Async Programming (with coMonads) for the Masses

Pause 'n' play: Formalizing asynchronous C♯⋆

G. Bierman[1], C. Russo[1], G. Mainland[1], E. Meijer[2], and M. Torgersen[2]

[1] Microsoft Research {gmb,crusso,gmainlan}@microsoft.com
[2] Microsoft Corporation {emeijer,madst}@microsoft.com

Abstract. Writing applications that connect to external services and yet remain responsive and resource conscious is a difficult task. With the rise of web programming this has become a common problem. The solution lies in using asynchronous operations that separate issuing a request from waiting for its completion. However, doing so in common object-oriented languages is difficult and error prone. Asynchronous operations rely on callbacks, forcing the programmer to cede control. This inversion of control-flow impedes the use of structured control constructs, the staple of sequential code. In this paper, we describe the language support for asynchronous programming in the upcoming version of C♯. The feature enables asynchronous programming using structured control constructs. Our main contribution is a precise mathematical description that is abstract (avoiding descriptions of compiler-generated state machines) and yet sufficiently concrete to allow important implementation properties to be identified and proved correct.

```
static long CopyTo(Stream src, Stream dst)
{
    var buffer = new byte[0x1000];
    var bytesRead = 0;
    var totalRead = 0L;
    while ((bytesRead
        = src.Read(buffer, 0, buffer.Length)) > 0)
    {
        dst.Write(buffer, 0, bytesRead);
        totalRead += bytesRead;
    }
    return totalRead;
}
```

Indeed may take long
(bad pun intended)

```
int Read(byte[] buffer, int offset, int count)

IAsyncResult BeginRead
      ( byte[] buffer
      , int offset
      , int count
      , AsyncCallback callback
      , Object state
      )

int EndRead(IAsyncResult asyncResult )
```

Asynchronous version

Or block anyway if ou are impatient

```
delegate void AsyncCallback(IAsyncResult ar)

interface IAsyncResult
{
      object AsyncState { get; }
      WaitHandle AsyncWaitHandle { get; }
      bool CompletedSynchronously { get; }
      bool IsCompleted { get; }
}
```

```
src.BeginRead(… ,
    result =>
    {
        var _src = result.AsyncState
                        as Stream;
        … _src.EndRead …
}, src)
```

"Safe" because you know from context Read is done

Designed before generics and lambdas

```
Func<int,bool> f = n => { System.Threading.Thread.Sleep(n); return true; };

var v = f.BeginInvoke(4711, result =>
        {
            result.IsCompleted.Dump("Completed!");
            var b = f.EndInvoke(result);
            b.Dump("value");
        }, null);

(!v.IsCompleted).Dump("Still running?");
```

Every delegate comes with BeginXXX, End XXX

Results λ SQL IL

Format Export

Still running?
True

Completed!
True

value
True

immediately

asynchronously

Query successful

Press Ctrl+Shift+F5 to cancel all threads

Show Explorer Panels (Shift+F8)

Stream.ReadAsync Method (Byte[], Int32, Int32)

msdn

.NET Framework 4.5 | 0 out of 1 rated this helpful - Rate this topic

Asynchronously reads a sequence of bytes from the current stream and advances the position within the stream by the number of bytes read.

Namespace: System.IO
Assembly: mscorlib (in mscorlib.dll)

Syntax

C# | C++ | F# | VB

```
[ComVisibleAttribute(false)]
[HostProtectionAttribute(SecurityAction.LinkDemand, ExternalThreading = true)]
public Task<int> ReadAsync(
        byte[] buffer,
        int offset,
        int count
)
```

Parameters

buffer

Type: System.Byte[]

The buffer to write the data into.

offset

Type: System.Int32

The byte offset in *buffer* at which to begin writing data from the stream.

count

Type: System.Int32

The maximum number of bytes to read.

Return Value

Type: System.Threading.Tasks.Task<Int32>

```
Func<int,Task<bool>> f = n => TaskEx.Run(delegate
{
    System.Threading.Thread.Sleep(n); return true;
});

var task = f(4711).ContinueWith(_task =>
{
   _task.IsCompleted.Dump("Completed!");
   var b = _task.Result;
   b.Dump("value");
});

(!task.IsCompleted).Dump("Still running?");
```

Show Explorer Panels (Shift+F8)

Results λ SQL IL

Still running?
True

Completed!
True

value
True

Similar as before
CPS galore

```
Func<int,Task<bool>> f = n => TaskEx.RunEx<bool>(async delegate
{
    await TaskEx.Delay(n);
    return true;
});

var task = f(4711);

(!task.IsCompleted).Dump("Still running?");

var b = await task;

task.IsCompleted.Dump("Completed!");
b.Dump("value");
```

Show Explorer Panels (Shift+F8)

Results λ SQL IL

Still running?
True

Completed!
True

value
True

Straightline code

```
async Task<long> CopyToAsync
      (Stream src, Stream dst)
{
   var buffer = new byte[0x1000];
   var bytesRead = 0;
   var totalRead = 0L;
   while ((bytesRead =
     await src.ReadAsync
             (buffer, 0,buffer.Length)) > 0)
   {
      await dst.WriteAsync(buffer, 0, bytesRead);
      totalRead += bytesRead;
   }
   return totalRead;
}
```

```
Task<long> CopyToManual(Stream src, Stream dst)
{
var tcs = new TaskCompletionSource<long>(); byte[] buffer =
null; int bytesRead = 0; long totalRead = 0;
TaskAwaiter<int> readAwaiter; TaskAwaiter writeAwaiter; var
state = 0; Action act = null;
act = () => { while (true) { switch (state) { case 0: buffer =
new byte[0x1000]; totalRead = 0; state = 1; break; case 1: state
= 2; readAwaiter = src.ReadAsync(buffer, 0,
buffer.Length).GetAwaiter(); if (readAwaiter.IsCompleted) { }
else { readAwaiter.OnCompleted(act); return;} break; case 2: if
((bytesRead = readAwaiter.GetResult()) > 0) {
state = 3; writeAwaiter = dst.WriteAsync(buffer, 0,
bytesRead).GetAwaiter(); if (writeAwaiter.IsCompleted) { }
else { writeAwaiter.OnCompleted(act); return;} } else state = 4;
break; case 3: writeAwaiter.GetResult(); totalRead += bytesRead;
state = 1; break; case 4: tcs.SetResult(totalRead); return; }}};
act();  return tcs.Task;
}
```

```
class Task<T>
{
    Task<S> ContinueWith<S>
        (Func<Task<T>, S> continuation){…}

    T Result { get{…}; }
}
```

```
ContinueWith :: Task<T> → (Task<T>→S) → Task<S>

Result :: Task<T> → T
```

Hey, I have seen that face before

Blenders

T--Insert-->B<T>

Once in never out

S -- F --> T

➔

B<S> -- Select(F) --> B<T>

G<G<S>> -- Blend-> G<S>

S --F--> B<T>

➔

B<S> -- Bs.SelectMany(F) =

Bs.Select(F).Blend()--> B<T>

LINQ!

```
B<S> Repeat<S>(S item, int n)
```

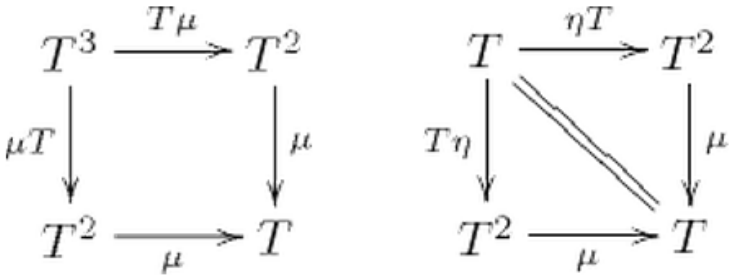
```
B<T> SelectMany<S,T>
  (B<S> blender,
    Func<S,B<T>> selector)
```

Formal definition [edit]

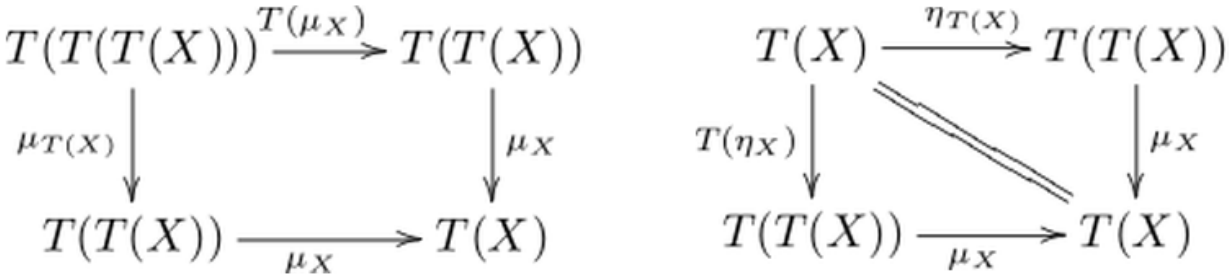
If C is a category, a **monad** on C consists of a functor $T: C \to C$ together with two natural transformations: $\eta: 1_C \to T$ (where 1_C denotes the identity functor on C) and $\mu: T^2 \to T$ (where T^2 is the functor $T \circ T$ from C to C). These are required to fulfill the following conditions (sometimes called coherence conditions):

- $\mu \circ T\mu = \mu \circ \mu T$ (as natural transformations $T^3 \to T$);
- $\mu \circ T\eta = \mu \circ \eta T = 1_T$ (as natural transformations $T \to T$; here 1_T denotes the identity transformation from T to T).

We can rewrite these conditions using following commutative diagrams:



See the article on natural transformations for the explanation of the notations $T\mu$ and μT, or see below the commutative diagrams not using these notions:



The first axiom is akin to the associativity in monoids, the second axiom to the existence of an identity element. Indeed, a monad on C can alternatively be defined as a monoid in the category $\mathbf{End}_C$ whose objects are the endofunctors of C and whose morphisms are the natural transformations between them, with the monoidal structure induced by the composition of endofunctors.

Gumballs

```
T Result(G<T> gumballs)
```

Once in never out

S -- F --> T

➔

G<S> -- Select(F) --> G<T>

G<S> -- Franchise--> G<G<S>>

G<S> --F--> T

➔

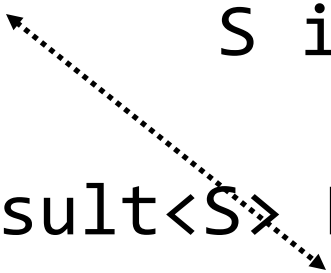
G<S> -- Gs.ContinueWith(F) =

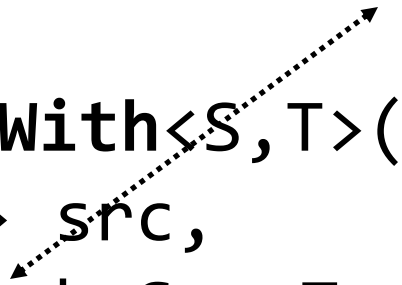
Gs.Franchise().Select(F)--> G<T>

Await (comonad)

```
S Result<S>(G<S> xs)

G<T> ContinueWith<S,T>(G<S> xs,
    Func<G<S>,T> continuation)
```

```
IEnumerable<S> Singleton<S>(
        S item)

S Result<S> Result(
        Task<S> task)

IEnumerable<T> SelectMany<S,T>(
        IEnumerable<S> src,
        Func<S, IEnumerable<T>> selector)

Task<T> ContinueWith<S,T>(
        Task<S> src,
        Func<Task<S>, T> continuation)
```





Dual (category theory)

From Wikipedia, the free encyclopedia

In category theory, a branch of mathematics, **duality** is a correspondence between properties of a category C and so-called **dual properties** of the opposite category C^{op}. Given a statement regarding the category C, by interchanging the source and target of each morphism as well as interchanging the order of composing two morphisms, a corresponding dual statement is obtained regarding the opposite category C^{op}. **Duality**, as such, is the assertion that truth is invariant under this operation on statements. In other words, if a statement is true about C, then its dual statement is true about C^{op}. Also, if a statement is false about C, then its dual has to be false about C^{op}.

Given a concrete category C, it is often the case that the opposite category C^{op} per se is abstract. C^{op} need not be a category that arises from mathematical practice. In this case, another category D is also termed to be in **duality** with C if D and C^{op} are equivalent as categories.

Blenders/Monads are the duals of gumball machines coMonads

One Thing C# Cannot Do is Higher-Order Kinds

8.8.4 The foreach statement

msdn

Visual Studio .NET 2003 | 7 out of 12 rated this helpful - Rate this topic

The `foreach` statement enumerates the elements of a collection, executing an embedded statement for each element of the collection.

foreach-statement:
foreach (*type* *identifier* in *expression*) *embedded-statement*

The *type* and *identifier* of a `foreach` statement declare the iteration variable of the statement. The iteration variable corresponds to a read-only local variable with a scope that extends over the embedded statement. During execution of a `foreach` statement, the iteration variable represents the collection element for which an iteration is currently being performed. A compile-time error occurs if the embedded statement attempts to modify the iteration variable (by assignment or the `++` and `--` operators) or pass the iteration variable as a `ref` or `out` parameter.

The type of the *expression* of a `foreach` statement must be a collection type (as defined below), and an explicit conversion (Section 6.2) must exist from the element type of the collection to the type of the iteration variable. If *expression* has the value `null`, a `System.NullReferenceException` is thrown.

A type `C` is said to be a collection type if it implements the `System.Collections.IEnumerable` interface or implements the collection pattern by meeting all of the following criteria:

- `C` contains a `public` instance method with the signature `GetEnumerator()` that returns a *struct-type*, *class-type*, or *interface-type*, which is called `E` in the following text.
- `E` contains a `public` instance method with the signature `MoveNext()` and the return type `bool`.
- `E` contains a `public` instance property named `Current` that permits reading the current value. The type of this property is said to be the element type of the collection type.

A type that implements `IEnumerable` is also a collection type, even if it does not satisfy the conditions above. (This is possible if it implements some of the `IEnumerable` members by means of explicit interface member implementation, as described in Section 13.4.1.)

The `System.Array` type (Section 12.1.1) is a collection type, and since all array types derive from `System.Array`, any array type expression is permitted in a `foreach` statement. The order in which `foreach` traverses the elements of an array is as follows: For single-dimensional arrays, elements are traversed in increasing index order, starting with index `0` and ending with index `Length - 1`. For multi-dimensional arrays, elements are traversed such that the indices of the rightmost dimension are increased first, then the next left dimension, and so on to the left.

A `foreach` statement of the form:

```
foreach (ElementType element in collection) statement
```

corresponds to one of two possible expansions:

```
void Main()
{
    var laser = new Laser{ Greeting = "Hello There!" };
    foreach(char c in laser)
    {
        Console.Write(c);
    }
}
```

Show Explorer Panels (Shif

```
public class Laser
{
    public string Greeting;

    public SummerSchool GetEnumerator()
    {
        return new SummerSchool{ Greeting = Greeting };
    }

    public class SummerSchool
    {
        int i = -1;
        public string Greeting;

        public bool MoveNext()
        {
            if(i++ < Greeting.Length-1) return true;
            return false;
        }

        public char Current { get{ return Greeting[i]; }}
    }
}
```

No IEnumerable
IEnumerator interfaces
in sight

Same for `await from-where-select`

http://blogs.msdn.com/b/pfxteam/archive/2011/01/13/10115642.aspx

```
Awaiter<OtherType> GetAwaiter(this MyType){ … }

class Awaiter<T>
{
    bool IsCompleted { get; }
    void OnCompleted(Action continuation);
    T GetResult(); // or void
}

OtherType x = await (E as MyType)
```

Await as explicit Anonymous “cast”

Trick: reuse `Task<T>` and `TaskAwaiter`