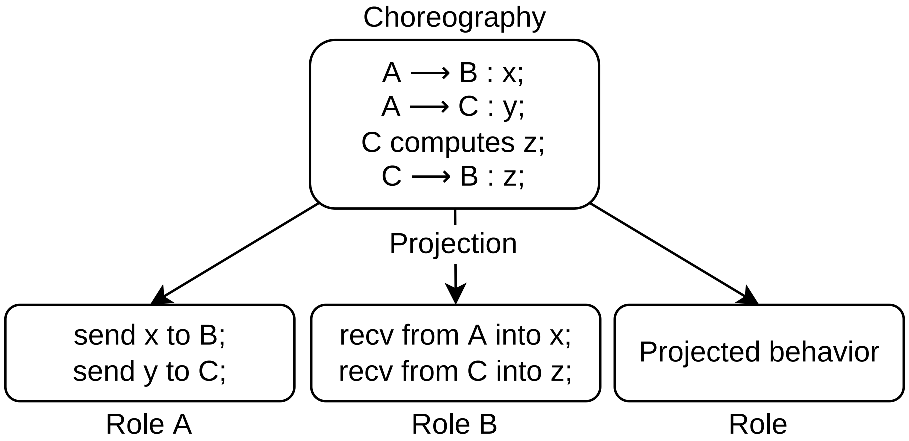
# Klor: Choreographies for the Working Clojurian

Lovro Lugović
lugovic@imada.sdu.dk
University of Southern Denmark

Sung-Shik Jongmans
ssj@ou.nl
Open University of the Netherlands

## Clojure

## Clojure + Choreographies



## Clojure + Choreographies = Klor

domain-specific language
for choreographic programming
embedded in Clojure

## Why Lisp & Clojure?

1. Lisp
2. JVM
3. concurrency
4. tooling

## Why Lisp & Clojure?

1. Lisp – metaprogramming, interactivity, functional programming
2. JVM
3. concurrency
4. tooling

## Why Lisp & Clojure?

1. Lisp – metaprogramming, interactivity, functional programming
2. JVM – ecosystem, interoperability
3. concurrency
4. tooling

## Why Lisp & Clojure?

1. Lisp – metaprogramming, interactivity, functional programming
2. JVM – ecosystem, interoperability
3. concurrency – immutability, persistent data structures
4. tooling

## Why Lisp & Clojure?

1. Lisp – metaprogramming, interactivity, functional programming
2. JVM – ecosystem, interoperability
3. concurrency – immutability, persistent data structures
4. tooling – editors, REPLs, visualizers, debuggers, build tools

## Clojure for the Uninitiated

```
(1 2)   [\a \b]   #{:a :b}   {:a 1 :b 2}
```

## Clojure for the Uninitiated

```
(1 2)   [\a \b]   #{:a :b}   {:a 1 :b 2}

(defn fact [n]
  (if (= n 0) 1 (* n (fact (dec n)))))
```

## Clojure for the Uninitiated

```
(1 2)   [\a \b]   #{:a :b}   {:a 1 :b 2}

(defn fact [n]
  (if (= n 0) 1 (* n (fact (dec n)))))

(filter (fn [n] (= (mod n 5) 0)) (range))
```

## Clojure for the Uninitiated

```
(1 2)   [\a \b]   #{:a :b}   {:a 1 :b 2}

(defn fact [n]
  (if (= n 0) 1 (* n (fact (dec n)))))

(filter (fn [n] (= (mod n 5) 0)) (range))

(.getDayOfMonth (java.time.LocalDate/now))
```

## Kloreographic Code

(`defchor` *name* [*role*$^+$] *type* [*param*$^*$] *expr*$^*$)

## Kloreographic Code

(`defchor` *name* [*role*$^+$] *type* [*param*$^*$] *expr*$^*$)

*type* ::= #{*role*$^+$} | [*type*$^+$] | (`->` *type*$^+$)

## Kloreographic Code

```
(defchor name [role⁺] type [param*] expr*)

type ::= #{role⁺} | [type⁺] | (-> type⁺)

expr ::= ...
       | (copy [role role] expr)
       | (narrow [role⁺] expr)
       | (lifting [role⁺] expr*)
       | (pack expr⁺)
       | (unpack [⟨pat expr⟩*] expr⁺)
       | (agree! expr⁺)
```

## A Taste of Klor

```
(defchor simple-1 [A B] (-> B) []
  (A (println "Hello!"))
  (B 123))
```

## A Taste of Klor

```
(defchor simple-1 [A B] (-> B) []
  (A (println "Hello!"))
  (B 123))

(defchor simple-2 [A B] (-> A B) [x]
  (A->B x))
```

## A Taste of Klor

```
(defchor simple-1 [A B] (-> B) []
  (A (println "Hello!"))
  (B 123))

(defchor simple-2 [A B] (-> A B) [x]
  (A->B x))

(defchor simple-3 [A B] (-> A #{A B}) [x]
  (A=>B x))
```

## Sharing Knowledge

```
(defchor maybe-inc [A B] (-> A B) [x]
  (if (A=>B (A (flip-coin)))
    (B (inc (A->B x)))
    (B (println "Nothing!"))))
```

## Sharing Knowledge

```
(defchor maybe-inc [A B] (-> A B) [x]
  (if (copy [A B] (A (flip-coin)))
    (B (inc (A->B x)))
    (B (println "Nothing!"))))
```

## Sharing Knowledge

```
(defchor maybe-inc [A B] (-> A B) [x]
  (if (copy [A B] (A (flip-coin)))
    (B (inc (A->B x)))
    (B (println "Nothing!"))))

(defchor solo-inc [A B] (-> #{A B} B) [x]
  (B (inc (narrow [B] x))))
```

## Sharing Knowledge

```
(defchor maybe-inc [A B] (-> A B) [x]
  (if (copy [A B] (A (flip-coin)))
    (B (inc (A->B x)))
    (B (println "Nothing!"))))

(defchor solo-inc [A B] (-> #{A B} B) [x]
  (B (inc x)))
```

## Sharing Knowledge

```
(defchor maybe-inc [A B] (-> A B) [x]
  (if (copy [A B] (A (flip-coin)))
    (B (inc (narrow [B] (copy [A B] x))))
    (B (println "Nothing!"))))

(defchor solo-inc [A B] (-> #{A B} B) [x]
  (B (inc x)))
```

## Lifting

```
(defchor maybe-inc [A B] (-> A B) [x]
  (if (A=>B (A (flip-coin)))
    (B (inc (A->B x)))
    (B (println "Nothing!"))))
```

## Lifting

```
(defchor maybe-inc [A B] (-> A B) [x]
  (if (A=>B (lifting [A] (flip-coin)))
    (lifting [B] (inc (A->B x)))
    (lifting [B] (println "Nothing!"))))
```

## Homogeneous Code

```
(defchor solo-inc [A B] (-> #{A B} B) [x]
  (B (inc x)))
```

## Homogeneous Code

```
(defchor solo-inc [A B] (-> #{A B} B) [x]
  (B (inc x)))

(defchor duo-inc [A B] (-> #{A B} #{A B}) [x]
  (inc x))
```

## Homogeneous Code

```
(defchor solo-inc [A B] (-> #{A B} B) [x]
  (lifting [B] (inc x)))

(defchor duo-inc [A B] (-> #{A B} #{A B}) [x]
  (inc x))
```

## Homogeneous Code

```
(defchor solo-inc [A B] (-> #{A B} B) [x]
  (lifting [B] (inc x)))

(defchor duo-inc [A B] (-> #{A B} #{A B}) [x]
  (lifting [A B] (inc x)))
```

## Choreographic Tuples

```
(defchor exchange-key [A B] (-> #{A B} #{A B} A B [A B])
  [g p sa sb]
  (pack (A (modpow ...)) (B (modpow ...))))
```

## Choreographic Tuples

```
(defchor exchange-key [A B] (-> #{A B} #{A B} A B [A B])
  [g p sa sb]
  (pack (A (modpow ...)) (B (modpow ...))))

(defchor secure [A B] (-> A B) [x]
  (unpack [[k1 k2] (exchange-key [A B] 5 23 (A 4) (B 3))]
    (B (bit-xor (A->B (A (bit-xor x k1))) k2))))
```

## Forced Agreement

```
(defchor exchange-key [A B] (-> #{A B} #{A B} A B #{A B})
  [g p sa sb]
  (agree! (A (modpow ...)) (B (modpow ...))))

(defchor secure [A B] (-> A B) [x]
  (unpack [[k1 k2] (exchange-key [A B] 5 23 (A 4) (B 3))]
    (B (bit-xor (A->B (A (bit-xor x k1))) k2))))
```

## Forced Agreement

```
(defchor exchange-key [A B] (-> #{A B} #{A B} A B #{A B})
  [g p sa sb]
  (agree! (A (modpow ...)) (B (modpow ...))))

(defchor secure [A B] (-> A B) [x]
  (let [k (exchange-key [A B] 5 23 (A 4) (B 3))]
    (B (bit-xor (A->B (A (bit-xor x k))) k))))
```

## Execution

(play-role *chor* {:role *role* ...} *arg*$^*$)

## Execution

```
(play-role chor {:role role ...} arg*)

(simulate-chor chor arg*)
```

## Conclusion

1. projection through macros
2. lightweight type system
3. sharing knowledge
4. lifting
5. homogeneous code
6. choreographic tuples
7. simulator

## Conclusion

1. projection through macros
2. lightweight type system
3. sharing knowledge
4. lifting
5. homogeneous code
6. choreographic tuples
7. simulator

- HM-style type inference
- concurrency up to data dependency (non-blocking `recv`)
- row polymorphism

## Thanks! Questions?

KλOR

lovrosdu/klor