



# Pattern matching for monadic computations



Tomáš Petříček, Don Syme

## **Background**

Important programming models

Computation expressions & asynchronous workflows

## **Our extension**

Merging computations

Choosing between computations

Pattern matching failures

## **Other notes**

Commutative monads

# Important programming models

---

- ▶ **Join calculus,  $C\omega$**  (Fournet et al., 1996), (Benton et al., 2004)
    - ▶ Synchronous and asynchronous methods (or channels)
    - ▶ Chords (joins) as a synchronization primitive
  - ▶ **Futures** and “parallel case” in **Manticore** (Fluet et al. 2008)
    - ▶ Futures are computations (possibly) running in background
    - ▶ Manticore supports them in the language implicitly
  - ▶ **Event-driven reactive programming** (FRP and async. workflows)
    - ▶ Sequential code that includes “waiting for an event occurrence”
    - ▶ Cooperative lightweight multithreading
  - ▶ Not just reactive & concurrent (e.g. working with **lists**)
-

# Embedding of models in the language

---

- ▶ **Extending the language** (Manticore, C $\omega$ )
    - ▶ Tailored syntax for the model, but long road to the real-world
    - ▶ No predominant programming model
  - ▶ **Implementing as a library** e.g. (Russo, 2007)
    - ▶ Restricted syntax is a problem (depends on the model, but...)
  - ▶ Somewhere in the middle...
    - ▶ Identify some recurring pattern
    - ▶ **Monads** – syntax for multiple libraries (sequential)
    - ▶ **Other** – arrows, applicative functors
-

# Our solution

---

- ▶ Pattern matching for monadic computations
  - ▶ F# computation expressions – based on monads

<i>cexpr</i>	=	<b>let</b> <i>pat</i> = <i>expr</i> <b>in</b> <i>cexpr</i>	Binding value
		<b>let!</b> <i>pat</i> = <i>expr</i> <b>in</b> <i>cexp</i>	Binding computation
		<b>return</b> <i>expr</i>	Returning value
		<b>return!</b> <i>expr</i>	Returning computation
		<b>match</b> <i>expr-list</i> <b>with</b>	Pattern matching on values

- ▶ We add the missing “Pattern matching on computations”
    - ▶ Proves useful for encoding reactive and concurrent models
    - ▶ We can support **joins**, **futures** and our **reactive model**
-

# F# computation expressions overview

---

- ▶ Simple syntactic transformation
  - ▶ Similar to Haskell's do-notation (but not used for side-effects)

```
maybe { let! n = TryReadInt()  
         let! m = TryReadInt()  
         let add = n + m  
         let sub = n - m  
         return add * sub }
```

```
maybe.Bind(TryReadInt(), fun n ->  
            maybe.Bind(TryReadInt(), fun m ->  
                        let add = n + m  
                        let sub = n - m  
                        maybe.Return(add * sub) ))
```

- ▶ Maybe *builder* defines the well-known primitives:
    - ▶ **bind** :  $M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b$
    - ▶ **return** :  $a \rightarrow M\ a$
-

# Asynchronous workflows

---

- ▶ Writing code that doesn't block threads when waiting
  - ▶ Creating threads is expensive on platforms like .NET, Java

```
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
    - ▶ Lightweight cooperative multithreading
    - ▶ Active objects communicating via messages
-

## Background

- Important programming models

- Computation expressions & asynchronous workflows

## **Our extension**

- Merging computations

- Choosing between computations

- Pattern matching failures

## Other notes

- Commutative monads

# Our extension (1.): Merging computations

---

- ▶ Pattern matching is often used on tuples of values
  - ▶ How can we use it on **tuples of computations**?
  - ▶ **We need an operation**  $\oplus : M a \rightarrow M b \rightarrow M (a, b)$
  - ▶ Can we implement this using *bind*? What laws should hold?
- ▶ What functions can we use as  $\oplus$  (we call it *merge*)?

```
let numbers xs ys = list {  
  let! xys = List.zip xs ys  
  match xys with  
  | x, y -> return 10 * x + y }  
  
> numbers [1; 2; 3] [6; 5; 4];;  
val it : int list = [16; 25; 34]
```



# Our extension (1.): Merging computations

---

- ▶ Example of **match!** that needs monad with *merge*:

We ignore “!”  
in patterns  
for now...

```
let numbers xs ys = list {  
  match! xs, ys with  
  | !x, !y -> return 10 * x + y }  
  | x, v -> return 10 * x + v }  
list.Bind(list.Merge(xs, ys), fun (x, y) ->  
  list.Return(10 * x + y))
```

- ▶ Reasoning and expectations about pattern matching
  - ▶ Rearranging arguments (and patterns) preserves the meaning
  - ▶ Pattern matching on values created using *return*...

$$\begin{aligned} \text{return } (a, b) &\equiv (\text{return } a) \textcircled{\parallel} (\text{return } b) \\ u \textcircled{\parallel} (v \textcircled{\parallel} w) &\equiv \text{map assoc } ((u \textcircled{\parallel} v) \textcircled{\parallel} w) \\ u \textcircled{\parallel} v &\equiv \text{map swap } (v \textcircled{\parallel} u) \end{aligned}$$
$$\begin{aligned} \text{let assoc } ((a, b), c) &= (a, (b, c)) \\ \text{let swap } (a, b) &= (a, b) \end{aligned}$$

---

# Our extension (2.): Choosing a computation

---

- ▶ Pattern matching usually has multiple clauses
  - ▶ Select the first matching clause and run the body
  - ▶ Keep semantics of ML “match” – run just the first clause!

$$\begin{array}{l} \mathbf{match! e with} \\ | !v \rightarrow \mathbf{return v} \end{array} \equiv \begin{array}{l} \mathbf{match! e with} \\ | !v \rightarrow \mathbf{return v} \\ | !v \rightarrow \mathbf{return v} \end{array}$$

- ▶ Choosing computations in asynchronous workflows
  - ▶ Returns the result of the first workflow that finishes:

```
let demo() =  
  Async.ChooseFirst  
  [ fetchPrice primaryUrl;  
    fetchPrice secondaryUrl ]
```

```
let fetchPrice url = async {  
  let! html = asyncDownload url  
  return extractPrice html }
```



# Our extension (2.): Choosing a computation

---

- ▶ In general, we want to run the body of the first clause
  - ▶ **We need an operation** `choose : [M (M a)] -> M a`
  - ▶ Takes a list of computations that produce body to run
  - ▶ The resulting computation runs the first produced body
    - ▶ For now, we're ignoring the fact that pattern matching may fail...

```
let demo() = async {  
  match! fetchPrice primaryUrl, fetchPrice secondaryUrl with  
  | !res, _ -> return res  
  | _, !res -> return res }
```

```
let demo() =  
  let a1, a2 = (fetchPrice primaryUrl), (fetchPrice secondaryUrl)  
  async.Choose  
  [ async.Map (fun res -> async { return res } ) a1;  
    async.Map (fun res -> async { return res } ) a2 ]
```

---

## Our extension (3.): Merging and choosing

---

- ▶ Clause – one or more *computation patterns* and a body
  - ▶ “!*pat*” – Requires a value that matches *<pat>* (Binding pattern)
  - ▶ “\_” – We don’t need a value of the computation (*ignore pattern*)
- ▶ **Example:** Unbounded producer/consumer buffer using joins

```
let putInt = new Channel<int>()
let putString = new Channel<string>()
let get = new Channel<ReplyChannel<string>>()

let buffer =
  join.Choose
    [ join.Merge(get, putInt) |> map (fun (chnl, n) ->
      join { chnl.Reply("Number: " + n.ToString()) });
      join.Merge(get, putString) |> map (fun (chnl, s) ->
      join { chnl.Reply("String:" + s) }) ]
```

# Our extension (4.): Pattern matching failures

---

- ▶ When pattern matching fails, we cannot produce a body
  - ▶ **choose** :  $[M (M a)] \rightarrow M a$  (Assuming matching succeeds)
  - ▶ **choose** :  $[M (Maybe (M a))]$   $\rightarrow M a$  (Supporting failure)
  - ▶ In reality (because F# is not a lazy language...)

```
type MaybeDelayed<'a> =  
    | Success of (unit -> 'a)  
    | Failure
```

- ▶ Representing failure inside the monad?
  - ▶ **Inner one?** Body could fail at later point:

```
match! ma with  
| !a -> printf "first"; return! fail  
| !a -> printf "second"; return a }
```

- ▶ **Outer one?** Should preserve structure (we want to use *map*)
-

# Our extension (4.): Pattern matching failures

---

- ▶ Future – starts running in the background when created
  - ▶ **Example:** multiplying leafs of a binary tree

```
let rec treeProd t = future {  
  match t with  
  | Node(lt, rt) ->  
    match! treeProd lt, treeProd rt with  
    | !0, _ -> return 0  
    | _, !0 -> return 0  
    | !a, !b -> return a * b  
  | Leaf n -> return n }
```

- ▶ Short-circuiting behavior when one future returns zero
  - ▶ First clause that succeeds (for non-zero values, the last one)
-

## Background

- Important programming models

- Computation expressions & asynchronous workflows

## Our extension

- Merging computations

- Choosing between computations

- Pattern matching failures

## **Other notes**

- Commutative monads

# Commutative monads

---

- ▶ We can use the following definition of *merge*:
  - ▶ All laws of  $\textcircled{\parallel}$  follow from monad laws and commutativity

```
let  $\textcircled{\parallel}$  ma mb = m {  
  let! a = ma  
  let! b = mb  
  return a, b }  
≡  
let  $\textcircled{\parallel}$  ma mb = m {  
  let! b = mb  
  let! a = ma  
  return a, b }
```

- ▶ Nicer syntax for working with commutative monads?
  - ▶ Outside of the monad:  $(mleft + mwid/2, mtop + mhgt/2)$

```
maybe { match! mleft, mtop, mwid, mhgt with  
  | !l, !t, !w, !h ->  
    return (l + w/2), (t + h/2) }  
let! n = mhgt  
return (l + w/2, t + h/2) }
```

---

# Selected references

---

- ▶ (Fournet et al., 1996) C. Fournet, G. Gonthier. *The reflexive chemical abstract machine and the join-calculus*. In *Proceedings of POPL 1996*.
  - ▶ (Benton et al., 2004) N. Benton, L. Cardelli, and C. Fournet. *Modern concurrency abstractions for C#*. *ACM Trans. Program. Lang. Syst*, 26(5):769–804, 2004.
  - ▶ (Fluet et al., 2008) M. Fluet, M. Rainey, J. Reppy and A. Shaw. *Implicitly-threaded parallelism in Manticore*. In *Proc. ICFP 2008*
  - ▶ (Russo, 2007) C. Russo. *The Joins concurrency library*. In *PADL 2007*.
-

Thank you!

Any comments, suggestions and questions are welcome!

## For more information:

- ❑ **Article draft** (link is also in the email from Don)  
<http://tomasp.net/academic/match-bang/match-bang.pdf>  
<http://tomasp.net/academic/match-bang/match-bang.docx>
- ❑ **Contact:** [tomas@tomasp.net](mailto:tomas@tomasp.net)