementing the method through the tuProlog engine.

A mapping between a Prolog method `m` and a predicate of the form $\texttt{p}(\texttt{t}_1, \texttt{t}_2, ..., \texttt{t}_n)$ is fully specified when the following elements are identified: *(i)* `p` and `n` (namely, the *name* and the *arity* of the predicate), *(ii)* the "logical" role of each term $\texttt{t}_i$ (e.g. a term could act as an input or an output to the Prolog engine), *(iii)* how each term $\texttt{t}_i$ is mapped into the signature of `m` (e.g. a term could be mapped into one of `m`'s arguments) and *(iv)* the Java types that can be associated to each term $\texttt{t}_i$.

The existing Annotation Layer as described in [4], defines a `@PrologMethod` annotation with attributes that precisely describes the above properties. The code in Figure 1 shows how this layer works, through a class `PermutationUtility` for retrieving all permutations of a given Java list. The abstract method `permutations()` has the signature one would use in an object-oriented context: its `@PrologMethod` annotation is used *(i)* to specify the intended behavior as a Prolog code, and *(ii)* for properly linking the method to the predicate `permutation/2`. The `predicate` attribute is set to `permutation(@X,-!Y)`: `X` is the input list and it should be ground (ISO Prolog notation `@`), while `Y` is an output eventually bound to a ground term (`P@J` notation `-!`)—see more on this attribute in [4]. The `signature` attribute is set to `(X)->{Y}`: `X` will be the only argument of the method, while `Y` will be the return value of the method, and will be accessed through an iterator. Hence, assuming that `permutations()` is invoked with list `[1,2,3]` as argument, the `P@J` framework automatically generates a goal term `permutation([1,2,3],Y)`, and the method will return an iterator over all instances of `Y`—which are valid permutations. The `types` attribute is set to {`List<Int>,List<Int>`}: both `X` and `Y` will be terms of the kind `List<Int>`. These attributes provide a complete description of the mapping, identifying the signature of the method shown in Figure 1—the reader interested in a full description of such attributes should refer to [4, 2]. Alternatively, it is possible to move the specification of clauses into a `PrologClass` annotation of the class: this is useful if such clauses are to be used by different methods.

Inside the method `main()`, an instance of the Prolog class is created exploiting the `newInstance()` factory method provided by the `P@J` framework. By exploiting the proxy

technique, 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.permutations(list)` is captured by the `P@J` framework which, in turn, triggers the resolution of the proper Prolog query to the tuProlog engine. Advantage for the programmer include the high-level of specification for the method behavior, and the simplicity in coding the client code.

3.1 The new Annotation Layer

In this paper an alternate annotation style for Prolog methods is proposed, relying on generic methods and inference, and promoting an easier and more direct mechanism to incorporate Prolog code into Java methods. Instead of specifying all the necessary attributes to bridge a method to a Prolog predicate, the proposed extension to the `P@J` framework allows the programmer to skip that specification. First of all, some information is immediately inferred from the shape of the signature. The method should be generic and in the following form

$$\texttt{<}\overline{\texttt{X}}\ \texttt{extends}\ \overline{\texttt{Tb}}\texttt{>}\ \texttt{Tr}^{\overline{\texttt{X}}}\ \texttt{m(}\texttt{X}_\texttt{i}\ \texttt{x}_1\texttt{,}\ \texttt{X}_\texttt{j}\ \texttt{x}_2\texttt{,}\ \texttt{X}_\texttt{k}\ \texttt{x}_3\texttt{,..)}$$

where overlines are used to express lists of elements as in [7], and hence $\overline{\texttt{X}}$ are the method type variables, $\overline{\texttt{Tb}}$ their bounds (which should be terms), $\texttt{Tr}^{\overline{\texttt{X}}}$ is the return type (a term type possibly constructed from type variables), and type arguments are precisely some type variables $\texttt{X}_\texttt{i}$, $\texttt{X}_\texttt{j}$, $\texttt{X}_\texttt{k}$ and so on. More precisely, each type variable can occur either *(i)* as component of the return type of `m` (output type variable), *(ii)* as one of `m`'s argument types (input type variable) or *(iii)* in both places (input/output type variable).

For a method of this kind it is possible to define a Java-Prolog mapping that avoids the need of manually specifying all the `@PrologMethod` annotation attributes previously required. This mapping is given accordingly to the following rules:

- The name of the method `m` should coincide with the predicate name `p` to be used
- Each type variable in $\overline{\texttt{X}}$ should have a name starting with `$`, and it will correspond to a logic argument of the template predicate `p`—the arity of `p` is equal to the number of such type variables
- Each argument of `m` is a type variable $\texttt{X}_\texttt{i}$ by construction, and it will correspond to a predicate argument in the left part of the `signature` attribute
- Each type variable occurring in `m`'s return type will correspond to a predicate argument in the right part of the `signature` attribute (if `m`'s return type is a subtype of `Iterable` then `m` is implicitly assumed to yield more than one result)
- The type of the i^{th} argument of `p` is given by the bound type $\texttt{Tb}_\texttt{i}$ in the generic method declaration

For example, consider the following Prolog method declaration

```
@PrologMethod ( clauses= ... )
<$X extends List<Int>,
 $Y extends List<Int>> Iterable<$Y> permutation($X l)
```

In this case, `P@J` assumes that *(i)* the name of the template predicate `p` is `permutation`, *(ii)* the arity of `p` is equal to 2 and `$X`, `$Y` map to first and second arguments of `p`, *(iii)* `$X` is an input type variable, *(iv)* `$Y` is an output type variable (and all its results will be considered by iteration), and *(v)* the type associated to both `p`'s arguments is `List<Int>`. This declaration is hence equivalent to the one reported in Figure 1, but is fairly more compact and might be more intuitively read by a programmer: only the `clauses` attribute corresponding to the Prolog-based implementation of the method has to be specified.

Starting from the above information, read by Reflection, the `P@J` framework is able to automatically construct the necessary bridge code inside the proxy class. Secondly, thanks to type inference, method invocations are properly checked. The `javac` compiler can properly select the correct method implementation, and then determine whether a Prolog method invocation is correct; recalling the example above, `permutation()` should be invoked with an argument that is a subtype of `List<Int>`, while the return type of permutation can be assigned to a variable whose type is compatible with `Iterable<List<Int>>`, or it can be used directly into a `for-each` loop to iteratively get lists of integers.

3.2 Invocation Styles

The code in Figure 2 shows how the predicate `length/2` discussed above could be implemented by means of `P@J`'s generic Prolog methods. The class `LengthSample` defines four generic Prolog methods `size()`, each one mapping a particular exploitation of the underlying Prolog predicate `length/2`, which is attached via the `PrologClass` annotation—the actual code for predicate `length/2` is not reported for it is a predicate in the Prolog ISO library. The first `size()` method corresponds to the case in which `length/2` is exploited in a predicative way: given a list `l` and an integer value `s`, this methods yields true when `l` is a list whose size is `s`. The second `size()` method accepts a list and yields an integer value corresponding to the size of the input list; conversely, the third `size()` method is used for generating a list (of variables) whose size is fixed by the input integer value. The last `size()` methods accepts no arguments and yields an `Iterable` object over compound terms of the kind `size(l,s)`, where `l` is a list with size `s`: this method can be used to generate lists (of variables) with increasing size as needed. Interestingly, all the abstract declarations of such methods can be seen as different views over the same Prolog code, enabling different usage scenarios for the single Prolog code provided.

The `main()` method shows the usage of the `size()` Prolog methods. Note that thanks to type inference the compiler automatically understands what is the right method to invoke, and accordingly, what is the correct return type: e.g., the code

```
for (Term<?> t : ls.size()){ ... }
```

is sufficient for iteratively generating lists with increasing size.

3.3 Adding constraints

The above schema can be extended to tackle ad-hoc requirements when bridging Prolog predicates and Java methods. Suppose that a Prolog method `permutation()` is to be

```
@PrologClass(clauses={"size(X,Y):-length(X,Y)."})
public abstract class LengthSample {

  @PrologMethod abstract <$Ls extends List<?>, $Ln extends Int> Boolean size($Ls expr, $Ln rest);

  @PrologMethod abstract <$Ls extends List<?>, $Ln extends Int> $Ln size($Ls expr);

  @PrologMethod abstract <$Ls extends List<?>, $Ln extends Int> $Ls size($Ln expr);

  @PrologMethod abstract <$Ls extends List<?>, $Ln extends Int> Iterable<Comp2<$Ls,$Ln>> size();

  public static void main(String[] args) {
    LengthSample ls = PJ.newInstance(LengthSample.class);
    java.util.List<?> v = java.util.Arrays.asList(new Object[] {12,"twelve",false});
    List<?> list = new List<Term<?>>(v);
    Int length = new Int(3);
    Boolean b=ls.size(list, new Int(3)); //true: 'list' is of size 3
    Int i=ls.size(list);                 //length of 'list' is 3
    List<?> l=ls.size(length);           //[_,_,_] is a list whose size is 3
    int cont = 0;
    for (Term<?> t : ls.size()) { //{[],[_],[_,_], ...} all lists whose size is less than 5
        System.out.println(t);
        if (cont++ == 5) break;
} } }
```

Figure 2: Implementing `length/2` in `P@J`

defined, accepting a list of some unknown type `X` and returning an `Iterable<List<X>>`, namely, all permutations of the input list. In particular there is a constraint involving both arguments: `permutation` accepts a list of some unknown type `X` and returns lists of the same unknown type `X`. Such a constraint could be expressed by introducing a further type argument as follows:

```
@PrologMethod (  clauses= ... )
<Z extends Term<?>,
 $X extends List<Z>,
 $Y extends List<Z>> Iterable<$Y> permutation($X l);
```

Since type variable `Z` has not a trailing `$`, it is not treated as a logical argument of predicate `permutation/2`, hence it does not interfere with the above mechanism for automatically constructing bridge information. On the other hand, `javac` compiler uses this variable for checking any further constraint by type inference: in this case, it takes care that `$X` and `$Y` are lists of a same type. This exploitation of additional type variables makes the expressiveness of Java generic methods fully available to the programmer without the need of imposing restrictions on Prolog method declarations.

3.4 Benefits of Type Inference

As the generic type system has been introduced in Java 5.0, the gap between required skills of API developers and API users seriously increased. The design of the Collections Framework (`java.util.*`), for instance, heavily relies on generics, wildcards, and type inference; on the other hand, users of the API may know very little about such concepts. In general, the user should just create generic collections and then call the API: the compiler is in charge of checking for an incorrect use of types. For instance, method `Collections.sort()` is declared in the Java API in the following (rather cumbersome) way:

```
static <T extends Comparable<? super T>>
                          void sort(List<T> list)
```

but it can be simply used as follows:

```
ArrayList<Number> l=new ArrayList<Number>();
l.add(3); l.add(2); l.add(1);
Collections.sort(l);
```

Type inference in method calls here take care of finding a proper instantiation of the method type parameters, and accordingly checks the validity of the invocation—it infers e.g. `Number` for `T`, and it checks that `Number` is a subtype of `Comparable<? super Number>`.

By relying on the expressiveness of generics, `P@J` follows a similar approach. Looking at the example in Figure 2, one might notice that, although the declaration of Prolog methods might involve some tricky aspect of generics, its exploitation in `main()` is instead rather simple: the four invocations of methods `size()` are all checked and typed automatically by the compiler—in particular, no cast operation is required. Differently from the previous version of the Annotation Layer as presented in [4] and shown in Figure 1, all such checks are based on the Java language features, instead of relying on additional annotation attributes like `signature`, `types`, and so on. Such attributes can not only be considered a significant additional "boilerplate code", but requires serious skills in writing the generic signature of the Prolog method—as the `permutations()` method shows. This is due to the fact that types in the signature of `permutations()` must be consistent with respect to the attributes of the `@PrologMethod` annotations attached to that method. As an example, the return type of `permutations()` is `Iterable<Var<List<Int>>>`. This type is consistent with the `P@J` notation `-!` attached to the argument `Y` of the `permutation` predicate, as specified in the attribute `predicate`.

Moreover, by this new Annotation Layer, the non-experienced developer of a Prolog method can even specify less generic information than actually possible, e.g. bounds to type variables may be skipped—this simplification might however subsequently require more code on the call side.

```
@PrologClass (
    clauses={"arc(a,b).","arc(a,d).","arc(b,e).","arc(d,g).","arc(g,h).","arc(e,f).","arc(f,i).","arc(e,h)."})
public abstract class Graph {
    @PrologMethod (
       clauses = {"path(X,X,[X]).",
                  "path(X,Y,[X|Q]):-arc(X,Z),path(Z,Y,Q)."})
    public abstract <$X,$Y,$P> Iterable<$P> path($X from, $Y to);

    public static void main(String[] s) throws Exception {
        Graph graph = PJ.newInstance(Graph.class);
        for (Object solution : graph.path(new Atom("a"), new Var<Atom>("X"))) {
            System.out.println(((List<Atom>)solution).size());
}   }   }
```

Figure 3: Computing paths in a graph with P@J

This is the case of the example in Figure 3; the predicate `path/3` accepts two nodes of the graph `X` and `Y`, respectively, and a path `P` that connects `X` to `Y`. The graph topology is specified by a set of facts of the kind `arc(a,b)`, meaning that there is an arc connecting the node `a` to another node `b`. The predicate `path/3` can be exploited in many ways since the user can ask *(i)* for a path connecting two given nodes `a` and `b` (accordingly to the aforementioned topology) or *(ii)* for all possible paths starting from a given node. This second case is illustrated in Figure 3 where, inside method `main()`, the user asks for all possible paths connecting a given node `a` to any other node (Prolog variable `X`). As here genericity is simply specified by having three type variables `<$X,$Y,$P>`, one per argument of the underlying Prolog predicate `path/3`, the reader might appreciate how simpler is to define a mapping between a Java (generic) method and a Prolog predicate—the conciseness of this example is comparable e.g. to the one in [6], where a true Java extension with logic programming is exploited.

Differently from the examples in Figure 2, since bounds to type variables are not specified then type inference cannot precisely type the result (`solution` is given type `Object`): the user has to insert a cast conversion to access the returned list. Hence, in `P@J`, library developers can either partially or fully rely on the potentiality of `javac`'s type inference: a carefull design fully adopting type inference simply frees the user of a library from most burden related to typing.

4. EXAMPLES

4.1 Abstract data types: A Prolog-based List Library

The example in Figure 4 shows how a library of functionalities for list-like collections can be built leveraging the `P@J` framework. This example demonstrates the ability of the `P@J` framework to define structure and behavior of abstract data types, relying on the declarative character of Prolog code. There, the specification of the mapping between Prolog method `m` and Prolog query of the form $\mathtt{p}(\mathtt{t}_1, \mathtt{t}_2, ..., \mathtt{t}_n)$ is given by a careful choice of type parameters, their bounds, and their position in the method signature. Interestingly, note that the whole class really resembles the description of an interface where each method is commented with a behavior specification in the style of abstract-data-types: `P@J` prepares an implementation of all the methods for free!

Other than usual constructions methods `nil()` and `cons()`, and a functionality `contains()` for checking element containment, it is worth focusing on a fully-bidirectional specification of method `concat()`—concatenating two lists. Type variables have the bound `Term<? extends List<?>>` for variables holding lists could be passed as well. In the client code, this method is exploited not just to concatenate two lists as usual, but to generate all the couples of lists whose concatenation produces list `[1,2,3,4,5]`—the intrinsic state-space navigation abilities of Prolog are here exacerbated.

Hence, assuming that `concat` is invoked with list `[1,2,3,4,5]` as argument `l_out` and two Prolog variables (whose type is `Var<List<Int>>`) as arguments `l1` and `l2`, respectively, the `P@J` framework automatically generates a goal term `concat(X,Y,[1,2,3,4,5])`, and the method will return an iterator over all instances of `X,Y`—which are the couples of lists whose concatenation produces list `[1,2,3,4,5]`.

4.2 A Prolog-based Context-Free Parser

Another interesting area where declarative specifications are fruitfully exploited is in building parsers and interpreters.

As an example, consider how the features of the `P@J` framework can be useful to rapidly prototype new parsers. Suppose to have the following definition of a (context-free) grammar for simple mathematical expressions:

```
<exp> ::= <term> | <term> (+|-) <exp>
<term> ::= <fatt> | <fatt> (*|/) <term>
<fatt> ::= <num> | ( <exp> )
<num> ::= 0 | 1 | 2 | ...
```

Figure 5 shows the code of the class `ExprParser`, an implementation of a mathematical expression parser built on top of `P@J`. `ExprParser` defines several Prolog methods, one for each non terminal symbols in the grammar definition given above: for example, a Prolog method `expr()` is defined for the non-terminal symbol `<expr>`. Following the standard encoding of parsers into Prolog, each non-terminal symbol is parsed by a predicate with two arguments, where the former should take the input character stream, and the latter is unified with the resulting character stream after parsing one string generated by the non-terminal symbol. At the top level, the parser should take the entire stream and yield the void stream. The reader may appreciate how the Prolog code strictly adheres to the BNF specifications of the grammar.

In particular, for the parsing to take place the user is required to pass to e.g. the Prolog method `expr()` (but the same holds for `term()`, `fact()` and `num()` as well) a list

```
@PrologClass public abstract class PrologList {
   @PrologMethod (
      clauses = {"nil([])."}
   )
   abstract public <$X> $X nil();

   @PrologMethod (
      clauses = {"cons(E,L,[E|L])."}
   )
   abstract public <$X,$Y extends List<?>,$Z extends List<?>> $Z cons($X element, $Y list);

   @PrologMethod (
      clauses = {"contains([E|_],E).",
                 "contains([_|T],E):-contains(T,E)."}
   )
   abstract public <$X extends List<?>,$Y> boolean contains($X list, $Y element);

   @PrologMethod (
      clauses = {"concat([],L,L).",
                 "concat([H|T],L,[H|E]):-concat(T,L,E)."}
   )
   abstract public <$X extends Term<? extends List<?>>,
                    $Y extends Term<? extends List<?>>,
                    $Z extends Term<? extends List<?>>>
                Iterable<Comp2<$X,$Y>> concat($X l1, $Y l2, $Z l_out);

   public static void main(String[] s) throws Exception{
      PrologList lists = PJ.newInstance(PrologList.class);
      List<Int> l=lists.nil();          // Creating an empty list []
      for (int i=5;i>0;i--){
          l=lists.cons(l,new Int(i));    // Adding elements...[1,2,3,4,5]
      }
      Var<List<Int>> l1 = new Var<List<Int>>();
      Var<List<Int>> l2 = new Var<List<Int>>();
      // Executing goal concat(X,Y,[1,2,3,4,5]).
      for (Comp2<Var<List<Int>>,Var<List<Int>>> c: lists.concat(l1,l2,l)){
           System.out.println("First list is: "+c.head);
           System.out.println("Second list is: "+c.rest.head);
           // Results: ([],[1,2,3,4,5]),([1],[2,3,4,5]),..,([1,2,3,4],[5]),([1,2,3,4,5],[])
}  }  }
```

Figure 4: A library for list processing in P@J

containing the tokenized expression along with the empty list (meaning that the parsing should consume the whole input). Note that, despite the grammar definition given above leads to a non-deterministic parser this has no impact on the parser built on top of the P@J framework, since non-determinism is handled natively in Prolog (through *backtracking*)—of course, a deterministic parser would be more efficient, though. The joint exploitation of generic methods and annotations allow for a more compact signature of Prolog methods, thus making the Java code more resembling the BNF grammar specifications. Within `main()`, a Java string correspondind to the expression to be parsed (e.g. "`12+3*4`") is split into a Prolog list containing all the tokens in the original expression (thus obtaining the list "`['12','+','3','*','4']`"). This can be done by invoking `Atom`'s `split()` factory method with the *regular expression* in charge of performing the tokenization (this method can be regarded as the Prolog counterpart of `String`'s `split()` method [13]). Once a list of atoms has been retrieved, that list could be passed as argument to the various Prolog methods defined in `ExprParser` to perform the parsing.

An interesting future work would be to exploit Prolog even to generate the abstract syntax tree of an expression, and to provide facilities for specifying interpreting and compiling mechanisms.

5. CONCLUSIONS

In this paper an extension to the P@J a framework has been described, that significantly improves the seamless integration of Prolog code into Java applications, exploiting tuProlog technology. The aim of this extension is to fill the gap between method invocation and Prolog goal satisfaction, exploiting Java type inference in method calls; moreover this extension enables those programmers who are familiar with Java mainstream programming to easily incorporate declarative features into their programs. The possibility of expressing rich (generic) types for the terms interfacing Java and Prolog, along with type inference for checking consistency and reconstructing bridging information is crucial for seamlessly integrating Prolog code into Java classes and methods.

Integrating object-oriented and logic-based programming has been the subject of several researches and corresponding technologies. Such proposals either attempt at joining the two paradigms [6, 12], or simply provide an interface library for accessing a Prolog engine from a mainstream object-oriented language such as Java [9, 5, 11]—see a more systematic description in [4]. Both solutions have however

```
@PrologClass
public abstract class ExprParser {

  @PrologMethod (clauses={"expr(L,R):-term(L,R).",
                          "expr(L,R):-term(L,['+'|R2]), expr(R2,R).",
                          "expr(L,R):-term(L,['-'|R2]), expr(R2,R)."})
  public abstract <$E extends List<?>, $R extends List<?>> Boolean expr($E expr, $R rest);

  @PrologMethod (clauses={"term(L,R):-fact(L,R).",
                          "term(L,R):-fact(L,['*'|R2]), term(R2,R).",
                          "term(L,R):-fact(L,['/'|R2]), term(R2,R)."})
  public abstract <$T extends List<?>, $R extends List<?>> Boolean term($T term, $R rest);

  @PrologMethod (clauses={"fact(L,R):-num(L,R).",
                          "fact(['(' | E],R):-expr(E,[')'|R])."})
  public abstract <$F extends List<?>, $R extends List<?>> Boolean fact($F fact, $R rest);

  @PrologMethod (clauses={"num([L|R],R):-num_atom(_,L)."})
  public abstract <$N extends List<?>, $R extends List<?>> Boolean num($N num, $R rest);

  public static void main(String[] args) throws Exception {
    ExprParser ep = PJ.newInstance(ExprParser.class);
    String tokenizer_regexp = "(?<!^)(\\b|(?=\\()|(?=\\))|(?=\\-)|(?=\\+)|(?=\\/)|(?=\\*))";
    List<Atom> exp1 = new Atom("12+(3-4)").split(tokenizer_regexp);
    List<Atom> exp2 = new Atom("(12+(3-4))").split(tokenizer_regexp);
    System.out.println(ep.expr(exp1, List.NIL)); //true, 12+(3*4) is an expression
    System.out.println(ep.fact(exp1, List.NIL)); //false, 12+(3*4) isn't a factor
    System.out.println(ep.expr(exp2, List.NIL));//true, (12+(3*4)) is an expression
    System.out.println(ep.fact(exp2, List.NIL));//true, (12+(3*4)) is a factor
} }
```

Figure 5: A parser for mathematical expressions in P@J

drawbacks: in the first case, the conceptual integrity of the programming model is sacrificed since the new language is typically more complex, thus making application development an harder task; in the second case, there is no true language integration, and some "boilerplate code" has to be implemented each time to fix the paradigm mismatch. On the other hand, the P@J framework trades off the two approaches in an interesting way, which can hopefully allow to better enjoy Prolog abilities into Java mainstream programming.

The P@J framework could be used as a basis for further investigation on the relationships between object-oriented/declarative programming languages. Moreover, P@J could serve as a basis for supporting high-level metaprogramming constructs as in [3]. Future works include investigating performance penalties, modeling inheritance between Prolog classes, applying dynamic dispatching techniques for Prolog methods, passing Java objects to Prolog theories, and modeling the state of Java objects as Prolog theories.

6. REFERENCES

[1] Dynamic Proxy Classes. http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html.

[2] P@J Official Website. http://www.alice.unibo.it/patj.

[3] J. Brichau, A. Kellens, K. Gybels, K. Mens, R. Hirschfeld, and T. D'Hondt. Application-Specific Models and Pointcuts Using a Logic Meta Language. In *ISC*, pages 1–22, 2006.

[4] M. Cimadamore and M. Viroli. A prolog-oriented extension of java programming based on generics and annotations. Proceedings of Principles and Practice of Programming in Java, 2007, Lisbon, Portugal, ACM Press. To appear., 2007.

[5] E. Denti, A. Omicini, and A. Ricci. Multi-paradigm java-prolog integration in tuprolog. *Sci. Comput. Program.*, 57(2):217–250, 2005.

[6] M. Espk. Japlo: Rule-based programming on java. 12(9):1177–1189, 2006.

[7] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23:396–450, 2001.

[8] A. Igarashi and M. Viroli. Variant parametric types: A flexible subtyping scheme for generics. *ACM Transactions on Programming Languages and Systems*, 28(5):795–847, 2006.

[9] JLog team. JLog – Prolog in Java. http://jlogic.sourceforge.net/, 2002.

[10] B. Joy, J. Gosling, G. Steele, and G. Bracha. *The Java Language Specification (Third Edition).* Addison-Wesley, New York, 2005.

[11] N. Kino. Jipl: Java interface to prolog. http://www.kprolog.com/jipl/, 2005.

[12] A. Omicini and A. Natali. Object-oriented computations in logic programming. In *Object-Oriented Programming*, volume 821 of *LNCS*, pages 194–212. Springer-Verlag, 1994. 8th European Conference (ECOOP'94), Bologna, Italy, 4–8 July 1994. Proceedings.

[13] Sun Microsystem. J2SE 6.0. http://java.sun.com, 2007.

[14] tuProlog Team. tuProlog at SourceForge. http://sourceforge.net/projects/tuprolog/, 2002.