Nemerle

Introduction to a Functional .NET Language

Michał Moskal
University of Wrocław
Computer Science Institute
Przesmyckiego 20
Poland, 50-151 Wrocław
moskal@nemerle.org

Paweł W. Olszta
University of Wrocław
Computer Science Institute
Przesmyckiego 20
Poland, 50-151 Wrocław
olszta@nemerle.org

Kamil Skalski
University of Wrocław
Computer Science Institute
Przesmyckiego 20
Poland, 50-151 Wrocław
skalski@nemerle.org

ABSTRACT

Nemerle is a new functional language designed from the ground up for the `.NET`. In this paper we have focused on features absent in traditional ML-like and object-oriented languages: variant inheritance, assertions and powerful code-generating macros. We also gave concern for the syntax and the "spirit" of Nemerle that makes it a good transition language for programmers with C# background.

Keywords
Functional programming, programming languages, metaprogramming.

1. MOTIVATION

Our objective was to create a statically typed functional language with well founded `.NET` [ISO03b] interoperability. The `.NET` environment, especially since the introduction of generics [Kenn01], provides an excellent platform for high-level language implementation which:

- comes with a rich class library in the core system
- gives access to vast cache of additional third party libraries
- provides automatic garbage collection and security features
- handles native code generation and low-level optimizations (JIT)
- guarantees portability of executables
- allows integration with existing development tools
- etc. etc.

Of course, the framework is strongly object-oriented and primarily focused on traditional object-oriented and imperative languages. Therefore ports of the existing functional languages to the `.NET` did not fit in as well as, for example, C# [ISO03a] does. Addressing this issue was the main idea behind the design of Nemerle.

In comparison to Haskell [Jon99] or SML [Mil91], Nemerle is not a pure language in the functional sense, allowing the programmer to create completely object-oriented and imperative programs. This makes Nemerle a good transition language for people with C-like imperative and object-oriented languages background. They can take advantage of the language imperative features until they gradually learn how to program in a functional fashion.

An easy access to imperative constructs is only one of the requirements needed in such a transition language. Probably the hardest thing about learning ML is understanding the compiler error messages about typing mismatches. It may seem odd at first glance, but this is the reality – the type inference is very nice when it works, but when it

fails, you are stuck with error messages hundred lines from the place of the the real error.

We have decided to avoid language constructions that produce typing errors in ML, while generating syntax errors in other languages (for example function application being just ϵ); requiring the typing to be explicit, at least for global functions – implicit typing is not really possible to achieve when aiming for a good support for methods overloading.

It seems easy to observe that it is the quality of the design of the object-oriented system that determines usability of a programming language. While the existing object-oriented extensions to functional languages are appealing because of their elegance, they do not fit the `.NET` framework at all. We have decided to make our object-oriented system simply mirror the `.NET` design.

2. OVERVIEW

At the high level Nemerle can be characterized as a combination of C# at the class level and a ML-like language at the expression level. However, the syntax of the ML fragment is much less ambiguous and more C-like than Algol-like. The result is an expression-oriented language with a feeling of C#.

Of course we also need variants[1], pattern matching and functional values. These can be thought of as extensions to the base C#-like language.

There are some other facts about Nemerle that are implied by the "not-so-ML-like `.NET` language" paradigm:

- The language is statically typed, but dynamic casting is available and can be used when needed.
- The language combines functional, object-oriented, and imperative features.
- The object system is a one-to-one mapping of CLR's – making it fairly easy to understand.
- The language interoperates fully with other `.NET` languages – it is both a CLS consumer and producer.

In the following sections we will show how the language looks like and how is it different from ML and C#. The reader is assumed to have some basic knowledge about both ML and C#.

[1]Called datatypes in SML, and sometimes sum types in Caml.

It is important to mention that the language is still evolving and that its design is quite flexible. Especially assertions and macros are relatively new features. We are open to any suggestions.

3. THE LANGUAGE

The top-level program structure reassembles C#. There are namespaces, then classes and finally methods. We also have modules (classes with all members static and public) and variants. Let us look at the famous example:

```
class Hello {
  public static Main () : void {
    System.Console.WriteLine ("Hello, "
      + "I have {0} years!", 22);
  }
}
```

Another way to write it could be:

```
using System.Console;

module Hello {
  public Main () : void {
    WriteLine ("Hello cruel world.");
  }
}
```

The basic building block of a method is a sequence. A sequence groups local definitions (specified with the **def** keyword), expressions computed for their side effects and the final expression returned as the value of entire sequence[2].

```
public static factorial (x : int) : int
{
  def loop (acc, x) {
    if (x <= 1)
      acc
    else
      loop (acc * x, x - 1)
  };
  loop (1, x)
}
```

In this example the local function is implicitly typed – its type is inferred automatically by the compiler. Global functions are explicitly typed by design of the language.

[2]We put here value bindings and side-effect expressions into one can. This is exactly how it works in imperative languages and (under the hood) in eager functional languages. It models real world behavior better, and should be easier to understand.

Mutable values

Mutable local values are defined using declarations like **mutable** x = *expression*;. The value x can be later used as a value bound with **def** without any explicit dereference operator[3], but can be assigned using the assignment operator (=).

```
public static factorial (x : int) : int
{
  mutable acc = 1;
  mutable k = n;
  while (k > 0) {
    acc = acc * k;
    k = k - 1;
  };
  acc
}
```

The **while** loop should be considered as just a different form of tail recursion. It is in fact implemented as a macro which generates the following code:

```
public static factorial (x : int) : int
{
  mutable acc = 1;
  mutable k = n;
  def loop () {
    when (k > 0) {
      acc = acc * k;
      k = k - 1;
      loop ()
    }
  };
  loop ();
  acc
}
```

Our optimizer is clever enough to recognize that it needs no new `loop` method here – it will just insert the `br` opcode at the IL level.

The **mutable** keyword can be also used as a modifier on fields. The contents of such fields can be modified using the same assignment operator.

Variants and pattern matching

Variants are compiled to subclassing and should be thought of as subtypes. For example:

```
variant BinaryTree <'a> {
  | Leaf { val : 'a; }
  | Node { left : BinaryTree <'a>;
           val : 'a;
           right : BinaryTree <'a>; }
}
```

[3]Like the `!` operator in ML.

Would be compiled to:

```
class BinaryTree<A> {}
class Node<A> : BinaryTree<A> {
  BinaryTree<A> left;
  A val;
  BinaryTree<A> right;
}
class Leaf<A> : BinaryTree<A> {}
```

However, in the absence of of generics support in the current Framework release type qualifiers are stored as attributes alongside the type declarations.

Of course we can use regular ML-like matching over variants:

```
count<'a> (t : BinaryTree <'a>) : int {
  match (t) {
    | Node (l, _, r) =>
      count (l) + 1 + count (r)
    | Leaf => 1
  }
}
```

Which can be shortened to:

```
count<'a> (t : BinaryTree <'a>) : int {
  | Node (l, _, r) =>
    count (l) + 1 + count (r)
  | Leaf => 1
}
```

The `'a` after `count` quantifies following occurrences of `'a`.

There is one tricky thing about the second line of our example. It could have been written in any of the following ways:

```
  | (Node) as n =>
    count (n.left) + 1 + count (n.right)
  | Node (l, _, r) =>
    count (l) + 1 + count (r)
  | Node { left = l; right = r } =>
    count (l) + 1 + count (r)
```

```
  | Node { left = l; val = _;
           right = r } =>
    count (l) + 1 + count (r)
```

In fact, when the compiler sees a tuple pattern and expects a record pattern, the tuple is transformed into a record. It is therefore not so painful to require variant members to be named.

It is also possible to have deep patterns like Foo (Bar (Baz)), to match constants and to match real tuples. We also support pattern guards – that is condition checked after pattern has matched.

Variant inheritance

The subtyping model allows the variants to carry slightly more information then their ML counterparts. In particular it is possible to make the variant base class have some fields, methods or even derive from some other class. This way all variant options can have some common part. An example (taken from Nemerle compiler, which is written in Nemerle itself):

```
class Located {
  file : string;
  line : int;
}

variant Expr extends Located {
  | E_call { fn : Expr;
             parms : list <Expr>; }
  | E_ref { name : string; }
}

public static dump (e : Expr) : void {
  print ("// " + e.file + ": " +
         e.line.ToString ());
  match (e) {
    | E_ref (name) => print (name)
    | E_call (fn, parms) =>
      dump (fn);
      List.iter (dump, parms)
  }
}
```

Constrained parametric types

Types can be parametrized over other types. Type arguments can be constrained. This works the same way as generics do in IL. It is also possible to parametrize methods.

```
variant tree <'a>
where 'a : IComparable <'a>
{
  | Node {
      left : tree <'a>;
      data : 'a;
      right : tree <'a>;
    }
  | Tip
}
```

This is the Nemerle way to do things that would have been done with functors in ML-like languages. It is not strictly as powerful, but seems to be good enough in practice and integrates well with the `.NET` framework.

4. ASSERTIONS

Currently we have C-like **assert** implemented as a macro. We plan implement have **require** and **ensure** to support design by contract, as well as several special assertions for mutable value enforcing invariants.

- mutable values **guarded** with assertions – update of this very value triggers associated assertion
- **guard** assertions that are checked after update of any value directly referenced from the assertion body; checks are performed until the end of the current block

It is possible to attach the **guard** assertions to local values, instance fields and static fields (global values).

We sometimes want assertions like `x + y == 5` to hold, with mutable `x` and `y`. To allow update of `x` immediately followed by update of `y` a **transaction** block is introduced. Assertions to be triggered during the **transaction** block are stacked, and executed when the control leaves it.

It is to be reconsidered when exactly assertions are checked. Enforcing a check after each update can be hard in presence of parameters passed by **ref**.

5. MACROS

Macros in Nemerle have much more to do with Meta Haskell [Shea02], CamlP4 [CamlP4] or Scheme Lisp code-generating macros, than with macros in the languages like C. Macros are essentially compiler plugins – pieces of the Nemerle code that take type or expression abstract syntax trees (or AST for short) and return some other expressions or types (also as AST).

Macros are by definition Turing-complete[4]. Macros can access external files, extract typing information from a running database and generally do whatever you can imagine.

Macros are executed at the compilation time. The code they generate is later statically type checked. Macros are thus safe. There is always a risk that a macro will crash (or loop) during the compilation, but there is no way to avoid that while retaining its expressiveness.

As said before, the macros are written in Nemerle itself. In principle it would be possible to use any other `.NET` language, but Nemerle provides a special code quotation syntax to construct and walk its own AST. It provides a clear separation of the meta-language from the object language it is describing.

The macros can be also executed at the run time, taking advantage of dynamic aspects of the `.NET` framework. This can be used for example to develop programming language interpreters, or to specialize the code for efficiency.

Our meta-system is closely interleaved with the compilation process. It can perform partial typing of program's AST. Compiler internal typing procedures are executed by macro code in arbitrary order and their result can be analyzed, giving much more information about the program.

Usage

Example uses of macros:

- extending the syntax of the language
- embedding special purpose sublanguages in Nemerle:
  - `printf` and `scanf` like functions
  - binding optional and named groups in regular expression to local variables
  - `$`-interpolation like in Bourne shell or Perl
  - binding results of SQL queries to local variables in a type safe way
  - special syntax for XPath or some other XML-matching constructions
- generation of AST from external files
  - Yacc and Burg-like tools
  - generating types from an XML schema or DTD
- generation of external files based on AST
  - pretty printing of the generated or original code
- generation of AST based on other AST
  - generating XML serialization methods
  - specialization of the code at the source language level
  - support for Aspects-Oriented Programming by adding cross-cutting "concerns" to the program in algorithmic and arbitrarily flexible way

[4]It is not by accident like in some other languages.

Example: regular expression macro

This macro matches given string against pattern in sequence binding matched groups to variables. Not the use of `printf` macro in this example.

```
regexp match (s) {
  | "a*.+" => printf ("a\n");
  | @"(?<num : int>\d+)-\w+" =>
    printf ("%d\n", num + 3);
  | "(?<name>(Ala|Kasia))? ma kota" =>
    match (name) {
      | None => printf ("noname?\n")
      | Some (n) => printf ("%s\n", n)
    }
  | _ => printf ("default\n");
}
```

Example: SQL queries macro

This macro requires an SQL parser, and access to the database we are working on, so that the types of table columns and stored functions can be determined. It is necessary to determine the types of SQL expressions, which can be later used to produce source language bindings for values returned by SQL queries.

```
ExecuteReaderLoop (conn,
  "SELECT salary, LOWER (name) AS lname"
  "  FROM employees"
  "  WHERE salary > $(min_salary * 3)")
    print ("$lname : $salary\n")
```

And the result:

```
def cmd = SqlCommand (
  "SELECT salary, LOWER (name)"
  "  FROM employees"
```

```
  "  WHERE salary > @parm1", conn);
cmd.Parameters.Add ("@parm1",
                    min_salary * 3);
def r = cmd.ExecuteReader ();
while (r.Read ()) {
  def salary = r.GetInt32 (0);
  def lname = r.GetString (1);
  printf ("%s : %d\n", lname, salary)
}
```

Example: A sample macro implementation

This is a sample implementation of a macro that adds the C#-like `foreach` loop to the language (together with the special syntax for this construct). This code comes directly from the compiler implementation.

```
macro @foreach (iter : funparm,
                collection, body)
syntax ("foreach", "(", iter, "in",
        collection, ")", body)
{
  match (iter) {
    | <[ funparm: $(iname : var)
                  : $ty ]> =>
      <[ def enumerator =
           $collection.GetEnumerator ();
         while (enumerator.NextMove ())
         {
           mutable $(iname : var) =
             (enumerator.Current :> $ty);
           $body;
         }
      ]>
    | _ =>
      Message.fatal_error (
        "iterator in ‘foreach’ must be "
        + "id with optional type")
  }
}
```

The code generated by the presented macro is constructed by the quotation construct in lines 6–13. Note that it creates code, which uses another syntax-extending macro, the `while` loop.

6. CODE GENERATION

The typed abstract syntax tree of expressions is converted into an intermediate functional description of a stack machine which is later used to build the compiler output using the API of `System.Reflection.Emit`

Optimizations are performed on both the typed AST level as well as on the intermediate representation level. For example tail calls are marked as such during AST generation, while matching automata generation is performed after intermediate code is generated.

Tail call elimination

We have implemented tail calls using the `tail.` prefix available in IL. However, it did not bring any improvements to the execution speed, it even slowed things down by a factor of 15%.

For tail calls to the current function, we have implemented simple transformation to argument assignment and `goto`. It brought a little speed improvement (over the version without tail calls), but reduced memory usage by about 20% (compared to the same version). Later we have implemented real loops (that is we do not always generate new method for local functions now), it made things faster by about 12%.

Matching optimizations

We are working on good matching code generation using a hashing function and binary search automates. This is, however, in a very early stage yet.

7. SUMMARY

We have shown the key points of a new functional language for the `.NET` framework. The language combines well-known concepts in a unique fashion. We believe that it could be used to teach the basics of functional programming and the `.NET`. We also hope it can be used outside academia as a real life, industry language.

8. REFERENCES

[Kenn01] Kennedy A., Syme D. Design and implementation of generics for the .NET Common language runtime in ACM SIGPLAN 2001 conf. proc., Snowbird, Utah, ACM Press, pp. 1–12, 2001.

[Mil91] Milner R., Tofte M., Harper R. The Definition of Standard ML. The MIT Press, 1991.

[Jon99] Jones S. P., Hughes J. Report on the Programming Language Haskell 98: A non-strict, purely functional language. Technical Report YaleU/DCS/RR-1106, Dept. of Computer Science, Yale University, 1999.

[Shea02] Sheard T., Jones S. P.. Template meta-programming for Haskell. In Proceedings of the Haskell workshop, pp. 1–16. ACM Press, 2002.

[ISO03a] International Organization for Standardization. C# Language Specification, ISO/IEC 23270:2003, 2003.

[ISO03b] International Organization for Standardization. Common Language Infrastructure, ISO/IEC 23271:2003, 2003.

[CamlP4] `http://caml.inria.fr/camlp4/`