Macros vs Monads

Rovers in Space

Lambda Jam, Chicago, 8 July 2013

Our journey programming rovers

- Our mission: write control code for the various rovers and satellites on Jupiter's moon Io
- Because we're, you know, rocket scientists

Focusing on details: A small part of our vast rover code base

- Shut batteries down when too cold.
- Send a notification when batteries are being shutdown.
- Also send weather reports even when it isn't too cold.

Let's get this done...

```clojure
(defn update-rover
  [rover {:keys [temp] :as forecast}]

  (when (< temp -35.3)
    (shutdown (:battery rover))
    (send-message {:to :nasa
                   :body "temp too low"}))
  (send-message {:to :nasa
                 :body forecast}))
```

Two weeks of debugging later...

- Had to stub out `shutdown`, `send-messages`, etc.
- Difficult to test specific interaction scenarios
- We *know* mutable state is hard... it's time to go pure functional

Benefits over mutable version

- No stubbing or mocking
- Don't have to suspend real threads
- Testing at the REPL
- Automated testing
- Fast simulated timeouts
- Entire relevant state visible

Pure Functions Need Impure Infrastructure

- Infrastructure examines return value, decides what to do

- Real-world infrastructure
  - Examines rover object, makes real things happen

- Testing infrastructure
  - Holds entire world in an immutable collection
  - Steps world from one state to the next

Two weeks of rewriting later...

```
(defn update-rover
  [{:keys [outbox battery] :as rover}
   {:keys [temp] :as forecast}]
  (let [rover
          (if (< temp -35.3)
            (assoc rover
             :battery (shutdown battery)
             :outbox
               (conj outbox
                 {:to :nasa
                  :body "temp too low"}))
        rover)]
    (update-in rover [:outbox] conj
             {:to :nasa :body forecast})))
```

argument

local

argument

argument

local

confusing order

rover 5 times

battery 3 times

outbox 4 times

over 50% more LOC

Should we try the State Monad?

- Designed to handle state in a pure functional way.

- Separates computation into two phases
  - Building up a monadic value of functions
  - Applying those functions to the state

State Monad Rover

```
(defn update-rover
  [{:keys [temp] :as forecast}]
  (m/seq
    [(if (< temp -35.3)
       (m/seq [(m/update-val
                 :outbox conj
                 {:to :nasa
                  :body "temp too low"})
               (m/update-val :battery shutdown)])
       (m/update-state identity))
     (m/update-val :outbox conj
                   {:to :nasa
                    :body forecast})]))
```

natural order

rover 0 times

battery 1 times

outbox 2 times

Benefits of the State Monad

- More concise
- Less visible plumbing
- Less naming of locals
- Less local state
- More natural flow
- More focus

Two weeks of monad videos later...

- Do we need the two run-time phases of the state monad?
- Where did my Clojure forms go?
- Monads are more awkward in a language with little or no static type inference.

Introducing Synthread

```clojure
(defn update-rover
  [rover {:keys [temp] :as forecast}]
  (-> rover
    (->/when (< temp -35.3)
      (->/assoc :battery :shutdown
                :outbox
                  (conj {:to :nasa
                         :body "temp too low"})))
    (->/assoc :outbox (conj {:to :nasa
                             :body forecast}))))
```

natural order

rover 2 times

battery 1 times

outbox 2 times

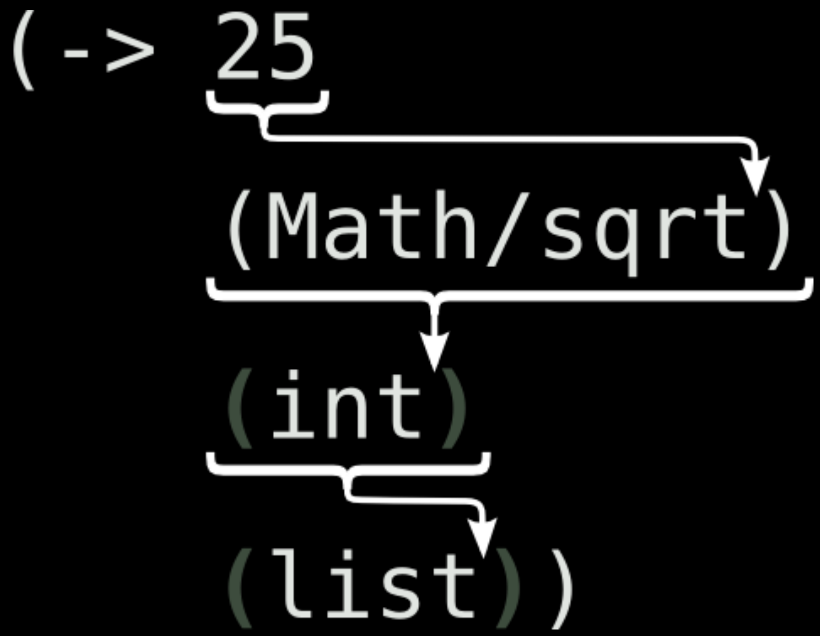
What is Synthread?

Just a library of macros aliased to the unusual name of **`->`**

```
(require '[lonocloud.synthread :as ->])
```

The macros explore using **`->`** instead of **`do`** in Clojure's standard forms.

Quick Review: -> macro



```
(-> 25
    (Math/sqrt)
    (int)
    (list))
```

Transforms a list of forms so that each form becomes the first argument to the following form.

Synthread Basics: -> as do

```clojure
(def topic (atom {}))

(do
  (swap! topic f1)

  (swap! topic f2)

  (swap! topic f3))
```

```clojure
(require '[synthread :as ->])

(->/do {} ;; topic
  f1

  f2

  f3)
```

Synthread macro groups

- Control Flow macros
- Updater macros
- Naming macros

Synthread Control Flow

Synthread defines the following control flow macros:

```
->/if
->/if-let
->/when
->/when-not
->/for
->/cond
```

Synthread Control Flow: ->/when

```
(def topic (atom {}))

(do
  (swap! topic f1)

  (when (odd? 1)
    (swap! topic f2)
    (swap! topic f3))

  (swap! topic f4))
```

```
(require '[synthread :as ->])

(-> {} ;; topic
  f1

  (->/when (odd? 1)
    f2
    f3)

  f4)
```

Also threaded

Synthread Control Flow: ->/if

```
(def topic (atom {}))

(do
  (swap! topic f1)

  (if (odd? 1)
    (swap! topic f2)
    (do
      (swap! topic f3)
      (swap! topic f4)))

  (swap! topic f5))
```

```
(require '[synthread :as ->])

(-> {} ;; topic
  f1

  (->/if (odd? 1)
    f2
    (->
      f3
      f4))

  f5)
```

Synthread Control Flow: ->/for

```clojure
(def topic (atom {}))

(do
  (swap! topic f1)

  (doseq [i (range 3)]
    (swap! topic f2 i)
    (swap! topic f3)))

  (swap! topic f4))
```

```clojure
(require '[synthread :as ->])

(-> {} ;; topic
  f1

  (->/for [i (range 3)]
    (f2 i)
    f3)

  f4)
```

Synthread macro groups

- ~~Control Flow macros~~
- Updater macros
- Naming macros

Synthread Updaters

Synthread defines following updater macros

```
->/assoc
->/in

->/each
->/each-as

->/first
->/second
->/last
->/rest
->/nth
```

Synthread Updater: ->/first

```
(require '[lonocloud.synthread :as ->])

(-> [0, 1, 2] ;; topic

  (->/first
    inc
    (* 2)))

;=> [2, 1, 2]
```

Synthread Updater: ->/in

```
(require '[lonocloud.synthread :as ->])

(-> {:a 1, :b {:sub-b 2}} ;; topic

  (->/in [:b :sub-b]
    inc
    (* 2)))

;=> {:a 1, :b {:sub-b 6}}
```

Synthread Updater: ->/assoc

```
(require '[lonocloud.synthread :as ->])

(-> {:a 1} ;; topic

  (assoc
    :b 2
    :c 3)
  (->/assoc
    :a inc
    :b (-> inc -)
    :c dec))

;=> {:a 2, :b -3, :c 2}
```

Synthread Updater: ->/each

```
(require '[lonocloud.synthread :as ->])

(-> {:a 1 :b 2 :c [1 2 3]}

  (->/in [:c]

    (->/each
      inc
      str)))

;=> {:a 1, :b 2, :c ["2", "3", "4"]}
```

Synthread macro groups

- ~~Control Flow macros~~
- ~~Updater macros~~
- Naming macros

Synthread Naming

`->/let`: name temporary values.

`->/as`: naming the topic's current value.

`->/aside`: debugging and side effects.

Synthread Rover

```
(defn update-rover
  [rover {:keys [temp] :as forecast}]
  (-> rover
    (->/when (< temp -35.3)
      (->/assoc :battery shutdown
                :outbox
                  (conj {:to :nasa
                         :body "temp too low"})))
    (->/assoc :outbox (conj {:to :nasa
                             :body forecast}))))
```

Benefits of ~~the State Monad~~ synthread

- More concise
- Less visible plumbing
- Less naming of locals
- Less local state
- More natural flow
- More focus

Synthread over the State Monad

- Names are direct analogies
- Just syntax sugar
- Less infectious
- Runtime environment identical to pure functional version

So macros beat monads?

- Of course not
- Monads are deeply established
    - terminology, conventions, multiple libraries
    - mathematical foundations
    - similarity across disparate languages
- Monads are more powerful
- Composable with other monads

Plumbing comparison

	order	rover	battery	outbox	LOC
Mutating	good	2	1	2	8
Functional	bad	5	3	4	14
State monad	good	0	1	2	12
Synthread	good	2	1	2	10

In conclusion...

- Mutable state is hard to work with, so...
- Use pure functions
- Synthread and the state monad help you write pure functions
- Monads provide maximum flexibility
- Synthread provides easy to use macros

Going where no monad has gone before...

What if the value we threaded through the Synthread macros was itself a monadic value?

Time for Questions!

Macros vs. Monads

Chris Houser

Jonathan Claggett

http://github.com/LonoCloud/synthread

LONOCLOUD

LAMBDA JAM
A Conference for Functional Programmers
Chicago, July 8-10, 2013