Exploring Concurrency

Introduction

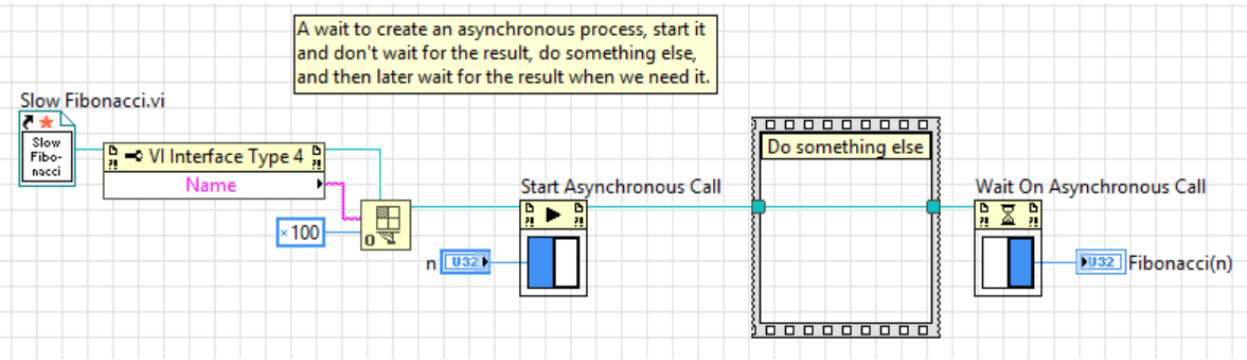
In general, I am interested in concurrency and in domain modeling. This project was meant to explore concurrency models that are inspired by async/await functionality seen in other modern languages and by actor model implementations, which could then be used to model more complicated processes not easily represented by the base language.

Overview

Asynchronous computation is an important concept in software engineering. I think many programmers don't deal with asynchronous computations on a daily basis, but those who do cloud computing, user interface design such that the underlying processes do not block user interaction, and hardware systems are faced with asynchronous operations daily. The common thread here is that for things that interact with the real world, that is computations that are side effectful, are often best modeled as asynchronous operations.

What I wanted to explore was how to build operations such that I could wrap normal processes as asynchronous processes that could be started and interacted with later. These processes could be started, other things could occur in the program, and then if needed, the result of the asynchronous processes could be waited upon, if they are to finish. In other cases, the asynchronous processes could be started and never waited upon to finish, until the entire program is done. An example of such a use case is that of actors, which will be explained later.

Some direct inspiration comes from two languages that I have used professionally, LabVIEW and F#. In LabVIEW, a language that is multithreaded and computes on cores in parallel by default, has explicit support for asynchronous processes. An example might look like this:



This is the maybe the simplest example of what we might want for an asynchronous process, in any language. However, part of my project was to build upon this functionality and extend it to more complicated designs.

As the final inspiration, which I think helps illustrate that we might want to "wrap" a normal process to make it into an asynchronous process automatically. In F#, such a wrapping is called an asynchronous

workflow. The example is below, and I think it provides a nice counterpoint to what I will show in Scheme.

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

let slowFibonacciAsync n =
    async {return (slowFibonacci n)}

// Start the async process and wait for its result.
let startedProcess = Async.RunSynchronously (slowFibonacciAsync 36)
```

As you can see, F# has special syntax for asynchronous workflows. That means that F# has to have explicit compiler support for such a language feature. This is true in LabVIEW as well. As we shall see in Scheme, we can create the exact same behavior with ZERO additional compiler support. Everything is there to modify the language, in a rather non-trivial way.

As an example of the API I have built, the above examples would look like:

```
;;; Computes the nth Fibonacci number, slowly
(define (slow-fibonacci n)
  (cond ((= n 0) 0)
    ((= n 1) 1)
    (else (+ (slow-fibonacci (- n 1))
         (slow-fibonacci (- n 2))))))

;;; Wraps the regular procedure slow-fibonacci as an asynchronous process
(define (async-slow-fibonacci n)
  (async (slow-fibonacci n)))

;;; This mimics the F# example.
(await-async (start-async (async-slow-fibonacci 20))) ; -> 6765

;;; This mimics the LabVIEW example.
(define started-process (start-async (async-slow-fibonacci 32))) ; -> Returns immediately
(do-something-else) ; -> Prints "Did something else." immediately after the above executes.
(display (await-async started-process)) ; -> Prints "2178309" once the computation is done.
```

Asynchronous Expressions API

I first created an asynchronous expressions library that allows one to create asynchronous expressions, start them as asynchronous processes, and then if needed, wait on the computed result. Additionally, I built functionality that easily allows one to build up many asynchronous expressions and start them all at once.

The API looks like this:

(define-record-type <async-expression> ...)	-> A record type is defined for asynchronous expressions.
(async *expression*)	-> Returns an asynchronous expression and does not evaluate expression.
(start-async *async-expression*)	-> Starts an asynchronous expression as an asynchronous process. Does not wait for it to complete. If the expression was created with (async expression), this executes (expression) in another thread. Returns an awaitable.
(await-async *awaitable*)	-> Waits for a started asynchronous process to finish and returns its result. This will block the calling thread indefinitely until the process is finished.
(start-async-synchronously *async-expression*)	-> Starts an asynchronous expression as an asynchronous process, waits for the process to finish, and returns the result.

This API is quite usable and user friendly. And nearly self-documenting. An example usage is:

```
(slow-fib 31)                    -> 1346269 (but waits for the computation, which is slow)
(define a (async (slow-fib 31))) -> returns an #[async-expression] immediately
(define b (start-async a))       -> returns an #[awaitable] immediately
(await-async b)                  -> 1346269 (blocks until the process has computed the result)
```

The major challenge in designing and implementing this API was primarily in understanding the timing of evaluation, or more explicitly, the delay of evaluation. By wrapping expressions and procedures as asynchronous expressions, we are explicitly saying “take this code and wrap it with something we can call it with later”. This is practically identical to promises, which using delay and force to delay evaluation and then later force the evaluation, but in this case, we are forcing the execution to happen asynchronously. That is, the execution happens on another thread. There was a design decision made to make every asynchronous expression be executed on its own new thread. This is so that thread management and coroutines could be avoided, as they aren’t needed to build up the conceptual ideas of the async/await API and then to use this API to do other things, like actors. This may not scale well and is maybe not “lightweight”, however, this is okay for a first pass design since these issues are not goals of this project. The implementation of the async macro is:

```
(define-syntax async
  (syntax-rules ()
    ((_ x)
     (let ((queue (make-thread-queue 1))) ;; 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))))))
```

One can see that is actually fairly simple once one comes up with the process. To start the asynchronous expressions, we simply force the internal promise:

```
(define (start-async async-expression)
  (if (async-expression? async-expression)
      (let ((promise (get-promise-thread async-expression)) ;; Get the async-expression parts
            (queue (get-result-value-queue async-expression)))
        (force promise)         ;; Force the async-expression's promise, starts the thread
        (make-awaitable queue)) ;; Return awaitable which is used to get the resulting value
      (error "The given expression is not an async-expression.")))
```

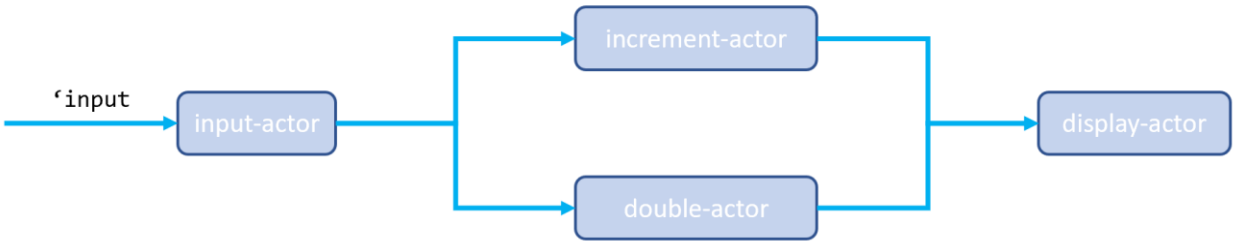
This returns an awaitable, which is a defined record structure. (To see the record definitions, refer to the attached code.) To await these processes, I used the rather simple solution of a single-element queue, which is always surprisingly useful in inner-process communication. Thus, awaiting an asynchronous process' value is simply to wait on the queue to receive an element:

```
(define (await-async awaitable)
  (cond ((async-expression? awaitable) (pop! (get-result-value-queue awaitable)))
        ((awaitable? awaitable) (pop! (get-thread-queue awaitable)))
        (else (error "The expression passed in is not an async-expression or awaitable."))))
```

Actors

Now that we have an easy way to launch processes asynchronously, we can play around with actors. **Actors are defined as asynchronous processes that carry state, can send and receive messages from other actors, and can launch other actors**. That's it! Actors are very simple. They can be thought of as little machines that sit around and listen to an inbox for new messages. When a new message arrives, the actor either processes the message by acting upon it somehow or discarding it. In between processing messages, the actor may of course be sending itself messages that performs some inner-actor computation.

A good way to get started is to think of an actor as modeling a procedure. That is, its state is a procedure, and the messages it listens for are input values to the procedure. When the message arrives, the procedure is applied to the value. Then the actor can send the result to other actors or maybe display the result to the console. Although we haven't introduced how this is done, it is already clear that we can now model asynchronous computations such as:



Actors are modeled fairly simply as a recursive procedure that waits for a message to arrive, processes the message, and then recursively iterates by passing its state, message processors, and inbox to the next iteration. We can define an actor loop as:

```
;;; This procedure defines the actual actor process. This is what is launched as an
;;; asynchronous process. The actor-loop processes messages as they arive and then recursively
;;; evaluates to wait for the next message. This procedure handles messages global to all
;;; actors, such as the stop message.
(define (actor-loop state message-processors inbox)
  (let* ((message (pop! inbox))                          ;; Wait for a message to arrive.
         (message-name (car message))                    ;; Get the message's name.
         (message-value (cadr message))                  ;; Get the message's value.
         (is-synchronous? (equal? 'sync (caddr message)))) ;; Check synchronous flag.
    (cond ((stop-message? message) state) ;; Received stop message, so stop the actor.
    (else (let ((procedure (get-message-processor-procedure message-name
                                                            message-processors)))
      (if (equal? 'no-message-processor procedure)    ;; Check for a msg processor
          (actor-loop state message-processors inbox) ;; Ignore the message
          (actor-loop (procedure state message-value) ;; Process the message and
                      message-processors              ;;   update the state
                      inbox)))))))
```

We can consider an actor's message processor to be a procedure that has a symbolic name associated with it and a procedure that is of the type `(lambda (state message-value) …) -> new-state`. That is, when a message is received, we check the actor's message processors for one that is associated with the message name received. Then we strip out the message processor's procedure, apply it to the current state of the actor and the message value, and then return a new state for the actor.

Since we have the ability to launch asynchronous processes with the above async API, we can now launch actors using this API:

```
;;; Launches an actor and returns the actor's address for sending messages to the actor.
(define (launch-actor actor)
  (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.
```

Using this fairly simple setup, plus all the supporting procedures for creating the various structures, creating message processors, etc., we can now create the lambda actor process depicted in the earlier diagram. I do not include that code here, since it takes up some space, but this can be found in the code base underneath the section title General Lambda Actors.

Conclusion

In the end, Scheme makes for a wonderful prototyping language for new APIs and new language designs on top of the existing Scheme substrate. Using a few dozen lines of code, excluding comments, one can very easily creating a working prototype of asynchronous expressions and processes, as done in this project. From there, various paradigms and use cases of asynchronous processes can be built up, actors being just one example. And even then, actors themselves can be considered as an asynchronous process combinator language. In general, everything worked out as planned, with a lot of time spent making sure the API was clean and useful. There were some issues with macros, which I have left out here due to space concerns. However, the issues are fairly well documented in the attached codebase. I think for future additions, one might consider integrating Prof. Sussman's pattern matching and generic dispatching into the actor framework.