HW9 – Concurrency

CS 476, Fall 2018
Due Dec. 5 at 2 PM

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 Evaluating Concurrent Programs

The file `hw9-base.ml` defines expressions, commands, and single-threaded semantics for a simple imperative language with fork-join concurrency. It has a few changes from the language we've worked with so far:

- It has global variables, which are accessed with the `Global` expression and modified with the `SetGlobal` command. At runtime, the globals are stored in a separate environment `g`, the global environment (of type `env`).
- There are no `Seq` or `Skip` commands. Instead, the body of a function and the first element of a configuration are both of type `cmd list`: a list `[`c_1`;`c_2`;...;`c_n`]` represents the commands c_1, c_2, ..., c_n executed in sequence, and an empty list represents code that is finished executing (analogous to `Skip`).

The file declares functions `eval_exp`, `eval_exps`, and `step_cmd1` for executing expressions, lists of expressions, and single threads respectively. There are also functions `run_prog` and `run_prog_n` for running example programs: `run_prog fs gs cs` gives the result of running the list of commands `cs` with defined functions `fs` and initial global environment `gs`, and `run_prog_n` takes an additional `int` argument `n` and runs the program for only `n` steps (which is useful for testing programs that would otherwise run forever). To test a program `test_prog` with functions `funs0` and initial global environment `globals0`, you could run

```
let (_, res) = run_prog funs0 globals0 test_prog;;
```

and then inspect the values of the global variables in `res`, as in HW5.

The state of an executing thread is a tuple of the form (i, c, k, r) where i is a thread id (tid), c is the list of commands being executed by the thread, k is the thread's stack, and r is the environment for the thread's local variables. The state of the whole program is a pair of a list of thread states and a global environment. The goal of this assignment is to write the function `step_cmd` that steps a concurrent program. The arguments to `step_cmd` are `funs`, the collection of function declarations; `threads`, the list of threads that are not currently being stepped; and `i`, `lc`, `k`, `r`, the tid, commands, stack, and local environment of the currently executing thread; and `g`, the global environment.

The semantic rules for concurrency allow us to nondeterministically choose any thread to execute at each step, but for this interpreter we will use *round-robin scheduling*: we rotate through the list of threads, stepping each one once and then moving it to the end of the list. If a thread cannot currently take a step (for instance, because it's waiting to join with a thread that has not yet terminated), it will be moved to the end of the list so that other threads can step. In `step_cmd`, this should be implemented by adding the resulting thread to the end of the list `threads`. The predefined `fail_config` shows how to add a thread to the end of the list, and should be used in any case where the current thread fails to step.

3 Problems

1. (5 points) The following "sequential step" rule allows a thread to execute any command that doesn't interact with other threads:

$$\frac{(c, k, \rho, g) \to (c', k', \rho', g')}{([...;(i, c, k, \rho);...], g) \to ([...;(i, c', k', \rho');...], g')}$$

   Implement this case of `step_cmd`, by adding a code to the wildcard case `_`. Remember that `step_cmd1` implements the single-thread step relation.

   Once you have completed this problem, `run_prog funs0 globals0 test_seq` should return a state with the global variable `x` set to 1 and `y` set to 2.

2. (9 points) Extend `step_cmd` to handle the `Fork` command, according to the following rule:

$$\frac{\texttt{funs}\ f = \texttt{FDecl}\ (...,...,[],c) \qquad i' \text{ fresh}}{([...;(i, \texttt{Fork}\ (x, f) :: cs, k, \rho);...], g) \to ([...;(i, cs, k, \rho[x \mapsto i']);(i', c, [], \texttt{empty\_state});...], g)}$$

   You can use the function `fresh_tid : unit -> tid` to make a fresh tid.

   Once you have completed this problem, `run_prog funs0 globals0 test_con` will still run forever, but `run_prog_n 3 funs0 globals0 test_con` should return a state in which `x` is 2.

3. (4 points) In order to implement the rule for `Join`, you will need to search the list of threads for a specific thread id and check whether it has terminated. Write a

function `terminated : tid -> thread -> bool` such that `terminated i t` returns true when all of the following conditions hold: the thread id of `t` is `i`, the stack of `t` is empty, and the first command in the command list of `t` is a `Return` command. (You can ignore the value being returned.)

As a test, `terminated 1 (1, [Return (Int 0)], [], empty_state)` should be true, while `terminated 1 (2, [Return (Int 0)], [], empty_state)` and `terminated 1 (1, [Assign ("x", Int 4); Return (Int 0)], [], empty_state)` should both be false.

4. (7 points) Extend `step_cmd` to handle the `Join` command, according to the following rule:

$$\frac{(e, \rho, g) \Downarrow j}{\begin{array}{c}([...; (i, \texttt{Join}\ e :: cs, k, \rho); ...; (j, \texttt{Return}\ \_ :: ..., [], ...); ...], g) \rightarrow \\ ([...; (i, cs, k, \rho); ...; (j, \texttt{Return}\ \_ :: ..., [], ...); ...], g)\end{array}}$$

For simplicity's sake, the rule does not remove the terminated thread, but simply checks that it exists in the thread list. You can use the built-in `find_opt` function, in combination with the `terminated` function from the previous problem, to find the thread j, or you can write your own search function for it.

Once you have completed this problem, `run_prog funs0 globals0 test_con` should terminate and return a state in which `x` is 2.