Compiling Prolog to Idiomatic Java

Michael Eichberg[1]

1 Department of Computer Science
Technische Universität Darmstadt
eichberg@informatik.tu-darmstadt.de

Abstract

Today, Prolog is often used to solve well-defined, domain-specific problems that are part of larger applications. In such cases, a tight integration of the Prolog program and the rest of the application, which is commonly written in a different language, is necessary. One common approach is to compile the Prolog code to (native) code in the target language. In this case, the effort necessary to build, test and deploy the final application is reduced.

However, most of the approaches that achieve reasonable performance compile Prolog to object-oriented code that relies on some kind of virtual machine (VM). These VMs are libraries implemented in the target language and implement Prolog's execution semantics. This adds a significant layer to the object-oriented program and results in code that does not look and feel native to developers of object-oriented programs. Further, if Prolog's execution semantics is implemented as a library the potential of modern runtime environments for object-oriented programs, such as the Java Virtual Machine, to effectively optimize the program is more limited. In this paper, we report on our approach to compile Prolog to high-level, idiomatic object-oriented Java code. The generated Java code closely resembles code written by Java developers and is effectively optimized by the Java Virtual Machine.

1998 ACM Subject Classification D.1.6 Logic Programming

Keywords and phrases Prolog, Compiling, Logic Programming, Object-oriented Programming

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.84

1 Introduction

One application area, in which Prolog is often used today, is static source code analysis [6, 7, 2, 11]. In this area, the tight integration of the Prolog program with the rest of the application is crucial. Developers using modern IDEs expect to be able to download corresponding source-code analysis plug-ins for their favorite IDE without requiring the installation of further tools. To solve the integration problem and to make deployment a non-issue, it is either possible to use an embedded Prolog interpreter that is written in the language of the rest of the application or to compile the Prolog code to (native) code in the target language. Unfortunately, the performance of Prolog interpreters is generally not sufficient for decent source code analyses. On the other hand, approaches that compile Prolog code to code in the target language usually offer reasonable performance. However, most approaches rely on some kind of virtual machine or framework implemented in the target language [1, 5]. This layer basically implements Prolog's execution semantics and is responsible for managing the overall control-flow and the memory. The result is, that the target code generated for the Prolog code often does not look and feel native to developers familiar with the target language.

In this paper, we report on our approach to compiling Prolog to high-level object-oriented Java code which is effectively optimized by the Java Virtual Machine. Our primary goal is to

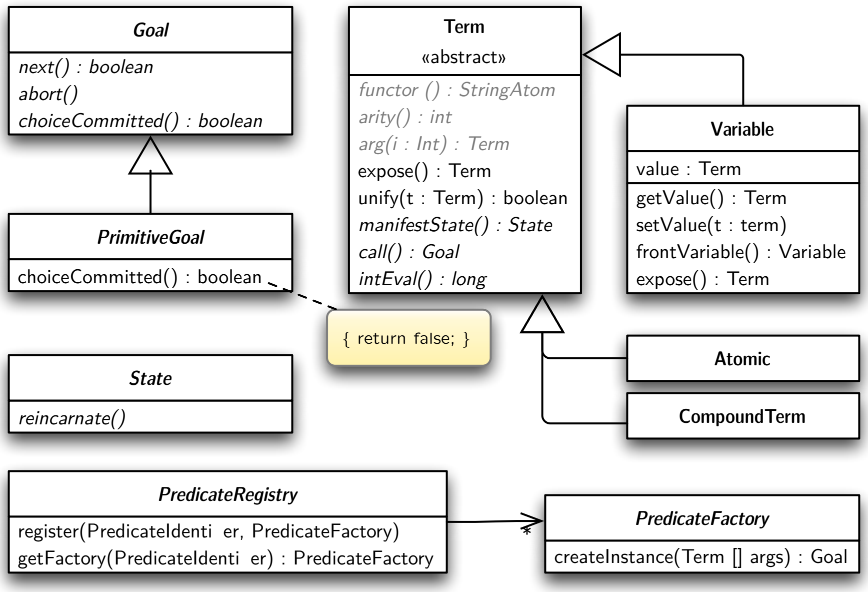

Figure 1 Core Classes and their Dependencies

generate code that is modular, easy to understand and easy to reuse/embed. I.e., we try to generate code that has the least possible impact on the overall application design, but which offers reasonable performance. To achieve this goal, we (A) rely on the functionality offered by Java Virtual Machines for memory management and (B) do not use a framework based approach which prescribes or manages the control-flow and memory used by the Prolog based part of the application. Additionally, at the interface level, the transformed code closely resembles code as written by Java developers to facilitate the integration between the Prolog based code and the (rest of the) Java application. By generating idiomatic code we also want to leverage the optimizations done by modern virtual machines. This optimizations often only fully apply to idiomatic object-oriented programs [4].

This paper is structured as follows. In the next section, we discuss our approach and its implementation — called SAE Prolog. After that, we discuss important related work in Section 3. Section 4 presents the results of our evaluation. This paper ends by drawing some final conclusions in Section 5.

2 Approach

2.1 Overview

The core idea of the chosen approach is to compile each Prolog predicate to one class that implements the complete semantics of the predicate. The goal is that classes, which implement predicates, have minimal dependencies. To this end, we defined one core interface `PrimitiveGoal` which basically defines one core method: `boolean next()` (cf. Figure 1). This interface is implemented by all classes that represent Prolog predicates and enables clients of the class (callers of the predicate) to iterate over all solutions given a certain set of arguments. I.e., the complete execution semantics of the predicate is implemented by the method `next`. The calling contract of `next` is:

```
1. Variable Rs = new Variable();
2. queens2 q = new queens2(atomic(8),Rs);
3. while (q.next()) System.out.println(Rs.toProlog());
```

Figure 2 Java code to get all solutions for the eight-queens problem.

- If `next` returns `true`, a (another) solution exists. A result is returned by binding it to the Prolog variable(s) passed to the class (predicate) when the instance was created.
- If `next` returns `false`, all parameters (Prolog variables) passed to the class are exactly as before the first call to `next`; i.e. variable bindings that were done while trying to prove the goal are undone.

Besides an implementation of the `next` method, each of the classes that implements a predicate has a constructor that defines one parameter for each argument of the corresponding Prolog predicate. The sole purpose of the constructor is to store the parameter values in corresponding fields to make them accessible later on.

For example, let's assume that we have defined a Prolog predicate `queens(N, Rs) :- ...` to solve the n-queens problem. In this case a class called `queens2` will be generated that defines a constructor (`public queens2(Term N,Term Rs){...}`) that takes two parameters: One for the problem size N and one to return a solution/to check a given solution (`Rs`). To get a solution for the eight-queens problem, it is now sufficient to create a new instance of the `queens2` class (Line 2 in Figure 2) and to pass in the problem size N (`atomic(8)`) and a free variable (`Rs` - Lines 1 and 2) to which the (next) solution will be bound, if a/another solution exist. After instantiation, we call `next` to get the first/next solution. If `next` returns `true` (Line 3), the solution is bound to the given Prolog variable; otherwise `Rs` is/remains a free variable.

The property that instances of Prolog variables have the initial state when a predicate has failed is crucial, if we have multiple dependent goals and backtracking does occur. For example, the query `?- X=a(Y),(X=a(1);X=a(2)).`has two solutions: Y == 1 and Y == 2. At runtime, when the `;/2` predicate is called/the `next` method is invoked, it first tries to prove the left goal `X=a(1)` and if it succeeds, the predicate immediately returns `true`. If we now ask for another solution, the left goal is again asked if there is another solution. Since there are no further solutions, the `;/2` predicate immediately tries to prove the right goal (`X=a(2)`). But, this requires that `Y` is (again) a free variable. If `Y` would still be bound to the value `1`, proving the right goal would fail. In general, in our approach we rely on the contract that – after a goal has failed – all terms are exactly as if we would have never tried to prove the (sub-)goal.

Given the proposed approach, it is crucial to be able to efficiently save the information which variable shares with which other variable and which variable is instantiated or free. To achieve this goal we use the State Pattern[8] to manifest the state of a (compound) term (`Term.manifestState():State` in Figure 1). Calling `manifestState` returns a `State` object that enables us to restore the term's state later on by calling `reincarnate`. The State object is basically a list of Prolog variables that do not reference other variables (at the object-oriented level). If several Prolog variables share, we chain these variables such that exactly one variable does not reference any other variable (called the `front variable` in Figure 1). For example, let's assume that the user first unifies the variable `X` with `Y` and then unifies `X` with `Z`. In this case, the Variable object (cf. Figure 1) that represents the Prolog variable `X` will reference the object that represents `Y`; i.e., the field `value` of the object representing `X` will reference the object that represents `Y` and the `value` field of `Y`

```
case X:    init new variables;
           goalStack.put( new <GOAL>(...) ); // fall through
case X+1:  if (!goalStack.peek().next()) {
             goalStack.scrapTopGoal();
             < continue with the next goal after failure >; }
           < continue with the next goal after success >;
```

Figure 3 Initializing and calling a non-deterministic goal.

will remain free. If the Prolog program then unifies `X` and `Z`, the `value` field of the object representing `Z` will reference the object referencing `Y`. If the Prolog program later on unifies `Y` and `Z` "nothing" happens. Whenever two variables are unified, we always first check if their *front variables* are different. In this case the *front variable* of `Z` and `Y` is `Y`. Hence, we know that these two variables already share and no further changes are necessary.

To support the cut operator, the interface `Goal` (cf. Figure 1) defines two further methods: `abort()` and `choiceCommitted():boolean` (cf. Figure 1). The latter method returns `true`, if the goal is the cut operator (`!/0`) or if a subgoal of an or(`;/2`) or and(`,/2`) goal returns `true`. If `true` is returned, we do commit to the current choices. If the immediate subsequent goal fails, no backtracking is done, but instead `abort` is called on all goals preceding the cut. I.e., all terms passed to the previous goals are reset to their initial states.

In our approach, we support calling of terms/predicates using `call(Goal)` by means of a factory [8]. For each predicate, we generate a second class that inherits from `PredicateFactory` (cf. Figure 1) and which defines a single method to create an instance of the predicate given the correct number of arguments. The factory object is registered with a central predicate registry (`PredicateRegistry`) that associates a predicate's identifier (`Functor/Arity`) with the corresponding factory. If the Prolog program at runtime calls a dynamic predicate or uses the `call/1` predicate, we first lookup the predicate's factory object and use it to create an instance of the actual predicate. The lookup functionality to call a term is implemented by the method `call():Goal` of class `Term` (cf. Figure 1).

2.2 Compiling Predicates to Java Code

In the previous section, we outlined the general approach and explained the public interface that all classes share that implement a predicate. In this section, we discuss how we translate a predicate that has multiple clauses to Java code.

The core idea is to represent a clause's control-flow using a switch statement where each goal that is not a conjunction or disjunction of goals is mapped to two case statements. In the following, we refer to these goals as primitive goals. From the point of view of the inter-goal control-flow the first case statement is responsible for handling the initial-call of the goal and the second case statement handles the redo case in case of backtracking. However, w.r.t. initializing and calling a goal the two case statements work closely together and "internally" there is no necessary strict separation.

In case of the call of a non-deterministic predicate the generated code is outlined in Figure 3. In this case, the first case statement first initializes new clause-local variables that are used by the goal, then the goal itself is initialized and then the goal instance is put on the clause-local goal stack. The second case statement then handles the initial call as well as all further attempts to re-satisfy the goal, in case of backtracking. All goals – except of unifications, arithmetic expressions or cuts – are compiled using this schema.

The cut operator is compiled as shown in Figure 4. The first case statement handles the

```
case X:   cutEvaluation = true;
          < continue with the next goal after success >;
case X+1: goalStack.abortAndScrapAllGoals();
          return false;
```

Figure 4 Result of the compilation of the cut operator.

```
case X:   State s1=t1.manifestState(), s2 = t2.manifestState();
          if(t1.unify(t2)) {
             goalStack.put(UndoGoal.create(s1,s2));
             < continue with the next goal after success >;
          } else {
             s1.reincarnate(); s2.reincarnate();
             < continue with the next goal after failure >; }
case X+1: goalStack.abortAndScrapAllGoals();
          < continue with the next goal after failure >;
```

Figure 5 Unification of two terms `t1` and `t2`.

initial "call" of the cut operator. The information that a cut was called is stored and we continue with the next goal. When a subsequent goal fails all previous goals on the goal stack are aborted and we let the clause as a whole fail.

If two terms are unified (cf. Figure 5), we first manifest the state of the terms, before we try to unify them. If they unify, we put a special undo-goal on the clause-local goal stack to be able to later on restore the state of both terms in case of backtracking. If the terms do not unify, we restore the initial state and continue with the next goal.

To be able to map a clause's control-flow to case statements of a switch statement, we first associate each primitive goal with a unique id in the range [0.."number of primitive goals"]. The id of the case statement that handles the initial call is then "`2 * Goal ID`" and the id of the second case statement is "`2 * Goal Id + 1`". Additionally, we analyze the control-flow of the clause to determine the order in which the primitive goals have to be called at runtime. This analysis associates each primitive goal with the goal which needs to be called next, if the current goal succeeds. Furthermore, it associates each primitive goal with the list of those goals that may need to be executed next if the goal fails. This list contains multiple entries if the clause's body contains one or more disjunctions. For example, in case of `(b;c),d`, the goal `b` or `c` may be the direct predecessor of `d` at runtime and if `d` fails at runtime, we need to continue with the goal that preceded `c`. However, the list of goals that need to be executed next if a goal fails, is not to be confused with the list of goals that may precede a certain goal at runtime. E.g., in case of `a,(b;c)` the goal that needs to be called next, if `b` fails, is `c` and not `a`, which will always directly precede `b`. In case of `c` the goal that needs to be executed next if `c` fails, is `a`. Finally, the analysis marks all goals that are not the unique predecessor of its successors as such.

Given the control-flow graph and the ids of the goals, we can then create the code to manage the control-flow of a clause. We discuss our translation scheme, based on the example: `a,(b;c),d`. The result of compiling `a,(b;c),d` is shown in Figure 6. In this case, the goal that will be called next if `a` has succeeded is `b`, if `a` has failed the corresponding list of next goals to execute is empty. In case of `b`, the next goal is `d` if `b` has succeeded and `c` otherwise. If `c` has failed, the next goal is `a` and `d` otherwise. Hence, the goal `d` has two predecessors which may have been called immediately before `d`: `b` and `c` and which may need

```
 1: private int caseToExecute = 0;
 2: private int caseToExecuteIfGoalDFails;
 3: boolean clause() {
 4:    eval_goals: do { switch (caseToExecute) {
 5:       case 0: goalStack.put(new a0());               // goal: a
 6:       case 1: if (!goalStack.peek().next()) {
 7:                  goalStack.scrapTopGoal();
 8:                  return false; }
 9:       case 2: goalStack.put(new b0());               // goal: b
10:       case 3: if (!goalStack.peek().next()) {
11:                  goalStack.scrapTopGoal();
12:                  caseToExecute = 4;
13:                  continue eval_goals; }
14:               caseToExecuteIfGoalDFails = 3;
15:               caseToExecute = 6;
16:               continue eval_goals;
17:       case 4: goalStack.put(new c0());               // goal: c
18:       case 5: if (!goalStack.peek().next()) {
19:                  goalStack.scrapTopGoal();
20:                  caseToExecute = 1;
21:                  continue eval_goals; }
22:               caseToExecuteIfGoalDFails = 5;
23:       case 6: goalStack.put(new d0());               // goal: d
24:       case 7: if (!goalStack.peek().next()) {
25:                  goalStack.scrapTopGoal();
26:                  caseToExecute = caseToExecuteIfGoalDFails;
27:                  continue eval_goals; }
28:               caseToExecute = 7;
29:               return true;
30:    }  } while (true);
31: }
```

Figure 6 The result of compiling “`a,(b;c),d`” to Java code.

to be called, if `d` fails. If `d` succeeds no further goal will be called. We need one variable to store the information which case statement needs to be executed next (`caseToExecute` - Line 1 in Figure 6). Additionally, we create one variable for each goal that has multiple predecessors (e.g., `caseToExecuteIfGoalDFails` Line 2) to store the information which goal preceded it at runtime.

If a goal succeeds, the id of the case statement that needs to be executed next (`caseToExecute`) is set to the id of the first case statement of the target goal (e.g., Line 15 in Figure 6). After that, the evaluation is continued by jumping to the respective case statement (Line 16). If a goal succeeds that has no successor, we set `caseToExecute` to the goal's second case statement. If the clause is called again in case of backtracking, the redo case is executed. After that, we return `true` (Lines 28 and 29). If the current goal is not the unique predecessor of its successor (e.g., `b` and `c` in our example), we additionally save the information which goal preceded its successor (line 14 and 22).

If a goal fails and there are no more alternatives we return `false` (line 8). If there is exactly one alternative, we set `caseToExecute` to the id of the first case statement of the target goal (Lines 13 and 22). After that, we continue the execution with the respective case statement (Lines 12 and 20). If the goal that needs to be executed next cannot be statically

```
    Prolog                      Java
 1:                             private int clauseToExecute = 0;
 2:                             public boolean next() {
 3:                                switch (clauseToExecute) {
 4: goal :- a,                     case 0: if (this.clause0()) return true;
 5:     !.                                 if (cutEvaluation) return false;
 6:                                        goalToExecute = 0; clauseToExecute = 1;
 7: goal :- b.                     case 1: if (this.clause1()) return true;
 8:                                        goalToExecute = 0; clauseToExecute = 2;
 9: goal :- c.                     case 2: return this.clause2();
10:                             }  }
```

Figure 7 Code to select the current clause.

```
member(X, [X|_]).
member(X, [_|Ys]) :- member(X, Ys).
```

Figure 8 The predicate `member/2`.

determined, we set the id of `caseToExecute` to the id that was specified by the goal that preceded the current goal (Line 26).

The approach that we have chosen to model the inter-clause control-flow is based on the idea discussed, e.g., in [3]. The control-flow is encoded in the `next` method and uses a switch statement to select the current clause. In Figure 7, the generated code is outlined. After creation, the first clause is called until it fails (Line 4). When it fails and there may have been a cut, we check whether it was effective. If so, the predicate as a whole fails (Line 5), otherwise we continue by calling the next clause (Line 7).

2.2.1 Optimizing Tail-recursive Calls

To demonstrate the potential of this approach, we briefly discuss our last call optimization. Currently, we optimize tail-recursive calls where we can statically decide that there are no more choice points left, when the tail call is made. For example, in case of the predicate `member/2` shown in Figure 8, it is easy to decide that there are no more choice points left if the tail-recursive call is made – "all goals" are deterministic.

The generated code is outlined in Figure 9: First, we save the state of the arguments passed to the predicate (Lines 3 and 4). This is necessary to be able to reset the arguments' state when the predicate eventually fails (Line 15). Second, we embed the `next` method's switch statement into an endless loop (Line 7–15) to be able to jump to a previous clause. If the tail-recursive clause succeeds, we continue with the next iteration of the loop (Line 13) and again start with the first goal of the first clause (Line 12). Third, before the tail call is made, we have to update the fields that store the predicates' arguments (Line 20).

3 Related work

Our approach to compiling Prolog code to Java code uses a large number of ideas originally explored in various Prolog implementations. For example, the idea to make the overall architecture of applications — where only some part is implemented in Prolog — a core design goal was, e.g., explored by tuProlog[10].

```
public member2(final Term arg0, final Term arg1) {
   // ... store the arguments for future use
   initialArg0state = arg0.manifestState();
   initialArg1state = arg1.manifestState();
}
public boolean next() {
   eval_clauses: do { switch(clauseToExecute) {
      ...
      case 1: // tail recursive clause with last call optimization
              if (clause1()) {
                 < reset clause local variables >;
                 goalToExecute = 0; clauseToExecute = 0;
                 continue eval_clauses; }
              abort(); return false;
   }  } while (true);
}
private clause1() {
    // forever, switch ...
    case 2: // tail call with last call optimization
            arg1 = < Ys >; // update arguments
            goalStack.scrapAllGoals();
            return true;
}
```

Figure 9 Code specific for for the predicate `member/2`.

The approach to compile each clause to its own method is also used by Prolog Café. However, compared to SAE Prolog, Prolog Café uses a WAM-based compilation schema with continuation passing style. The general idea of using a switch statement to branch over the different clauses of a predicate was also described in [3]. However, to the best of our knowledge, the idea to directly map the control-flow of a clause to a switch statement where each goal is represented using two case statements was not explored previously. The proposed approach enables us to treat a cut operator as a normal goal during the calculation of the control-flow graph.

The idea to generate "idiomatic code" was also explored previously. In [9] an approach is described to create idiomatic C code and [4] discuss the generation of idiomatic C# code. However, the meaning of the term idiomatic differs. We consider the code that we generate as idiomatic, because all classes that implement a predicate have a very lightweight interface. Furthermore, we rely on the Java virtual machine for memory management and general optimizations and do not implement our own abstraction layer. In the cited paper, the term idiomatic is used to describe code which uses the primitive data-types and control-flow constructs of the target language. But, to generate such idiomatic code the Prolog predicate requires type and mode annotations and has to be (semi-)deterministic. In all other cases a WAM-based translation scheme is used. Hence, the overall architecture is not idiomatic w.r.t. our understanding of the term.

4 Evaluation

We have done a twofold preliminary evaluation of our approach. First, we compared SAE Prolog to other Prolog implementations to assess its performance. Second, we evaluated it

Table 1 Performance comparison. All times are given in seconds; the programs tak, qsort, primes, queens-8 (findall), chat parser were each run several times.

Program	JLog 1.3.6	Prolog Café 1.2.5	SWI-Prolog (-O) 5.8.3, 64 bits	SAE Prolog 0.1.2
8-queens (findall)	141,85	6,30	6,15	5,09
22-queens	515,87	19,80	23,45	15,65
qsort	*failed*[a]	*failed (OOM)*[b] 4,60	4,20	4,74
primes	848,77	*failed (OOM)* 21,00	18,76	16,41
nrev	*failed (OOM)*	0,29	0,17	0,85
tak	*failed (OOM)*	21,88	16,36	14,69
chat parser	250,94	18,92	9,54	54,01

[a] The query did not return the expected result.
[b] After changing the test harness, we were able to run the unmodified program.

w.r.t. the desired property that it generates code that the Java virtual machine is able to effectively optimize. All performance measurements were done on a 2,33 GHz Core 2 Duo, with 3GB RAM and a Java HotSpot(TM) 64-Bit Server VM.

4.1 Performance Comparison

We compared our implementation with: JLog[`http://jlogic.sourceforge.net`], Prolog Café[1] and SWI-Prolog[12]. JLog is a Java-based Prolog interpreter that has a straightforward implementation and a clean object-oriented architecture. Prolog Café specifically targets performance and translates Prolog code to Java code by means of the WAM. SWI-Prolog is a mature Prolog implementation that is implemented in C and uses the ZIP VM.

The results of our performance evaluation are summarized in Table 1. When we compare the results of these Prolog implementations, we immediately see that JLog generally performs the worst, which was expected given that it is a Prolog interpreter. Prolog Café and SWI-Prolog are roughly on par. SAE Prolog is a bit faster for queens, tak and primes and is on par with Prolog Café in case of qsort. In case of chat parser and nrev SAE Prolog is considerably slower than Prolog Café and SWI-Prolog. Given the early state of the implementation of SAE Prolog these results are encouraging. Implementing well-known techniques, such as term-indexing, will lead to a significant speedup, as preliminary, manual tests have shown and will help us to close this gap.

4.2 Optimizability of the Generated Code

To assess the optimizability of the generated code, we made a detailed analysis of two of the benchmarks. First, we searched for the first solution to the twenty-queens problem. Finding the first solution usually takes more than one second, but the amount of code (3 predicates) that is involved is very small. The second benchmark that we used is the chat parser benchmark. In this case, the time to execute it once is also very small, but the number of involved predicates is much higher (> 100). We run the base benchmark several times without restarting the VM to make sure that the VM “fully optimized” the given program. Additionally, we did repeat each experiment five times. The results are shown in the Figures 10 and 11.

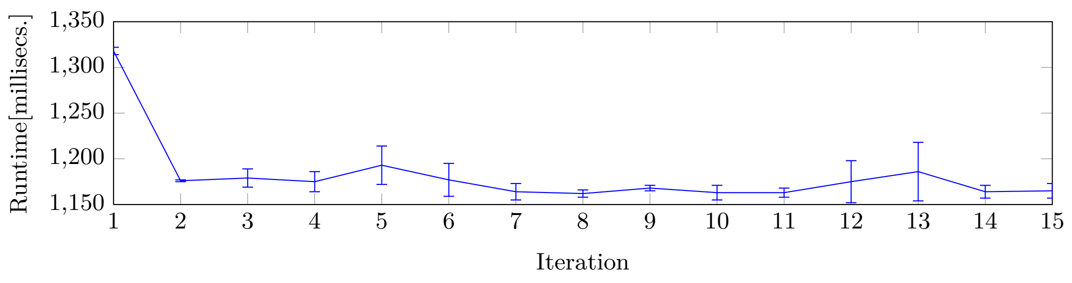

Figure 10 Performance development of the twenty-queens (first solution) benchmark.

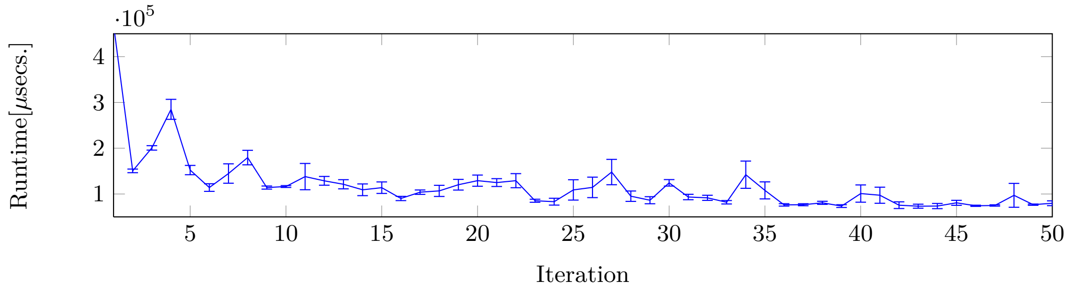

Figure 11 Performance development of the chat parser benchmark.

As shown in the respective figures, the first iteration is always dramatically slower than every succeeding iteration. The initial translation of the Java bytecode to machine code is done as part of this iteration. However, it takes the Java Virtual Machine (JVM) several more iterations (8 for twenty-queens, and 50 for chat parser) before no further performance improvements are measurable. In case of the chat parser benchmark the effect of the step-wise optimizations done by the JVM are immediately obvious. Many of the predicates are not utilized very often and therefore several runs are required before the code is "fully optimized". If we take the second iteration as the base line, we can conclude that the JVM is able to effectively optimize the generated code. The VM is able to improve the performance by a factor of 2 to 3 when compared to the second iteration.

We have repeated this experiment using Prolog Café to gain a better understanding of the optimizability of the code generated by our approach. In case of Prolog Café the VM is also able to further improve the performance. But, the effect is smaller and it takes the VM longer to reach a stable state.

5 Conclusion

In this paper, we have presented an approach to compiling Prolog code to idiomatic object-oriented (Java) code that does not use a virtual machine or framework to implement Prolog's execution semantics. Instead each Prolog predicate is compiled to one class that looks and feels natural to object-oriented developers. Our preliminary evaluation shows that Prolog applications where efficient head unification is not a major concern already perform well and are competitive. Furthermore, we have shown that a modern virtual machine, such as the Java Virtual machine, is able to very effectively optimize the generated program.

Acknowledgment The authors would like to thank Andreas Sewe for providing deep insights into the inner workings of current Java Virtual Machines.

References

1 Mutsunori Banbara, Naoyuki Tamura, and Katsumi Inoue. Prolog cafe : A prolog to java translator system. In *INAP*, pages 1–11. Springer-Verlag, 2005.

2 William C. Benton and Charles N. Fischer. Interactive, scalable, declarative program analysis: from prototype to implementation. In *Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '07, pages 13–24, New York, NY, USA, 2007. ACM.

3 Joanne L. Boyd and Gerald M. Karam. Prolog in 'c'. *SIGPLAN Not.*, 25:63–71, July 1990.

4 Jonathan J. Cook. Optimizing P#: Translating prolog to more idiomatic C#. In *Proceedings of CICLOPS 2004*, pages 59–70, 2004.

5 Jonathan J. Cook. P#: a concurrent prolog for the .net framework. *Softw. Pract. Exper.*, 34:815–845, July 2004.

6 Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. Defining and continuous checking of structural program dependencies. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 391–400, New York, NY, USA, 2008. ACM.

7 Henry Falconer, Paul H. J. Kelly, David M. Ingram, Michael R. Mellor, Tony Field, and Olav Beckmann. A declarative framework for analysis and optimization. In *Proceedings of the 16th international conference on Compiler construction*, CC'07, pages 218–232, Berlin, Heidelberg, 2007. Springer-Verlag.

8 Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

9 Fergus Henderson and Zoltan Somogyi. Compiling mercury to high-level c code. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 197–212, London, UK, 2002. Springer-Verlag.

10 Giulio Piancastelli, Alex Benini, Andrea Omicini, and Alessandro Ricci. The architecture and design of a malleable object-oriented prolog engine. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 191–197, New York, NY, USA, 2008. ACM.

11 Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. Monitoring compliance of a software system with its high-level design models. In *Proceedings of the 18th international conference on Software engineering*, ICSE '96, pages 387–396, Washington, DC, USA, 1996. IEEE Computer Society.

12 Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. Swi-prolog. *Theory and Practice of Logic Programming (TPLP)*, to appear.