Reactive programming, WinForms, .NET

Björn Dagerman

Motivation

- Troublesome for many students previous years
- Most student solutions we see are imperative
- Useful techniques when working with GUI’s, web, ...
- Well suited to be used with functional concepts like higher-order functions
- Give an introduction to WinForms and .NET

Goals

After the lecture you should know how to:

- Handle event streams
- Set up asynchronous workflows
- Structure reactive programs
- Use WinForms
- How some feeling for what to look at next (continued learning)

.NET & CLR

Microsoft F#

F# is a functional programming language for the **.NET Framework**. It combines the succinct, expressive, and compositional style of functional programming with the **runtime, libraries, interoperability, and object model of .NET**. [F# home page]

.NET & CLR

.NET provides a run-time environment called the Common Language Runtime (CLR)

Compilers and tools expose the CLR’s functionality

Allows you do develop code that targets the runtime

Such code is called *managed code*

.NET & CLR

Benefits of managed code include:

- Cross-language integration
- Cross-language exception handling
- Enhanced security
- Versioning and deployment support
- Simplified model for component interaction
- Debugging and profiling services

Using .NET libraries from F#

F# has been a first-class .NET citizen since its early days

It can access any of the standard .NET components

Any .NET language can access code developed in F#

For example, F# doesn't have its own GUI library. However, by going through .NET we can create GUI's in F#

Windows Forms

A GUI class library part of .NET

We will use WinForms to create GUI applications in L4 and (some) projects

Alternatives: using the graphical Windows Forms Designer, or programmatically writing the necessary code

Windows Forms Designer is unfortunately not supported directly by F#

Windows Forms – Windows Forms Designer

Workaround: create a project in a language that supports the designer, i.e. C# and create the GUI, then either

- Add this project to your F# solution, or
- Export as library (F# to C# or vice-versa)

Windows Forms – programmatically

Straight-forward approach. Remember to add a reference to

`System.Windows.Forms`

Opening a window using F# interactive:

```
> open System.Windows.Forms;;

> let form = new Form(Text = "Demo", Visible = true, TopMost = true);;
```

Reactive Programming

Create programs that wait for some input or event

Examples: GUI’s, Server applications

Reactive programming

Asynchronous programming describes programs and operations that once started are executed in the background and terminate at some “later time”

Dataflow programming is a way of modeling programs as a series of connections between data. The main concern is how the data moves and propagates through these connections

We combine these concepts with those of functional programming, such as higher-order functions (map, reduce, filter, …)

Reactive programming – Events

Events are a recurring idiom in .NET programming

An event is something you can listen to by registering a callback

Reactive programming – Events

Example using F# interactive:

```
> open System.Windows.Forms;;

> let form = new Form(Text = "Click Form", Visible = true, TopMost = true);;

val form : Form = System.Windows.Forms.Form, Text: Click Form

> form.Click.Add(fun evArgs -> printfn "Clicked!");;
```

Opens a window

When clicked, ”`Clicked!`” is printed to the console

`form.Click` is an event

`form.Click.Add` registers an event handler (also known as a callback)

Reactive programming – Events as First-Class Values

Events in F# (such as `form.Click`) are first-class values, meaning you can pass them around like any other value

We can use the combinators in the `Event` module to `map`, `filter`, and otherwise transform the event stream in compositional ways

Example:

```
form.MouseMove
    |> Event.filter (fun args -> args.X > 100)
    |> Event.listen (fun args -> printfn "Mouse, (X, Y) = (%A, %A)" args.X args.Y)
```

Reactive programming – Observables

Events are a F# idiom to express configurable callback structures

F# also supports a more advanced mechanism for configurable callbacks that is more compositional than events: *Observables*

Example:

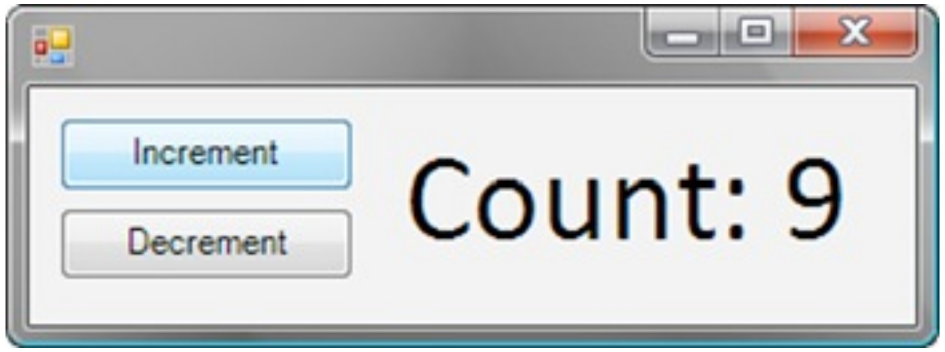
```
> open System.Windows.Forms;;

> let form = new Form(Text = "Click Form", Visible = true, TopMost = true);;

val form : Form

> form.Click |> Observable.add (fun evArgs -> printfn "Clicked!");;
```

Overview of the most important functions of the Observable module

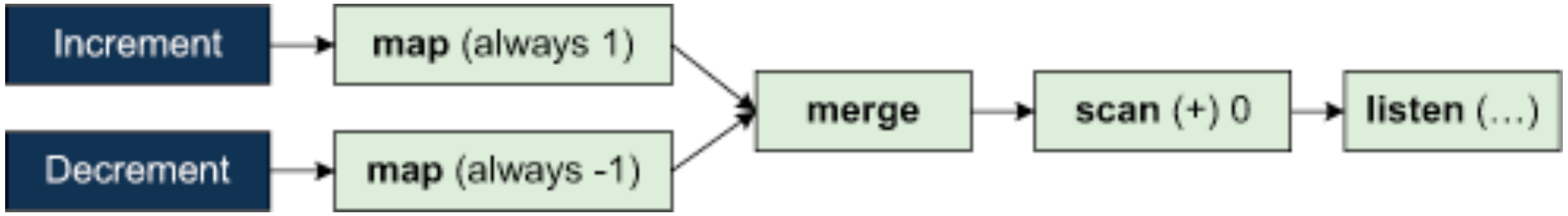
Function	Type and description
filter	`('T -> bool) -> IObservable<'T> -> IObservable<'T>` Returns an event that's triggered when the source event occurs, but only if the value carried by the event matches the specified predicate. This function corresponds to `List.filter` for lists.
map	`('T -> 'U) -> IObservable<'T> -> IObservable<'U>` Returns an event that's triggered every time the source event is triggered. The value carried by the returned event is calculated from the source value using the specific function. This corresponds to the `List.map` function.
add	`('T -> unit) -> IObservable<'T> -> unit` Registers a callback function for the specified event. The given function is called whenever the event occurs. This function is similar to `List.iter`.
scan	`('U -> 'T -> 'U) -> 'U -> IObservable<'T> -> IObservable<'U>` This function creates an event with internal state. The initial state is given as the second argument and is updated every time the source event occurs using the specified function. The returned event reports the accumulated state every time the source event is triggered, after recomputing it using the source event's value.
merge	`IObservable<'T> -> IObservable<'T> -> IObservable<'T>` Creates an event that's triggered when either of the events passed as arguments occurs. Note that the type of the values carried by the events (`'T`) has to be the same for both events.
partition	`('T -> bool) -> IObservable<'T> -> IObservable<'T> * IObservable<'T>` Splits an event into two distinct events based on the provided predicate. When the input event fires, the `partition` function runs the predicate and triggers one of the two created events depending on the result of the predicate. The behavior corresponds to `List.partition` function.

Reactive programming – Observables

Let's make a program where we change the value of a `label` by clicking on buttons:



We start by defining the Event-processing pipeline for this program:



Starting from the left, "Increment" and "Decrement" are the source events. The other boxes are events created using processing functions

The idea is that we take the click events and transform them such that they propagate integer values

Reactive programming – Observables

In code:

```
//helper function
let always x = (fun _ -> x)

//event processing code
let incEvent = btnUp.Click |> Observable.map (always 1)
let decEvent = btnDown.Click |> Observable.map (always -1)

Observable.merge incEvent decEvent
  |> Observable.scan (+) 0
  |> Observable.add (fun sum -> lbl.Text <- sprintf "Count: %d" sum)
```

`incEvent` and `decEvent` have type `IObservable<int>`. They represent events carrying integers

We merge two events creating an event that will be triggered when either button is pressed

Because the event carries integers, we can use `scan` to sum the values (starting with 0). We use `(+)` for aggregation, meaning every click will either add +1, or -1

Reactive programming – Example

Let's create a program that monitors the users download folder and unpacks any new .rar files:

```
//Monitors files in the user's Downloads folder
let fileWatcher = new FileSystemWatcher(
Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.UserProfile
), "Downloads"))

//Checks if a file is archived or not
let isArchived(fse:FileSystemEventArgs) =
   let archive = FileAttributes.Archive
   (File.GetAttributes(fse.FullPath) &&& archive) = archive

//Unpacks a file using the unrar command
let unpack(fse:FileSystemEventArgs) =
    let command = "/c unrar e " + fse.FullPath.ToString()
    System.Diagnostics.Process.Start("CMD.exe", command) |> ignore

//The programs control flow
fileWatcher.Changed                //all new files
   |> Observable.filter isArchived //ignore those that are not archived
   |> Observable.add unpack        //unpacks them
```

Asynchronous workflows

Asynchronous: something that is not synchronous,
i.e., non-blocking IO

Used to perform requests that are not completed immediately

Key observation: we don't want these requests to block our current thread!

Instead of waiting, multiple requests can be sent and results can be handled as soon as it becomes available. Example: web crawlers

Reactive programming – Asynchronous workflows

When designing applications that don't react to external events, we have many constructs that makes it easy to describe what the application does:

if-then-else expressions, for loops and while loops in imperative languages

higher-order functions and recursion in functional languages

Reactive programming – Asynchronous workflows

A typical GUI program that reacts to multiple events usually involves some mutable state. Depending on the event, this state changes somehow and more code may be run as a response

Difficult to understand all possible states and the transitions between them

Using asynchronous workflows we can write our code in such a way that the control flow becomes visible even for reactive programs

Reactive programming – Asynchronous workflows

In F# we design asynchronous workflows using the async block

```
async { some expression }
```

and the `let!` (bang) primitive

Common constructs used in async { ... } workflow expressions

Construct	Description
let! *pat* = *expr*	Execute the asynchronous computation expr and bind its result to pat when it completes. If expr has type Async<'a>, then pat has type 'a. Equivalent to async.Bind(expr,(fun pat -> ...)).
let pat = expr	Execute an expression synchronously and bind its result to pat immediately. If expr has type 'a, then pat has type 'a. Equivalent to async.Let(expr,(fun pat -> ...)).
do! *expr*	Equivalent to let! () = expr.
do *expr*	Equivalent to let () = expr.
return *expr*	Evaluate the expression, and return its value as the result of the containing asynchronous workflow. Equivalent to async.Return(expr).
return! *expr*	Execute the expression as an asynchronous computation, and return its result as the overall result of the containing asynchronous workflow. Equivalent to expr.
use *pat* = *expr*	Execute the expression immediately, and bind its result immediately. Call the Dispose method on each variable bound in the pattern when the subsequent asynchronous workflow terminates, regardless of whether it terminates normally or by an exception. Equivalent to async. Using (expr,(fun pat -> ...)).

Reactive programming – Asynchronous workflows

Example:

```
let form, label = new Form(...), new Label(...)

let rec loop(count) = async{
  let! args = Async.AwaitObservable(label.MouseDown)
  label.Text <- sprintf "Clicks: %d" count
  return! loop(count + 1) }
do
  Async.StartImmediately(loop(1))
  Application.Run(form)
```

Counting is done in a single recursive function that implements an asynchronous workflow

`AwaitObservable` wait until the first occurrence of the given event (`label.MouseDown`)

Appears to create an infinite loop. Yet, the construction is valid as it stats by waiting for the `MouseDown` event

Reactive programming – Asynchronous workflows

Example:

```
let form, label = new Form(...), new Label(...)

let rec loop(count) = async{
  let! args = Async.AwaitObservable(label.MouseDown)
  label.Text <- sprintf ”Clicks: %d” count
  return! loop(count + 1) }
do
  Async.StartImmediately(loop(1))
  Application.Run(form)
```

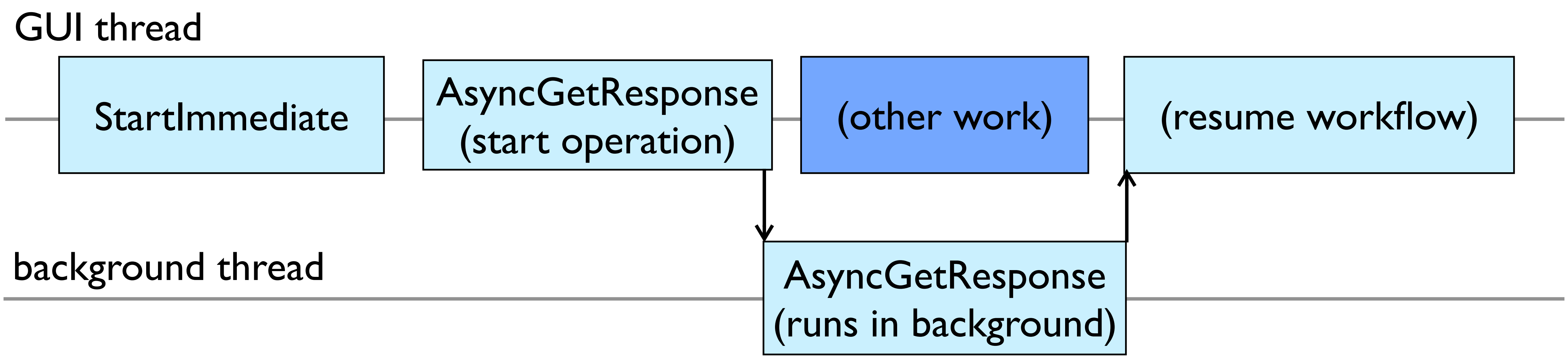
The `Async.StartImmediately` primate runs the workflow on the current thread

`Application.Run` starts the application. The current thread will be the GUI thread

WARNING: Accessing `Windows.Form` controls from outside the GUI thread causes undefined behavior!

Reactive programming – Asynchronous workflows

Let's see what happens when we use the `StartImmediate` primitive to run a workflow containing a call to some async operation



When we run an asynchronous operation (using the `let!` primitive), the GUI thread is free to perform other work

When the workflow running on a GUI thread spends most of the time waiting for completion of an asynchronous operation, the application won't become unresponsive

Summary

`AwaitObservable` waits for the first occurrence of an event

Async workflows can yield only a single value

If we want to handle multiple occurrences we can use recursion

Using recursion allows us to store the current state in the function parameters

Reading guide

***Real World Functional Programming: With Examples in F# and C#*.**
Petricek, T., & Skeet, J. (2009). Manning Publications Co.

Chapter 16 Developing reactive functional programs
Chapter 12 Sequence expressions and alternative workflows
Chapter 6 Processing values using higher-order functions
Chapter 7 Designing data-centric programs
Chapter 4 Exploring F# and .NET libraries by example

Expert F♯ 3.0

Syme, D., Granicz, A., & Cisternino, A. (2012). Berkeley: Apress.

Chapter 11 Reactive, Asynchronous, and Parallel Programming
Chapter 2.2 Using Object-Oriented Libraries in F#

Beginning F♯

Pickering, R., & De la Maza, M. (2009). Apress.

Chapter 8 User interfaces

F# for Quantitative Finance
Astborg, J. (2013). Packt Publishing Ltd.

Chapter 2.1 Structuring your F# program
Chapter 2.5 Asynchronous and parallel programming