Encoding monadic computations in C# using iterators

Tomas Petricek

Charles University in Prague, Faculty of Mathematics and Physics
tomas@tomasp.net

Abstract. *Many programming problems can be easily solved if we express them as computations with some non-standard aspect. This is a very important problem, because today we're struggling for example to efficiently program multi-core processors and to write asynchronous code. Unfortunately main-stream languages such as Java or C# don't support any direct way for encoding unrestricted non-standard computations. In languages like Haskell and F#, this can be done using monads with syntactic extensions they provide and it has been successfully applied to a wide range of real-world problems. In this paper, we present a general way for encoding monadic computations in the C# 2.0 language with a convenient syntax using an existing language feature called iterators. This gives us a way to use well-known non-standard computations enabling easy asynchronous programming or for example the use of software transactional memory in plain C#. Moreover, it also opens monads in general to a wider audience which can help in the search for other useful and previously unknown kinds of computations.*

1 Introduction

In functional programming languages such as Haskell and F#, monadic computations are used to solve wide range of problems. In Haskell [5], they are frequently used to deal with state or I/O, which is otherwise difficult in a purely functional language. F# uses monadic computations to add non-standard aspects such as asynchronous evaluation, laziness or implicit error handling to an existing piece of code written in F#. In this article, we'll prefer the F# point of view, meaning that we want to be able to adjust C# code to execute differently, using additional aspects provided by the monadic computation.

The primary motivation for this work is that monadic computations are very powerful technique for dealing with many modern computing challenges caused by the rise of multi-core processors and distributed web based applications. The standard F# library uses monadic computations to implement asynchronous workflows [16] which make it easy to write communication with the web and other I/O operations in the natural sequential style, but without blocking threads while waiting. In Haskell, monadic computations are used for example to implement software transactional memory, which is a concurrent programming mechanism based on shared memory, which avoids the need for explicit locking [3].

The motivation for this article is that we want to be able to use the techniques just described in a main-stream and widely used C# language. The main contributions of this paper are following:

- As far as we're aware, we show for the first time that monadic computations can be encoded in C# in a syntactically convenient way without placing any restrictions on the C# statements that can be used inside the computation. This can be done purely as a library without changing the language using widely adopted C# 2.0 features (Section 3).
- We use the presented technique to implement a library that makes it easier to write scalable multi-threaded applications that perform long running I/O operations. We demonstrate it using several case study examples (Section 4).
- Finally, we describe a method for systematical encoding of arbitrary monadic computation in C# (Section 5). This technique can be used for implementing other useful computations such as software transactional memory and others.

There are several related projects, mostly concerned with asynchronous programming (Section 6), but our aim is wider and focuses on monadic computations in general. However, asynchronous computations can nicely demonstrate the problem.

1.1 Asynchronous computations in C# today

Since we're using asynchronous computations as the primary real-world motivation for this paper, we should first clarify what problem we want to solve is. Let's start by looking at naive synchronous code that downloads the first kilobyte of web site content:

```
1: var req = HttpWebRequest.Create(url);
2: var rsp = req.GetResponse();
3: var strm = rsp.GetResponseStream();
4: var read = strm.Read(buffer, 0, 1024);
```

On lines 2 and 4 we're performing I/O operations that can take a long time, but that aren't CPU bounded. When running the operation, the executing thread will be blocked, but it cannot perform any other

work in the meantime. If we wanted to run hundreds of downloads in parallel, we could create hundreds of threads, but that introduces significant overheads (such as allocation of kernel objects and thread stack) and also increases context switching. The right way to solve the problem on the .NET platform is to use the *Asynchronous Programming Model* (APM):

```
1: var req = HttpWebRequest.Create(url);
2: req.BeginGetResponse(a1 => {
3:   var rsp = req.EndGetResponse(a1);
4:   var strm = rsp.GetResponseStream();
5:   strm.BeginRead(buffer, 0, 1024, a2 => {
6:     int read = strm.EndRead(a2);
7:     // ...
8:   }, null);
9: }, null);
```

In this context "asynchronous" means that the program invokes start of the operation, registers a callback, transfers the control to the system and releases the current thread, so that it can perform other work. In the snippet above, we're starting two operations on lines 2 and 5 and we're using the C# 3.0 lambda function notation "=>" to specify the callback function that will be eventually invoked.

The code above is far less readable than the first synchronous version, but that's not the only problem. To download the whole page, we'd need to call the `BeginRead` method in a loop until we fetched the whole page, but that's ridiculously difficult, because we can't use any higher level constructs such as the `while` loop when writing code using nested callbacks. For every simple problem, the programmer has to explicitly write a state machine using mutable state.

It is worth pointing out that using asynchronous model does not in principle increase the CPU parallelism in the application, but it still significantly improves the performance and makes the application more scalable because it considerably reduces the number of (expensive) threads the application creates.

2 Background

To write non-blocking asynchronous code, we can use *continuation passing style* where the next piece of code to execute after an operation completes is given as a function as the last argument to the operation. In the snippet above we've written the code in this style explicitly, but as we've seen this isn't a satisfying solution.

2.1 F# asynchronous workflows

In F#, we can use *asynchronous workflows*, which is one particularly useful implementation of monadic computations that is already defined in F# libraries. This feature hasn't been described in the literature before, so we quickly review it here.

When we wrap code inside an `async` block, the compiler automatically uses continuation passing style for specially marked operations. Moreover, we can use all standard language constructs inside the block including for example the `while` loop:

```
1: let downloadUrl(url:string) = async {
2:  let req = HttpWebRequest.Create(url)
3:  let! rsp = req.AsyncGetResponse()
4:  let strm = rsp.GetResponseStream()
5:  let buffer = Array.zeroCreate(8192)
6:  let state = ref 1
7:  while !state > 0 do
8:   let! read = strm.AsyncRead(buffer,0, 8192)
9:   Console.WriteLine("got {0}b", read);
10:  state := read }
```

This function downloads the entire content of a web page in a loop. Although it doesn't use the data in any way and only reports the progress, it nicely demonstrates the principle. Its body is an `async` block, which specifies that the function doesn't actually run the code, but instead returns a value representing computation that can be executed later.

In the places where the original C# code executed asynchronous operations, we're now using the `let!` Keyword (lines 3 and 8), which represents monadic value binding. This means that instead of simply assigning value to a symbol, the computation invokes `Bind` operation that is provided by the async value (called *computation builder*) giving it the rest of the code wrapped inside a function as an argument. The `Bind` member specifies non-standard behavior of the operation. In this case the behavior is that the operation is executed asynchronously.

The computation builder (in this case `async`) also defines the meaning of other primitive constructs such as the while loop or returning the result from a function. These primitive operations are exposed as standard methods with well-defined type signatures:

```
Bind   : Async<'a> * ('a -> Async<'b>) ->
         Async<'b>
Return : 'a -> Async<'a>
While  : (unti -> bool) * Async<unit> ->
         Async<unit>
```

The first two functions are standard operations that define the abstract monadic type as first described in [18]. These operations are also called *bind* and *unit*. The `Bind` member takes an existing computation and a function that specifies how to produce subsequent computation when the first one completes and composes these into a single one. The `Return`

member builds a computation that returns the given value. The additional `While` member takes a predicate and a computation and returns result that executes the computation repeatedly while the predicate holds.

When compiling code that uses computation expressions, the F# compiler syntactically transforms the code into code composed from the calls to these primitive operations. The translated version of the previous example can be found in the online supplementary material for the article [12].

2.2 C# Iterators

One of the non-standard computations that is very often used in practice is a computation that generates a sequence of values instead of yielding just a single result. This aspect is directly implemented by C# iterators [2], but without any aim to be more generally useful. In this article, we show that it can be used in a more general fashion. However, we start by briefly introducing iterators. The following example uses iterators to generate a sequence of all integers:

```
1: IEnumerator<int> GetNumbers() {
2:   int num = 0;
3:   while (true) {
4:     Console.WriteLine("generating {0}", num);
5:     yield return num++;
6:   }
7: }
```

The code looks just like ordinary method with the exception that it uses the `yield return` keyword to generate elements a sequence. The `while` loop may look like an infinite loop, but due to the way iterators work, the code is actually perfectly valid and useful. The compiler translates the code into a state machine that generates the elements of the sequence lazily one by one. The returned object of type `IEnumerator<int>` can be used in the following way:

```
1: var en = GetNumbers();
2: en.MoveNext();
3: Console.WriteLine("got {0}", en.Current);
4: en.MoveNext();
5: Console.WriteLine("got {0}", en.Current);
```

The call to the `GetNumbers` method (line 1) returns an object that represents the state machine generated by the compiler. The variables used inside the method are transformed into a local state of that object. Each call to the `MoveNext` method (lines 2 and 4) runs one step of the state machine until it reaches the next yield return statement (line 5 in the earlier snippet) updating the state of the state machine. This also executes all side-effects of the iterator such as printing to the console, so the program above shows the "generating" message directly followed by "got" for numbers 0 and 1. There are two key aspects of iterators that are important for this paper:

- The iterator body can contain usual control structures such as loops or exception handlers and the compiler automatically turns them into a state machine.
- The state machine can be executed only to a certain point (explicitly specified by the user using `yield return`), then paused and later resumed again by invoking the `MoveNext` method again.

In many ways this resembles the continuation passing style from functional languages, which is essential for monadic computations and F# asynchronous workflows.

3 Monadic computations in C#

Now that we've introduced asynchronous workflows in F# (as an example of monadic computations) and C# iterators, we can ask ourselves the question whether iterators could be used for encoding other non-standard computations then code that generates a sequence.

The key idea of this article is that it is indeed possible to do that and that we can write standard C# library to support any monadic computation. In this section, we'll briefly introduce how the library looks using the simplest possible example and in section 5 we'll in detail explain how the encoding works.

3.1 Using option computations

As the first example, we'll use computations that produce value of type `Option<'a>` [1], which can either contain no value or a value of type `'a`. The type can be declared using F#/ML notation like this:

```
type Option<'a> = Some of 'a | None
```

Code that is composed from simple computations that return this type can return `None` value at any point, which bypasses the entire rest of the computation. In practice this is useful for example when performing series of data lookup that may not contain the value we're looking for. The usual way for writing the code would check whether the returned value is `None`

[1] In Haskell, this type is called `Maybe` and the corresponding computation is known as Maybe monad.

after performing every single step of the computation, which significantly complicates the code [2].

To show how the code looks when we apply our encoding of the option computation using iterators, we'll use method of the following signature:

```
ParseInt : string -> Option<int>
```

The method returns `Some(n)` when the parameter is a valid number and otherwise it returns `None`. Now we can write code that reads a string, tries to parse it and returns 10 times the number if it succeeds. The result of the computation will again be the option type.

```
1: IEnumerator<IOption> ReadInt() {
2:   Console.Write("Enter a number: ");
3:   var optNum = ParseInt(Console.ReadLine());
4:   var m = optNum.AsStep();
5:   yield return m;
6:   Console.WriteLine("Got a valid number!");
7:   var res = m.Value * 10;
8:   yield return OptionResult.Create(res);
9: }
```

The code reads a string from the console and calls the `ParseInt` method to get `optNum` value, which has a type `Option<int>` (line 3). Next, we need to perform non-standard value binding to access the value and to continue running the rest of the computation only when the value is present. Otherwise the method can return `None` as the overall result straight ahead.

To perform the value binding, we use the `AsStep` method that generates a helper object (line 4) and then return this object using yield return (line 5). This creates a "hole" in the code, because the rest of the code may or may not be executed, depending on whether the `MoveNext` method of the returned state machine is called again or not. When `optNum` contains a value, the rest of the code will be called and we can access the value using the `Value` property (line 7)[3].

Finally, the method calculates the result (line 7) and returns it. To return from a non-standard computation written using our encoding, we create another helper object, this time using `OptionResult.Create` method. These helper objects are processed when executing the method.

To summarize, there are two helper objects. Both of them implement the `IOption` interface, which means that they can both be generated using yield return. The methods that create these two objects have the following signatures:

```
AsStep : Option<'a> -> OptionStep<'a>
OptionResult.Create : 'a -> OptionResult<'a>
```

The first method creates an object that corresponds to the monadic bind operation. It takes the option value and composes it with the computation that follows the `yield return` call. The second method builds a helper that represents monadic unit operation. That means that the computation should end returning the specified value as the result.

3.2 Executing option calculation

When we write code in the style described in the previous section, methods like `ReadInt` only return a state machine that generates a sequence of helper objects representing bind and unit. This alone wouldn't be at all useful, because we want to execute the computation and get a value of the monadic type (in this case `Option<'a>`) as the result. How to do this in terms of standard monadic operations is described in section 5, but from the end user perspective, this simply means invoking the `Apply` method:

```
Option<int> v = ReadInt().Apply<int>();
Console.WriteLine(v);
```

This is a simple piece of standard C# code that runs the state machine returned by the `ReadInt` method. `Apply<'a>` is an extension method [4] defined for the `IEnumerator<IOption>` type. Its type signature is:

```
Apply : IEnumerable<IOption> -> Option<'a>
```

The type parameter (in the case above `int`) specifies what the expected return type of the computation is, because this unfortunately cannot be safely tracked in the type system. Running the code with different inputs gives the following console output:

```
Enter a number: 42      Enter a number: $%?!
Got a valid number!     None
Some(420)
```

Strictly speaking, `Apply` doesn't necessarily have to execute the code, because its behavior depends on the monadic type. The `Option<'a>` type represents a value, so the computation that produces it isn't delayed. On the other hand the `Async<'a>` type, which represents asynchronous computations is delayed meaning that the Apply method will only build a computation from the C# compiler generated state machine.

[2] We could as well use exceptions, but it is generally accepted that using exceptions for control flow is a wrong practice. In this case, the missing value is an expected option, so we're not handling exceptional condition.

[3] The F# code corresponding to these two lines is: `let! value = optNum`

[4] Extension methods are new feature in C# 3.0. They are standard static methods that can be accessed using dot-notation as if they were instance methods [1].

The encoding of non-standard computations wouldn't be practically useful if it didn't allow us to compose code from primitive functions and as we'll see in the next section, this is indeed possible.

3.3 Composing option computations

When encoding monadic operations, we're working with two different types. The methods we write using the iterator syntax return `IEnumerator<IOption>`, but the actual monadic type is `Option<'a>`. When writing code that is divided into multiple methods, we need to invoke method returning `IEnumerator<IOption>` from another method written using iterators. The following example uses the `ReadInt` method from the previous page to read two integer values (already multiplied by 10) and add them.

```
1: IEnumerator<IOption> TryCalculate() {
2:   var n = ReadInt().Apply<int>().AsStep();
3:   yield return n;
4:   var m = ReadInt().Apply<int>().AsStep();
5:   yield return m;
6:   var res = m.Value + n.Value;
7:   yield return OptionResult.Create(res);
8: }
```

When the method needs to read an integer, it calls the `ReadInt` to build a C# state machine. To make the result useable, it converts it into a value of type `Option<int>` (using the `Apply` method) and finally uses the `AsStep` method to get a helper object that can be used to bind the value using `yield return`.

We could of course provide a method composed from `Apply` and `AsStep` to make the syntax more convenient, but this paper is focused on explaining the principles, so we write this composition explicitly.

The previous example also nicely demonstrates the non-standard behavior of the computation. When it calls the `RadInt` method for the second time (line 4) it does that after using non-standard value binding (using `yield return` on line 3). This means that the user will be asked for the second number only if the first input was a valid number. Otherwise the result of the overall computation will immediately be `None`.

Calculating with options nicely demonstrates the principles of writing non- standard computations. We can use non-standard bindings to mark places where the code can abandon the rest of the code if it already knows the overall result. Even though this is already useful, we can make even stronger point to support the idea by looking at asynchronous computations.

4 Case study: asynchronous C#

As discussed in the introduction, writing non-blocking code in C# is painful even when we use latest C# features such as lambda expression. In fact, we haven't even implemented a simple loop, because that would make the code too lengthy. We've seen that monadic computations provide an excellent solution [5] and we've seen that these can be encoded in C# using iterators.

As next, we'll explore one larger example that follows the same encoding of monadic computations as the one in the previous section, but uses a different *monad* to write asynchronous code that doesn't block the tread when performing long-running I/O. The following method reads the entire content of a stream in a buffered way (using 1kb buffer) performing each read asynchronously.

```
1: IEnumerator<IAsync> ReadToEndAsync(Stream s)
   {
2:   var ms = new MemoryStream();
3:   byte[] bf = new byte[1024];
4:   int read = -1;
5:   while (read != 0) {
6:     var op = s.ReadAsync(bf, 0, 1024).
       AsStep();
8:     yield return op;
9:     ms.Write(bf, 0, op.Value);
10:    read = op.Value;
11:  }
12:  ms.Seek(0, SeekOrigin.Begin);
13:  string s = new StreamReader(ms).
     ReadToEnd();
14:  yield return AsyncResult.Create(s);
15: }
```

The code uses standard `while` loop which would be previous impossible. Inside the body of the loop, the method creates an asynchronous operation that reads 1kb of data from the stream into the specified buffer (line 6) and runs the operation by yielding it as a value from the iterator (line 7). The operation is then executed by the system and when it completes the iterator is resumed. It stores the bytes to a temporary storage and continues looping until the input stream is fully processed. Finally, the method reads the data using `StreamReader` to get a string value and returns this value using `AsyncResult.Create` method (line 14).

The encoding of asynchronous computations is essentially the same as the encoding of computations with option values. The only difference is that the method now generates a sequence of `IAsync` values. Also, the `AsStep` method now returns an object of type `AsyncStep<'a>` and similarly, the helper object used

[5] To justify this, we can say that asynchronous workflows are one of the important features that contributed to the recent success of the F# language.

for returning the result is now `AsyncResult<'a>`. Thanks to the systematic encoding described in section 5, it is very easy to use another non-standard computation once the user understands one example.

The method implemented in the previous listing is very useful and surprisingly, it isn't available in the standard .NET libraries. We'll use it to asynchronously download the entire web page. However, we first need to asynchronously get the HTTP response, so we'll write the code as another asynchronous method using iterator syntax. In section 3.3, we've seen how to compose *option computations* and since the principle is the same, it isn't surprising that composing *asynchonous computations* is also straightforward:

```
1: var stream = resp.Value.GetResponseStream();
2: var html = ReadToEndAsync(stream).
3:            Execute<string>().AsStep();
4: yield return html;
5: Console.WriteLine(html.Value);
```

The listing assumes that we already have a response object (`resp`). So far we haven't seen how to actually start the download, because *asynchronous computations* are delayed, meaning that we're just constructing a function that can be executed later. This makes it possible to compose large number of computations and spawn them in parallel. The runtime then uses only a few threads, which makes it very efficient and scalable. You can find full example that shows how to start the download in the online supplementary material for the article [12]

Implementing the functionality we presented in this section asynchronously using the usual style would be far more complicated. For example, to implement the `ReadToEndAsync` method we need two times more lines of very dense C# code. However, the code also becomes hard to read because it cannot use many high-level language features (e.g. `while` loop), so it would in addition also require a decent amount of comments[6].

5 Encoding arbitrary monads

As we've seen in the previous two sections, writing monadic computations in C# using iterators requires several helper objects and methods. In this section, we'll look how these helpers can be defined.

Unfortunately, C# doesn't support higher-kinded polymorphism, which is used when defining monads in Haskell and is available in some object-oriented language such as Scala [10]. This means that we can't write code that is generic over the monadic type (e.g. `Option<'a>` and `Async<'a>`). As a result, we need to define a new set of helper objects for each type of computation. To make this task easier, we provide two base classes that encapsulate functionality that can be reused. The code that we need to write is the same for every computation, so writing it is a straightforward task that could be even performed by a very simple code-generator tool.

In this section, we'll look at the code that needs to be written to support calculations working with option values that we were using in section 3. The code uses only two computation-specific operations. Indeed, these are the two operations bind and unit that are used to define monadic computations in functional programming ("O" is a shortcut for "Option"):

```
Bind : O<'a> -> ('a -> O<'b>) -> O<'b>
Return : 'a -> O<'a>
```

The bind operation uses the function provided as the second parameter to calculate the result when the first parameter contains a value. The unit operation wraps an ordinary value into an option value. The implementation of these operations is described elsewhere, so we won't discuss it in detail. You can for example refer to [11]. We'll just assume that we already have `OptionM` type with the two operations exposed as static methods.

5.1 Defining iterator helpers

As a first thing, we'll implement helper objects that are returned from the iterator. We've seen that we need two helper objects - one that corresponds to bind and one that corresponds to unit. These two objects share common interface (called `IOption` in case of option computations) so that we can generate a single sequence containing both of them. Let's start by looking at the interface type:

```
interface IOption {
  Option<R> BindStep<R>(Func<Option<R>> k);
}
```

The `BindStep` method is invoked by the extension that executes the non-standard computation (we'll discuss it later in section 5.2). The parameter `k` specifies a continuation, that is, a function that can be executed to run the rest of the iterator. The continuation doesn't take any parameters and returns an option value generated by the rest of the computation. The implementation of the helper objects that implement the interface looks like this:

[6] The source code implementing the same functionality in the usual style can be found in [12]

```
class OptionStep<T> : MonadStep<T>, IOption {
 internal Option<T> Input { get; set; }
 public Option<R> BindStep<R>(Func<Option<R>> k)
 {
  return OptionM.Bind(Input,
  MakeContinuation(k));
 }
}
class OptionResult<T> : MonadReturn<T>, IOption
{
 internal OptionResult(T value) : base(value)
 { }
 public Option<R> BindStep<R> (Func<Option<R>>
 k) {
  return OptionM.Return(GetResult<R>());
 }
}
```

The `OptionStep<'a>` type has a property named `Input` that's used to store the option value from which the step was constructed using the `AsStep` method. When the `BindStep` method is executed the object uses the monadic bind operation and gives it the input as the first argument. The second argument is more interesting. It should be a function that takes the actual value extracted from the input option value as an argument and returns a new option value. The extracted value can be used to calculate the result, but there is no way to pass a value as an argument back to the iterator in the middle of its evaluation, which is why the function given as the parameter to `BindStep` method doesn't take any parameters.

As we've seen in the examples, the `OptionStep<'a>` helper exposes this value as the `Value` property. This property is inherited from the `MonadStep<'a>` type. The `MakeContinuation` which we use to build a parameter for monadic bind operation is also inherited and it simply stores the input obtained from bind into `Value`, so that it can be used in the iterator and then runs the parameter-less continuation `k`.

The `OptionResult<'a>` type is a bit simpler. It has a constructor that creates the object with some value as the result. Inside the `BindStep` method, it uses the monadic unit operation and gives it that value as the parameter. This cannot be done in a statically type-checked way, so we use the inherited `GetResult` method that performs dynamic type conversion. Finally, the `OptionResult.Create` and `AsStep` methods are just simple wrappers that construct these two objects in the syntactically most pleasant way.

5.2 Implementing iterator evaluation

Once we have an iterator written using the helpers described in the previous section, we need some way for executing it using the non-standard behavior of the monadic computation. The purpose of the iterator isn't to create a sequence of values, so we need to execute it in some special way. The following example shows an extension method `Apply` that turns the iterator into a monadic value. In case of options the type of the value is `Option<'a>` but note that the code will be exactly the same for all computation types.

```
static class OptionExtensions {
  public static Option<R> Apply<R>
      (this IEnumerator<IOption> en) {
    if (!en.MoveNext())
      throw new InvalidOperationException
        ("Enumerator ended without a result!");
    return en.Current.BindStep<R>(() =>
      en.Apply<R>());
  }
}
```

The method starts by invoking the `MoveNext` method of the generated iterator to move the iterator to the next occurrence of the `yield return` statement. If the return value is `false`, the iterator ended without returning any result, which is invalid, so the method throws an exception.

If the iterator performs the step, we can access the next generated helper object using the `en.Current` property. The code simply invokes `BindStep` of the helper and gives it a function that recursively calls the `Apply` method on the same iterator as the argument. Note that when the helper is `OptionResult<'a>`, the continuation isn't used, so the recursion terminates.

It is worth noting that for monadic computations with zero operation we can also write a variant of the `Apply` method that doesn't require the iterator to complete by returning a result. In that case, we'd modify the method to return a value constructed by the monadic *zero* operation instead of throwing an exception in the case when the iterator ends.

Finally, there are also some problems with using possibly deeply nested recursive calls in a language that doesn't guarantee the use of tail-recursion. We can overcome this problem by using some technique for tail-call elimination. Schinz [15] gives a good overview in the context of the Scala language. Perhaps the easiest option to implement is to use a trampoline [17].

5.3 Translation semantics

To formalize the translation in a more detail, we've also developed a translation semantics for the library. We first define an abstract language extension for C# that adds monadic compuations as a language feature to C# in a way similar to F#. Then we show that any code written in the extended C# can be translated to the standard C# 2.0 by using the iterator encoding

presented in this article. The grammar of the language extension as well as the semantic rules are available in the online supplementary material [12].

6 Related work and conclusions

There is actually one more way for writing some monadic computations in C# using the recently added query syntax. The syntax is very limited, but may be suitable for some computations. We'll briefly review this option and then discuss other relevant related work and conclusions of this paper.

6.1 LINQ queries

As many people already noted [8], the LINQ query syntax available in C# 3.0 is also based on the idea of monad and can be used more broadly than just for encoding computations that work with lists. The following example shows how we could write the computation with option values using LINQ syntax:

```
1: Option<int> opt =
2:   from n in ReadInt()
3:   from m in ReadInt()
4:   let res = m + n
5:   select res;
```

The implementation of library that allows this kind of syntax is relatively easy and is described for example in [11]. This syntax is very restricted. In it allows non-standard value bindings corresponding to the bind operation using the from keyword (lines 2 and 3), standard value bindings using the let construct (line 4) and returning of the result using select keyword (line 5). However, there are no high-level imperative constructs such as loops which were essential for the asynchronous example in section 4. With some care, it is possible to define several mutually recursive queries, but that still makes it hard to write complex computations such as the one in section 4.

On the other hand, query syntax is suitable for some monadic computations where we're using only a limited language. Parser combinators as described for example in [6] can be defined using the query syntax [4]. In general, C# queries are a bit closer to writing monads using the Haskell's list comprehension notation, while using iterators as described in this article is closer to the Haskell's do-notation.

6.2 Related work

The principle of using a main-stream language for encoding constructs from the research world has been used with many interesting features including for example Joins [14]. FC++ [8] is a library that brings many functional features to C++, including monads, which means it should be possible to use it for re-implementing some examples from this paper.

There are also several libraries that use C# iterators for encoding asynchronous computations. CCR [7] is a more sophisticated library that combines join patterns with concurrent and asynchronous programming, which makes it more powerful than our encoding. On the other hand it is somewhat harder to use for simple scenarios such as those presented in this paper.

Richter's library [13] is also focused primarily on asynchronous execution. It uses yield return primitive slightly differently - to specify the number of operations that should be completed before continuing the execution of the iterator. The user can then pop the results from a stack.

7 Conclusions

In this paper, we have presented a way for encoding monadic computations in the C# language using iterators. We've demonstrated the encoding with two examples - computations that work with option values and computations that allow writing of non-blocking asynchronous code.

The asynchronous library we presented is useful in practice and would alone be an interesting result. However, we described a general mechanism that can be useful for other computations as well. We believe that using it to implement for example a prototype of software transactional memory support for C# can bring many other interesting results.

References

1. G.M. Bierman, E. Meijer, M. Torgersen: *Lost in translation: formalizing proposed extensions to C#.* In Proceedings of OOPSLA 2007.
2. ECMA International.: C# Language Specification.
3. T. Harris, S. Marlow, S. Peyton-Jones, M. Herlihy: *Composable memory transactions.* In Proceedings of PPoPP 2005.
4. L. Hoban: *Monadic parser combinators using C# 3.0.* Retrieved May 2009, from `http://blogs.msdn.com/lukeh/archive/2007/08/19/monadic-parser-combinators-using-c-3-0.aspx`
5. P. Hudak, P. Wadler, A. Brian, B.J. Fairbairn, J. Fasel, K. Hammond et al.: *Report on the programming language Haskell: A non-strict, purely functional language.* ACM SIGPLAN Notices.
6. G. Hutton, E. Meijer: *Monadic parser combinators.* Technical Report. Department of Computer Science, University of Nottingham.
7. G. Chrysanthakopoulos, S. Singh: *An asynchronous messaging library for C#.* In proceedings of SCOOL Workshop, OOPSLA, 2005.

8. B. McNamara, Y. Smaragdakis: *Syntax sugar for FC++: lambda, infix, monads, and more.* In Proceedings of DPCOOL 2003.
9. E. Meijer: *There is no impedance mismatch (Language integrated query in Visual Basic 9).* In Dynamic Languages Symposium, Companion to OOPSLA 2006.
10. A. Moors, F. Piessens, M. Odersky: *Generics of a higher kind.* In Proceedings of OOPSLA 2008.
11. T. Petricek, J. Skeet: *Functional Programming for the Real World.* Manning, 2009.
12. T. Petricek: *Encoding monadic computations using iterators in C# 2.0 (Supplementary material).* Available at `http://tomasp.net/academic/monads-iterators.aspx`
13. J. Richter: *Power threading library.* Retrieved May 2009, from `http://www.wintellect.com/PowerThreading.aspx`
14. C.V. Russo: *The Joins concurrency library.* In Proceedings of PADL 2007.
15. M. Schinz, M. Odersky: *Tail call elimination on the Java Virtual Machine.* In Proceedings of BABEL 2001 Workshop on Multi-Language Infrastructure and Interoperability.
16. D. Syme, A. Granicz, A. Cisternino: *Expert F#*, Apress, 2007.
17. D. Tarditi, A. Acharya, P. Lee: *No assembly required: Compiling standard ML to C.* School of Computer Science, Carnegie Mellon University.
18. P. Wadler: *Comprehending monads.* In Proceedings of ACM Symposium on Lisp and Functional Programming, 1990.