Communicating Sequential Processes in R5RS Scheme

CSC 33500 Programming Language Paradigms — Fall 2016

Professor Douglas Troeger

by Nelson Batista and Irving Derin

Introduction

Concurrency is a very important method for structuring computer programs to allow them to perform various actions as if they were instead acting as a set of individual processes running in parallel. Running programs in parallel allows them to execute overall more quickly, since neither is waiting for the completion of the other. However, whereas parallelism involves running multiple programs (or processes) at precisely the same moment, concurrency is a method of structuring programs into more individual units which may run in parallel or may run at differing times. While this difference sounds trivial, concurrency allows for greatly increased efficiency, regardless of whether the individual components are actually running in parallel.

For instance, the Go programming language has been described as "a concurrent [programming] language" by one of its creators, Rob Pike. His example is that of a gopher attempting to burn a large pile of outdated books by putting them into a wheelbarrow and delivering them to an incinerator, then going back to get more books and repeating the process until the pile of books is empty.

In this case, concurrency would be the introduction of one or more gophers to the process of the book-burning, each acting independently of the others. One gopher may be the wheelbarrow gopher, moving the wheelbarrow full of books to the incinerator. Another gopher may be the incinerator gopher, emptying the wheelbarrow into the incinerator. Still another gopher may be the book gopher, responsible for loading books from the pile to the wheelbarrow. Pike notes that the design of each gopher must be careful not to have it prevent the others from doing its job. The execution of the book-burning program, so to speak, is represented by each gopher's movement as it performs its respective task. While each gopher may be running with the others at the same time (in parallel), it is not necessary in order for the concurrent nature of the program structure to be maintained. Additionally, assuming each gopher has been implemented properly, the overall book-burning program will run much more efficiently, regardless of whether the gophers are actually running in parallel (which may be impossible to truly be the case if, say, the program is running on a single-core CPU).

This idea of ensuring that each gopher is properly coordinated with the others is itself an important aspect of concurrency. There are several methods of implementing proper coordination. Among them is the idea of communicating sequential processes (CSP). CSP is a language defined by Anthony Hoare in his 1977 paper *Communicating Sequential Processes*. In it, he describes a language in which objects and various other parts of the program are modeled as processes which can set off and react to events. These events are communicated to other processes through channels.

This idea is further explored in the paper 2011 *Places: Adding Message-Passing Parallelism to Racket* by Kevin Tew et al., which defines its own set of functions for creating and acting on different processes which can pass messages among themselves and run in parallel. This along with the lack of similar work in Scheme inspires the possibility of implementing Hoare's CSP in Scheme in a similar manner.

We first begin with the most fundamental building blocks: channels and processes. These we build as higher level objects that will be composed of primitives that we recreate from Hoare's paper. Our goal is to achieve a balance of the mathematical rigor CSPs provides, with higher level constructs to allow easy use, much like the types expressed in the Go programming language (Golang).

The Channel Primitive

The Channel primitive takes strongly from Hoare's definition of a channel being a dotted pair. It contains a channel name as well as a value. We add to that a flag that will be set when some process places a value on that channel. The basic implementation of CSPs uses unbuffered channels, forcing synchronous execution. It is possible, as Golang shows, to add a buffer to the channel. In Hoare's text, he discusses the disadvantages of using buffers, such as dead-locks created from buffer mismatches, or the possibility of crashing a system from unbounded buffers.

We start off the creation of our channel primitives by defining two constructors: `make-channel` and `make-channel!`. The two differ in that the latter modifies a global list of channels, while the former creates a simple channel. The use of a global set of named channels is not expressed in Hoare's text, but allowed us some flexibility in how we would approach dispatching events to processes we've created.

When interacting with the channels, we use two main functions: `read-channel!` and `write-channel!`. The purpose of these functions is to produce and consume messages that are sent on the global channels.

The Actor Model

This seems like an appropriate place to sidebar about the Actor Model. Described by Hewitt in 1973 and 1977, the Actor model is a similar concurrency model to CSPs. Both address the problems of shared memory by using message passing. As we can see in our implementation of channels, the main difference lies here in how messages are handled. In the Actor Model, processes are "actors", with mailboxes. When a process interacts with another process, it does it by sending a message to the mailbox of the individual process. In CSPs, there are global channels, where processes consume and produce messages to those channels respectively.

Implementation of the Queue Data Structure

The most important aspect of our implementation of the process primitive was the implementation of a process queue. This necessarily requires implementation of a queue data structure, which is not normally available in Scheme (lists are essentially stacks, as push is represented by cons and pop by car).

While the queue as a concept should be familiar to most of our audience, we describe it in passing as a data structure onto which multiple values may be placed, and values in it are accessed in the order which they were placed onto the queue. The first element placed into the queue by a push operation is the first element retrieved by a pop operation. The queue is thus an example of a “first in, first out” data structure. R5RS’s use of what are essentially stacks for its sole data structure makes an implementation of queues a bit non-obvious, as opposed to other languages which offer data structures with random access.

Our code, adapted from the implementation here, contains all the various functions expected from a queue, with the exception of `full?`.

Queues are declared by the `make-queue` function. This initializes a queue with an empty list as its list of values, and an empty list as the most recent value put in the queue (to implement the `rear` function).

The `push` function takes a value and a queue, and appends the value to the queue. The way it does this is by using the `set-cdr!` function provided in R5RS to change the cdr of the queue to the value contained within a list. This creates a new queue with the pushed value as the last element in its list of values. The `push` function also sets the cdr of the queue to the value pushed so that it can be accessed in constant time by the `rear` function.

The Process Primitive

In our implementation, we took a very simple approach to the process itself. It is simply a function that is placed onto our queue implementation. This was implemented in an incredibly crude manner. Given a better understanding of continuations, there would be a queue of continuations in our queue. A process would only have to enter onto that queue if it's being blocked by a `read-channel!` or `write-channel!`. One unsuccessful method I tried with trying to create a

The Prefix Operator

The prefix operator takes an event and a process. The operator creates a process which performs the event given, then behaves like the process given. For instance, $(x \to P)$ ("x then P") refers to the process which first completes event x then behaves like process P.

In his text, Hoare offers several examples to better illustrate potential uses of the operator as well as to clarify its functionality. Perhaps the simplest of these is his proposed clock process:

$$CLOCK = (tick \to CLOCK)$$

This defines a clock process recursively. It is a process which simply ticks, then behaves like a clock process, which also simply ticks, then behaves like a clock process, and so on in an infinite loop. The result is a process which ticks indefinitely without doing anything else. Below is a similar clock process, a clock which ticks, then tocks, then behaves like a clock process:

$$CLOCK = (tick \to (tock \to CLOCK))$$

This demontstrates the possibility of nested prefix operators. The clock process is now defined as a process which ticks, then behaves like a process which "tocks", so to speak, then behaves like a clock. We thus have a process which alternates ticking and tocking, more inline with the idea of a physical clock.

In our implementation, it was quite trivial to think of an event as simply the execution of a single function. As such, the implementation of the prefix operator follows naturally as a function which executes some `event` argument before adding a separate `proc` argument to the process queue to be executed like any other process. The function is thus defined as `(prefix event proc)` and, exactly as described, takes an event and a process as arguments and performs the event (executes the function passed as `event`) before adding the process to the event queue to be executed.

Thus, a process can be define using the prefix operator by running `(coroutine (prefix x p))`, which pushes to the process queue a process which performs some action `x`, then behaves like another proccess `p`, exactly as described.

There are a number of limitations to this approach, however. If the event passed happens to write to a channel, for instance, it is necessary for at least one other process be in the process queue. This is because the function `write-channel!` calls `start`, which pops the process queue and applies the element popped. If the event writes to a channel (perhaps defined outside so that it is contained within the event's closure, or defined within the event function itself) without any processes in the queue, it will call `start`, and an error will be thrown as an empty list is not a function which can be applied.

One possible way to circumvent this problem is by pushing the process to the queue before doing the event. This would guarantee that at least one process is on the queue before the event is run, preventing any single `write-channel!` call within the event from causing problems. However, this approach would not only violate the premise of the prefix operator in that the event is to be performed *before* behaving like the given process, but would also do nothing to guard against more than a single `write-channel!` call within the event. It is thus more practical, if more dangerous, to leave the order as-is. This keeps the implementation faithful to Hoare's description of the prefix operator, but requires a precondition that the event passed will either not make calls to `write-channel!` or will make sure that the process queue has at least one process in it before making any such calls.

The Deterministic Choice Operator

The deterministic choice operator is a bit more complex. It is technically a combination of two processes defined by prefix operators. The exact notation is:

$$(x \to P) \,\Box\, (y \to Q)$$

This is a process which can perform one of two possible actions. Depending on which action is completed first, that action's corresponding process will become the behavior of the process.

We can consider a simplified version of an example provided by Hoare. We can model a vending machine as a process which waits for a customer to select a product to dispense. Say the vending machine offers chips and candy. In this case, the customer selecting chips represents event x, while P is represented by a machine which dispenses a single bag of chips before behaving like a vending machine (in that it waits for an input before dispensing a single item). The situation is similar for candy. We note that this demonstrates the ability of deterministic choice to be defined recursively in a similar manner to prefixing. P and Q are both processes which themselves contain determinstic choice. Of course, this is partially due to them being essentially the same process, but the idea is that it is a possibility.

Implementing deterministic choice in Scheme proved to be more challenging than would immediately seem. Ultimately, the design settled on was that of a function which takes 7 arguments: the two events, their respective processes, two symbols, and a channel. The choice is therefore ultimately left to the environment. The channel is used to read a symbol. For this, we use `peak-channel` rather than `read-channel!` so as to not modify the channel in the process of checking whether its value is equal to the symbol passed to the operator. If it is, then the process defined by the prefix operator applied to the symbol's corresponding event and process is pushed to the process-queue. Otherwise, the operator pushes an instance of itself being called with the same arguments to the process queue. This, in a sense, allows the process to wait for one of the two symbols to be sent on the channel. Thus, we model the event deciding which process is to determine the behavior of the deterministic choice operator as a combination of a particular value being read from a channel and an event being executed through the prefix operator's normal functionality. This leaves the actual choice up to the environment (specifically, the execution of other processes) in which the operator is being used, since they are responsible for sending one of the two necessary symbols on the channel, and thus responsible for allowing the deterministic choice operator to do anything other than continuously push itself onto the process queue.

It may appear that the order in which the channel's first entry is checked would favor the first process's execution, since the first process is the one that has its value checked for presence on the channel. However, our implementation of channels ensures that only one value is ever on the channel at once, thus preventing the case in which both process' symbols are on the channel at once, which would stop one process from ever being able to enter. On the other hand, this introduces a separate issue in that, since the operator's method of waiting involves placing itself back on the process queue, there is no guarantee that a process won't write a value to the channel, expecting the write process to set off the deterministic choice, only to have the subsequent `start` call execute a different process.

The Non-Deterministic Choice Operator

Random Number Generator

Due to the non-deterministic nature of this operator, it proved useful to implement a (very) crude psuedo-random number generator. This was tricky, however, since R5RS does not provide any functions to generate random numbers, nor does it provide access to system variables, such as the current time, which could be used as an acceptable seed to an algorithm which generates them. As such, it was a bit of a challenge to find a value which could be used as a seed while also being volatile enough to not generate numbers which are predictable in their randomness. For instance, values such as the length of the host operating system name (if it were available), would be a poor choice for such a generator, as any particular system would always generate the same numbers in the same order since the seed would only change if the operating system itself were changed. Ultimately, we settled on using the length of the process queue, which changes quite frequently, especially in any real-world application of our CSP implementation. While this is *far* from optimal, for our purposes and the limits of the language it had to suffice.

As for the algorithm itself, in the interest of simplicity, we chose to implement a linear congruential generator. This generator is very simple. Starting from a seed value X_0, the linear congruential generator generates numbers using the recursive formula $X_{n+1} = (aX_n + c) \bmod m$, where a, c, and m are constants. In the interest of not digressing too far from the project itself, we will omit a more detailed explanation and instead explain that, for this particular generator, we used the same constants used by glibc, with $a = 1103515245$, $c = 12345$, and $m = 2^{31}$.

Non-Deterministic Choice

The non-deterministic choice operator is similar to its deterministic counterpart, in that the environment provides a particular event and, in doing so, sets off a particular process, but the key difference is that the environment ultimately cannot predict which process will begin. The notation is:

$$(a \to P) \sqcap (b \to Q)$$

Thus it is also a combination of prefix operators. The operator only responds if both a and b are received, at which point one of the two processes will be selected. For this, we used the random number generator. If the generator returns a 0, the first process is selected and placed on the process queue. Otherwise, the generator returned a 1, and the other process is selected instead.

To achieve this, within the function `non-determ-choice`, we defined a helper function called `phase2`. When `non-determ-choice` checks the channel and sees that its current value is equal to one of the two symbols passed, the other symbol, along with the remainder of `non-determ-choice`'s arguments to `phase2` and places it in the process queue. When `phase2` runs, it checks the channel for the other value and, if it is on the channel, uses the random number generator to generate a 0 or a 1 (by calling `(modulo (rng) 2)`), and from that selects one of the two event-process pairs to turn into a prefix-type process before placing it in the process queue.

If, at any time, the process reads the channel and does not find one of the necessary values, the process is restarted by placing `non-determ-choice` back in the process queue with its initial arugments. This holds even if `phase2` is the process checking the channel, since the non-deterministic-choice need not do anything if *both* values are not sent on the channel, so we may simply wait for them.

The Interleaving Operator

The interleaving operator proved to be a more significant challenge than it may seem. Its description is quite simple: it takes two processes and forms a process which executes actions from both processes concurrently. The order of their interweaving is undefined. However, this presents a very interesting challenge. It requires our system to, in some way, have some knowledge about the processes themselves.

The operator is denoted by:

$$P \,|||\, Q$$

As described, this creates a new process which executes actions of both processes in arbitrary order. This would thus suggest that we need to implement not only some type of meta-concurrency (while we have implemented concurrency, there is still usuaully one process running in its entirety at a time, though the processes should be designed around both this and the provided operators and support for channels). However, we note that it is not necessary for a given process to finish executing entirely before another process begins execution. Indeed, each call to `write-channel!` starts execution of a new process. Thus, through clever use of `write-channel!` calls and perhaps some shenanigans involving continuations, it may be possible to perhaps place the two processes on the process queue, before calling `write-channel!` with some dummy value to immediately begin execution of whatever is in the queue. However, there are no guarantees that the process executing will be the one pushed to the queue (if there is so much as a single other process on the queue at the time of pushing, that will be run instead), so we were ultimately unable to attain a satisfactory implementation of this particular operator. This is left as an exercise to the reader.

Hiding

Hiding is another operator that proved to be a bit out of our scope. The hiding operator is, in a sense, the opposite of the prefix operator. Whereas the prefix operator creates a process which executes a particular event before behaving like another process, the hiding operator prevents certain events from occurring. Hoare provides an excellent example in the "noisy" vending machine. The machine normally makes several sounds, "clink" and "clunk" during its operation, in addition to the normal behavior of a vending machine. For example, the noisy vending machine, which accepts a coin before dispensing chocolate, may be modeled as:

$$NOISYVM = (coin \rightarrow (clink \rightarrow (clunk \rightarrow (chocolate \rightarrow NOISYVM))))$$

However, should we want a vending machine which simply accepts a coin before dispensing chocolate (without making any noise), we may describe one using the hiding operator like so:

$$VM = NOISYVM \setminus \{clink, clunk\}$$

This is equivalent to:

$$VM = (coin \rightarrow (chocolate \rightarrow (NOISYVM \setminus \{clink, clunk\})))$$

Or, substituting our previous definition,

$$VM = (coin \rightarrow (chocolate \rightarrow VM))$$

As expected. The difficulty in implementing such an operator should now be apparent. Hoare offers an implementation which relies on knowledge of the process's alphabet, which our implementation of processes does not provide. Specifically, one may implement a function which takes both the process and the set of its alphabet's events which must not occur as arguments, then check each action the process takes to see if it is among the set of events which must not occur, and force the event to either not occur or occur in such a way that its effects are invisible to the system. As our implementation of processes does not concern itself with alphabets, and R5RS provides little in the way of reflection, such an implementation cannot be made without substantial effort towards rebuilding our CSP implementation.

Discussion

In our journey to implement CSPs in Scheme, we were confronted with a constant challenge of balancing the CSP Book by Anthony Hoare and a practical implementation of CSPs such as Golang. In this fashion we have written out the scheme implementation of some of the operators talked about by Hoare. This struggle between the two sides gave us a unique understanding of the difficulties of implementing new, fundamental elements to a programming language. All while still keeping them useful.

One example that immediately comes are our implementations of the choice operators.

With the difficulties we had implementing processes, we have developed code that reflects what we see in the original paper.