# Encoding monadic computations using iterators in C# 2.0

(Supplementary material)

## 1. F# asynchronous workflows

The following example demonstrates how the F# compiler translates asynchronous workflow (or any monadic computation in general) to calls to primitive methods provided by the computation builder such as `Bind`, `While` and `Return`. The original code written by the user looks like this:

```
let downloadUrl(url:string) = async {
  let req = HttpWebRequest.Create(url)
  let! rsp = req.AsyncGetResponse()
  let strm = rsp.GetResponseStream()
  let buf = Array.zeroCreate(8192)
  let state = ref 1
  while !state > 0 do
    let! read = strm.AsyncRead(buf, 0, 8192)
    Console.WriteLine("got {0}b", read);
    state := read }
```

The compiler translates each use of `let!` keyword into a call to the `Bind` member that takes the rest of the computation wrapped into a function as the last parameter. Similarly, `while` loops are translated into calls to the `While` member:

```
let req = HttpWebRequest.Create(url)
  async.Bind(req.AsyncGetResponse(), fun rsp ->
    let strm = rsp.GetResponseStream()
    let buf = Array.zeroCreate(8192)
    let state = ref 1
    async.While((fun () -> !state > 0),
      async.Bind
        (strm.AsyncRead(buf, 0, 8192), fun read ->
          Console.WriteLine("got {0}b", read);
          state := read
          async.Return() )))
```

In some cases, the F# compiler also needs other primitives such as `Combine` or `Zero`. These cases are documented in the F# language specification[1].

## 2. Case Study: Asynchronous C#

In this section, we look at simple asynchronous method that downloads all data from a stream in a buffered way and then interprets the data as a string. The first listing shows how the code looks when written using the asynchronous library presented in the article:

```
IEnumerator<IAsync> ReadToEndAsync(Stream s) {
  var ms = new MemoryStream();
  byte[] bf = new byte[1024];
  int read = -1;
  while (read != 0) {
    var op = s.ReadAsync(bf, 0, 1024).AsStep();
    yield return op;
    ms.Write(bf, 0, op.Value);
    read = op.Value;
  }
  ms.Seek(0, SeekOrigin.Begin);
  string s = new StreamReader(ms).ReadToEnd();
  yield return AsyncResult.Create(s);
}
```

To implement the same functionality in the usual programming style in C#, we need to create a class that represents a state machine. In this case, there is only a single state, which is to read the next 1kb of data from the stream. When the operation returns 0 bytes, meaning that the download has completed, it converts the data into string and returns the string (by calling a continuation), otherwise it recursively continues downloading:

```
class ReadToEndState {
  MemoryStream ms = new MemoryStream();
  Stream stream;
  Action<string> k;

  // Initialize state machine for downloading stream
  public ReadToEndState
      (Stream stream, Action<string> k) {
    this.stream = stream;
    this.k = k;
  }

  internal void Step() {
    byte[] buffer = new byte[1024];
    // Read 1kb  of data asynchronously
    stream.BeginRead(buffer, 0, 1024, ar => {
      var count = stream.EndRead(ar);
      ms.Write(buffer, 0, count);
      if (count == 0) {
        ms.Seek(0, SeekOrigin.Begin);
        string s = new StreamReader(ms)
                     .ReadToEnd();
        // Return the parsed string via continuation
        k(s);
      } else {
        // Run the state-machine step repeatedly
        Step();
      }
    }, null);
  }
}

static void ReadToEndAsync
    (this Stream stream, Action<string> k) {
  // Construct state-machine and start the first step
  new ReadToEndState(stream, k).Step();
}
```

[1] Available online at: http://research.microsoft.com/apps/pubs/default.aspx?id=79948