Joinads

A retargetable control-flow construct for reactive, parallel and concurrent programming

Tomáš Petříček (**tomas.petricek@cl.cam.ac.uk**)
University of Cambridge, UK

Don Syme (**don.syme@microsoft.com**)
Microsoft Research, Cambridge, UK

The two key points of the talk

- **Language extension**

  We add language support for concurrent, parallel and reactive programming

- **Multi purpose**

  We do this without committing the language to one particular programming model

- **We extend F# computation expressions**

  Similar approach could be used in other functional languages (especially Haskell's do-notation)

Reactive, concurrent and parallel

- Programming with *futures*
  - Running in background and eventually gives a result
  - Language support in Manticore (Fluet et al. 2008)
- Event-based programming
  - Lightweight threads, communicating using *events*
  - Functional Reactive Programming (Elliott 2000)
- Join-calculus
  - *Joins* execute when certain *channels* contain values
  - Both languages (Conchon, Fessant 1999) and libraries (Russo 2007)

Bringing programming models to practice

- Language-based solutions
  - Language supports only one model
- Library-based encodings
  - Restricted syntax is limiting
- **Our approach:** Support a recurring pattern
  - Successfully used by monads (and arrows & idioms)
  - One syntactic extension works for many libraries

Overview

- **Background**
  - Computation expressions overview
- Our extension
  - Choosing between computations
  - Merging computations
  - What are joinads?
- Interesting relations
  - Joinads and other computation types

Computation expressions by example

Function of type **int -> Event<int>**

Computation expression creating **Event<int>**

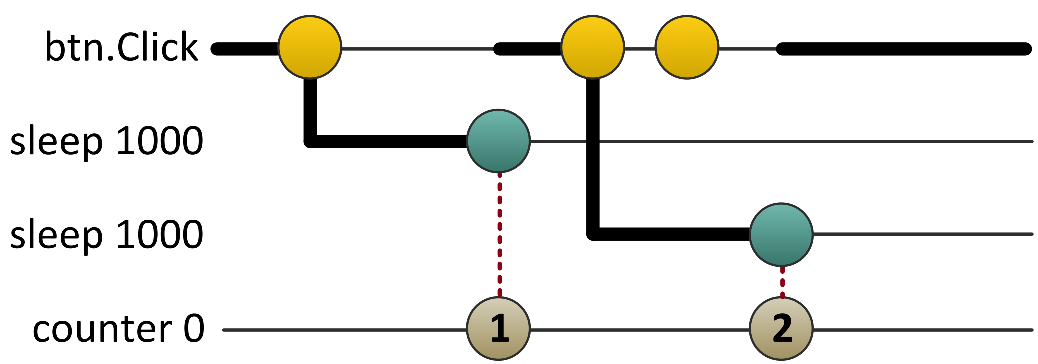
```
let rec counter n = event {
  let! args = btn.Click
  let! time = Event.sleep 1000
  return n + 1
  return! counter (n + 1) }
```

Waiting for the first occurrence of an event **Event<MouseInfo>**

Recursive looping

Trigger the created event

- Event is modeled as a sequence of time-value pairs



F# computation expressions

Computation expression syntax

cexpr =		**let** *pat* = *expr* **in** *cexpr*	Binding value
	|	**let!** *pat* = *expr* **in** *cexpr*	Binding computation
	|	**return** *expr*	Returning value
	|	**return!** *expr*	Returning computation
	|	**match** *expr-list* **with** …	Pattern matching on values

Notation for writing computations **('do' in Haskell)**

- Translates to primitive function calls

bind : M<a> → (a → M<b>) → M<b>
unit : a → M<a>
combine : M<a> → M<a> → M<a>

F# computation expressions

- Computation expression syntax

cexpr =		**let** *pat* = *expr* **in** *cexpr*	Binding value
	|	**let!** *pat* = *expr* **in** *cexpr*	Binding computation
	|	**return** *expr*	Returning value
	|	**return!** *expr*	Returning computation
	|	**match** *expr-list* **with** …	Pattern matching on values

- Our extension adds the obvious

	|	**match!** *expr-list* **with** …	Pattern matching on computations

  - …and two primitive functions for the translation
  - They specify what **match!** actually means

Overview

- Background
  - Computation expressions overview
- **Our extension**
  - Choosing between computations
  - Merging computations
  - What are joinads?
- Interesting relations
  - Joinads and other computation types

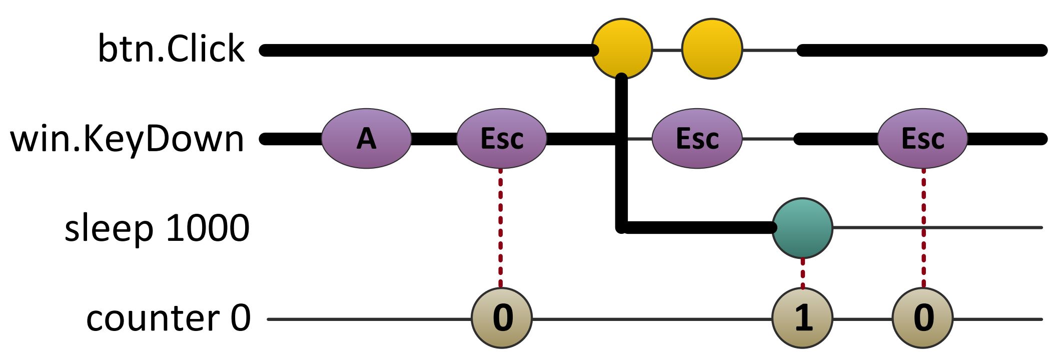
Choosing between computations

- Operation **choose** composes multiple clauses
  - Wait for events in parallel & run the first enabled body

Button was clicked
Wait & increment

Runs only when key
matches pattern

```
let rec counter n = event {
  match! btn.Click, win.KeyDown with
  | !_, _ ->   let! _ = Event.sleep 1000
               return n + 1
               return! counter (n + 1)
  | _, !Esc -> return 0
               return! counter 0 }
```



What patterns can we write?

- New syntactic category *computation pattern*

cexpr =	**match!** *expr-list* **with** *cpat-list* → *cexpr* | …		Pattern matching on computations with a list of clauses
cpat =	_		Ignore computation pattern
|	!*pat*		Bind computation using standard pattern

- Note the difference between "_" and "!_"
  - **!Esc** is a *non-exhaustive* computation binding
  - **!_** is *exhaustive* but needs a value to match on
  - _ matches even if we don't have a value

Merging computations

- Binding values from multiple computations
  - All clauses so far had only single binding pattern
  - Operation **merge** combines computations

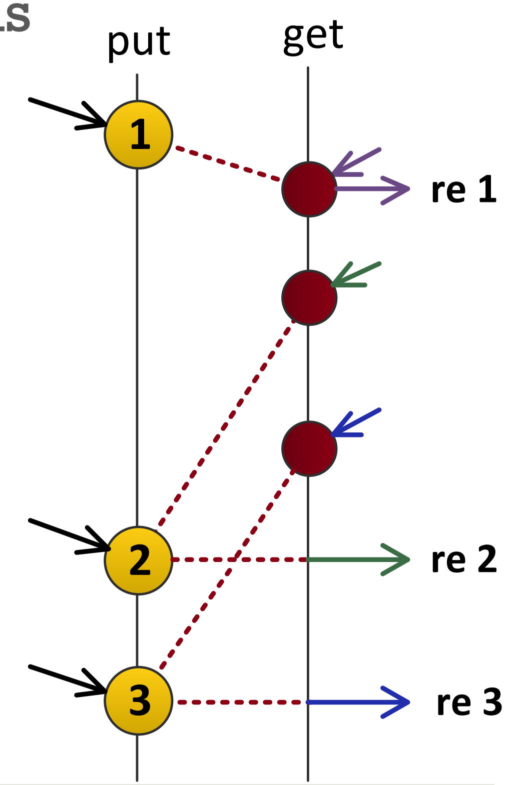
Asynchronous input & Synchronous output

```
let put = new Channel<int>()
let get = new Channel<ReplyChan<string>>()

let buffer = join {
  match! put, get with
  | !num, !chnl ->
     reply chnl (sprint "re %d" num)
```

Computation based on joins

Pattern "joining" the two channels



What is a joinad?

```
map     : (a → b) → M<a> → M<b>
merge   : M<a> → M<b> → M<a * b>
choose  : list<M<option<M<a>>>> → M<a>
```

- The **match!** syntax translates to these
  - **merge** – Combines two computations into a single
  - **choose** – Finds the first enabled computation from a list of clauses and returns computation that runs the body
- **Call to Action:** Formalization of Joinads
  - Are these the simplest primitives we can use?
  - How to find complete laws about the primitives?

Overview

- Background
  - Computation expressions overview
- Our extension
  - Choosing between computations
  - Merging computations
  - What are joinads?
- **Interesting relations**
  - Joinads and other computation types

Joinads and monads

- Joinads do not imply monads or otherwise
  - Many computations are both joinad and monad
- Can we get **merge** inside monad for free?
  - The type is $M<a> \rightarrow M<b> \rightarrow M<a * b>$
  - Want commutativity $\text{merge } u\ v \equiv \text{map swap } (\text{merge } v\ u)$

```
let merge ma mb = m {
  let! a = ma
  let! b = mb
  return a, b }
```

```
let merge ma mb = m {
  let! b = mb
  let! a = ma
  return a, b }
```

Commutative monads!

Summary & Questions?

- Language extension for multiple models
  - **Reactive** based on events (similar to FRP)
  - **Parallel** based on futures (related to Manticore)
  - **Concurrent** based on join calculus (JoCaml, Cω)
  - …and possibly many others
- Theoretically interesting
  - More work to be done on the formal model…

tomas.petricek@cl.cam.ac.uk

The end of the universe

Joinad computations for futures

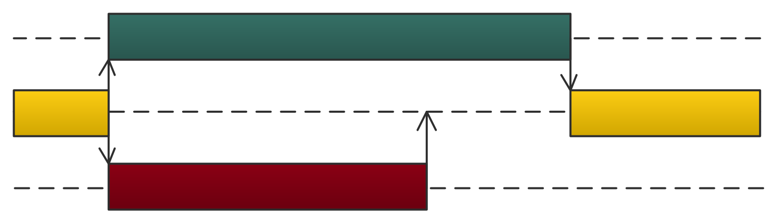
- Future is computation running in background
  - Binding means waiting for the completion

```
let multiply f1 f2 = future {
  match! f1, f2 with
   | !a, !b -> return a * b
   | !0, _  -> return 0
   | _,  !0 -> return 0 }
```

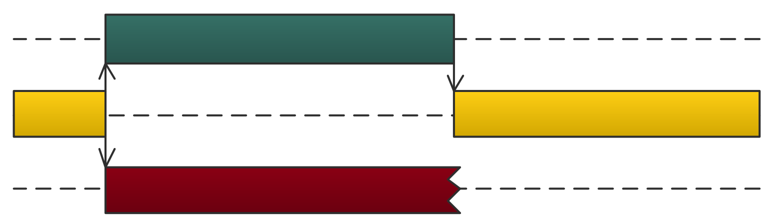
One of the futures completed and produced zero

Both futures completed

Case **!a, !b**



Case **!0, _**



Desugaring of computation expressions

```
let rec counter n = event {
  let! _ = btn.Click
  let! _ = Event.sleep 1000
  return n + 1
  return! counter (n + 1) }
```

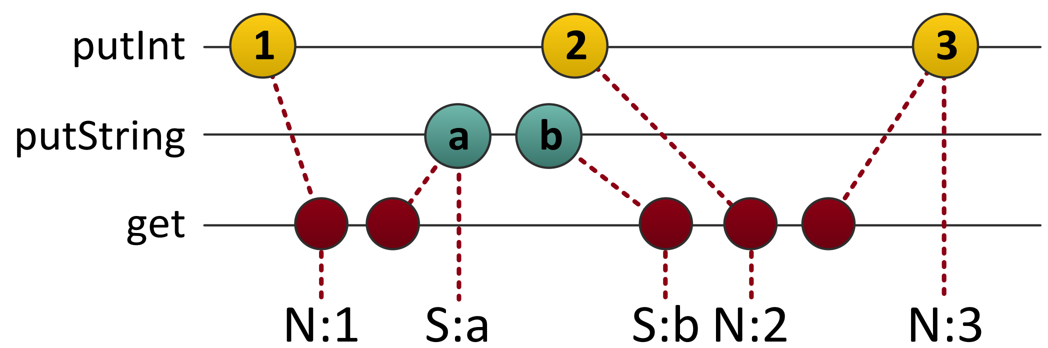
Waiting for an event

- Functions are associated with the **event** builder
  - **return** and **let!** translate to **Return** and **Bind**
  - Sequencing of expressions translates to **Combine**

```
let rec counter n =
  event.Bind(btn.Click, fun _ ->
    event.Bind(Event.sleep 1000, fun _ ->
      event.Combine
        ( event.Return(n + 1),
          counter (n + 1) )))
```

Continuation called once when event occurs

Desugaring of joinads




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

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

Pattern matching done inside **Map**

Multiple binding patterns turned into **Merge**

Choose operation explained

List of clauses

Body to run when selected

```
val choose : list<M<option<M<'a>>>> → M<'a>
```

Produces bodies of a clause

Runs the selected body

- Type signature resembles monadic **join**
  - Should behave the same for singleton list with “Some”
  - Outer computation
    - Maps matching inputs into clauses to be executed
  - Inner computation
    - Represents the body

Joinad laws: Where do they come from?

- Transformations that shouldn't change meaning

match! *m* **with** !*var* -> *expr* ≡ **let!** *var* = *m* **in** *expr*

Replace trivial "match!" with binding

match! m { **return** e_1 },
m { **return** e_2 } **with**
| !var_1, !var_2 -> *cexpr*

≡

match e_1, e_2 **with**
| var_1, var_2 -> *cexpr*

Pattern matching on two "units"

match! …, $m_{p(i)}$, … **with**
| …, $cpat_{1,\,p(i)}$, … -> $cexpr_1$
| …
| …, $cpat_{k,\,p(i)}$, … -> $cexpr_k$

≡

match! …, m_i, … **with**
| …, $cpat_{1,\,i}$, … -> $cexpr_1$
| …
| …, $cpat_{k,\,i}$, … -> $cexpr_k$

Reordering of computations & patterns

match! *m* **with**
| !var_1 -> <*cexpr*>$_1$
| !var_2 -> <*cexpr*>$_2$

≡

match! *m* **with**
| !var_1 -> <*cexpr*>$_1$

Match first enabled clause

Joinad laws: Simplified form

- Merge operation (written as ⦶)
  - Commutativity is related to commutative monads

$$
\begin{aligned}
u \oplus (v \oplus w) &\equiv \text{map assoc}\ ((u \oplus v) \oplus w) && (\textit{associativity}) \\
u \oplus v &\equiv \text{map swap}\ (v \oplus u) && (\textit{commutativity}) \\
\text{unit}\ (u, v) &\equiv (\text{unit}\ u) \oplus (\text{unit}\ v) && (\textit{unit merge})
\end{aligned}
$$

where assoc ((a, b), c) = (a, (b, c)) **and** swap (a, b) = (b, a)

- Choose operation
  - Should always select the first enabled clause (formal definition doesn't make things much simpler)
  - For monads, should generalize **bind** operation

Translation of Joinads

- **Merge** inputs for pattern matching and **map**
- Translate clauses using ⟨ – ⟩ and apply **choose**

$$\langle\!\langle\ \textbf{match!}\ expr_1, \ldots, expr_k\ \textbf{with}\ ccl_1 \mid \ldots\ ccl_p\ \rangle\!\rangle_m \equiv$$
$$\textbf{let}\ v_1 = expr_1\ \textbf{in} \ldots\ \textbf{let}\ v_k = expr_k\ \textbf{in}$$
$$\text{choose}_\text{m}\ [\ \langle\ ccl_1\ \rangle_{m,\ (v1,\ \ldots,\ vk)}; \ \ldots\ ; \ \langle\ ccl_p\ \rangle_{m,\ (v1,\ \ldots,\ vk)}\ ]$$

$$\langle\ cpat_1, \ldots, cpat_k\ \texttt{->}\ cexpr\ \rangle_{m,\ (v1,\ \ldots,\ vk)} \equiv$$
$$\text{map}_\text{m}\ (\textbf{function}\ (pat_1, \ldots), pat_n \rightarrow \text{Some}\ \langle\!\langle\ cexpr\ \rangle\!\rangle_m$$
$$\mid \_ \rightarrow \text{None})\ cargs$$

$$\textbf{where}\quad \{\ (pat_1, v_1), \ldots, (pat_n, v_n)\ \} = \{\ (pat, v_i) \mid cpat_i = !pat;\ 1 \le \text{i} \le \text{k}\}$$
$$cargs = v_1 \oplus_\text{m} \ldots \oplus_\text{m} v_{n\text{-}1} \oplus_\text{m} v_n \quad \text{for}\ \ n \ge 1$$