Blake Mitchell
4/1/2019

Exploring Concurrency

Overview

In general, I am interested in concurrency and in domain modeling. While I have built concurrent applications for hardware systems, which in some ways utilizes building a model of that encapsulates both abstract and external, or real, concurrent processes, I have always built upon some existing concurrency substrate. This project is to explore building up some concurrency layers in MIT Scheme, which don't seem to be readily available. Threading models and continuations are sort of mysterious to me at this point, and so I would like to learn more about them. Building some concurrency solutions seems like a good way to get to know these concepts better.

Goals

- Build a substrate of concurrency primitives on top of MIT Scheme's thread procedures such that more complex concurrent and parallel process can be built from these.
- Using MIT Scheme's queue library with the above concurrency primitives, build a small actor library.
- If time allows, it's possible the above two libraries could be used to implement dataflow processes.

Description

Not many languages are great at concurrency, but many languages have been slowly adding primitives for building up concurrent processes, normally underneath the name of "async", for example async/await in C# or Python and async workflows in F#. It is very helpful to have concurrency primitives at your disposal so that you don't have to constantly deal with the how of concurrency but rather on the what and your specific application of some concurrent processes. Taking inspiration from various languages, one could potentially implement the following syntax, behavior, and library in Scheme:

```scheme
(define example-process ...)
(define async-example-process (async example-process)) ; wraps a process as an async process
(define example-processes (map async (list ...)))      ; wraps multiple processes
(start-async async-example-process)                    ; starts & doesn't wait for completion
(define result (wait-async async-example-process))     ; starts and waits for async process to
                                                       ; return a value
(define result-list
  (wait-async (start-parallel example-processes))) ; starts multiple async processes in
                                                   ; parallel and waits for return values and
                                                   ; puts them in a list named result-list
```

It is possible that in implementing this type of library, one might end up with more granular concurrency procedures that could be used to build up other, but different, concurrency libraries. This could be useful since there is an ecology of different concurrent processes, and the above API is just a view into one of those.

Once you can launch and run asynchronous processes, such as those above, one might want the ability to communicate between these processes. One such way is the actor model, where the parallel processes, i.e., the actors, communicate via messages. The messages must use some sort of transport, and a natural transport for intraprocess communication are queues. An actor API might look like this:

```
; Creates an actor but does not start the actor process
(define (create-actor name initial-state message-processors) ...)

; Starts the actor process and returns an address for the actor
(define (start-actor actor) ...)

; Sends a message to an actor via address or name. Does not return a processed value.
(define (send-message message actor-address-or-name) ...)

; Batch sends multiple messages to a single actor via address or name
(define (send-messages messages actor-addresses-or-name) ...)

; Sends a synchronous message, waits for the message to be processed, and returns the result
(define (send-sync-message actor-address-or-name message) ...)

; Stops a running actor after it processes the messages received before the stop message
(define (stop-actor actor-address-or-name) ...)
```

Underneath the hood, this library would use the asynchronous library built above, Scheme queues, and some newly built actor-specific primitives.

Challenges

MIT Scheme's thread model is undocumented, as are its queue libraries. However, I found some documentation regarding the thread model here (http://web.mit.edu/benmv/6.001/www/threads.txt). I will need to read and become familiar with at least the thread.scm, queue.scm, and thread-queue.scm files found in the /src/runtime directory within the MIT Scheme system. There will probably be other portions of the source code that I will need to reference as well.