Title

Embedding Prolog in the Java environment

Authors

Miguel Calejo, João Pedro Sousa

Abstract

In this paper we present the development of a Java/Prolog interface, based on native methods, that allow Java applications to use an existing Prolog system. Future developments, including a browser plug-in to allow it's use within WWW pages, are also discussed.

A preliminary version of the Java/Prolog interface can be found at <http://www.servisoft.pt/dev/>.

Introduction

NanoProlog is a Prolog compiler based on a WAM implementation, written in C by Artur Miguel Dias at *Universidade Nova de Lisboa* (http://www-ctp.di.fct.unl.pt/~amd/nano/). It has been ported to several platforms, including: MacOS (using ThinkC 5.0), VAX VMS, NeXTstep, MIPS, Digital Alpha, Sun's SPARC, IBM's RS6000, and HP (all of these using *gcc*).

This system's small size and good performance make it easily manageable, making it a good fit for integration with another language. NanoProlog is freely available under the terms of the GNU General Public License. Our work, the Java API to the NanoProlog system, is also available under the same terms.

NanoProlog & Java

The objective of this project was the development of a Java API that would make the NanoProlog system easily accessible from Java applications.

The main disadvantage of this approach is that the NanoProlog system is entirely written in C, and as such, the only way for the system to be easily accessible from Java is by encapsulating its features in Java classes implementing native methods. Using native methods, we sacrifice one of the main characteristics of the Java language: portability. A Java class containing native methods must be ported to each platform in which it is intended to be run. As the NanoProlog system had already been ported to a wide range of platforms, we don't consider this to be as serious as it might have seam. Only some extra work is involved in porting the developed API to each of these platforms.

First Steps

Our implementation was done on MacOS 7.5, using Metrowerks CodeWarrior 9 for the C development, and using Sun's JDK 1.0.2 for the Java development.

As the original NanoProlog system had been developed on MacOS using ThinkC 5.0, the first step was porting the NanoProlog stand-alone C-written application to CodeWarrior. Soon, we had a NanoProlog system up-and-running, compiled to PowerPC native code using SIOUX (Simple Input/Output Extensions). Basically, SIOUX provides MacOS applications with a window emulating a standard terminal, so that we could work with standard `printf/scanf` and related functions.

API Specification

When specifying the Java API we wanted it to be as simple, clean, and close to the Prolog semantics as possible. These were its main requisites:

- Preserve traditional Prolog programming style by supporting Prolog streams integrated with Java's, assuming an ongoing Prolog computation context, instead of leading the programmer to think *"I should do top goals only"* as other APIs do.
- Encapsulate Prolog effectively and efficiently, to support real-world applications written in Java.
- From the perspective of the Java programmer, treat the NanoProlog system as a black-box as much as possible, with a clear and intuitive interface.

We wanted to be able to do something like the following excerpt in Java:

```
PrologEngine PE = new PrologEngine(...);
PE.doGoal("consult('input1.pl').");       // 'input1.pl' contains the
                                          // declaration of inlist/2
PE.goalTerm = new Term("inlist(X,[1,2,3,4])");
while (PE.executeGoal()) {
      System.out.println(PE.goalTerm);
      System.out.println(PE.goalTerm.initialVars[0]);
      System.out.println();
}
PE.terminateGoal();
```

Listing 1: Example of desired functionality.

The above simple example creates a new instance of a Prolog engine (parameters explained below), and then executes a goal. This particular goal is the consulting of a text file containing Prolog code. After this code has been loaded and executed, we initialize the `goalTerm` field with a new Prolog term and execute it until no more solutions are found. The call to the `terminateGoal()` method is always required, either when the search for more solutions is finished, or when we want to abruptly terminate the search.

To keep things as simple and separated as possible, we decided to specify three base classes:

1. One for the PrologEngine itself (`class Servisoft.PrologEngine`).
2. Another for the Prolog terms (`class Servisoft.Term`) representing a term read into the Prolog heap.
3. And finally, an exception to signal errors related to the execution of Prolog (`class Servisoft.PrologException extends Exception`).

The basic public members of these classes are as follows:

```
package Servisoft;

public class PrologEngine {

   public PrologEngine(int stacks,
                       int heap,
                       int maxSteps,
                       InputStream bootS,
                       InputStream inputS,
                       OutputStream outputS,
                       OutputStream errorS) throws PrologException;

   public Term goalTerm;

   public boolean executeGoal() throws PrologException;

   public void terminateGoal() throws PrologException;

   public int lastExecutionCalls();

   public native int availableHeap();

   public int availableStacks();

}

//---------------------------------------------------------------------------

public class Term {

   public Term(InputStream is);

   public Term(String s);

   public Term[] initialVars;

   public String toString();

   public Object primitiveType();

}

//---------------------------------------------------------------------------

public class PrologException extends Exception {

   public PrologException();

   public PrologException(String s);
}
```

Listing 2: Public members of the Java classes implementing the NanoProlog Java API.

Most of the methods are self-explanatory, but some remarks are in order.

When creating a `PrologEngine` we can specify a maximum number of inference steps that can be executed (third parammeter of the constructor). If this number is ever exceeded during the lifetime of the `PrologEngine` object, a `PrologException` is thrown. If no maximum number of steps is specified no checkings of this kind are made, and the system can work “forever”. This helps minimizing the occurrence of infinite loops.

Another safety limitation that can be introduced in a PrologEngine is the specification of the maximum amount of memory that can be used. Both the stack and the heap size can be specified.

An important parammeter in the constructors is the one related to the boot stream (parammeter `InputStream bootS`). This stream is completely read at boot (construction) time, and is used to install the main predicates of the NanoProlog system, including the top-level one. If this parammeter is null, a suitable file is provided with the correct contents.

The method `executeGoal()` uses the term stored in `goalTerm` and executes it (either the call or redo port), and returns `true` or `false`, depending on its execution. While it returns true, we can keep calling it and any free variables that exist on `goalTerm` may became instantiated; these are stored in the `initialVars` array of the `Term` object.

The method `terminateGoal()`can be called, either when the search for more solutions has finished (`executeGoal()` returned `false`), or when we want to abruptly terminate (cut) the search.

Based on these, some convenience methods were later developed, like the method `doGoal()` which can be used when we want a single solution from `executeGoal()`. It loads `goalTerm` with the goal to be executed (passed as a `String`), calls `executeGoal()`, and then calls `terminateGoal()`. The value returned is the one resulting from `executeGoal()`.

As previously mentioned, the `Term` class encapsulates a Prolog term read into the NanoProlog heap. This term might contain free variables, which are themselves stored in an array of `Term`’s inside the original `Term` object (array `initialVars`). We can know how many free variables existed originally on a `Term` by getting the value stored in `initialVars.length`.

The method `primitiveType()` returns a `String`, a `Integer`, or a `Float` object if the `Term` object is convertible to one of these basic Java types. If the `Term` is a non-instantiated variable, a structure or a list, it returns `null`.

From the viewpoint of a Java programmer, this specification seems reasonable:

- Limit on the number of inference steps (this helps preventing infinite loops).
- Pre-specification of the maximum amount of memory to be used.
- Easy accountability of the inference steps and memory used.

- Easy conversion of Prolog terms to Java basic types (`String`, `Integer`, and `Float`).

Example

The following example shows some of the functionality that we obtain by using tha NanoProlog Java API.

```
import java.io.*;
import Servisoft.*;


public class PEtest {

   public static void main() {
      FileInputStream  fis;
      FileOutputStream fos;
      PrintStream      ps;
      PrologEngine     PE;

      try {
         fis = new FileInputStream("input1.pl");
      } catch (FileNotFoundException fnfe) {
         System.out.println("Could not open 'input1.pl'!");
         return;
      }

      try {
         fos = new FileOutputStream("results.txt");
         ps = new PrintStream(fos);
      } catch (IOException ioe) {
         System.out.println("Could not create 'results.txt'!");
         return;
      }

      try {
         PE = new PrologEngine(
              /* Stack          --> */   256,
              /* Heap           --> */   512,
              /* Boot stream    --> */   null,
              /* Input stream   --> */   fis,
              /* Output stream  --> */   fos,
              /* Error stream   --> */   System.err);

      } catch (PrologException pex) {
         System.out.println("Error while creating a 'PrologEngine'");
         System.out.println("The error was: " + pex.getMessage() );
         return;
      }


      try {
         ps.println();
         PE.goalTerm = new Term("inlist(X,[9,8,7,6,5,4])");
         ps.println("Initial goal: " + PE.goalTerm);
         while (PE.executeGoal()) {
            ps.print("Current goal: " + PE.goalTerm);
            ps.print("   Inference steps used = " + PE.lastExecutionCalls());
            ps.println();
         }
         PE.terminateGoal();

         PE.doGoal("assert(fact1)");

         ps.println();
```

```
         if (PE.doGoal("fact1"))
            ps.println(PE.goalTerm + " is true.");
         else
            ps.println(PE.goalTerm + " is false.");

         ps.println();
         if (PE.doGoal("fact2"))
            ps.println(PE.goalTerm + " is true.");
         else
            ps.println(PE.goalTerm + " is false.");

      } catch (PrologException pex) {
         System.out.println("Error while executing a Prolog goal");
         System.out.println("The error was: " + pex.getMessage() );
         return;
      }

      ps.println();
      ps.println("-- the end --");
   }

}
```

Listing 3: Java code making use of the NanoProlog Java API.

The file "input1.pl" contains the following Prolog predicate:

```
inlist(X,[X|_]) :- true.
inlist(X,[_|T]) :- inlist(X,T).
```

Listing 4: Contents of the file "input1.pl" used in the example presented in Listing 3.

Finally, the result of executing the Java code presented in Listing 3 (file "results.txt") is as follows:

```
Initial goal: inlist(_0,[9,8,7,6,5,4])
Current goal: inlist(9,[9,8,7,6,5,4])   Inference steps used = 2
Current goal: inlist(8,[9,8,7,6,5,4])   Inference steps used = 1
Current goal: inlist(7,[9,8,7,6,5,4])   Inference steps used = 1
Current goal: inlist(6,[9,8,7,6,5,4])   Inference steps used = 1
Current goal: inlist(5,[9,8,7,6,5,4])   Inference steps used = 1
Current goal: inlist(4,[9,8,7,6,5,4])   Inference steps used = 1

fact1 is true.

fact2 is false.

-- the end --
```

Listing 5: Output file resulting from the execution of the example presented in Listing 3.

This small example shows how easy it is to integrate calls to the NanoProlog system from within Java code. Also note the easy integration of Java-based streams with Prolog-based streams.

Performance

In testing, we noticed a slight decrease in performance of Prolog code, when compared to the full native, non-Java based NanoProlog system. We attribute this mainly to the glue code and to the I/O subsystem.

In our implementation, most of the I/O done by the Prolog (native) side is transparently done through calls to Java streams. This gives us tremendous flexibility, as the Prolog side can transparently read or write to Java strings, Java based files, sockets, or whatever Java stream has been implemented.

The performance hit averaged 8-10%, mainly due to the native methods glue code overhead, but could reach as much as 20-25% in I/O intensive Prolog code.

Limitations

Presently we can only create an instance of a PrologEngine per each Java Virtual Machine. This happens so because the C source code for the NanoProlog is all written towards a single Prolog instance. Although NanoProlog supports threaded code, it's design did not foresee the concurrent execution of two (or more) Prolog threads on the same process. Neither should it: if you wanted a second instance of a NanoProlog system, all you had to do was launch a second copy from the command line. Another obstacle is the non-re-entrant nature of the NanoProlog code. There is no guarantee that while accessing the system concurrently by two diferent Java threads (each one dealing with it's own instance of a `PrologEngine`), the execution of one function (in behalf of one Java thread) wouldn't destroy the work being done by other functions somewhere else (in behalf of other Java threads).

It should be possible to separate all the WAM registers, stacks, and memory areas used in the NanoProlog implementation, and therefore achieve multiple-instance support by creating new sets of these as required; please see the section "Future Work", bellow.

Future Work

The next step to be taken is the development of a Netscape plug-in, to allow the use of the NanoProlog system by Java applets embeded in HTML documents. Using Netscape's browser LiveConnect architecture, Java applets can communicate with plug-ins (and JavaScript). This way it's possible to implement the NanoProlog Java API native methods in a plug-in, and then instantiate Java objects from classes that mirror this plug-in functionality. Prolog code will also become accessible to (dynamically interpreted) JavaScript scripts, allowing easy integration of Prolog code from within HTML pages. That is, it will be possible to create Prolog terms, pass them to the NanoProlog system for execution, and get the result back in JavaScript.

There have been some changes in the way native methods are implemented in JDK 1.1. Fortunately these changes converge to the way native methods are implemented in Netscape's plug-ins (and vice-versa).

The porting and adaptation of this API to JDK 1.1 is also a priority. A more ambitious goal would be the implementation of this API as a JavaBean, therefore making it usefull in new and exciting ways. Imagine, for example, a JavaBean implementing the NanoProlog API connected to another JavaBean implementing searches on a database of flight timetables. After "loading" the bean with the proper rules, we could find the most economical, the fastest, or the most efficient route for a specific voyage. Instead of a database of flight timetables we could have financial data, a knowledge base, etc. The level of reusability of this bean would be huge.

Conclusion

As can be seen, the integration of Java and the Prolog world is perfectly feasable, and brings advantages to both sides. As always, some compromises must be made, especially regarding performance and portability, but the gains are worth it.

Acknowledgments

This work was conducted at Servisoft within the "Inference Objects" project (JNICT PBIC/TIT/1207/92)