Reactive Programming with F#

Tomáš Petříček

Microsoft C# MVP
http://tomasp.net/blog
A little bit about me…

- Real-World Functional Programming
  - with Jon Skeet
  - Today’s talk based on some ideas from Chapter 16

- Worked on F# at MSR
  - Internships with Don Syme
  - Web programming and reactive programming in F#
  - Some Visual Studio 2010 IntelliSense
What is this talk about?

> It is not about *concurrent programming*
  > Multiple threads, various programming models
  > **Immutable data** using Tasks or Parallel LINQ
    > We have full control over the control flow
  > **Message passing** using F# `MailboxProcessor`
    > Processors react to received messages

> It is about *reactive programming*
  > Components that react to events in general
    > `MailboxProcessor` is one possible implementation
  > Can be single-threaded – running on GUI thread
Single-threaded reactive programming

> Single-threading makes GUI simple (possible!)
  > Reactive part of the application reacts quickly
  > Expensive work should be done in background

> **Declarative** – **what** to do with received data
  > Define *data-flow* using event combinators
    ⊕ Simple & elegant ⊘ Limited expressivity

> **Imperative** – **how** to react to received data
  > Define *control-flow* using asynchronous workflows
  > ⊘ Write more code ⊗ Easy for difficult tasks
Talk outline

- **Writing reactive GUIs declaratively**
  - Declarative GUI programming in WPF
  - Using F# event combinators
- **Writing reactive GUIs imperatively**
  - Using the `AwaitObservable` primitive
  - Understanding threading
- **Asynchronous programming with events**
  - Asynchronous HTTP web requests
Everybody loves declarative style!

- Used by numerous .NET libraries
  - LINQ for specifying queries in C#
  - Specifying layout of user interface in WPF/Silverlight
- Can be used for specifying reactive aspects too!

```xml
<Button Content="Click me!">
  <i:Interaction.Triggers>
    <i:EventTrigger EventName="Click">
      <ei:CallMethodAction MethodName="Process" (...) />
    </i:EventTrigger>
  </i:Interaction.Triggers>
</Button>
```
Everybody loves declarative style! (2.)

> Specifying more complex behaviors
> We can write new Triggers and Actions...
> For example *Silverlight Experimental Hacks* Library
> We can specify conditions for triggers

```xml
<Button Content="Click me!" i:Interaction.Triggers>
  <ex:EventTrigger EventName="Click">
    <ex:EventTrigger.Conditions>
      <ex:InvokingConditions>
        <ex:InvokingCondition ElementName="chkAllow"
                               Property="Enabled" Value="True" />
      </ex:InvokingConditions>
    </ex:EventTrigger.Conditions>
    <ex:PropertyAction PropertyName="Visible" Value="True" />
  </ex:EventTrigger>
</i:Interaction.Triggers></Button>
```

Triggered only when `chkAllow.Enabled == true`

Displays some control
DEMO

Introducing F# event combinators
Digression: Dynamic invoke in F#

- Access members not known at compile-time
- Simple version of `dynamic` keyword in C#
- We can easily define behavior of the operator

```fsharp
let (?) (this : Control) (prop : string) : 'T =
  this.FindName(prop) :?> 'T
```

- How does it work?
- When we write...

```fsharp
let ball : Ellipse = this?Ball
```

- ...the compiler treats it as:

```fsharp
let ball : Ellipse = (?) this "Ball"
```
More about F# events

> Events in F# are *first-class values*

> Implement interface type `IEvent<'T>`

> Events carry values `'T` such as ` MouseEventArgs`

> Can be passed as arguments, returned as results

> We use functions for working with *event values*

```fsharp
Event.map : ('T -> 'R) -> IEvent<'T> -> IEvent<'R>
Event.filter : ('T -> bool) -> IEvent<'T> -> IEvent<'T>
```

> Create new event that carries different type of value and is triggered only in some cases

> `Event.add` registers handler to the final event
Two interesting event combinators

> Merging events with `Event.merge`

\[
\text{IEvent<'T> -> IEvent<'T> -> IEvent<'T>}
\]

> Triggered whenever first or second event occurs
> Note that the carried values must have same type

> Creating stateful events with `Event.scan`

\[
('\text{St} -> 'T -> '\text{St}) -> '\text{St} -> \text{IEvent<'T> -> IEvent<'St>}
\]

> State is recalculated each time event occurs
> Triggered with new state after recalculation
Creating ColorSelector control

- Three sliders for changing color components
- Box shows current color

- Data-flow diagram describes the activity

```
red.Changed → map updateRed

green.Changed → map updateGreen

blue.Changed → map updateBlue

merge

scan applyUpdate initial
```
DEMO

Writing ColorSelector control with F# events
Accessing F# events from C#

> Events in F# are values of type \( \text{IEvent<'T>} \)
> Enables F# way of working with events
> Attribute instructs F# to generate .NET event

```fsharp
[<CLIEvent>]
member x.ColorChanged = colorChanged
```

> \( \text{IEvent<'T>} \) vs. \( \text{IObservable<'T>} \) in .NET 4.0
> You can work with both of them from F#
> Using combinators such as \( \text{Observable.map} \) etc.
> Observable keeps separate state for each handler
> Can be confusing if you add/remove handlers
Talk outline

> Writing reactive GUIs declaratively
  > Declarative GUI programming in WPF
  > Using F# event combinators

> Writing reactive GUIs imperatively
  > Using the `AwaitObservable` primitive
  > Understanding threading

> Asynchronous programming with events
  > Asynchronous HTTP web requests
Creating SemaphoreLight control

- **Typical approach** – store state as `int` or `enum`
  - Imperative code uses mutable fields
  - With event combinators, we use `Event.scan`
- Difficult to read – what does state represent?
- It is hard to see what the transitions are!

- **Better approach** – write workflow that loops between states (points in code)
  - Asynchronous waiting on events causes transitions
DEMO

Writing SemaphoreLight with workflows
Workflows for GUI programming

> **Async.AwaitObservable** operation

    AwaitObservable : IObservable<'T> -> Async<'T>

> Creates workflow that waits for the first occurrence
  > Currently not part of F# libraries / PowerPack
  > Sometimes, using **IObservable<'T>** is better
  > Works because **IEvent<'T> : IObservable<'T>**

> **Async.StartImmediate** operation

    Starts the workflow on the **current** (e.g. GUI) thread
    > Callbacks always return to original kind of thread
    > All code in the demo runs on GUI thread as required!
Writing loops using workflows

Using looping constructs like `while` and `for`

```javascript
let semaphoreStates2() = async {
  while true do
    for current in [ green; orange; red ] do
      let! md = Async.AwaitObservable(this.MouseLeftButtonDown)
      display(current) }
```

Functional style – using recursion

```javascript
let rec semaphoreStates() = async {
  for current in [ green; orange; red ] do
    let! md = Async.AwaitObservable(this.MouseLeftButtonDown)
    display(current)
  do! semaphoreStates() }
```
Break: Time for a bit of Art…
Application for drawing rectangles

Choosing between multiple transitions?

- `AwaitObservable` taking two events
- Resume when the first event fires
DEMO

Drawing rectangles in Silverlight
Waiting for multiple events

> Choosing between two (or more) events

```csharp
AwaitObservable : IObservable<'T> * IObservable<'U> -> Async<Choice<'T, 'U>>
```

> Specify two different transitions from single state

> Overloads for more events available too

```csharp
let! evt = Async.AwaitObservable
    (main.MouseLeftButtonDown, main.MouseMove)
match evt with
| Choice1Of2(up) ->
    // Left button was clicked
| Choice2Of2(move) ->
    // Mouse cursor moved
```

Overload taking two events as parameters

Returns Choice<'T1,'T2>
Talk outline

- Writing reactive GUIs declaratively
  - Declarative GUI programming in WPF
  - Using F# event combinators
- Writing reactive GUIs imperatively
  - Using the `AwaitObservable` primitive
  - Understanding threading
- Asynchronous programming with events
  - Asynchronous HTTP web requests
Patterns for asynchronous programming

- **Begin/End pattern used by standard libraries**

```csharp
let hr = HttpWebRequest.Create("http://...")
let! resp = hr.AsyncGetResponse()
let sr = resp.GetResponseStream()
```

- **Event-based pattern used more recently**

```csharp
let wc = new WebClient()
wc.DownloadStringCompleted.Add(fun res ->
    let string = res.Result )
wc.DownloadStringAsync("http://...")
```

- *Can we write this using **AwaitObservable**?*
  
  - *Little tricky* – need to attach handler *first!*
Performing asynchronous calls correctly

> Introducing `GuardedAwaitObservable` primitive

```javascript
async {
    let wc = new WebClient()
    let! res =
        Async.GuardedAwaitObservable wc.DownloadStringCompleted
            (fun () => wc.DownloadStringAsync(new Uri(uri)))
    // (...) }
```

> Calls a function after attaching event handler
> We cannot accidentally lose event occurrence

> Mixing asynchronous I/O and GUI code
> If started from GUI thread, will return to GUI thread
> We can safely access controls after HTTP request
DEMO

Social rectangle drawing application

web 2.0 inside!!
Brief summary of the talk

- Reactive code can run on the GUI thread!
- Two programming styles in F#:
  - **Declarative** or **data-flow** style
    - Using `Event.scan` combinators
  - **Imperative** or **control-flow** style
    - Using `AwaitEvent` primitive
- In both cases, we can use diagrams
- Web requests from workflows
  - Both common patterns work
Thanks!

Questions?
What do you need to run samples?

- Samples will be on my blog (below)
- Get F# and F# PowerPack ([http://www.fsharp.net](http://www.fsharp.net))
- Get Silverlight Developer tools (F# included!)
  - [http://www.silverlight.net/getstarted](http://www.silverlight.net/getstarted)

Blog & contacts

- “Real-World Functional Programming”
  - [http://functional-programming.net](http://functional-programming.net)
- My blog: [http://tomasp.net/blog](http://tomasp.net/blog)
- Contact: tomas@tomasp.net