Prolog Cafe: A Prolog to Java Translator System

Mutsunori BANBARA[1], Naoyuki TAMURA[1], and Katsumi INOUE[2]

[1] Information Science and Technology Center, Kobe University
1-1 Rokkodai, Nada, Kobe 657-8501, Japan
{banbara,tamrua}@kobe-u.ac.jp
[2] National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
ki@nii.ac.jp

Abstract. We present the Prolog Cafe system that translates Prolog into Java via the WAM. Prolog Cafe provides multi-threaded Prolog engines. A Prolog Cafe thread seem to be conceptually an independent Prolog evaluator and communicates with each other through shared Java objects. Prolog Cafe also has the advantages of portability, extensibility, smooth interoperation with Java, and modularity. In performance, our translator generates faster code for a set of classical Prolog benchmarks than an existing Prolog-to-Java translator jProlog.

1 Introduction

Recent development of Prolog in Java suggests a successful direction to extend not only Prolog to be more networked and mobile, but also Java applications to be more intelligent. We aim to develop a Java-conscious Prolog system that has the advantages of portability, extensibility, interoperability, parallelism, and modularity.

In the implementation of Prolog, the Warren Abstract Machine (WAM) [1, 2] has became a *de facto* standard. WAM is flexible enough for several extensions such as higher-order, concurrent, constraint, and linear logic programming. WAM has been also a basis for compiling Prolog into C [3], C++, Java, C#.

We present the Prolog Cafe system that translates Prolog into Java. The execution model of translated Java classes is based on the WAM. Main features of Prolog Cafe are as follows:

- *Portability*
  Prolog Cafe is a 100% pure Java implementation and is portable to any platform supporting a Java compiler.
- *Extensibility*
  The output of Prolog Cafe translator can be well structured and readable. Prolog Cafe is therefore easily expandable with increasing Java class libraries.
- *Interoperability*
  From the Java side, the translated code of Prolog Cafe can be easily embedded into Java applications such as Applets and Servlets. From the Prolog side, any Java object can be represented as a Prolog term, and its methods and fields can be exploited from Prolog.

– *Parallelism*
  Prolog Cafe provides multi-threaded Prolog engines. A Prolog Cafe thread seem to be conceptually an independent Prolog evaluator, in which a Prolog goal runs on a local runtime environment. Prolog Cafe threads communicate with each other through shared Java objects.
– *Modularity*
  Prolog modules are translated into separate Java packages that can be imported from each other.

In performance, our translator generates faster code for a set of classical Prolog benchmarks than an existing Prolog-to-Java translator jProlog. Main differences from the previous version of Prolog Cafe [4] are new features of interoperability, parallelism, and modularity listed above.

Usefulness of Prolog Cafe has been shown through several applications: P# [5], Multisat [6], Maglog [7], Holoparadigm [8], and CAWOM [9]. Among these applications, we give a brief introduction to the Multisat system, a parallel execution system of SAT solvers.

The reminder of this paper is organized as follows. After showing the translation method of existing Prolog to Java translators in Section 2, Section 3 presents the Prolog Cafe system in detail. Related work is discussed in Section 4, and the paper is concluded in Section 5.

2 Existing Prolog to Java Translator Systems

In this section, we present how jProlog and LLPj translate Prolog into Java. Due to space limitations, we use the following simple example:

```
p :- q, r.
q.
```

2.1 jProlog

The jProlog [10] system, developed by B. Demoen and P. Tarau, is a first generation Prolog to Java translator via the WAM. jProlog is based on *binarization transformation* [11], a continuation passing style compilation used in BinProlog.

In jProlog approach, each clause is first translated into a binary clause by binarization and then translated into one Java class. Each predicate with the same name/arity is translated a set of classes; there is one class for direct entry point, and other classes for clauses. Each continuation goal is translated into a term object, that is executed by referring to the hash table to transform it into its corresponding predicate object.

Our example is first translated into the following binary clauses where `Cont` represents a continuation goal:

```
p(Cont) :- q(r(Cont)).
q(Cont) :- call(Cont).
```

and then each of them is translated into the following Java classes:

```
public class pred_p_0 extends Code { % entry point of p/0
    static Code cl1 = new pred_p_0_1();
    static Code q1cont;
    void init(PrologMachine mach) {
        q1cont = mach.LoadPred("q",0); % get continuation goal q/0
    }
    Code Exec(PrologMachine mach) {
        return cl1.Exec(mach); % call the clause p(Cont) :- q(r(Cont))
    }
}
class pred_p_0_1 extends pred_p_0 { % p(Cont) :- q(r(Cont)).
    Code Exec(PrologMachine mach) {
        PrologObject continuation = mach.Areg[0]; % get Cont
        mach.Areg[0] = new Funct("r".intern(), continuation); % create r(Cont)
        mach.CUTB = mach.CurrentChoice;
        return q1cont; % call q/0
    }
}
public class pred_q_0 extends Code { % entry point of q/0
    static Code cl1 = new pred_q_0_1();
    Code Exec(PrologMachine mach) {
        return cl1.Exec(mach); % call the clause q(Cont) :- call(Cont).
    }
}
class pred_q_0_1 extends pred_q_0 { % q(Cont) :- call(Cont).
    Code Exec(PrologMachine mach) {
        mach.CUTB = mach.CurrentChoice;
        return UpperPrologMachine.Call1; % call Cont
    }
}
```

The PrologMachine class represents a Prolog engine and has a runtime environment such as argument registers (`Areg[1]` – `Areg[`n`]`), cut register (`CUTB`), choice point stack, trail stack, and so on. The `Code` class is a common superclass of all predicates. This class has the `Exec` method, and its argument is an object of currently activated Prolog engine. Continuation goal is always stored in `Areg[0]` at term level. Prolog cut (!) is implemented by the traditional way used in usual WAM-based Prolog systems. Translated code is executed in the following supervisor function:

```
code = goal predicate;
while (ExceptionRaised == 0) {
    code = code.Exec(this) ;   % this indicates Prolog engine
}
```

This function iteratively invokes the `Exec` method until the failure of all trials. The `Exec` method does certain actions referring to a Prolog engine and returns a continuation goal.

jProlog provides a simple and sound basis for translating Prolog into Java. It also supports intuitionistic assumption, backtrackable destructive assignment, and delayed execution. However, there are some points that can be improved since jProlog is an experimental system. For example, jProlog has no support for multi-threaded execution and does not incorporate well-known optimizations such as indexing and specialization of head unification.

2.2 LLPj

LLPj [12] is a Java implementation of a linear logic programming language. LLPj takes a different approach from jProlog for translating Prolog into Java.

In LLPj approach, each predicate with the same name/arity is translated into only one class, in which each clause is translated into one method. Each continuation goal is translated into a predicate object rather than a term. The previous example is translated as follows:

```
public class PRED_p_0 extends Predicate {
    public PRED_p_0 ( Predicate cont) {
        this.cont = cont;          % get Cont
    }
    public void exec() {
        if(clause1()) return;      % call the clause p(Cont) :- q(r(Cont))
    }
    private boolean clause1() {    % p(Cont) :- q(r(Cont)).
        try {
            Predicate new_cont = new PRED_r_0(cont);   % create r(Cont)
            (new PRED_q_0(new_cont)).exec();           % call q/0
        } catch (CutException e) {
            if(e.id != this) throw e;
            return true;
        }
        return false;
    }
}
public class PRED_q_0 extends Predicate {
    public PRED_q_0 ( Predicate cont) {
        this.cont = cont;          % get Cont
    }
    public void exec() {
        if(clause1()) return;      % call the clause q(Cont) :- call(Cont).
    }
    private boolean clause1() {    % q(Cont) :- call(Cont).
        try {
            cont.exec();           % call Cont
        } catch (CutException e) {
            if(e.id != this) throw e;
            return true;
        }
        return false;
    }
}
```

The `Predicate` class is a common superclass of all predicates that has the `exec` method and the `cont` and `trail` fields. These two fields are used to store a continuation goal and a trail stack respectively. The `exec` method invokes, in order, the methods corresponding to clauses. Thus, everything we need at runtime is maintained in each predicate locally, and there is no implementation of Prolog engine such as `PrologMachine` in jProlog. Prolog cut (!) is implemented by using Java's exception handlers `try` and `catch` block. Translated code is executed by invoking the `exec` method:

```
code = goal predicate;
code.exec();
```

LLPj is comparable in performance to jProlog for some Prolog benchmarks used in Section 3.4. The weakness of this approach is that the invocation of `exec` will invoke other nested `exec` methods, and never return until the system reaches the first solution. This leads to a memory overflow for large programs.

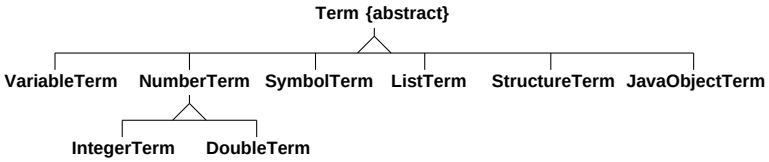

Fig. 1. Term Structure of Prolog Cafe

3 Prolog Cafe

In this section, we presents the Prolog Cafe system, especially its translation method and new features of interoperability and parallelism. In performance, we compare Prolog Cafe with jProlog for a set of classical Prolog benchmarks. We also give a brief introduction to the Multisat system.

3.1 Translating Prolog into Java

Each term is translated into a Java object of classes in Figure 1: `VariableTerm`, `IntegerTerm`, `DoubleTerm`, `SymbolTerm`, `ListTerm`, and `StructureTerm`. The `Term` class, which has an abstract method `unify`, is a common superclass of these classes. The `JavaObjectTerm` class is used to represent any Java object as a Prolog term.

The Prolog Cafe translator is based on the jProlog approach with featuring LLPj. Prolog Cafe incorporates the optimization of indexing (only first level) and specialization of head unification. Basically, each predicate is translated into a set of classes; one class is for an entry point of a predicate, and the other classes are for clauses and choice instructions [3]. Each continuation goal is translated into a predicate object rather than a term. Our previous example is translated as follows:

```
import jp.ac.kobe_u.cs.prolog.lang.*;
public class PRED_p_0 extends Predicate {
    public PRED_p_0(Predicate cont) {
        this.cont = cont;                    % get Cont
    }
    public Predicate exec(Prolog engine) {   % p(Cont) :- q(r(Cont)).
        engine.setB0();
        Predicate p1 = new PRED_r_0(cont);   % create r(Cont)
        return new PRED_q_0(p1);             % call q/0
    }
}
public class PRED_q_0 extends Predicate {
    public PRED_q_0(Predicate cont) {        % get Cont
        this.cont = cont;
    }
    public Predicate exec(Prolog engine) {   % q(Cont) :- call(Cont).
        return cont;                         % call Cont
    }
}
```

[3] It is noted that a predicate is translated into only one class in the deterministic case such as a predicate consisting in only one clause.

The output of Prolog Cafe consists of two classes and is smaller than that of jProlog. Translating continuations into predicate objects makes it possible to avoid the overhead of referring to the hash table. We can also avoid memory overflow occurred in LLPj since translated code is executed by a supervisor function.

Besides static predicates, Prolog Cafe can handle dynamic predicates and has provided a small Prolog interpreter. An implementation of `assert` and `retract` was discussed in the paper [4].

3.2 Interoperation with Java

It is possible to represent any Java object as a Prolog term, invoke its methods, and access to its fields by using the following built-in predicates:

- `java_constructor(+Class, ?Object)`
- `java_method(+Object, +Method, ?Return)`
- `java_get_field(+Object, +Field, ?Value)`
- `java_set_field(+Object, +Field, +Value)`

Object creation is performed by calling `java_constructor/2` where the first argument is an atom or compound term. A call to `java_constructor(`$f(a_1, \ldots, a_n)$`, O)` creates an object with the class name f and the argument objects $a'_1, \ldots, a'_n$. The created object is wrapped by `JavaObjectTerm` as a Prolog term and then unified with `O`. Each argument a_i is converted to a certain java object a'_i by the mapping that supports Java widening conversion. We omit the detail of data conversion here. Method invocation is performed by calling `java_method/3` where the first argument is an atom or Java object. It is noted that the second argument represents a static method when the first argument is an atom. A call to `java_method(O,` $g(b_1, \ldots, b_n)$`, R)` invokes a method with the method name g and the argument objects $b'_1, \ldots, b'_n$. The return value is wrapped by `JavaObjectTerm` as a Prolog term and then unified with `R`. Each argument b_i is converted to a certain Java object b'_i. Field access is performed by calling `java_get_field/3` and `java_set_field/3` in the similar way.

```
main :-
    java_constructor('java.awt.Frame', X),
    java_method(X, setSize(400,300), _),
    java_get_field('java.lang.Boolean', 'TRUE', True),
    java_method(X, setVisible(True), _).
```

For example, a call to `main` in the above code shows an empty frame on your screen.

3.3 Explicit Parallelism

Prolog Cafe provides multi-threaded Prolog engines. A Prolog Cafe thread seem to be conceptually an independent Prolog evaluator, in which a Prolog goal runs on a local runtime environment. The `PrologControl` class is an implementation

of Prolog Cafe thread and provides several methods for the control such as `start`, `stop`, and `join`. From the Prolog side, the control of thread can be programmed as follows:

```
start(Goal, Engine) :-
    java_constructor('PrologControl', Engine),
    copy_term(Goal, G0),
    java_wrap_term(G0, G),
    java_method(Engine, setPredicate(G), _),
    java_method(Engine, start, _).
stop(Engine) :- java_method(Engine, stop, _).
join(Engine) :- java_method(Engine, join, _).
sleep(Wait)  :- java_method('java.lang.Thread', sleep(Wait), _).
```

Creating thread is performed by call `start/2`. This predicate is similar to the `launch_goal/2` of Ciao Prolog [13] and the `new_engine/3` of Jinni [14]. A call to `start(Goal, Engine)` first creates a thread object and unifies it with `Engine` by using `java_constructor/2`, and it sets a copy of `Goal` to `Engine`, and then invokes the method `start` of thread object. Execution of `Goal` is thus proceeds on the new thread in an independent runtime environment. A call to `join` waits for the thread until the assigned goal succeeds or fails. The `stop/1` predicate is used to kill the thread. Calling the `sleep/1` causes the currently executing thread to sleep for the specified number of milliseconds.

Each goal, which runs separately on a thread, communicates through a shared Java object as a Prolog term. Synchronization can be done by the built-in predicate `synchronized/2`. A call to `synchronized(+Object, +Goal)` locks the `Object` and then execute `Goal`. This predicate is implemented by using the `synchronized` block of Java.

3.4 Experiments

We compare Prolog Cafe with jProlog. Table 1 shows the performance results of a set of classical Prolog benchmarks. A time of "?" in jProlog means that we met some errors during the compilation of generated Java code because of few built-in predicates. All times in millisecond were collected on Linux system (Xeon 2.8GHz, 2G memory) with JDK 1.5.0.

Prolog Cafe generates 2.7 times faster code than jProlog in average. This speedup is due to indexing, specialization of head unification, and first-call-optimization. For the purpose of reference, we add the execution time of SWI-Prolog to Table 1. SWI-Prolog is an efficient Prolog compiler system that compile Prolog into WAM. Compared with SWI-Prolog with `-O` option, Prolog Cafe is only 1.3 times slower for the `queens_10` (all solutions) and is about 3 times slower in average. It is noted that Prolog Cafe is about 10 times faster for the `tak` than SWI-Prolog without `-O` option.

Prolog Cafe is disadvantageous in performance compared with SWI-Prolog, but still has some possibilities for speedup. This is because Prolog Cafe is a simple implementation and does not incorporate well-known optimizations, such as register allocation, last-call-optimization, and global analysis.

Table 1. Comparison for Prolog Cafe vs jProlog vs SWI-Prolog

Prolog Programs	Runs	Avg. Prolog Cafe 0.9.3	jProlog 0.1	SWI-Prolog 5.0
`browse`	10	441.1	3011.8	218.0
`ham`	10	556.4	741.6	135.0
`nrev`(300 elem.)	10	34.1	115.3	15.0
`tak`	10	229.0	302.6	47.0
`zebra`	10	61.8	62.2	6.0
`cal`	10	223.7	?	36.0
`poly_10`	10	224.3	?	10.0
`queens_10` (all sol.)	10	462.9	?	344.0
`sendmore`	10	66.6	?	18.0
Average of Ratio	10	1.00	2.77	0.30

3.5 Multisat: A Parallel Execution System of SAT Solvers

Multisat [6] is a heterogeneous SAT solver integrating both systematic and stochastic algorithms, and includes solvers reimplemented in Java: Satz [15], Walksat [16], and Chaff [17]. In Multisat, multiple solvers run in parallel on Prolog

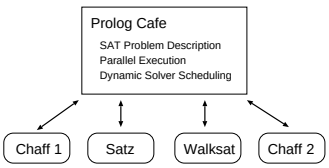

Fig. 2. A Multisat architecture

Cafe threads. Each solver can be dynamically created and killed from Prolog Cafe. Figure 2 shows a situation that Satz, Walksat and two Chaffs are created from Multisat. Main features of Multisat are as follows:

1. *Competitive Mode*
   Running multiple solvers in parallel, solvers compete with each other and share resources. No control is performed at all so that threads are processed equally. In the competitive mode, a solver that most fits the given problem firstly returns the output. Hence, solvers' strength and weakness for each SAT problem can be observed in this mode.
2. *Cooperative Mode*
   In the cooperative mode, solvers exchange derived clauses. Here, Chaff generates a new clause called a *lemma* when the current assignment leads to a conflict. Such a lemma is utilized to avoid the same conflict. However, un-

restricted incorporation of lemmas involves a memory explosion. Here, we further use these lemmas in other solvers that are running in parallel.

3. *Dynamic Preference*
   In Multisat, solvers run in parallel on Prolog Cafe threads. The status of each thread is either active or suspended. In default, each active solver is given the same amount of resources. In the scheduling system, active threads become suspended by method calls like `sleep()` and `wait()` or by input-output and synchronized blocks. On the other hand, scheduling threads is also possible according to their priorities using `setPriority` method.

Effectiveness of Multisat has been shown through applications to SATLIB problems, SAT planning, and job-shop scheduling. Multisat has solved SATLIB problems efficiently in average compared with single solvers. Multisat is particularly useful for SAT planning and job-shop scheduling because this type of problem suite should alternately handle satisfiable and unsatisfiable problems.

4 Related Work

In addition to jProlog, LLPj, and Prolog Cafe, a number of Prolog in Java have been recently developed: MINERVA, Jinni, W-Prolog, tuProlog, PROLOG+CG, JIProlog, JavaLog, DGKS Prolog, JLog, and XProlog. MINERVA [18] and Jinni [14] are commercial Prolog systems which compile Prolog into their own virtual machines that are executed in Java. W-Prolog [19] and others are implemented as interpreter systems. Each system has strength and weakness for our requirements. MINERVA and Jinni might be more efficient but are not possible to produce standalone executables. MINERVA has also no support for multi-threaded execution. Prolog Interpreters such as W-Prolog might be simpler but does not give nice performances. We thus decided to adopt the approach of translating Prolog into Java. This approach has the advantage of giving comparatively nice speedup in performance and producing standalone executables.

5 Conclusion

In this paper, we have presented the Prolog Cafe system, a Prolog-to-Java source-to-source translator system via the WAM. We have shown how Prolog Cafe translate Prolog into Java and new features of smooth interoperation with Java and explicit parallelism. In performance, our translator generates 2.7 times faster code for a set of classical Prolog benchmarks than an existing Prolog-to-Java translator jProlog. We have also given a brief introduction to the Multisat system as an application of Prolog Cafe. Hence, we conclude that Prolog Cafe can be well suited to develop Java-based intelligent applications such as mobile agent programs. Prolog Cafe is distributed as open source software under GPL license. The newest package (version is 0.9) is available from

`http://kaminari.istc.kobe-u.ac.jp/PrologCafe/`

There are some important future topics. Supporting ISO Prolog is necessary. Eclipse plug-in for Prolog Cafe can be a promising tool for improving usability.

References

1. Aït-Kaci, H.: Warren's Abstract Machine. MIT Press (1991)
2. Warren, D.H.D.: An abstract Prolog instruction set. Technical Report Technical Note 309, SRI International, Menlo Park, CA (1983)
3. Codognet, P., Diaz, D.: `WAMCC`: Compiling Prolog to C. In Sterling, L., ed.: Proceedings of International Conference on Logic Programming, The MIT Press (1995) 317–331
4. Banbara, M., Tamura, N.: Translating a linear logic programming language into Java. In: Proceedings of the ICLP'99 Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages. (1999) 19–39
5. Cook, J.J.: P#: a concurrent prolog for the .net framework. Software: Practice and Experience **34** (2004) 815–845
6. Inoue, K., Sasaura, Y., Soh, T., Ueda, S.: A competitive and cooperative approach to propositional satisfiability. DISCRETE APPLIED MATHEMATICS ((submitted))
7. Kawamura, T., Kinoshita, S., Sugahara, K.: Implementation of a mobile agent framework on java environment. In Gonzalez, T., ed.: Proceedings of the IASTED International Conference Parallel and Distributed Computing and Systems. (2004) 589–593 MIT, Cambridge, USA.
8. Barbosa, J.L.V., Yamin, A.C., Augustin, I., Vargas, P.K., Geyer, C.F.R.: Holoparadigm: a multiparadigm model oriented to development of distributed systems. In: Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS 2002). (2002) 6 pages
9. Wohlstadter, E., Jackson, S., Devanbu, P.T.: Generating wrappers for command line programs: The cal-aggie wrap-o-matic project. In: Proceedings of International Conference on Software Engineering. (2001) 243–252
10. Demoen, B., Tarau, P.: jProlog home page (1996) `http://www.cs.kuleuven.ac.be/~bmd/PrologInJava/`.
11. Tarau, P., Boyer, M.: Elementary Logic Programs. In: Proceedings of Programming Language Implementation and Logic Programming. Number 456 in Lecture Notes in Computer Science, Springer (1990) 159–173
12. Banbara, M., Tamura, N.: Java implementation of a linear logic programming language. In: Proceedings of the 10th Exhibition and Symposium on Industrial Applications of Prolog. (1997) 56–63
13. Carro, M., Hermenegildo, M.V.: Concurrency in prolog using threads and a shared database. In Schreye, D.D., ed.: Proceedings of the 15th International Conference on Logic Programming (ICLP'99). (1999) 320–334
14. Tarau, P.: Jinni: a lightweight java-based logic engine for internet programming. In Sagonas, K., ed.: Proceedings of JICSLP'98 Implementation of LP languages Workshop. (1998) invited talk.
15. Li, C.M., Anbulagan: Heuristics based on unit propagation for satisfyability problems. In: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI 1997). (1997) 366–371
16. B.Selman, H.Kautz, B.Cohen: Local search strategies for satisfiability testing. In D.S.Johnson, M, A., eds.: Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge. Volume 26 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society (1996) 521–531
17. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient sat solver. In: Proceedings of the 38th Design Automation Conference (DAC 2001), ACM (2001) 530–535
18. IF Computer: MINERVA home page (1996) `http://www.ifcomputer.com/MINERVA/`.
19. Winikoff, M.: W-Prolog home page. `http://goanna.cs.rmit.edu.au/~winikoff/wp/` (1996)