Prolog.NET Manual

Ali M. Hodroj
ali@hodroj.net

November, 11 2006

Abstract. Prolog.NET is an implementation of a Prolog for the .NET Framework. It allows interoperation between Prolog programs and other .NET languages. This lets Prolog and .NET programmers use .NET class libraries from Prolog, as well as compile Prolog predicates to call them from a .NET language. Another purpose of this project is to use Prolog as a declarative meta-language that can reason against object-oriented languages implemented on the .NET framework.

Prolog.NET can be downloaded from the Web site: http://prolog.hodroj.net. This manual is for release 0.2.

1 Introduction

The Prolog.NET language is an implementation of Prolog on the .NET Framework. The motivation behind this project was to study the issues concerned with implementing a logic programming language on the .NET framework. By combining the logic/declarative nature of Prolog with .NET's language interoperability, we make use of Prolog an efficient declarative meta-language that can be used to reason about the structure and behavior of .NET programs and assemblies.

2 Overview of Prolog.NET

The Prolog.NET system consists of a Prolog compiler and a runtime environment. The compiler (**prologc.exe**) is responsible for compiling Prolog programs into either abstract machine code to be executed by the runtime environment or .NET intermediate language executed directly by the .NET Common Language Runtime. The runtime environment is a .NET implementation of a Warren Abstract Machine with an extended instruction set that facilitates interoperability between Prolog and .NET. Upon installing Prolog.NET, the Compiler libraries as well as the runtime environment will be automatically installed in the Global Assembly Cache (GAC) by the installer. Thus, there's no need to copy the interpreter DLL file into the directory you're executing the compiled Prolog program in.

There are three output formats that a Prolog program can be compiled into using the Prolog.NET compiler: Executables (.exe), Class libraries (.dll), or Abstract Machine files (.xml). Both executables and class libraries will require the runtime environment installed in the global assembly cache in order to execute. However, the compiler also uses the ILMerge utility from Microsoft Research to merge the runtime environment library with the executable or DLL so that they can be run or linked on a system that doesn't have the runtime environment installed. Abstract machine files, are XML files that represent the Prolog programs in Warren Abstract Machine format. These files can be executed using the **prolog.exe** program, which is the main driver for the runtime environment interpreter.

3 Compiling Prolog programs into .NET assemblies

We can compile Prolog programs in to .NET assemblies using the Prolog.NET compiler. There are two types of assemblies that we can generate using our compiler: dynamically-linked libraries and executables. The .NET assemblies generated using our compiler will require the runtime library installed in the global assembly cache

(GAC). However, we can also use our compiler to merge the generated assembly with our runtime library so that we don't need to have the runtime available in the GAC. The type of assembly target can be specified via the **/target** switch in the **prologc** executable.

3.1 Generating a .NET Library Assembly (DLL)

In order to compile a Prolog program into a DLL assembly the **/target:dll** switch must be specified with the compiler. Every Prolog source code file is compiled into one .NET class. The class name has to be specified via the **class/1** directive in the source file. Once compiled, every Prolog predicate is compiled into a public class member method. Since predicates can either be true or false, we set the return types of the compiled methods to be **System.Boolean**. The examples below show how a Prolog program is transformed into a class in .NET IL:

```
:- class('MyMath').


factorial(0,1).

factorial(N,F) :- N > 0, M is N - 1, factorial(M,Fm), F is N * Fm.


fib(0,1).

fib(1,1).

fib(N,F) :- N > 1, N1 is N - 1, N2 is N - 2, fib(N1,F1), fib(N2,F2), F is F1+F2.
```

The class in table 1 above would be compiled via the following command:

```
prologc.exe /target:dll math.pro
```

into the following:

```
public class MyMath {
      public bool factorial(object arg1, object arg2) { … }
      public bool fib(object arg1, object arg2) { … }
}
```

We can also specify the namespace that the class exists in via the **namespace/1** directive in our Prolog program. The parameter type has type **System.Object** which allows us to pass arbitrary variable types to the predicate, this is known in .NET as "boxing". We introduce the **AbstractTerm** class type which is the equivalent of a Prolog variable on the .NET side as a container for a value obtained after unifying it with a predicate term. In order to use AbstractTerm, the **Axiom.Runtime** namespace must be used. The following is a list of Prolog/.NET equivalent data types that can be used to call a Prolog predicate:

Prolog Data Type	.NET Data Type
Integer	System.Int32
Constant	System.String
String	System.String
List	System.Collections.ArrayList
Structure	StructureTerm
Variable	AbstractTerm

An example of data conversion is shown in the example below after compiling the Prolog code from table 1 and using **factorial/2** and **fib/2** predicates in .NET:

```
using System;
using Prolog.Assembly;
using Axiom.Runtime; // required only for AbstractTerm

namespace PrologAndMath
{
    class Program
    {
        static void Main(string[] args)
        {
            MyMath math = new MyMath();

            // Calculate the factorial
            AbstractTerm a = new AbstractTerm();
            math.factorial(3, a);
            Console.WriteLine("Factorial of 3 is: " + a);

            // calculate the fifth fibonacci (should be 8)
            AbstractTerm fibValue = new AbstractTerm();
            math.fib(5, fibValue); // this is like fib(5, X) in Prolog
            Console.WriteLine("Fibonacci of 3 is: " + fibValue);
        }
    }
}
```

Every Prolog.NET release includes more examples like the above in the **samples** directory.

3.2 Generating a .NET Process Assembly (EXE)

The Prolog.NET compiler can compile Prolog programs into .NET executable assemblies via the /target:exe switch. By default the executable generated will require having Prolog.NET installed in order to execute, this is because it makes use of the Runtime library (which is installed in the GAC). However, you can provide the /static switch to the compiler to build a static executable that can be executed on a system that doesn't have Prolog.NET installed.

In Prolog.NET, the **main/0** predicate is the entry point of the executable when its run. For instance, the famous hello world program:

```
% hello world

main :- write('Hello, World!'), nl.
```

Would be compiled via:

```
prologc.exe /target:exe hello.pro
```

Once compiled, we can execute it vias:

```
C:\> hello.exe

Hello, World!

C:\>
```

The hello.pro program above can also be compiled into a static executable via the following command:

```
prologc.exe /static /target:exe hello.pro
```

The Prolog.NET compiler uses the ILMerge utility from Microsoft Research to merge the .NET executable with the Prolog.NET runtime libraries.

4 Calling .NET from Prolog

There are five built-in predicates in Prolog.NET that allow using .NET objects and methods from Prolog: **object/2, invoke/3, get_property/3, set_property/3,** and **::/2**. This section describes how each can be used in a Prolog program. In order to use the object-oriented predicates against classes and objects in an assembly or namespace, the **using/1** or **assembly/1** directives should be specified. The **using/1** predicate takes a .NET namespace as its only argument, while the **assembly/1** takes an assembly name. They both load an assembly so that the class types defined in it can be used from Prolog. The following are examples of each:

using('System.Collections').

assembly('MyMath.dll').

4.1 Object Instantiation

When the Prolog.NET runtime environment looks for a class to be instantiated, it first looks in the assemblies that have been loaded via **using/1**. If not found, it will then look in the assemblies loaded via **assembly/1**. To instantiate an object from .NET we can use the **object/1** built-in predicate:

object(+ClassName, -ObjectTerm).

The first argument is an atom of the class type which is created and placed in the second argument, known as an object term. The following is an example of creating an ArrayList in Prolog:

arraylist(X) :- object('System.Collections.ArrayList',X).

4.2 Invoking Methods

There are two ways to invoke methods in from Prolog, either using the **invoke/3** or **::/2** built-in predicates. The advantage of the latter is that it provides a more OOP-familiar syntax and its method's return value can be evaluated using the **is/2** predicate.

The **invoke/3** predicate takes three arguments. The first argument is the object term or the class type name in case we are invoking a static method. The second argument is a functor representing the method name as well as aguments passed to it. Finally, the third term is unified with the value returned from invoked:

invoke(+ClassTypeOrObject, +MethodName(+Arguments…),?ReturnValue).

The following is an example of invoking the **Add()** method from an example Calculator class that we implemented ourselves.

dotnet_add(X,Y,Z) :- object('Calc', O), invoke(O, 'Add'(X,Y), Z).

Since the return value is unified with the last argument, querying the following two goals is similar:

add :- dotnet_add(1,2,Z), Z = 3.

add2 :- dotnet_add(1,2,3).

The other way to invoke a method is by using the **::/2** built-in predicate, which has the following syntax:

+ClassTypeOrObject :: +MethodName(+Args…)

For instance, we can invoke the **Console.WriteLine()** method in the following way:

'Console'::'WriteLine'('Hello, World!').

We can also use the **::/2** predicate in an **is/2** expression. For example, rew-writing the above **dotnet_add/3** rule using **::/2** would look like:

dotnet_add(X,Y,Z) :- object('Calc',O), Z is O::'Add'(X,Y).

4.3 Getting/Setting Properties

We can get and modify that values of class properties, which are responsible for setting and getting non-public variables in a class. We use two predicates to achieve this: **get_property/3** and **set_property/3**. Both of these predicates are use almost the same way as the **invoke/3** predicate. The predicate signatures are:

get_property(+ClassTypeOrObject, +PropertyName, ?Value)

set_property(+ClassTypeOrObject, +PropertyName, +Value)

The first argument term of each predicate is the instantiated object or a class type (in case we are getting/setting a static property of a class). The second argument is an the property name. The third argument in **get_property/3** is unified with the value returned from the property, while in **set_property/3**, it's the value that the property needs to be set to. The following is an example of getting a property; We check if a **System.Collections.ArrayList** object has no elements by checking whether the **ArrayList.Count** property returns 0:

empty_array_list(X) :- get_property(X, 'Count', 0).

The following is an example of using set property to set the **Max** property of a **Calculator** class:

set_property('Calculator', 'Max', 3).

This is equivalent in C# to:

Calculator.Max = 3;

5 Built-In Prolog Predicates

The Prolog.NET library currently has a limited set of built-in predicates that are common to Prolog implementations. This section lists the predicates available.

5.1 Input/Output

- write/1
- writeln/1
- nl/0
- get0/1
- skip/1
- put/1

5.2 Comparison

- =/=/2
- =:=/2
- >=/2
- =</2
- </2
- >/2

5.3 Control

- call/1.

5.4 Equality/Unification

- =/2
- \=/2

5.5 Meta-Logic

- is/2
- atom/1
- bound/1
- char/1
- free/1
- var/1
- nonvar/1
- integer/1

5.6 .NET Interoperability

- object/2
- invoke/3
- ::/2
- get_property/3
- set_property/3

6 Related Links

The following is a list of links related to Prolog.NET:

Prolog.NET: http://prolog.hodroj.net/

ILMerge: http://research.microsoft.com/~mbarnett/ILMerge.aspx