LAMBDA CALCULUS INTERPRETER

Estela Pillo González

Design of Programming Languages

Year 2024/2025

Contents

1 INTRODUCTION

The objective of this project is to improve a lambda-calculus interpreter. This manual will serve as a guide, containing a summary of the new features and test examples. It will also explain how the features have been developed. All the examples present in this document, as well as the expected results, are available in `examples.txt` and `examplesresults.txt`.

2 USER MANUAL

2.1 Improvements in the Introduction and Writing of Lambda Expressions

2.1.1 Recognition of Multi-line Expressions.

In the current code, expressions can be entered across multiple lines, as the Enter key is no longer interpreted as the end of an expression. A new symbol, semicolon (;;), has been introduced to mark the end of expressions.

```
>>let
    x = 1
in
    succ x;;
- : Nat = 2
```

2.1.2 Implementation of a More Complete "Pretty-Printer"

With this new implementation, the number of unnecessary parentheses is reduced. Additionally, to improve readability, indentations and line breaks have been added.

```
>>lambda x : Nat. x;;
- : Nat-> Nat = lambda x:Nat. x
```

```
>>letrec sum : Nat -> Nat -> Nat =
                lambda n : Nat. lambda m : Nat. if iszero n
                    then m else succ (sum (pred n) m)
        in 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 Extensions to the Lambda-Calculus Language

2.2.1 Addition of an Internal Fixed-Point Combinator

The introduction of the internal fixed-point combinator allows us to define recursive functions without the need for auxiliary functions. To do this, we will use ”letrec”, followed by the function we wish to define.

```
>>let prod =
        lambda n : Nat. lambda m : Nat.
            (letrec aux : Nat -> Nat -> Nat -> Nat = lambda
                n : Nat. lambda m : Nat. lambda k : Nat.
                if iszero n then 0
                else if iszero (pred n) then m
                    else (aux (pred n) (letrec sum : Nat ->
                        Nat -> Nat =
                                    lambda i : Nat. lambda
                                        j : Nat. if iszero i
                                        then j else succ
                                        (sum (pred i) j)
                                    in sum m k) k)
        in aux n m m)
    in prod 2 16;;
- : Nat =  32
```

```
>>let fib =
        lambda n : Nat.
            (letrec aux : Nat -> Nat -> Nat -> Nat = lambda
               n : Nat. lambda a : Nat. lambda b : Nat.
                if iszero n then a
                else (aux (pred n) (letrec sum : Nat -> Nat
                   -> Nat =
                                    lambda i : Nat. lambda
                                       j : Nat. if iszero i
                                       then j else succ
                                       (sum (pred i) j)
                                    in sum a b) a)
            in aux n 0 1
            )
    in fib 8;;
- : Nat =  21

>>let fact =
        lambda n : Nat.
            (letrec factaux : Nat -> Nat -> Nat = lambda n
               : Nat. lambda acc : Nat.
                if iszero (pred n) then acc
                else (factaux (pred n) (
                    letrec prod : Nat -> Nat -> Nat -> Nat
                       = lambda p : Nat. lambda q : Nat.
                       lambda k : Nat.
                        if iszero p then 1
                        else if iszero (pred p) then q
                            else (prod (pred p) (letrec sum
                               :  Nat -> Nat -> Nat =
                                    lambda i : Nat. lambda
                                       j : Nat.
                                    if iszero i then j else
                                       succ(sum (pred i) j)
                                in sum q k) k)
                    in prod n acc acc))
            in factaux n 1)
    in fact 5;;
- : Nat =  120
```

2.2.2 Addition of a Global Definitions Context

With this new functionality, we can associate names of free variables with values or terms so that they can be used in subsequent lambda expressions. The syntax follows the form: identifier = term. This functionality also allows for type definitions, with the syntax being id = type.

```
>> y = 3;;
y : Nat = 3

>> x = 2;;
x : Nat =  2

>> letrec sum: Nat -> Nat -> Nat =
        lambda n : Nat. lambda m : Nat. if iszero n then m
            else succ(sum (pred n) m)
    in sum x y;;
- : Nat = 3

>> succ(y);;
- : Nat = 4

>> x = true;;
x : Bool = true

TYPES:

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

>> X = Bool;;
type X = Bool

>> let id_bool = L x:X. x in id_bool true;;
- : X = true
```

2.2.3 Addition of the String Type

The String type is introduced to support characters, along with the operations “concat” and “length”. Concat allows the concatenation of two Strings. On the other hand, applying length to a String will return a value representing the number of characters in that String.

```
>> "a";;
- : String = "a"

> x="abc";;
x : String = "abc"

>> length x;;
- : Nat =  3

>> concat "a" "b";;
- : String = "ab"

>>  lambda s : String. s ;;
- : (String) -> (String) = (lambda s:String. s)

>> (lambda s : String. s) "abc";;
- : String = "abc"

>>    letrec replicate : String -> Nat -> String =
        lambda s : String. lambda n : Nat.
            if iszero n then "" else concat s (replicate s
                (pred n))
    in
        replicate "abc" 3
    ;;
- : String = "abcabcabc"
```

2.2.4 Addition of Tuples

The addition of tuples allows the creation of lists with any number of elements, which are enclosed in square brackets and separated by commas. Projection operations are also implemented to access the different elements of the tuples.

```
 >> x = {1,2,3,"hola"};;
 x : {Nat, Nat, Nat, String} = {1, 2, 3, "hola"}

 >> y = {"uno","dos"};;
 y : {String, String} = {"uno", "dos"}

 >> tuple2 = {x,y};;
 tuple2 : {{Nat, Nat, Nat, String}, {String, String}} =
     {{1, 2, 3, "hola"}, {"uno", "dos"}}

>> tuple2.2;;
- : {String, String} = {"uno", "dos"}

>> tuple2.1;;
- : {Nat, Nat, Nat, String} = {1, 2, 3, "hola"}

>> tuple2.1.1;;
- : Nat = 1

>> f = L x:Nat.x;;
f : Nat-> Nat =  lambda x : Nat.  x

>> g = L x:String.x;;
g : String-> String =  lambda x : String.  x

>> tuple = {f 3, g "hola", "mundo"};;
tuple : {Nat,  String,  String} =  {3,  "hola",  "mundo"}
```

2.2.5 Addition of Records

In this section, records (finite sequences of fields of any type, labeled) are implemented, along with their projection operations, based on the field labels. Projection operations are also included, which allow obtaining the value of the elements in the record.

```
>> a=5;;
a : Nat = 5

>> b={x=31,y=a};;
b : {x:Nat,y:Nat} = {x : 31, y : 5}

>> b.x;;
- : Nat = 31

>> test1 = {x = 2, y = "hola", z = 77};;
test1 : {x:Nat,y:String,z:Nat} = {x : 2, y : "hola", z : 77}

>> test1.y;;
- : String = "hola"
```

2.2.6 Addition of Variants

For this section, variants are introduced, which constitute a tagged generalization of binary sum types.

```

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

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

>> z0 = <zero=true> as Int;;
z0 : <pos : Nat,  zero : Bool,  neg : Nat> =  <zero =
    true> as Int

>> n5 = <neg=5> as Int;;
n5 : <pos : Nat,  zero : Bool,  neg : Nat> =  <neg =  5> as
    Int
```

```

>> abs = L i : Int.
case i of
<pos=p> => (<pos=p> as Int)
| <zero=z> => (<zero=true> as Int)
| <neg=n> => (<pos=n> as Int);;
abs : <pos : Nat, zero : Bool, neg : Nat>->
 <pos : Nat, zero : Bool, neg : Nat> =
 lambda i : Int.
  case i of <pos = p> => <pos = p> as Int | <zero = z>
     =>
   <zero = true> as Int | <neg = n> => <pos = n> as Int

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

>> abs z0;;
- : <pos : Nat, zero : Bool, neg : Nat> = <zero = true>
    as Int

>> abs n5;;
- : <pos : Nat, zero : Bool, neg : Nat> = <pos = 5> as
    Int
```

2.2.7 Addition of Lists

The incorporation of lists allows the creation of finite sequences of elements of the same type. The typical operations to obtain the head, the tail, and check if it is empty are added. Additionally, the following three lambda expressions are implemented:

- **Length:** Recursively calculates the length of a list based on the sum.
- **Append:** Concatenates two lists.
- **Map:** Applies a function to the elements of a list and returns the list of resulting values.

```
>> lstnil = nil[Nat];;
lstnil : List[Nat] = nil[Nat]

>> x = 4;;
x : Nat = 4

>> lst1 = cons[Nat] 2 (cons[Nat] 3 (nil[Nat]));;
lst1 : List[Nat] =  cons[Nat] 2 (cons[Nat] 3 nil[Nat])

>> lst2 = cons[Nat] 4 (cons[Nat] 5 (nil[Nat]));;
lst2 : List[Nat] =  cons[Nat] 4 (cons[Nat] 5 nil[Nat])

>> lst3 = cons[Nat] x (nil[Nat]);;
lst3 : List[Nat] =  cons[Nat] 4 nil[Nat]

>> head[Nat] lst1;;
- : Nat = 2

>> tail[Nat] lst1;;
- : List[Nat] = cons[Nat:3,nil[Nat]]

>> isnil[Nat] lstnil;;
- : Bool =  true

>> letrec len : (list[Nat]) -> Nat =
  lambda lst : list[Nat].
    if (isempty[Nat] lst) then 0
    else (succ(len(tail[Nat] lst)))
in len lst1;;
- : Nat = 2

>>cons[Nat] 3 nil[Nat];;
- : List[Nat] =  cons[Nat] 3 nil[Nat]

>>cons[Nat] 5 (cons[Nat] 3 nil[Nat]);;
- : List[Nat] =  cons[Nat] 5 (cons[Nat] 3 nil[Nat])

MAP
>> letrec map : List[Nat] -> (Nat -> Nat) -> List[Nat] =
lambda lst: List[Nat]. lambda f: (Nat -> Nat).
        if (isnil[Nat] (tail[Nat] lst)) then
                cons[Nat] (f (head[Nat] lst)) (nil[Nat])
        else
                cons[Nat] (f (head[Nat] lst)) (map
                    (tail[Nat] lst) f)
in map lst1 f;;
- : List[Nat] =  cons[Nat] 2 (cons[Nat] 3 nil[Nat])
```

```

APPEND
>>letrec append: List[Nat] -> List[Nat] -> List[Nat] =
lambda l1: List[Nat]. lambda l2: List[Nat].
        if isnil[Nat] l1 then
                l2
        else
                cons[Nat] (head[Nat] l1) (append (tail[Nat]
                    l1) l2)
in append lst1 lst2;;
- : List[Nat] =
 cons[Nat] 2 (cons[Nat] 3 (cons[Nat] 4 (cons[Nat] 5
     nil[Nat])))

LENGTH
>> length = letrec len : (List[Nat]) -> Nat = lambda l :
    List[Nat]. if (isnil[Nat] l) then 0 else (succ (len
    (tail[Nat] l)))
in len lst1;;
length : Nat =  2
```

2.2.8 Addition of Subtyping

For this section, I have written a function that implements the subtyping polymorphism for records and functions (it checks if two types satisfy the subtyping relationship).

```
>> a = {x = 1, y = 1, z = {x = 1}};;
a : {x:Nat,y:Nat,z:{x:Nat}} = {x : 1, y : 1, z : {x : 1}}

>> b = {x = 1, y = 1, z = a};;
b : {x:Nat,y:Nat,z:{x:Nat,y:Nat,z:{x:Nat}}} = {x : 1, y :
    1, z : {x : 1, y : 1, z : {x : 1}}}

>> let idr = lambda r:{}. r in idr {x=0, y=1};;
- : {} = {x : 0, y : 1}

>> a = {x=1, y=1, z={x=1}};;
a : {x:Nat,y:Nat,z:{x:Nat}} = {x : 1, y : 1, z : {x : 1}}

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

3 TECHNICAL MANUAL

3.1 Improvements in the Introduction and Writing of Lambda Expressions

3.1.1 Recognition of Multi-line Expressions

Using the function `check_semicolon`, the input is read until a double semicolon is detected. Additionally, the function `remove_newlines` is used to correctly remove line breaks and treat them as part of a single expression.

3.1.2 Implementation of a More Complete "Pretty-Printer"

Specific functions are used to process different elements (`print_type`, `print_term`, etc.) to facilitate future changes and improve code structure. For printing more complex structures like tuples and records, the functions `print_tuple` and `print_record` were implemented. Terms were categorized into different groups for clarity:

- `print_atomic_term`
- `print_app_term`
- `print_path_term`

Functions such as `open_box`, `close_box`, and `print_space` were also used to enhance readability. In summary, the printing of different types and terms was separated as much as possible to improve the code's maintainability.

3.2 Extensions to the Lambda-Calculus Language

3.2.1 Addition of an Internal Fixed-Point Combinator

- `lexer.mll`: Added the keyword `letrec`, which passes the `letrec` token to the parser.
- `parser.mly`: Added the `letrec` token along with its corresponding grammatical rules.
- `lambda.ml`: Updated existing functions to handle the new term `TmFix`.
- `lambda.mli`: Added `TmFix` to the list of terms.

3.2.2 Addition of a Global Definitions Context

A new type, `command`, has been added to define the commands available in the system:

- Commands for evaluating expressions (`Eval`).
- Commands for binding a name to a term (`Bind`, value binding).

- Commands for binding a name to a type (TBind, type binding).

The binding type was also introduced to define the types of bindings associated with each variable in the context. These bindings can be of two types:

- TyBind: Associates a type with a variable.
- TyTmBind: Associates both a type and a term with a variable.

The context type was added to represent the execution environment. It is implemented as a list of tuples (string * binding), where each tuple contains a variable name (as a string) and its corresponding binding (either type or value). This allows for managing variable scopes effectively within the system.

Key functions introduced include:

- apply_ctx: Substitutes all free variables in a term with their corresponding values from the global context.
- execute: Differentiates between the various commands and executes the appropriate action based on the command type.

Additionally, the functions addbinding and getbinding were modified to handle both type and value bindings. This resulted in four distinct functions:

- addtbinding: Adds a type binding.
- addvbinding: Adds a value binding.
- gettbinding: Retrieves a type binding.
- getvbinding: Retrieves a value binding.

I created the type TmVar to represent variables and retrieve the associated values. The TmVar term is used to handle variable lookups within the context. This term is used in various parts of the code, such as in managing variable bindings in expressions like TmLetIn, where the variable is substituted, and also in cases where the variable's value is required for further evaluation.

I included the base_ty function to normalize complex types by reducing them to their base forms.

3.2.3 Addition of the String Type

To incorporate the String type, the following changes were made:

- Added the new type TyString.
- Introduced the terms TmString (to represent string literals) and TmConcat (for string concatenation).
- Implemented TmLength to calculate the length of a string.

All necessary modifications were made to the functions in lambda.ml to ensure exhaustive pattern matching. Additionally, the corresponding tokens (string, concat, and length) and parsing rules were added to the lexer and parser.

3.2.4 Addition of Tuples

To implement tuples in the lambda calculus system, the new type `TyTuple` was added to represent tuples. Additionally, the following terms were introduced:

- `TmTuple of term list`, which represents a tuple containing a list of terms.
- `TmProjection of term * string`, which represents the projection of a specific element from a tuple.

The functions in `lambda.ml` were updated to include exhaustive pattern matching for `TmTuple` and `TmProjection`, ensuring proper handling of tuple-related operations.

The lexer was modified by adding the following tokens to `lexer.mll` to handle the syntax for tuples:

- `LBRACE` for the opening brace.
- `RBRACE` for the closing brace.
- `COMMA` for separating tuple elements.

In the parser, the corresponding grammar rules were added to `parser.mly`. These include recognizing and constructing tuples using braces ({}) and commas (`,`), as well as rules for handling tuple projections.

3.2.5 Addition of Records

For the implementation of records, the `TyRecord` and `TmRecord` types were added to the `lambda.ml` and `lambda.mli` files.

The necessary pattern matching was implemented in the `free_vars`, `is_val`, `eval1`, `typeof`, and other functions in `lambda.ml`.

Additionally, the required rules were added in the `parser.mly` file to allow the parsing and construction of `TmRecord` and `TyRecord` type terms.

3.2.6 Addition of Variants

For the implementation of variants, the following types and terms were added:

- `TyVariant of (string * ty) list`
- `TmTag of string * term * ty`
- `TmCase of term * (string * string * term) list`

The necessary support was implemented in the corresponding functions to handle these new types and terms. Specifically, the pattern matching in functions such as `free_vars`, `is_val`, `eval1`, and `typeof` was updated to properly handle the variant types and terms.

Additionally, the required tokens and rules were added in the `parser.mly` file (`case`, `of`, `larrow`, `rarrow`, `as`, `pipe`, `doublearrow`) to enable the syntactic parsing of variant-related constructs.

3.2.7 Addition of Lists

For the implementation of lists, the types and terms `TmHead`, `TmTail`, `TmCons`, `TmNil`, `TmisEmpty`, and `TyList` were added to the `lambda.ml` and `lambda.mli` files, which allow for manipulating lists, obtaining the head and tail of lists, constructing lists, representing empty lists, and checking if a list is empty.
The necessary pattern matching was implemented in the corresponding functions to handle these terms in `lambda.ml`. Additionally, the tokens `LIST`, `CONS`, `NIL`, `HEAD`, `TAIL`, and `ISEMPTY` were added in the `parser.mly` file to enable the parsing of list-related constructs.

3.2.8 Subtyping Implementation

For the implementation of subtyping, I added the `subtypeof` function. This function recursively compares two types to determine if two specified types are subtypes of one another.
This function replaces `typeof` in `TmApp`, `TmFix`, and `TmCons`.