1 Instructions

Begin by downloading the file hw9-base.ml from the course website and renaming it to hw9.ml. This file contains the functions that you will use and modify in the homework. Submit your completed hw9.ml via Gradescope. As always, please don’t hesitate to ask for help on Piazza (https://piazza.com/class/jkh8q52qrh06v).

2 Simulating Concurrent Programs

1. (5 points) The rule for a completed parallel composition is:

\[(\text{skip} \parallel \text{skip}, \sigma) \rightarrow (\text{skip}, \sigma)\]

Add a case to step_cmd that implements this rule.

Once you have completed this problem, run_prog prog1 should return (Skip, σ₁), where σ₁ is the empty state. To test this, you can use pattern-matching or snd to extract the state, and then look up variables in the state to make sure that none of them have values. For instance, you should be able to reproduce the following interaction:

```ocaml
# let res1 = run_prog prog1;;
val res1 : config = (Skip, <fun>)
# let state1 = snd res1;;
val state1 : state = <fun>
# state1 "x";;
- : value option = None
# state1 "y";;
- : value option = None
```

2. (10 points) The remaining rules for parallel composition are:

\[
\begin{align*}
(c_1, \sigma) \rightarrow (c'_1, \sigma') & \quad & (c_2, \sigma) \rightarrow (c'_2, \sigma') \\
(c_1 \parallel c_2, \sigma) \rightarrow (c'_1 \parallel c_2, \sigma') & \quad & (c_1 \parallel c_2, \sigma) \rightarrow (c_1 \parallel c'_2, \sigma')
\end{align*}
\]
Add a case to `step_cmd` that implements these rules. It should be a separate case from the one you added in problem 1, and should come after problem 1’s case, so that the skip case takes priority. Because these rules are nondeterministic, you can use any approach to decide whether to step \( c_1 \) or \( c_2 \), but you should only return `None` if both sides are unable to step. Here is one fairly simple approach:

- If both \( c_1 \) and \( c_2 \) can successfully step, randomly choose one of the two rules to apply. For instance, you can call `Random.int 2` to get an `int` that is randomly either 0 or 1, and then choose which rule to apply based on that int.
- If only one of \( c_1 \) and \( c_2 \) can step, apply the rule for the one that can step.
- If neither \( c_1 \) nor \( c_2 \) can step, return `None`.

Once you have completed this problem, `run_prog prog2` should return \((\text{Skip}, \sigma_2)\), where \( \sigma_2 \) is a state in which each of \( x \) and \( y \) is either 1 or 2 (the exact values will depend on the order in which the sides executed).

3. (5 points) The rules for locks are:

\[
\begin{align*}
\sigma(x) &= 0 \\
(\text{acquire}(x), \sigma) &\rightarrow (\text{skip}, \sigma[x \mapsto 1])
\end{align*}
\]

\[
\begin{align*}
\sigma(x) &= 1 \\
(\text{release}(x), \sigma) &\rightarrow (\text{skip}, \sigma[x \mapsto 0])
\end{align*}
\]

Add cases to `step_cmd` that implement these rules.

Once you have completed this problem, `run_prog prog3` should return \((\text{Skip}, \sigma_3)\), where \( \sigma_3 \) is a state in which either \( x \) is 1 and \( y \) is 2, or \( x \) is 2 and \( y \) is 1. The state may be different each time, but you should never see a result in which \( x \) and \( y \) have the same value.