Logic Java: Combining Object-Oriented and Logic Programming

Tim A. Majchrzak and Herbert Kuchen

Department of Information Systems
University of Münster
Münster, Germany
{kuchen,tima}@ercis.de

Abstract. We have developed the programming language Logic Java which smoothly integrates the object-oriented language Java and logic programming concepts such as logic variables, constraint solving, and backtracking. It combines the advantages of object-orientation such as easy maintainability and adaptability due to inheritance and encapsulation of structure and behavior with the advantages of logic languages such as suitability for search problems. Java annotations and a symbolic Java virtual machine are used to handle the logic programming concepts. In contrast to previous approaches to integrate object-oriented and logic programming, we preserve the syntax of Java. Our language is not split into two distinguishable parts but as closely integrated as possible. Besides the design and implementation of Logic Java, providing a suitable interface between conventional and logic computations is the main contribution of this paper. A killer application, which can hardly be implemented more elegantly in any other language, is the tool Muggl which systematically generates glass-box test cases for Java programs. Applications requiring a substantial amount of search are also well suited.

1 Introduction

Object-oriented programming is the dominating programming paradigm. Concepts such as inheritance and encapsulation of structures and behavior [20] provide advantages w.r.t. maintainability and adaptability [33]. Although all application domains can be handled in principle, there are other programming paradigms which are better suited for specific application areas. For example (constraint) logic languages such as *Prolog* [42] are well-suited for search problems due to their built-in search mechanism [36]. Even though declarative paradigms are seldom used in business contexts, there are exceptions. The functional language *Erlang* [3] is successfully used in the telecommunication industry [32,43]; OCaml [35] is applied in the financial services industry [11]. This observation encourages the development of new problem-adequate languages.

We have developed a novel approach called *Logic Java* which smoothly combines the object-oriented language Java [4] with logic programming concepts such as logic variables, constraint solving, and backtracking. It preserves the syntax of *Java* and uses Java annotations to add the mentioned concepts. Java compilers can still be used without modification. However, we replace the usual Java virtual machine (JVM) [19] by a symbolic one. This symbolic Java virtual machine (SJVM) provides the usual components of a JVM *and* additional features which are known from virtual machines for logic programming languages such as the *Warren Abstract Machine* (WAM) [1], namely logic variables, choice-points, a trail, and a backtracking mechanism. If logic features are not used, the SJVM behaves just as the JVM and causes little performance overhead.

Logic computations can be nested into conventional Java computations by using a corresponding Java annotation (see Sect. 3). If a computation, which is not using logic variables, is nested into a logic computation, it will just behave as a usual Java computation. There is no need to annotate it as *conventional.* Nesting logic computations into logic computations does not change the evaluation mode. The outermost computation is always a conventional one.

Besides the design and implementation of Logic Java, the interface between both types of computations is one of the main contributions of this paper. We explain it with several examples that can be found in the course of this paper.

We assume the reader to be roughly familiar with Java [4] and logic programming (LP). Logic languages such as Prolog provide so called *logic variables*, which are initially *unbound* and then *bound* to some *constructor terms* during a computation [2]. Such a binding happens if an argument of a *predicate* call is unified with the left-hand side of a Prolog rule. Unifying two terms will cause the occurring logic variables to be bound to terms in such way that both terms become identical. If e.g. the so-called *goal* `nat(X)` is evaluated in the context of the program in Listing 1.1, the system will apply the first rule for the predicate `nat` and bind the logic variable `X` to a term only consisting of the constant `zero` and the goal will succeed since the right-hand side of the rule is empty and hence no further computations are necessary. If the user wants another *solution*, the system will *backtrack*, apply the second rule for `nat`, and bind `X` to the term `suc(Y)`. Since the right-hand side of the second rule is not empty, the computation continues and the subgoal `nat(Y)` is solved e.g. by using the first rule and by binding `Y` to `zero`. Thus, the new solution will bind `X` to the term `suc(zero)`.

Listing 1.1. A very simple Prolog program

```
nat(zero).
nat(suc(Y)) :- nat(Y).
```

Our paper is structured as follows. Details of the language design are explained in Sect. 3. Before, we give a detailed overview of related work in Sect. 2. Section 4 describes the implementation of Logic Java based on a SJVM. Section 5 contains a discussion of strengths and weaknesses and remarks concerning limitations of our approach. In Sect. 6 we conclude and point out future work.

2 Related Work

There is a plethora of work on the combination of programming languages and on multi-paradigmatic approaches. A much cited paper on the family of concurrent logic programming languages [38] suggests that research reached its climax at the end of the 1980s. It also cites a lot of lesser known approaches not further discussed here. The high number of approaches that address the idea from many different directions underline the importance of the topic.

The following approaches have been developed from a viewpoint of declarative programming. They include *syntactic sugar* to embed object-orientation (OO) features into declarative languages. The power of these languages is preserved. This makes them interesting for declarative programmers but using the OO extensions is not necessarily convenient. Thus, these languages will hardly be given attention by OO programmers. *Oz* is a lazy constraint language with concurrency that offers some object-oriented features [41]. Many multi-paradigmatic languages are based on Haskell or Curry [16], which by itself is multi-paradigmatic and combines functional and logic programming. There are extensions for an object-oriented design [17]. Special approaches incorporate further paradigms leading to e.g. constraint functional logic programming [12].

Prolog has frequently been combined with object-orientation. A classic paper by McCabe [25] presents a new language and case studies of an OO language on top of Prolog. A 1983 approach discusses object-oriented programming in concurrent Prolog [39]. *Visual Prolog* (formerly known as *Turbo Prolog* [15]) offers an OO extension for Prolog [37]. Its main purpose is the design of artificial intelligence applications, though. *Logtalk* [30] and *Prolog++* [29] are two well known approaches of adding OO features to Prolog. Regardless of the integration of object-orientation, their syntax is similar to that of pure Prolog.

Declarative meta languages such as *SOUL* [26] only roughly relate to our approach. Despite offering object-oriented functionality, embedding logic programming into an OO language is not their main purpose. For more details on the application of SOUL e.g. check [10]. Not only general approaches exist. Many authors address niche problems. E.g., there is a fuzzy object-oriented logic programming system [5,7], an object-oriented logic language for modular system specification [28] and an object-oriented logic programming environment for modeling [34]. Even a US patent (# 4.989.132) of a "tool [...] which integrates an object-oriented programming language system, a logic programming language system, and a database" exists. Other approached address special contexts such as distributed computing, e.g. ObjVProlog-D [24]. Due to their special nature, our work does not compete with any of these approaches.

All approaches so far discussed combine more than one paradigm. Yet their concepts differ from our ideas. Most notably, in almost all cases a logic language is extended with object-oriented functionality or the extension of an OO language significantly changes it. Both issues hinder a widespread reception of the languages. There is one recent approach which has similar concepts as ours. Cimadamore and Viroli combine Java with Prolog [8,9]. Despite their use of generics and annotation, their work has a different focus. It allows Prolog code to be integrated into Java and enables an exchange of data between Java and Prolog. While we embed logic programming into a Java virtual machine (VM), Cimadamore and Viroli start with an existing Prolog engine.

3 Design of Logic Java

3.1 General Principles

Before presenting Logic Java, we would like to state its design principles. Firstly, the language should be *easy to learn for Java users*. We reach this by preserving the Java syntax. This also ensures that we trivially reach the second aim: the language should be as *homogeneous* as possible. In particular, there should be no different syntax for logic and OO computations. Thirdly, there should be *no performance penalty* for conventional Java computations. We guarantee this by using a symbolic Java virtual machine which behaves as the conventional Java virtual machine if no logic computations are required. Fourthly, *several search strategies* should be supported. For efficiency, Prolog just provides the (incomplete) *depth-first search* strategy. This causes problems when writing Prolog programs. Essentially, the declarative character of the language is lost since the programmer has to avoid infinite computations. Our implementation provides the complete search strategy *iterative deepening* in addition to depth-first search. Other strategies such as breadth-first search can be added since the strategy is a modular and modifiable part of the implementation. And fifthly, *we do not mean to change Java* in a way that would require programmers to change the way they use it. Rather, we want to augment it with new constructs. Consequently, it can be perfectly used as it always was – or, if problems demand it, in its extended version with the additional power of another paradigm. We fully adhere to the Java language specification [14].

We now describe the design of the language. Logic programming concepts, namely *logic variables*, *unification*, and *backtracking*, will have to be provided or appropriately replaced when integrating Java and logic programming. There are (at least) three feasible ways for introducing logic variables:

– Providing a generic type `LogicVariable<T>` which marks variables of type `T` as logic variables.
– Annotating variables as logic variables.
– Introducing a default initialization as logic variables depending e.g. on a specific naming scheme (as e.g. in Fortran [27]).

Using generic types (*generics* [31]) is a strategy often sought. However, a wrapper class causes overhead at runtime due to the costs of object generation and (automated) (un-)boxing. A default initialization depending on naming is error prone and inflexible. Using annotations does not only provide full flexibility but also offers the best runtime characteristics. In particular, it is possible to annotate primitive types. We therefore introduce the annotation `@LogicVariable` which can be used on fields (class members) and local variables of methods.

Unification is a special case of constraint solving and many practical Prolog programs do not only rely on the rather simple constructor term unification but integrate domain specific constraint solvers which lead to a smaller search space and more efficient programs. Logic Java provides just constraint solving and no non-trivial unification for parameter passing. Unification can however be simulated using equality constraints.

The SJVM adds new constraints to the constraint store when it processes a conditional jump instruction such as `if_icmpeq` in Java bytecode. In case of `if_icmpeq` this is an equality constraint relating the two topmost entries on the stack. Other conditional jump instructions produce disequality or inequality constraints. Moreover, the SJVM will create a *choice point*. After finishing the first alternative of the computation (successfully or not), the SJVM will backtrack and replace the mentioned constraint in the constraint store by its negation.

If a method shall be evaluated as a logic computation using logic variables, constraint solving and backtracking, we annotate it with `@Search`. This annotation offers optional parameters for configuration. `strategy` can be set to change the search strategy used when executing logically. At the moment, depth first search (`SearchStrategy.DEPTH_FIRST`) and iterative deepening depth first (`SearchStrategy.ITERATIVE_DEEPENING`) are supported. The latter forces *backtracking* if no solution has been found when reaching a specified depth of search. The parameter `deepeningStartingDepth` can be set to a positive integer value. By specifying a `deepeningIncrement`, search will start over with a maximum depth which is set to the sum of the two parameters.

3.2 Introductory Examples

Let us illustrate Logic Java and its concepts using an example. The method `smm()` finds a solution for a classical problem solved by logic programming: SEND + MORE = MONEY where each character has to be replaced by a different digit (Listing 1.2). `SendMoreMoney` is a normal Java class. It has eight fields annotated with `@LogicVariable` to represent the characters. `smm()` is annotated with `@Search` to enable logic processing. The search strategy used is depth first; the parameter could have been omitted since this is the default search strategy. Please note that `allDifferent(int[])` does *not* need to be annotated since the nested call passes logic variables to it.

The outer condition of the `if` statement represents the main problem: $((1000s + 100e + 10n + d) + (1000m + 100o + 10r + e)) = (10000m + 1000o + 100n + 10e + y)$. The inner condition of the `if` statement ensures that variables assume pairwise different values. If any condition is not satisfied, an `EmptySolution` is generated. This signals that the currently considered branch of the computation could not provide a solution. If conditions are satisfied, i.e. the puzzle is solved, the values of the eight variables are (in this case) stored as an array and saved as a `Solution`. The `Solutions` container is used and either takes an instance of `Solution` or `EmptySolution` as an argument.

The predefined class `Solutions` provides the interface between logic and conventional Java computation. Note the quantity mismatch at this point. From a Java point of view, the constructor `Solutions` gets just one – possibly empty – solution as argument. The SJVM treats the type `Solutions` specifically. Rather than returning a single solution directly, it collects all solutions of a logic computation and returns them after all branches of the logic computation are finished.[1] Duplicate empty solutions are removed. If there is a non empty solution, empty

[1] Infinite computation can be avoided by using the iterative deepening search strategy.

solutions are removed. In a simple case like the present one and also for the majority of applications, a solution just consists of a value without references to logic variables. Here, this value is an array of integers. Such a solution can be fetched by the enclosing computation using the `getSolution()` method. The `main(String... args)` method shows how this can be done. Later, we will describe the general case, which is a bit more complicated to handle.

Listing 1.2. Send more money in Logic Java

```
public class SendMoreMoney {
    @LogicVariable
    protected int e, d, m, n, o, r, s, y;

    @Search(strategy=SearchStrategy.DEPTH_FIRST)
    public Solutions<Integer[]> smm() {
        if (    (s * 1000 + e * 100 + n * 10 + d)
                +  (m * 1000 + o * 100 + r * 10 + e)
                == (m * 10000 + o * 1000 + n * 100 + e * 10 +
                    y)) {
            int[] variables = {e, d, m, n, o, r, s, y};
            if (allDifferent(variables)) {
                Integer[] solution = {e, d, m, n, o, r, s, y};
                return new Solutions(new
                    Solution<Integer[]>(solution));
            }
        }
        return new Solutions(new EmptySolution());
    }

    public boolean allDifferent(int[] a) {
        for (int i = 0; i < a.length; i++) {
            for (int j = i + 1; j < a.length; j++) {
                if (a[i] == a[j]) return false;
            }
        }
        return true;
    }

    public static void main(String... args) {
        SendMoreMoney sendM = new SendMoreMoney();
        Solutions<Integer[]> solutions = sendM.smm();

        for (Solution<Integer[]> solution : solutions) {
            Integer[] values = solution.getSolution();
            for (int value : values)
                System.out.println(value);
        }
    }

}
```

The `main(String...)` method executes `smm()`, gets the solutions and iterates over them. `Solutions` implements the `Iterator` interface which is known to be very helpful [13]. It can be used for handling the solutions one by one. In this example, we simply write the values of the variables to standard output. We do not need to explicitly state that the values of variables should be single digits since the first solution returned will be the simplest one.

Note that a pure Java implementation of the send-more-money example would need to program the search explicitly, e.g. by using 8 nested loops. This would be a bit clumsy but manageable. The 8 nested loops would essentially correspond to depth-first search. Since the search space is finite, the solution would be found eventually. In situations where depth-first search runs into an infinite computation, a complete search strategy such as iterative deepening would have to be implemented explicitly. Then, a pure Java program would be extremely more complex than a Logic Java program.

Consider a Logic Java program which computes all solutions of Fermat´s problem (Listing 1.3). It finds suitable natural numbers a, b, c, n such that $a^n + b^n = c^n$. The trivial implementation of method `power(int, int)` is omitted. We encourage the reader to try to implement a corresponding pure Java program.

Listing 1.3. Fermat's Last Theorem in Logic Java

```
public class Fermat {
    @LogicVariable
    protected int a, b, c, n;

    @Search(strategy=SearchStrategy.ITERATIVE_DEEPENING,
        deepeningIncrement=5)
    public Solutions<Integer[]> fermat() {
        if (power(a,n) + power(b,n) == power(c,n)) {
            Integer[] solution = { a, b, c, n };
            return new Solutions<Integer[]>(
                new Solution<Integer[]>(solution) );
        } else {
            return new Solutions<Integer[]>(new
                EmptySolution());
        }
    }

}
```

In the above example, all logic variables are bound to values by equality constraints on integers. As explained, such a solution can be fetched by the enclosing computation using the `getSolution()` method. In general, a solution consists of a resulting value and a set of constraints which have been accumulated when producing the result. The result can be a logic variable (represented by a corresponding predefined type) or it may contain (possibly indirect) references to logic variables. A constraint is also represented by an object of a corresponding predefined class and it may (possibly indirectly) refer to objects representing logic variables. If the user wants to process such a solution, appropriate predefined methods of class `Solutions` and other predefined classes can be used in order to extract the required information. A complete description of the predefined classes for internally representing solutions, logic variables, and constraints is out of scope of this paper. We briefly sketch some possibilities only.

Besides `getSolution()` the class `Solution` offers additional methods for extracting the result and constraints of a solution and for working with them. The following list shows the most important methods. Assuming that a solution consists of a result r and a set s of constraints, both may contain logic variables.

`void addConstraint(Constraint)` adds a constraint to s. When adding it, the solver is invoked and the changed constraint system processed.

`boolean isSatisfiable()` checks whether s is still satisfiable.

`T findExampleResult()` finds an arbitrary example result of type T which is obtained by instantiating all logic variables occurring in r by values which are free of logic variables (so called *ground* values). The chosen values have to correspond to s. E.g. if r consists of $X + 5$ (where X is a logic variable) and s is the set $\{X <= 3, X > 1\}$, possibly delivered results can be 7 and 8, respectively. One of them will be chosen randomly.

`isGround()` checks whether r does not contain logic variables.

Listing 1.4. inInterval in Logic Java

```java
public class Value {
    @LogicVariable
    protected double x;

    @Search(strategy=SearchStrategy.ITERATIVE_DEEPENING,
        deepeningIncrement=5)
    public Solutions<Double> inInterval(double x1, double
        x2) {
        if (x1 <= x && x <= x2)
            return new Solutions<Double>(new
                Solution<Double>(x));
        else
            return new Solutions<Double>(new
                EmptySolution());
    }

    public static void main(String... args) {
        Solutions<Double> solutions
            = (new Value()).inInterval(3.0, 5.0);
        for (Solution<Double> sol : solutions)
            if (sol.isGround())
                System.out.println(sol.getSolution());
            else
                System.out.println(
                    sol.findExampleResult().getSolution());
    }
}
```

Listing 1.4 demonstrates the exemplary usage of some of the methods provided with Logic Java. `inInterval(double, double)` checks whether the value of the variable x is contained in the interval determined by both parameters. Note that in contrast to many other integration approaches Logic Java seamlessly works with arbitrary basic values such as doubles. Any primitive type of Java can be used: integers (byte, char, short, int, long), floating point numbers (float, double), and boolean values (boolean). The call of the method inside of method `main(String... args)` returns a `Solutions` object containing one solution consisting of the unbound logic variable `x` as result and the set of constraints $\{x >= 3.0, x <= 5.0\}$. Using `findExampleResult()`, we choose one possible value for `x` which corresponds to the constraints, e.g. 4,2.

4 Implementation

The core of the implementation of Logic Java is the symbolic Java virtual machine. It was adapted from a previous project [22,23]. Using *choice points* and *backtracking*, the SJVM processes all possible paths through a considered Java program read from the bytecode in a `class` file. Choices, e.g. conditional jumps or switching instructions, lead to the generation of *choice points*. For each such instruction, a *constraint* is generated describing the condition under which the considered branch can be entered. This constraint is added to the constraint store. The set of constraints encountered so far is processed by the built-in *constraint solver*. If the current set of constraints is not satisfiable, the considered branch of the computation is abandoned and backtracking occurs.

The architecture of the SJVM is depicted in Fig. 1. The SJVM consist of a conventional JVM which has been extended by features known from virtual machines of logic programming languages, namely logic variables (and corresponding data structures to handle their computation), choice-points, a trail, and a backtracking mechanism. Moreover, it has to manage the search strategy (depth-first search or iterative deepening). An in-depth discussion of these components is out of scope but given in [22].

Our solver has been specifically designed to process constraints generated while executing Java bytecode. During execution, linear and non-linear constraints are encountered. Whilst it is of course possible to program a method to calculate e.g. the logarithm in Java, on a bytecode level only simple instructions are used.[2] This includes basic arithmetic operations (addition, subtraction, multiplication, division, remainder), logic operations (and, or, xor) and conditional jumps. Bit operations occurring in constraints are simulated by arithmetic operations. Moreover, constraints are transformed into certain normal forms, namely equations, disequations, and inequations of polynomials. To be able to handle different types of constraints efficiently, multiple solvers have been implemented including a *simplex* solver for linear equations, a *Fourier-Motzkin* solver for

[2] There are (very) complex instructions in Java bytecode such as those for method invocation. However, they are handled the same way in the JVM and SJVM.

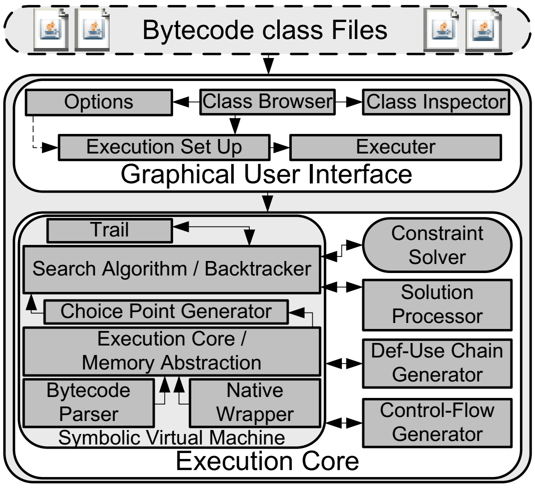

Fig. 1. Architecture of the Symbolic Java Virtual Machine and auxiliary components

linear inequations and a *bisection* solver for non-linear constraints. More details on transformations and on the solver are given in a distinct article [18].

Fig. 2 shows a small part of the bytecode generated for an implementation of *send-more-money* (cf. Listing 1.2). Numbers in the first column are instruction offsets rather than line numbers. The second column shows the Bytecode instruction and additional bytes, and the third column shows affected variables or source code statements. The operation is briefly explained in the fourth column.

When the instruction at offset 100 is reached, the outer `if` has been executed successfully and the branch where its condition is met is processed. Thus, the constraint on the constraint store before continuing execution is $((1000s+100e+10n+d)+(1000m+100o+10r+e)) == (10000m+1000o+100n+10e+y)$. Processing the inner `if` begins by pushing the logic variables e and d onto the stack. Field access is fully qualified in Java bytecode i.e. first the object reference (*this*) to the applicable class is loaded onto the stack and then its field is accessed. Eventually, the instruction at offset 108 compares the two values. Please note that the comparison in Java bytecode is negated; while we want to check for inequality, the instruction checks equality. If $e = d$ execution jumps to the instruction following the successful return from execution (offset 480: $97 + ((1 << 8) \,|\, 127)$). If any conditions from the outer or inner `if` are not met, objects for `EmptySolution` and `Solutions` are created and returned. However, if $e \neq d$ execution continues with the next step (offset 111). Value e is loaded and pushed onto the stack again since the next step is its comparison with m.

When executing `if_icmpeq`, the constraint $e = d$ is added to the constraint store. Immediately, the solver checks the constraint system for satisfiability. If it is not satisfiable, execution is not continued and the constraint is removed. Otherwise, execution continues until it is finished or another unsatisfiable constraint is met. In both cases, backtracking is started. Since the instruction offers

off	bytecode	Java	operation
...			
100	**aload_0**	*this*	Load *this* onto the stack.
101	**getfield** 0 119	*this.e*	Get value from field e and push it.
104	**aload_0**	*this*	Load *this* onto the stack.
105	**getfield** 0 123	*this.d*	Get value from field d and push it.
108	**if_icmpeq** 1 116	$if(e == d)$	Generate constraint $e == d$ and a corresponding choice point.
111	**aload_0**	*this*	Load *this* onto the stack.
102	**getfield** 0 119	*this.e*	Get value from field e and push it.
...			

Fig. 2. First Bytecode example

a second alternative for $e \neq d$, this constraint is put onto the constraint store. Again, satisfiability is checked and execution then continues. If all conditions are satisfied, the encountered solution is added to the previously encountered ones. Possibly existing empty solutions are removed. If all solutions have been collected, a new instance of `Solutions` is generated and returned (see Fig. 3).

Additional aspects besides implementing the SJVM had to be considered. First, a new package with support for the annotations has been created that allows to mark logic variables and methods that should be executed in the logic computation mode. Second, new classes have been introduced to store solutions and make them accessible.

The SJVM ensures that logic variables are initialized correctly. If an object is instantiated (by the `new` statement of Java), the JVM generates an internal representation of an object reference (*objectref*). It then checks whether fields of the underlying class have been marked with `@LogicVariable`. If so, it initializes the fields to logic variables. Otherwise, they simply take default values and are used as constants. The same applies to local variables of a method that are annotated `@LogicVariable`. They are initialized when the method is invoked and a so called *frame* is generated.

The annotation of logic variables works fine for member variables. Unfortunately, the annotation of local variables of a method is not reflected in class files even if `Target` is set to `ElementType.LOCAL_VARIABLE` and `Retention` is set to `RetentionPolicy.CLASS` or `RetentionPolicy.RUNTIME`. The claimed technical reasons for the decision to let the Java compiler ignore annotations of local variables are not convincing. We hope that they will be reflected in class files with the release of Java 7. Up to then, only member variables can be used as logic variables. This restriction is unaesthetic but not serious.

Our approach is robust w.r.t unnecessary usage of the annotations. Nesting methods annotated with `@Search` does no harm; the same applies to annotation of variables that are are used in searches. Logic variables passed as parameters remain logic variables regardless of a potential duplicate annotation. They will neither be reset nor would the doubled annotation disable logic processing. Calling methods that take a mixture of constant and logic parameters are handled, too: constant values are simply calculated in the constant way. Our JVM

off	bytecode	Java	operation
...			
408	**bipush** 8		Push 8 onto the stack.
410	**newarray** 10	*new int[8]*	Generate a new array with 8 elements and push it onto the stack.
412	**dup**		Duplicate the topmost stack element i.e. the array.
413	**iconst_0**		Push 0 onto the stack.
414	**aload_0**	*this*	Load *this* onto the stack.
415	**getfield** 0 119	*this.e*	Get value from field *e* and push it.
418	**iastore**		Save *e* as the first elements of the array.
...			
470	**astore_1**		Save the array to local variable 1.
471	**new** 0 33	*new*	Create the new object.
474	**dup**		Duplicate the topmost stack element i.e. the object reference.
475	**aload_1**		Load the array onto the stack.
476	**invokespecial** 0 133		Collect all solutions.
479	**areturn**	*return*	Return from execution with the topmost element from the stack i.e. the object reference.
...			

Fig. 3. Second Bytecode example

adheres to the the specification [19]; the decrease in efficiency between "normal" and logic mode on constant operations is negligible.

5 Strength and Limitations

Our logic extension of Java is suited to applications that require a substantial amount of search. In particular, combinatorial problems such as *n-queens*, and *graph coloring* can easily be handled. But Logic Java is not only suited for toy problems. Also practically relevant combinatorial problems such as *crew scheduling problems*, *machine planning* and various forms of allocation and resource planning problems can be handled. However, we cannot easily find optimal solutions since Logic Java does not include an optimization algorithm. However, with help of constraints corresponding to a lower or upper bound of some objective function only solutions of a certain minimal quality will be considered. Logic Java is also well suited for certain games. For example, it can be used in artificial intelligences that determine the moves of computer opponents, in particular if this requires the exploration of large search spaces.

One of the most convincing applications of Logic Java is the test-case generator Muggl [22]. Muggl systematically generates a minimal set of glass-box test cases for Java classes such that predefined code coverage criteria such as control-flow and/or data-flow coverage are met. It calls the method that shall be tested with logic variables as parameters. Each solution of a symbolic execution of the code provides a system of constraints which describes a set of equivalent test cases (w.r.t. the coverage criterion). Any solution of such a system of constraints can be used as a test case, which tests all equivalent behaviors of the program.

Compared to other approaches which integrate object-oriented and logic programming (see Sect. 2), Logic Java has the advantage that it is very close to Java; in fact it subsumes it. Most practitioners use OO languages for their daily routine. OO languages are taught in most computer science degree programs and they are well understood by almost all scientists that require programming as well. Thus, it should be easier for them to learn Logic Java than a logic programming language which has been extended by object-oriented features.

It often requires a lot of programming or leads to programs that are hard to read if problems similar to our examples are implemented in pure OO languages. At the same time, logic programming is rarely used for practical applications. Using the flexibility and versatility of an OO language and the libraries available for it is especially appealing if logic sub-routines can be used. Implementing logic programs exactly where you need them and without the need to write wrapper classes or to establish links between single software systems is extremely helpful.

Logic Java has strengths beyond the amenities of the general multi-paradigmatic approach. First of all, it is very easy to learn for users that have knowledge in OO programming. Of course, advanced concepts to successfully create logic programs are (very) hard to master. Nevertheless, users of Logic Java can broaden their knowledge as required for the tasks of their choice while starting with nothing but knowledge of Java. Secondly, the sophisticated functions to alter solutions, modify the constraints and get expressions calculated by the solver are very powerful tools. Users with advanced logic programming knowledge will find them useful. Thirdly, domains formerly dominated by logic programming become open for object-oriented programs. In particular, it becomes possible to interchange data between the "two worlds". And fourthly, annotations offer a very flexible way to implement the logic features.

Our approach also has some limitations. As already mentioned, due to the Java compiler local variables cannot be used as logic variables. Instead, member variables have to be used, until the mentioned problem is fixed, possibly with Java 7. While Logic Java has the basic functionality of Prolog, some features are missing. Most notably, there is no equivalent for the cut operator !. However, it is not difficult to simulate it, e.g. by using globally available static class variables. They guide the control flow and the backtracking mechanism.

It is currently not possible to combine logic and concurrent programming in Logic Java. While the JVM offers concurrency, it is not yet implemented for symbolic computations. Combining concurrency with symbolic computation is very tricky, in particular in the presence of backtracking. One could use some of the concepts taken from implementations of or-parallel Prolog versions such as Aurora [21,40]. However, we have not yet done so. At the moment, threads can only be used outside of logic computations and they must not interact with them.

Currently, only our JVM can be used to execute programs written in Logic Java. Since we used annotations to add the logic functionality, an *Annotation*

Processor [6] in combination with a library with the backtracking and solving functionality could be used to run Logic Java on any standard compatible [19] JVM.

6 Conclusion and Future Work

We have presented the programing language Logic Java, which combines object-oriented and logic programming. Starting from a discussion of related work, we formulated design goals and introduced the concepts of Logic Java. Along with several examples we explained its implementation as well as the interface between logic and conventional computations. Then we identified strengths and limitations of the approach and we named suitable application areas.

Studying previous approaches we observed that they often expand a language or even invent a new one. Most of them require programmers to learn new concepts and to distinguish different syntactic categories for logic and conventional object-oriented computations. We believe that our approach to combine object-oriented and logic programming is smoother than others. We preserve the Java syntax and we use a single execution mechanism for logic and conventional Java computations, namely the Symbolic Java Virtual Machine. Even though we identified some limitations of our approach, there are ways to decrease the negative effects or even circumvent many of them.

A strength of our approach is the integrated constraint solver. Even though it would be possible to work with external constraint solving libraries to solve problems such as send more money in Java, using Logic Java is more convenient and more versatile. Programmers hardly have to think about constraint solving and do not have to *program* the solver; constraint solving is neatly integrated into logic computation and done to speed up execution.

We will continue by refining Logic Java. A main aim of our future work will be the extensive experimental evaluation of our approach. More examples have to be implemented and thoroughly tested. Results gained from this will facilitate further improvement of the specification.

References

1. Aït-Kaci, H.: Warren's abstract machine: a tutorial reconstruction. MIT Press, Cambridge (1991)
2. Apt, K.R.: From logic programming to Prolog. Prentice-Hall, Upper Saddle River (1996)
3. Armstrong, J.: The development of Erlang. In: ICFP 1997: Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming, pp. 196–203. ACM, New York (1997)
4. Arnold, K., Gosling, J., Holmes, D.: The Java Programming Language, 4th edn. Addison-Wesley, London (2005)
5. Baldwin, J., Martin, T., Vargas-Vera, M.: Fril++: object-based extensions to Fril. In: Martin, T., Fontana, F. (eds.) Logic Progr. and Soft Computing, pp. 223–238. Research Studies Press, Hertfordshire (1998)

6. Bloch, J.: Effective Java, 2nd edn. Prentice Hall, Upper Saddle River (2008)
7. Cao, T.H., Rossiter, J.M., Martin, T.P., Baldwin, J.F.: On the implementation of Fril++ for object-oriented logic programming with uncertainty and fuzziness. Technologies for Constructing Intelligent Systems: Tools, 393–406 (2002)
8. Cimadamore, M., Viroli, M.: A Prolog-oriented extension of Java programming based on generics and annotations. In: Proceedings PPPJ 2007, pp. 197–202. ACM, New York (2007)
9. Cimadamore, M., Viroli, M.: Integrating Java and Prolog through generic methods and type inference. In: Proc. SAC 2008, pp. 198–205. ACM, New York (2008)
10. D'Hondt, M., Gybels, K., Jonckers, V.: Seamless integration of rule-based knowledge and object-oriented functionality with linguistic symbiosis. In: Proc. of the 2004 ACM SAC, SAC 2004, pp. 1328–1335. ACM, New York (2004)
11. Eber, J.M.: The financial crisis, a lack of contract specification tools: What can finance learn from programming language design? In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 205–206. Springer, Heidelberg (2009)
12. Fernandez, A.J., Hortala-Gonzalez, T., Saenz-Perez, F., Del Vado-Virseda, R.: Constraint functional logic programming over finite domains. Theory and Practice of Logic Programming 7(5), 537–582 (2007)
13. Gibbons, J., Oliveira, B.: The essence of the iterator pattern. J. Funct. Program. 19(3-4), 377–402 (2009)
14. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java(TM) Language Specification, 3rd edn. Addison-Wesley Professional, London (2005)
15. Hankley, W.J.: Feature analysis of turbo prolog. SIGPLAN Not. 22, 111–118 (1987)
16. Hanus, M., Kuchen, H., Moreno-Navarro, J.: Curry: A Truly Functional Logic Language. In: Proceedings ILPS 1995 Workshop on Visions for the Future of Logic Programming, pp. 95–107 (1995)
17. Kuchen, H.: Implementing an Object Oriented Design in Curry. In: Proceedings WFLP 2000, pp. 499–509 (2000)
18. Lembeck, C., Caballero, R., Mueller, R.A., Kuchen, H.: Constraint solving for generating glass-box test cases. In: Proceedings WFLP 2004, pp. 19–32 (2004)
19. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification, 2nd edn. Prentice-Hall, Englewood Cliffs (1999)
20. Louden, K.C.: Programming Languages. Wadsworth, Belmont (1993)
21. Lusk, E., Butler, R., Disz, T., Olson, R., Overbeek, R., Stevens, R., Warren, D.H., Calderwood, A., Szeredi, P., Haridi, S., Brand, P., Carlsson, M., Ciepielewski, A., Hausman, B.: The Aurora or-parallel Prolog system. New Gen. Comput. 7(2-3), 243–271 (1990)
22. Majchrzak, T.A., Kuchen, H.: Automated Test Case Generation based on Coverage Analysis. In: TASE 2009: Proceedings of the 2009 3rd IEEE International Symposium on Theoretical Aspects of Software Engineering, pp. 259–266. IEEE Computer Society, Los Alamitos (2009)
23. Majchrzak, T.A., Kuchen, H.: Muggl: The Muenster Generator of Glass-box Test Cases. In: Becker, J., Backhaus, K., Grob, H., Hellingrath, B., Hoeren, T., Klein, S., Kuchen, H., Müller-Funk, U., Thonemann, U.W., Vossen, G. (eds.) Working Papers No. 10. European Research Center for Information Systems, ERCIS (2011)
24. Malenfant, J., Lapalme, G., Vaucher, J.: ObjVProlog-D: a reflexive object-oriented logic language for distributed computing. In: Proceedings OOPSLA/ECOOP 1990, pp. 78–81. ACM, New York (1991)
25. McCabe, F.G.: Logic and objects. Prentice-Hall, Upper Saddle River (1992)
26. Mens, K., Michiels, I., Wuyts, R.: Supporting software development through declaratively codified programming patterns. Expert Syst. Appl. 23(4), 405–413 (2002)

27. Metcalf, M., Cohen, M.: Fortran 95/2003 Explained, 3rd edn. Oxford University Press, Oxford (2004)
28. Morzenti, A., Pietro, P.S.: An Object-Oriented Logic Language for Modular System Specification. In: America, P. (ed.) ECOOP 1991. LNCS, vol. 512, pp. 39–58. Springer, Heidelberg (1991)
29. Moss, C.: Prolog++: The Power of Object-Oriented and Logic Programming, 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston (1994)
30. Moura, P.: From plain prolog to logtalk objects: Effective code encapsulation and reuse. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 23–23. Springer, Heidelberg (2009)
31. Naftalin, M., Wadler, P.: Java Generics and Collections. O'Reilly Media, Inc., Sebastopol (2006)
32. Nyström, J.H.: Productivity gains with Erlang. In: Proceedings CUFP 2007. ACM, New York (2007)
33. Opdyke, W.F.: Refactoring object-oriented frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA (1992)
34. Page, Jr., T.W.: An object-oriented logic programming environment for modeling. Ph.D. thesis, University of California, Los Angeles (1989)
35. Rémy, D.: Using, understanding, and unraveling the oCaml language from practice to theory and vice versa. In: Barthe, G., Dybjer, P., Pinto, L., Saraiva, J. (eds.) APPSEM 2000. LNCS, vol. 2395, pp. 413–536. Springer, Heidelberg (2002)
36. Salus, P.H.: Functional and Logic Programming Languages. Sams, Indianapolis (1998)
37. Scott, R.: A Guide to Artificial Intelligence with Visual Prolog. Outskirts Press (2010)
38. Shapiro, E.: The family of concurrent logic programming languages. ACM Computing Surveys 21(3), 413–510 (1989)
39. Shapiro, E.Y., Takeuchi, A.: Object Oriented Programming in Concurrent Prolog. New Generation Comput. 1(1), 25–48 (1983)
40. Szeredi, P.: Solving Optimisation Problems in the Aurora Or-parallel Prolog System. In: ICLP 1991: Pre-Conference Workshop on Parallel Execution of Logic Programs, pp. 39–53. Springer, London (1991)
41. Van Roy, P., Brand, P., Duchier, D., Haridi, S., Schulte, C., Henz, M.: Logic programming in the context of multiparadigm programming: the Oz experience. Theory and Practice of Logic Programming 3(6), 717–763 (2003)
42. Warren, D.H.D., Pereira, L.M., Pereira, F.: Prolog – the language and its implementation compared with Lisp. In: Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages, pp. 109–115. ACM, New York (1977)
43. Wiger, U.: 20 years of industrial functional programming. In: ICFP 2004: Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, pp. 162–162. ACM, New York (2004)