Dot-Scheme
A PLT Scheme FFI for the .NET framework

Pedro Pinto
Blue Capital Group
105 Concord Dr., Chapel Hill, NC
27514
+ 1 919 9606042 ex17
pedro@bluecg.com

ABSTRACT

This paper presents the design and implementation of dot-scheme, a PLT Scheme Foreign Function Interface to the Microsoft .NET Platform.

Keywords

Scheme, FFI, .NET, CLR.

1. INTRODUCTION

Scarcity of library code is an often cited obstacle to the wider adoption of Scheme. Despite the number of existing Scheme implementations, or perhaps because of it, the amount of reusable code directly available to Scheme programmers is a small fraction of what is available in other languages. For this reason many Scheme implementations provide Foreign Function Interfaces (FFIs) allowing Scheme programs to use library binaries originally developed in other languages.

On Windows platforms the C Dynamic Link Library (DLL) format has traditionally been one of the most alluring targets for FFI integration, mainly because Windows's OS services are exported that way. However making a Windows C DLL available from Scheme is not easy. Windows C DLLs are not self-describing. Meta-data, such as function names, argument lists and calling conventions is not available directly from the DLL binary. This complicates the automatic generation of wrapper definitions because it forces the FFI implementer to either a write C parser to extract definitions from companion C Header files, or, alternatively, to rely on the Scheme programmer to provide the missing information.

Permission to make digital or hard copies, to republish, to post on servers or to redistribute to lists all or part of this work is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To otherwise copy or redistribute requires prior specific permission.

Fourth Workshop on Scheme and Functional Programming.
November 7, 2003, Boston, Massachusetts, USA.
Copyright 2003 Pedro Pinto.

These issues, along with other problems including the mismatch between C's manual memory management and Scheme's garbage collection, make the use and implementation of C FFIs difficult tasks.

Recently the advent of the .NET platform [10] has provided a more attractive target for Windows FFI integration. The .NET platform includes a runtime, the Common Language Runtime or CLR, consisting of a large set of APIs covering most of the OS functionality, a virtual machine language (IL) and a just-in-time compiler capable of translating IL into native code. The CLR offers Garbage Collection services and an API for accessing the rich meta-data packaged in CLR binaries. The availability of this meta-data coupled with the CLR's reflection capabilities vastly simplifies the implementation and use of Scheme FFIs.

The remainder of this paper will illustrate this fact by examining the design and implementation of dot-scheme, a PLT Scheme [7] FFI to the CLR. Although dot-scheme currently targets only PLT Scheme, its design should be portable to any Scheme implementation that can be extended using C.

2. Presenting dot-scheme

The dot-scheme library allows the use of arbitrary CLR libraries, also called assemblies, from Scheme. Consider the following C# class:

```
using System;
public class Parrot
{
   public void SayHello(string name)
   {
      Console.WriteLine ("Hello {1}.", name);
   }
}
```

Assuming this code is compiled into an assembly, `parrot-assembly.dll`, then the `Parrot` class is available from the following Scheme code:

```
(require (lib "dot-scheme.ss" "dot-scheme"))

(import-assembly "parrot-assembly")
(:say-hello (new ::parrot) "John")

> Hello John.
```

The meaning of the Scheme code should be easy to infer. After importing the dot-scheme library bindings the program uses the `import-assembly` macro to load the `parrot-assembly` binary:

```
(import-assembly "parrot-assembly")
```

When loading is completed, `import-assembly` iterates through all the types contained in `parrot-assembly` generating appropriate wrapper bindings. The identifier `::parrot` is bound to a Scheme proxy of an instance of the CLR Type class. This instance contains meta-data associated with the `Parrot` type and can be used as an argument to the `new` function:

```
(new ::parrot)
```

The `new` function constructs an instance of the Parrot class and returns a proxy. This proxy is then used as the first argument (corresponding to the "this" pointer in C#) of the `:say-hello` function:

```
(:say-hello (new ::parrot) "John")
```

The second argument is a Scheme string. Internally `:say-hello` will extract the actual Parrot reference from the first argument, convert the Scheme string to a CLR string and then invoke the `SayHello` method on the `Parrot` object. The result of this call is a CLR string which is automatically converted to a Scheme string and returned to the top level.

In general, using a CLR assembly is fairly straightforward. The user needs only to specify which assembly to import and dot-scheme will load it and generate the appropriate wrapper code. This is true not only in this toy example but also when importing definitions from large, complex libraries such as the CLR system assemblies. For example using any one of the three-hundred plus types that comprise the CLR GUI framework is just as simple:

```
(require (lib "dot-scheme.ss" "dot-scheme"))
(import-assembly "system.windows.forms")
(::message-box:show "Hello!")
```

In general using CLR types through dot-scheme is no harder than using regular PLT Scheme modules [5]. In fact it is possible to muddle the distinction between PLT modules and CLR assemblies:

```
(module forms mzscheme
(require (lib "dot-scheme.ss"
              "dot-scheme"))
(import-assembly "system.windows.forms")
(provide (all-defined)))
```

This code above defines a PLT Scheme module named `forms`. When the module is loaded or compiled the expansion of the `import-assembly` macro creates a set of bindings within the module's scope. These bindings are exported by the declaration `(provide (all-defined))`. Assuming the module is saved in a file `forms.ss` and placed in the PLT collections path then access to the CLR GUI from Scheme simply entails:

```
(require (lib "forms.ss"))
(::message-box:show "Hello again!")
```

There is not much more to be said about using dot-scheme. Scheme's macro facilities coupled with the richness of the meta-data contained in CLR assemblies make it possible to generate Scheme wrappers from the binaries themselves. The power of this combination is apparent in the simplicity with which CLR types can be used. Perhaps even more striking though is how straightforward it is to achieve this level of integration. This should become apparent in the next section.

2.1 High-level architecture

The dot-scheme architecture can be thought of as defining two layers:

- A core layer, responsible for managing storage of CLR objects as well as CLR method dispatch. This layer is implemented in 1200 lines of Microsoft Managed C++ (MC++).
- A code generation layer, responsible for generating wrapper bindings for CLR types. These wrappers are implemented in terms of the primitives supplied by the core layer. The code generation layer is implemented in 700 lines of Scheme.

2.2 The Core Layer

Dot-scheme memory management and method dispatch are implemented in a PLT extension. A PLT extension is a DLL written in C/C++ and implementing Scheme callable functions [6]. These functions can use and create Scheme objects represented in C/C++ by the type `Scheme_Object`. At runtime Scheme programs can dynamically load extensions and use extension functions as if those functions had been defined in Scheme.

Using Microsoft's Managed C++ (MC++), a dialect of C++ which can target the CLR, it is possible to create a PLT extension that targets the CLR. Such an extension is able to use any of the CLR types as well as any of the library calls provided by the PLT runtime. From the point of view of the Scheme process that loads the extension the usage of the CLR is invisible. The extension functions initially consist of a small stub that transfers control to the CLR runtime. When the function is invoked for the first time the CLR retrieves the associated IL, translates it to machine code and replaces the method stub with a jump to the generated code.

The core layer is implemented in MC++ and so can bridge the PLT Scheme and CLR runtimes.

2.2.1 *Object Representation*

The first challenge faced when attempting to use .NET from Scheme is how to represent CLR data in Scheme. There are two categories of CLR data to consider. Primitive types such as integers, doubles, Booleans and strings have more or less direct equivalents in the Scheme type system and so can simply be copied to and from their Scheme counterparts. Non primitive CLR types present a more interesting problem. The CLR type system consists of a single-rooted class hierarchy where every type is a subclass of Object (even primitive types such as integers and floats can be boxed, that is their value, along with a small type descriptor, can be copied to the heap and Object references used to access it). Object life-time in the CLR is controlled by a Garbage Collector. To understand the interaction between the Scheme Garbage Collector and the CLR it is first necessary to examine the CLR's Garbage Collection strategy.

The CLR Garbage Collector is activated when the CLR heap reaches a certain size threshold. At this point the Garbage Collector thread will suspend all other threads and proceed to identify all objects that are reachable from a set of so called roots. Roots are Object references that are present in the stack, in static members or other global variables, and in CPU registers. Objects that are not reachable from this set are considered collectable and the space they occupy in the heap is considered empty. The Garbage Collector reclaims this space by moving objects to the empty space[1] and updating all changed references.

Clearly for this algorithm to work the Garbage Collector must be aware of all active Object references within a process. As a consequence it is not possible to pass Object references to code that is not running under the control of the CLR. This includes the Scheme runtime and so, to represent CLR Objects, another level of indirection is needed. Dot-scheme implements this indirection by storing references to CLR Objects in a CLR hash table using an integer key.

[1] This is a very simplified explanation. For details see [8]

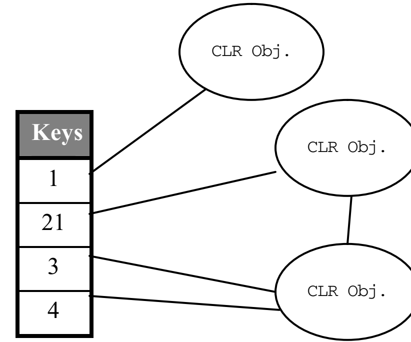

Figure 2. CLR Object management

This key can be packaged in a `Scheme_Object`, which can then be associated with a finalization callback and registered with the Scheme garbage collector. As long as the CLR Object remains in the hash table it will not be collected by the CLR (since the hash table itself is referenced by a static variable, and therefore is reachable from a root). When the Scheme runtime determines that the reference is no longer needed the finalization callback is invoked. At this point the associated integer key is used to locate and remove the CLR Object reference from the hash table. Other references to the same Object may exist either in the hash table or in other CLR objects and so the Object may continue to exist until all references go out of scope.

Notice that it will take at least two independent garbage collections for an Object to be collected, one by the Scheme runtime and one by the CLR, but otherwise this process is undistinguishable from regular Scheme garbage collection.

In terms of CLR Object representation the above is almost all that is necessary. One subtlety remains though. Consider the following classes:

```
class A
{
   virtual string SayName()
   {return "A";}

   string SayNameAgain ()
   {return "A";}
}
class B : public A
{
   override string SayName()
   {return "B";}

   string SayNameAgain()
   {return "B";}
}
```

Now assume an instance of `B` and an instance of `A` are created and references of type `A` associated to each of them:

```
A a1 = new B();
A a2 = new A();
```

What happens when `SayName` is invoked? It depends:

```
a1.SayName2() -> returns "B"
a2.SayName2() -> return "A"
```

This result should not come as a surprise. Basic dynamic dispatch is happening here. At run-time the CLR dispatches on the type of the Object pointed by `a1` and `a2`. Using the Object representation strategy described above it would be easy to wrap the `SayName` method in Scheme and still support dynamic dispatch. Assuming `a-ref` is a reference to a subtype of `A` then:

```
(say-name2 a-ref)
```

could be implemented in a PLT extension as follows[2]:

```
Scheme_Object SayName2(Scheme_Object a)
{
   A a = (A) DecodeCLRObject (a);
   return EncodeCLRObject(a.SayName2());
}
```

where `EncodeCLRObject` returns a CLR Object given an integer key (packaged in a `Scheme_Object`) and `DecodeCLRObject` stores a CLR Object and returns its key packaged as a CLR Object. In this particular case this strategy would produce the correct result. However there is another scenario. Consider the following code:

```
B bRef = new B();
A aRef = bRef;
```

what should happen when `SayNameAgain` is invoked? Again, it depends:

```
bRef.SayNameAgain () -> returns "B"
aRef.SayNameAgain () -> return "A"
```

Despite the fact that an identically named method was invoked on the same Object on both calls the actual method that is executed depends on the type of the reference used to access the Object. In contrast with Scheme where only values have types, in the CLR both an Object and the references used to manipulate it have a type. Furthermore both have a role in method dispatch, the first at runtime and the second at compile time[3]. In Scheme variables are typeless. Assume `SayNameAgain` is invoked from Scheme on a reference to an instance of `B`:

```
(say-name-again b-ref)
```

In this case problem a problem arises. What should `say-name-again` do? Should it invoke `A.SayNameAgain` or `B.SayNameAgain`? The runtime type of `b-ref` is insufficient to make this decision, so somehow it is necessary to convey the missing type information.This can be done in two ways: the reference type can be implied in the function call, for example by creating two functions, `say-name-again-A` and `say-name-again-B`, or it can be encoded in the reference itself. The latter approach is more natural for users familiar with the CLR and leads to simpler code and so is the one preferred by dot-scheme.

In terms of the structures described above only a small change is required. Instead of a CLR Object the hash table mentioned must store an object reference structure consisting of an Object and Type pair:

```
class ObjRef
{
   ObjRef (Object o, Type t)
   {obj = o; type = t;}

   Object obj;
   Type type;
}
```

The `obj` reference plays the same role as before pointing to the CLR Object of interest. The `type` reference is used to record the dispatch type associated with `obj`.

With typed references comes the need to occasionally circumvent the type system. In dot-scheme the `Cast` function changes the

[2] Throughout this paper we will present examples in C# despite the fact that the dot-scheme core layer is implemented in MC++. MC++'s syntactic noise would likely cause distraction without offering any additional insights.

[3] Note that this issue can be seen as a special case of the general problem of resolving overloaded method calls. The CLR resolves such calls at compile time based on the static types used in the method call.

dispatch type associated with a CLR reference. Implementing `Cast` is straightforward:

```
Scheme_Object Cast (Scheme_Object o,
                    Scheme_Object typeName)
{
   Type tt =
      Type.GetType(toCLRString(typeName));

   ObjRef or = DecodeCLRObject(o);

   If (tt.IsAssignableFrom(or.obj.GetType()))
      return
         EncodeCLRObject(new ObjRef(obj,tt));
   else
      throw Exception ("Invalid cast");
}
```

The code above starts by using the CLR Type class to retrieve the Type instance associated with the class named by `typeName`. After checking the validity of the cast the code then creates and returns a new `ObjRef` containing the same CLR Object but a different dispatch type.

This addition completes the description of dot-scheme's memory management strategy. To summarize, the problem of representing CLR objects in Scheme can be reduced to the implementation of the following interface:

```
Scheme_Object EncodeObjRef (ObjRef);
ObjRef DecodeObjRef (Scheme_Object ref);
void RemoveObjRef (Scheme_Object ref);
Scheme_Object Cast(Scheme_Objectm ref,
                   Scheme_Object typeName);
```

The semantics of these operations should be clear:

- `EncodeObjRef` will add the `ObjRef` passed as an argument to an internal hash table and package the associated integer key in a `Scheme_Object`. This object is then registered with the Scheme garbage collector by associating it with the `RemoveObjRef` finalization callback.
- `DecodeObjRef` will extract an integer key from the passed `Scheme_Object` and return the `ObjRef` that is associated with it.
- `RemoveObjRef` will obtain an integer key from its argument in the same way as `DecodeObjRef` but instead of returning the associated `ObjRef` it will remove it from the internal hash table.
- `Cast` creates a new `ObjRef` based on the one passed as an argument. The new `ObjRef` will be associated with the type named by `typeName`.

2.2.2 Method dispatch

Dynamic method dispatch is a complex process [2]. Consider the steps required to determine what method should be invoked by the C# code below:

```
obj.f(arg2, arg3, ... argn)
```

First, at compile time, the type of the reference `obj` is located. Within the type's scope the compiler will search for a method named `f`. Since the CLR supports method overloading several candidate methods may exist. The compiler must use the static types of the `arg2…argn` expressions to select between candidate methods. This disambiguation process is not entirely straightforward. Because of sub-typing it is still possible for several identically named methods to have signatures that are compatible with the types of the expressions `arg2…argn`. In this case the C# compiler will try to select the "most specialized" method, i.e. the method whose formal argument types are closer in the inheritance tree to the actual argument types. If a decision is possible there is still one more step. If the method found is not virtual then the compiler will emit a direct method call. If the method is virtual then final resolution is deferred until run time and the compiler simply records the index of the virtual method found. At runtime this index will be used to retrieve the corresponding method from the method table associated with the Object referenced.

In order to stay close to normal CLR semantics dot-scheme emulates this lookup process. Since Scheme is dynamically typed no reference types are available at compile time and so all the steps above have to be performed at run-time.

Dot-scheme makes use of the CLR reflection API to implement most of this process. The reflection API offers methods allowing the retrieval of type information and the invocation of methods on types which can be unknown at compile time. Every CLR Object implements a `GetType` method which returns an instance of the CLR Type class holding type information for the specific CLR Object. Using this Type Object it is possible to locate and invoke methods on the original instance. Dot-scheme relies on these capabilities to implement its dispatch algorithm. A simplified version of this algorithm is presented below. For brevity, error processing and handling of void return types are omitted.

```
ObjRef Dispatch (String methodName,
                 ObjRef self,
                 ObjRef [] args)
{
   Type [] argTypes = new Type [args.Length];
   Object []argValues = new Object[args.Length];

   For (int n = 0; n < args.Length; i++) {
      argTypes[n] = args[n].type;
      argValues[n] = args[n].obj;
   }

   MethodInfo mi =
      self.type.GetMethod (methodName,
                           argTypes);

   Object result =
      mi.Invoke(self.obj,methodName, args);

   return new ObjRef (result,
                      mi.GetReturnType());

}
```

The first step taken by `Dispatch` is extracting the types of the arguments used in the call. Note that the reference types, not the actual argument types are used. Using the extracted types `Dispatch` queries the Type Object associated with the reference on which the method call is invoked. `Type.GetMethod` is a CLR method which implements the method lookup algorithm described earlier. The result of `GetMethod` is a `MethodInfo` instance which contains meta-data associated with the method found. `MethodInfo.Invoke` is then used to execute the method. The result of this call, along with the associated reference type, is packaged in an `ObjRef` and returned.

Note that despite the relative complexity associated with method dispatching, the above code is straightforward. All the heavy lifting is done by the CLR. The reflection API is used to locate the appropriate method and, once a method is found, to construct the appropriate stack frame and transfer control to the target method.

Dot-scheme actually implements two additional variations on the dispatch code above, one for dispatching constructor calls and another for dispatching static member calls, but in essence its dispatch mechanism is captured in the code above.

2.2.3 The Core API

As mentioned earlier the core layer is implemented through a PLT Scheme extension. This extension implements the object management and dispatch mechanisms described above. In order to make its services available to the Scheme runtime the core layer exports the following Scheme constructs:

- `obj-ref`. `obj-ref` is a new Scheme type. Internally this type simply packages an integer key that can be used to retrieve ObjRef instances.
- `(cast obj-ref type-name) -> obj-ref`. The `cast` function takes as argument an `obj-ref` and a string. `obj-ref` indicates the Object reference that is the source of the cast and the string names the target type.
- `(dispatch self method-name arg…) -> result | void`. The `dispatch` function will invoke the instance method named by the `method-name` string on the CLR Object associated with the first `arg` passing the remaining parameters as arguments. Internally the implementation will examine both the `self` and `arg` parameters to determine if they correspond to `ObjRef's` or Scheme primitive types. In the first case the associated `ObjRef` instance is retrieved. In the second case a new `ObjRef` is created and the Scheme value is copied to the equivalent CLR Type. The resulting list of arguments is then passed to the MC++ dispatch call described earlier. The result of the method call, if any, is either copied to a Scheme value and immediately returned or encoded as an integer key and returned.
- `(dispatch-static type-name method-name arg …) -> result | void`. `dispatch-static` is similar to `dispatch-instance` but in this case there is no self reference. Instead the type named by the string `type-name` is searched for a static method named `method-name`.
- `(dispatch-constructor type-name arg…) -> result | void`. `dispatch-constructor` is similar to `dispatch-static` except for the fact that a constructor method is implied.

This API is sufficient to provide access to almost all of the CLRs functionality. The rest of dot-scheme's implementation simply provides syntactic sugar over these five definitions. A natural syntactic mapping is an important factor in the determining the popularity of a FFI and so the next section will examine dot-scheme's efforts in this area.

2.3 The Code Generation Layer

The Core API is all that is necessary to manipulate the Parrot class introduced earlier. The original example could be rewritten in terms of the Core API primitives:

```
(require (lib "dot-scheme-core.ss"
              "dot-scheme"))

(dispatch-instance
   (dispatch-constructor
      "ParrotAssembly, Parrot"
      "John"))
    "SayHello"
    "John")

> Hello John.
```

However this syntax is irritatingly distant from normal Scheme usage. This can easily be remedied with the help of some utilities:

```
(define-syntax import-method
    (syntax-rules ()
      ((_ ?scheme-name ?clr-name)
       (define (?scheme-name self . args)
         (apply dispatch-instance
                (cons self
                      (cons ?clr-name
                            args)))))))

(define-syntax import-type
  (syntax-rules ()
    ((_ ?scheme-name ?clr-name)
     (define ?scheme-name
       (dispatch-static "System.Type"
                        "GetType"
                        ?clr-name)))))

(define (new type-object . args)
  (apply
    (dispatch-constructor
      (dispatch-instance
        type-object
        "get_AssemblyQualifiedName")
       args)))
```

Now it is possible to write:

```
(import-type ::parrot "Parrot, ParrotAssembly")
(import-method :say-hello "SayHello")

(:say-hello (new ::parrot) "John")

> Hello John.
```

The new syntax looks like Scheme but requires typing import-statements. When importing a large number of types this may become tedious. Fortunately the CLR allows the contents of an assembly to be inspected at run-time. Using the primitives in the core layer it is possible to take advantage of the Reflection API to obtain a complete list of types in an assembly. It is then a simple matter to iterate through each type, generating the appropriate import-type/method expressions.

In fact this is almost exactly how the `import-assembly` syntax-case [3] macro works. There is a complication though. Because the CLR and Scheme use different rules for identifier naming and scoping it is not possible to map CLR names directly to Scheme.

Dot-scheme addresses these issues by renaming CLR methods and types in a way that is compatible with Scheme naming rules and hopefully produces bindings that can be easily predicted from the original CLR names. The problems addressed by this process include:

- Case sensitivity. The CLR is case sensitive while standard Scheme is not. Dot-scheme addresses this issue by mangling CLR identifiers, introducing a '-' before each upper-case character but the first (if the first character is lower-case a '-' is inserted at the beginning). Because '-' is an illegal character for CLR identifiers this mapping is isomorphic.
- Identifier scoping. In the CLR, method and type names have different scopes. It is perfectly legal to have a method `A` defined in a class `B` and a method `B` defined in a class `A`. In Scheme there is only one namespace so if both methods and types were mapped in the same way collisions could occur. Dot-scheme address this issue by prefixing type names with '::' and method names with ':'.
- Namespaces. Languages that target the CLR provide some mechanisms to segregate type definitions into namespaces. A namespace is an optional prefix for a type name. The problem faced by dot-scheme is what to do when identifiers coming from different namespaces collide. Dot-scheme's solution for this issue is very simple. For each CLR type dot-scheme will create two bindings. One of the bindings will include the namespace prefix associated with the type and the other will not. In most of the cases the Scheme programmer will use the shorter version. In case of a collision the long name can be used.
- Differences in method name scoping. In the CLR each method name is scoped to the class in which it is declared. Since dot-scheme dispatches instance method calls by searching for a method declared in the type of the first argument, instance method name collisions pose

no difficulties. However static methods present a different challenge. In this case there is no "this" pointer to reduce the scope of the method name search. Dot-scheme addresses this issue by prefixing each static method name with the type name the method is associated with.

The code generation layer in dot-scheme consists essentially of a set of macros that generate bindings according to the rules described above.

3. Future work

As presented dot-scheme allows the creation and use of CLR objects. This is, of course, only half the problem. Callbacks from the CLR to Scheme would also be very useful, in particular for implementing Graphical User Interfaces. The CLR allows the generation of code at run-time so it should be possible for dot-scheme to generate new CLR classes based on Scheme specifications. Future work will investigate this possibility. Performance issues are also likely to be visited. Currently dot-scheme's implementation favors simplicity over performance when a choice is necessary. As the system matures we expect this bias to change.

4. Conclusion

This paper supports two different goals. Ostensibly the goal has been to present the design and implementation of a Scheme FFI. A more covert but perhaps more important goal was to alert Scheme implementers to the ease with which bindings to the CLR can be added to existing Scheme implementations.

The CLR presents an important opportunity for Scheme. It provides a vast API covering most of the OS services and is an active development platform currently supporting more than two dozen different languages. In this role as a universal binary standard the CLR is even more enticing. By providing access to the CLR, Scheme implementers gain access to libraries written in a number of languages including C++, Lisp, Smalltalk and, ironically, other Schemes.

These facts have not gone unnoticed in other language communities including Haskell, Ruby and Perl which already provide some sort of FFI integration with the CLR [4, 9, 1]. Scheme has some potential advantages in this area however. Its unique syntax definition capabilities can arguably be used to achieve a simpler and more natural result than what is possible in other languages.

5. REFERENCES

[1] ActiveState. PerlNet http://aspn.activestate.com/ASPN/docs/PDK/PerlNET/PerlNET.html

[2] Box, D. and Sells, C. Essential .NET, Volume I: The Common Language Runtime. Addison Wesley, 2002, Ch. 6.

[3] Dybvig, R. Writing Hygienic Macros in Scheme With Syntax-Case, Indiana University Computer Science Department Technical Report #356, 1992

[4] Finne, S. Hugs98 for .NET, http://galois.com/~sof/hugs98.net/

[5] Flatt, M. PLT MzScheme: Language Manual, 2003, Ch. 5

[6] Flatt, M. Inside PLT MzScheme, 2003.

[7] PLT, http:///www.drscheme.org

[8] Richter, J. Applied Microsoft .NET Programming. Microsoft Press, 2002. Ch. 19.

[9] Schroeder B, Pierce J. Ruby/.NET Bridge http://www.saltypickle.com/rubydotnet/

[10] Thai, T. and Lam H. .NET Framework Essentials. O'Reilly, 2001