

P@J : extending Java with declarative programming

Maurizio Cimadamore **Mirko Viroli**
{maurizio.cimadamore|mirko.viroli}@unibo.it

aliCE research group
ALMA MATER STUDIORUM—Università di Bologna

6th International Workshop on Multiparadigm Programming
with Object-Oriented Languages - 31th, July 2007



Joining Declarative and OO Programming

Our aim

Bringing declarative, logic-based features into OO programming

- Prolog-based implementation of Java methods

Why

Many advantages:

- Enabling technology for developing frameworks for intelligent software components
- Some algorithms can be better described in terms of LP rather than OOP
- Exploiting plain “old” OO interaction model could be crucial for maximizing usability



Prolog into Java

Prolog engines providing a Java API

- SICStus [4, 1]
- SWI [5, 3]
- ...

Prolog engines running on top of a JVM

- tuProlog (developed in our department) [6]
- JLog [2]
- ...



The tuProlog engine

100% Pure Java

tuProlog is a Prolog engine entirely written in Java

- The engine is itself a Java object!

ISO compliant

Full-fledged Prolog engine

- DCG
- I/O predicates
- ...

Minimality

The engine can be configured according to the application needs



Permutation using tuProlog

The PermutationUtility class

```
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).";

    public static void main(String[] s) throws TuPrologException {
        LinkedList<Integer> l=new LinkedList<Integer>();
        l.add(1);
        l.add(2);
        l.add(3);
        ... //to be continued!
    }
}
```



Permutation using tuProlog

Step 1: building 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));
}
Struct list_struct = new Struct(term_array);
```

The need for conversion

- In Prolog everything is a *term*...
- ...In tuProlog everything is a Term
- User data to be converted for executing a query with tuProlog
 - Java LinkedLists must be converted into tuProlog Structs!



Permutation using tuProlog

Step 2: building the goal

```
Var x = new Var("X");  
Struct goal = new Struct("permutation",list_struct,x);
```

The goal Term

- a goal term is required for executing our Prolog query
- goal is assigned the Struct representing the compound term `permutation([1,2,3],X)`



Permutation using tuProlog

Step 3: setting up the engine

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

Engine setup

- the tuProlog engine must be created. . .
- . . . and initialized with a Theory object
- Theory objects are built from:
 - String-based representation (here)
 - Stream (e.g. a file)



Permutation using tuProlog

Step 4: browsing the solution tree

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

Browsing is a bit verbose

- Prolog queries triggered by the solve method...
- ...further solutions retrieved using the solveNext method
- SolveInfo provides access to the result of a Prolog query



Permutation using tuProlog

Step 5: retrieving plain Java lists

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

The need for conversion

- In tuProlog everything is a Term object. . .
- . . . the solution of a Prolog query is itself a Term!
- Another conversion required for switching back to plain Java LinkedLists



What is missing?

Marshaling/unmarshaling support

The user is responsible for switching between Java/tuProlog representation of her data

Result browsing

Browsing solutions requires a great amount of boilerplate code

Integration with Java 5 constructs

Why not use *generic types*, *enhanced for loop*, etc. ?

Abstraction

The user is exposed to details such as:

- engine setup
- goal term/Theory object creation



Overview of the P@J framework

tuProlog engine

Overview of the P@J framework

P@J Generic API layer

Hierarchy of generic classes modeling Prolog terms that allows for flexibly mapping Prolog predicates to Java method signatures

tuProlog engine

Overview of the P@J framework

Annotation layer

Custom Java annotations providing an automatic mapping between Java method calls and Prolog query resolutions

P@J Generic API layer

Hierarchy of generic classes modeling Prolog terms that allows for flexibly mapping Prolog predicates to Java method signatures

tuProlog engine

Mapping Prolog predicates to Java methods

length(List, Length)

Different exploitations of the same predicate:

- `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;...`

The problem

- How to define a Java method modeling the predicate `length/2` ?
- Java has its own input/output bindings



Mode of an argument

The ISO Standard

ISO Prolog defines modes for arguments as follows:

- + The argument shall be instantiated
- @ The argument shall remain unaltered. Unless the argument is a compound term this is the same as the mode +
- The argument shall be a variable that will be instantiated if the goal succeeds.
- ? The argument shall be instantiated or a variable.

In practice

Each argument of a Prolog predicate plays different roles depending on the context in which it is used!



Mapping predicates to Java methods

```
boolean length(Term list, Term length);
```

- Pros: this signature works no matter which the role of the Prolog argument is
- Cons: weak typing; what happens if an Int is passed to list?

```
boolean length(Struct list, Var length);
```

- Pros: stricter signature: cannot pass an Int to list!
- Cons: input/output bindings implicitly stated in the signature
 - this method rather models length(+List, -Length)
 - Multiple signatures required, one for each binding



P@J generic API layer

Definitions

```
Term<X extends Term<?>> { ... }  
Int extends Term<Int> { ... }  
Atom extends Term<Atom> { ... }  
List<E extends Term<?>> extends Term<List<E>> { ... }  
...  
Var<X extends Term<?>> extends Term<X> { ... }
```

The idea

- X in Term<X> is a placeholder for a concrete term type
 - e.g. $X \Rightarrow \text{Int}$ since Int subclasses Term<Int>
- This holds for all terms but for variables
- It follows that e.g. Term<Int> is a supertype of both:
 - Int – because Int is a subtype of Term<Int>
 - Var<Int> – because Var<Int> is a subtype of Term<Int>



Mapping predicates to Java methods in P@J

```
length(Term<List<?>> list, Term<Int> length);
```

<i>Prolog</i>	<i>Java</i>
<code>length([a,b,c],3).</code>	<code>length(new List<Atom>(…),new Int(3));</code>
<code>length([a,b,c],Y).</code>	<code>length(new List<Atom>(…),new Var<Int>("Y"));</code>
<code>length(X,3).</code>	<code>length(new Var<List<?>>("X"),new Int(3));</code>
<code>length(X,Y).</code>	<code>length(new Var<List<?>>("X"),new Var<Int>("Y"));</code>

Notice that...

- this signature works no matter which the role of the Prolog argument is
- strong typing — cannot pass an `Int` to `list`!
- wildcards can be exploited as in `List<?>` if only partial knowledge on types is required
- some expertize required to understand the generics/wildcards syntax



Mode of an argument in P@J

`length(?List,?Length)`

We have that:

- `?List` \Rightarrow `Term<List<?>>`
- `?Length` \Rightarrow `Term<Int>`

Standard modes

<i>Prolog argument</i>	<i>Java type</i>
<code>@List</code>	<code>List<Int></code>
<code>+List</code>	<code>List<? extends Term<Int>></code>
<code>-List</code>	<code>Var<? extends List<? extends Term<Int>>></code>
<code>?List</code>	<code>Term<? extends List<? extends Term<Int>>></code>

Extended modes

<i>Prolog argument</i>	<i>Java type</i>	<i>Meaning</i>
<code>!List</code>	<code>Var<List<Int>></code>	output ground
<code>?!List</code>	<code>Term<? extends List<Int>></code>	input ground/output ground



Permutation using P@J - Generic API layer

The PermutationUtility class revised

```
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).";

    public static void main(String[] s) throws TuPrologException {
        LinkedList<Integer> l=new LinkedList<Integer>();
        l.add(1);
        l.add(2);
        l.add(3);
        ... //to be continued!
    }
}
```



Permutation using P@J - Generic API layer

Step 1: building the goal

```
List<Int> list = new List<Int>(1);  
Var<List<Int>> x = new Var<List<Int>>("X");  
Compound2<List<Int>,Var<List<Int>>> g;  
g = Compound.make("permutation",new Term<?>[]{list,x});
```

Better integration with Java types

- P@J terms are created from Java objects via constructor calls
- Generic types enforce static type checking

The Compound hierarchy

```
CompNil extends Compound {...}  
CompCons<H extends Term<?>,R extends Compound> extends Compound {...}  
Compound1<X0 extends Term<?>> extends CompCons<X0,CompNil> {...}  
...
```



Permutation using P@J - Generic API layer

Step 2: setting up the engine

```
PJProlog pj = new PJProlog();  
Theory t = new Theory(theory);  
pj.setTheory(t);
```

Engine initialization still required

- PJProlog is a thin wrapper around the tuProlog engine dealing with P@J terms *natively*
 - Theory is another P@J class extending List<Clause<?,?>>
 - Clause<H,B> is the P@J class representing the clause H:-B



Permutation using P@J - Generic API layer

Step 3: browsing solutions

```
Vector<Collection<Integer>> solutions;  
solutions = new java.util.Vector<Collection<Integer>>();  
for (Solution<Compound2<List<Int>,Var<List<Int>>>> s : pj.solveAll(g)) {  
    List<Int> li = s.getSolution().get1().getValue();  
    solutions.add(li.toJava());  
}
```

Painless solution browsing

- All solutions retrieved in a single for-each statement!
 - PJProlog's solveAll() accepts a P@J goal term and yields an Iterable object
- Java objects are created from P@J terms calling the toJava() method



P@J annotation layer

Aim

Further reducing the gap between Java and Prolog

How

- Custom Java annotations:
 - `@PrologClass` — used for annotating a class defining one or more Prolog methods
 - `@PrologMethod` — used for defining a method whose implementation is given in Prolog
- Static type checking using `javac` and *custom annotation processor* (Java 6)
- High degree of customization



Permutation using P@J - annotation layer

The PermutationUtility class revised

```
@PrologClass (
  clauses = {"remove([X|T],X,T).",
            "remove([X|U],E,[X|V]):-remove(U,E,V)."}
public abstract class PermutationUtility {
  @PrologMethod (
    clauses = {"permutation([],[]).",
              "permutation(U,[X|V]):-remove(U,X,Z),permutation(Z,V)."},
    predicate = "permutation(@X,-!Y)",
    signature = "(X)->{Y}",
    types = {"List<Int>","List<Int>"}
  abstract Iterable<List<Int>> permutations(List<Int> c);
  ...
}
```



Permutation using P@J - annotation layer

main method revised

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

Seamless integration between Java and Prolog!

- Stubs for Prolog classes generated *on-the-fly*
 - see *Dynamic Proxy Classes* (`java.lang.reflect.Proxy`)



@PrologClass annotation

@PrologClass in action

```
@PrologClass (  
  clauses = {"remove([X|T],X,T).",  
            "remove([X|U],E,[X|V]):-remove(U,E,V)."}  
public abstract class PermutationUtility { ... }
```

- @PrologClass must annotate an *abstract* class/interface
- Global clauses are specified by the clauses attribute
- remove is accessible from all Prolog methods of PermutationUtility



@PrologMethod annotation

@PrologMethod in action

```
@PrologMethod (  
    clauses = {"permutation([], []).",  
              "permutation(U, [X|V]):-remove(U,X,Z),permutation(Z,V)."},  
    predicate = "permutation(@X,-!Y)",  
    signature = "(X)->{Y}",  
    types = {"List<Int>", "List<Int>"})  
abstract Iterable<List<Int>> permutations(List<Int> c);
```

- predicate — the (annotated) Prolog predicate acting as template for the underlying query
- signature — provides the binding between method and predicate arguments:
 - argument *c* maps into the first argument of `permutation/2`
 - the method returns the second argument of `permutation/2`
 - *curly braces* denote a multiple result query
- @PrologMethod must annotate an *abstract* method



Changing the signature of a Prolog method

Changing the signature of Permutations

signature	<i>Prolog method' signature</i>
(X)->(X,Y)	Compound2<List<Int>,List<Int>> permutations(List<Int> l)
(X)->(Y)	List<Int> permutations(List<Int> l)
(X,Y)->()	Boolean permutations(List<Int> l, Var<List<Int>> x)
(X)->{Y}	Iterable<List<Int>> permutations(List<Int> l)

- *input arguments* -> *output arguments* | {*output arguments*}
- argument names consistent wrt the ones in predicate
- Look at the return type. . .
 - Multiple output arguments \Rightarrow return type is Compound
 - *curly braces* \Rightarrow return type is Iterable
 - no output argument \Rightarrow return type is Boolean



Typechecking via argument annotations

Changing the role of `List<Int>`

<i>i</i> _{th} term of predicate	Type
@X	List<Int>
+X	List<? extends Term<Int>>
-X	Var<? extends List<? extends Term<Int>>>
?X	Term<? extends List<? extends Term<Int>>>
!X	Var<List<Int>>
?!X	Term<? extends List<Int>>

- Prolog classes/methods can be statically checked using javac
 - javac accepts annotation processors via the `-proc` option
 - A true language extension!
- e.g. the signature of a given Prolog method is checked against...
 - the types specified in the `types` attribute
 - the mode of the arguments specified in the `predicate` attribute
 - the binding specified in the `signature` attribute



Other approaches

- You might think that this only applies to the tuProlog engine
- Unfortunately, almost *all* Prolog engines providing a Java interface suffer from the problems discussed here!
 - JLog engine provides marshaling/unmarshaling through the `jTermTranslation` class
- Perhaps, the only solution is *Japlo* [7]
 - Java-based programming language with declarative features
 - Java programs can embed Prolog clauses via *rule* declarations
 - New syntax: Prolog theories must be rewritten as Japlo rules
- Japlo is neither Prolog nor Java!



Present and future of P@J

Present

P@J as a framework bringing declarative features into Java







- Makes Prolog accessible to Java programmers without pain
- Prolog theories included into Java programs without changes
- Try P@J at: www.alice.unibo.it/patj/!

Future

- Supporting inheritance between Prolog classes
- Modeling the state of a Java object as a Prolog theory
- Compact syntax: avoiding specifying annotations where possible





References I

-  Documentation for prologbeans.
<http://www.sics.se/isl/sicstuswww/site/index.html>.
-  Jlog official website.
<http://jlogic.sourceforge.net>.
-  Jpl: A bidirectional prolog/java interface.
<http://www.swi-prolog.org/>.
-  Sicstus prolog user's manual.
<http://www.sics.se/isl/sicstuswww/site/index.html>.
-  Swi-prolog official website.
<http://www.swi-prolog.org/>.
-  E. Denti, A. Omicini, and A. Ricci.
Multi-paradigm java-prolog integration in tuprolog.
Sci. Comput. Program., 57(2):217–250, 2005.



References II

-  M. Espák.
Japlo: Rule-based programming on java.
12(9):1177–1189, 2006.
-  B. Joy, J. Gosling, G. Steele, and G. Bracha.
The Java Language Specification (Third Edition).
Addison-Wesley, New York, 2005.



P@J : extending Java with declarative programming

Maurizio Cimadamore **Mirko Viroli**
{maurizio.cimadamore|mirko.viroli}@unibo.it

aliCE research group
ALMA MATER STUDIORUM—Università di Bologna

6th International Workshop on Multiparadigm Programming
with Object-Oriented Languages - 31th, July 2007

