MANUAL

Lambda-Calculus Evaluator

Authors: José Manuel Amestoy López - manuel.amestoy@udc.es

Anxo Abuelo Veira - anxo.abuelo@udc.es

Introduction:

This technical manual shows the implementations that have been modified and the changes made to incorporate new functionalities. The following sections describe the modifications we have made to the original code, the newly added features, and the reasons behind these changes.

1.1.-Multi-line expressions:

The original code could only recognize single-line expressions. To enable the recognition of expressions spanning multiple lines, we modified the main.ml file. We added the function read_until_double_semicolon. This function reads the input until it encounters ";;" removes line breaks thereafter, and analyzes the input for execution.

Example:

"esto

es

una

multilinea";;

In variants, we have added rules to the parser to recognize the grammar of variants and have attempted to make use of BindTy, but we have not been able to get it to work. Therefore, variants are not implemented in our interpreter.

1.2.-More comprehensive "*Pretty-printer*":

To implement the pretty printer, we have made numerous modifications to both `string_of_type` and `string_of_term`. Additionally, we have added line breaks for `Eval`, `Bind`, `else`, and `TmAbs`. Furthermore, we have created a function to achieve a cascading effect by adding spaces corresponding to the number of line breaks before each new line.

Example:

- : 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)))

- : Nat -> Nat -> Nat =

lambda n : Nat.

lambda m : Nat. I

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))

2.1.-Internal fixed-point combinator:

We have introduced the internal fixed-point combinator, enabling us to define functions in a directly recursive manner without the need for additional auxiliary functions. In the parser.mly file, we made modifications by adding a new token called LETREC along with the necessary grammar rules for its proper functioning. These changes are implemented in the term field, allowing the interpreter to specifically identify when the invoked function is recursive. The LETREC implementation is summarized by invoking TmLetIn, with one of its arguments being TmFix, a new term crafted to represent the function calling itself.

After incorporating these changes in the parser.mly file, we made adjustments in the lambda.ml file, focusing particularly on the typeof and eval functions. In the eval function, we perform the evaluation of the term until a value is encountered, and if the term is an abstraction, it needs to be substituted with the given parameter for that abstraction. In the typeof function, we ensure that the type of the term matches the expected type. Additionally, we added the corresponding implementations in the free_vars and subst functions.

Examples:

Multiplication:

```
letrec sum : Nat -> Nat -> Nat =
  lambda n : Nat. lambda m : Nat. if iszero n then m else succ (sum (pred n) m)
in
  letrec prod : Nat -> Nat -> Nat =
    lambda n : Nat. lambda m : Nat. if iszero m then 0 else sum n (prod n (pred m))
  in
    prod 12 5
  ;;
```

Fibonacci:

```
letrec sum : Nat -> Nat -> Nat =
  lambda n : Nat. lambda m : Nat. if iszero n then m else succ (sum (pred n) m) in
    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))) in
            fib 10;;
```

Factorial:

```
letrec sum : Nat -> Nat -> Nat =
  lambda n : Nat. lambda m : Nat. if iszero n then m else succ (sum (pred n) m)
in
  letrec prod : Nat -> Nat -> Nat =
    lambda n : Nat. lambda m : Nat. if iszero m then 0 else sum n (prod n (pred m))
  in
    letrec factorial : Nat -> Nat =
     lambda n : Nat.
       if iszero n then 1
       else prod n (factorial (pred n))
    in
    factorial 5;;
```

2.2.-Context of global definitions:

We've enhanced our interpreter by introducing capabilities that enable the association of free variable names with values or terms. To begin, we've modified the implementation in lambda.ml by introducing the 'type' command, which serves three purposes:

- 'Eval': for the evaluation of terms.

- 'Bind': for binding a name to a term.

- 'BindTy': , for binding a name to a type.

Both 'eval' and 'eval1' have undergone changes to include a context as a parameter. Additionally, the 'TmVar' rule has been incorporated to retrieve the value assigned to a variable. We've introduced a new function named 'apply_ctx,' designed to substitute the name associated with a value with the actual value during term evaluation.

At this point, we've introduced the 'execute' function, which enables the interpreter to distinguish between 'Eval', 'Bind' and 'BindTy' executing the corresponding instructions. In the main file, it's crucial to consider the new context and pass it as a parameter to the main loop function.

Examples:

```
a = "abc";;
a;;
```

2.3.-Type String and Operation : length

For the definition of the String type, we declare a type called TyString and introduce two new terms: TmString for evaluating strings and TmConcat for string concatenation. To recognize these operations and types, two new strings ("String" and "concat") will be added to the lexer to create new tokens used by the parser (CONCAT and STRING).

The length function is used to determine the length or the number of characters in a String. You can apply it to a given String and it will return an integer value representing the number of characters in that string. This operation is based on ocaml's String.length implementation.

Example:

length "abc";;

length "";;

length "holaquetal";;

concat “abc” “dfg”;;

2.4.-Tuples

For the incorporation of tuples, we added a new type, TyTuple. We also introduced two new terms, TmTuple and TmPro. The second one is used for performing projection operations, which involve accessing an element of a tuple. The operations related to TmTuple and TmPro are in lambda.ml, and their definitions are based on lists and operations on lists. To implement TmPro, we utilized the List.nth function, which requires an integer to access the tuple. Additionally, we modified the parser.mly by adding a new rule, pathTerm, which evaluates whether a projection is being performed on a pathTerm.

Example:
tupla = {37, "hola", false, {2,5}};;

tupla.2;;

tupla.3.1;;

2.5.-Records

We defined the record type as TyReg of (string * ty) list, which is an implementation very similar to tuples. In this case, we have TmReg, where the difference lies in how it is represented inside the brackets, where it is necessary to indicate the type of the elements using ":". We leveraged the implementation of TmPro for records, but instead of using List.nth, we used List.assoc and provided the value of the element we want to retrieve. In addition to the implementation in lambda.ml, we added rules in the parser for records.

Example:

a=5;;

b= {x=31, y=a};;

b.x;;

2.7.-Variants

In variants, we have added rules to the parser to recognize the grammar of variants and have attempted to make use of BindTy, but we have not been able to get it to work. Therefore, variants are not implemented in our interpreter.

2.8.-Lists

For the implementation of lists, we will define a type TyList for the list type. Similarly, we will define two terms for list construction: TmCons and TmNil. For evaluation, we will use the terms TmIsEmpty, TmHead, and TmTail to check if a list is empty, obtain the head of a list, and retrieve the tail of a list, respectively. TmIsEmpty will return true or false depending on whether the list is empty, TmHead will return the head of the list with its type, and finally, TmTail will return a list with the elements that form the tail of the given list as a parameter.

Example:

lstnil = nil[Nat];;

x = 4;;

lst1 = cons[Nat:x, (cons[Nat:3 , (nil[Nat])])];;

head[Nat] lst1;;

tail[Nat] lst1;;

isempty[Nat] lstnil;;

Tamaño de una lista

```
letrec len : (list[Nat]) -> Nat =
  lambda lst : list[Nat].
    if (isempty[Nat] lst) then 0
    else (succ(len(tail[Nat] lst)))
in len lst1;;
```

Aplicar función a una lista

```
letrec map : (Nat -> Nat) -> list[Nat] -> list[Nat] =
  lambda f : (Nat -> Nat).
  lambda lst : list[Nat].
    if isempty[Nat] lst then
       (nil[Nat])
    else
       cons[Nat:(f (head[Nat] lst)),(map f (tail[Nat] lst))]
in map (lambda x : Nat. succ x) lst1;;
```

Concatenación de dos listas

```
letrec append : list[Nat] -> list[Nat] -> list[Nat] =
  lambda lst1 : list[Nat].
  lambda lst2 : list[Nat].
    if isempty[Nat] lst1 then
      lst2
    else
      cons[Nat:(head[Nat] lst1),(append (tail[Nat] lst1) lst2)]
in append lst1 lst2;;
```

2.9.-Subtyping

To introduce subtyping, we have created a new function in lambda.ml called subtypeof. This function replaces typeof in TmApp, TmFix and TmCons and checks whether two given types are subtypes of each other.

Example:

```
a = {x=1, y=1, z={x=1}};;

b = {x=1, y=1, z=a};;
```