HW4 – Interpreting Functions

CS 476, Fall 2023

1 Instructions

Begin by downloading the file `hw4-base.ml` from the course website, and renaming it to `hw4.ml`. Then fill in your answers to the problems, adding or modifying definitions as you see fit. Submit your completed `hw4.ml` via Gradescope. As always, please don't hesitate to ask for help on Piazza (`https://piazza.com/class/lkwp62qwo734i9`).

2 Adding Functions to the Interpreter

The file `hw4-base.ml` defines the types `exp` of expressions and `cmd` of commands. It also defines two main functions: `eval_exp`, a big-step-style interpreter for expressions, and `step_cmd`, a small-step-style interpreter for commands.

An `env` is a map from identifiers to the `entry` type, which can be either `Val` of a value (an `IntVal`/`BoolVal`) or a function definition `Fun`, which contains the list of parameter names and the function body. For instance, if `lookup r "f"` returns `Some (Fun (["x"; "y"], Return (Var "x")))`, this means that in the environment `r`, the function `f` is defined as `f(x, y){ return x }`.

The function `eval_exp : exp -> env -> value option` takes an expression e and an environment r and returns a value option: either `Some` v, if e evaluates to v, or `None`, if e fails to evaluate. The function `step_cmd : config -> config option` takes a `config`, a configuration of the form (c, k, r) where c is a `cmd`, k is a `stack`, and r is an `env`, and returns a configuration option:

- `Some` (c', k', r'), if $(c, k, r) \to (c', k', r')$
- `None`, if there is no step that (c, k, r) can take

The `cmd` type already includes constructors for function calls and returns. `Call` takes two identifiers, representing the variable and function name, and a list of expressions, representing the arguments. For example, `Call ("x", "f", [Num 1, Num 2])` represents the command `x := f(1, 2)`. `Return` takes one expression, which computes the return value of the function. Your job is to extend the `step_cmd` function to implement these commands.

1. (3 points) Define an object `my_prog : cmd` that represents the program `x := 5`. Confirm that when you run it, it returns an environment where `x` is 5. You can

confirm that using the `run_config` function, which takes a `config` and applies `step_cmd` to the `config` for as many steps as possible. For instance, if you write

```
let test1 = run_config (my_prog, [], empty_env);;
```

then the variable `test1` will hold the results of running `my_prog` starting with an empty environment (no variables defined) and an empty stack. The return value of `run_config` will be a `config`; you can access its components with code like

```
let (res_c, res_k, res_r) = test1;;
```

to store the resulting command, stack, and environment in variables `res_c`, `res_k`, and `res_r` respectively. You can then look up variables in `res_r` to see whether the right environment was produced:

```
lookup res_r "x";;
- : entry option = Some (Val (IntVal 5))
```

If the call to `run_config` runs forever, you can run the program step by step manually by calling `step_cmd` on your starting configuration, then calling `step_cmd` on the result, etc., and see where it gets stuck.

2. (4 points) Add a case to `step_cmd` for the return statement, according to the following rule:

$$\frac{(e, \rho) \Downarrow v}{(\texttt{return}\ e, (\rho_0, x) :: k, \rho) \rightarrow (\texttt{skip}, k, \rho_0[x \mapsto v])}$$

Test your new case by running `run_config` with a `return` command and a non-empty stack, and confirm that the stack gets popped and the return value is stored in the variable from the popped stack frame, as demonstrated in `ret_test1` in the sample code.

3. (8 points) Add a case to `step_cmd` for the call statement, according to the following rule:

$$\frac{([e_1; ...; e_n], \rho) \Downarrow [v_1; ...; v_n] \qquad \rho(f) = ((x_1, ..., x_n), c)}{(x\ \texttt{:=}\ f(e_1, ..., e_n), k, \rho) \rightarrow (c, (\rho, x) :: k, \rho[x_1 \mapsto v_1; ...; x_n \mapsto v_n])}$$

A function `add_args` has been provided that takes an env r, a list of variables, and a list of values, and returns the new env $r[x_1 \mapsto v_1; ...; x_n \mapsto v_n]$. There is also a function `eval_exps` that takes a list of expressions and returns the list of their values.

You can test your code with `run_config`, or use `run_prog`, which runs an entire program starting from an initial environment. If you have correctly defined all commands, then `run_prog prog1 env0` should result in the command `Skip`, the stack `[]`, and an environment in which `x` is 3 and `y` is undefined (i.e., `None`).