CSED321 Assignment 5 - *Untyped Lambda Calculus*

Due Tuesday, April 4nd

In this assignment, you will implement the operational semantics of the untyped λ-calculus. The goal of this assignment is to develop the skill of *interpreting inference rules as algorithms,* which is absolutely crucial in implementing programming languages. That is, given a system of inference rules, we wish to extract an algorithm corresponding to the inference rules, and this assignment is designed to help you to develop such a skill.

Consider the following reduction rules for the reduction judgment $e \mapsto e'$ based on the call-by-value reduction strategy:

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}\ \mathit{Lam} \qquad \frac{e_2 \mapsto e_2'}{(\lambda x.\, e)\ e_2 \mapsto (\lambda x.\, e)\ e_2'}\ \mathit{Arg} \qquad \frac{}{(\lambda x.\, e)\ v \mapsto [v/x]e}\ \mathit{App}$$

The rule *Lam* says that if e_1 reduces to e_1', then $e_1\ e_2$ reduces to $e_1'\ e_2$. This literal interpretation answers the question of "why is a given reduction valid?" Conceptually the input to the problem is a reduction judgment $e \mapsto e'$, and the answer is either *yes* with a derivation tree that justifies the reduction, or *no* which means that the reduction is invalid.

In implementing the operational semantics, however, we would be interested in "how to reduce a given expression." In this case, the input to the problem is conceptually a certain expression e, and the answer is either another expression e' with a derivation tree of $e \mapsto e'$, or *no* which means that there is no expression that e reduces to. Therefore we need to interpret the reduction rules not literally but *algorithmically.* In this particular case, we need to interpret the reduction rules from the conclusion to the premises, *i.e.*, in the bottom-up way.

Let us interpret the reduction rules algorithmically. Since the input to the problem is an expression and the output is either another expression or *no*, we introduce a function `step` that takes an expression e and returns an Ocaml option type of expressions:

- `step` e returns `Some` e' for another expression e' if e reduces to e'.
- `step` e returns `None` if there no expression that e reduces to.

Note that Ocaml option types are declared as: `type 'a option = Some of 'a | None`. Then how do we rewrite the rule *Lam*, for example, in terms of the function `step`?

Intuitively, the rule *Lam* should be interpreted as follows: (There is an answer right below, but you are encouraged to try to figure out an algorithmic interpretation of the rule *Lam* on your own. Try to figure out how to rewrite the reduction rule using `step`.)

1. Consider the case in which `step` takes $e_1\ e_2$ as an argument.
2. `step` makes a recursive call to determine the expression that e_1 evaluates to, if any.
3. If `step` e_1 returns `Some` e_1', we return `Some` $e_1'\ e_2$.
4. If `step` e_1 returns `None`, we return `None`.

The other two rules can be interpreted in a similar way. The goal of this part is to implement such a function `step` to implement the reduction rules.

In implementing the function `step`, you will need to implement functions for substitution $[e'/x]e$. Its inductive definition is given in the course notes as follows, where the last equation deals with the case when a variable capture happens:

$$\begin{array}{rcll}
[e/x]x & = & e & \\
[e/x]y & = & y & \textit{if } x \neq y \\
[e/x](e_1\ e_2) & = & [e/x]e_1\ [e/x]e_2 & \\
[e'/x]\lambda x.\, e & = & \lambda x.\, e & \\
[e'/x]\lambda y.\, e & = & \lambda y.\, [e'/x]e & \textit{if } x \neq y,\ y \notin FV(e') \\
[e'/x]\lambda y.\, e & = & \lambda z.\, [e'/x][y \leftrightarrow z]e & \textit{if } x \neq y,\ y \in FV(e'),\ z \notin \{x, y\} \cup FV(e) \cup FV(e')
\end{array}$$

That is, you need to extract functions from the inductive definitions of: (i) $FV(e)$ for free variables in e, (ii) $e \equiv_\alpha e'$ for the α-equivalence between e and e', and (iii) $[e'/x]e$ for substituting e' for x in e. Unlike the previous assignments, this assignment does not provide the specification for these functions except for their inductive definitions, all of which can be found in the course notes. All we care about is the correctness of `step` and nothing else.

The reason why we do not give out the specification for these functions (other than their inductive definitions) is to teach students an important principle in software development: *design and specification.* Half the battle in software development is actually to figure out "what to implement" rather than "how to implement." E.g., the implementation of `reach`, `distance`, and `weight` in Assignment 3 would have been a lot more difficult if students were instructed to start from scratch. The programming part in this assignment is essentially no different: you will spend most of your time *designing* your code rather than actually writing it.

Remark. Think a lot before you type anything on the screen. You don't even have to turn on your computer before you finalize the design – what functions to implement, their types, their invariants, etc. You might well be tempted to start with a (bad) design without giving enough consideration to its correctness, but eventually it will waste you more time than it saves. The amount of time you will spend (or waste) doing this assignment will be directly proportional to the number of times you ignore this advice.

Programming Instruction

Download the zip file `hw5.zip` from the course webpage or by running the command `receive` on the server `141.223.163.223`, and unzip it on your working directory. It will create a bunch of files. First see `uml.ml`. `UML` stands for `Untyped ML`, and you will be implementing an interpreter of UML which is another name of the (untyped) λ-calculus.

```
type var = string

type exp =
    Var of var
  | Lam of var * exp
  | App of exp * exp
```

The datatype `exp` corresponds to the syntactic category expression in the course notes:

- `Var` x denotes a variable x as an expression in UML.
- `Lam` $(x,\ e)$ denotes a λ-abstraction $\lambda x.\, e$ in UML.
- `App` $(e_1,\ e_2)$ denotes an application $e_1\ e_2$ in UML.

Next see hw5.mli and hw5.ml. The goal of this assignment is to implement two function stepv and stepn, for call-by-value and call-by-name reduction strategies, respectively:

```
exception NotImplemented

(* one-step reduction in the call-by-value reduction strategy,
   returns NONE if impossible *)
val stepv : Uml.exp -> Uml.exp option

(* one-step reduction in the call-by-name reduction strategy,
   returns NONE if impossible *)
val stepn : Uml.exp -> Uml.exp option
```

That is, stepv and stepn take an expression e of type Uml.exp, and return Some e' of type Uml.exp option for another expression e' that e reduces to; if there is no such expression e', it returns None. You will also need to implement substitution as well.

After implementing stepv and stepn, run the command make to compile the sources files. There are two ways to test your code. First, you can run mainV for the call-by-value strategy (and mainN for the call-by-name strategy). At the UML prompt, enter a UML expression followed by the semicolon symbol ;. (The syntax of UML will be given shortly.) Each time you press the return key, a reduced expression according to your stepv function is displayed.

```
$ ./mainV
Uml> (lam x. x) (lam y. y);
((lam x. x) (lam y. y))
Press return:
(lam y. y)
Press return:
Uml>
```

Alternatively you can use those functions in loop.ml in the interactive mode of OCaml. (You don't actually need to read loop.ml.) At the OCaml prompt, type #load "lib.cma";; to load the library for this assignment. Then open the structure Loop:

```
$ ocaml
        OCaml version 4.05.0

# #load "lib.cma";;
# open Loop;;
#
```

You type loop (Hw5.stepv (wait show));; at the OCaml prompt, and then enter a UML expression followed by the semicolon symbol ;.

```
# loop (step Hw5.stepv (wait show));;
Uml> (lam x. x) (lam y. y);
((lam x. x) (lam y. y))
Press return:
(lam y. y)
Press return:
Uml>
```

Each time you press the return key, a reduced expression is displayed. To skip all intermediate steps, you can try loop (step Hw5.stepv show);;.

Or you may use a UML expression stored in a separate file. We provide three UML files: `nat.uml`, `rec.uml`, and `fib.uml` in the `tests` directory.

```
# loopFile "tests/nat.uml" (step Hw5.stepv (wait show));;
...
```

If you want to see the entire reduction sequence without pressing the return key, use the function `step Hw5.stepv show`:

```
# loopFile "tests/nat.uml" (step Hw5.stepv show);;
...
```

If you want to skip all intermediate steps and see only the final result, use the function `eval Hw5.stepv show`:

```
# loopFile "tests/nat.uml" (eval Hw5.stepv show);;
...
```

To test your `stepn` function, use `Hw5.stepn` instead of `Hw5.stepv` in the above examples.

The Syntax of UML

The syntax for `UML` closely resembles that for the λ-calculus. The only difference is the use of the keyword `lam` in place of λ, and syntactic sugar `let` x `=` e `in` e' for `(lam` x`.` e'`)` e.

$$\text{expression} \qquad e \quad ::= \quad x \mid \texttt{lam}\ x\texttt{.}\ e \mid e\ e \mid \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e$$

The following UML expression computes a Church numeral for a natural number eight (which is found in `nat.uml`):

```
let one = lam s. lam z. s z in
let add = lam x. lam y. lam s. lam z. y s (x s z) in
let two = add one one in
let four = add two two in
let eight = add four four in
eight
;
```

Submission Instruction

1. Make sure that you can compile `hw5.ml` by running `make`.
2. When you have the file `hw5.ml` ready for submission, run the `handin` command in the same directory, and your file will be submitted automatically.