Reactive Programming, and WinForms in F#

Björn Lisper
School of Innovation, Design, and Engineering
Mälardalen University

`bjorn.lisper@mdh.se`
`http://www.idt.mdh.se/~blr/`

Interaction

So far we’ve seen some simple interaction for F# programs:

- Return a value from an evaluated expression
- Print to the console
- Reading and writing files

But this is very limited. Real interactive systems must be able to concurrently wait for different inputs, and react to them when they arrive

Example: a GUI with different buttons and mouse, concurrently reading clicks and mouse coordinates

Reactive programs read such inputs, and react to them

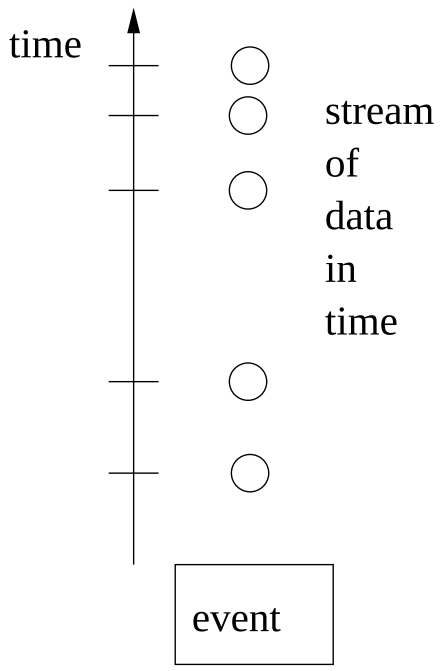
We will give an introduction how to write reactive programs in F#

Events

Inputs can be organised as **events**

An event is basically a **stream of data**: coordinates for a mouse, data representing key clicks, etc.

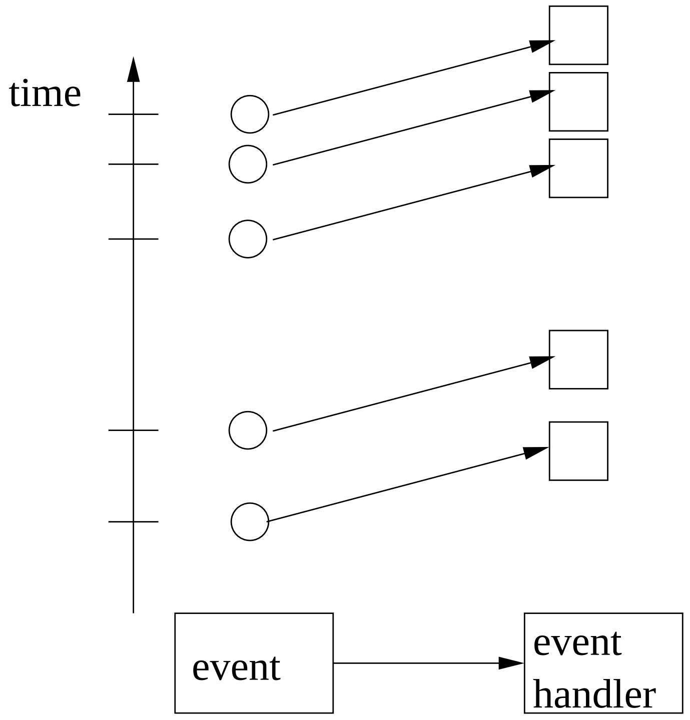
A reactive program will typically handle many events concurrently



Handling Events

An **event handler** can be connected to an event

Listens to the stream of data, and performs some action for each item in the stream



In .NET, an event handler connects to an event by *adding* itself to the event

Once added, it will receive all data in the stream and can take action accordingly

Several handlers can be added to the same event

Events in F#

F# has a data type `IEvent<'a>` for events

Events are first-class citizens just like any other data: can be moved around, copied, stored in data structures, ...

Since events are data streams, they are similar to sequences

There is an `Event` module with functions on events. Some examples:

Some Functions on Events

A selection:

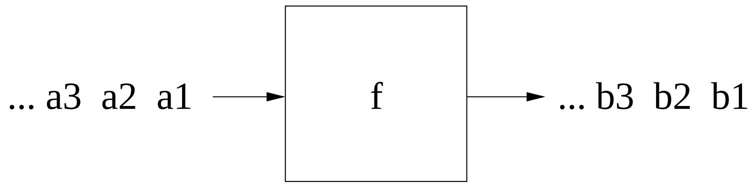
```
Event.filter : ('a -> bool) -> IEvent<'a> -> IEvent<'a>
Event.map : ('a -> 'b) -> IEvent<'a> -> IEvent<'b>
Event.merge : IEvent<'a> -> IEvent<'a> -> IEvent<'a>
Event.partition : ('a -> bool) -> IEvent<'a> ->
                  IEvent<'a> * IEvent<'a>
Event.scan : ('a -> 'b -> 'a) -> 'a -> IEvent<'b> ->
             IEvent<'a>
```

Note that some are the same as for sequences (and lists, and arrays)!

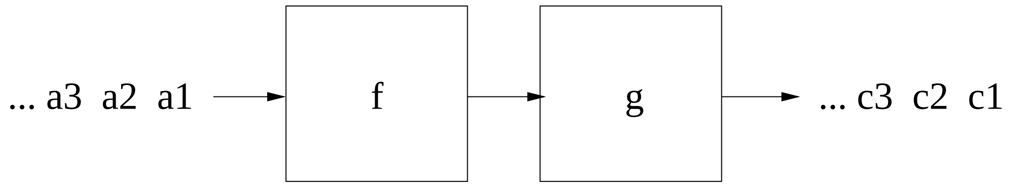
The same style of programming can be used for events!

Dataflow Style Programming

The `Event` functions support a **dataflow** style of programming, where functions operate on streams of data:



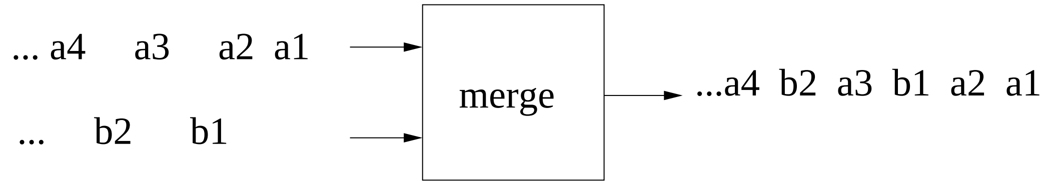
Functions can be combined to create new ones:



How do the Event Functions Work?

`Event.map`, `Event.filter`: as you would expect

`Event.merge` merges two streams into one, in the order that the elements arrive:



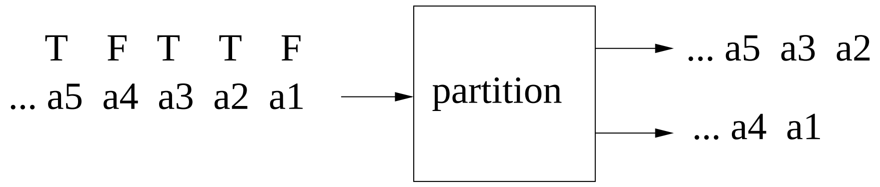
`(Event.)merge` is associative, and commutative:

```
merge e1 (merge e2 e3 ) = merge (merge e1 e2) e3
merge e1 e2 = merge e2 e1
```

`Merge` is useful for joining events, like the clicks from different buttons

Event.partition

`Event.partition` splits a stream into two, depending in the outcome of a predicate:

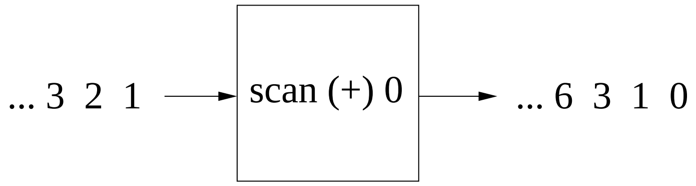


Event.scan

There is a function `List.scan` on lists. It is a version of fold that computes the list of all partial “sums” rather than just the final sum, for instance:

```
List.scan (+) 0 [1;2;3] = [0;1;3;6]
```

`Event.scan` does the same on events:



Event Handling in F#

In F#, event handlers are functions. The function is applied to each element in the stream, in order (much like `map`)

The event handler is used only for its side effect! Must have type of form
`'a -> unit`

Event handlers are added to events using the `Add` member (OO part of F#):

```
e.Add(h)
```

Adds event handler `h` to event `e`

Event Handler, Simple Example

Adding an event handler to an event `e` of type `IEvent<int>`: the handler prints the elements of the event on the console as they appear:

```
e.Add(fun x -> printf "%d\n" x)
```

Print the elements in `e` incremented by one:

```
(e |> Event.map (fun x -> x+1)).Add(fun x -> printf "%d\n" x)
```

Illustrates how the dataflow programming style is supported by the forward pipe operator

Event Handler Example Revisited

An alternative to the `Add` member:

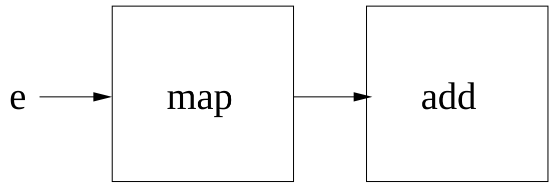
```
Event.add :  ('a -> unit) -> IEvent<'a> -> unit
```

A function that adds an event handler to an event

`e |> Event.add h` is equivalent to `e.Add(h)`

Goes well with the dataflow style, for instance:

```
e |> Event.map (fun x -> x+1) |> Event.add (fun x -> printf "%d\n" x)
```



Windows Forms

GUI’s is an important instance of reactive systems

They should respond to user inputs: clicks, mouse moves, keystrokes, taps on the screen, ...

Windows Forms is a GUI class library that is part of .NET

Module `System.Windows.Forms`

Supports event-based GUI user interaction

GUI’s using Windows Forms can be programmed in F#

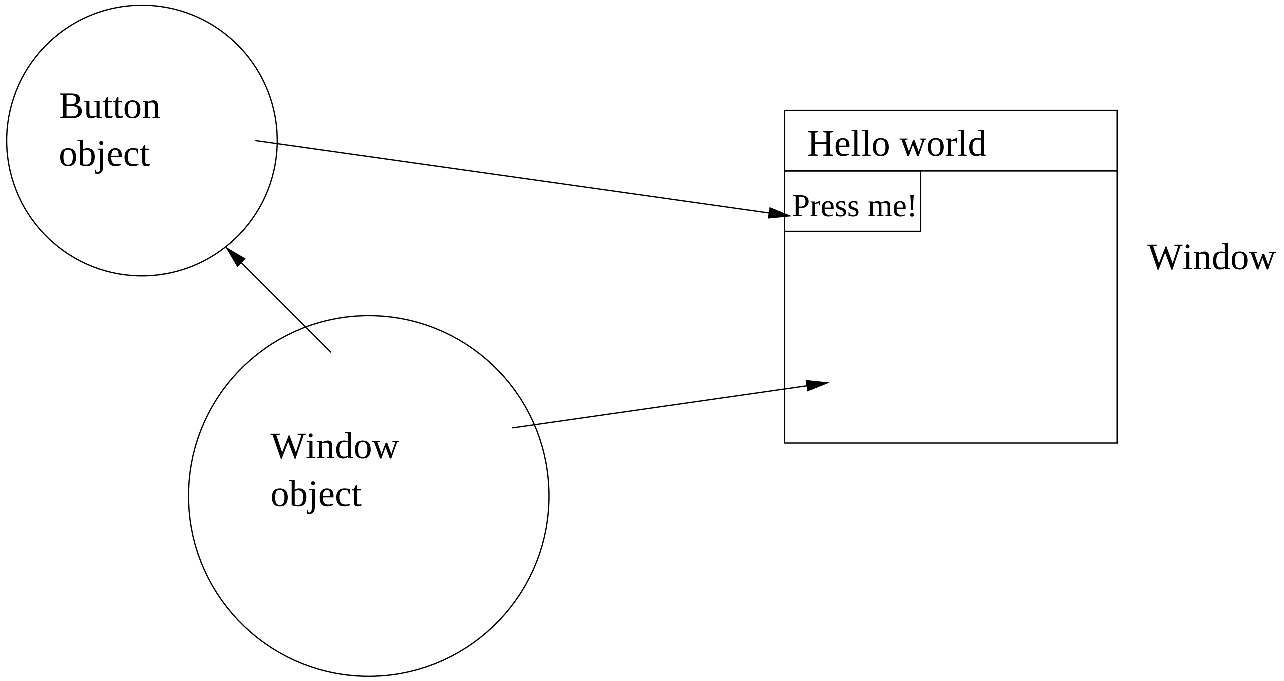
We will do some simple examples here

Windows Forms (II)

In Windows Forms, each GUI component is represented by an object:

- A window
- A button
- Etc.

The object holds the representation: position, style, fill colour, text(s), etc.


Button
object
Hello world
Press me!
Window
Window
object

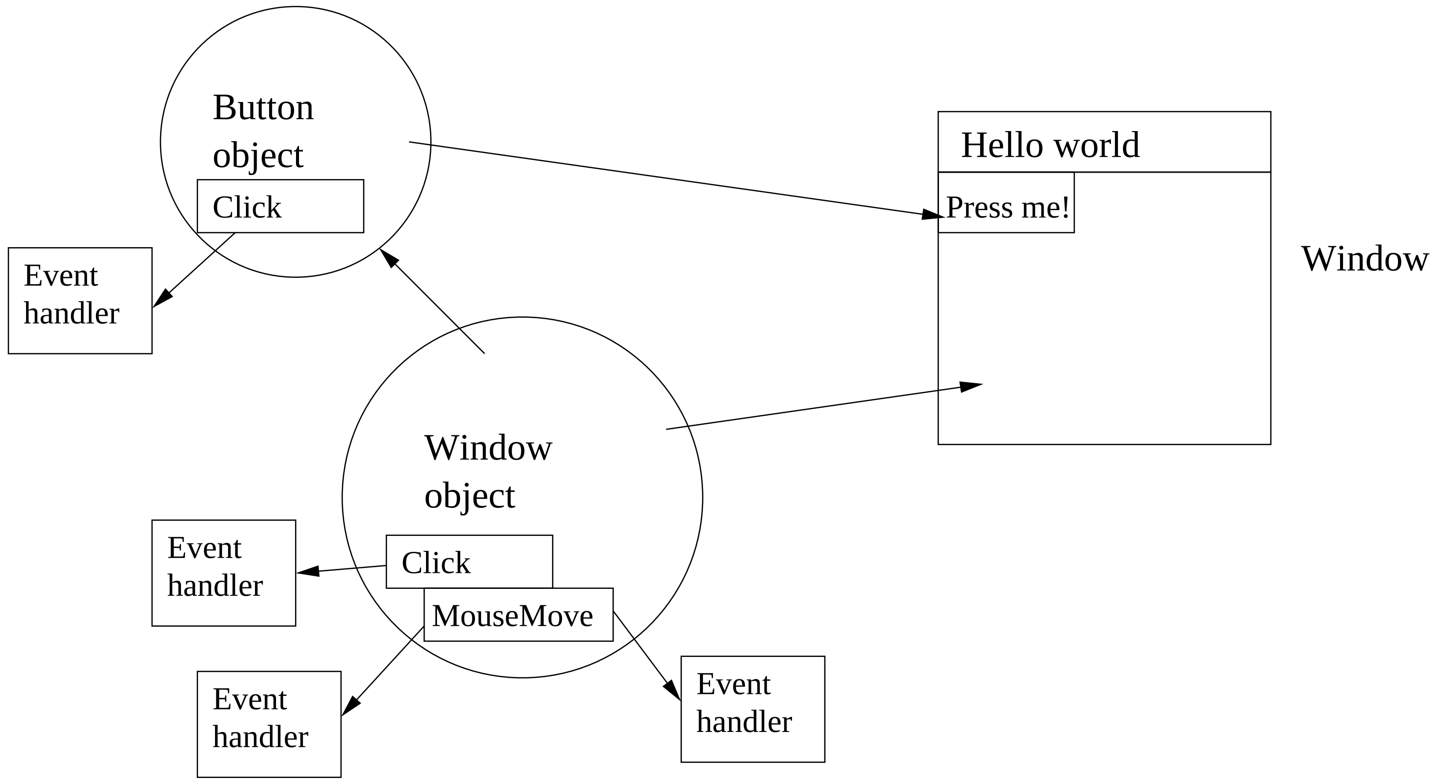
Windows Forms and Events

Each Windows Forms object holds a number of events, to which event handlers can add themselves

For instance, each button object has a “Click” event

A window object has a Click event, and a MouseMove event

Etc.


Button object
Click
Event handler
Hello world
Press me!
Window
Window object
Event handler
Click
MouseMove
Event handler
Event handler

A Simple Example

```
open System.Windows.Forms // Module for .NET GUI handling
open System.Drawing // Namespace for colors etc.

let form = new Form(Text="Hello World",Visible=true)
// Create a new window, and make it visible

let button = new Button(Text="Press here!")
// Create a new button

button.BackColor <- Color.Red
button.Size <- new Size(50,50)
button.Location <- new Point(25,25)
// make it red, size 50x50 pixels, offset (25,25) pixels

button.Click.Add(fun _ -> printfn "You pressed me!!")
// add a handler for the button’s "Click" event
```

```
form.Controls.Add(button)
// add the button to the window

form.MouseMove.Add
  (fun args -> printfn "Mouse, (x,y) = (%A,%A)" args.X args.Y)
// add a handler for the window "MouseMove" event

let e = form.MouseMove |> Event.filter (fun args -> args.X > 100)
                       |> Event.map (fun args -> args.X + args.Y)
// Define a new event, created from the MouseMove event

e.Add (fun n -> printfn "Mouse sum = %d" n)
// Add an event handler to our new event

Application.Run(form) // Finally start the execution of the window
```

Observables

The `Event` module has a problem. Sometimes the use of its functions will cause **memory leaks**

A problem if to be used in real applications

The `Observable` module provides an alternative implementation of events that does not suffer from memory leaks

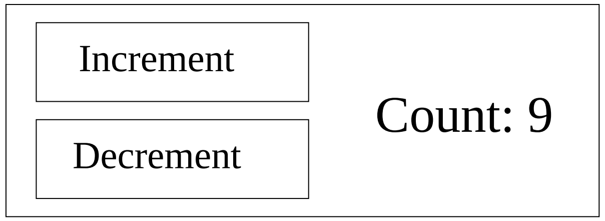
Observables are just like events, but with type `IObservable<'a>`

All `Event` functions have `Observable` counterparts

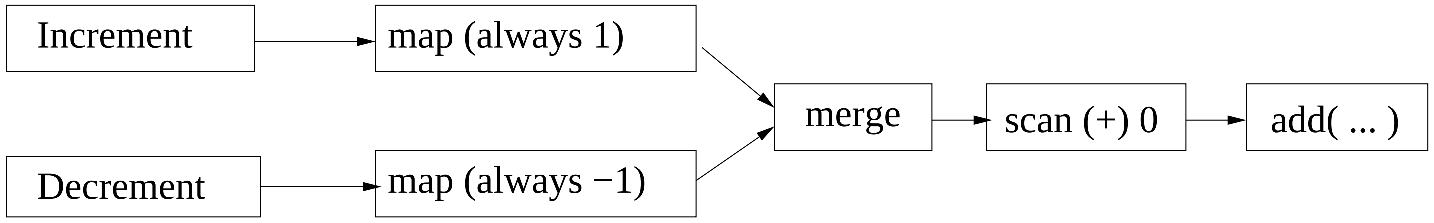
In most applications events can be replaced by observables right off

An Example with Observables

A window with a counter, and an increment (+1) and a decrement (-1) button:



A dataflow design for the solution:



Idea: turn click events into streams of 1's and -1's, sum them successively with `scan`, update the counter display for each new sum

An Example with Observables (II)

```
// Set upp buttons, label (sketch)
let btnUp   = new Button(...)
let btnDown = new Button(...)
let lbl = new Label(...)

//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)
```

A non-GUI Example

A program that monitors a user’s “Download” folder, and unpacks any new archive files

Uses an event belonging to a “`FileSystemWatcher`” object

This event is triggered each time there is some change to the contents in a specified folder

The program monitors this event, retrieves information which changes are due to new archive files, and unpacks exactly these files (ignoring other changes)

Code on next page

```
// 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

// Event handler that 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 program’s data flow
fileWatcher.Changed              // changes to Download folder
|> Observable.filter isArchived // ignore those that are not arrivals of archives
|> Observable.add unpack        // unpack archives
```

Asynchronous Workflows

The dataflow paradigm provides a nice and clean model for event processing

However, sometimes a closer control of the event processing is needed

Asynchronous workflows provide this

Gives detailed control over the event handling, including the ability to wait for the arrival of new elements in event streams without blocking other activities

This is of course crucial. An application cannot freeze while waiting for some input to appear

Async Blocks

Asynchronous workflows are created using **Async blocks**:

```
async { some expression }
```

A kind of *computation expression*

Specifies in detail when acticities will start, when to wait for inputs (and then whether to block other activities or not), etc.

Non-blocking Evaluation

Inside an async block, a special form of let declaration can be used:

```
let! var = expr
```

`expr` must be an asynchronous computation. The execution of the declaration will execute `expr` in a background thread, thus not blocking the main thread. When `expr` completes, `var` is bound to the result

Obviously a good thing to use when the evaluation of `expr` involves waiting for the next element in an event stream!

```
return! expr
```

Execute the asynchronous computation `expr`, and return the result

An Example

A simple program that counts mouse clicks one by one, using an asynchronous workflow

The elements in the event stream are processed one by one. Once an element is processed, recursion is used to process the rest of the elements

`Async.AwaitObservable(e)` is an asynchronous workflow that waits for the next element in event stream `e`, and returns it when it arrives

`Async.StartImmediate(a)` starts the asynchronous workflow `a` on the current thread

Example Code

```
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.StartImmediate(loop(1))
Application.Run(form)
```

Some Observations

Async block with `Async.AwaitObservable` vs. dataflow for events is similar to explicit recursion with head/tail vs. builtin higher-order functions for lists

In both cases a choice between explicit processing element by element, and processing using predefined function blocks

In the example, the `loop` function will recurse indefinitely

This is intentional! The program is still responsive since each step in the recursion includes interaction. Reactive programs are typically designed to run forever

In `loop`, state is maintained through the `count` argument. This provides a side effect-free way to maintain state