From Iterators to Computations and Beyond

Claudio V. Russo
Microsoft Research
crusso@microsoft.com

Gavin Bierman
Microsoft Research
gmb@microsoft.com

Tomas Petricek
Faculty of Mathematics and Physics
Charles University
tomas@tomasp.net

Abstract

Many applications today have to perform potentially long-running operations, typically arising from network access, calling a database or web service, or performing I/O operations in general. Whilst often peripheral to the main purpose of the application, these operations can have a big impact on the overall codebase. To stop these long-running operations blocking the main calling thread, the current practice is to use non-blocking, *asynchronous* operations instead. Asynchronous operations are typically issued by passing callbacks, which quickly leads to a mess of continuation passing code. This not only inverts the control structure of the original application, but also prevents the programmer from using standard control-flow constructs. Whilst perhaps acceptable to a functional programmer, this mode of programming is especially debilitating for the imperative programmer, who is accustomed to writing code using stateful loops, mutating local variables and so on. All this state must be manually saved, before initiating the asynchronous operation, then restored upon resumption, in its callback.

In this paper we describe a simple yet expressive technique for writing strongly typed, asynchronous code in a familiar sequential style in C♯ and related languages. Our proposal does not require either extensions to the language or to the runtime, but builds on existing support for *iterators*.

1. Introduction

Dealing with long-running operations such as network access, or calling a web service, is an everyday problem for developers. The issue is stopping such operations blocking the calling thread, and the typical solution is to make these operations *asynchronous*. In other words, the operations should complete in parallel from the main application thread and thus finish asynchronously. Thus, when an application calls an asynchronous method, it can continue executing while the asynchronous method performs its task.

In this paper we are concerned with the practicalities of asynchronous programming in object-oriented languages such as C♯ and Java. As we shall see, the situation is not entirely satisfactory, with the required programming style being awkward (the code has to be written in a continuation-passing style), fragile (continuation-passing is difficult and prone to error) and weak (many useful language constructs, including loops, are not directly available). Our aim is to show that asynchronous programming can be supported in an entirely different way by building on the power of an existing language feature: *iterators*.

Although our techniques are not specific to any particular language, this paper is intentionally concrete, with many code fragments to illustrate our techniques and several examples to show them in action.[1] We fix on the C♯ programming language [6], not least because C♯ provides quite rich support for iterators. We give a brief introduction to C♯ iterators in §2.1. However, we repeat that our techniques would work for any language that supports iterators, e.g. Python.

Let us consider a simple motivating example, taken from [11]. Suppose we wish to download the first kilobyte of content from some website:

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

The problem with this code is that the second and fourth lines invoke methods that are potentially long running. Whilst executing these operations the main executing thread is blocked. Furthermore, if we wanted to scale this code and download from hundreds of sites in parallel, for example, we would be forced to create hundreds of threads which introduces significant overheads and context switches.

The solution to hiding latency is to use asynchronous programming. The .NET framework's *Asynchronous Programming Model* provides the so-called APM design pattern

[1] All the code in this paper can be downloaded from `http://research.microsoft.com/users/crusso/comp/comp.zip`.

for asynchronous operations.[2] This design pattern is implemented as two methods, named by convention Begin*Name* and End*Name*, that together form the asynchronous version of the *Name* operation. After calling the Begin*Name* method, the application can continue executing on the calling thread, whilst the asynchronous operation completes. For every call to Begin*Name* there should be a matching call to End*Name* to get the results of the operation.

The .NET framework provides a number of useful asynchronous operations adhering to the APM design pattern. For example, our code from above can be rewritten as follows:

```
var req = HttpWebRequest.Create(url);
req.BeginGetResponse(a1 => {
   var rsp = req.EndGetResponse();
   var strm = rsp.GetResponseStream();
   strm.BeginRead(buffer, 0, 1024, a2 => {
      int read = strm.EndRead(a2);
         //...
   }, null);
}, null);
```

In this code fragment, we make use of C♯'s lambda expressions: `a => {...}`, which is shorthand for an anonymous method expression `delegate(T a){...}`, where `T` is the inferred type for the parameter `a`. We use lambda expressions to register callbacks for the APM methods `BeginGetResponse` and `BeginRead`. Further details of the APM design pattern can be found in §2.3.

Whilst asynchronous, this explicit construction of the continuation (the nested callbacks) makes the code much less readable than the familiar direct style of the synchronous code. This resulting style is often known as "inversion of control", as the asynchronous method is called with the computation that should be executed after the method has completed. Control has been inverted as it has been passed to the asynchronous method. But there is a bigger problem: imagine that we wished to download not just the first kilobyte of content, but repeatedly read one kilobyte chunks until we had fetched the whole page. The natural solution would be to use a loop, but this is not available to us as we can not have the loop spanning across nested callbacks. The only solution at this point is to essentially code up some form of state machine, using shared state.

It is exactly this sort of low-level, state machine logic that we should like to avoid. Our aim is to provide an asynchronous programming style that is close to the natural, direct synchronous style, where all the relevant state is preserved automatically across asynchronous method calls. Our inspiration comes from the *asynchronous workflows* that were introduced in the F♯ functional programming language [24].

F♯ supports the notion of an `async` block. Inside the block, the compiler automatically generates the continuation-passing code for specially marked operations. Moreover, all the standard language constructs can be used inside the block, including loops. For example, here is the F♯ asynchronous workflow equivalent of the previous code (it simply downloads the entire contents of the webpage, printing its progress):

```
let downloadURL(url:string) = async{
   let req = HttpWebRequest.Create(url)
   let! rsp = req.AsyncGetResponse()
   let strm = rsp.GetResponseStream()
   let buffer = Array.zeroCreate(1024)
   let state = ref 1
   while !state > 0 do
      let! read = strm.AsyncRead(buffer, 0, 1024)
      Console.WriteLine("loaded {0} bytes", read);
         state := read
}
```

Notice that in this code where we previously used explicitly the APM methods, in F♯ asynchronous workflows, we use the `let!` keyword, which denotes the monadic value binding, and calls to the F♯ primitive asynchronous actions (which wrap the APM `begin` and `end` calls). For asynchronous workflows, the construct `let! x = e1` means that expression `e1` is performed asynchronously and when it has completed the result is bound to `x`. In the context of this paper, it is important to observe two things of the code fragment above: (1) that the code is very similar to the familiar, direct, synchronous code and does not have any explicit inversion of control; and (2) that all the usual F♯ language constructs are also available inside the `async` block including, for example, the `while` loop.

The aim of this paper is to provide similar support for asynchronous programming in C♯. However, we'd like to offer this support without a change to the language. In F♯ the `async` block is actually just syntactic sugar, which is preprocessed into calls to the underlying `AsyncBuilder` object. Unfortunately, the resulting code relies heavily on the very continuation-passing style that we are trying to avoid. In this paper, we hope for the best of both worlds: no change to the language and no continuation-passing code.

Our core idea is to build on the surprisingly rich expressive power of C♯ iterators. We use these to define what we call *computations*. Just as an iterator is a method that yields an ordered sequence of values, a computation yields an ordered sequence of instructions. It is simple to translate from arbitrary method blocks to computations. However, computations can be co-operatively suspended and resumed at user-specified yield points. Unlike ordinary iterators, but like F♯'s asynchronous workflows, these computations may be deeply nested. Suspending and resuming a computation is a con-

[2] Actually, it provides two design patterns: the so-called `IAsyncResult` pattern described here, and the event pattern. As the former is more powerful we shall not discuss the latter pattern.

stant time operation, regardless of nesting depth. In addition, a computation may start on one CLR thread and be resumed on another. Although built from iterators, our computations (like ordinary methods) may return at most one value. Importantly, our computations are strongly typed. Exceptions raised by computations are propagated in the usual way, and with care can be handled appropriately.

This paper is organized as follows. We have attempted to make this paper self-contained, and so in §2 we informally discuss the particular features of C♯ and .NET framework that are perhaps not so well-known. In §3 we give an informal introduction to computations. In §4 we show how an example program can be rewritten using computations. In §5 we show how computations can be used to provide type-safe, nested, asynchronous programming without inversion of control in C♯. In §6 we give details of our actual implementation. We show in §7 how our technique is extensible, showing how futures (§7.1) and reactive programming (§7.2) can be coded up. A further benefit of our approach is that instructions can be given alternative interpretations. In §8 we show how this can be used to give a debugging interpretation whereby the CLR stack can be used to interpret a computation's stack transitions. In §9 we consider the problem of adapting our techniques to encode arbitrary monadic code. We review related work and propose some future work in §10, before concluding briefly in §11.

2. Background

In this section we give details of the specific C♯ and .NET features that we use extensively in this paper, including C♯ iterators and the APM design pattern. The expert reader can safely skip this section.

2.1 C♯ iterators

An enumerator is an object that allows traversal over a collection and is represented in the .NET framework by the `IEnumerator<T>` interface. It has a `Current` property to obtain the value of the current element, and a method `MoveNext` to move on to the next element (which returns a boolean value indicating whether the end of the collection has been reached.

Typically collection classes implement the `IEnumerable<T>` interface. An enumerator can then be obtained by calling the `GetEnumerator` method on an `IEnumerable` object. (This abstraction enables us to have multiple enumerators over the same collection.) For example, if we have a collection `names` of type `IEnumerable<String>` then we can print out all the elements as follows:

```
IEnumerator<String> cursor = names.GetEnumerator();
while (cursor.MoveNext())
   Console.WriteLine(cursor.Current)
```

This code pattern is very common, and C♯ provides the convenient `foreach` statement to allow much of boilerplate code to be hidden.

```
foreach (String n in names)
   Console.WriteLine(n)
```

While consuming enumerators is quite straightforward, the difficulty has traditionally been in implementing the `IEnumerator` interface as it often requires complicated state-machine machinery. Fortunately, C♯ 2.0 made it easier to write methods that return enumerators, by allowing it to be implemented using an iterator block. Such a method is referred to as an *iterator*.[3] An iterator is a statement block that yields an ordered sequence of values (all of the same type). An iterator is distinguished from a normal statement block by the presence of one or more `yield` statements. The `yield return` statement produces the next value of the iteration, and the `yield break` statement indicates that the iteration is complete. Iterators are not permitted to contain general `return` statements. This leads to very compact code, for example:

```
public static IEnumerator<string> Sons(){
   yield return "Reuben";
   yield return "Simeon";
   yield return "Levi";
   yield return "Judah";
   //...
}
```

Behind the scenes, the compiler converts iterators into enumerators, encapsulating the code in the iterator block as a finite state machine tracking the state of local variables and the position of the last `yield return` statement, and implementing the enumerator interfaces. The iterator given above can, for example, be consumed directly as a normal enumerator; the following code prints the first two elements of the iterator:

```
var sonsofJacob = Sons();
sonsofJacob.MoveNext();
Console.WriteLine(sonsofJacob.Current);
sonsofJacob.MoveNext();
Console.WriteLine(sonsofJacob.Current);
```

For the purposes of this paper, it is important to understand the operational behaviour of the iterator. When we call the method `Sons()`, none of the method body is executed. What is returned is the compiler-generated `IEnumerator<string>` object. Only when we invoke the `MoveNext` method on the enumerator, will this start executing the iterator body. It is executed until it reaches a `yield return` statement, at

[3] Unfortunately, a C♯ enumerator is often called an iterator in other languages, and a C♯ iterator is often called a generator. Clearly, such a conflict is confusing, but we shall stick with the .NET terminology.

which point computation of the iterator is suspended, and the invocation of the `MoveNext` is completed. The value of the `yield return` expression is then available as the value of the `Current` property of the enumerator. Should the `MoveNext` method be invoked again on the enumerator, computation of the iterator is resumed from the statement following the previous `yield return` statement. It is this co-operative, coroutine-like behaviour between the iterator and the calls to the enumerator that we will exploit extensively in this paper.

A more interesting example of an iterator is as follows.

```
public static IEnumerator<string> Numbers(){
   int i = 0;
   while (true)
      yield return i++.ToString();
}
```

This iterator both maintains state (the value of the local variable `i`), yields values from within a control structure and could potentially be called infinitely often. However the lazy behaviour of the generated enumerator object is evidenced by the following code which prints the first four natural numbers.

```
var nats = Numbers();
for (int j = 1; j <= 4; j++){
   nats.MoveNext();
   Console.WriteLine(nats.Current);
}
```

Note that `yield return`s may have enclosing `finally` clauses: to ensure these get executed, even when an enumeration is abandoned, users of the low-level `IEnumerator` methods are also expected to call its (inherited) `Dispose()` method.

2.2 Other C♯ features

There are some lesser well-known C♯ features that we shall make extensive use of in the rest of this paper that we shall describe briefly here. First, C♯ supports *nested classes*: classes defined inside another. A class `N` nested inside class `C` is referred to outside the class definition as class `C.N`. A nested class can refer to the static members of its enclosing class, but not the instance members (unlike Java's inner classes). A nested class can also refer to any generic type parameters of its containing class (but not vice-versa). For example, the following is valid:

```
class Container<S> {
   class Nested<T> {
      S a;
      T b;
   }
}
```

C♯ allows a class definition to be split, possibly across over two or more source files, by using a `partial` modifier. This indicates that other parts of the class can be defined in the namespace. All the parts must use the `partial` modifier and must have the same accessibility. *Partial classes* allow simple separation of concerns of the source code, for example:

```
// File A
public partial class CoOrds {
    private int x;
    private int y;
}
// File B
public partial class CoOrds {
    public void PrintCoOrds() {
        Console.WriteLine("CoOrds: {0},{1}", x, y);
    }
}
```

C♯ provides a general means to extend existing types and constructed typed with additional methods. These additional methods are known as *extension methods*, and are essentially static methods that can be invoked using instance method syntax. Extension methods are declared by specifying the modifier `this` on the first parameter of the method. For example, an extension method with the following signature:

```
public static int WordCount(this String str){ ... }
```

If this method is in scope, then it can be invoked `s.WordCount()` where `s` is a string.

2.3 Details of the APM Pattern

As alluded to in §1, in the .NET framework, any synchronous operation that supports an asynchronous calling convention should adhere to the APM `IAsyncResult` design pattern. According to the APM, given a synchronous operation, `Op`, with method signature

```
U Op(T1 t1,...,Tn tn)
```

its asynchronous variant is meant to provide two methods with names `BeginOp` and `EndOp` and the following derived signatures:

```
IAsyncResult BeginOp(T1 t1,...,Tn tn,
                     AsyncCallback callback,
                     object state);
U EndOp(IAsyncResult iar);
```

Note that the initial parameters to `BeginOp`, and the return type of `EndOp`, coincide with `Op`'s original signature.

Here, `IAsyncResult` is an interface, and `AsyncCallback` is a delegate type that takes an `IAsyncResult` argument and returns `void`. Both types are defined in the .NET framework. (We can ignore the `state` argument for our purposes.)

Operationally, the `BeginOp(t1,...,t2,callback,state)` method is meant to return immediately after *initiating* the

asynchronous operation, but without waiting for it to finish. When supplied with a non-null `callback`, the callback will be run, once the operation completes, on a thread from the thread pool. To terminate the protocol, the callback is required to eventually execute `EndOp` on its actual argument, and thus obtain the result (a value of type `U` or an exception) of the operation, as well as disposing of any resources used by the operation.

Note that the callback and state argument are optional, so `BeginOp` also returns a handle (typically the same `IAsyncResult` value that would have been supplied to the callback) on which a thread may wait before eventually supplying it to `EndOp` to complete the protocol. This waiting can be performed in various ways, including polling and blocking on an OS wait handle (see [17] for more details). Since we would like to avoid blocking, yet be called on completion, we will focus on the simple call-with-callback pattern.

As a minor aside, we should point out that the APM pattern is only weakly typed (perhaps because it pre-dates .NET Generics). Calling `EndOp(iar)` can fail with an `InvalidCast` exception (e.g., when applied to the `IAsyncResult` of an unrelated operation).

3. An informal introduction to computations

Our main contribution is to show how we can use the power of iterators to define the notion of a *computation*, which is a special instance of an iterator that yields instructions.

In this section we attempt to give an intuitive, informal introduction to computations. To start, we first recall the use of *call stacks* in the implementation of most programming languages. A call stack is just a list a *method frames*. Each frame records the state (i.e. the current program counter and the values of any arguments and local variables) of some finite state machine (the actual code for the method). All but the topmost frame denote suspended computations. Execution proceeds within the topmost frame until it encounters one of several instructions that exit the frame and transition the stack: a return instruction, depositing a value and popping the stack; a call instruction, growing the stack with a new frame; an uncaught throw instruction, unwinding the stack to propagate its exception; a blocking call, suspending execution of the entire stack until some condition is enabled by a concurrent thread. Thus we can think of execution of the current frame as essentially the processing of a stream of instructions that manipulate the entire call stack. A frame is naturally typed by the type of value it returns to its caller. In turn, instructions are naturally typed by the type of frame that issues them and, when appropriate, by the return type of the callee they invoke.

The essence of our notion of a computation is that one can view a single method computing a value of type `T`, as equivalently a *stream* of instructions of type `I<T>`, and thus as a value of type `IEnumerable<I<T>>`. A particular instance of a method, i.e. a frame, is just an enumerator of type `IEnumerator<I<T>>`, obtained from the method value. Finally, a stack of frames, corresponding to a snapshot of program execution, is just a stack of instruction enumerators of various types, each linked to its preceding frame by a call instruction of the appropriate type.

What is the set of instructions? For starters (we will encounter more instructions later), a `T`-returning frame may execute the following instructions:

`Return(t)` returns the value `t` of type `T`, popping the stack.

`Call<U>(c)` calls another `U`-returning computation `c`. Executing this instruction suspends the current frame until `c` is done, by installing a new frame for `c` on the stack. The callee has type `IEnumerable<I<U>>` so its frame will have type `IEnumerator<I<U>>`. Note that `U` is completely independent of `T`: as with ordinary methods calls, the return type of a caller and its callee are unrelated. Thus we support calling various types of callees from the same computation.

`TailCall(c)` calls another `T`-returning computation `c`. Execution of this instruction discards the current frame and installs a frame for `c`. Since the new frame must return to the caller of the current frame, `c` must have the *same* return type, and thus computation type, as its caller (i.e. `IEnumerable<I<T>>`).

More precisely, these are families of instructions, indexed by the return type `T`, and are thus represented by instances of nested classes `I<T>.Return`, `I<T>.Call<U>`, `I<T>.TailCall` —each concrete instruction class deriving from the abstract class `I<T>`.

Since C♯ yields are statements, and cannot return values, we work around this restriction by communicating return values through the instruction that was yielded. Thus an instruction serves a second role—acting as a receptacle for its result—typically a value but possibly an exception. The result is accessed through that instruction's `Value` property. Since the type of the instruction issued is known to the enclosing frame, the type of value returned by any instruction can both vary at each call site yet remain strongly typed.[4]

How do we execute a computation? We first create a singleton stack with an instance of the computation (an enumerator) and an initial call instruction. Then we repeatedly transition the stack one instruction at a time until it is empty. The result of the computation can then be read off the initial call instruction.

How do we transition the stack? We pull the next instruction from the frame on top of the stack (as the frame is an enumerator we simply used `MoveNext` and `Current`). If there

[4] Our instruction set is an instance of a *Generalized Algebraic Datatype* [8]. For example, the return type `U` of the `Call<U>` instruction (itself a subclass of `I<T>`) is an existential type, hidden from the consumer of the instruction stream, but known to the producer.

is none, control has fallen off the end of the enumerator and we pop the stack (returning a default value to the caller). If pulling the instruction fails by throwing an exception, we catch and store that exception in the call instruction waiting on the stack, then pop the stack. If there is some instruction, we transition the current stack according to that instruction and return the resulting stack: this stack may be the same, smaller or larger, depending on the instruction.

The instruction interpreter does not explicitly switch on the tag of a finite set of instructions. Instead, each instruction has a virtual method (declared in `I<T>` but overridden in the instruction class) that determines its own effect on the stack. The use of subclassing and virtual methods ensures that our instruction set is *extensible*.

In order to faithfully propagate exceptions from callee to caller, every call to a `U`-computation must be followed by an immediate read of its `Value` property: this will either return a proper value of type `U`, or re-throw any stored exception (which can then be caught or propagated in the usual manner). Thus every call site should adhere to an *allocate-yield-read* protocol:

```
var i = new I<T>.Call<U>(comp(...));
                    // allocate a call
yield return i;     // yield it
var u = i.Value;    // read a U or throw
```

Failure to read a call's `Value` property is not catastrophic but any exception that was raised will be silently discarded.

4. A simple example: Fibonacci

It's time for an example. Consider this (stylized) recursive Fibonacci function:

```
static int fib(int k) {
 if (k <= 1) return k;
 else {
  var n = fib(k - 1);
  var m = fib(k - 2);
  return n + m;
 }
}
```

To express this as a computation, we rewrite `fib` to return a stream of `int`-instructions, preserving the general control-flow but re-expressing each return and (recursive) call as yielding an instruction.

```
// naive Fibonacci
static IEnumerable<I<int>> Fib(int k) {
 if (k <= 1) yield return new I<int>.Return(k);
 else {
  var ni = new I<int>.Call<int>(Fib(k - 1));
  yield return ni;
  var n = ni.Value;
  var mi = new I<int>.Call<int>(Fib(k - 2));
  yield return mi;
  var m = mi.Value;

  yield return new I<int>.Return(n + m);
 }
}
```

Although perhaps not pretty, this fragment contains no explicit continuation-passing code. Indeed, it's clear that a compiler or preprocessor could generate this code from the original using a simple source-to-source translation. At this point we can now see the main advantages of our approach. Firstly, the bulk of the code remains as normal, just like the code inside the `async` blocks in F♯. Only some of the operations, method returns and method calls, need to be rewritten (much as with the use of `let!` in F♯ asynchronous workflows). Whilst this is a little tedious to perform by hand, the resulting code is still more readable than continuation-passing code.

Having transformed the program, we can now get an instance of the computation (an enumerator) and define an interpreter that "executes" the computation. In fact, we can execute the computation synchronously, or asynchronously.[5]

```
public static class Computation {
 // run the computation synchronously
 // may block on thread migration
 public static T Run<T>(this IEnumerable<I<T>> comp)

 // runs the computation asynchronously
 // returns a thunk to wait on.
 public static Func<T> Spawn<T>(
  this IEnumerable<I<T>> comp)

 // APM interface (non-blocking)
 public static IAsyncResult Begin<T>(
  this IEnumerable<I<T>> comp,
  AsyncCallback callback, object state)

 public static T End<T>(IAsyncResult asyncResult)
}
```

Given a computation `comp`, the extension method `comp.Run()` allocates a singleton stack with an initial `I<A>.Call<A>(comp)` instruction and an enumerator for `comp`, executes the stack and returns the result of `comp`. For example:

```
// run Fib(1000) synchronously
var i = Fib(1000).Run();
```

Extension method `comp.Spawn()` is similar but spawns the computation on a worker thread. It immediately returns a function that, when invoked, waits until the call has returned with a result, blocking if necessary. For example:

[5] The implementation of these methods is discussed in §6.2.

```
// run Fib(1000) asynchronously
var future = Fib(1000).Spawn();
// meanwhile, do some work
// finally, wait until/unless future is done.
int j = future();
```

Finally, we illustrate the use of the APM method pair, first using a non-blocking callback, and then using explicit synchronization:

```
// spawned with a callback
Fib(10).Begin(r2 =>
  Console.WriteLine(1 * Computation.End<int>(r2)),
  null);

// spawned without a callback
var r1 = Fib(10).Begin(null, null);
// meanwhile, do something
// later, synchronize
int k3 = Computation.End<int>(r1) + 0;
```

4.1 Tail Calls

Our naïve implementation of the Fibonacci function will not run out of CLR stack, but it may run out of heap and will certainly put huge pressure on the garbage collector. A more efficient implementation uses the `TailCall(comp)` instruction, allowing the implementation to run with constant stack depth:

```
// tail-recursive Fibonacci
static IEnumerable<I<int>> TailFib(
  int k, int prev, int curr) {
 if (k == 1)
  yield return new I<int>.Return(curr);
 else
  yield return new I<int>.TailCall(
   TailFib(k - 1, curr, prev + curr));
}
```

Of course, since this is a self tail call we could just have written a loop, but we also support tail calls to *other* computations.

5. Asynchronous programming

We are now in a position to use computations to deliver an asynchronous programming style analogous to F♯'s asynchronous workflows. We now have a first-class representation of a suspended computation (i.e. a stack), and we can use that representation to implement more advanced instructions that modify its execution. We introduce a family of instructions that lets us perform asynchronous method calls using matching pairs of methods adhering to the APM design pattern (see §2.3).

Fortunately, the details of the APM can be hidden from the programmer by providing a easy-to-use helper method:

```
I<T>.APM<U>.Call(BeginOp,t1,...,tn,EndOp)
```

The helper takes matching `BeginOp` and `EndOp` methods, bracketing the `Op`'s proper arguments and returns an instruction of type `I<T>`.

To perform the operation, the user just issues the instruction using the same *allocate-yield-read* protocol used for the `I<T>.Call<U>` instruction:

```
var i = I<T>.APM<U>.Call(BeginOp,t1,...,tn,EndOp)
yield return i;
var u = i.Value;
```

Under the hood, the instruction transitions the stack by first initiating the asynchronous operation, using `BeginOp` with a private callback. On completion, the callback resumes execution of the pending stack, on the thread pool. The interpretation of the instruction itself is to just return the empty stack (having saved the pending stack in the callback). On resumption, reading the `Value` property of the instruction will call `EndOp` on the `IAsyncResult` passed to the callback; thus completing the APM protocol.

As a concrete example, let us return to the example from §1, and define an I/O-bound computation, `ReadToEnd(s)`, that reads an entire stream, `s`, of characters in 1K chunks, using non-blocking I/O:

```
// read a whole stream, asynchronously, in 1K chunks.
static IEnumerable<I<string>> ReadToEnd(Stream str)
{
 var ms = new MemoryStream();
 byte[] bf = new byte[1024];
 int read = -1;

 while (read != 0) {

  var beginRead = I<string>.APM<int>.
     Call(str.BeginRead,bf, 0, 1024, str.EndRead);
  yield return beginRead;
  read = beginRead.Value;

  ms.Write(bf, 0, read);
 }
 ms.Seek(0, SeekOrigin.Begin);
 var s = new StreamReader(ms).ReadToEnd();
 yield return new I<string>.Return(s);
}
```

The code asynchronously calls `s.BeginRead(...)` in a loop, preserving computational state across calls. Although we could have achieved a similar effect using iterators that return higher-order values (as in the CCR [2]) the fact that we can also nest computations is novel. It lets us suspend and resume execution of outer computations without needing to block or prematurely exit them.

For example, the following method, `DownloadPage(url)`, downloads the contents of a URL using a nested call to the previous computation, `ReadToEnd(s)`:

```
// download a web page, asynchronously,
// then return the source.
static IEnumerable<I<string>> DownloadPage(String url){
 var req = WebRequest.Create(url);

 var getResponse = I<string>.APM<WebResponse>.
      Call(req.BeginGetResponse, req.EndGetResponse);

 yield return getResponse;
 var resp = getResponse.Value;

 var str = resp.GetResponseStream();

 var readToEnd = new I<string>.
    Call<string>(ReadToEnd(str));
 yield return readToEnd;
 var html = readToEnd.Value;

 yield return new I<string>.Return(html);
}
```

The remainder of the call to `DownloadPage(url)` will resume on some thread once the call to `ReadToEnd(str)` returns on that thread.

Although its execution is eventually spread across several callback threads, the code for `DownloadPage` and `ReadToEnd` is written in a sequential style (preserving state across asynchronous calls, including those deeper down the call-chain).

Notice that `DownloadPage(url)` also illustrates the ability to call several callees with different, yet strongly-typed, return types, in this case `WebResponse` and `String`.

The extensibility of the instruction set enables us to code up other behaviours. In §7 we give two further examples: futures and reactive programming.

6. Implementation

This section gives an overview of the implementation of computations. The stress that our implementation is strongly-typed, cast-free and does not rely on C♯'s runtime type passing (except, perhaps, where used within the .NET APM).

6.1 Stacks

Our representation of stacks uses just two concrete classes, `Id` and `Cons`, for constructing initial and chained stacks, respectively. However, we employ an interesting hierarchy of abstract, generic superclasses, with the following API.

```
public abstract partial class Stack {
  public abstract void Resume()
}

public abstract partial class Stack<A> : Stack {
  public abstract Stack<A> Next();

public abstract partial class Cont<T> : Stack<A> {

  public Cont<U> Push<U>(I<T>.Call<U> call)

 public partial class Cons<U> : Cont<T> {
   public Cont<U> Pop(T t);
   public Cont<T> TailPush(IEnumerable<I<T>> comp)
 }
}

public partial class Id : Cont<A> {
  public Id(I<A>.Call<A> call)
  public A Answer()
 }
}
```

The type parameters are used to capture both statically invariant and dynamically changing aspects of stack typing through respectively, invariant type parameters and hidden existential types. The hierarchy allows us to keep one eye fixed on the invariant *answer* type of a stack, while turning a blind eye to changes in the local return type of its currently topmost frame.

Our hierarchy partitions stacks into those with unknown answer type (that can just be resumed for their side-effect) and those with a known answer type `A`, that may yield some `A`-typed answer (which is either an `A`-value or a raised exception). Generic stacks are further refined into *continuations*, `Cont<T>`, with a topmost computation returning `T` for the same answer type `A`. The `Cons<U>` class is used to compose a `T`-computation with a some calling `U`-continuation to produce a `T`-continuation. The concrete `Id` continuation class is used to construct a singleton stack with coinciding return and answer type `A`.

From a functional perspective, viewed as term constructors for their superclasses, their constructors are carefully arranged to have the following (functional) types:

```
Id: I<A>.Call<A> → Cont<A>
Cont<T>.Cons<U>:
 (IEnumerator<I<T>> × I<U>.Call<T> × Cont<U>) → Cont<T>
```

These types encode the fact that an arbitrary `Stack<A>` is constructed from a sequence of nested, independently typed continuations with the same answer type, with each continuation linked to its tail continuation by a bridging call instruction. `Cons` continuations are thus daisy-chained by matching `Call` instructions until terminated by an identity continuation, `Id`.

At the highest-level of abstraction, every stack is just a heap-allocated value of abstract (non-generic) class `Stack`. Every stack supports a `Resume()` method for continuing its execution. Note that since a computation can migrate to another thread, `Resume()` may return before the computation

has produced a result, which we will signal by returning the *empty* stack, represented by `null`.

Immediately below the arbitrary `Stack` class lies the generic stack class, `Stack<A>`.

```
public abstract partial class Stack<A> : Stack {
 public abstract Stack<A> Next();
 public sealed override void Resume() {
  var s = this;
  while (s != null) s = s.Next();
 }
}
```

This class provides a single virtual method, `Next()`, for transitioning this stack to another of the same answer type, `A`. The typing of `Next()` captures the fact that, although the shape of the stack may change during execution, its answer type does not. By construction, `A` will be the return type of the innermost call instruction, i.e. the initial call. The implementation of the abstract method `Stack.Resume()` transitions `this` stack, calling `Next()` in a loop until it is empty.

Despite the fixed answer type, the immediate return types of intermediate frames may vary. We cater for this variation through an additional abstract subclass, `Cont<T>`, subclassing `Stack<A>`. Every concrete stack belongs to some continuation class, `Cont<T>`, working on a topmost `T`-computation to produce an ultimate answer of type `A`.

```
public abstract partial class Stack<A> {
 public abstract partial class Cont<T> : Stack<A> {

  public Cont<U> Push<U>(I<T>.Call<U> call){
   return new Cont<U>.Cons<T>(
          call.comp.GetEnumerator(), call, this);
  }

 }
}
```

A `T`-continuation only has a method, `Push<U>(comp)`, which pushes a `U`-computation, `comp` onto this `T`-continuation, by installing a new `Enumerator` as the stack's frame. The result is a composite `U`-continuation of the same answer type.

The actual implementations of the above abstract classes are provided by two concrete classes, `Cons<U>` and `Id`:

```
public partial class Cons<U> : Cont<T> {
 private readonly IEnumerator<I<T>> frame;
 private readonly I<U>.Call<T> call;
 private readonly Cont<U> tail;

 internal Cons(IEnumerator<I<T>> frame,
           I<U>.Call<T> instr,
           Cont<U> tail) {
  this.frame = frame;
  this.call = instr;
  this.tail = tail;
 }
 public Cont<U> Pop(T t) {
  try {
   frame.Dispose();
   call.Value = t;
   return tail;
  }
  catch (Exception e) {
   call.Throw(e);
   return tail;
  };
 }
 public override Stack<A> Next() {
  bool hasNext = false;
  try { hasNext = frame.MoveNext(); }
  catch (Exception e) {
   call.Throw(e);
   return tail;
  }
  if (!hasNext) {
   call.Value = default(T);
   return tail;
  };
  return (frame.Current == null) ? this
         : frame.Current.Step<A, U>(this);
 }
 public Cont<T> TailPush(IEnumerable<I<T>> comp) {
  // Beware! this discards pending finally clauses
  // by not omitting frame.Discard()
  return new Cons<U>(
          comp.GetEnumerator(), call, tail);
 }
}
```

A `Cons<U>` instance consists of (i) an active `frame` (an enumerator of `T`-instructions), (ii) a `I<U>.Call<T>` instruction awaiting this frame's result and (iii) the suspended `U`-continuation to resume after this frame's computation has produced a result. By construction, both `this` and `this.tail` must have the same answer type `A`, but may have different return types `T` and `U`.

Method `Pop(T t)` is used to implement `Return`. It writes `t` to the `Value` property of its waiting call instruction before returning the pending continuation (the tail of the stack). A minor subtlety is that we need to ensure any `finally` clauses enclosing the `yield return I<T>.Return(t);` instruction are executed *before* returning; this is accomplished by disposing the frame. Moreover, we need to communicate any exception that may have been thrown by running those finally clauses.

The `Next()` override method transitions the frame by one iteration. If `MoveNext()` raised an exception (thrown by this iteration), we record the exception in the waiting call instruction and return the `tail` continuation. If there is no instruction (i.e. `MoveNext()` returned `false`) this means the

frame has ended without issuing a return instruction. In this case, we just succeed with the default value and return the stack tail (this is a design choice, as we could fail instead). A `null` instruction simply returns the current stack. Finally, we `Step` the stack according to the current instruction, `frame.Current`.

Method `TailPush(comp)` is used to implement tail calls. It simply discards the current frame (and any pending `finally` clauses!) and installs a new one obtained from its argument computation. Since `TailPush` must preserve the original call instruction that the tail continuation will read from, it is clear that `TailPush`, unlike `Push`, cannot change the current continuation type `T`. In principle, given the linearity constraints already imposed by our use of iterators, we could also reuse the original `Cons` cell instead of discarding it and allocating a new one. The fact that we ignore pending `finally` clauses is unfortunate but not unreasonable. If computations were a language feature, rather than a library, one would statically reject tail calls from within `finally` clauses anyway. [6]

```
public partial class Id : Cont<A> {
 private readonly I<A>.Call<A> call;
 public Id(I<A>.Call<A> call) {
  this.call = call;
 }
 private enum State {
  Pending, Continued, Consumed
 };
 private State state = State.Pending;
 public override Stack<A> Next() {
  lock (this) {
   state = State.Continued;
   Monitor.Pulse(this);
   return null;
  }
 }
 public A Answer() {
  lock (this) {
   while (state == State.Pending) {
    Monitor.Wait(this);
   }
   switch (state) {
    case State.Continued: {
      state = State.Consumed; return call.Value;
     }
    default: throw new InvalidOperationException();
   }
  }
 }
}
```

Note that, for the identity continuation, because of its particular base class `Cont<A>`, the return type of the call and answer type of the continuation *coincide* `Id : Cont<A>` and `Cont<A> : Stack<A>`. This makes it possible to wait for the `Answer()` to the computation by waiting for the first (and last) call to `Id.Next()`, upon which the result of the embedded call instruction can be retrieved from its `Value` property.

Since execution of the stack may have migrated to another thread, `Next()`, may well be called from a different thread than the one calling `Answer()`. For this reason, we protect the private state of the `Id` continuation with the condition variable, `state`. The caller of `Answer()` will block until or unless notified by some thread finishing in `Next()`.

6.2 Execution

With our stack API in hand, we can now show how to actually execute computations and give the implementation of the `Run` and `Spawn` methods that were introduced in §4.

The implementation of `Computation.Run` just allocates an initial stack, resumes it and calls `id.Answer()`:

```
// run the computation synchronously
// may block on thread migration
public static T Run<T>(this IEnumerable<I<T>> comp)
{
 var call = new I<T>.Call<T>(comp);
 var id = new Stack<T>.Cont<T>.Id(call);
 var s = id.Push(call);
 s.Resume();
 return id.Answer(); // may block
}
```

`Computation.Spawn` just queues the same task to the `ThreadPool` and returns a delayed call to `id.Answer()`:

```
// runs the computation asynchronously
// returns a thunk to wait on.
public static Func<T> Spawn<T>(
 this IEnumerable<I<T>> comp)
{
 var call = new I<T>.Call<T>(comp);
 var id = new Stack<T>.Cont<T>.Id(call);
 var s = id.Push(call);
 ThreadPool.QueueUserWorkItem(_ => s.Resume());
 return () => id.Answer(); // may block
}
```

The APM methods, `Computation.Begin` and `Computation.End`, are not much more complex. Their implementation uses an off-the-shelf `IAsyncResult` implementation due to Richter [17]; we omit the details for lack of space.

6.3 Instructions

All concrete instructions derive from an abstract instruction class, `I<T>`. The parameter `T` determines the type of the value, `t`, returned by a `I<T>.Return(t)` instruction:

```
public abstract partial class I<T> {
 internal protected abstract Stack<A> Step<A, U>(
```

[6] Interestingly, attempting to `Dispose` the frame and propagate any exceptions forces one to weaken the return type of `TailPush` to the less informative `Stack<A>` type.

```
  Stack<A>.Cont<T>.Cons<U> stack);
}
```

The abstract `Step(stack)` method transitions a non-empty stack `Stack<A>.Cont<U>.Cons<T>` that placed a call to a `T`-computation (`stack.call`) from some continuation of type `Stack<A>.Cont<U>` (i.e. `stack.tail`). Interpreting an instruction should be independent of, and thus parametric in, both the answer type of the stack, `A`, and the return type of the caller, `U`, and its continuation.

As a first approximation, we can think of `I<T>` as an *algebraic datatype* of instructions. Concrete instructions types are subclasses of `I<T>`. For convenience, these are declared as nested classes within `I<T>`. Corresponding to the term constructors of our algebraic datatype, we find the following instruction subclasses.

6.3.1 Return

The instruction `new I<T>.Return(value)` steps a stack by popping with the return value.

```
public partial class Return : I<T> {
 private readonly T value;
 public Return(T value) {
  this.value = value;
 }
 internal protected override Stack<A> Step<A, U>(
   Stack<A>.Cont<T>.Cons<U> stack) {
  return stack.Pop(value);
 }
}
```

6.3.2 Call

The instruction `new I<T>.Call<U>(c)` steps the current stack by `Push`ing itself onto the stack.

The result of a call - a value of type `U` or an exception - is stored in, and subsequently retrieved from, the instruction's `Value` property. To detect protocol violations, we add a state variable to ensure that the `Value` is only accessed after it has been written. The state variable tracks whether the call is still `Pending` execution, has `Succeeded` (by assignment to `Value` in `Return.Step()` or implicit `Exit()` in `Cons<U>.Next()`. or has `Failed` with some exception `exception` set by `Throw()`. Unlike `Id.Answer()` (see above), the `Value` property requires no synchronization.

```
public partial class Call<U> : I<T> {
 internal readonly IEnumerable<I<U>> comp;
 public Call(IEnumerable<I<U>> comp) {
  this.comp = comp;
 }
 internal protected override Stack<A> Step<A, V>(
  Stack<A>.Cont<T>.Cons<V> stack) {
  return stack.Push(this);
 }
 private enum State {
  Pending, Succeeded, Failed, Consumed
 };
 State state = State.Pending;
 private Exception exception;
 private U value = default(U);
 public U Value {
   get {
   switch (state) {
    case State.Succeeded: {
      state = State.Consumed; return value;
     }
    case State.Failed: {
      state = State.Consumed; throw exception;
     }
    default:
     throw new System.InvalidOperationException();
   }
  }
  internal set {
   this.value = value;
   state = State.Succeeded;
  }
 }
 internal void Throw(Exception exception) {
  this.exception = exception;
  state = State.Failed;
 }
}
```

6.3.3 Tail Call

`TailCall`'s step method trivially calls `Cont<T>.TailPush` on the current stack and stored computation.

```
public partial class TailCall : I<T> {
 private readonly IEnumerable<I<T>> comp;
 public TailCall(IEnumerable<I<T>> comp) {
  this.comp = comp;
 }
 internal protected override Stack<A> Step<A, U>(
   Stack<A>.Cont<T>.Cons<U> stack) {
  return stack.TailPush(comp);
 }
}
```

6.3.4 APM calls

We only show the generic implementation for a 1-argument APM operation (for `t1` of *any* type `T1`) (other arities follow the same pattern):

```
public static partial class APM<U> {
 public delegate U EndOp(IAsyncResult r);

 public static Operation<T1> Call<T1>(
   Operation<T1>.BeginOp beginOp, T1 t1,
   EndOp endOp)
 { return new Operation<T1>(beginOp, t1, endOp); }
```

```
public partial class Operation<T1> : I<T> {

 public delegate IAsyncResult BeginOp(
  T1 t1, AsyncCallback cb, Object state);

 private readonly T1 t1;
 private readonly BeginOp beginOp;
 private readonly EndOp endOp;

 private IAsyncResult r;
 public U Value { get { return endOp(r); } }

 public Operation(BeginOp beginOp,T1 t1,
                  EndOp endOp) {
  this.beginOp = beginOp;
  this.t1 = t1;
  this.endOp = endOp;
 }
 internal protected override Stack<A> Step<A, V>(
  Stack<A>.Cont<T>.Cons<V> stack) {
  beginOp(t1,
          r => { this.r = r; stack.Resume();},
          null);
  return null;
 }
}
}
```

The `EndOp` delegate, common to all operation classes, and the nested `BeginOp` delegate, particular to this operation's arity, capture the static aspects of the .NET APM protocol.[7]

`APM<U>.Call(b,t1,e)` is just a helper method that allows us to omit specifying the C♯ inferable type parameter, `T1`, to the constructor of the instruction class proper, `Operation<T1>`.

`Operation<T1>'s` constructor saves its arguments in private fields. Stepping the operation empties the stack but first calls `beginOp` with the user's argument and a manufactured callback, initiating the APM protocol. The callback eventually writes its `IAsyncResult` argument to another private field before resuming the saved stack. The operation's `Value` property applies `EndOp` to the `IAsyncResult` field, returning a value of type `U` or throwing any exception stored in the `IAsyncResult`, completing the APM protocol.

7. Extending the instruction set

A key feature of our technique is that it is *extensible*. The instruction set is not fixed. We can add instructions simply by declaring new *subclasses* of `I<T>` (or indeed, deriving from existing instruction types). This is in contrast to techniques that extend the language; there every new extension requires a change to the language. We described earlier in §5 how to add instructions to access APM methods from computations to get asynchronous programming. In this section we give two other examples: supporting programming with futures, and reactive programming.

7.1 Futures

Futures are an established concurrency abstraction used to represent the eventual value of a concurrent computation. First-class, generic futures (with explicit waiting) are simple to code up.

Here's the basic idea. Given a computation `comp` (that returns a `U` value), the extension method `comp.Future()` creates a future value. This immediately spawns a worker thread to run `comp` in parallel and returns a *value* of type `Future<U>`. When the current (or another) computation actually needs to synchronize with the future, it yields an `I<T>.Await<U>` instruction, obtained from the future itself by the method `future.Wait<T>()`. The instruction logically waits (without blocking) until or unless the computation is done. Finally, the result of the computation can be read off the future's `Value` property (but only after synchronizing via `Wait<T>()`). Between spawning the computation, and obtaining its value, the main computation is, of course, free to perform other tasks.

For example, with futures, we can write a method that downloads two pages in parallel, returning the pair of their results:

```
static IEnumerable<I<Pair<string, string>>>
  DownloadTwoPages(String url1, String url2) {
 var req1 = WebRequest.Create(url1);
 var req2 = WebRequest.Create(url2);

 // spawn the second download as a parallel future
 var future = DownloadPage(url2).Future();

 // run the first download sequentially
 var call = new I<Pair<string, string>>.
  Call<string>(DownloadPage(url1));
 yield return call;
 var s1 = call.Value;

 // wait for the future
 yield return future.Wait<Pair<string, string>>();
 var s2 = future.Value;

 yield return new I<Pair<string, string>>.
  Return(new Pair<string, string>(s1, s2));
}
```

The code spawns a parallel download of the second web page and continues with a sequential download of the first. Both computations run asynchronously, without blocking. The outer computation will eventually resume (after yielding `future.Wait`) on either the last thread in `DownloadPage(url1)`, if it finished later, or the last thread in `DownloadPage(url2)`, if the future finished later, but it will not block.

[7] We could also have used instantiations of general purpose function spaces from the Framework, but these are more descriptive.

Our implementation uses the call to `Future()` to first allocate the future object then immediately spawn the computation using an APM call, passing a callback to signal the future when done:

```
static class Futures {
 public static Future<T> Future<T>(
  this IEnumerable<I<T>> comp) {
  var f = new Future<T>(Computation.End<T>);
  comp.Begin(f.AsyncCallback, null);
  return f;
 }
}
```

The actual `Future<U>` class is defined as follows:

```
public class Future<U> {
 public delegate U EndOp(IAsyncResult r);
 private readonly EndOp endOp;
 internal object _lock = new object();
 internal IAsyncResult r;
 internal Stack stack;
 internal Future(EndOp endOp) { this.endOp = endOp; }
 internal void AsyncCallback(IAsyncResult r) {
  Stack stack;
  lock (_lock) {
   this.r = r;
   stack = this.stack;
   // pulse only needed for blocking Step()
   Monitor.Pulse(_lock);
  }
  if (stack != null) stack.Resume();
 }
 public U Value { get { return endOp(r); }
 }
 public I<T>.Await<U> Wait<T>() {
  return new I<T>.Await<U>(this);
 }
}
```

The constructor stores an APM protocol's end operation (`endOp`) in the future, to be invoked from the getter of its `Value` property. When the future's computation has finished, its own APM callback (`Future<U>.AsyncCallback(r)`) will write its `IAsyncResult` argument to a private field of the future, checking if there is a logically waiting stack and resuming that stack if there is.

The `Wait<T>()` method returns the actual instruction with which some `T`-computation can synchronize with this future. Note that `Wait<T>()` should be generic since the future does not, in general, know which computation will wait for it.

Finally, we come to the implementation of the new `Await<U>` instruction:

```
public partial class Await<U> : I<T> {
 private Future<U> future;
 public Await(Future<U> future) {
  this.future = future;
 }
 internal protected override Stack<A> Step<A,V>(
  Stack<A>.Cont<T>.Cons<V> stack) {
  lock (future._lock) {
   if (future.r != null) return stack;
   else { future.stack = stack; return null; }
  };
 }
}
```

When stepped, the `Await<U>` instruction forces the interpreter to logically wait for its future's completion. The `Await<U>.Step(stack)` method checks if its associated future has already received an `IAsyncResult`. If not, it writes the pending stack to the future's stack field and returns the empty stack to the driver, thus logically waiting without blocking. If it does find a result, the future has completed, so `Step(stack)` just returns the current stack to continue. The future's fields are protected by a briefly-held, private lock, `_lock`, to avoid the obvious race. Its worth pointing out that the future cannot know the answer type of its stack field, which is why the field has the non-generic type `Stack` (c.f. §6.1), not `Stack<A>`. Fortunately, it doesn't matter, since all the future has to do is call `stack.Resume()` to continue its execution (and `Resume` does not depend on the answer type).

Basing our implementation of futures on the APM protocol makes it easy to turn *any* APM operation into a future, using simple wrapper methods. We omit the details.

7.2 Reactive programming

So far, our examples focused on asynchronous communication, but we can also use iterators for easy development of rich user interface interactions. Our solution is based on the `AwaitEvent` operation from F♯'s libraries. We create an instruction that waits until some specified event occurs and then resumes the computation.

7.2.1 Waiting for events in F♯

In F♯, this feature is embedded into asynchronous workflows. We can, for example, write a recursive workflow that counts the number of clicks on the window (adapted from [12, Chapter 16]):

```
let rec loop count = async {
  let! _ = Async.AwaitEvent lbl.MouseDown
  lbl.Text <- sprintf "Clicks: %d" count
  return! loop (count + 1) }

Async.StartImmediate (loop 1)
```

The workflow is started on the main user interface thread using the `StartImmediate` primitive on the last line. When it reaches the `AwaitEvent` operation, it registers a callback with the `MouseDown` event. The workflow is resumed exactly once when the event occurs for the first time. It updates the displayed counter and recursively calls itself using `return!` and

then starts waiting for the next occurrence of the event. Note that the workflow always runs on the main user interface thread, which guarantees that no occurrence of `MouseDown` can be missed while processing the previous one.

7.2.2 Using AwaitEvent

We start with an example that uses `AwaitEvent` to create an application for drawing rectangles. The application has two states. In the *Waiting* state, it waits until the user presses a button to start drawing. When that happens, it transitions to the *Drawing* state when it continually updates the rectangle and waits until the button is released. In the first state, we use `AwaitEvent` to wait for `MouseDownEvt`, which is a .NET event represented as a value:

```
IEnumerable<I<Unit>> Waiting() {
  var meWait = I<Unit>.AwaitEvent(MouseDownEvt);
  yield return meWait;
  MouseEventArgs me = meWait.Value;
  yield return new I<Unit>.TailCall(Drawing(me));
}
```

The computation waits for the first occurrence of the event and then reads the `Value` property, which contains information carried by the event. In this case, the information contains coordinates of the cursor when the event occurred. Next, we use the `TailCall` instruction to transition to the *Drawing* state. We give it the value `me`, containing the original cursor coordinates as an argument:

```
IEnumerable<I<Unit>> Drawing(MouseEventArgs src) {
  var choice = MouseMoveEvt.Or(MouseUpEvt);
  var evtWait = I<Unit>.AwaitEvent(choice);
  yield return evtWait;
  var evt = evtWait.Value;
  if (evt.Tag == ChoiceTag.First) {
    DrawRectangle(src, evt.First);
    yield return new I<Unit>.TailCall(Drawing(src));
  } else {
    StoreRetangle(src, evt.Second);
    yield return new I<Unit>.TailCall(Waiting());
  }
}
```

The code waits either for `MouseMoveEvt` or for `MouseUpEvt`, whichever occurs first. This is done using a simple `Or` combinator for events that, written functionally, has the following type:

```
IEvent<A> → IEvent<B> → IEvent<Choice<A, B>>
```

Once the `AwaitEvent` instruction produces a value, we test which of the two events occurred. In the first case, we update the displayed rectangle and continue in the *Drawing* state. When the user finishes the drawing and releases the button, we store the created rectangle and return back to the *Waiting* state.

Note that the computation, as we wrote it, never stops. It keeps waiting for the events during the entire application life-time, but because it is implemented in a non-blocking fashion, it doesn't consume any resources. We start the computation when initializing the user interface using `Waiting().Spawn()`.

7.2.3 Implementing AwaitEvent

Now we consider implementing the `AwaitEvent` operation by extending our instruction set:

```
public static AwaitEvt<U> AwaitEvent<U>(
  IEvent<U> evt) {
 return new AwaitEvt<U>(evt);
}
public partial class AwaitEvt<U> : I<T> {
 private readonly IEvent<U> evt;
 private U value;
 private bool HasCompleted = false;
 public U Value {
  get {
   if (!HasCompleted)
    throw new InvalidOperationException();
   return value;
  }
 }
 public AwaitEvt(IEvent<U> evt) {
  this.evt = evt;
 }
 protected internal override Stack<A> Step<A, V>(
  Stack<A>.Cont<T>.Cons<V> stack) {
  IDisposable cleanup = null;
  // subscribe for one event only
  cleanup = evt.Subscribe(v => {
   this.value = v;
   this.HasCompleted = true;
   cleanup.Dispose(); // unsubscribe
   stack.Resume();
  });
  return null;
 }
}
```

Method `AwaitEvt<U>` is a simple helper that aids inference of `U`. The interesting thing is the `AwaitEvt` instruction it constructs. Its interpretation suspends the computation, but uses the `Subscribe` method of the event to register a one-shot event handler. The `IDisposable` token this returns is later used to remove the subscription, once the event has fired, and just before resuming the computation.

Note that we recursively reference the `cleanup` *value* from its definition. but this is safe: the reference is only applied after it has been properly set. [8]

[8] In F♯, this can be done more elegantly using value recursion [23].

8. Re-interpreting instructions for debugging

One drawback of micro-threading libraries that manage their own stack (like ours) is that code written using them is hard to debug: the continuation of a call is hidden in a data structure, not in the host's native call stack. However, since our computations are just returning *interpreted* streams of instructions, there is no reason to adopt the same interpretation in debug mode. Our implementation exploits this to provide debuggable alternatives to the execution methods of the `Computations` class (`Run` / `Spawn` / `Begin`). The debug variants just use the C♯ stack to interpret a computation's virtual stack transitions, by calling different overloads of `Resume` and `Step`.

Implementing this feature requires the addition of a second abstract `Step` method on `I<T>`, providing that instruction's debug interpretation.

```
public abstract partial class I<T> {
 internal protected abstract IEnumerator<I<T>>
  Step<U>(IEnumerator<I<T>> frame,I<U>.Call<T> call);
}
```

The debug version of `Resume` and, for example, `Run` are almost trivial:

```
namespace Debug {
 public static partial class Computations {
  internal static void Resume<T, U>(
    IEnumerator<I<T>> frame, I<U>.Call<T> call) {
   while (frame != null) {
    try {
     if (frame.MoveNext()) {
      frame = (frame.Current == null) ?
                frame
              : frame.Current.Step(frame, call);
     }
     else {
      call.Value = default(T);
      return;
     };
    }
    catch (Exception e) {
     call.Throw(e);
     return;
    }
   }
  }
  public static T Run<T>(this
    IEnumerable<I<T>> comp) {
   var call = new I<Unit>.Call<T>(comp);
   Resume(comp.GetEnumerator(), call);
   return call.Value;
  }

  // etc.
 }
```

Now let us implement the instructions. The instruction `I<T>.Return.Step(frame, call)` returns a `null` frame, causing `Resume` to return:

```
public partial class Return : I<T> {
 internal protected override IEnumerator<I<T>>
   Step<U>(IEnumerator<I<T>> frame,
           I<U>.Call<T> call) {
  frame.Dispose();
    //run pending finally clauses
  call.Value = value;
  return null;
 }
}
```

`I<T>.Call<U>` synchronously invokes `Resume` (defined below) on the callee's enumerator and `this` before returning its input, `frame`, unchanged.

```
public partial class Call<U> : I<T> {
 internal protected override IEnumerator<I<T>>
   Step<V>(IEnumerator<I<T>> frame,
           I<V>.Call<T> call) {
  Debug.Computations.Resume(comp.GetEnumerator(),
                            this);
  return frame;
 }
}
```

In essence, all we are doing is interpreting `Return` using C♯'s `return`, and `Call` as a C♯ method call. We omit the details for the remaining instructions, which are mostly straightforward. Note that APM operations are just invoked synchronously, using the callback-free APM pattern.

9. From iterators to general monads

The ability to extend the set of instructions means that we can construct computations that contain some non-standard behaviour (such as asynchronous execution of an operation). Another common way of representing such computations is to use *monads* [25]. Petricek [11] has given an encoding of monads using iterators in C♯. The solution is, however, not entirely type-safe. It is interesting to look at this problem again using the techniques presented in this paper.

We consider an example based on the *Maybe* monad, which represents computations that may fail, returning a special value indicating the failure, or succeed producing some value as the result. The following example [11] shows a computation that calls a method `TryReadInt` (which may fail) until the user enters a number larger than 10 and then returns the entered number.

```
IEnumerator<IOption> TryReadLargeInt() {
 var n = TryReadInt().Apply<int>().AsStep();
 yield return n;
 if (n.Value > 10) {
  yield return OptionResult.Create(n.Value);
```

```
 } else {
  var m = TryReadLargeInt().Apply<int>().AsStep();
  yield return m;
  yield return OptionResult.Create(m.Value);
 }
}
```

When using monads, a computation is encoded using two monadic operations called *unit*: `T` → `M<T>` and *bind*: `M<T>` → (`T` → `M<U>`) → `M<U>` satisfying the usual monad laws. The C♯ encoding above, creates a sequence of monad-specific operations. The first two uses of `yield return` correspond to the *bind* operation and the last one represents *unit*. Note that the `Apply<T>` operation relies on a dynamic type cast, because the return type of the encoded monadic operation is simply `IEnumerable<IOption>>` and doesn't contain the type of the actual returned value.

9.1 Instructions for encoding monads

To implement monadic computations using our encoding, we can provide two instructions that represent monadic *bind* and *unit* and rewrite the previous example as follows:

```
static IEnumerable<OptionM.I<int>> TryReadLargeInt() {
 var n = new OptionM.I<int>.Call<int>(TryReadInt());
 yield return n;
 var res = n.Value;
 if (res > 10) {
  yield return
   new OptionM.I<int>.Return(OptionM.Unit(res));
 }
 else {
  yield return
   new OptionM.I<int>.TailCall(TryReadLargeInt());
 }
}
```

This version is type-safe, because it returns a sequence of `int`-returning instructions. Another benefit of our instruction-based encoding is that we can still use other non-monadic instructions for manipulating the stack. In the example above, we use `TailCall` in the false branch, instead of calling `Bind`, directly followed by `Return`.

9.2 Supporting monads in the interpreter

To encode monadic computations as a sequence of instructions, we need a slightly different variant of the instruction type and the interpreter. In our interpreter for computations, a computation returns a value only after reducing the entire stack of continuations. However, monadic computations may be delayed and return a value early, before and indeed without evaluating the entire continuation. This is simply because *bind(e,k)* does not necessarily have to apply *k* to produce its value - it could just discard *k* or compose it with something else.

For example an encoding of the *Continuation* monad would immediately return a value of type (`T` → `A`) → `A`. The result represents a delayed thunk, which starts evaluating the computation when we provide it with a *continuation* to call when the computation completes. To support such monadic computations we need to adjust our interpreter.

First, let's assume we have some arbitrary monad `Monad`, with computation type constructor `M<T>`, and with unspecified operations `Monad.Bind` and `Monad.Unit` with the appropriate types:

```
// The computation type
public abstract class M<T> {}
// The Monad operations
public class Monad {
  // the bind of the monad
  public static
    M<T> Bind<T, U>(M<U> r, Func<U, M<T>> f)
  // the unit of the monad
  public static M<T> Unit<T>(T t)
  // etc.
}
```

Then our `Stack<A>` hierarchy and implementation can remain roughly similar to the earlier one (some details omitted):

```
public abstract class Stack<A> {
 public abstract M<A> Next();
 public abstract class Cont<T> : Stack<A> {
  public Cont<U> Push<U>(Monad.I<T>.Call<U> call) {
   return new Cont<U>.Cons<T>(
          call.comp.GetEnumerator(), call, this);
  }
  public class Cons<U> : Cont<T> {
   public override M<A> Next() {
    bool hasNext = false;
    try { hasNext = frame.MoveNext(); }
    catch (Exception e) {
     call.Throw(e);
     return tail.Next();
    }
    if (!hasNext) {
     call.Value = default(T);
     return tail.Next();
    };
    return (frame.Current == null) ? this.Next()
           : frame.Current.Step(this);
   }
   public M<A> Pop(T t) {
    try {
     frame.Dispose();
     call.Value = t;
     return tail.Next();
    }
    catch (Exception e) {
     call.Throw(e);
     return tail.Next();
    };
```

```
   }
  }
  public class Id : Cont<A> {
   public Id(Monad.I<T>.Call<A> call) {
   public override M<A> Next() {
    return Monad.Unit<A>(call.Value);
   }
  }
 }
}
```

But with these notable differences:

- `Stack<A>.Next()` does not return an intermediate `Stack<A>` but produces an answer of type `M<A>` (to accommodate early returns). To do this, it must recurse in `Cons<U>.Next()`.
- On the final transition, `Id.Next()` applies `Monad.Unit` to the value of its call, turning its continuation argument of type `A` back into the monadic type, `M<A>`.
- `Pop(T t)` is a method that takes a `T`-value and returns the value of `tail.Next()`, obtained by resuming the tail. This means that `Pop` can be used as a function of type `T` $\rightarrow$ `M<A>`. Indeed, it will get passed as the continuation of the argument `Monad.Bind` in the interpretation of `Return` (see below).

Of course, monadic instructions must now be allowed to return values of type `M<A>`, not just `Stack<A>`, since we need to accommodate early exit with a monadic value.

```
public abstract partial class I<T> {
 public abstract M<A> Step<A, U>(
  Stack<A>.Cont<T>.Cons<U> stack);
}
```

`Return` instructions - which are different from the monad's *unit* - take monadic arguments, which get interpreted by a call to the monad's bind with continuation `stack.Pop` (see above). Note that `Bind` need not apply `stack.Pop` now nor, indeed, ever.

```
public partial class Return : I<T> {
 public M<T> v;
 public Return(M<T> v) {
  this.v = v;
 }
 public override M<A> Step<A, U>(
   Stack<A>.Cont<T>.Cons<U> stack) {
  return Monad.Bind<A, T>(v, t => stack.Pop(t));
 }
}
```

`Call` instructions `Push` their argument (as before) then recursively call `Next()` on the resulting stack. Note that the `Value` of a call instruction, as read by its continuation, is *not* monadic, but just `U`.

```
public partial class Call<U> : I<T> {
 public IEnumerable<I<U>> comp;
 public U Value { //... }
 public Call(IEnumerable<I<U>> c) {
   this.comp = c;
 }
 public override M<A> Step<A, V>(
     Stack<A>.Cont<T>.Cons<V> stack) {
    return stack.Push(this).Next();
 }
 public void Throw(Exception exception) { ... }

}
```

Any monad-specific operations, for example, the *update* behaviour of the state monad, or the *throw* of the exception monad (which we haven't shown) are then easily added as additional instructions producing monadic values, either directly, or derived from the current stack by calling `Next()`.

Of course, we can only use this encoding for a monad whose *bind* operation invokes its continuation at most once. Examples of such monads abound: the maybe monad, the state monad, and the exception monad to name but a few. Unfortunately, monads that fail this criterion are the full continuation monad and the list monad (since bind maps its continuation over the list). We have successfully implemented the resumption monad this way, but the resumption must be used linearly or not at all.

Since our monadic computations need to be interpreted recursively, the encoding of monads using iterators, unlike our earlier encoding of vanilla computations, suffers from C♯'s lack of support for tail-calls. Petricek [11] suggested introducing a *trampoline* [4] to avoid this problem, but this remains future work.

10. Related and future work

There is a vast body of related work. As far as we are aware, iterators first appeared in CLU [9], but they have subsequently appeared in many languages, including Java, Python and Ruby. Interestingly, there is less formal work on iterators than might be expected; an exception is an operational semantics of a variant of iterators in Cω [1].

It is well known that recursive enumerations of C♯ iterators may have quadratic running time. Jacobs et al. [7] demonstrate how to compile nested C♯ enumerators more efficiently, with constant time yield and suspend, by using a generalized implementation scheme for `IEnumerator<T>`. Their technique essentially extends the current compilation to finite-state machines to push-down automata that maintain an auxilliary stack of *nested* enumerators. Their implementation even spots and supports tail recursion. However, in the setting of nested iterators, the authors can rely on the type of element yielded by a nested iterator to be the same as that of the enclosing iterator. For nested computations, we must, instead, allow the instruction type, and thus element type,

to vary. Our computations support an extensible instruction set, while Jacobs et al. need just four constant instructions, returned by a mediating `MoveNext()` method, to control the enumerator stack.

Our typed encoding of stacks is closely related to Pitts' typing of frame stacks in his treatment of operational semantics [13], an idea that goes back to Wright and Felleisen's [26] work on evaluation contexts and Felleisen and Friedman's work on control operators and abstract machines [3]. Typed frame stacks are also a prominent feature of the Harper-Stone redefinition of Standard ML [5].

Microsoft's Concurrency Coordination Runtime (CCR) provided the original inspiration for this work. The CCR [2, 16] is a concurrency library for C♯ based on asynchronous message passing. CCR processes are lightweight tasks written in continuation-passing style. However, the pain of writing CPS code in C♯ is alleviated by the use of C♯ iterators returning CCR synchronization primitives to suspend and resume processes at explicit synchronization steps. Since all communication is asynchronous, CCR tasks written as enumerables never return, and thus, unlike our instruction streams, need not be indexed by their return type.

F♯'s [24, Chapter 9] computation expressions are inspired by Haskell's monadic syntax and facilitate writing structured computations as higher-order values. The syntax is expanded *before* type checking, (much like C♯'s LINQ syntax) allowing different interpretations of, for instance *bind*, for sequencing subcomputations, and *unit* for embedding values as trivial computations, to be employed. In particular, as we have seen, F♯'s asynchronous workflows provide a particular computation type for APM-computations. However, other computation expressions can be treated differently. For example, the compilation of F♯ sequence expressions corresponds to C♯'s more limited iterators, and are compiled to finite state-machines, presumably because these are more efficient than the nested first-class functions emitted for computation expressions.

Richter's `AsyncEnumerator` library for C♯ [18–21] uses C♯ iterators to alleviate programming against the .NET APM [17]. Richter's iterators invariably yield integers counting the number of APM operations issued since the last suspension, and requiring completion before the next resumption. Upon resuming, the user can pop the required results from a result-stack tied to the enumerator. This additional flexibility in the API allows the user to begin several APM calls in parallel yet wait for all or just a subset to complete. The only way for us to do this is to extend the instruction set. However, the protocol required to use the framework is also more involved and error prone than ours, requiring an additional `AsyncEnumerator<T>` argument to each iterator (to both access its queue of `IAsyncResults` (from nested APM calls) and to return the value of the outer iterator itself.

For example, here is how one might write the Fib example:

```
static IEnumerator<Int32>
         Fib(AsyncEnumerator<long> ak, long k) {
 if (k <= 1) {
   ak.Result = k;
 }
 else {
  var an = new AsyncEnumerator<long>();
  an.BeginExecute(Fib(an, k - 1), ak.End());
  yield return 1;
  var n = an.EndExecute(ak.DequeueAsyncResult());

  var am = new AsyncEnumerator<long>();
  am.BeginExecute(Fib(am, k - 2), ak.End());
  yield return 1;
  var m = am.EndExecute(ak.DequeueAsyncResult());
  ak.Result = m + n;
 }
}
```

There are several avenues for further work. We should like to formalize our approach, which would enable us to compare our techniques more clearly with more formal work on abstract machines. Such a formal model might help us in quantifying which monads our approach of §9 can handle. (We conjecture that we can handle only affine monads.)

We hope to explore how well our micro-threading technique integrates with other synchronization libraries. We have already linked to a .NET implementation of Reppy's parallel CML event combinators [14, 15] (through a non-blocking API), but not, as of yet, to the Joins library [22].

We are exploring generalizing C♯'s iterators to allow some of our encoding to be more succinct and avoid passing values through the heap (although this generalization is interesting in its own right!). Another interesting direction would be to consider the work of Liu et al. on interruptible iterators [10].

Finally, we'd like to consider other language implications of our work. One simple improvement would be some syntactic sugar that would eliminate some of the boilerplate associated with our techniques. It would also be interesting to see if there was some pattern-based translation scheme for code that would target iterators, much in the style of F♯'s computation expressions.

11. Conclusions

In this paper we have demonstrated how iterators can be used as the basis of a translation of standard sequential code into computations. The benefit of this is that computations can be co-operatively suspended and resumed at yield points. We can define various interpreters for computations, including asynchronous interpreters, and we can define special instructions to, for example, call APM-compliant asynchronous methods. This allows asynchronous programming that is strongly typed, free of continuation-passing code and can

be deeply nested. The effect is similar to F♯ asynchronous workflows, but we do not require any language extensions. We hope that our work gives further evidence of the rich expressive power of iterators.

References

[1] G. Bierman, E. Meijer, and W. Schulte. The essence of data access in Cω. In *Proceedings of ECOOP*, 2005.

[2] G. Chrysanthakopoulos and S. Singh. An asynchronous messaging library for C♯. In *Proceedings of SCOOL*, 2005.

[3] M. Felleisen and D. Friedman. Control operators, the SECD-machine and the λ-calculus. Technical Report 197, Computer Science Department, Indiana University, 1986.

[4] S. Ganz, D. Friedman, and M. Wand. Trampolined style. In *Proceedings of ICFP*, 1999.

[5] R. Harper and C. Stone. An Interpretation of Standard ML in Type Theory. Technical Report CMU-CS-97-147, School of Computer Science, Carnegie Mellon University, 1997.

[6] A. Hejlsberg, M. Torgersen, S. Wiltamuth, and P. Golde. *The C♯ Programming Language*. Addison-Wesley, third edition, 2009.

[7] B. Jacobs, E. Meijer, F. Piessens, and W. Schulte. Iterators revisited: Proof rules and implementation. In *Proceedings of FTjFP*, 2005.

[8] A. Kennedy and C. Russo. Generalized algebraic data types and object-oriented programming. In *Proceedings of OOPSLA*, 2005.

[9] B. Liskov. A history of CLU. In *Proceedings of HOPL*, 1993.

[10] J. Liu, A. Kimball, and A. Myers. Interruptible iterators. In *Proceedings of POPL*, 2006.

[11] T. Petricek. Encoding monadic computations in C♯ using iterators. In *Proceedings of ITAT*, 2009.

[12] T. Petricek and J. Skeet. *Real-World Functional Programming*. Manning, 2009.

[13] A. M. Pitts. Operational semantics and program equivalence. In *Applied Semantics*, pages 378–412. Springer-Verlag, 2002.

[14] J. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[15] J. Reppy, C. Russo, and Y. Xiao. Parallel concurrent ML. In *Proceedings of ICFP*, 2009.

[16] J. Richter. Concurrent affairs: Concurrency and coordination runtime. MSDN Magazine, September 2006.

[17] J. Richter. Concurrent affairs: Implementing the CLR asynchronous programming model. MSDN Magazine, March 2007.

[18] J. Richter. Concurrent affairs: Simplified APM with C♯. MSDN Magazine, November 2007.

[19] J. Richter. Concurrent affairs: Simplified APM with the AsyncEnumerator. MSDN Magazine, June 2008.

[20] J. Richter. Concurrent affairs: More AsyncEnumerator features. MSDN Magazine, August 2008.

[21] J. Richter. *Power Threading Library*. Wintellect, 2009. URL `http://www.wintellect.com/PowerThreading.aspx`.

[22] C. V. Russo. The Joins concurrency library. In *Proceedings of PADL*, 2007.

[23] D. Syme. Initializing mutually referential abstract objects: The value recursion challenge. In *Proceedings of ML Workshop*, 2005.

[24] D. Syme, A. Granicz, and A. Cisternio. *Expert F♯*. Apress, 2007.

[25] P. Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52. Springer-Verlag, 1995.

[26] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, Nov. 1994.