

White Paper



Dynamic Language Runtime

Chandrasekhar Prabala

Abstract

Dynamic Language Runtime is Microsoft's new addition on top of the CLR which makes the creation and usage of Dynamic Languages on the .NET framework a lot easier. It also enables existing languages on the .NET framework like C#, add dynamic typing capabilities. This paper provides a high level overview of the DLR and how it works. This paper assumes that the reader has some background knowledge on dynamically typed languages.

Introduction

The front end of a compiler, famously known as the compile time portion of a compiler is divided into 4 phases:

- Lexical analysis
- Semantic checking
- Syntax checking
- Intermediate code generation

Lexical analysis is assigned with the task of breaking down the program string into useful pieces of information known as lexemes. Syntax checking verifies whether the code that has been written obeys the conventions that the language has adopted in terms of naming schemes, usage of language idioms, etc. It creates a syntax tree which is then taken by the intermediate code generation phase and converted into a language that the compiler can then use to make intelligent choices upon. Etched between these two phases is semantic checking. Semantic checking goes beyond keywords and their naming schemes and checks whether they have been used in the proper sense. Any language that is of practical importance today has a type system. A type gives meaning to data. It also specifies the operations that can be performed on that kind of data. When a type has been defined and a datum is associated with that type, any further manipulations on the datum has to obey the rules that are laid out in the definition of that type. Based on how a programming language performs this rule verification, also famously known as type checking, programming languages are classified into the following.

- Statically typed languages
- Dynamically typed languages

Statically typed languages

A language is said to be statically typed if the type checking is carried out at compile time. Static typing guarantees that a majority of type related programming errors are eliminated at compile time itself. Thus, one need not spend more time debugging and testing. Program efficiency can be reasoned and improved. Type related errors are notorious and can be difficult to debug, and static typing offers that protective layer. There is a related notion known of strong typing and weak typing. Strong typing checks if the types are correctly mixed and matched. It is more or less to do with automatic type conversion. Consider the following example:

```
"1" + 1
```

In a language like C# which is strongly typed, the above results in an error. The reason is that a string is being added to an integer. However, in a language like Perl, the above results in 2. The language Perl is weakly typed and the string "1" is converted into integer 1 and the addition is performed. In a language like C#, no such automatic conversions take place and the programmer is obliged to specify such type conversion requirements through code, if any.

Examples of statically and strongly typed languages include C#, Java, F# etc. Examples of statically typed and weakly typed languages are C/C++, Perl etc.

Dynamically typed languages

A language is said to be dynamically typed if a large part of the type checking is carried out at runtime and not at compile time. Proponents of dynamically typed languages argue that static typing offers a thin veil of protection from type errors but in that process ends up being conservative and thus loses on flexibility of type checking. One of the most common actions that involve dynamism is generation of types based on data that's available at runtime. Programs written using dynamically typed languages have to be well tested for type errors before they can be released. However these languages are very useful for unit testing purposes, cross platform scripting, runtime reflection, etc. Examples of dynamically typed languages include python, Java script, Lisp, Ruby etc. Refer to [4] for a motivation on static typing vs. dynamic typing in the context of modern high level languages.

Dynamism on the .NET platform

Though the .NET framework has had no out-of-the box support for true dynamism, there were ways by which dynamism could be achieved through programming. One way it could be achieved is by using reflection. Consider the following example:

There is a class called FactorialClass with a single method called Factorial.

```
public class FactorialClass
{
    public int Factorial(int n)
    {
        int fact = 1;
        while (n > 0)
        {
            fact = fact * n;
            n--;
        }
        return fact;
    }
}
```

If we have to invoke the method Add at run time using reflection the way we write code is given below.

```
Assembly assem = Assembly.GetExecutingAssembly();
Object fact= Activator.CreateInstance(assem.GetType("ReflectionExample.FactorialClass"));
Type factType = fact.GetType();
MethodInfo factMethod = factType.GetMethod("Factorial");
object res = factMethod.Invoke(fact, new object[] { 10 });
int factorial = Convert.ToInt32(res);
Console.WriteLine(factorial);
```

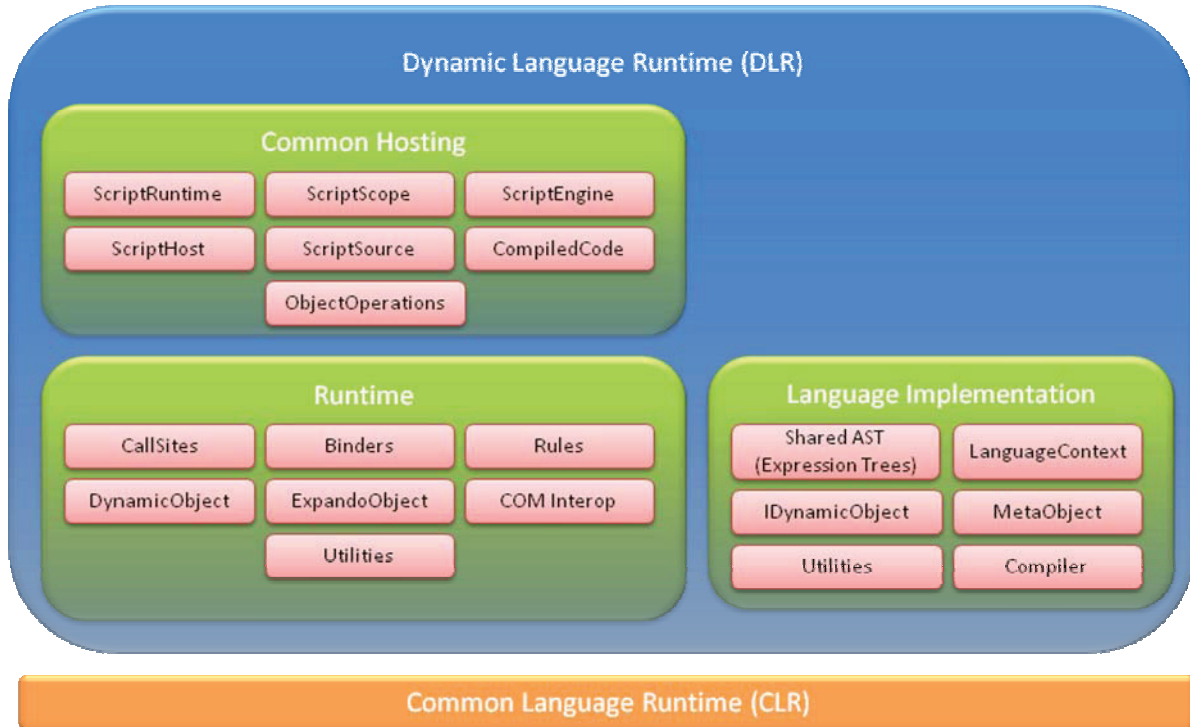
It's immediately clear how tedious and verbose it is to work with types dynamically at runtime.

Dynamic Language Runtime

The DLR (Dynamic Language Runtime) adds a set of libraries which makes doing the above much easy and fun. It adds a set of features designed specifically to meet the needs of dynamic languages such as a shared dynamic type system, a common hosting model and abilities to generate fast dynamic code and fast symbol tables.

In order to provide out-of-the box support for dynamic programming languages on the .NET framework, Microsoft has released the Dynamic Language Runtime (DLR) as part of .NET 4.0. The DLR sits on top of the CLR and has architecture as depicted below.

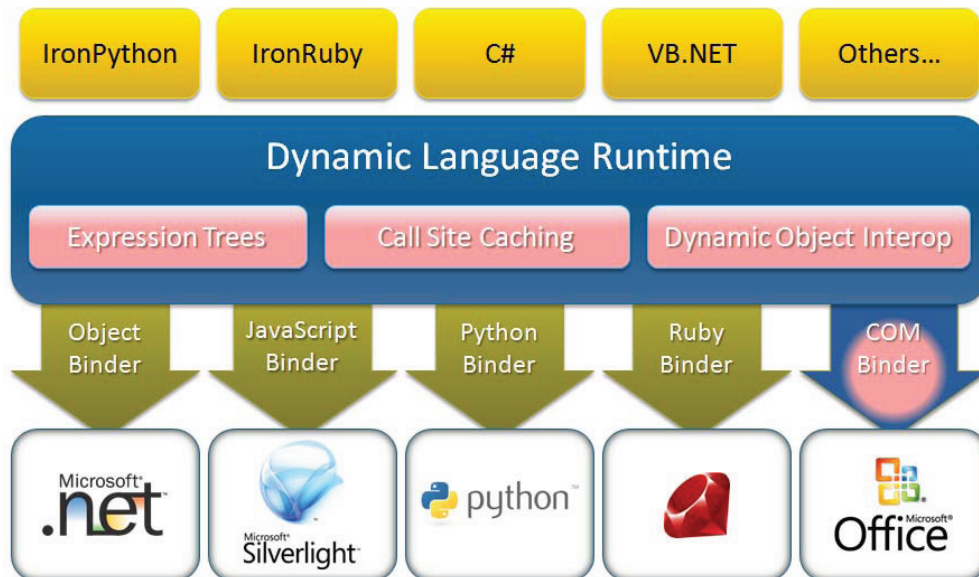
Fig 1. DLR Architecture (Source - www.codeplex.com/dlr)



A description of the architecture from a developer’s perspective is presented here. For a deeper understanding please refer to the documentation on codeplex [1].

The picture below shows how DLR fits into the big .NET picture.

Fig 2. (Source - www.codeplex.com/dlr)



The above picture gives a bird’s eye view of the architecture. At the top of the picture there are various languages that one can program in and right at the bottom there are various technologies and dynamic languages. The DLR with the help of appropriate binders enables data access across these various technologies and languages. Binders are discussed a bit down the line.

Following are 3 features of the DLR that are of primary importance to the developer.

- Dynamic Dispatch
- Call Site Caching
- Expression Trees

Dynamic Dispatch

Just because the DLR is implemented on top of the CLR, all objects do not get dynamic behavioural capabilities automatically. For an object to behave dynamically, the corresponding class should have dynamic dispatch capabilities. For a class to have dynamic dispatch capabilities, it needs to implement the `IDynamicObject` interface or simply inherit from the `DynamicObject` class and override a few of its methods. The following is the example of the `Medication` class that inherits from the `DynamicObject` class and overrides a couple of the class's methods.

```
public class Medication : DynamicObject
{
    private Dictionary<string, string> _Medicines = new Dictionary<string, string>();

    public override bool TryGetMember(GetMemberBinder binder, out object result)
    {
        result = _Medicines[binder.Name];
        return true;
    }

    public override bool TrySetMember(SetMemberBinder binder, object value)
    {
        _Medicines[binder.Name] = value.ToString();
        return true;
    }
}
```

The class `Medication` overrides two methods `TryGetMember` and `TrySetMember`, implementation of these methods helps the `Medication` class create properties and set their values dynamically. The `_Medicines` dictionary helps with holding the property and the associated value. The `Medication` class can now be used in the following fashion.

```
dynamic medication = new Medication();
medication.Name = "Paracetamol";
medication.Strength = "500mg";
medication.ExpiryDate = DateTime.Parse("12/12/2005");

console.WriteLine(" Prescribed Medication {0} is of strength {1} and its expiry
date is {2}" , medication.Name, Medication.Strength, medication.ExpiryDate);
```

Look how we gave our C# `Medication` class dynamic capabilities. We are able to add properties to it at runtime. The `DynamicObject` class has various other methods that one can override to add additional features. For example, if one overrides the `TryInvokeMember()` method one gets the capabilities to add methods to the object at runtime. The above code snippet also introduces the C# 4.0 dynamic key word. The `dynamic` keyword declares that the variable type should be resolved at runtime instead of at compile time. The above methodology provides us with obvious advantages.

- It simplifies code drastically.
It enables dynamism to creep into languages on the DLR.

Having said the above there are a few disadvantages as well.

- There won't be intellisense capabilities obviously because the type members are not known at compile time.
- There is no compile time type checking.
- Dynamic dispatch comes with a slight execution overhead.

Expression Trees

Intermediate language representations are mostly of two types. The first is a code representation, an example is 3-address code and the other is tree representation, an example is abstract syntax tree. .NET uses MSIL, an intermediate language which is a code based intermediate language representation. With the introduction of LINQ a tree based representation became necessary, thus expression trees were introduced. However, note that expression trees are finally converted into MSIL. C# provides facilities to express code as an expression tree. Expression Trees in DLR are the version next of the expression trees that were introduced along with LINQ in .NET 3.5 or they derive from that idea. Expression trees help in expressing code as data. Following is an example.

```
Expression<Func<int, int, int>> expression = (a, b) => a + b;
Console.WriteLine(" The left part of the expression: " +
"{0}{4} The NodeType: {1}{4} The right part: {2}{4} The Type: {3}{4}",
left.Name, body.NodeType, right.Name, body.Type, Environment.NewLine);
```

In the above example, the lambda expression (a,b) => a + b in code is expressed in a tree data structure. The above code yields an output as shown below.

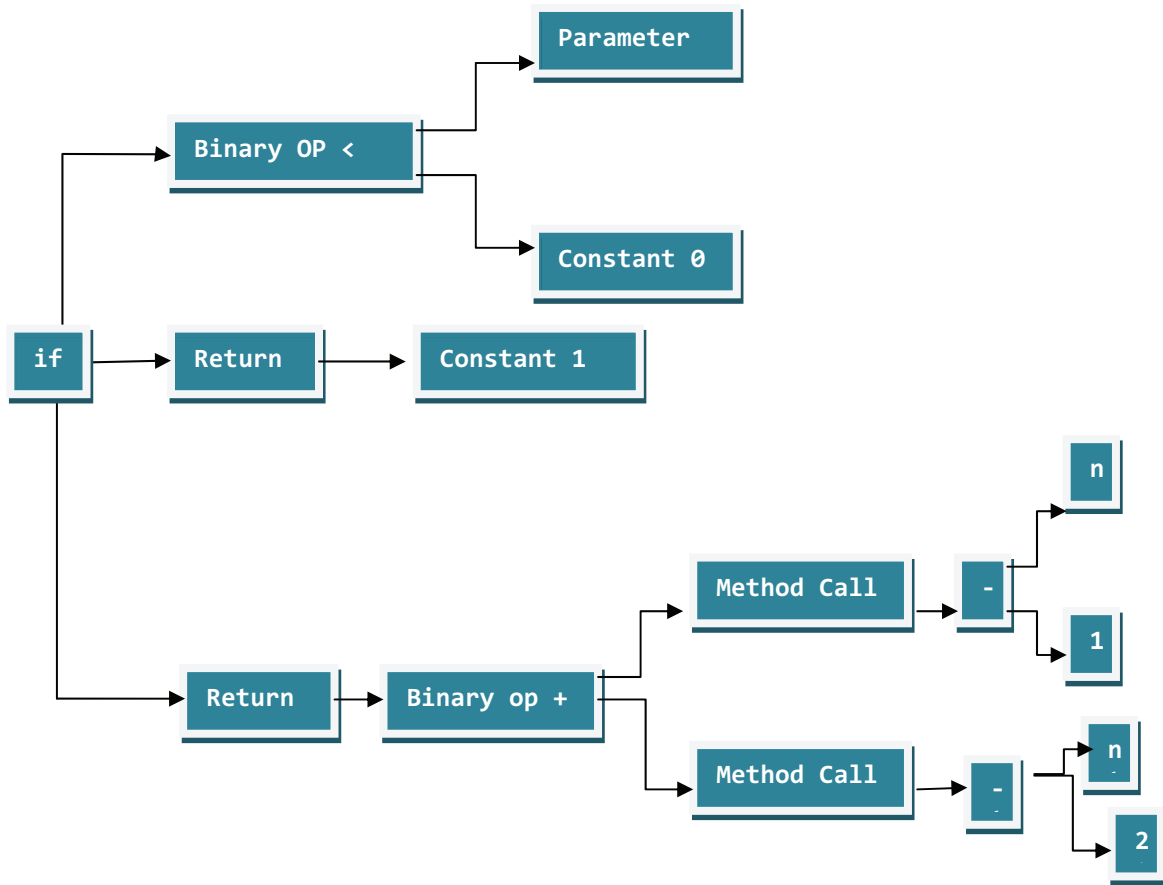
```
The left part of the expression: a
The NodeType: Add
The right part of the expression: b
The Type: System.Int32
```

Expression trees help in analyzing, transforming and composing the code. For example LINQ to SQL queries are taken and converted into T-SQL queries. PLINQ takes a LINQ query and runs it on multiple cores. The advantage is that the compiler now gets a chance to optimize the code and generate a very efficient machine code. This is the version of expression trees found in C#. In this version, the data at every node is static. The expression trees were extended to hold data, which would be discovered later, i.e. the nodes should be un-typed and late bound. Expression trees add more specialized nodes that support concepts unique to dynamic languages.

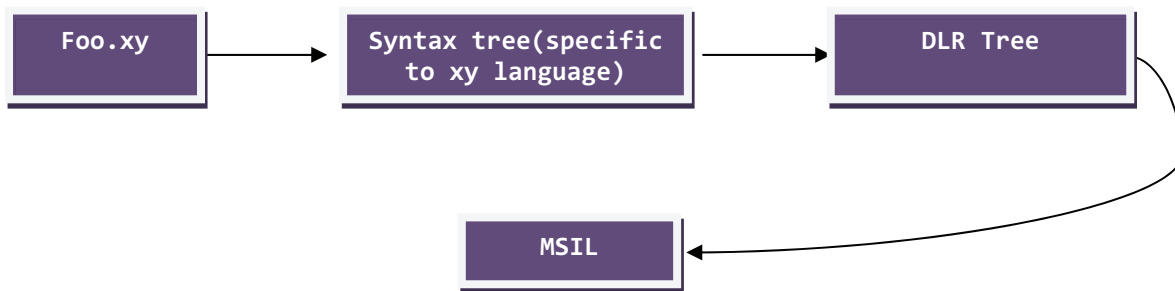
As an example consider the following piece of code that calculates a Fibonacci number.

```
int Fibonacci(int n)
{
    if (n < 1)
    {
        return 1;
    }
    else
    {
        return Fibonacci(n - 1) + Fibonacci(n - 2);
    }
}
```

The expression tree generated for this code is as shown in the figure.



Any language which wishes to make use of the DLR has to produce these Expression trees and not the IL. The DLR will take care of producing the necessary IL. Pictorially the idea is shown below.



Call Site Caching

Call sites are based on an old idea known as the polymorphic inline caching. Inline caching is a mechanism by which the necessary information is cached inline in the code. In DLR this idea was implemented using delegates and generics. Let's take an example to explain this. Let us say we had a piece of code as shown below.

```
if(x == 0) {...}
```

The way this is compiled into a call site in the DLR is that a site object is created in a static field

```
static CallSite<Func<CallSite, Object, int, bool>> _site = ...
```

This is compiled into code which looks like this. This calls a delegate on the site object that we have created earlier. Here, the `_site` object acts as the inline cache.

```
if(_site.Target(_site, x, 0)) {...}
```

The delegate `Target` points to some code which knows how to do the actual operation. An example is shown below:

```
static bool _0(CallSite site, object x, int y)
{
    return site.Update(site, x, y);
}
```

If we carefully look at the code we see that `x` is an object. Because it's a dynamic language, `x` need not be integer it can be anything say, `Bigint` or `long`. So `x` becomes an object. In any reasonable language `0` is an integer so `y` becomes an `int`. Since, if statement needs to return a Boolean value the return type of the operation is `bool`. Now if we look at the method body, the body calls a method `update` which updates the call site. This is where the speed comes from. Say, first time an integer is passed into `x`, the update method is generated in the following way.

```
static bool_1(CallSite site, object x, int y)
{
    if(x is int)
    {
        return (int)x == y;
    }
    else
    {
        return site.Update(site, x, y);
    }
}
```

The update method generated generates code for integer comparison. So, the next time `Integer` is passed into `x` again the execution is really fast. Say, now `Bigint` is passed, because this is a dynamic language as said earlier it's not necessary that one should pass integer input always. The target is now updated in the following way.

```
static bool_1(CallSite site, object x, int y)
{
    if(x is int)
    {
        return (int)x == y;
    }
    else if (x is BigInteger)
    {
        return BigInteger.op_equality((BigInteger)x, y);
    }
    else
    {
        return site.Update(site, x, y);
    }
}
```

This target now handles both integers and Bigint and now the third time, say something else is passed, target is updated again. This is the polymorphism part of the polymorphic inline cache. The first time the code is executed with integers there is no target code, thus a cache miss is said to have occurred. This is where a binder comes into the picture. Binder represents the language specific semantics for performing a specific operation at the callsite, including any metadata. Binders get called when there is a cache miss; they inspect the operands and compute how to perform the requested operation. Binders communicate with callsite using expression trees. From now onwards if the code is called with integers, the code that was generated to handle integers is reused and thus there is a cache hit. The next time the code is called with big integers again there is a cache miss and the binder comes into the picture. This is how call site caching works and this enables dynamic dispatch faster and thus makes, otherwise horribly slow dynamic languages work faster on the .NET framework. It should also be noted that this mechanism is built on top of the expression trees.

Iron Python & Iron Ruby

Python and Ruby are often quoted as examples for dynamic languages. These are implementations of python/ruby on the .NET CLR. They are true to their original language specifications and do not modify any of the features of the language. Introductions to these languages are out of the scope of this paper.

C# 4.0

C# 4.0 introduced a lot of new features. One of the most important amongst those is the 'dynamic' keyword. With the introduction of the dynamic keyword the calculator example presented earlier can be re written in the following way.

```
dynamic fact = new FactorialClass();
int fac = fact.Factorial(5);
Console.WriteLine(fac);
```

'dynamic' in the above code is a static type. Whenever this keyword is used, the keyword tells the compiler that the type will be resolved at runtime and all activities concerning the type will be handled by the type and the compiler can blissfully stay away.

The above code has the same effect as that of the code that used reflection seen earlier. The addition of dynamic capabilities to C# enables the language to support very nice expression when working with dynamic objects via COM, HTML DOM and .NET Reflection.

Conclusion

Dynamic Language Runtime is a welcome addition on top of the .NET framework. This addition brings more variety to the framework and will attract those who believe in the Pythons' and the Rubys' to .NET. The DLR can be used to build custom Domain Specific Languages (DSL) which are inherently dynamic in nature. All that one needs to do is write the parser in such a way that it outputs an expression tree. The DLR can also be used to enable meta- programming where code is generated as and when required and lives shortly in the memory, does the necessary task and deletes itself. Meta programming is the next big step after Model Driven Development (MDD). More importantly DLR will make the life of an average C# and VB developer easier by letting them work with COM, JSON, XML objects. It makes the web developer make use of a language like Python to create robust web applications with ease. It also opens up a host of possibilities. In time we can see java objects being consumed in .NET if an appropriate binder is written.

References

1. www.codeplex.com/dlr
2. [http://msdn.microsoft.com/hi-in/vsx/default\(en-us\).aspx](http://msdn.microsoft.com/hi-in/vsx/default(en-us).aspx)
3. Compilers principles, Techniques and Tools, Aho, Sethi and Ulman(Dragon book)
4. Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages.

About the Author

Chandrasekhar Prabala is a senior software engineer in the High performance computing (HPC) group with Infosys' Microsoft Technology center (MTC). He has more than 2 ½ years of software development experience in F#, C#, ASP.NET and WCF.

Acknowledgements

I profusely thank S. Naveen Kumar (Principal Architect, MTC) and Sudhanshu M. Hate (Senior Technical Architect, MTC) for their support and valuable guidance in bringing out the paper in the current form.

I would also like to thank Atul Gupta (Principal Architect, MTC) and Sripriya Thothadri (Technical Architect, MTC) for helping out at various stages during the review process.



For more information, contact askus@infosys.com

About Infosys

Many of the world's most successful organizations rely on Infosys to deliver measurable business value. Infosys provides business consulting, technology, engineering and outsourcing services to help clients in over 30 countries build tomorrow's enterprise.

For more information about Infosys (NASDAQ:INFY), visit www.infosys.com.