Journal of Universal Computer Science, vol. 12, no. 9 (2006), 1177-1189
submitted: 31/12/05, accepted: 12/5/06, appeared: 28/9/06 © J.UCS

Japlo: Rule-based Programming on Java

Miklós Espák
(University of Debrecen, Hungary
espakm@inf.unideb.hu)

Abstract: Imperative programming languages (such as Java) are the most widespread programming languages recently. Besides being quite easy to get familiar with them, they are also perfectly suitable for business software development. Although the productivity of imperative languages is much acclaimed, some problems are much easier to solve in a logical language. The paper introduces a Java language extension called Japlo, which fits the Prolog language constructs into Java harmonically. Blurring the borders between the representatives of these two paradigms, the author aims at making the logical programming tools more easily available for Java programmers. Japlo does not only provide a foreign language interface to Prolog programs, but you can write Prolog rules within the language relying on some basic concepts (static typing, expression orientation) and the grammatic structure of Java.
Key Words: Java, Prolog, Japlo
Category: D.1.5, D.1.6, D.3.3

1 Java-Prolog hybrid language

Although Prolog is a very expressive language, its use is rather strange for most of the users of object-oriented (OO) languages. The aim of the author is to present a Java language extension, in which it is easy to get familiar with writing logic rules also for programmers experienced in Java. Though the syntax of these rules differs from the Prolog standard (ISO/IEC 13211-1), they can be converted to the standard easily, and vice versa, that is the new syntax does not take away anything from the expressiveness of Prolog.

The language is called *Japlo*. The name is a – maybe strange – contraction of the words *Java* and *Prolog*. It was created with respect to the simple sounding and that it should not conflict with the name of any similar project.

2 The language

2.1 Terms

The data objects of Prolog are called *terms*. A term is either a *constant* (integer, float or atom), a variable or a compound term. The syntax of number constants is the same as in Java.

Two kinds of notation are mainly used in Prolog implementations for differentiating variables and atoms. In the Prolog standard the character sequences

```
anIdentifier AnotherOne ‘anAtom ‘aCompoundTerm(1, 2)
_ _bornToBeWild // wildcard identifiers
```

Figure 1: Identifiers, atoms, compound terms.

with a capital initial letter denote variables, and those with a lower case one denote atoms [SWI]. In some embedded Prolog implementations, in which the host language treats identifiers case insensitively – a special initial character is used, for example the `?` in Allegro Prolog [Allegro]. In these languages the character sequences starting with a question mark denote variables, any other identifier is an atom.

For better integration, the variables in rules are denoted by Java identifiers. This is different from both usual notation. Though the Java syntax is case sensitive, upper and lower case letters are allowed in identifiers equally, so regarding identifiers with lower case initial letter as atoms would be confusing. For other syntactical reason it is also problematic to use the `?`. Hence Japlo identifiers are the same as Java identifiers, and atoms and compound terms can be written using the *symbol operator*: ‘ (backquote) (Fig. 1).

In Prolog there is a special identifier, which matches any terms. This is usually denoted by `_` (ISO Prolog) or `?`. In Japlo the wildcard variable names start with `_`.

2.2 Types

As mentioned before, Japlo is a strongly typed language. This means that – in contrast with Prolog – variables[1] have to be declared specifying their type, making static type checking possible.

Java types can be used in the language and there are also some new types introduced for classifying Prolog data objects. These include the aforementioned *atom* type and the *list* types. The new types fit into the Java type system: they are instances of `java.lang.Class`, lists and atoms are instances of `java.lang.Object` (more precisely, atoms are `japlo.lang.Atom` objects).

2.2.1 Lists

As Japlo is a strongly typed Prolog implementation, there are also list types, which are characterized by the type of their elements (that is a common supertype of each of their elements), which is called its basetype. The name of a list

[1] The type of the formal arguments and the return value of rules have also to be declared (see below).

```
Double{} dList = {5.3, 3 - 4};
{‘first, ‘second | dList}
{1, 2, 3 | {4, 5, 6}} = {1, 2, 3, 4, 5, 6}
(Atom{}) {‘apple, ‘pear}
```

Figure 2: Defining lists.

type looks like the name of its basetype followed by braces (e.g. `Object{}`). List literals can be specified between braces, the elements of the list are separated by commas. The tail of a list can be specified after the `|` character, as illustrated in Fig. 2. Generally, for any list object of type `T{}` its head has to be of type `T` and its tail of type `T`.

If a list literal stands in a declaration, its type corresponds to the type of the variable just being declared. In other cases the compiler considers it `java.lang.Object{}`. List types different from the default can be specified by explicit type conversion.

2.2.2 Variables, declarations

In contrast with the Prolog standard Japlo is a strongly typed language. Every variable must be declared before use, including the formal arguments of rules and the variables introduced within the body of a rule. The type of the variables can only be a reference type, the value of the unbound variables is undefined or `null`.

Three kinds of objects can be declared within a Japlo rule: formal arguments, the elements of the pattern of formal arguments and local variables. The way of declaring formal arguments and local variables is the same as in Java. The elements of the argument patterns are declared automatically according to the declaration type of the argument. An example can be seen later, in Fig. 8.

2.3 Operators

Following the Java terminology, the equivalents of Prolog predicates are called operators. Most of the Java operators can be used in rules, but some of them have a modified meaning. There are also new ones.

The `=` (matching) operator unifies its operands. One of the most conspicuous difference contrary to the assignment operator of Java is that the concept of left and right values does not exist: operands take part in the operation equally (Fig. 3).

Its opposite, the `!=` operator succeed if and only if the matching has failed. The operator `==` succeeds when its operands are identical. There is no matching:

```
Integer{} list1 = {1, _, 3};
{_, 2, 3} = list1;
```

Figure 3: Matching operator

Japlo operator	Prolog predicate(s)
`if (`*condition*`)` *action*	*condition* `->` *action*
`if(`*condition*`)` *action1* `else` *action2*	*condition* `*->` *action1* `;` *action2*
`do` *action* `while (`*condition*`);`	`repeat,`*action*`,\+`*condition*`,!`
`while (`*condition*`)` *action*	*condition*`,repeat,`*action*`,\+`*condition*`,!`
goal1 `&` *goal2*	`(K =` *goal1*`, K =` *goal2*`)`[2]
goal1 `|` *goal2*	`(K =` *goal1*`; K =` *goal2*`)`[2]

Table 1: Control operators

when one of the operands is an unbound variable then it fails. Its opposite is the `!==` operator.

The operators `false`, `true`, `!`, `!+` correspond to the standard Prolog predicates `fail`, `true`, `!`, `\+` (in the same order). There are also control operators following the syntax of the Java if, if-else, while and do-while statements. Their semantics is summarized in Table 1.

The `&`, `|` operators are discussed in the next subsection.

Arithmetic predicates are substituted by the corresponding Java operators where they exist. They are summarized in Table 2.

Prolog predicate	`<`	`>`	`=<`	`>=`	`=\=`	`=:=`	`mod`
Japlo operator	`<`	`>`	`<=`	`>=`	`!!=`	`===`	`%`

Table 2: Arithmetic operators

2.4 Rules

The definition of rules is similar to method definitions. The definition starts with the `rule` keyword. Rules have a name, may have formal arguments and a return type. The return type may be `void`. Rules with `void` return type correspond to legacy Prolog rules, and are called *procedure rules*. In the case of rules with some other return type (from now *function rules*), it can be regarded as a special

[2] the value of the expression is K

(zeroth) formal argument. This argument has no name, value can be *'assigned to'* (unified with) it using the `return` operator: `return <expression>;`. In contrast with other formal arguments, the return type argument must be output argument. Fig. 5 shows an example for function rules.

The body of a rule is a so called compound query, which can contain other queries or may be empty. The form of specifying non-compound queries may remind Java: non-compound queries end with a semicolon. Compound queries are represented as a sequence of queries within braces, and can contain variable declaration, application of rules or operators, Java expressions or compound queries.

Queries written successively are AND-connected. OR connection can be expressed using the `|` symbol. AND connection is stronger than OR, so it is not necessary to parenthesize OR-connected compound queries. Fig. 4 illustrates an example for that. It contains two rules. The first one describes an *acyclic directed graph* with its directly connected nodes, and the second one specifies whether two nodes are connected through a path.

```
rule static void arc(Atom from, Atom to) {
  from = ‘a; to = ‘b;
| from = ‘a; to = ‘c;
| from = ‘b; to = ‘c;
| from = ‘c; to = ‘d;
}
rule static void path(Atom from, Atom to) {
  arc(from, to);
|
  Atom through;
  arc(from, through);
  path(through, to);
}
```

Figure 4: Defining rules.

2.5 Rule dispatch

As rules in Prolog languages and as Java methods, rules can be overloaded in Japlo, too. In contrast with the method dispatch of Java, not only the name and the number and type of arguments are taken into consideration at selecting rules. Declaration of formal arguments can contain a pattern, with which the

```
rule static Atom{} path(Atom from, Atom to) {
  arc(from, to);
  return {from, to};
|
  Atom through;
  arc(from, through);
  return {from | path(through, to)};
}
```

Figure 5: Function rules

```
rule static void arc(Atom _=‘a, Atom _=‘b) {}
rule static void arc(Atom _=‘a, Atom _=‘c) {}
rule static void arc(Atom _=‘b, Atom _=‘c) {}
rule static void arc(Atom _=‘c, Atom _=‘d) {}
rule static void path(Atom from, Atom to) { arc(from, to); }
rule static void path(Atom from, Atom to) {
  Atom through;
  arc(from, through);
  path(through, to);
}
```

Figure 6: Overloading rules.

actual arguments of rule applications are unified. Fig. 6 shows an alternative definition of the rules of Fig. 4. In case of overloaded rules a matching rule is searched for according to the order of their definitions. The selection of the rule, which can be applied, is performed using unification. Patterns can contain free names, which are declared implicitly according to the type of the pattern.

Using up return values you can express more concise – and easier to read, hopefully – such problems, in which successive manipulation on some data is needed. Fig. 7 shows an example for that. The problem is to find the nodes which cannot be avoided going from *a* to *d*. (Every path between *a* and *d* contains them.) The name of the nodes has to be printed in sorted order. The solution assumes the `intersect` rule, which takes a list of lists as its argument and returns a new list, which contains the intersection of the lists in the argument.

The figure illustrates solving the problem in Prolog and in Japlo using function rules. Besides that the latter solution is somewhat shorter, the introduction of the list variables could also be avoided.

The `&` and `|` operators (in contrast with the short-circuit `&&` and `||` operators)

```
findall(P, path(a, d, P), L1), intersect(L1, L2),
sort(L2, L3), write(L3).

write(sort(intersect(findall(path('a, 'd)))));
```

Figure 7: Using up return value for subsequently manipulation on data.

```
rule static Object memberf(Object{} _={first | rest}) {
  return first; | return memberf(rest);
}

// writing the nodes which are on one of the paths
// between 'a and 'd and between 'b and 'd:
write(memberf(path('a, 'd)) & memberf(path('b, 'd)));
// Compile error: The elements are not boolean!
write(memberf(path('a, 'd)) && memberf(path('b, 'd)));
```

Figure 8: Unificator operators

unify their operands and the value of the expression will be the unified value. Fig. 8 shows a modified version of `member/2` using return value. The code sample writes out the common elements of two lists, in sorted order.

3 Accessing Java elements from rules

Within the definition of rules (apart from their formal arguments and local variables) any visible Java name can be referred to. You can construct (instantiate) objects, call methods or access fields just as in Java. The members of the declaring class can be accessed without qualification. Non-static rules are also allowed: the `this` pseudo variable can be used as itself or for qualifying members of the declaring class.

The fields do not behave as Prolog variables, they can be assigned to any times, the modification is a simple assignment, not unification.

Any Java expression ending with a semicolon makes a goal, whose evaluation always succeeds.

4 Application of rules from Java

Atoms, variables and lists can be specified in the same way as in rules. Variables are Java variables, including the wildcard variable (`_`), which is not allowed to be declared. Variables having undefined or `null` value are unbound, other variables are regarded as bound. Primitive typed values are wrapped automatically, these values are considered as bound. Lists can also be specified in the same way as in rules. There is implicit conversion between lists and arrays of the same basetype.

The form of rule applications is the same as at method invocations, but arguments are passed by value-result. Because of this, left-values have to be passed for every output arguments[3]. In the case of function rules – as long as you would like to use up the returned value – the form *left_value* `=` *rule_name*`(` *actual_arg_list* `)` has to be used. Just like method invocations, rule applications are expressions, too. The name of a rule can be qualified by the name of a class or an instance. Rules can be applied anywhere, where an expression of the required type can occur.

4.1 Existence of a solution

When you are only interested in whether a rule can be satisfied for some arguments then the `+` unary prefix operator has to be used. The operator has to be placed before the name of the rule applied. The type of the result of this expression is `boolean` and its value is `true` if and only if the rule can be satisfied.

4.2 Obtaining one solution

It happens often that there is no need for every solution, or it is known that exactly one solution exists. In this case the rule can be applied in the form *rule_name*`(`*actual_arg_list*`)` or *left_value* `=` *rule_name*`(`*actual_arg_list*`)`, respectively, according as it is a function rule or not. Fig. 9 shows an example for that. If the matching does not succeed, a `japlo.lang.NoMatchException` unchecked exception is thrown.

```
Atom{} p = path('a, 'd);
```

Figure 9: Obtaining one solution: looking for a path between *a* and *d*.

4.3 Executing a statement for each solution

The most common intention with rule application is to perform some activity depending on the solutions. This is what the *enhanced for loop* serves for. In

[3] In predicate descriptions the arguments, which are output of a predicate, are usually preceded by a '-'.

case of function rules the head of the loop can be specified like the for-each loop of Java5 ([Java5]). An example can be seen in Fig. 10. In case of procedure rules the head should contain only the application of the rule, the declaration and the colon is omitted.

```
for (Atom{} p: path('a, 'd))
  System.out.println(p);
```

Figure 10: For each solution

4.4 Collecting solutions

Collecting every solution can be performed in the similar way as obtaining the first solution. The only difference is that in case of an unbound variable of type `T` the type of the argument has to be `Collection<? super T>` instead of `T`. During the application of the rule the elements are added to the collection in the order of being found. That is, you should use a sorted container (e.g. `SortedSet`) if you would like to obtain the elements in a certain order as in Fig. 11.

```
Collection<Atom> to = new TreeSet<Atom>();
path('a, to);
for (Atom node: to)
  System.out.printf("There is a path from a to %s\n", node);
```

Figure 11: Collecting the places that can be reached from *a*.

As it is not guaranteed for any `Collection` that the `add` method inserts its argument after the last element (so traversing the collection would reflect the order of finding the elements), so in case of more unbound variables `List<? super T>` objects should be passed. Using lists it can be guaranteed that the bound values of the individual solutions are accessed together when traversing the solutions parallelly. The order of the elements reflects, in which order the prolog engine has found them. An example for the use can be seen in Fig. 12.

5 Japlo vs. JLog

Japlo is intended to provide an intuitive interface over JLog. To set Japlo against JLog, I show, how one of the previous examples could be programmed using the

```
List<Atom> from = new ArrayList<Atom>();
List<Atom> to = new ArrayList<Atom>();
path(from, to);
for (int i=0; i<from.size(); ++i)
  System.out.printf("There is a path from %s to %s\n",
                from.get(i), to.get(i));
```

Figure 12: Collecting every path in arbitrary order.

JLog API. Fig. 13 shows a JLog variant of Fig. 10. The `jPrologAPI` constructor consults the ”path.pl” file, which contains the definition of the `path` and `arc` rules. The solutions of a query (sets of bindings) are obtained in `Hashtable` objects, in which the keys are the name of the variables.

```
jPrologAPI prolog = new jPrologAPI(new FileInputStream("path.pl"));
Hashtable varz = prolog.query("path(a, d, P)");
while (varz != null) {
    System.out.println(varz.get("X"));
    varz = prolog.retry();
}
prolog.stop();
```

Figure 13: Obtaining one solution using JLog.

The disadvantage of querying this way is that Java objects cannot be passed to rules as arguments. Although in this simple example there is no need to pass Java objects, it is obvious that passing objects to rules should be a very rudimentary feature of such a hybrid language. To achieve this the goal object should be constructed at runtime. The second example show how this can be done for the previous example. The `buildKnowledge()` method constructs the *knowledge*, which contains the definition of the `path` and `arc` rules. This code is 14 lines long, the previous solution stands of 7 lines while the Japlo version is only 2 lines long.

6 Implementation

The implementation is based on *JLog* [JLog] and the *Polyglot* compiler framework [Pol].

```
jKnowledgeBase knowledge = buildKnowledge();
jProver prover = new jProver(knowledge);
jPredicateTerms goal = new jPredicateTerms();
jCompoundTerm goalArgs = new jCompoundTerm(3);
goalArgs.addTerm(new jAtom("a"));
goalArgs.addTerm(new jAtom("d"));
jVariable P = new jVariable();
goalArgs.addTerm(P);
goal.addPredicate(new jPredicate("path", goalArgs));
boolean b = prover.prove(goal);
while (b) {
    System.out.println(P.getValue());
    b = prover.retry();
}
```

Figure 14: Obtaining one solution using JLog.

The compiler was created using Polyglot. Polyglot is an extensible compiler framework for Java. It is an ideal choice for creating Java-like languages, because it is highly extensible: the grammar, the classes representing the nodes of the abstract syntax tree (AST) and any phase of the compilation process can be modified in such way, that the original remains untouched. Of course, new AST node types and new phases in the compilation process can be introduced, and you can also remove some when they are not needed. Meanwhile, the original compiler is not modified, the new one is created only by code reuse.

The Japlo compiler translates rules into pure Java code. For a rule it creates a final field in the declaring class, which will hold the JLog rule object (`ubc.cs.JLog.Foundation.jRule`), and an initializer block, which constructs this object. The rule objects must be constructed at run-time because they may need to access the private fields of the declaring class. For example, assigning to a field is translated to an anonymous class extending an abstract predicate class. (Anonymous classes can access the private members of their declaring class.) Dynamic creation of rule objects may be slow, especially in case of instance rules. This performance problem could be avoided when not the whole rule would be constructed at run-time, but just its predicates which need to access some non-public member of the declaring class. Then these predicates can be passed to the static part of the rule as an additional argument, and the rule can perform them by the `call` meta-predicate. In case of instance rules the target object (`this`) has to be passed, too. Then the static part can be instantiated at load-time (through static initializators) or even at compile-time. In the latter case this rule object is created by the compiler and is serialized, and a custom class loader is responsible for loading the rules. Separating the static and dynamic part of rules in this way is the subject of later development.

Besides that the 'call sides' of Japlo rules have also to be replaced, so that the values of the variables that get bound by the rule application, are assigned to the corresponding actual arguments (when they are variables). To achieve this the application of procedure rules is translated into a block, in which the necessary assignments are performed. Similarly, the application of function rules is translated into a complex expression. Because Java does not have the comma operator, this functionality is performed by the `|` logical operator (not short-circuit), `==` and `?:`.

7 Summary

This paper shows how Prolog can be fit harmonically into the Java language, which has completely different concepts and syntax. The intention with creating this language was to make the means of Prolog programs more closer to and more intuitively to use for Java programmers. Although there exist several solutions for connecting Java and Prolog programs, in hardly any of them the Prolog and Java program elements can be used together, within a single program ([Kiev]). Most implementations provide an application programming interface (API) for accessing foreign Prolog programs or creating and evaluating Prolog goals dynamically. Japlo is not a whole Prolog implementation, just a lingual interface, which is build upon the JLog. ([JLog]).

The motivation behind creating the language was given by *aspect-oriented programming* (AOP) and *declarative programming* (DMP) [DMP]. DMP states that declarative languages are the most suitable for reasoning about the metainformation of programs. Although, the Japlo language itself does not support metaprogramming directly, it could be an ideal interface for accessing such a logic meta-database of Java programs from themselves. The idea of function rules and unificator operators was inspired by the pointcut designators of an AOP language called *AspectJ* [AspectJ]. Pointcut designators are logical expressions, which denote certain set of points in the structure or the execution of a program. So, the language can be an ideal choice to serve as a base for an AOP or DMP framework.

References

[onlisp] Graham, P.: "On Lisp" ;
Prentice Hall, 1993
[SWI] Wielemaker, J.: "SWI-Prolog 5.4.3 Reference Manual";
`http://www.swi-prolog.org`

[Allegro] Franz Inc.: “Allegro Prolog”; http://www.franz.com/support/documentation/7.0/doc/prolog.html

[Java5] Sun Microsystems: “JDK 5.0 Documentation”; http://java.sun.com/j2se/1.5.0/docs/index.html

[Kiev] Kizub, M.: “Kiev Language Specification”; http://kiev.forestro.com/kiev.html

[JLog] Holst, G.: JLog - Prolog in Java http://jlogic.sourceforge.net

[Pol] Nystrom, N., Clarkson, M. R., Myers, A. C.: “Polyglot¿ An Extensible Compiler Framework for Java”; http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cs/TR2002-1883

[DMP] Mens, T., Wuyts, R., De Volder, K., Mens, K.: “Workshop Proceedings – Declarative Metaprogramming to Support Software Development”; ACM SIGSOFT Software Engineering Notes, 2003 jan., http://www.iam.unibe.ch/~scg/Archive/Papers/Mens03a.pdf

[AspectJ] Kiczales, G.: “AspectJ(tm): Aspect-Oriented Programming in Java”; Lecture Notes in Computer Science, Vol.2591, 2003.