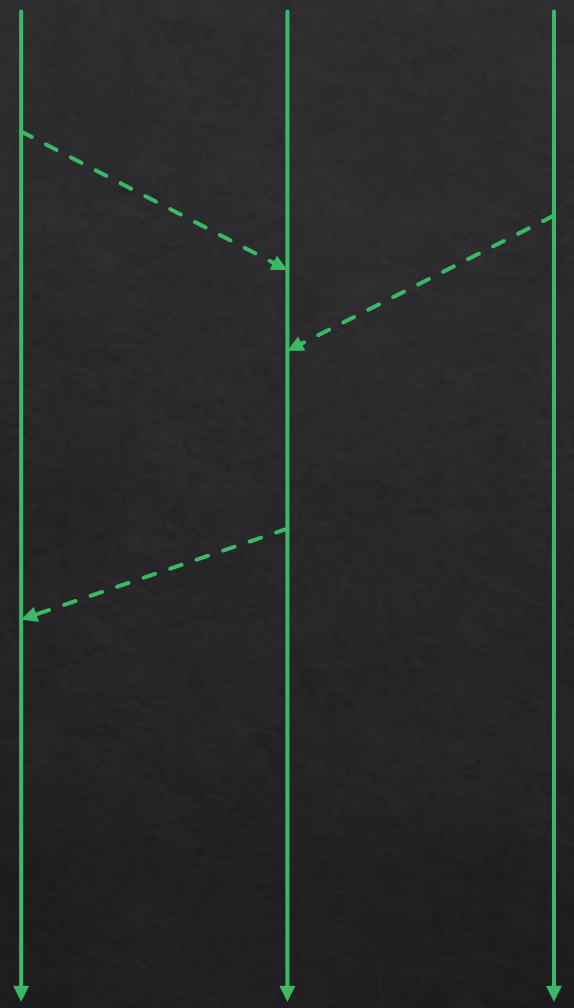


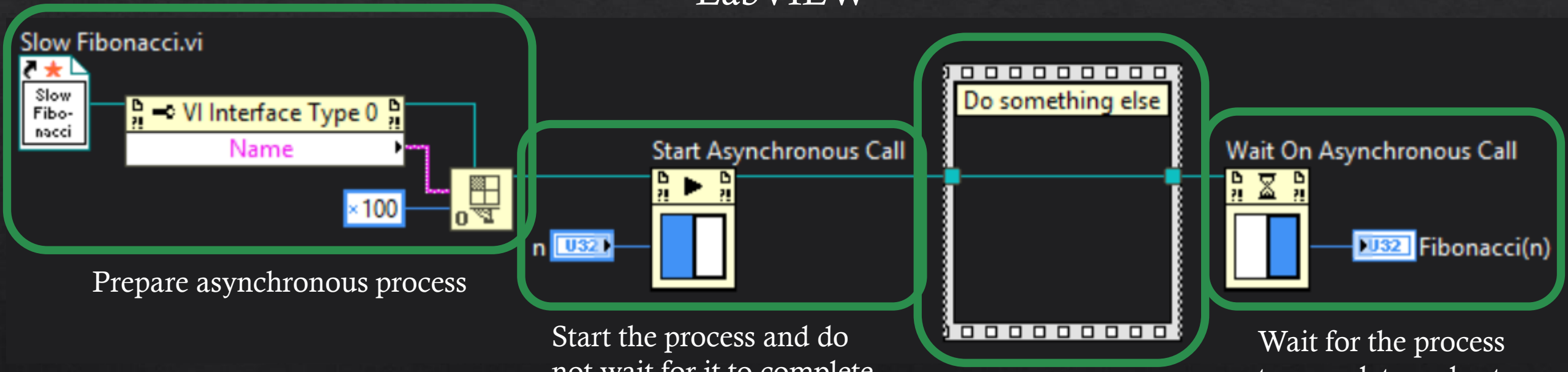
Exploring Concurrency

Blake Mitchell

Inspiration

- Async/await paradigms and APIs are in many modern languages.

LabVIEW



Prepare asynchronous process

Start the process and do not wait for it to complete

Potentially do something else while the asynchronous process is running elsewhere.

Wait for the process to complete and get its return value

Inspiration

F#

```
let rec slowFibonacci n =
    match n with
    | 0 -> 0
    | 1 -> 1
    | x -> slowFibonacci(x-1) + slowFibonacci(x-2)

/// Define an asynchronous workflow
let slowFibonacciAsync n = async {return (slowFibonacci n)}
/// val slowFibonacciAsync: n:int -> Async<int>

/// Start the workflow and don't wait for it to complete.
let startedWorklow = Async.Start(slowFibonacciAsync 36)
/// val startedWorfklow: unit

/// Start the workflow and wait on the result.
let result = Async.RunSynchronously (slowFibonacciAsync 36)
/// val result: int
```

Async expressions in Scheme

```
(async expression)                -> Returns <async-expression> immediately and expression is not evaluated.

(start-async async-expression) -> Starts the asynchronous process and returns <awaitable> immediately.

(await-async awaitable)           -> Waits for the asynchronous process to complete and returns its result

(start-async-synchronously async-expression) -> Starts the asynchronous process but waits for the result.
```

Async examples

```
> (define async-example (async (begin (sleep 2000) (display "Waited 2 seconds."))))
;Value: async-example

> async-example
;Value: #[<async-expression> 12 expression: (begin (sleep 2000) (display "Waited 2 seconds."))]

> (define awaitable-example (start-async async-example))
;Value: awaitable-example

> awaitable-example
;Value: #[awaitable 13]

> Waited 2 seconds.
```

Displayed to the console after two seconds.

Demo

async structure

```
;;; Defines a record structure for an async-expression.
;;; This is mainly used as a wait to hide the underlying implementation
;;; of the async-expression and for pretty printing at the REPL.
(define-record-type <async-expression>
  (make-async-expression promise-thread          ;; Constructor
                         result-value-queue
                         symbolic-expression)
  async-expression?                              ;; Predicate
  (promise-thread get-promise-thread)            ;; Accessors
  (result-value-queue get-result-value-queue)
  (symbolic-expression get-original-expression))
```

async macro

```
;;; Defines a macro that takes the arguments to async and returns an asynchronous process.
;;; This macro is used so that the arguments are not evaluated. Call like (async body).
(define-syntax async
  (syntax-rules ()
    ((_ x)
      (let ((queue (make-thread-queue 1))) ;; Create a single-element queue for the return value
        (make-async-expression
          (delay (thread (thunk (push! queue x)          ;; Delayed evaluations
                                stop-current-thread))) ;; Thread can be restarted
          queue
          (quote x))))))
```

async macro

```
(delay (thread (thunk (push! queue process)))
```

- `delay` is used to create a promise
  - Delays expression that can be forced later, which caches the result.
- `thread` is used to create the asynchronous process
- A `thunk` is what thread expects
  - This is how the thread creation process normally delays evaluation
- When the delay is forced, `process` evaluates
- And then places its value on the `queue`.

Some failure

✓ `(async expression)  ->  #[<async-expression>]`

```
> (list (async (display 1)) (async (display 2)))
;Value: ( #[<async-expression> 16 expression: (display 1)]
          #[<async-expression> 17 expression: (display 2)] )
```

✗ `(async expression ...)  ->  ( #[<async-expression> 1] ... )`

```
> (async (display 1) (display 2))
```

- Because start-async and await-async are just procedures, they can be mapped.

- Example:

```
> (map await-async (map start-async (list (async (* 2 3)) (async (* 4 5)))))
;Value: (6 20)
```

Some failure

✓ `(async expression)` -> `#[<async-expression>]`
✗ `(async expression ...)` -> `( #[<async-expression> 1] ... )`

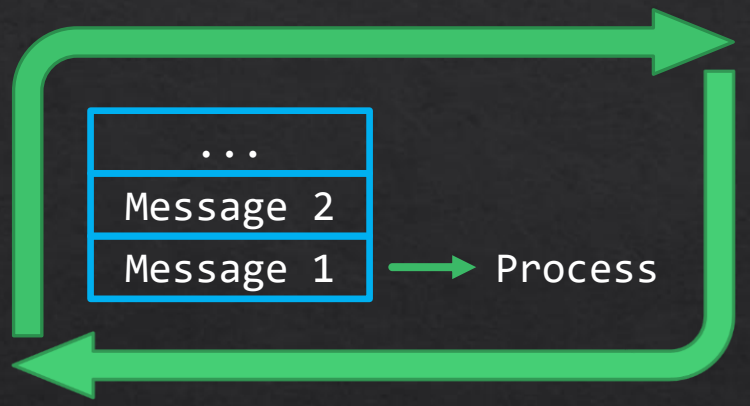
- You cannot map a macro with `map`
- Created a macro that symbolically maps `async`
  - Works for simple processes but led to lexical binding issues
- Probably possible but I wasn't able to use `syntax-rules` to handle parameterized async processes.
- Example:

```
> (define (tester x) (* 2 x))
> (define (async-tester y) (new-async (test y)))
> (start-async (async-tester 3))
;The thread #[thread 15] signalled an error #[condition 16 "unbound-variable"]:
Unbound variable: y
```

This works with current implementation of async

Actors

- Actors are:
  - concurrent, stateful processes
  - that can send messages to one another
  - and launch other actors.



- So actors can be modeled with a fairly simple process:

```
(define (actor-loop state message-processors inbox)
  (let* ((message (pop! inbox))                    ;; Wait for a message to arrive.
         ...
    (cond ((stop-message? message) state)           ;; Check for the stop message, which stops the actor
          (else ...                                ;; Otherwise, process the message.
            (actor-loop new-state                  ;; Go back and listen for a new message
                        new-message-processors
                        inbox))
```

Actors

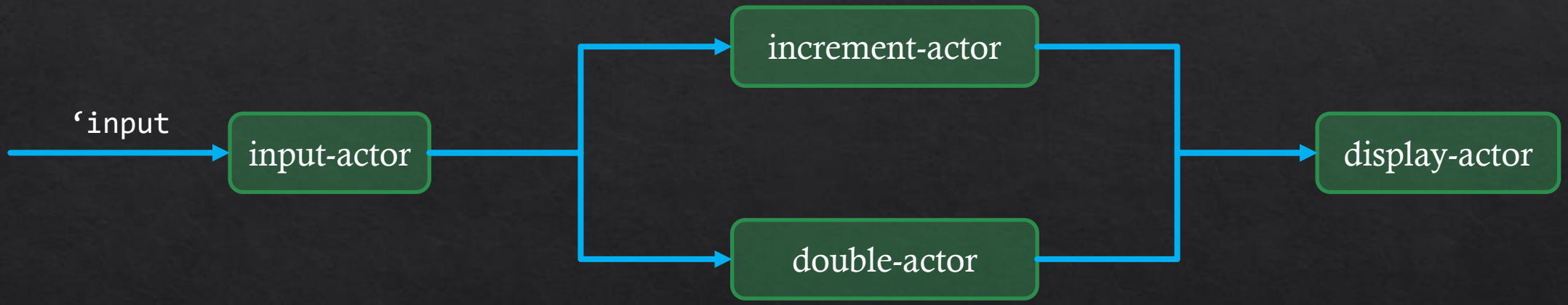
```
;;; A message procedure is of the form (lambda (state message-value) ...) -> state
(define (create-message-processor name message-procedure) ...)

(define (create-actor initial-state message-processors) ...) -> #[actor]

;;; Launches the actor as an asynchronous process.
(define (launch-actor actor) -> #[actor-address]
  (let* ((async-expression (async (actor-loop (actor:get-initial-state      actor)
                                              (actor:get-message-processors actor)
                                              (actor:get-message-queue      actor))))
         (awaitable (start-async async-expression)))
    (make-actor-address (actor:get-message-queue actor)))) ;; Return the actor's address

(define (send-message actor-address message-name message-value)
  (let ((address (check-for-address actor-address)))
    (push! (actor-address:get-queue address)
           (list message-name message-value 'async))))
```

Lambda actors



Demo

Conclusions

- Created an async/await API
  - `async`, `start-async`, `await-async`
- Created an actor framework and API using async expressions and processes
  - `create-message-processor`, `create-actor`, `launch-actor`, `send-message`
- Many examples
  - Lambda actors
- Future work
  - Update the `async` macro
  - Potentially integrate Prof. Sussman's pattern matching and generic dispatching into actors
  - Teachable actors?