LAMBDA CALCULUS INTERPRETER

Rubén Fernández Farelo
Samuel Otero Agraso

Grupo de prácticas: 3.1

Diseño de Lenguajes de Programación
Curso 2024/25

Índice

1. Introduction

This user manual aims to explain in detail the implementation of the lambda calculus interpreter in OCaml, describing the internal workings of the code and the modules where the various functionalities of the project have been developed. Illustrative examples with ready-to-run code will also be provided, allowing the user to understand the behavior of the interpreter and experiment with the newly implemented functionalities.

2. Improvements in the introduction and writing of lambda expressions:

2.1. Recognition of Multi-line Expressions

A function called **read_multiline** was created in the main, which allows the user to continuously input until the termination symbol ;; is encountered. This function concatenates the input lines and removes the delimiter at the end. This ensures that the user can continue writing their expression over multiple lines without the interpreter prematurely evaluating it.

The implementation of this function is based on recursion, where the function calls itself to keep reading lines until the termination condition is met. Additionally, it has been improved to remove comments of the type (* ... *) from each line before processing it. This is achieved using a regular expression that searches for and replaces the comments with an empty string.

```
(*Ejemplo de un uso de una sentencia multilinea terminada en ;;*)
let x = 1
in
succ(pred(x));;
```

2.2. Implementation of a More Complete Pretty-Printer

The pretty-printer is an essential tool for improving the readability of code and lambda expressions in the interpreter's output. Its main goal is to minimize the use of unnecessary parentheses, ensuring that the result is presented clearly and concisely. To achieve this, a series of functions can be implemented to handle the different structures of expressions, deciding when it is really necessary to include parentheses according to the language's grammar.

2.2.1. Design of the Pretty-Printer

The pretty-printer is designed to improve the readability of lambda expressions by reducing unnecessary parentheses and applying an organized structure through the OCaml formatting library. Its key principles are:

- **Reduction of Parentheses:** To simplify reading, the pretty-printer minimizes the use of parentheses, relying on the hierarchy of expressions and structural organization, avoiding enclosing each sub-expression when not necessary.
- **Organization with Boxes and Indentation:** The functions open_box and close_box allow dividing each section into "boxes.ºr formatting zones, maintaining an organized structure and automatic indentation adjustment. These boxes dynamically adjust, improving presentation without additional parentheses.
- **Specific Printing Functions:** Each type of expression has its own printing function (string_of_ty, string_of_term, etc.), which evaluates whether parentheses are necessary based on the context, ensuring a clean output. For correct use, both a pretty printer for terms and types are used.

2.2.2. Implementation in `lambda.ml`

To implement the pretty-printer, the printing functions string_of_ty and string_of_term were modified to now return unit and perform formatting directly.

- **Structure in Sub-functions:** string_of_term is divided into specific sub-functions: appTerm_to_string handles function applications and concatenations (TmApp, TmConcat), while atomicTerm_to_string handles atomic terms like TmTrue, TmFalse, variables, and literals.

- **Reduction of Parentheses:** The sub-functions check the context they are in, using boxes and indentation instead of parentheses in complex expressions, simplifying the output.

2.2.3. Function `pretty_printer`

For screen output, pretty_printer uses string_of_ty and string_of_term to represent the type and value of a term. This function uses open_box, print_string, print_space to align the identifier and type, and force_newline to ensure each output is on a separate line.

2.2.4. Example

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

(*Nos da esta salida mas legible, de forma escalonada y con menos
    paréntesis*)

- :  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))
```

3. Extensions of the Lambda Calculus Language

3.1. Incorporation of an Internal Fixed-Point Combinator

The goal of this section is to extend the lambda calculus language with an internal fixed-point combinator that allows for the direct and more intuitive definition of recursive functions. This enables the user to declare recursive functions without manually using the fixed-point combinator fix in each recursive call. Instead of using a complex construction to define recursion (like fix sumaux in the example), it can be done through a more direct and readable declaration, similar to letrec, which allows for the declarative definition of recursive functions.

3.1.1. Parser and Lexer

- **New token LETREC**: A new token LETREC is introduced in the parser.mly file so that the language recognizes this keyword.
- **Derivation rule**: In parser.mly, a new derivation rule is established to identify and process expressions with letrec.
- **In lexer.mll**: This token is added so that the lexer recognizes its use in the source code.

3.1.2. Definition of the new term TmFix

- **In lambda.mli**: The new term TmFix is defined in the term type. This term represents the use of the fixed-point combinator and will allow the evaluation of recursive functions internally in the system.

3.1.3. Implementation of TmFix in lambda.ml

- **Typing (typeof)** : A new typing rule for TmFix is added. The rule verifies that the term to which TmFix is applied (t1) is of function type and that the output type matches the input type.
- **Pretty printer (string_of_term)** : To correctly visualize expressions that include TmFix.

- **Free variables (free$_vars$)** : $TheTmFixcaseisaddedtothefree\_varsfunctiontocorrectlyidentifyc$
  $Thesubstitutionrulef orTmFixisimplemented, allowingvariableswithinrecursiveexpressionstob$

- **Rule E-FIXBETA**: Defines that when evaluating a TmFix that applies a lambda term (TmAbs), the variable of the recursive function is replaced by the term itself, thus achieving the recursion effect.

- **Rule E-Fix**: Defines the evaluation of TmFix when a lambda term is not yet encountered. It recursively evaluates the sub-term (t1) until a lambda function is obtained to which E-FIXBETA can be applied.

3.1.4. Usage Examples

```
(*Ejemplo de la suma*)
letrec sum : Nat -> Nat -> Nat =
lambda n : Nat. lambda m : Nat. if iszero n then m
else succ (sum (pred n) m)
in
sum 25 25;;

(*Ejemplo del producto*)
letrec sum : Nat -> Nat -> Nat =
lambda n : Nat. lambda m : Nat. if iszero n then m else succ (sum
    (pred n) m) in
letrec mult : Nat -> Nat -> Nat =
lambda n : Nat. lambda m : Nat. if iszero n then 0 else sum (mult
    (pred n) m) m in
mult 3 4;;

(*Ejemplo de 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 (pred n))) (fib (pred n))
in fib 6;;
```

```
(*Ejemplo del 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 mult : Nat -> Nat -> Nat =
lambda n : Nat. lambda m : Nat. if iszero n then 0 else sum (mult
    (pred n) m) m in
letrec fac : Nat -> Nat =
lambda n : Nat. if iszero n then 1 else mult n (fac (pred n))
in fac 5;;
```

3.2. Incorporation of a Global Definitions Context

A global definitions context is implemented that allows associating variable names with specific terms and creating aliases for types, facilitating the reuse of identifiers and types in lambda expressions. The definition syntax is:

identifier=term

For example:

```
(* Contexto de definiciones globales*)
x = 5;;
succ x;;
f = lambda y : Nat. x;;
f 3;;

(* Definición de variables Alias *)
N = Nat;;
NinN = N -> N ;;
N3 = N -> NinN;;
letrec sum : N3 =
lambda n : Nat. lambda m : Nat. if iszero n then m
else succ (sum (pred n) m)
in
sum 5 5;;
```

Thus, expressions like `id x` can be evaluated correctly with the previous definitions.

3.2.1. Parser and Lexer

- **QUIT Token:** Added in `parser.mly` to allow the user to explicitly exit the interpreter.
- **Tokens for Type Aliases:** In `lexer.mll`, the `IDT` token is introduced to recognize type aliases that begin with uppercase letters, distinguishing them from variable identifiers (token `IDV`).
- **Derivation rules:** In `parser.mly`, rules are incorporated for:
  - Definition of variables with the pattern `IDV EQ term EOF` to associate identifiers with terms using `Bind`.
  - Definition of type aliases with the pattern `IDT EQ ty EOF` using `TBind`.
  - Evaluation of expressions (`term EOF`) using the `Eval` command.
  - The `QUIT EOF` instruction to exit the interpreter.

3.2.2. Modifications in `lambda.mli`

- **New command types:** `Eval`, `Bind`, `TBind`, and `Quit` are added to the `command` type to evaluate expressions, associate terms with names, define type aliases, and exit the interpreter.
- `binding` **type:** The `binding` type is extended to support type aliases (`TyBind`) and term associations (`TyTmBind`), allowing both terms and types to be stored in the context.
- **Context handling functions:** Binding functions are divided into:
  - `addtbinding` and `gettbinding` to handle type aliases.
  - `addvbinding` and `getvbinding` to handle terms associated with variable names.
- **Execute function:** Processes commands in the current context, displaying results to the user and updating the context, especially for the `TBind` command.

3.2.3. Implementation in `lambda.ml`

- **Command and binding types:** The `command` and `binding` types are implemented in `lambda.ml`, with a `TBind` case to allow type aliases.
- **Context functions:** The `addtbinding` and `addvbinding` functions register type aliases and terms in the context, while `gettbinding` and `getvbinding` retrieve these bindings according to the current context.
- **Context application in evaluation:** The `apply_ctx` function ensures that free variables in a term are resolved using the context, replacing them with the associated terms or types.
- **Execute function:** The `execute` function processes the `Eval`, `Bind`, and `TBind` commands, resolving type aliases and updating the context with `addtbinding`. The `Quit` command raises `End_of_file` to safely exit the interpreter.

3.3. Incorporation of the String Type

The `String` data type has been added to the system to handle character strings and perform concatenation operations. This allows defining string-type terms and combining them using the `concat` operation.

3.3.1. Parser and Lexer

- **New tokens STRING and CONCAT:** The `STRING` and `CONCAT` tokens were added in `parser.mly` to recognize expressions with strings and concatenation. A `STRINGV` token was also included to represent string values.
- **Derivation rules and string recognition:** In `parser.mly`, derivation rules were added to process concatenation expressions between terms and strings. In `lexer.mll`, rules were added to interpret strings delimited by quotes and remove the quotes when processing them.

3.3.2. Definition of New Terms

- **Term TmConcat:** In `lambda.mli`, the `TmConcat` term was added to the `term` type to handle the concatenation operation.

- **Type TyString:** The `TyString` type was added to represent strings in the type system.

3.3.3. Implementation in `lambda.ml`

- **Typing (typeof):** A typing rule for `TmConcat` was added to ensure that both terms to be concatenated are of type `TyString`.
- **Pretty Printer (string_of_term):** The pretty printer was extended to correctly display `TmString` and `TmConcat` expressions.
- **Free variables and substitution:** In `free_vars` and `subst`, the `TmString` and `TmConcat` cases were added to identify and handle free variables in concatenation expressions.
- **Evaluation (eval1):** Evaluation rules for `TmConcat` were defined, including cases where both strings are fixed, where one is an evaluable term, and where both require recursive evaluation.

3.3.4. Usage Examples

```
"Hola Mundo!";;
concat "Hola, " "mundo!";;
lambda s : String. s;;
(lambda s : String. s) "abc" ;;
```

3.4. Implementation of Tuples

In this section, tuples with support for any number of elements and projection operations based on position were added.

3.4.1. Parser and Lexer

In `parser.mly`, the `COMMA`, `LKEY`, and `RKEY` tokens were added to recognize the braces and commas ",". Rules were implemented for tuple terms and types that allow grouping sequences of elements within the braces. The sequences of elements are constructed recursively, allowing tuples to accept multiple types and terms in a single structure.

3.4.2. Modifications in `lambda.mli` and `lambda.ml`

The `TyTuple` and `TmTuple` types were defined in `lambda.mli` to represent tuple types and tuple terms, respectively. Several key modifications were made in `lambda.ml` to handle tuples:

- **Tuple printing:** In the `string_of_ty` and `string_of_term` functions, cases were added to display tuples in the format with elements separated by commas.
- **Type resolution and evaluation:** In `resolve_base_type` and `typeof`, rules were added to evaluate and type each element within a tuple, returning a `TyTuple` with the respective types.
- **Free variables and substitution:** The `free_vars` and `subst` functions were modified to process each element in tuples and apply operations recursively.
- **Tuple evaluation:** `eval1` was extended to apply evaluations to each element of the tuple in sequence, ensuring that each term is a value before continuing.

3.4.3. Usage Examples

```
(*Tuples*)

{5 ,true,"abc"};;
{5 ,true,"abc"}.1;;
{5 ,true,"abc"}.4;;
{5 ,{true,"abc"}};;
{5 ,{true,"abc"}}.2.1;;
t = {5 ,{true,"abc"}}.2.1;;
```

3.5. Implementation of Records

In this section, records were added, which are data structures that allow grouping labeled elements for efficient access to their values using specific names.

3.5.1. Parser and Lexer

In `parser.mly`, the tuple tokens such as `COMMA`, `LKEY`, and `RKEY` were reused, and the `DOT` and `EQ` tokens were added. The new `pathTerm` rule was introduced in place of `atomicTerm` within `appTerm`. This rule allows accessing record elements using the `.` syntax. Additionally, the possibility of defining records in the grammar was added through the `tmSequenceRecord` rule for terms and `tySequenceRecord` for types.

3.5.2. Modifications in `lambda.mli` and `lambda.ml`

The `TyRecord` type and the `TmRecord` and `TmGetElem` terms were defined to represent and manipulate records and access their elements.

- **Record printing:** In the `string_of_ty` and `string_of_term` functions, cases were added to display records in the `label=value` format, allowing their elements to be concatenated for easier visualization.
- **Type resolution and evaluation:** In `resolve_base_type` and `typeof`, rules were added to compute the types of each field in a record and return a `TyRecord` with the corresponding types. Additionally, in `TmGetElem`, checks were implemented to access fields of a record or elements of a tuple (in the case of numeric indices).
- **Free variables and substitution:** The `free_vars` and `subst` functions were extended to operate recursively on each element of a record, applying the necessary operations to each term within the structure.
- **Record evaluation:** In `eval1`, a procedure was implemented to evaluate the terms within a record in sequence, as well as a direct access rule to obtain the value of a specific element when the record is fully evaluated.

3.5.3. Usage Examples

```
{id=30,partido="Juventus vs Madrid"};;
r = {text="Hola", n={x=5,y=10}, label=true};;
r.text;;
r.n;;
r.n.x;;

{x = 2, y = 5 , z= 0};;
{x = 2, y = 5 , z= 0}.x;;
r = {x = 2, y = 5 , z= 0}.x;;

p = {na = {"Luis" ,"vidal"},e = 28 };;
p.na;;
p.na.1
p.e;;
```

3.6. Implementation of Variants

In this section, variants were added, which are a tagged generalization of binary sum types, allowing the definition of types with multiple labeled options associated with specific data. This enriches the language with a more flexible structure for modeling data.

3.6.1. Parser and Lexer

New tokens (`CASE`, `AS`, `OF`, among others) were added to support the syntax of variants. In the parser, rules were implemented to:

- Define variants in the type grammar as labeled lists (`TyVariant`).
- Create terms that represent variant labels (`TmTag`).
- Handle multiple patterns with `case` for variant destructuring (`TmCase`).

3.6.2. Modifications in `lambda.mli` and `lambda.ml`

New terms and types The following were added:

- `TyVariant`, to define variants as lists of labels with their associated types.
- `TmTag`, to represent a specific label within a variant.
- `TmCase`, to evaluate expressions based on labeled patterns.

Changes in main functions

- **Printing:** The text conversion functions were updated to clearly display variants, labels, and patterns.
- **Type resolution:** Support was added to normalize labels and types in variants.
- **Type checking:**
  - In `TmTag`, it is validated that the label and the type of the value match the variant definition.
  - In `TmCase`, it is verified that each branch is correctly labeled and produces the same output type.
- **Substitution and free variables:** Functions were extended to operate on labels and patterns in variants.
- **Evaluation:**
  - Rules were added to evaluate labels in variants.
  - Multiple pattern evaluation was implemented, selecting and evaluating the corresponding branch according to the label.

3.6.3. Usage Examples

```
(* Definición de una variante *)
Int = <pos:Nat, zero:Bool, neg:Nat>;;
```

```
(* Creación de valores *)
p3 = <pos=3> as Int;;
z0 = <zero=true> as Int;;
n5 = <neg=5> as Int;;

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

(*Implementacion de la funcion add y pruebas*)
sum = letrec sum : Nat -> Nat -> Nat =
lambda n : Nat. lambda m : Nat. if iszero n then m
else succ (sum (pred n) m)
in
sum ;;


add =
  letrec add : Int -> Int -> Int =
    lambda i1 : Int. lambda i2 : Int.
      case i1 of <zero=z1> => i2
    | <pos=p1> =>(case i2 of<zero=z2> => i1
    | <pos=p2> => (<pos=sum p1 p2> as Int)
    | <neg=n2> =>(if iszero p1 thenif iszero n2 then <zero=true> as
        Int else <neg=n2> as Int
    else
        if iszero n2 then <pos=p1> as Int
            else add (<pos=pred p1> as Int) (<neg=pred n2> as Int)))
              | <neg=n1> => (case i2 of <zero=z2> => i1
                   | <pos=p2> => add i2 i1
                   | <neg=n2> => (<neg=sum n1 n2> as Int))
  in add
  ;;

add p3 p5;;
add p5 p3 ;;
add p5 z0;;
add z0 p5 ;;
add p3 p5 ;;
add n5 p3;;
add p3 n5;;
```

3.7. Implementation of Lists

Lists were added to the language, representing finite sequences of homogeneous elements, along with basic operations: checking if a list is empty, obtaining its head, its tail, and constructing new lists.

3.7.1. Parser and Lexer

New tokens and rules Tokens and rules were added to handle lists, including:

- **List type:** `TyList`.
- **Operations:** `TmNil` (empty), `TmCons` (construction), `TmIsNil` (empty or not), `TmHead` (head), and `TmTail` (tail).

3.7.2. Modifications in `lambda.mli` and `lambda.ml`

Main changes Lists were integrated into:

- **Type resolution and printing:** Support for `TyList` and its operations.
- **Type checking:**
  - `TmNil` returns `TyList`.
  - `TmCons` ensures that the elements are homogeneous.
  - `TmIsNil`, `TmHead`, and `TmTail` verify and return the appropriate type.
- **Evaluation:** Rules for empty lists, non-empty lists, and associated operations.

3.7.3. Usage Examples

```
nil[Nat];;
cons[Nat] 3 nil[Nat];;
l1 = cons[Nat] 8 (cons[Nat] 5 nil[Nat]);;
isnil[Nat] l1;;
head[Nat] l1;;
tail[Nat] l1;;
```

Basic Operations

Advanced Functions

- **Length of a list** (`length`)**:** Calculates the number of elements.

```
letrec length : List[Nat] -> Nat =
    lambda l: List[Nat].
        if isnil[Nat] l then 0
        else succ (length (tail[Nat] l))
in length l1;;
```

- **List concatenation** (`append`)**:** Joins two lists.

```
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 l1 l2;;
```

- **Mapping** (`map`)**:** Applies a function to each element.

```
f = lambda x: Nat. pred x;;

letrec map: List[Nat] -> (Nat -> Nat) -> List[Nat] =
    lambda lst: List[Nat]. lambda f: (Nat -> Nat).
        if isnil[Nat] lst then nil[Nat]
        else cons[Nat] (f (head[Nat] lst)) (map (tail[Nat] lst) f)
in map l1 f;;
```

3.8. Implementation of Subtyping

This section added support for subtyping in the type system, allowing greater flexibility in handling records and functions. This polymorphism allows more specific types to be used where a more general type is expected, respecting compatibility rules.

3.8.1. Changes Made

Subtyping Function: A function was developed to check if one type is a subtype of another. The conditions considered include:

- Two types are compatible if they are equal.
- A record is a subtype of another if it contains at least the same fields with compatible types.
- A function is a subtype of another if the parameters meet a contravariance relationship (the expected type can accept more general types) and the returns meet covariance (the result can be more specific).

Typing of Function Applications: Type checking in function applications was adapted to incorporate subtyping. Now, a more specific type argument can be used in a function that expects a more general type.

3.8.2. Usage Examples

Example 1: Identity Function for Records A function that takes a record and returns it without modifications:

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

Example 2: Field Projection A function that receives a record with at least one field `x : Nat` and returns its value:

```
(lambda r : {x : Nat }. r. x ) {x = 0, y = 1};
```