Reactive pattern matching for F#
Part of “Variations in F#” research project

Tomáš Petříček, Charles University in Prague
http://tomasp.net/blog
tomas@tomasp.net

Don Syme, Microsoft Research Cambridge
The key theme of the talk

Languages support (overly) rich libraries for encoding concurrent and reactive programs

In practice, used in modes or design patterns such as tasks, threads, active objects, etc.

Languages can provide better support for concurrent and reactive programming

We don’t have to commit the language to one specific mode or design pattern
Agenda

Background

Asynchronous programming in F#

Reactive programming

Writing user interface control logic
Pattern matching on events
Programming with event streams

Concurrency

Pattern matching and concurrency
## Computation expressions

Compose expressions in a customized way

```pseudocode
<builder> { let! arg = function1() 
  let! res = function2(arg) 
  return res }
```

Meaning is defined by the `<builder>` object

» For example, we could propagate “null” values (aka the “maybe” monad in Haskell)

```pseudocode
let LoadFirstOrder(customerId) = 
  nullable { let! customer = LoadCustomer(customerId) 
    let! order = customer.Orders.FirstOrDefault() 
    return order } 
```
Asynchronous workflows

Writing code that doesn’t block threads

```plaintext
let http(url:string) = async {
    let req = HttpWebRequest.Create(url)
    let! rsp = req.AsyncGetResponse()
    let reader = new StreamReader(rsp.GetResponseStream())
    return! reader.AsyncReadToEnd()
}

let pages = Async.Parallel [ http(url1); http(url2) ]
```

We can use it for various design patterns

› Fork/Join parallelism involving I/O operations
› Active objects communicating via messages
Agenda

Background
Asynchronous programming in F#

Reactive programming
Writing user interface control logic
Pattern matching on events
Programming with event streams

Concurrency
Pattern matching and concurrency
Reactive programming with *async*

Concurrent *design patterns* using *async*
  » Concurrently executing, communicating agents
  » Using thread pool threads to run computations

Reactive programming design pattern
  » Uses the same language and additional libraries
  » Multiple agents running on a *single thread*
  » Agents mostly wait for events, then *react quickly*
Example: counting clicks

Show the number of left button mouse clicks

```fsharp
let rec loop(count) = async {
    let! me = Reactive.AwaitEvent(lbl.MouseDown)
    let add = if me.Button = MouseButtons.Left then 1 else 0
    lbl.Text <- sprintf "Clicks: %d" (count + add)
    return! loop(count + add)
}

loop(0) |> Async.Start
```

This looks like an “aggregation” of events

» Can we make it simpler? Yes, in this particular case...
Example: counting clicks

Modification - let's limit the “clicking rate”

```plaintext
let rec loop(count) =
    async {
        let! me = Reactive.AwaitEvent(lbl.MouseDown)
        let add = if me.Button = MouseButtons.Left then 1 else 0
        lbl.Text <- sprintf "Clicks: %d" (count + add)
        let! _ = Reactive.Sleep(1000)
        return! loop(count + add)
    }

loop(0) |> Async.Start
```

How can we describe agents in general?

» Agent is often just a simple state machine!
Agents as state machines

The elements of a state machine

» States and transitions

» In each state, some events can trigger transitions

» We can ignore all other events

We need one more thing...

» Selecting between several possible transitions
Selecting between transitions

Single-parameter `AwaitEvent` isn’t sufficient

» Select cannot be encoded in our base language

Resume when the first of the events occurs

```
let rec active(count) = async {
  match ev with
  | Choice1Of2(_, _) ->
    return! inactive()
  | Choice2Of2(_, _) ->
    printfn "count = %d" (count + 1)
    return! active(count + 1) }

and inactive() = async {
  let! me = Async.AwaitEvent(frm.MouseDown)
  return! active(0) }

Async.Start(inactive())
```
Agenda

Background
  Asynchronous programming in F#

Reactive programming
  Writing user interface control logic
  Pattern matching on events
  Programming with event streams

Concurrency
  Pattern matching and concurrency
Adding language support for joining

Let’s start with the previous version

```fsharp
let rec active(count) = async {
    let! activeAsyncAwaitEvent(frm.KeyPress, frm.MouseDown) =
        match! frm.KeyPress, frm.MouseDown with
        | Choice1Of2(_) ->
            return! inactive()
        | Choice2Of2(_) ->
            printfn "count = %d" (count + 1)
            return! active(count + 1) }
```

Computation expression specifies the semantics

» Here: Wait for the first occurrence of an event

» Pattern matching is more expressive than ‘select’
Expressive power of joins

Matching events against *commit patterns*

» Either commit ("!<pattern>") or ignore ("_")
» Important difference between "_" and "_"

Filtering – we can specify some pattern

```csharp
match! agent.StateChanged, frm.MouseDown with
| !(Completed res), _ -> printfn "Result: %A" res
| _, !me -> // Process click & continue looping
```

Joining – wait for the first occurrence of each

```csharp
match! frm.MouseDown, frm.MouseUp with
| !md, !mu ->
```
Agenda

Background
  Asynchronous programming in F#

Reactive programming
  Writing user interface control logic
  Pattern matching on events
  Programming with event streams

Concurrency
  Pattern matching and concurrency
Turning agents into event streams

Agents often perform only event transformations
  » Repeatedly yield values and may eventually end
  » “Event streams” can be elegantly composed
Turning agents into event streams

Agents often perform only event transformations
  » Repeatedly yield values and may eventually end
  » “Event streams” can be elegantly composed

Library support using computation expressions

```plaintext
let rec active(count) = eventStream {
  match! frm.Click, frm.KeyPress with
  | !ca, _ -> return! inactive()
  | _, !ka ->
    printfn "count=%d" (count + 1)
    return! active(count + 1) }

inactive().Add(fun n -> printfn "count=%d" n)
```
Agenda

Background
  Asynchronous programming in F#

Reactive programming
  Writing user interface control logic
  Pattern matching on events
  Programming with event streams

Concurrency
  Pattern matching and concurrency
Concurrency using Futures

Computation that eventually completes
  » Used for encoding task-based parallelism
  » Similar to async, but used for CPU-bound concurrency

```csharp
var f1 = Future.Create(() => {
    /* first computation */
    return result1;
});
var f2 = Future.Create(() => {
    /* second computation */
    return result2;
});
UseResults(f1.Value, f2.Value);
```

Synchronization (join) point - blocks until both complete
Pattern matching on Futures

What does “match!” mean for Futures?

“!” pattern: Wait for the computation to complete

“_” pattern: We don’t need the result to continue

Example: Multiplying all leaves of a binary tree

```
let rec treeProduct(tree) = future {
  match tree with
  | Leaf(num) -> num
  | Node(l, r) -> match! treeProduct(l), treeProduct(r) with
    | !pl, !pr -> return pl * pr
    | pl!pr!_ -> return pl * pr
}
```

» Joining of futures is a very common task

» Patterns give us additional expressivity
Concurrent using $C \omega$ joins

Simple unbounded buffer in $C \omega$

```c
public class Buffer {
    public async Put(string s);
    public string Get() & Put(string s) { return s; }
}
```

» Single synchronous method in join pattern
» The caller blocks until the method returns

Joins on channels encoded using “!” patterns:

```c
let put = new Channel<>().
let get = new Channel<ReplyChannel<>>().
joinActor { while true do
    match! put, get with
    | !v, !chnl -> chnl.Reply(v) } |> Async.Spawn
```
Time for questions & suggestions!

» Many components could be single threaded
» Direct way for encoding state machine is essential
» Language features can/should be generally useful

Thanks to:

» Don Syme, Claudio Russo, Simon Peyton Jones,
  James Margetson, Wes Dyer, Erik Meijer

For more information:

» Everything is work in progress
» Feel free to ask: tomas@tomasp.net