COMP 321 FINAL Fall 2021

1 Logistics

- **Collaboration policy: you are not allowed to discuss the problems in this handout with anyone besides the professor. Once you begin working on the exam, you are not allowed to refer to any sources besides this semester's course materials (textbook and lecture materials).**
- To hand in, please upload `final.pdf` and `check.sml` and `step.sml` to the `final/` handin folder in your Google Drive handin folder.

2 Delayed Substitutions

2.1 Motivation

All semester, we have been using substitution $e'[e/x]$ in our operational semantics. However, substitution as we have described it is a bit unrealistic for an actual implementation, because substituting e in for x in e' requires traversing the entire program e', which adds linear overhead to each step that uses substitution. For example, consider the program

```
let add : (nat -> (nat -> nat)) be
    fix add : (nat -> (nat -> nat)).
      lam x:nat. lam y:nat. natcase x
          { z => y
          | s x' => s (add @ x' @ y)} in
let a : nat be 1 in
let b : nat be 2 in
let c : nat be 3 in
let d : nat be 4 in
let e : nat be 5 in
add @ (add @ (add @ a @ b) @ (add @ c @ d)) @ e
end
end
end
end
end
end;
```

In call-by-value/eager evaluation order, this program takes 7 steps to step through the `let`s to

```
<add> @ (<add> @ (<add> @ 1 @ 2) @ (<add> @ 3 @ 4)) @ 5
where
<add> = lam x:nat. lam y:nat. natcase x { z => y
  | s x' => s (fix add : (nat -> (nat -> nat)).
                lam x:nat. lam y:nat. natcase x
                { z => y
                | s x' => s (add @ x' @ y)} in @ x' @ y)}
```

The seven steps are

1. Unfold the `fix`
2. Reduce the `let add`, substituting `<add>` for each occurrence of the variable `add`.
3. Reduce the `let a`, substituting `1` for `a`
4. Reduce the `let b`, substituting `2` for `b`
5. Reduce the `let c`, substituting `3` for `c`
6. Reduce the `let d`, substituting `4` for `d`
7. Reduce the `let e`, substituting `5` for `e`

However, each operational step takes time linear in the remaining program. E.g. substituting for `add` recurs through the whole expression

```
let a : nat be 1 in
let b : nat be 2 in
let c : nat be 3 in
let d : nat be 4 in
let e : nat be 5 in
add @ (add @ (add @ a @ b) @ (add @ c @ d)) @ e
end
end
end
end
end
end;
```

and substituting for `a` recurs through

```
let b : nat be 2 in
let c : nat be 3 in
let d : nat be 4 in
let e : nat be 5 in
<add> @ (<add> @ (<add> @ 1 @ b) @ (<add> @ c @ d)) @ e
end
end
end
```

```
end
end
end;
```

etc.

This means that counting the number of operational semantics steps does not really capture the time complexity of the program.

2.2 Delaying Substitution

One way to address this problem is to use *delayed substitutions*. That is, rather than thinking of substitution as a *meta-level function* (`Syntax.subst` in the SML code from the homeworks) that *immediately* processes an expression by doing the substitution, we think of substitution as a new expression constructor.

$$\theta ::= \emptyset \mid \theta, e/x$$
$$e ::= \ldots \mid e[\theta]$$

There are two typing rules for substitutions themselves:

$$\frac{}{\Gamma \vdash \emptyset : \emptyset} \qquad \frac{\Gamma \vdash \theta : \Delta \quad \Gamma \vdash e : \tau \quad x \text{ not in } \Delta}{\Gamma \vdash (\theta, e/x) : (\Delta, x : \tau)}$$

And one typing rule for the new expression constructor, representing the application of a substitution θ to an expression e:

$$\frac{\emptyset \vdash \theta : \Delta \quad \Gamma, \Delta \vdash e : \tau}{\Gamma \vdash e[\theta] : \tau}$$

A (simultaneous) substitution is a list of expression-name pairs, and the application of a substitution to an expression, $e[\theta]$, is now a new expression constructor. The typing rules for $\Gamma \vdash \theta : \Delta$ say that a substitution θ substitutes for the variables in a context Δ if it lists an expression of the appropriate type for each variable in Δ.

The typing rule for applying a substitution to an expression is like a simultaneous n-ary let. However, to simplify the operational semantics, we restrict $e[\theta]$ so that θ itself does not have any free variables (which will be sufficient for the substitutions that come up in the operational semantics).

The idea for improving the efficiency of evaluation is an operational semantics rule like

$$(\lambda x.e)\, e_2 \mapsto e[e_2/x]$$

now refers to a *delayed* substitution $e[e_2/x]$, so this single step does not recur through the entire expression e.

2.3 Task

For this assignment, we will apply the idea of explicit substitutions to the same language as in the previous homework (functions, natural numbers, lists, let, general recursion), which has the following typing rules (in addition to the new ones for delayed substitutions in Section 2.2):

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{}{\Gamma \vdash \mathsf{zero} : \mathsf{nat}} \qquad \frac{\Gamma \vdash e : \mathsf{nat}}{\Gamma \vdash \mathsf{succ}(e) : \mathsf{nat}}$$

$$\frac{\Gamma \vdash e : \mathsf{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \mathsf{nat} \vdash e_1 : \tau}{\Gamma \vdash \mathsf{natcase}\{e, e_0, x.e_1\} : \tau}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$\frac{}{\Gamma \vdash \mathsf{nil} : \mathsf{list}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{nat} \quad \Gamma \vdash e_2 : \mathsf{list}}{\Gamma \vdash \mathsf{cons}(e_1, e_2) : \mathsf{list}}$$

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

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

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \mathsf{fix}\, x : \tau . e : \tau}$$

Task 1 (10%). Implement the above typing rules for delayed substitutions in `check.sml`.

Task 2 (30%). Give a full **call-by-value/eager** operational semantics for this language, defining both the value judgement and the step judgement. The *values* of type nat and list should be numerals and fully evaluated lists of numerals, respectively (i.e. they should not involve any delayed substitutions). Many rules can be reused or slightly modified from previous languages, but you should write them all out explicitly. You will need to add new rules for the delayed substitutions as well. Your step rules should not have any linear-in-the-expression time overhead related to substitution (but see the caveats below).

You don't need to prove this, but your operational semantics rules should satisfy the following: Let $|e|$ be a translation from the above language to one without delayed substitutions, which works by recuring over e and immediately applying any delayed substitutions it finds (using the meta-operation of substitution that we have used throughout the semester). Then:

- If e done then $|e|$ done.
- If $e \mapsto e'$ then $|e| \equiv |e'|$ (where $\equiv$ refers to definitional equality, i.e. the congruent equivalence relation generated by stepping).

Task 3 (20%). For all of your operational semantics rules from the previous task related to free variables uses (x), functions ($\lambda x : \tau . e$) and function application ($e_1\ e_2$)), show the case of type preservation for that rule.

Task 4 (40%). Implement the `progress` function for this language in `step.sml`.

2.4 Caveats / Extra Credit

You do not need to address the following sources of inefficiency in your implementation:

1. α-renaming: You can assume that α-renaming can be done in constant time. This is not true for the support code for the assignment, which (like all previous homeworks) uses named variables, and the function `freshenTop` takes time linear in the term. This inefficiency can be fixed by either using something called *de Bruijn form*, where variables are represented by numbers, or by using delayed variable swaps in addition to delayed substitutions.

   If you finish the rest of the exam, you can get some extra credit by implementing delayed variable swapping as well.

2. Finding the location to evaluate: the search rules in a standard operational semantics traverse the expression to find the location to be evaluated, and this work is repeated for each step. This can be fixed by using an explicit control stack so that this traversal happens only once (PFPL Chapter 28).

   If you finish the rest of the exam, you can get some extra credit by implementing stacks as well.

3. Operations on substitutions: in the support code, we represent substitutions as lists, which means that operations like looking up the expression associated with a variable and appending two substitutions take time linear in the substitution. This can be addressed by using a better representation (e.g. hash tables or balanced binary trees) for which these operations are $O(1)$ or $O(\log n)$ time.

4. In some sense, delayed substitutions are trading improved running time for using extra memory. You don't need to worry about the space usage of delayed substitutions for this assignment.