Encoding monadic computations using C# 2.0 iterators

Tomáš Petříček, Matemtaicko-fyzikální fakulta UK

http://tomasp.net/blog
tomas@tomasp.net

The key theme of the talk

Functional languages have interesting solutions to many real-world problems…

- Working with state, Computations that can fail, …
- Asynchronous programming [Syme et al. 2008]
- Concurrency using transactions [Harris et al. 2005]

Unfortunately, only a few companies really use functional languages in the real-world.

We show that we can express the concept that makes this possible using just C# 2.0

Agenda

Introduction

- **Motivation – two frequent problems**
- **Background – monadic computations in F#**

Encoding monadic computations in C#

- Working with `null` values
- Asynchronous programming

Conclusions

- Future work – other interesting applications

Working with ‘null’ values

We need to check for `null` after every call...

```
static Product GetProduct() {
  Console.Write("Enter ID:");
  var id = ReadLineOrNull();
  if (id != null) {
    Console.WriteLine("- got non-null id");
    var prod = Products.FirstOrDefault(p => p.ID == id);
    if (prod != null) {
      Console.WriteLine("- found product");
      return prod;
    }
  }
  return null;
}
```

Non-standard aspect of the computation

... repeated!

Asynchronous programming

Running operations, which can take a long time

- » Communication with the web, performing I/O…
- » The application should not block the thread!
  *When I click on **Xyz**, it's time for a coffee…*

Can we create new thread for each operation?

- » The thread is not doing anything most of the time!
- » **Not a good idea** - threads are expensive (.NET/Java)

The idiomatic solution is to use callbacks

- » Callback gets called when the operation completes
- » No threads are blocked in the meantime

Asynchronous programming

We specify the rest of the operation as a callback

```
static void DownloadAsync(string url) {
  var req = HttpWebRequest.Create(url);
  req.BeginGetResponse(ar => {
    var response = req.EndGetResponse(ar);
    Stream resp = response.GetResponseStream();
    byte[] buffer = new byte[8192];
    resp.BeginRead(buffer, 0, 8192, ar2 => {
      int read = resp.EndRead(ar2);
      Console.WriteLine("got first {0} bytes", read);
    }, null);
  }, null);
}
```

Non-standard aspect of the computation

… again!

This becomes really, really, **really** difficult!

» No high-level control flow constructs (e.g. `while`)

How would I like to write this?

Mark code as *nullable* or *asynchronous*…

- » Define these non-standard aspects as libraries
- » Compiler inserts non-standard behavior automatically

Nothing new in Haskell or F# [Wadler 1990]

- » Monad – defines the *non-standard behavior*
- » Abstract algebraic structure with two operations
- » Supported by Haskell/F# language syntax

How monads work in F#?

Adding non-standard behavior to existing code:

Computation builder

Non-standard operation

```
let GetProduct() = nullable {
   Console.Write("Enter ID:")
   let! id = ReadLineOrNull()
   Console.WriteLine("- got non-null id")
   let! prod = Products.FirstOrDefault(fun p -> p.ID = id)
   Console.WriteLine("- found product")
   return prod }
```

Meaning is defined by the *computation builder*

» **let!** is language syntax for using monads

Agenda

Introduction

- Motivation – two frequent problems
- Background – monadic computations in F#

Encoding monadic computations in C#

- **Working with `null` values**
- Asynchronous programming

Conclusions

- Future work – other interesting applications

How to do the same thing in C#?

`yield return` in C# 2.0 creates a “hole” in the code

- » Used for on-demand enumeration of elements
- » We can later specify what happens at that point

```
static IEnumerator<INull> GetProduct() {
  Console.Write("Enter ID:");
  var id = ReadLineOrNull().AsStep();
  yield return id;
  Console.WriteLine("- got non-null id");

  var prod = Products.FirstOrDefault
     (p => p.ID == id.Value).AsStep();
  yield return prod;
  Console.WriteLine("- found product");

  yield return NullResult.Create(prod.Value);
}
```

Specifies the non-standard aspect

Non-standard operation

...again!

What have we achieved so far?

Avoid unnecessary **repetition** of code

» Non-standard aspect is hidden in a library

No need to **nest** the operations

» Program looks like usual sequential code

```
operation
block {
    operation
    block {
        operation
    }
}
```

→

```
operation
operation
operation
```

We can use **higher-level** language constructs

» For example loops (e.g. `while`), exceptions, etc…

Agenda

Introduction

- Motivation – two frequent problems
- Background – monadic computations in F#

Encoding monadic computations in C#

- Working with `null` values
- **Asynchronous programming**

Conclusions

- Future work – other interesting applications

Asynchronous programming today

System notifies the caller when operation completes

Hand-written state machine

- Difficult to write & read
- **Example** – implements simple loop (35 lines)

Used less often than it should!

- … and applications hang

```
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();
}
```

We can do better than that!

```
var ms = new MemoryStream();
int read = -1;
while (read != 0) {
  byte[] buffer = new byte[1024];
  var count = stream.ReadAsync(buffer, 0, 1024).AsStep();
  yield return count;
  ms.Write(buffer, 0, count.Value);
  read = count.Value;
}
ms.Seek(0, SeekOrigin.Begin);
string s = new StreamReader(ms).ReadToEnd();
yield return AsyncResult.Create(s);
```

Waits for completion of the operation

Inside 'while' loop!

Why is this code sample better?

- Total **14 lines** of code – less than half of the original
- Preserves the **logic** of the algorithm
- We describe a **systematic** encoding

Agenda

Introduction

- Motivation – two frequent problems
- Background – monadic computations in F#

Encoding monadic computations in C#

- Working with `null` values
- Asynchronous programming

Conclusions

- **Future work – other interesting applications**

Future work

*Asynchronous and multi-core are **important** today!*

Asynchronous programming

» Integration with more real-world libraries

Software transactional memory (STM)

» Concurrent programming without locks

» Based on transactions from database world

Non-standard computation for STM

» Transaction log keeps track of state changes

» Implements transaction manager and scheduler

Time for questions & suggestions!

- **We can use advanced functional ideas in C# 2.0**
- **It makes asynchronous programming a lot easier**
- **There are potentially many useful applications**

Paper and supplementary code:

- http://tomasp.net/academic/monads-iterators.aspx
- Feel free to ask: tomas@tomasp.net

Backup slides

How to do the same thing in C#?

Insert non-standard behavior at specified points

- We need to fill-in the holes in the code
- C# 2.0 iterators give us a way to create those holes:

```
static IEnumerator<int> GetNumbers() {
  int i = 0;
  while (true) {
    yield return i;
    i = i + 1;
  }
}
```

- Transforms the code into a state machine
- We can run parts of the code step-by-step