COMP 321 Homework 6, Fall 2021: Lazy Evaluation

Thus far, we have focused on *call-by-value/eager* languages. Recall that call-by-value means that the argument to a function is evaluated before substituting its value into the function body

$$\frac{e_1\ \mathsf{value} \quad e_2 \mapsto e_2'}{e_1\ e_2 \mapsto e_1\ e_2'} \qquad \frac{e_2\ \mathsf{value}}{(\lambda x : \tau.e)\ e_2 \mapsto e[e_2/x]}$$

Eager evaluation means that a compound data structure is a value only when its components are a value. For example, the rules making $\mathsf{cons}(e_1, e_2)$ eager are

$$\frac{e_1\ \mathsf{value} \quad e_2\ \mathsf{value}}{\mathsf{cons}(e_1, e_2)\ \mathsf{value}} \qquad \frac{e_1 \mapsto e_1'}{\mathsf{cons}(e_1, e_2) \mapsto \mathsf{cons}(e_1', e_2)} \qquad \frac{e_1\ \mathsf{value} \quad e_1 \mapsto e_1'}{\mathsf{cons}(e_1, e_2) \mapsto \mathsf{cons}(e_1, e_2')}$$

An alternative is *call-by-name/lazy* languages. A call-by-name evaluation order for functions means that the whole un-evaluated expression e_2 is substituted into the body of the function, replacing the above rules with

$$\frac{}{(\lambda x : \tau.e)\ e_2 \mapsto e[e_2/x]}$$

One argument for call-by-name evaluation is that, if x is never used by e, then e_2 is never evaluated — in call-by-value, it is always evaluated, whether it is used or not.

A lazy evaluation order for $\mathsf{cons}(e_1, e_2)$ means that cons is a value regardless of whether or not its subexpressions are, replacing the above rules with

$$\frac{}{\mathsf{cons}(e_1, e_2)\ \mathsf{value}}$$

One advantage of lazy data structures is that they can easily represent infinite structures, e.g. an infinite list of all natural numbers $[0, 1, 2, 3, \ldots]$.

Some languages adopt a call-by-name/lazy evaluation order entirely, while other languages are generally call-by-value/eager but provide support for using call-by-name/laziness in specifically marked places (to some extent, this can even be implemented as a library).

However, most languages/libraries do not actually use call-by-*name*/lazy evaluation as described above, but a variant called *call-by-need*. These differ when a function uses its input more than once. For example, in naïve call-by-name

$$(\lambda x : \mathsf{nat}.x + x)\ e_2 \mapsto e_2 + e_2$$

and so e_2 will be evaluated *twice* — which could be expensive if it is a large, time-consuming program. In call-by-need, the idea is that e_2 is evaluated *at most once*. If x is never used, e_2 is not evaluated at all. But if x is used, then e_2 is evaluated the first time x is used, but the value of e_2 is reused the second time x is used.

The same policy is used for the pieces of data structures (e.g. the e in $\succ e$ and the e_1 and e_2 in $\mathsf{cons}(e_1, e_2)$), let-bound expressions, and the bodies of fixed points.

This intuition can be implemented using the same technical machinery of a memory that we have used for mutable references in the last few lectures. The goal for this assignment is to understand that technical machinery, and to understand call-by-need/lazy evaluation order, and to see an example where the same syntax of a programming language can be given different operational semantics. You will first extend the rules for call-by-need from the textbook to check your understanding, and then implement them.

1 Rules for lists and let

A full description of call-by-need evaluation is in Section 36.1 and 36.2 of PFPL. Some notes on notation:

- In class, we wrote $\mu \mid e$ for a memory μ and a expression e. In the book, this is annotated with the *store typing* Σ, which maps locations to the type of expression that should be stored in that location in the memory, and the triple (Σ, μ, e) is written $\nu\Sigma\{e \mid\mid \mu\}$. For example

$$\nu(l_1 \sim \mathsf{nat}, l_2 \sim \mathsf{list})\{e \mid\mid l_1 \hookrightarrow (1+1), l_2 \hookrightarrow \mathsf{cons}(@l_1, \mathsf{nil})$$

  represents the memory where l_1 has type nat and l_2 has type list (lists of natural numbers) and l_1 contains the expression $1+1$ and l_2 contains the expression $\mathsf{cons}(@l_1, \mathsf{nil})$

- The extension of a memory μ with a new binding $l \hookrightarrow e$ is written $\mu \otimes l \hookrightarrow e$. In class we wrote $\mu, l = e$.

- The use of a location l is written as an expression constructor $@l$. In class we wrote just l for this.

- In class, when we wrote the operational semantics for mutable references, we left the memory typing Σ out of the step rules. While the memory typing is not strictly necessary for defining how a program steps, outputing the new memory typing is helpful for re-type checking a program after it takes a step, which can be helpful for debugging.

When we wrote operational semantics for mutable references in class, a memory location $@l$ was treated as a value of reference type. For call-by-need/lazy implementation of PCF, there are no reference types — mutable memory is being used behind the scenes only to implement a more efficient variant of call-by-need. Thus, a memory location is not treated as a value, but as an expression that steps. Intuitively, to evaluate $@l$, you look at the binding of l in the memory. If it is bound to a value, then you (re)use that value (36.3a). If it is bound to a non-value expression, you evaluate the expression in the memory, and return its value, leaving the value in the memory for any subsequent use (36.3b). It is possible to write programs where the value of a location is needed during the calculation of the value of that very location. In call-by-name, such a program would infinite-loop. In call-by-need, we can use a special memory binding called a "black hole" to detect such infinite loops and stop the computation. This is accomplished by binding a location to a "black hole" while evaluating the expression for that location.

How do locations get bound to expressions in the memory? When evaluating a successor $\mathsf{succ}(e)$ (36.3c), you don't evaluate e right away, but instead bind a new memory location to it. Similarly for function calls (36.3h) and fixed points (36.3i).

Task 1 (20%). Extend the rules in Section 36.1 with call-by-need/lazy operational semantics rules (value and step rules) for

1.
$$\frac{}{\Gamma \vdash_\Sigma \mathsf{nil} : \mathsf{list}}$$

2.
$$\frac{\Gamma \vdash_\Sigma e_1 : \mathsf{nat} \quad \Gamma \vdash_\Sigma e_2 : \mathsf{list}}{\Gamma \vdash_\Sigma \mathsf{cons}(e_1, e_2) : \mathsf{list}}$$

3.
$$\frac{\Gamma \vdash_\Sigma e : \mathsf{list} \quad \Gamma \vdash_\Sigma e_0 : \tau \quad \Gamma, x : \mathsf{nat}, xs : \mathsf{list} \vdash_\Sigma e_1 : \tau}{\Gamma \vdash_\Sigma \mathsf{listcase}(e, e_0, x.xs.e_1) : \tau}$$

4.
$$\frac{\Gamma \vdash_\Sigma e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash_\Sigma e_2 : \tau_2}{\Gamma \vdash_\Sigma \mathsf{let}(\tau_1, e_1, x.e_2) : \tau_2}$$

Task 2 (20%). Prove the cases of type preservation (Theorem 36.1) for the step rules you defined in the previous task. Specifically, the theorem you need to prove by induction is the generalization at the start of the Proof:

> If $\cdot \vdash_\Sigma e : \tau$ and $\vdash_\Sigma \mu : \Sigma$ and $\nu\Sigma\{e \mid\mid \mu\} \mapsto \nu\Sigma'\{e' \mid\mid \mu'\}$ then Σ' is an extension of Σ and $\vdash_{\Sigma'} \mu' : \Sigma'$ and $\cdot \vdash_{\Sigma'} e' : \tau'$.

Recall from lecture that Σ' being an extension of Σ ($\Sigma' \supseteq \Sigma$) means that every location in Σ is in Σ' with the same type, but Σ' can contain additional locations with any type. The judgement $\vdash_\Sigma \mu : \Sigma$ means that each location l in the memory μ contains an expression of the type τ associated with l in Σ — and this expression may use all of the locations in the memory, to allow for circular references (see rule 36.5b). Additionally, the black holes are ignored/deemed to have any type.

2 Implementation

Next, you will implement the operational semantics (including the rules from the book and your new rules).

2.1 Syntax

The syntax is mostly the same as previous assignments. To keep things simple, natural numbers and lists have one-step case-analysis constructs `NatCase` and `ListCase` (not general pattern matching), which are written in concrete syntax as

```
natcase e {z => e0 | s x => e1 }
listcase e {nil => e0 | cons <x,xs> => e1 }
```

Additionally, unlike in previous assignments, let bindings

```
let x : t be e1 in e2
```

must be annotated with the type `t` of `e1` (you should see why when you write your step rules for `let`).

Some notes on the support code in `Syntax`:

- `location` is implemented by an `int`, and there is a function `freshLoc : unit -> location` for creating a new location.
- Memories are represented by

  ```
  datatype memory_entry = Defined of exp | BlackHole
  type memory = (location * typ * memory_entry) list
  ```

  A memory entry is either `Defined e` (written as just e in PFPL) or the special entry `BlackHole` (●).

  In the step rules, we always work with a memory typing $\Sigma = l_1 \sim \tau_1 \ldots l_n \sim \tau_n$ and a memory $\mu = l_1 \hookrightarrow e_1 \ldots l_n \hookrightarrow e_n$ where each location l_i in Σ occurs once in μ and vice versa. To simplify the code, we zip these together into one list, so `memory` is a list of triples (l_i, τ_i, e_i), so the memory

  $$(l_1, \tau_1, e_1), \ldots, (l_n, \tau_n, e_n)$$

  represents both the memory typing

  $$l_1 \sim \tau_1, \ldots, l_n \sim \tau_n$$

  and the memory

  $$l_1 \hookrightarrow e_1, \ldots, l_n \hookrightarrow e_n$$

2.2 Type checker

The type checker is provided for you in `check.sml`. It is essentially the standard type checker for numbers and functions and lists, except the `checkOpen` function also takes a memory typing as input and uses that to determine the types of locations. `check_memory` implements rule 36.5b.

2.3 Operational semantics

For the progress function for this language, we use the following result type:

```
datatype result =
    Done
  | Stepped of Syntax.memory * Syntax.exp
  | Loops
```

The specification for progress is then

```
Given |-_Sigma mu : Sigma and |-_Sigma e : tau, return
  Done iff e is a value
  Loops iff {{mu || e}} loops by accessing a black hole
  Stepped (mu',e') iff {e || mu} |-> {e' || mu'}
```

The "loops" judgement is defined in rules 36.6a-d, and you will need analogous rules for `let` and `listcase`. This judgement detects when a program accesses a location while computing the value that is supposed to be stored in that location. One example of a looping program is `fix x:nat.x`.

Task 1 (53%). Implement `progress` in `step.sml`.

2.4 Examples and Testing

The module `Top` includes the usual testing functions from previous homeworks.

Note that `Top.file_eval` and `Top.loop_eval` do not print anything for black holes — the evaluation will simply get stuck. On the other hand `Top.file_step` and `Top.loop_step` will print whether the program reached `Done` (by printing VALUE) or `Loops` (by printing BLACKHOLE).

In call-by-name/lazy evaluation, successor and cons are values regardless of what is inside of them. This makes the results of computation hard to interpret — for example, if you add two numbers, then the value will be either 0 or successor, but not tell you what it is successor of. The predecessor is only computed when some future computation calls for it. To make it easier to test, there are additional test functions `Top.loop_eval_deep n` and `Top.file_eval_deep filename n` that evaluate a program "deeply", i.e. they evaluate inside successor and cons as well. You can think of this as a top-level "to string" function that forces all of a value to be evaluated. The number `n` controls how many levels deep to evaluate before stopping, which is useful for printing parts of infinite structures.

Infinite lists can be defined using fixed points. For example, an infinite list of ones (see `examples.pcf`):

```
fix ones : list. cons < 1 , ones >
```

Task 1 (2%). Describe in words what this infinite list contains:

```
let
  addone : (list -> list) be
    fix addone : (list -> list).
      lam xs : list.
      listcase xs {
         nil => nil
       | cons <y,ys> => cons <s y , addone @ ys > }
in
  fix nums : list. cons<0 , addone @ nums >
end;
```

Task 2 (5%). The Fibonacci numbers $1, 1, 2, 3, 5, 8, 13, \ldots$ are defined by

$$\begin{aligned} fib(0) &= 1 \\ fib(1) &= 1 \\ fib(n) &= fib(n-1) + fib(n-2) \end{aligned}$$

Define an infinite list of all Fibonacci numbers.