Assignment 5:
Concurrency

15-312: Principles of Programming Languages (Spring 2023)

This assignment studies concurrency through the lens of a concurrent programming language known as Concurrent Algol (**CA**).

Effectful computations (involving side-effects) distinguish themselves from "pure" computation by involving some sort of interaction. Think IO effects (user interaction) or reference cells (state/memory interaction). We seek to make this more explicit by introducing interaction as a primitive notion.

To this end, **CA** augments our existing expression/command modal separation with two new sorts accomodating interaction: *processes* and *actions*. The formulation of **CA** as presented in this assignment is derived from that of Chapter 40 of **PFPL** and relevant supplements posted on the course website.

CA is conceptually similar to the concurrent programming language Go.

1 Syntax and Semantics

First, we will introduce the user language of **CA**.

Typ	τ	::=	$\ldots$	$\ldots$	other types
			$\texttt{cmd}(\tau)$	$\tau\,\texttt{cmd}$	command
			$\texttt{chan}(\tau)$	$\tau\,\texttt{chan}$	channel
Exp	e	::=	$\ldots$	$\ldots$	other expressions
			$\texttt{cmd}[\tau](m)$	$\texttt{cmd}\,m$	encapsulation
			$\texttt{chref}\langle a\rangle$	$\&\,a$	channel reference
Cmd	m	::=	$\texttt{ret}(e)$	$\texttt{ret}\,e$	return
			$\texttt{bnd}(e\,;x\,.\,m)$	$\texttt{bnd}\,x \leftarrow e\,;m$	sequence
			$\texttt{spawn}(e)$	$\texttt{spawn}(e)$	spawn process
			$\texttt{emitref}(e_1\,;e_2)$	$\texttt{emitref}(e_1\,;e_2)$	send message on channel
			$\texttt{sync}(e)$	$\texttt{sync}(e)$	receive message on channel
			$\texttt{newchan}\langle\tau\rangle(a\,.\,m)$	$\texttt{newchan}\langle\tau\rangle(a\,.\,m)$	new channel

You may think of *other types* and *other expressions* as common features discussed earlier in the semester, such as products, sums, and natural numbers.

Definition 1.1 (Synchronicity)**.** We say that an operation runs synchronously if it must complete before further operations can begin. Conversely, we say that an operation runs asynchronously if it immediately returns, running concurrently and thus not blocking further operations.

We informally describe the behavior of each command as follows:

- $\texttt{ret}(e)$: An "interface" between the expression and command sorts, including all expressions as trivial commands.
- $\texttt{bnd}(e;x.m)$: Sequencing of computation, synchronously running the suspended computation e and then binding its result to x and running m. Note that e is an expression, allowing the programmer to *compute* a command to run.
- $\texttt{spawn}(e)$: Creation of a new concurrent process, where e is the (encapsulated) command that will be asynchronously executed by the process. return type of the command as usual. Returns a "return channel" channel reference, which the spawned process uses to send the value returned by the command.
- $\texttt{emitref}(e_1\,;e_2)$: Sends the value e_2 along channel reference e_1 asynchronously, returning immediately regardless of whether the message has been received yet.
- $\texttt{sync}(e)$: Waits for a value to be transmitted along channel reference e synchronously, blocking until a value is received. Upon seeing a transmission, returns the value.
- $\texttt{newchan}\langle\tau\rangle(a\,.\,m)$: Creates a new channel for use within the command m.

1.1 Statics

The statics of the language are defined with three judgments. As usual, $\Gamma \vdash_\Sigma e : \tau$ is the typing rule for expressions, here including a channel context $\Sigma = a_1 \sim \tau_1, \ldots, a_n \sim \tau_n$ describing the names and associated types of channels that are available for use. The rules for the new forms are given below.

$\boxed{\Gamma \vdash_\Sigma e : \tau}$

$$\frac{\Gamma \vdash_\Sigma m \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau}{\Gamma \vdash_\Sigma \mathtt{cmd}[\,\tau\,](\,m\,) : \mathtt{cmd}(\,\tau\,)}(\mathrm{SE}_1) \qquad \frac{}{\Gamma \vdash_{\Sigma, a \sim \tau} \mathtt{chref}\langle a \rangle : \mathtt{chan}(\,\tau\,)}(\mathrm{SE}_2)$$

The judgment $\Gamma \vdash_\Sigma m \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau$ formalizes the typing for commands.

$\boxed{\Gamma \vdash_\Sigma m \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau}$

$$\frac{\Gamma \vdash_\Sigma e : \tau}{\Gamma \vdash_\Sigma \mathtt{ret}(\,e\,) \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau}(\mathrm{SM}_1) \qquad \frac{\Gamma \vdash_\Sigma e : \mathtt{cmd}(\,\tau\,) \qquad \Gamma, x : \tau \vdash_\Sigma m \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau'}{\Gamma \vdash_\Sigma \mathtt{bnd}(\,e\,;x\,.\,m\,) \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau'}(\mathrm{SM}_2)$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{cmd}(\,\tau\,)}{\Gamma \vdash_\Sigma \mathtt{spawn}(\,e\,) \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \mathtt{chan}(\,\tau\,)}(\mathrm{SM}_3) \qquad \frac{\Gamma \vdash_\Sigma e_1 : \mathtt{chan}(\,\tau\,) \qquad \Gamma \vdash_\Sigma e_2 : \tau}{\Gamma \vdash_\Sigma \mathtt{emitref}(\,e_1\,;e_2\,) \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \mathtt{unit}}(\mathrm{SM}_4)$$

$$\frac{\Gamma \vdash_\Sigma e : \mathtt{chan}(\,\tau\,)}{\Gamma \vdash_\Sigma \mathtt{sync}(\,e\,) \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau}(\mathrm{SM}_5) \qquad \frac{\Gamma \vdash_{\Sigma, a \sim \tau} m \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau'}{\Gamma \vdash_\Sigma \mathtt{newchan}\langle \tau \rangle(\,a\,.\,m\,) \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} \tau'}(\mathrm{SM}_6)$$

1.2 Examples

Using the informal description of the dynamics, we consider some simple examples.

Example 1.2. Consider the following command:

$$\mathtt{bnd}(\,\mathtt{cmd}(\,\mathtt{emitref}(\,\mathtt{chref}\langle a \rangle\,;312\,)\,)\,;u\,.\,\mathtt{sync}(\,\mathtt{chref}\langle a \rangle\,)\,)$$

Running this command given a channel $a \sim \mathtt{nat}$ would behave as follows:

1. Send 312 on channel a asynchronously, continuing immediately while the message 312 is available on channel a.
2. Synchronize on channel a, receiving the message 312 immediately because it is already available on channel a.
3. Return 312 as the result.

Example 1.3. Consider the following command:

$$\mathtt{bnd}(\,\mathtt{cmd}(\,\mathtt{sync}(\,\mathtt{chref}\langle a \rangle\,)\,)\,;u\,.\,\mathtt{emitref}(\,\mathtt{chref}\langle a \rangle\,;312\,)\,)$$

Running this command given a channel $a \sim \mathtt{nat}$ would behave as follows:

1. Synchonize on channel a, but getting "blocked"/stuck forever because no messages are available on channel a.

Example 1.4. Consider the following command:

$$\mathtt{bnd}(\,\mathtt{cmd}(\,\mathtt{emitref}(\,\mathtt{chref}\langle a\rangle\,;312\,)\,)\,;u\,.\,\mathtt{bnd}(\,\mathtt{cmd}(\,\mathtt{sync}(\,\mathtt{chref}\langle a\rangle\,)\,)\,;x\,.\,\mathtt{sync}(\,\mathtt{chref}\langle a\rangle\,)\,)\,)$$

Running this command given a channel $a \sim \mathtt{nat}$ would behave as follows:

1. Send 312 on channel a asynchronously, continuing immediately while the message 312 is available on channel a.
2. Synchonize on channel a, receiving the message 312 immediately because it is already available on channel a.
3. Synchonize on channel a, but getting "blocked"/stuck forever because no messages are available on channel a.

Task 1.1 (5 pts). Let $f : \tau_1 \to \tau_2$ and $\Sigma = a \sim \tau_1, b \sim \tau_2$. Write a command $\vdash_\Sigma m \mathrel{\dot{\sim}} \mathtt{unit}$ that reads a value x from channel a and sends $f(\,x\,)$ over channel b.

Henceforth, we abbreviate $\mathtt{bnd}(\,e\,;x\,.\,\mathtt{ret}(\,x\,)\,)$ as $\mathtt{do}(\,e\,)$.

Example 1.5 (Producer)**.** Let:

$$m_{\text{emit}} = \mathtt{bnd}(\,\mathtt{cmd}(\,\mathtt{emitref}(\,\mathtt{chref}\langle a\rangle\,;n\,)\,)\,;x\,.\,\mathtt{do}(\,f(\,n'\,)\,)\,)$$

$$m_{\text{produce}} = \mathtt{do}(\,\mathtt{ap}(\,\mathtt{fun}[\,\mathtt{nat}\,;\mathtt{cmd}(\,\mathtt{unit}\,)\,](\,f\,.\,n\,.\,\mathtt{ifz}(\,n\,;\mathtt{cmd}(\,\mathtt{ret}(\,\mathtt{triv}\,)\,)\,;n'\,.\,\mathtt{cmd}(\,m_{\text{emit}}\,)\,)\,)\,;10\,)\,)$$

Running this command given a channel $a \sim \mathtt{nat}$ spawns 10 asynchronous processes, each of which emits a number $1 \leq n \leq 10$ on channel a and immediately exits. Notice that we use a recursive function to compute a "large" command.

Task 1.2 (15 pts). Write a "consumer" m_{consume} that reads 10 natural numbers over channel a and returns their sum. You may use $e_1 + e_2$ to sum two natural numbers.

Running $\mathtt{bnd}(\,m_{\text{produce}}\,;u\,.\,m_{\text{consume}}\,)$ should evaluate to 55.

Hint. You may wish to model your solution on m_{produce}.

Remark. The producer-consumer pattern is common in concurrent programming.

2 Processes

Underlying **CA** is a sort of *procesess*, which concurrently communicate. Processes send and receive messages over channels, which we will denote a.

Proc	p	$::=$	$\texttt{stop}$	$\mathbf{1}$	nullary concurrent composition
			$\texttt{conc}(\, p_1 \,;\, p_2 \,)$	$p_1 \otimes p_2$	binary concurrent composition
			$\texttt{newch}[\,\tau\,](\, a \,.\, p \,)$	$\nu\, a \sim \tau \,.\, p$	new channel
			$\texttt{run}\langle a\rangle(\, m \,)$	$\texttt{run}\langle a\rangle(\, m \,)$	atomic
			$\texttt{send}\langle a\rangle(\, e \,)$	$!\, a(\, e \,)$	send on channel a
			$\texttt{recv}\langle a\rangle(\, x \,.\, p \,)$	$?\, a(\, x \,.\, p \,)$	receive on channel a
Action	α	$::=$	ε	ε	silent
			$\texttt{snd}\langle a\rangle(\, e \,)$	$a\,!\,e$	send
			$\texttt{rcv}\langle a\rangle(\, e \,)$	$a\,?\,e$	receive

Atomic Processes Atomic processes $\texttt{run}\langle a\rangle(\, m \,)$ contain commands m, which will be user-written programs; we will introduce commands in Section 1. It will be arranged in the dynamics that the channel a of an atomic process eventually receives the result of the command m; you may observe that channel a serves as a unique "process ID" for a given atomic process.

Remark. Processes and actions are not user-level constructs, like stacks k and states s in **KPCF**. Programmers will write commands m, *not* processes p.

2.1 Statics

The statics for processes are given in Appendix B.3.1.

Processes p are identified up to structural congruence, described in Appendix B.3.2. Of note, structural congruence states that:

- Concurrent composition operators $\mathbf{1}$ and $- \otimes -$ form a commutative monoid, allowing us to reorder concurrent composition of processes.
- New channels $\nu\, a \sim \tau \,.\, p$ can be hoisted to the top of a process.

For example, we have:

$$(\nu\, a_1 \sim \tau_1 \,.\, p_1) \otimes \mathbf{1} \otimes (?\, b(\, x \,.\, \nu\, a_2 \sim \tau_2 \,.\, p_2 \,)) \equiv \nu\, a_1 \sim \tau_1 \,.\, \nu\, a_2 \sim \tau_2 \,.\, p_1 \otimes ?\, b(\, x \,.\, p_2 \,)$$

2.2 Dynamics

The dynamics of processes are given via actions α; the judgment $p \overset{\alpha}{\underset{\Sigma}{\mapsto}} p'$ states that process p steps to p' with action α. The key rules are given below; we provide the rules for atomic processes in Appendix C.3.

$\boxed{p \overset{\alpha}{\underset{\Sigma}{\mapsto}} p'}$

$$\frac{p_1 \overset{\alpha}{\underset{\Sigma}{\mapsto}} p_1'}{p_1 \otimes p_2 \overset{\alpha}{\underset{\Sigma}{\mapsto}} p_1' \otimes p_2}(\mathrm{P}_1) \qquad \frac{p_1 \overset{a!e}{\underset{\Sigma, a \sim \tau}{\mapsto}} p_1' \qquad p_2 \overset{a?e}{\underset{\Sigma, a \sim \tau}{\mapsto}} p_2'}{p_1 \otimes p_2 \overset{\varepsilon}{\underset{\Sigma, a \sim \tau}{\mapsto}} p_1' \otimes p_2'}(\mathrm{P}_2) \qquad \frac{p \overset{\alpha}{\underset{\Sigma, a \sim \tau}{\mapsto}} p' \qquad \vdash_\Sigma \alpha \ \textsf{action}}{\nu\, a \sim \tau \,.\, p \overset{\alpha}{\underset{\Sigma}{\mapsto}} \nu\, a \sim \tau \,.\, p'}(\mathrm{P}_3)$$

$$\frac{}{!\, a(\, e\,) \overset{a!e}{\underset{\Sigma, a \sim \tau}{\mapsto}} \mathbf{1}}(\mathrm{P}_4) \qquad \frac{}{?\, a(\, x \,.\, p\,) \overset{a?e}{\underset{\Sigma, a \sim \tau}{\mapsto}} \{e/x\}p}(\mathrm{P}_5)$$

Task 2.1 (5 pts). Let $\Sigma = a \sim \texttt{nat}, b \sim \texttt{nat}, c \sim \texttt{nat}$. Provide a process p with $\vdash_\Sigma p\ \textsf{proc}$ such that if p receives a natural number n along channel a, it will send n along channels b and c.

Task 2.2 (10 pts). Consider the following process p:

$$!\, a(\, 1\,) \otimes !\, a(\, 2\,) \otimes ?\, a(\, x \,.\, !\, b(\, x\,)\,)$$

We have $\vdash_\Sigma p\ \textsf{proc}$, where $\Sigma = a \sim \texttt{nat}, b \sim \texttt{nat}$. Provide two processes, p_1' and p_2', such that $p \overset{\varepsilon}{\underset{\Sigma}{\mapsto}} p_1'$ and $p \overset{\varepsilon}{\underset{\Sigma}{\mapsto}} p_2'$ but it is *not* the case that $p_1' \equiv p_2'$. This shows that the dynamics of **CA** are nondeterministic!

2.3 Safety

Like the other languages we have considered in this course, **CA** satisfies progress and preservation theorems. Here, we will consider the progress theorem.

One might imagine that the progress theorem might say the following:

Theorem 2.1 (Faulty Progress). *If $\vdash_\Sigma p$ proc, then either $p \equiv \mathbf{1}$ or $p \overset{\varepsilon}{\underset{\Sigma}{\mapsto}} p'$ for some p'.*

This theorem holds for some processes p. For example:

$$!\, a(\, 312\,) \otimes ?\, a(\, x \,.\, \mathbf{1}\,) \overset{\varepsilon}{\underset{a \sim \texttt{nat}}{\mapsto}} \mathbf{1}$$

However, in general, this theorem is false!

Task 2.3 (10 pts). Provide a process p that is a counterexample to Theorem 2.1. Briefly (one sentence) justify why your choice of p is a counterexample. You should not use atomic processes $\texttt{run}\langle a\rangle(\, m\,)$ in your counterexample.

Hint. Note the ε action in Theorem 2.1.

Hint. Your counterexample can be very simple!

The true progress theorem involves non-ε actions:

Theorem 2.2 (Progress)**.** *If* $\vdash_\Sigma p$ proc*, then either* $p \equiv \mathbf{1}$ *or* $p \equiv \nu\,\Sigma'\{p'\}$ *such that* $p' \overset{\alpha}{\underset{\Sigma,\Sigma'}{\mapsto}} p''$ *for some* p'' *and some* $\vdash_{\Sigma,\Sigma'} \alpha$ action*.*

Recall the following statics rule from Appendix B.3.1:

$$\frac{}{\Gamma \vdash_\Sigma \mathbf{1}\ \mathsf{proc}}(\mathrm{SP}_1) \qquad \frac{\Gamma \vdash_\Sigma p_1\ \mathsf{proc} \qquad \Gamma \vdash_\Sigma p_2\ \mathsf{proc}}{\Gamma \vdash_\Sigma p_1 \otimes p_2\ \mathsf{proc}}(\mathrm{SP}_2) \qquad \frac{\Gamma \vdash_{\Sigma,a\sim\tau} e : \tau}{\Gamma \vdash_{\Sigma,a\sim\tau} !\,a(\,e\,)\ \mathsf{proc}}(\mathrm{SP}_5)$$

Task 2.4 (15 pts). Prove Theorem 2.2 for these three statics rules, using the dynamics given above and the structural congruence rules in Appendix B.3.2.

3 Implementing the Dynamics

In this section, you will implement parts of the dynamics for **CA**.

In order to work up to structural congruence of processes, we will use an "execution context" data structure to maintain a canonical form for processes, thus mediating communication. You should read `lang-ca/execution-context.sig` to understand how this data structure will behave.

Remark. You may attempt the following tasks in either order.

3.1 Dynamics

Task 3.1 (30 pts). Implement the remaining cases of `progress` in `lang-ca/dynamics-ca.fun`.

`progress (a, m)` transitions an atomic processes $\texttt{run}\langle a\rangle(m)$ to an execution context (i.e., a collection of atomic processes). In particular, `progress` will be based directly on rules R_1-R_7 from Appendix C.3, so it will *not* be recursive. The output in each case will be built via the operations provided by `structure EC`, where:

```
structure EC : EXECUTION_CONTEXT where type chan = CA.Chan.t
and type msg  = CA.Exp.t
```


Testing To test your implementation, move the reference solution heap image into `lang-ca/` and include your dynamics file and the updated top-level structure:


```
smlnj @SMLload lang-ca
- use "dynamics-ca.fun"; use "dynamics-ca.sml"; use
   "interpreter-ca.sml";
(* ... *)
- InterpreterCA.evalFile "tests.lam";
```


This will make use of the reference implementation of `EC`. You should see the following output:

```
hello
0
hello
fib  1 = 1
fib  2 = 1
fib 10 = 55
b  (* nondeterministic, could be c *)
abc
hello
```

You can also run:

```
- InterpreterCA.repl ();
-> load "tests.lam";
```

for a more verbose output, including printing the execution context. You may find it useful to change (or comment out parts of) `tests/tests.lam` when debugging.[1]

3.2 Execution Context

A process can be put into a *canonical form* containing:

- For each channel a, *either* a collection of surplus messages (of the form $!\,a(\,e\,)$) *or* a collection of identifiers for waiting processes. There will never be both a surplus and a deficit simultaneously: if a message and a waiter would be present on a channel, the waiter should be given the message and marked as ready.
- A map from waiter identifiers to true waiters (of the form $x\,.\,p$).
- A collection of ready processes, each of the form $\mathtt{run}\langle a\rangle(\,m\,)$.

We will call such a canonical form an *execution context* and represent all processes in this form. In this subsection, we will implement this data structure, using rules P_1-P_5 (transitions for non-atomic processes) from Section 2.2 and rules E_1-E_{13} (structural congruence) from Appendix B.3.2 to convert a process to normal form.

The signature in `lang-ca/execution-context.sig` details which operations should be supported by an execution context. The type parameter `'a` represents an abstract atomic process type; in the dynamics, it will be instantiated at `CA.Chan.t * CA.Cmd.t`.

All new channel identifiers $\nu\,\tau \sim a\,.\,p$ can be hoisted to the top via rules E_{12} and E_{13}. Therefore, in the code, we do *not* include an operation on execution contexts to bind a new channel; this is handled by `Chan.new`, and we may assume capture is never incurred.

Task 3.2 (40 pts). Implement `functor ExecutionContext` in `lang-ca/execution-context.fun` according to the specification in `lang-ca/execution-context.sig`. Comments in the file describe how you should understand the types.

Observe that `functor ExecutionContext` is parameterized on arbitrary channel and message types. Thus, your implementation will not involve any **CA**-specific constructs.

Additionally, it is parameterized on a queue data structure, which you should use internally to queue messages, waiters, and ready processes. To simulate nondeterminism, the provided implementation may be randomized.

Testing To test your execution context independently, you can run `TestHarness.testEC`. By default, `TestHarness` introduces random dequeueing, so you may notice that tests fail only sometimes. If you wish to test deterministically, you can use structure `Queue (val random = false)` in `tests/tests.sml`. In the same file, you can read the test cases to aid with debugging (using `TestHarness.testEC true` for verbose test output).

[1]In `lang-ca/interpreter-ca.fun`, the function `evalCmd` builds the initial processes. In `lang-ca/process-executor.sml`, the function `run` contains the entry point for the dynamics.

You should now be able to run the aforementioned **CA** test cases using your execution context, as well, by loading `sources.cm`.

4 References

Next, we will implement free assignable references in Concurrent Algol.

4.1 Reference Cells

Task 4.1 (35 pts). In `tests/refs.lam`, implement the signature of three functions that define ref cells of natural numbers, a type named `ref`:

- `newref`, when given an initial value, should return a ref containing that value.
- `deref`, when given a ref cell, should return the contents of the cell, such that the cell may be read again to obtain the same value in the future until the cell is reset.
- `set`, when given a ref cell and a new value, should update the contents of the cell such that future dereferences return the new value until the cell is reset again.

You are given the test file `tests/refs-test.lam` to verify your implementation. Feel free to write more tests of your own.

To help you get started, first take a look at `tests/tests.lam` and `tests/tests2.lam`, which give some examples of using the CA concrete syntax. If you would like a complete, formal presentation of the concrete syntax, be sure to look at Section A.

Next, please check out `tests/prng.lam` and `tests/prng-test.lam`. This is an implementation of a pseudorandom number generator (PRNG), specifically one called Blum Blum Shub. It defines a type named `prng` with two functions:

- `newprng`, when given a modulus and seed value (both natural numbers), should return a prng initialized with those values
- `next`, when given a prng, should output the value stored within the prng as the next pseudorandom number, then update the prng's value to be the square of the previous value mod the modulus

The key takeaways from the PRNG example are:

- how it simulates "memory" or "state" using the `server` process
- how `next` communicates back and forth with the `server` process using channel-passing

Do familiarize yourself with the PRNG example, as your `ref` implementation will likely use similar strategies and code constructs.

4.2 Atomic References

The mutable references we implemented work well if we agree to be careful with concurrent reads and writes. If we try to do many reads and writes concurrently, though, we can encounter race conditions!

For example, consider incrementing a reference cell. Normally, it requires a read from the cell, then a write of the incremented value back to the cell. If there is only one thread performing the increment operation, everything will proceed as planned.

However, what if multiple threads try to increment a reference simultaneously? The first thread may read from the cell, followed immediately by the second thread. Then, both threads write back the value they saw plus one—the same value for both threads. One of the increments got lost, and we have a *race condition*, meaning the increment operation is not *thread-safe*.

Your next task is to make the references thread-safe by providing *atomic* operations that cannot be interrupted in such ways.

Task 4.2 (35 pts). In `tests/atomic.lam`, implement three operations on a new type `atomic_ref`:

- `new_atomic_ref`, when given a reference, wraps it to make it atomic.
- `incr`, which performs an atomic increment of the cell: no two threads performing a concurrent `incr` operation may interfere with each other.
- `cmpxchg`, which performs an atomic compare-and-exchange: given a ref cell, a value `expected`, and a value `new`, if the cell currently has the value `expected`, update it to have value `new` and return true. Otherwise, make no change to the contents and return false. The entire operation must happen atomically.

In particular, `incr` and `cmpxchg` must be atomic *with respect to each other*, meaning that concurrent invocations of either function must not interfere; the result should always look as if the threads had performed their actions sequentially instead.

Hint. You may find the notion of a *lock* useful, though it is not required. A lock is an "object" that only one process may "have" at a time; it can be used to guarantee that even given multiple concurrent threads, only one thread will perform some sensitive operation at a time (such as accessing a ref cell).

It does so through two operations:

- *acquire* which is called to request the lock and stall the thread until it is safe to proceed, and
- *release* which is called afterward, allowing another thread to now proceed.

Hint. A lock may be implemented easily using the features of Concurrent Algol. Think about how you might use a channel to hold an "object" that only one process can have at a time.

To test your code, you may run the test file `tests/atomic-test.lam`.

A Syntax

Here, we outline the concrete syntax of **CA**.

Sort			Abstract Syntax	Concrete Syntax	Description
Typ	τ	::=	$\mathtt{unit}$	`unit`	unit type
			$\mathtt{void}$	`void`	void type
			$\mathtt{bool}$	`bool`	bool type
			$\mathtt{nat}$	`nat`	natural number
			$\mathtt{sum}(\tau_1 ; \tau_2)$	`tau1 + tau2`	sum type
			$\mathtt{prod}(\tau_1 ; \tau_2)$	`tau1 * tau2`	product type
			$\mathtt{arr}(\tau_1 ; \tau_2)$	`tau1 -> tau2`	function type
			$\mathtt{cmd}(\tau)$	`cmd[tau]`	command
			$\mathtt{chan}(\tau)$	`chan[tau]`	channel
Exp	e	::=	$\mathtt{triv}$	`()`	unit
			$\mathtt{in}\langle\mathtt{l}\rangle[\tau_1 ; \tau_2](e)$	`inl[tau1 + tau2] e`	left injection
			$\mathtt{in}\langle\mathtt{r}\rangle[\tau_1 ; \tau_2](e)$	`inr[tau1 + tau2] e`	right injection
			$\mathtt{case}(e ; x_1 . e_1 ; x_2 . e_2)$	`case e of {x1 => e1 | x2 => e2}`	case expression
			$\mathtt{pair}(e_1 ; e_2)$	`(e1, e2)`	pair/tuple
			$\mathtt{split}(e ; x_1, x_2 . e')$	`split e is x1, x2 in e'`	split
			$\lambda[x](\tau . e)$	`fn (x : tau)=> e`	lambda expression
			$\mathtt{fun}[\tau_1 ; \tau_2](f . x . e)$	`fun f (x : tau1) : tau2 = e`	function
			$\mathtt{if}(e ; e_1 ; e_2)$	`if e then e1 else e2`	conditional
			$\mathtt{ifz}[\tau](e ; e_1 ; x . e_2)$	`ifz e then e1 else x => e2`	ifz
			$\mathtt{let}(e ; x . e_1)$	`let val x = e in e1 end`	let
			$\mathtt{cmd}[\tau](m)$	`cmd m`	encapsulation
			$\mathtt{chref}\langle a\rangle$	`chan[a]`	channel reference
Cmd	m	::=	$\mathtt{ret}(e)$	`ret(e)`	return
			$\mathtt{bnd}(e ; x . m)$	`val x = e in m`	sequence
			$\mathtt{spawn}(e)$	`spawn(e)`	spawn process
			$\mathtt{emitref}(e_1 ; e_2)$	`emit(e1, e2)`	send message on channel
			$\mathtt{sync}(e)$	`sync(e)`	receive message on channel
			$\mathtt{newchan}\langle\tau\rangle(a . m)$	`newchan x ~ tau in c`	new channel

We also provide various constant forms, such as `true`, `false`, `zero`, `succ e`.

Also take note of various pieces of (very helpful) syntactic sugar for writing commands:

- `emit[c](e)` is sugar for `emit(chan[c], e)`
- `sync[c]` is sugar for `sync(chan[c])`
- `do e` is sugar for `val x = e in ret(x)`

- `{c1,...,cn}` is sugar for `val _ = cmd(c1)in ... in cn`

```
directive ::= type t = tau | fun f (x1 : tau1) ... (xn : taun) : tau = term
  | val x = term | m | c | load f
decl d ::= fun f (x1 : tau1) ... (xn : taun) : tau = term | val x = term
type tau ::= t | tau1 + tau2 | tau1 * tau2 | tau1 -> tau2 | rec(t.tau)
  | unit | void | nat | bool | cmd[tau] | chan[tau]
cmd c ::= ret(e) | val x = e in c | spawn(e) | emit(e1,e2) | emit[k](e)
  | sync(e) | sync[k] | newchan x ~ tau in c | print e | do e | {c1,...,cn}
atom a ::= (e) | n | '...' | true | false | zero | x | () | (e1,e2)
  | case e of {x1 => e1 | x2 => e2} | let d1...dn in e end
  | chan[k]
tree s ::= a1 ... an | s1 + s2 | s1 - s2 | s1 * s2 | s1 / s2 | s1 = s2
  | s1 != s2 | s1 < s2 | s1 <= s2 | s1 > s2 | s1 >= s2 | s1 && s2 | s1 || s2
term e ::= s
  | fn x : tau => e | fn(x : tau) => e | split e is x1, x2 in e' | inl[tau]e | inr[tau]e
  | fold[tau] e | unfold e | if e then e1 else e2
  | succ e | ifz e then e1 else x => e2 | abort[tau] e | cmd c | !e
```

The parser will delineate directives by ";". See `tests/tests.lam` for some examples.

B Statics

B.1 Expressions

$\boxed{\Gamma \vdash_\Sigma e : \tau}$

$$\frac{\Gamma \vdash_\Sigma m \mathrel{\dot\sim} \tau}{\Gamma \vdash_\Sigma \texttt{cmd}[\,\tau\,](\,m\,) : \texttt{cmd}(\,\tau\,)}(\text{SE}_1) \qquad \frac{}{\Gamma \vdash_{\Sigma,a\sim\tau} \texttt{chref}\langle a\rangle : \texttt{chan}(\,\tau\,)}(\text{SE}_2)$$

B.2 Commands

$\boxed{\Gamma \vdash_\Sigma m \mathrel{\dot\sim} \tau}$

$$\frac{\Gamma \vdash_\Sigma e : \tau}{\Gamma \vdash_\Sigma \texttt{ret}(\,e\,) \mathrel{\dot\sim} \tau}(\text{SM}_1) \qquad \frac{\Gamma \vdash_\Sigma e : \texttt{cmd}(\,\tau\,) \qquad \Gamma, x : \tau \vdash_\Sigma m \mathrel{\dot\sim} \tau'}{\Gamma \vdash_\Sigma \texttt{bnd}(\,e\,;x\,.\,m\,) \mathrel{\dot\sim} \tau'}(\text{SM}_2)$$

$$\frac{\Gamma \vdash_\Sigma e : \texttt{cmd}(\,\tau\,)}{\Gamma \vdash_\Sigma \texttt{spawn}(\,e\,) \mathrel{\dot\sim} \texttt{chan}(\,\tau\,)}(\text{SM}_3) \qquad \frac{\Gamma \vdash_\Sigma e_1 : \texttt{chan}(\,\tau\,) \qquad \Gamma \vdash_\Sigma e_2 : \tau}{\Gamma \vdash_\Sigma \texttt{emitref}(\,e_1\,;e_2\,) \mathrel{\dot\sim} \texttt{unit}}(\text{SM}_4)$$

$$\frac{\Gamma \vdash_\Sigma e : \texttt{chan}(\tau)}{\Gamma \vdash_\Sigma \texttt{sync}(\,e\,) \mathrel{\dot\sim} \tau}(\text{SM}_5) \qquad \frac{\Gamma \vdash_{\Sigma,a\sim\tau} m \mathrel{\dot\sim} \tau'}{\Gamma \vdash_\Sigma \texttt{newchan}\langle\tau\rangle(\,a\,.\,m\,) \mathrel{\dot\sim} \tau'}(\text{SM}_6)$$

B.3 Processes

B.3.1 Typing

The judgment $\Gamma \vdash_\Sigma p$ proc describes the valid processes relative to the signature Σ and context Γ. Processes have to be typed under a specific context due to the existence of accepting processes.

$\boxed{\Gamma \vdash_\Sigma p \ \mathsf{proc}}$

$$\frac{}{\Gamma \vdash_\Sigma \mathbf{1}\ \mathsf{proc}}(\text{SP}_1) \qquad \frac{\Gamma \vdash_\Sigma p_1\ \mathsf{proc} \qquad \Gamma \vdash_\Sigma p_2\ \mathsf{proc}}{\Gamma \vdash_\Sigma p_1 \otimes p_2\ \mathsf{proc}}(\text{SP}_2) \qquad \frac{\Gamma \vdash_{\Sigma,a\sim\tau} p\ \mathsf{proc}}{\Gamma \vdash_\Sigma \nu\, a \sim \tau\,.\,p\ \mathsf{proc}}(\text{SP}_3)$$

$$\frac{\Gamma \vdash_{\Sigma,a\sim\tau} m \mathrel{\dot\sim} \tau}{\Gamma \vdash_{\Sigma,a\sim\tau} \texttt{run}\langle a\rangle(\,m\,)\ \mathsf{proc}}(\text{SP}_4) \qquad \frac{\Gamma \vdash_{\Sigma,a\sim\tau} e : \tau}{\Gamma \vdash_{\Sigma,a\sim\tau}\ !\,a(\,e\,)\ \mathsf{proc}}(\text{SP}_5) \qquad \frac{\Gamma, x : \tau \vdash_{\Sigma,a\sim\tau} p\ \mathsf{proc}}{\Gamma \vdash_{\Sigma,a\sim\tau}\ ?\,a(\,x\,.\,p\,)\ \mathsf{proc}}(\text{SP}_6)$$

B.3.2 Structural Congruence

Processes are identified up to structural congruence, an equivalence relation written $p_1 \equiv p_2$.

$\boxed{p_1 \equiv p_2}$

First, we state that $p_1 \equiv p_2$ is an equivalence relation (reflexive, symmetric, and transitive):

$$\frac{p_1 =_\alpha p_2}{p_1 \equiv p_2}(\text{E}_1) \qquad \frac{p_2 \equiv p_1}{p_1 \equiv p_2}(\text{E}_2) \qquad \frac{p_1 \equiv p_2 \qquad p_2 \equiv p_3}{p_1 \equiv p_3}(\text{E}_3)$$

Then, we state that $p_1 \equiv p_2$ is a congruence:

$$\frac{p_1 \equiv p_1' \qquad p_2 \equiv p_2'}{p_1 \otimes p_2 \equiv p_1' \otimes p_2'}(\mathrm{E}_4) \qquad \frac{p \equiv p'}{\nu\, a \sim \tau \,.\, p \equiv \nu\, a \sim \tau \,.\, p'}(\mathrm{E}_5) \qquad \frac{p \equiv p'}{?\, a(\, x \,.\, p\,) \equiv ?\, a(\, x \,.\, p'\,)}(\mathrm{E}_6)$$

We guarantee that $\mathbf{1}$ and $- \otimes -$ form a commutative monoid:

$$\frac{}{p_1 \otimes (p_2 \otimes p_3) \equiv (p_1 \otimes p_2) \otimes p_3}(\mathrm{E}_7) \qquad \frac{}{p \otimes \mathbf{1} \equiv p}(\mathrm{E}_8) \qquad \frac{}{p_1 \otimes p_2 \equiv p_2 \otimes p_1}(\mathrm{E}_9)$$

We allow new channels to be reordered, removed if unused, and hoisted to the top level provided they do not incur capture. Rule E_{12} is called *scope extrusion*, since it allows the scope of a to be extruded out of the concurrent composition.

$$\frac{a_1 \neq a_2}{\nu\, a_1 \sim \tau_1 \,.\, \nu\, a_2 \sim \tau_2 \,.\, p \equiv \nu\, a_2 \sim \tau_2 \,.\, \nu\, a_1 \sim \tau_1 \,.\, p}(\mathrm{E}_{10}) \qquad \frac{a \notin p}{\nu\, a \sim \tau \,.\, p \equiv p}(\mathrm{E}_{11})$$

$$\frac{a \notin p_2}{(\nu\, a \sim \tau \,.\, p_1) \otimes p_2 \equiv \nu\, a \sim \tau \,.\, (p_1 \otimes p_2)}(\mathrm{E}_{12}) \qquad \frac{a_1 \neq a_2}{?\, a_2(\, x \,.\, \nu\, a_1 \sim \tau \,.\, p\,) \equiv \nu\, a_1 \sim \tau \,.\, ?\, a_2(\, x \,.\, p\,)}(\mathrm{E}_{13})$$

B.4 Actions

Actions also admit a statics, given by the judgment $\vdash_\Sigma \alpha$ action.

$\boxed{\vdash_\Sigma \alpha \ \mathsf{action}}$

$$\frac{}{\vdash_\Sigma \varepsilon \ \mathsf{action}}(\mathrm{A}_1) \qquad \frac{\vdash_{\Sigma, a \sim \tau} e : \tau}{\vdash_{\Sigma, a \sim \tau} a\,!\,e \ \mathsf{action}}(\mathrm{A}_2) \qquad \frac{\vdash_{\Sigma, a \sim \tau} e : \tau}{\vdash_{\Sigma, a \sim \tau} a\,?\,e \ \mathsf{action}}(\mathrm{A}_3)$$

C Dynamics

C.1 Expressions

$\boxed{e\ \mathsf{val}_{\Sigma}}$

$$\frac{}{\texttt{chref}\langle a\rangle\ \mathsf{val}_{\Sigma, a\sim\tau}}(\mathrm{V}_1) \qquad \frac{}{\texttt{cmd}[\,\tau\,](\,m\,)\ \mathsf{val}_{\Sigma}}(\mathrm{V}_2)$$

C.2 Commands

$\boxed{m \underset{\Sigma}{\Rightarrow} m'}$

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\texttt{ret}(\,e\,) \underset{\Sigma}{\Rightarrow} \texttt{ret}(\,e'\,)}(\mathrm{M}_1) \qquad \frac{e \underset{\Sigma}{\longmapsto} e'}{\texttt{bnd}(\,e\,;x\,.\,m\,) \underset{\Sigma}{\Rightarrow} \texttt{bnd}(\,e'\,;x\,.\,m\,)}(\mathrm{M}_2)$$

$$\frac{e \underset{\Sigma}{\longmapsto} e'}{\texttt{spawn}(\,e\,) \underset{\Sigma}{\Rightarrow} \texttt{spawn}(\,e'\,)}(\mathrm{M}_3) \qquad \frac{e_1 \underset{\Sigma}{\longmapsto} e_1'}{\texttt{emitref}(\,e_1\,;e_2\,) \underset{\Sigma}{\Rightarrow} \texttt{emitref}(\,e_1'\,;e_2\,)}(\mathrm{M}_4)$$

$$\frac{e_1\ \mathsf{val}_{\Sigma} \qquad e_2 \underset{\Sigma}{\longmapsto} e_2'}{\texttt{emitref}(\,e_1\,;e_2\,) \underset{\Sigma}{\Rightarrow} \texttt{emitref}(\,e_1\,;e_2'\,)}(\mathrm{M}_5) \qquad \frac{e \underset{\Sigma}{\longmapsto} e'}{\texttt{sync}(\,e\,) \underset{\Sigma}{\Rightarrow} \texttt{sync}(\,e'\,)}(\mathrm{M}_6)$$

C.3 Processes

In this formulation of Concurrent Algol, there is no command level transitions for the effects. It does **not** follow that our language is free of effects, instead all effects have been lifted into process level and carried out at process level. The rules for transitioning processes that does not involve lifting effects from commands are given in Section 2.2.

Rule P_1, together with congruence, allows for independent transition of both concurrent sub-processes. P_2 enables communications between concurrent processes.

Finally, we may now consider the actions and effects of the commands:

$\boxed{p \overset{\alpha}{\underset{\Sigma}{\mapsto}} p'\text{, continued}}$

$$\frac{m \underset{\Sigma}{\Rightarrow} m'}{\mathtt{run}\langle a\rangle(\,m\,) \overset{\varepsilon}{\underset{\Sigma}{\mapsto}} \mathtt{run}\langle a\rangle(\,m'\,)}\,(\mathrm{R}_1) \qquad \frac{e\ \mathsf{val}_{\Sigma,a\sim\tau}}{\mathtt{run}\langle a\rangle(\,\mathtt{ret}(\,e\,)\,) \overset{\varepsilon}{\underset{\Sigma,a\sim\tau}{\mapsto}} !\,a(\,e\,)}\,(\mathrm{R}_2)$$

$$\frac{}{\mathtt{run}\langle a\rangle(\,\mathtt{bnd}(\,\mathtt{cmd}[\,\tau'\,](\,m_1\,)\,;x\,.\,m_2\,)\,) \overset{\varepsilon}{\underset{\Sigma,a\sim\tau}{\mapsto}} \nu\, b \sim \tau'\,.\,\mathtt{run}\langle b\rangle(\,m_1\,) \otimes ?\,b(\,x\,.\,\mathtt{run}\langle a\rangle(\,m_2\,)\,)}\,(\mathrm{R}_3)$$

$$\frac{}{\mathtt{run}\langle a\rangle(\,\mathtt{spawn}(\,\mathtt{cmd}[\,\tau\,](\,m\,)\,)\,) \overset{\varepsilon}{\underset{\Sigma,a\sim\mathtt{chan}(\,\tau\,)}{\mapsto}} \nu\, b \sim \tau\,.\,\mathtt{run}\langle b\rangle(\,m\,) \otimes \mathtt{run}\langle a\rangle(\,\mathtt{ret}(\,\mathtt{chref}\langle b\rangle\,)\,)}\,(\mathrm{R}_4)$$

$$\frac{e\ \mathsf{val}_{\Sigma,a\sim\mathtt{unit},b\sim\tau}}{\mathtt{run}\langle a\rangle(\,\mathtt{emitref}(\,\mathtt{chref}\langle b\rangle\,;e\,)\,) \overset{\varepsilon}{\underset{\Sigma,a\sim\mathtt{unit},b\sim\tau}{\mapsto}} !\,b(\,e\,) \otimes \mathtt{run}\langle a\rangle(\,\mathtt{ret}(\,\langle\rangle\,)\,)}\,(\mathrm{R}_5)$$

$$\frac{}{\mathtt{run}\langle a\rangle(\,\mathtt{sync}(\,\mathtt{chref}\langle b\rangle\,)\,) \overset{\varepsilon}{\underset{\Sigma,a\sim\tau,b\sim\tau}{\mapsto}} ?\,b(\,x\,.\,\mathtt{run}\langle a\rangle(\,\mathtt{ret}(\,x\,)\,)\,)}\,(\mathrm{R}_6)$$

$$\frac{}{\mathtt{run}\langle a\rangle(\,\mathtt{newchan}\langle\tau\rangle(\,b\,.\,m\,)\,) \overset{\varepsilon}{\underset{\Sigma,a\sim\tau}{\mapsto}} \nu\, b \sim \tau\,.\,\mathtt{run}\langle a\rangle(\,m\,)}\,(\mathrm{R}_7)$$

In our setup, all command level effects have been elevated to the process level. In particular, a sequence of two commands is achieved by spawning a process for each of the commands, and have the second process wait for the result from the first one on a secret channel. Type annotation makes it possible to figure out the type τ and τ' in R_4 and R_3 respectively. Notice that the emit commands generates new processes. This should remind you that we are implementing asynchronous sends.