Structure of Programming Languages, Assignment 4

Due: Wednesday, Nov 22

This homework is organized differently from previous ones in that it contains two separate parts. Part I is mostly theoretical and involves completing the operational semantics definition for a new language and proving the progress and preservation lemmas.

You can select to do either Part I or Part II (you do not have to do both, but can elect to do so). If you do both parts, you can elect to replace the score for a past homework with the score for the optional part in this homework.

If you elect to completely only Part II, you need the answer to problem 1 in Part I. Email me and I will send it to you. Note that this means that you will not be able to get points for problem 1 if you later change your mind and decide to do Part I, too.

Note: In this assignment you need to pair up with another student. Submit a single solution, clearly indicating the names of both students. You can work alone if you prefer, just indicate in your submission that it was individual work. If you are working with someone else, remember to add `hw4/team.txt` file with both students' names in both students' Bitbucket repositories.

Part I: Mutable Heap

The first part of this assignment investigates adding a mutable heap to the simply-typed lambda-calculus. We did not do this in lecture, so we start with a syntax, small-step semantics, and typing rules for the lambda-calculus with mutation. *(Some rules are missing; see problem 1.)* Our syntax for creating, mutating, and retrieving the contents of a reference are like in ML. Important comments and definitions follow.

$$\begin{array}{rcl} e & ::= & \lambda x.\ e \mid x \mid e\ e \mid c \mid \mathsf{ref}\ e \mid\ !e \mid e{:=}e \mid l \\ v & ::= & \lambda x.\ e \mid c \mid l \\ H & ::= & \cdot \mid H, l \mapsto v \end{array} \qquad \begin{array}{rcl} \tau & ::= & \mathsf{int} \mid \tau \to \tau \mid \tau\ \mathsf{ref} \\ \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \mid \Gamma, l{:}\tau \end{array}$$

$\boxed{H; e \to H'; e'}$

$$\text{BETA}\ \frac{}{H; (\lambda x.\ e)\ v \to H; e[v/x]} \qquad \text{APP1}\ \frac{H; e_1 \to H'; e_1'}{H; e_1\ e_2 \to H'; e_1'\ e_2} \qquad \text{APP2}\ \frac{H; e_2 \to H'; e_2'}{H; v\ e_2 \to H'; v\ e_2'}$$

$$\text{ALLOC}\ \frac{l \notin H}{H; \mathsf{ref}\ v \to H, l \mapsto v; l} \qquad \text{REF1}\ \frac{H; e \to H'; e'}{H; \mathsf{ref}\ e \to H'; \mathsf{ref}\ e'} \qquad \text{GET}\ \frac{}{H; !l \to H; H(l)} \qquad \text{DEREF1}\ \frac{H; e \to H'; e'}{H; !e \to H'; !e'}$$

$\boxed{\Gamma \vdash e : \tau}$

$$\text{INT}\ \frac{}{\Gamma \vdash c : \mathsf{int}} \qquad \text{VAR}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \text{LAM}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \to \tau_2} \qquad \text{APP}\ \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1}$$

$$\text{REF}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{ref}\ e : \tau\ \mathsf{ref}} \qquad \text{ASSIGN}\ \frac{\Gamma \vdash e_1 : \tau\ \mathsf{ref} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \tau} \qquad \text{LABEL}\ \frac{\Gamma(l) = \tau}{\Gamma \vdash l : \tau\ \mathsf{ref}}$$

- The formal definition of substitution has been omitted.
- Our heap H is "threaded through" the whole program, like in IMP. The syntax $H(l)$ means lookup l in H. The syntax $H, l \mapsto v$ means the heap that is like H except l maps to v.
- References can hold values of any type, but the type of value held in a specific reference never changes.
- At run-time, a reference is a label (think of it as an address) l. Labels are distinct from variables. Labels would never exist in a source program, but our Preservation lemma will require us to type-check them. Therefore, our definition of Γ includes types for labels. Notice how the typing rule for labels is different from the rule for variables.
- The Preservation and Progress Lemmas will also require us to "typecheck a heap". We say $\vdash H : \Gamma$ if H is $\cdot, l_1 \mapsto v_1, \ldots l_n \mapsto v_n$ and Γ is $\cdot, l_1{:}\tau_1, \ldots, l_n{:}\tau_n$ (i.e., Γ has no variables and exactly the same labels as H) and for all $1 \leq i \leq n$, $\Gamma \vdash v_i : \tau_i$ (i.e., every value in the heap has the type Γ says is at that label). Note we could define $\vdash H : \Gamma$ with inference rules, but this description will suffice.
- We say "Γ_2 extends Γ_1" if for all l and x, (($\Gamma_1(l) = \tau$ implies $\Gamma_2(l) = \tau$) and ($\Gamma_1(x) = \tau$ implies $\Gamma_2(x) = \tau$)). That is, Γ_2 has to have everything Γ_1 does and with the same types, but it can have more. Note every Γ extends itself.

Part I Problems

1. The semantics is missing rules for assignment expressions. Add them. Enforce left-to-right evaluation. The result of an assignment should be (1) a heap where one label maps to a different value and (2) the value that was assigned. Note (2) is unlike ML, where the result is `()`.
2. The type system is missing a rule for dereference. Add it.
3. Prove this Progress Lemma: If $\vdash H : \Gamma$ and $\Gamma \vdash e : \tau$ and e is not a value, then there exists an H' and e' such that $H; e \rightarrow H'; e'$.

   Assume (i.e, use without proof) this (trivial) Canonical Forms Lemma: If $\vdash H : \Gamma$ and $\Gamma \vdash v : \tau$ then:

   - If τ is int, then v is some c.
   - If τ is some $\tau_1 \rightarrow \tau_2$, then v is some $\lambda x.\ e$.
   - If τ is some $\tau'\ \mathsf{ref}$, then v is some l and $H(l)$ is some v'.
4. Prove this Preservation Lemma: If $\vdash H : \Gamma$ and $\Gamma \vdash e : \tau$ and $H; e \rightarrow H'; e'$, then there exists a Γ' such that Γ' extends Γ, $\vdash H' : \Gamma'$, and $\Gamma' \vdash e' : \tau$.

   Assume (i.e., use without proof) these lemmas:

   - Weakening: If $\Gamma \vdash e : \tau$ and Γ' extends Γ, then $\Gamma' \vdash e : \tau$. (Hint: The heap requires using this lemma directly in certain cases of the Preservation proof.)
   - Heap Lookup: If $\vdash H : \Gamma$, then $\Gamma \vdash H(l) : \Gamma(l)$.
   - Heap Extension: If $\vdash H : \Gamma$ and $l \notin H$ and $\Gamma \vdash v : \tau$, then $\vdash H, l \mapsto v : \Gamma, l{:}\tau$.
   - Heap Update: If $\vdash H : \Gamma$ and $\Gamma(l) = \tau$ and $\Gamma \vdash v : \tau$, then $\vdash H, l \mapsto v : \Gamma$.
   - Substitution: If $\Gamma, x{:}\tau' \vdash e : \tau$ and $\Gamma \vdash e' : \tau'$, then $\Gamma \vdash e[e'/x] : \tau$.
5. Explain why the Preservation Proof would not succeed if it did not state that Γ' extends Γ.

What to turn in:

- Place all electronic submission files in your bitbucket cis-624 repo, in a `hw4` subdirectory.
- Part I: Hard-copy (written or typed) answers to problems 1–5. If electronic, name your file partI.pdf (or .txt, etc.)

Part II. Interpreter and type checker

`hw4.tar`, available on the course website, contains several Caml files you will need.

The second part of this assignment investigates implementing a type-checker and a *left-to-right*, *large-step*, *environment-based* interpreter for our language in ML.

Language and Concrete Syntax: The code provided to you defines abstract syntax and a parser for the simply-typed λ-calculus with integers, addition, multiplication, greater-than, conditionals (0 is false, other integers are true), and a heap. To make parsing and type-checking easier, functions have the concrete form `(fn x:t. e)`. Specfically, they must be surrounded by parentheses, they must have explicit argument types, and the ":" and "." must be present. Conditionals, assignment, dereference, allocation, and let-expressions also require parentheses around them; see the `example` file provided. The parser desugars (i.e., converts) let to function application. Advice: If you get a parser error when running `./hw4` on some input , make a copy of your input program and use manual binary search to find it.

Large-Step: A large-step interpreter for a language with a heap must produce a heap and a value as results and "thread the heap along". For example, if we were using substitution (**we are not**; see below), the code for application would correspond to this inference rule:

$$\frac{H; e_1 \Downarrow H_1; \lambda x.\ e_3 H_1; e_2 \Downarrow H_2; v_2 H_2; e_3[v_2/x] \Downarrow H_3; v}{H; e_1 e_2 \Downarrow H_3; v}$$

Environments: An environment-based interpreter does not use substitution. See class website for additional reading on environments vs substitution. Instead, the program state includes an environment, which maps variables to values. To implement lexical scope correctly, functions are not values—they evaluate to *closures* (written $<e, E>$ below). Here is a formal large-step semantics for a small language without a heap (see above). Pay particular attention to the rule for function application—we evaluate function bodies using the environment in its closure! Notice we do not "thread the environment along"; evaluation does not produce an environment.

$$\begin{array}{rcl} e & ::= & c \mid x \mid \lambda x.\ e \mid e\ e \mid <\lambda x.\ e, E> \\ E & ::= & \cdot \mid E, x \mapsto v \\ v & ::= & c \mid <\lambda x.\ e, E> \end{array}$$

$$\frac{}{E; v \Downarrow v} \qquad \frac{}{E; x \Downarrow E(x)} \qquad \frac{}{E; \lambda x.\ e \Downarrow <\lambda x.\ e, E>}$$

$$\frac{E; e_1 \Downarrow <\lambda x.\ e_3, E_1> \qquad E; e_2 \Downarrow v_2 \qquad E_1, x \mapsto v_2; e_3 \Downarrow v}{E; e_1\ e_2 \Downarrow v}$$

(Problems on next page.)

Part II Problems

6. Complete the type-checker `Main.typecheck`. This also serves the purpose of defining the typing rules, i.e., you are writing the typing rules in OCaml (you do not have to do this separately on paper). Your type-checker should succeed only for "source programs" – reject any expression containing a label or a closure. Raise `TypeError` if and only if the expression does not typecheck under the provided context. Use the little list library in the file for managing the context (but remember to catch `ListError` where necessary).

7. Complete the large-step interpreter `Main.interpret`. This also serves the purpose of defining large-step semantics (you do not have to provide the large-step rules on paper, but you are free to do this for yourself). Remember to "thread the heap through." In particular, at a function call, the body is evaluated using the environment in the closure but the current heap. Use the little list library in the file for managing environments and heaps. Your interpreter should raise an exception if it gets stuck. (But expressions that type-check should not get stuck).

8. The provided example program should produce the value 3. Explain why by explaining how the "function" bound to `newf` behaves.

9. The file `infinite` contains a program that type-checks and for which the interpreter will run forever. In this language, a function can call itself via "backpatching" through the heap[1]. In the provided file `factorial`, replace the expression bound to `fact` so that it produces a function that computes the factorial of positive numbers. Hint: Use the same "backpatching" trick used in the provided `infinite` program.

What to turn in:

- Place all electornic submission files in your bitbucket repo, in a `hw4` subdirectory.
- Part I: Hard-copy (written or typed) answers to problems 1–5. If electronic, name your file partI.pdf (or .txt, etc.)
- Part II:
  - Answer to problem 8 in a text file `prob8.txt` (or pdf, etc). OCaml source code in a file called `main.ml`. *Do not modify OCaml files other than `main.ml`.*
  - File `factorial`.

[1] Backpatching is the activity of filling up unspecified information using appropriate semantic actions.