Lambda calculus interpreter in Ocaml

Programming languages development

Martín do Río Rico `martin.dorio`
Yago Fernández Rego `yago.fernandez.rego`

December 8, 2023

Contents

1 Introduction **2**

2 Sections **2**

2.1 Upgrades to lambda-expressions input . . . 2
2.1.1 Multi-line expressions . . . 2
2.1.2 Pretty-printer . . . 2
2.2 New functionalities . . . 3
2.2.1 Recursivity . . . 3
2.2.2 Global context . . . 4
2.2.3 String type . . . 4
2.2.4 Tuples . . . 5
2.2.5 Registers . . . 5
2.2.6 Variants . . . 5
2.2.7 Lists . . . 6
2.2.8 Subtyping . . . 8

1 Introduction

This manual aims to explain the functionalities implemented in the evaluator as a result from the work carried out in this assignment.
We also aim to explain in relatively broad strokes the modifications made from the original code, acompanied by an explanation of those changes.

2 Sections

2.1 Upgrades to lambda-expressions input

2.1.1 Multi-line expressions

Following Ocaml's conventions we decided to use a double semi-colon ';;' as an expression-breaking symbol.
In order for this to work, we implemented a new function ('read_input') on 'main.ml', where we take the input string and keep apending all new lines to a list until the breaking character is found, this function has been named.
This list is then tokenized and passed as a parameter to eval, as in the given code.

2.1.2 Pretty-printer

This section has been done by implementing a new function 'printer' that is called on 'execute' on 'lambda.ml'.
This function then discerns between types and terms and calls the respective function: 'print_type' or 'print_term'.
Both have been implemented as a recursive call of several functions that call eachother following the syntax of the language.
With this we were able to reduce the number of unnecesary parenthesis, since it can better analyze the structure of the term/type passed to it.

Furthermore, following the recomendations given by the professor, we used Ocaml's 'Format' module in order to properly index the printing results. This has been done by calling 'open_box' and 'close_box' as the structure of the data passed deepens.

As such, the results are as following:

```
>> letrec sum : Nat -> Nat -> Nat =
    lambda n : Nat. lambda m : Nat. if iszero n then m else (succ (sum (pred n) m))
;;


sum : Nat -> Nat -> Nat =
 lambda n:Nat.
  lambda m:Nat.
   if iszero n then m else
    succ
     (((fix
```

```
        lambda sum:Nat -> Nat -> Nat.
         lambda n:Nat.
          lambda m:Nat. if iszero n then m else succ ((sum (pred n)) m))
       (pred n))
      m)
```

2.2 New functionalities

2.2.1 Recursivity

The internal fixed point combiner has been implemented to let the user declare recursive functions in a direct manner, A new token, ‘LETREC‘, has been added to the syntax of the interpreter, so that it is possible for the interpreter to indetify that the function being called is recursive.
Following the summary of rules given to us, we implemented the general recursion rules, with the new building case called ‘TmFix‘ that will take a term as argument.

As requested, three functions have been implemented, to verify the behaviour of the new syntax:

```
letrec sum: Nat -> Nat -> Nat =
    lambda n : Nat. lambda m : Nat.
    if iszero n then m
    else succ (sum (pred n) m)
;;

letrec prod: Nat -> Nat -> Nat =
    lambda n : Nat. lambda m : Nat.
    if iszero n then 0
    else
      if iszero m then 0
   else sum n (prod n (pred m))
;;

letrec fact: Nat -> Nat -> Nat =
    lambda n : Nat. lambda m : Nat.
    if iszero m then 1
    else prod n (fact n (pred m))
;;

letrec fib: Nat -> Nat =
    lambda n : Nat.
    if iszero n then 0
    else
      if iszero (pred n) then 1
   else sum (fib (pred n)) (fib (pred (pred n)))
;;
```

2.2.2 Global context

The ability to associate free variables with values or terms has been added.
This has been implemented by adding a new type, context ‘(string * binding) list‘ and the binding type ‘—TyBind of type —TyTmBind of (ty * term)‘ this second type has been made in order for the context to support adding both terms and types to the context.

In order for this context to be available to the code, this list is added to the inputs of eval, starting as an empty list and adding each of the new bindings.

This addition is made by adding two new cases on the execute function, that, in case of being activated, adds a new term to the context (’addbinding’) or a new type (’addtbinding’). This pattern-matching is archieved by adding two new cases to the parser, BindTm BindTy.
The syntax to add new variables would be:

```
>> x = 1;;
x : Nat = 1

>> N = Nat;;
N : type = Nat
```

2.2.3 String type

The new type ‘String‘ is implemented.
Both the new type itself ‘TyString‘ and a new term ‘TmString‘ were added, as well as several modifications on the lambda.ml so that pattern-matchings were exhaustive.
In addition, we created a new function, ‘concat‘ that concatenates two strings. This has been done by adding a new token ‘CONCAT‘, that invokes the corresponding functions on the lambda.ml file.

```
>> concat "hello " "world";;
- : String = hello world
```

Similarly, we implemented a ‘Char‘ type in a similar fashion, but using ” as markers instead of ”” and we used it in the function ‘first‘, that takes a String an returns its first Char.
As well as a ‘sub‘ function that returns the substring of a given String, that is, the string without its first character:

```
>> first "Ocaml";;
- : Char = O

>> sub "Ocaml";;
- : String = caml
```

2.2.4 Tuples

We chose to implement tuples rather than pairs. In order for any number of elements to be supported, we implemented the type TyTuple as a list of types, and TmTuple as a list of terms. From the user's point of view, tuples are written by encapsulating a list of elements, separated by commas in curly brackets.

In order for this syntax to be recognized by the interpreter, new tokens were created on the lexer, namely LCURLY ({) and RCURLY (}) and new cases have been added to the parser.ml so that it can recursively recognize the contents of the tuples.

The usage would be as follows:

```
>> {1,2,3};;
- : {Nat, Nat, Nat} = {1, 2, 3}
```

2.2.5 Registers

We implemented registers in a similar fashion as tuples, but instead of a list of terms in the case of the TmRegister and a type list in the case of TyRegister, we use a list of (string * term) and a list of (string * type) respectively.
This is done so that we can store the tags of the respective elements.

Similarly to the tuples implementation, we needed to add a new case on the parser so that the contents of a given register can be parsed.

As such, the usage would be:

```
>> {x=1, y=2, z=3};;
- : {x=Nat, y=Nat, z=Nat} = {x=1, y=2, z=3}

>> {};;
- : {} = {}
```

2.2.6 Variants

In a similar fashion to registers, we create a new type TyVariant (string * ty) list and for the labels we created TmLabel (string * term * string)

```
>> Int = <pos:Nat, zero:Bool, neg:Nat>;;
Int : type = <pos : Nat, zero : Bool, neg : Nat>

>> p3 = <pos=3> as Int;;
p3 : <pos : Nat, zero : Bool, neg : Nat> = <pos = 3>
```

We have not been able to implement the ‘case‘ functionability, thus we cannot implement the requested functions for this section.

2.2.7 Lists

Lists have been implemented as ‘ty * term * term‘, storing only one type Also, some other terms have been added:

- ‘TmIsEmpty‘: to check if a given list is empty
- ‘TmEmptyList‘: the representation of an empty list
- ‘TmHead‘: head of a list
- ‘TmTail‘: tail of a list

The corresponding syntax would be as following:

```
>> list = l.[Nat] 1 (l.[Nat] 2 (l.[Nat] 3 (l.[Nat] 4 (null[Nat]))));;
list : [Nat] = [1, 2, 3, 4]

>> head[Nat] list;;
- : Nat = 1

>> tail[Nat] list;;
- : [Nat] = [2, 3, 4]

>> isEmpty[Nat] list;;
- : Bool = false

>> empty = null[Nat];;
empty : [Nat] = []

>> isEmpty[Nat] empty;;
- : Bool = true
```

Moreover, the requested functions have also been implemented:

```
null[Nat];;

lst = l.[Nat] 1 (l.[Nat] 2 (l.[Nat] 3 (l.[Nat] 4 (null[Nat]))));;

head[Nat] lst;;

lst2 = l.[Nat] 5 (l.[Nat] 6 (l.[Nat] 7 (l.[Nat] 8 (null[Nat]))));;

tail[Nat] lst2;;


letrec len : ([Nat]) -> Nat = lambda l : [Nat]. if (isEmpty[Nat] l) then 0 else
    (succ (len (tail[Nat] l)))
in len lst;;

f = lambda x:Nat . succ x;;

letrec map : [Nat] -> (Nat -> Nat) -> [Nat] =
lambda lst: [Nat]. lambda f: (Nat -> Nat).
       if (isEmpty[Nat] (tail[Nat] lst)) then
              l.[Nat] (f (head[Nat] lst)) (null[Nat])
       else
              l.[Nat] (f (head[Nat] lst)) (map (tail[Nat] lst) f)
in map lst f;;



letrec append: [Nat] -> [Nat] -> [Nat] =
lambda l1: [Nat]. lambda l2: [Nat].
       if isEmpty[Nat] l1 then
              l2
       else
              l.[Nat] (head[Nat] l1) (append (tail[Nat] l1) l2)
in append lst lst2;;
```

2.2.8 Subtyping

In order to implement this section, we created a new function ‘subtype‘ that checks if two types have a subtyping relation.
This function is called on the ‘typeof‘ function in the lambda.ml, in the cases of TmApp, TmFix and TmList.
On registers this is not necesary since polimorfism was already implemented from the beginning, since it stored a list of (string * types), different types can be stored.

```
(* tuples do not have subtyping applied to them *)
>> (L x:{}. 1) {1};;
type error: parameter type mismatch
>> (L x:{Nat}. 1) {1};;
- : Nat = 1

(* every register has {} as a supertype *)
>> (L x:{}. 1) {};;
- : Nat = 1
>> (L x:{}. 1) {x=1};;
- : Nat = 1
```

Also, as requested, two lambda expressions containing subtyping operations are given:

```
let id = lambda r : {}. r;;

(L x:{x:Nat}. x.x) {x=1,y=2};;
(L x:{x:Nat}. x.x) {x=1,y=2,z=3};;

f = lambda r:({}->Nat). r;;
```