CSED321 Assignment 7 - *De Bruijn Indices*

Due Thursday, May 2

In this assignment, you will implement the operational semantics of TML (Typed ML) using de Bruijn indices. TML was introduced in Assignment 6, but this assignment does not consider the type system of TML. The abstract syntax for TML is shown below.

$$\begin{array}{rrl} \textsf{type} & A ::= & \textsf{bool} \mid \textsf{int} \mid A \to A \mid A \times A \mid \textsf{unit} \mid A + A \\ \textsf{expression} & e ::= & x \mid \lambda x{:}A.\, e \mid e\ e \mid (e, e) \mid \textsf{fst}\ e \mid \textsf{snd}\ e \mid () \\ & \mid & \textsf{fix}\ x{:}A.\, e \mid \textsf{inl}_A\ e \mid \textsf{inr}_A\ e \mid \textsf{case}\ e\ \textsf{of inl}\ x.\, e \mid \textsf{inr}\ x.\, e \\ & \mid & \textsf{true} \mid \textsf{false} \mid \textsf{if}\ e\ \textsf{then}\ e\ \textsf{else}\ e \mid \widehat{n} \mid \textsf{plus} \mid \textsf{minus} \mid \textsf{eq} \end{array}$$

Translating into De Bruijn Indices

The first part of this assignment is to implement a function that translates a TML expression into an expression with de Bruijn indices. For example, it converts $\lambda x{:}A.\, \lambda y{:}B.\, x\ y$ into $\lambda.\lambda.1\ 0$. The abstract syntax for TML with de Bruijn indices is shown below.

$$\begin{array}{rrl} \textsf{expression} & e ::= & i \mid \lambda.e \mid e\ e \mid (e, e) \mid \textsf{fst}\ e \mid \textsf{snd}\ e \mid () \\ & \mid & \textsf{fix}\ e \mid \textsf{inl}\ e \mid \textsf{inr}\ e \mid \textsf{case}\ e\ \textsf{of inl}\ e \mid \textsf{inr}\ e \\ & \mid & \textsf{true} \mid \textsf{false} \mid \textsf{if}\ e\ \textsf{then}\ e\ \textsf{else}\ e \mid \widehat{n} \mid \textsf{plus} \mid \textsf{minus} \mid \textsf{eq} \end{array}$$

where i denotes a de Bruijn index of a variable. Notice that this syntax does not use types at all. In fact, we do not have to keep types because types play no role in evaluating.

To deal with expressions containing free variables, we use a *naming context* which is similar to a typing context. A naming context assigns the de Bruijn index i to a free variable x:

$$\textsf{naming context} \quad \Gamma ::= \cdot \mid \Gamma, x \to i$$

We assume that the leftmost free variable in an expression is given the lowest index. For example, the naming context of the expression $\lambda x{:}A.\, a\ x\ b\ c$ is $\Gamma = a \to 0, b \to 1, c \to 2$, and this expression is translated into $\lambda.1\ 0\ 2\ 3$. In the same way, $\lambda x{:}A.\, \lambda y{:}B.\, a\ x\ b\ c$ is translated into $\lambda.\lambda.2\ 1\ 3\ 4$, and $\lambda x{:}A.\, \lambda y{:}B.\, a\ b$ is translated into $\lambda.\lambda.2\ 3$. When a free variable appears more than twice, the position of the leftmost free variable determines its index. For example, $a\ b\ c\ a$ is translated into $0\ 1\ 2\ 0$, and $a\ b\ c\ c\ b$ is translated into $0\ 1\ 2\ 2\ 1$.

Implementing the Operational Semantics

In the second part, you will implement the operational semantics of TML with de Bruijn indices. We have already implemented the operational semantics of untyped λ-calculus in Assignment 5. Similarly, in this assignment, you will implement the operational semantics of TML for both call-by-value (eager) and call-by-name (lazy) evaluation strategies. For your reference, we give the operational semantics of TML in Figure 1.

Remark. We do not give the definition of values for the abstract syntax with de Bruijn indices; you have to find the definition of values v. You also have to find substitution rules for TML. Since our expression uses de Bruijn indices, you should not need to implement α-conversion.

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

$$\frac{e_1 \mapsto e_1'}{(e_1, e_2) \mapsto (e_1', e_2)}\ \mathit{Pair} \qquad \frac{e_2 \mapsto e_2'}{(v_1, e_2) \mapsto (v_1, e_2')}\ \mathit{Pair}'$$

$$\frac{e \mapsto e'}{\mathsf{fst}\ e \mapsto \mathsf{fst}\ e'}\ \mathit{Fst} \qquad \frac{}{\mathsf{fst}\ (v_1, v_2) \mapsto v_1}\ \mathit{Fst}' \qquad \frac{e \mapsto e'}{\mathsf{snd}\ e \mapsto \mathsf{snd}\ e'}\ \mathit{Snd} \qquad \frac{}{\mathsf{snd}\ (v_1, v_2) \mapsto v_2}\ \mathit{Snd}'$$

$$\frac{e \mapsto e'}{\mathsf{inl}_A\ e \mapsto \mathsf{inl}_A\ e'}\ \mathit{Inl} \qquad \frac{e \mapsto e'}{\mathsf{inr}_A\ e \mapsto \mathsf{inr}_A\ e'}\ \mathit{Inr}$$

$$\frac{e \mapsto e'}{\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2 \mapsto \mathsf{case}\ e'\ \mathsf{of}\ \mathsf{inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2}\ \mathit{Case}$$

$$\frac{}{\mathsf{case}\ \mathsf{inl}_A\ v\ \mathsf{of}\ \mathsf{inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2 \mapsto [v/x_1]e_1}\ \mathit{Case}'$$

$$\frac{}{\mathsf{case}\ \mathsf{inr}_A\ v\ \mathsf{of}\ \mathsf{inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2 \mapsto [v/x_2]e_2}\ \mathit{Case}''$$

(a) Eager Rules

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

$$\frac{e \mapsto e'}{\mathsf{fst}\ e \mapsto \mathsf{fst}\ e'}\ \mathit{Fst} \qquad \frac{}{\mathsf{fst}\ (e_1, e_2) \mapsto e_1}\ \mathit{Fst}' \qquad \frac{e \mapsto e'}{\mathsf{snd}\ e \mapsto \mathsf{snd}\ e'}\ \mathit{Snd} \qquad \frac{}{\mathsf{snd}\ (e_1, e_2) \mapsto e_2}\ \mathit{Snd}'$$

$$\frac{e \mapsto e'}{\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2 \mapsto \mathsf{case}\ e'\ \mathsf{of}\ \mathsf{inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2}\ \mathit{Case}$$

$$\frac{}{\mathsf{case}\ \mathsf{inl}_A\ e\ \mathsf{of}\ \mathsf{inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2 \mapsto [e/x_1]e_1}\ \mathit{Case}'$$

$$\frac{}{\mathsf{case}\ \mathsf{inr}_A\ e\ \mathsf{of}\ \mathsf{inl}\ x_1.\, e_1 \mid \mathsf{inr}\ x_2.\, e_2 \mapsto [e/x_2]e_2}\ \mathit{Case}''$$

$$\frac{e = \mathsf{plus}\ \text{or}\ \mathsf{minus}\ \text{or}\ \mathsf{eq} \quad e_1 \mapsto e_1'}{e\ (e_1, e_2) \mapsto e\ (e_1', e_2)}\ \mathit{Arith}' \qquad \frac{e = \mathsf{plus}\ \text{or}\ \mathsf{minus}\ \text{or}\ \mathsf{eq} \quad e_2 \mapsto e_2'}{e\ (v_1, e_2) \mapsto e\ (v_1, e_2')}\ \mathit{Arith}''$$

(b) Lazy Rules

$$\frac{e_1 = \mathsf{plus}\ \text{or}\ \mathsf{minus}\ \text{or}\ \mathsf{eq} \quad e_2 \mapsto e_2'}{e_1\ e_2 \mapsto e_1\ e_2'}\ \mathit{Arith}$$

$$\frac{}{\mathsf{plus}\ (n_1, n_2) \mapsto n_1 +_{\mathit{Int}} n_2}\ \mathit{Plus} \qquad \frac{}{\mathsf{minus}\ (n_1, n_2) \mapsto n_1 -_{\mathit{Int}} n_2}\ \mathit{Minus}$$

$$\frac{n_1 =_{\mathit{Int}} n_2}{\mathsf{eq}\ (n_1, n_2) \mapsto \mathsf{true}}\ \mathit{Eq} \qquad \frac{n_1 \neq_{\mathit{Int}} n_2}{\mathsf{eq}\ (n_1, n_2) \mapsto \mathsf{false}}\ \mathit{Eq}'$$

$$\frac{}{\mathsf{fix}\ x\!:\!A.\, e \mapsto [\mathsf{fix}\ x\!:\!A.\, e/x]e}\ \mathit{Fix} \qquad \frac{e \mapsto e'}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mapsto \mathsf{if}\ e'\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2}\ \mathit{If}$$

$$\frac{}{\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mapsto e_1}\ \mathit{If}_{\mathsf{true}} \qquad \frac{}{\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mapsto e_2}\ \mathit{If}_{\mathsf{false}}$$

(c) Common Rules

Figure 1: The operational semantics of TML

Programming instruction

Download the zip file `hw7.zip` from the course webpage or by running `receive` on the server `141.223.163.223`, and unzip it on your working directory. It will create a bunch of files.

First see `tml.ml` which declares a structure `Tml`. The types `tp` and `texp` correspond to the syntactic categories type and expression, respectively, and the type `exp` corresponds to the expression of the abstract syntax for TML with de Bruijn indice.

```
exception NotImplemented
exception Stuck

type var = string
type index = int

type tp =                                (* types *)
    Bool                                   (* bool *)
  ...

type texp =                              (* expressions *)
    Tvar of var                            (* variable *)
  ...

type exp =                               (* exps with de Bruijn indices *)
    Ind of index                           (* variable *)
  | Lam of exp                             (* lambda abstraction *)
  ...
```

Next see `hw7.mli` and `hw7.ml`. The goal is to implement three functions as follows: `texp2exp` takes a TML expression of type `Tml.texp` and returns its translated expression of type `Tml.exp`; `stepv` and `stepn` take an expression e of type `Tml.exp`, and return another expression e' that e reduces to, while an exception `Stuck` is raised if there is no such expression e'. The function `stepv` uses the call-by-value strategy, and `stepn` uses the call-by-name strategy, You may need to implement some auxiliary functions for substitutions with de Bruijn indices.

```
(* translate TML expressions into expressions with de Bruijn's index *)
val texp2exp : Tml.texp -> Tml.exp

(* one-step reduction using eager evaluation; raises Stuck if impossible *)
val stepv : Tml.exp -> Tml.exp

(* one-step reduction using lazy evaluation; raises Stuck if impossible *)
val stepn : Tml.exp -> Tml.exp
```

Remark. You have to think about how to represent the naming context which is required to implement this function. Think about it and create your own naming context. Then implement the function `texp2exp` correctly. Otherwise never touch the other parts of this assignment, because they depend on this function.

After implementing the functions `texp2exp`, `stepv`, and `stepn` in `hw7.ml`, run `make` to compile the sources files. There are two ways to test your code. First you can run `mainD` (for `texp2exp`), `mainV` (for `stepv`), or `mainN` (for `stepn`). At the TML prompt, enter a TML expression followed by a semicolon. (The syntax of TML will be given shortly.)

```
$ ./mainD
Tml> fun x : (int -> int) -> x y z;
(lam. ((0 1) 2))
Tml> if a then b else c a;
(if 0 then 1 else (2 0))
Tml> ^C
$ ./mainV
Tml> snd ((fun x : int -> x) (fun x : int -> x), 2);
(snd (((lam. 0) (lam. 0)),<2>))
Press return:
(snd ((lam. 0),<2>))
Press return:
<2>
Press return:
Tml> ^C
$ ./mainN
Tml> snd ((fun x : int -> x) (fun x : int -> x), 2);
(snd (((lam. 0) (lam. 0)),<2>))
Press return:
<2>
Press return:
```

Alternatively you can use those functions in `loop.ml` in the interactive mode of OCaml. At the OCaml prompt, type `#load "lib.cma";;` to load the library for this assignment. Then open the structure `Loop` and `Hw7`. You can test your implementation for `stepv` as follows:

- `loop (step stepv (wait show))`: Display the reduction sequence step by step.
- `loop (step stepv show)`: Display the entire reduction sequence at once.
- `loop (eval stepv show)`: Display the final result only.
- Use `loopFile` to read in a file.

To test your `stepn` function, use `Hw7.stepn` instead of `Hw7.stepv`. Here are some examples:

```
$ ocaml
       OCaml version 4.05.0

# #load "lib.cma";;
# open Loop;;
# open Hw7;;
# loop (step stepv (wait show));;
Tml> (fun x : (int -> int) -> x) (fun y : int -> y z);
((lam. 0) (lam. (0 1)))
Press return:
(lam. (0 1))
Press return:
Tml> ^C
# loop (eval stepv show);;
Tml> fst (((fun x : int -> x) (+ (100,200))), 2);
<300>
Tml>
```

The Syntax of TML

The concrete syntax for TML is defined as a subset of OCaml. All entries in the definition below are arranged in the same order that their counterparts in the abstract syntax above; e.g., `match` e `with inl` x `->` e `|` `inr` x `->` e is related to case e of inl $x.\,e$ | inr $x.\,e$.

type	A	::=	`bool` | `int` | A `->` A | A `*` A | `unit` | A `+` A | `(`A`)`
expression	e	::=	x | `fun` x `:` A `->` e | e e | `(`e`,` e`)` | `fst` e | `snd` e | `()`
		|	`fix` x `:` A `->` e | `inl (`A`)` e | `inr (`A`)` e
		|	`match` e `with inl` x `->` e `|` `inr` x `->` e
		|	`true` | `false` | `if` e `then` e `else` e | $\widehat{n}$ | `+` | `-` | `=`
		|	`let` x `:` A `=` e `in` e' | `let rec` x `:` A `=` e `in` e'
		|	`(`e`)`
integer	$\widehat{n}$	::=	`0` | `1` | `2` | $\cdots$

`(`A`)` is the same as A, and is used to alter the default right associativity of `->`. For example, A_1 `->` A_2 `->` A_3 is equal to A_1 `->` `(`A_2 `->` A_3`)`, not `(`A_1 `->` A_2`)` `->` A_3. Similarly `(`e`)` is the same as e, and is used to alter the default left associativity of applications. For example, e_1 e_2 e_3 is equal to `(`e_1 e_2`)` e_3, not e_1 `(`e_2 e_3`)`. We add two forms of syntactic sugar:

`let` x `:` A `=` e `in` e'	for	`(fun` x `:` A `->` e'`)` e
`let rec` x `:` A `=` e `in` e'	for	`(fun` x `:` A `->` e'`) (fix` x `:` A`.` e`)`

Submission Instruction

1. Make sure that you can compile `hw7.ml` by running `make`.
2. When you have the file `hw7.ml` ready for submission, run the `handin` command in the same directory, and your file will be submitted automatically.