University of Massachusetts Lowell — Comp 3010: Organization of Programming Languages

Assignment 5

DUE: Thursday, Nov 7, 2024, 11:59PM

Submission instructions: We are using GradeScope for PDF submission and the coding assignment. That is, you must submit your work as a PDF document. If you are not able to submit your assignment, please contact course staff. Some points to note:

- Your submission must be a PDF. We encourage you to use LaTeX (see below). If you produce your assignment using a different tool, please make sure your assignment is legible and follows these submission instructions.
- The first page of your assignment should state your name, UML ID, and a list of collaborators (if any), and *all other pages of your other pages should not contain any identifying information*, i.e., they should not contain your name.
- We will provide a LaTeX template you can use to write the answers to your assignment.

1 Substitution (20 points)

In class we described *capture-avoiding substitution* $e_1\{e_2/x\}$ in order to describe β-reduction. We did not, however, formally define it. We rectify this by giving the following inductive definition of substitution.

$$
\begin{aligned}
y\{e/x\} &= \begin{cases} e & \text{if } x = y \\ y & \text{if } x \neq y \end{cases} \\
(e_1\ e_2)\{e/x\} &= e_1\{e/x\}\ e_2\{e/x\} \\
(\lambda y.\, e')\{e/x\} &= \begin{cases} \lambda y.\, e' & \text{if } x = y \\ \lambda y.\, (e'\{e/x\}) & \text{if } x \neq y \text{ and } y \notin FV(e) \\ \lambda z.\, ((e'\{z/y\})\{e/x\}) & \text{if } x \neq y \text{ and } y \in FV(e) \text{, where} \\ & \qquad z \notin FV(e) \cup FV(e') \cup \{x\} \end{cases}
\end{aligned}
$$

The function FV takes a lambda calculus expression e and returns the set of free variables of e. Thus, $y \in FV(e)$ if and only if y is a free variable of e. For example $FV(\lambda x.\, z\ \lambda y.\, x\ y) = \{z\}$, and for any expression e, e is a closed term if and only if $FV(e) = \varnothing$.

(a) Give an inductive definition of the function FV.

(b) Show the result of the following substitutions.

- $(\lambda z.\, y\ \lambda y.\, y\ z\ w)\{(\lambda x.\, x)/y\}$
- $((\lambda x.\, x\ y)\ (\lambda z.\, x\ z))\{(\lambda w.\, w\ w)/x\}$
- $(\lambda y.\, x\ y)\{(\lambda z.\, y\ z)/x\}$
- $((\lambda x.\, \lambda w.\, w\ x)\ \lambda y.\, x\ y)\{(w\ w)/x\}$

(c) Consider the following alternate (and incorrect) definitions for substitution for abstractions: $(\lambda y.\, e')\{e/x\}$ (for the other cases, the definition remains the same). For each alternate definition give an example substitution in which the original and alternate definitions produce different results, and give those results.

(i)

$$
(\lambda y.\, e')\{e/x\} = \begin{cases} \lambda y.\, e' & \text{if } x = y \\ \lambda y.\, (e'\{e/x\}) & \text{if } x \neq y \end{cases}
$$

(ii)

$$
(\lambda y.\, e')\{e/x\} = \begin{cases} \lambda y.\, e' & \text{if } x = y \\ \lambda y.\, (e'\{e/x\}) & \text{if } x \neq y \text{ and } y \notin FV(e) \\ \lambda z.\, ((e'\{z/y\})\{e/x\}) & \text{if } x \neq y \text{ and } y \in FV(e) \text{, where} \\ & \qquad z \notin FV(e) \cup \{x\} \end{cases}
$$

2 Implementing the Lambda Calculus (30 points)

The easiest way to get familiar with the lambda calculus is to play with it. And the easiest way to play with it is to have an interpreter for it. So let's derive an interpreter for the lambda calculus in OCaml using the small-step operational semantics.

Consider the following data type for lambda terms:

```
type lterm =
    LId of string
  | LLam of string * lterm
  | LApp of lterm * lterm
```

where:

- `LId "x"` represents the identifier x;
- `LLam ("x",`T`)` represents the lambda term $\lambda x.\, t$ where t is represented by T;
- `LApp (`T_1`,`T_2`)` represents the application $t_1\ t_2$ where t_i is represented by T_i.

From here on, to simplify the presentation, we will talk about elements of `lterm` as the lambda terms they represent.

We have provided you with a function `LambdaParser.parse` in source file `hw5.ml` that takes a lambda term written as a string in a reasonable notation and returns an element of the `lterm` data type:

```
# LambdaParser.parse "/x./y.x y";;
- : lterm = LLam ("x", LLam ("y", LApp (LId "x", LId "y")))
# LambdaParser.parse "(/x./y.x y) z1 z2";;
- : lterm = LApp (LApp (LLam ("x", LLam ("y", LApp (LId "x", LId "y"))),
                        LId "z1"), LId "z2")
```

There is also a function `LambdaParser.pp` that does the reverse, taking an element of `lterm` and producing a string suitable for printing.

```
# LambdaParser.pp (LLam("x",LId("y")));;
- : string = "/x.y"
# LambdaParser.pp (LApp(LLam("x",LId("y")),(LApp(LId("z1"),LId("z2")))));;
- : string = "(/x.y) (z1 z2)"
```

(a) The main small-step operational semantics rule of the lambda calculus is the β-reduction rule

$$\beta\text{-REDUCTION}\ \frac{}{(\lambda x.\, e_1)\ e_2 \longrightarrow e_1\{e_2/x\}}$$

It relies on a notion of (capture-avoiding) substitution $e_1\{e_2/x\}$. We use the formal definition of capture-avoiding substitution defined in the previous question.

$$y\{e/x\} = \begin{cases} e & \text{if } x = y \\ y & \text{if } x \neq y \end{cases}$$

$$(e_1\ e_2)\{e/x\} = e_1\{e/x\}\ e_2\{e/x\}$$

$$(\lambda y.\, e')\{e/x\} = \begin{cases} \lambda y.\, e' & \text{if } x = y \\ \lambda y.\, (e'\{e/x\}) & \text{if } x \neq y \text{ and } y \notin FV(e) \\ \lambda z.\, ((e'\{z/y\})\{e/x\}) & \text{if } x \neq y \text{ and } y \in FV(e)\text{, where} \\ & \quad z \notin FV(e) \cup FV(e') \cup \{x\} \end{cases}$$

It uses a function FV taking a lambda calculus expression e and returning the set of free variables of e. Thus, $y \in FV(e)$ if and only if y is a free variable of e. For example $FV(\lambda x.\, z\ \lambda y.\, x\ y) = \{z\}$, and for any expression e, e is a closed term if and only if $FV(e) = \varnothing$.

Code a function

```
substitute :  lterm -> string -> lterm -> lterm
```

where `substitute` $t_1\ \ x\ \ t_2$ returns the result of substituting t_2 for x in t_1.

You will need to implement a function corresponding to FV, and figure out how to create an identifier z that is not in $FV(e) \cup FV(e') \cup \{x\}$ to implement the capture-avoiding aspect of the substitution function.

```
# let p = LambdaParser.parse;;
val p : string -> lterm = <fun>
# let pp = LambdaParser.pp;;
val pp : lterm -> string = <fun>
# pp (substitute (p "/z.y (/y.y z w)") "y" (p "/x.x"));;
- : string = "/z.(/x.x) (/y.y z w)"
# pp (substitute (p "(/x.x y) (/z.x z)") "x" (p "/w.w w"));;
- : string = "(/x.x y) (/z.(/w.w w) z)"
# pp (substitute (p "(/y.x y)") "x" (p "/z.y z"));;
- : string = "/y_0.(/z.y z) y_0"
# pp (substitute (p "(/x./w.w x) (/y.x y)") "x" (p "w w"));;
- : string = "(/x./w.w x) (/y.w w y)"
```

(b) Code a function

```
reduce :  lterm -> lterm option
```

where `reduce` t applies the one-step reduction relation of the lambda calculus to term t. If $t \longrightarrow t'$, then `reduce` t returns `Some` t', otherwise, it returns `None`.

Because ultimately we care about finding normal forms of terms, we ask you to implement the *normal order evaluation strategy*: your function `reduce` should return the term obtained by applying the *outer-most left-most* reduction it can find to t.

```
# let p = LambdaParser.parse;;
val p : string -> lterm = <fun>
# let pp x = match x with None -> "" | Some t -> LambdaParser.pp t;;
val pp : lterm option -> string = <fun>
# pp (reduce (p "(/x.x) y"));;
- : string = "y"
# pp (reduce (p "y (/x.x)"));;
- : string = ""
# pp (reduce (p "(/x.x) (/y.y)"));;
- : string = "/y.y"
# pp (reduce (p "/z.(/x.x) (/y.y)"));;
- : string = "/z./y.y"
# pp (reduce (p "(/x.x) ((/y.y) z)"));;
- : string = "(/y.y) z"
# pp (reduce (p "(/x.(/z.z) x) (/y.y)"));;
- : string = "(/z.z) (/y.y)"
# pp (reduce (p "(/x.(/z.z) (/x.x)) (/y.y)"));;
- : string = "(/z.z) (/x.x)"
```

(c) Use your `reduce` function to code a function

```
normal_form :  lterm -> lterm
```

where `normal_form` t returns the normal form of t.

The function obviously should not terminate if t has no normal form.

We've provided you with a function

```
eval: string -> string
```

that calls your function `normal_form`, taking in a lambda term as a string and using `LambdaParser.parse` to convert it to a `lterm` value suitable for `normal_form`. The result is converted back in a string format using function `LambdaParser.pp`.

For extra fun, you can make `normal_form` print the sequence of intermediate terms it goes through during execution. (We will only test for the final result, so feel free to format your output as you see fit.)

```
# eval "/x./y.x y";;
Term already in normal form
- : string = "/x./y.x y"
# eval "(/x./y.x y) z1 z2";;
   (/x./y.x y) z1 z2
 = (/y.z1 y) z2
 = z1 z2
- : string = "z1 z2"
# eval "(/x./y.y x) z1 z2";;
   (/x./y.y x) z1 z2
 = (/y.y z1) z2
 = z2 z1
- : string = "z2 z1"
# eval "(/x./y./z.y) z1 z2 z3";;
   (/x./y./z.y) z1 z2 z3
 = (/y./z.y) z2 z3
 = (/z.z2) z3
 = z2
- : string = "z2"
# eval "(/x./y.y x) z1 (/z2.z2 z2)";;
   (/x./y.y x) z1 (/z2.z2 z2)
 = (/y.y z1) (/z2.z2 z2)
 = (/z2.z2 z2) z1
 = z1 z1
- : string = "z1 z1"
```

You should upload `hw5.ml` under gradescope assignment **HW5: Lambda Calculus**.