# Lambda – Calculus Interpreter

Authors: Sergio Puertas Pérez, Lucas Grandal Lama

## 1. Improvements in the introduction and writing of lambda expressions.

### 1.1 Multi-line character recognition:

In main.ml we added the recursive function aux that matches the last index, it returns Some index if it finds ';' and if it doesn't it returns None. In some index we check if the line ends with ';' which would mean our line has ';;' at the end of the file so the prompt would be finished. If this is true then it concatenates all the accumulated lines in ls into a string divided by spaces. If it is not the end of prompt then we call prompt so that the user knows he has to keep writing and adds a line to ls and calls the recursion function. In None we call prompt for the same purpose as before and recurse with the line appended in ls.

EXAMPLES OF USE:

```
input = "Long live

  rock'n roll

   - Rainbow";;
```

### 1.2 Pretty-printer

In lambda.ml we added string_of_ty_prec in the type management section this receives a ty and prec and adds parenthesis if the prec is greater than zero. Then we also added indentation by adding an auxiliary function in the term management section that add white spaces where they are needed.

## 2. Extensions of the Lambda Calculus Language.

### 2.1 Internal Fixed-Point Combinator

We added the new term fix in lambda.ml, in typeof we check the type of t1 given the context and verifies the type of t1 to make sure it is valid for the fixed-point operator. We expect the type of t1 is of the type TyArr where we check that the entry type is the same as the out type and raise an error if they are not the same. If it is not a type arrow then

we raise an error indicating a type arrow was expected. In eval1 we add the cases of tmFix that are tmFix that recursively evaluates the expression until it is completely evaluated where we call the other case which basically applies the substitution , we also add the case for tmFix in free_vars and the substitution function to substitute the free variables, calculated in free_vars, in our tmFix. We also add the type of term in lambda.mli.

In lexer.mll as well as parser.mly we add the token letrec. Then in parser.mly we need a parser case which we add to the term section, these expressions must be of the type "letrec name : types = term in term". We use TmLetIn to generate a new expression with the name $2 which is an abstraction evaluated in $8, with type $4 and $6 content.

EXAMPLES OF USE:

letrec sum : Nat -> Nat -> Nat = lambda n : Nat. lambda m : Nat. if iszero n then m else succ (sum (pred n) m) in sum 2 3

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

## 2.3 String Type

In lambda.ml we add string type TyString, the term TmString and also the required function to concatenate two strings TmConcat that takes two terms (two strings to concatenate). In string_of_ty the type of type string is a "String" so we add this. In typeof, the type of the concat operation will be TyString if both terms are strings, if any of them is not a string then we will raise an error, because both elements must be strings. In string_of_term we add two cases one for the string term which will be the string wrapped in quotation marks and another for the concat function which will print each of the strings to be concatenated between parentheses. We add TmString to free_vars, but we think it might not be necessary to add it as the string is substituted by the same string. We add the TmString to isval so our strings are a valid term. In eval1, we add the cases for concat when two expressions are evaluated where we return the concatenation as TmString, when only one is evaluated, and when none of them are evaluated.

In the lexer we add the tokens for concat and string for the latter we add another token STRINGV that takes a valid string (does not contain ", ; or \n) and strips it from the quotation marks. In the parser, we add the tokens defined in the lexer and the appTerm for concat as well as the case for STRINGV in the atomicTerm section, and the case for the string type in the atomicTy section.

EXAMPLES OF USE:

"This is a String";;

string= "Hi, how are you doing";;

"hello" ^ " my friend";;

## 2.4 Tuples

In lambda.ml we add in the list of types the type tuple which is a list of types and in the term list we add the tuple term as a list of terms and the TmProj that is the projection of the elements based on their position. In the string_of_ty, we add the case for the tuple type, with the options for empty tuple, tuple with one element and tuple with more than one element. The typeof TmTuple will be a tyTuple with the types of the elements of the TmTuple and the typeof TmProjection will be the type of the nth element, we will raise an error if the index is not valid(index does not exist). For the string_of_term, in TmTuple it will be the string of the terms in the tuple and for TmProj it will be the string of the nth term. In free_vars the free variables will be a list of the free variables of each term of the tuple and for TmProj the free variables of the nth term. In the subst, we substitute all the terms in our tuple and for the projection only the selected term. We have to add tuple to isval so that a tuple is valid if all its terms are valid. In the evaluation function we have one case for the tuple and another two for the  projection, in the tuple case, if the tuple is empty we have no rule to apply, if the fist element has been evaluated we evaluate the rest of the tuple, if the element is not a value, we evaluate the term and contuniue with the rest of the tuple unchanged. The cases for TmProj are the one for which the element has been evaluated and the one for when it has not been evaluated where it evaluates it and calls the first case with the result of the evaluation.

As for the parser and lexer, we add the tokens necessary for the projection: the comma, the left bracket, and the right bracket. We add the atomic term for the tuples term, add the projTerm, that can be either the projection term and the integer or the atomicTerm. We also need to add the tuplesTY which is the list of the types of the elements in the tuple.

EXAMPLES OS USE:

tuple= {“nano”, 33, true};;

tuple.2;;

## 2.5 Records

In lambda.ml, we added tyRecord, a list of string and type as wel as its corresponding term TmRecord defined in the same way but in the records section. In string_of_ty we added the case for TyRecord that recurses trough the whole tuple of records and for each record it separates it from another using commas and gets the string and type separated by a colon, all these records are then wrapped in brackets. In the typeof section we add the case indicating the type of a TyRecord which is the TyRecord and the type of each tuple inside of it. In string_of_term we apply the same logic as for the type to the term (brackets with the records separated by commas). The free variables of a record are the free variables of each record term inside the tyRecord. In the substitution function we apply the substitution to the terms inside of each of the string : term tuples. As well as for tuples, the term record is valid if all the terms inside are valid. In the evaluation rules, if a record is empty we have no rules to evaluate it. If it is not empty if the first term is evaluated it continues evaluating the rest of terms, if it is not evaluated it evaluates it and continues to evaluate the rest, we also had to modify our projection's typeof so it also accepts the TyRecord in order to be able to also project records.

In the parser and lexer we add the comma, lbracket and rbracket tokens in order to recognize our record syntax. Then in the parser, we add the record term to our atomic term list and define our recordTM and nonEmptyRecordTM as well as define the recordTY and nonemptyrecordTY.

EXAMPLES OF USE:

driver = {nombre = "Fernando", victories = 32, active = true} ;;

driver.victories;;

## 2.7 Variants

In lambda.ml we add the type TyVariant and the terms for the tagging and the case structure. In teh string of type we add the case for variants where we define the recursive function that for the empty doesn't return anything, for the variant of one element it prints the string and its type and for more than one variant it returns the string

and type of the first and calls the recursive function with the remaining elements. In typeof we add the rules for tag and case. In the first one the rule checks if a tag fits into a specific variant type. If it does it checks if the tag exists in the defined variant. For the case, For the case, the rule first checks that the type of the term `t` is a `variant type`, which means the term is expected to match one of the variant tags defined in the variant type. It then compares the tags defined in the variant type with the tags used in the cases to ensure they align. If the tags match, it proceeds to type check each case, ensuring that the terms in each case have the same type. If all case branches have the same type, the rule is valid. Otherwise, if the case branches have different types or the tags don't match we raise an error. In free_vars the free variables for the tag are the free variables of the term and for the case they are the free variables of the term and the and the set of variables in the labels of the cases. List.fold_left iterates over the cases, and for each case it calculates the free variables of the term excluding the identifier. In subst, for the tag it substitutes the term and for case it applies substitution to each term in the cases, if the case identifier matches the variable , it leaves the case unchanged otherwise it applies substitution on the associated term. In the evaluation rules, for the tag we evaluate the term of the tag and for case we have two rules, if we have a case with term and type we evaluate the term, if we have a tag and a list of cases if the term of the tag is valid, already evaluated, we check the corresponding case with the tag and substitute the value of the term with the identifier.

In the lexer and parser we add the tokens “<”, “>” CASE, AS, OF and in the parser in the term section we add the options for building the tag and the cases, as well as the rule for the variant type in atomicTy, we have also defined the structures for cases and case separately in order for our program to be more understandable and manage a non fixed number of cases.

EXAMPLE OF USE:

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

p3 = <pos=3> as Int;;

z0 = <zero=true> as Int;;

n5 = <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 p3;;

abs z0;;

abs n5;;
```

## 2.8 Lists

In lambda.ml we add the type list, and in the term list we add the term list as well as all its associated functions: nil, cons, isnil, head and tail. We define in string_of_ty the TyList which will display List and the type of the elements inside the list between square brackets. In the typeof section we have the type of TmNil which will be a TyList of a certain type, Tm Cons, that will raise an error if either one of the lists to append don't have the type specified in cons; TmIsNil, that will raise an error if the term is not a list of type ty, if there is no error it returns a boolean; TmHead and TmTail the same thing occurs as with TmIsNil, if the term is not a list of type ty raise an error and if there is no problem TmHead results in ty and TmTail in a TyList ty. In string of term we define the string for all the terms, which will be the name of the term between square brackets the list type and between parenthesis the different terms the different functions work with. The free variables are the free variables of the term for tail,head and isnil, the union of the free_vars in the terms in cons and none in the empty list nil. Same thing for the substitution, in head, tail and isnil we apply the substitution to the term, in cons we apply substitution to both of the terms and for nil we do not need to apply any substitution. In evaluation, we have two cases for cons, the case in which the first element is not evaluated so we call TmCons with the first element evaluated, and the case where the first element is already evaluated where we get TmCons with both elements evaluated. For IsNil we have three cases: IsNil of a Nil term is an empty list so this means that the statement is true, if the term is TmCons, then the list is not empty, false; Then for terms that are not fully reduced we evaluate the subterm recursively. For

head and tail  we have two cases each, the first when applied over a cons where in head we return the first term and in the tail the second term of cons in the other case we recurse for both evaluating the subterm. Lastly in isval, we add the cases for TmCons that is valid when both terms are valid and for TmNil when it is empty it is true.

For the lexer and the parser we add the tokens “[”, “]”, HEAD, TAIL, CONS, NIL, ISNIL. In the parser we add the atomic terms for each of these operations following their defined structure.

EXAMPLES OF USE:

nil[Nat];;

list = cons[Nat] 1 (cons[Nat] 2 (cons[Nat] 3(cons[Nat] 4)));;

head[Nat] list;;

tail[Nat] list;;

isnil[Nat] list;;

## 2.9 Subtyping

In lambda.ml we added the function subtypeof that checks if one type (t1) is a subtype of another type (t2). For function types (TyArr), it ensures contravariance in arguments (x1 is a subtype of y1) and covariance in results (y2 is a subtype of x2). For record types (TyRecord), it verifies that every field in r1 exists in r2 with a compatible type. If the types are neither functions nor records, the function checks if t1 and t2 are identical. The function is recursive, processing nested types and fields as needed, enabling a comprehensive structural subtyping check.

EXAMPLES OF USE:

Any function using an abstraaction or a fix where subtypeof is implemented can serve as a check to see subtyping works.

letrec sum : Nat -> Nat -> Nat = lambda n : Nat. lambda m : Nat. if iszero n then m else succ (sum (pred n) m) in sum 2 3;;

## 3.Required functions to test functioning.

### 3.1 FIXED POINT COMBINER

PRODUCT:

```
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 n then
            0
        else
            sum (prod (pred n) m) m
in prod 2 3;;
```

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 n then
            0
        else
            sum (prod (pred n) m) m in
```

```
        letrec fac: Nat -> Nat =
        lambda n : Nat.
            if iszero n then
                1
            else
                prod n (fac (pred n))
in fac 3;;
```

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;;
```

## 3.2 LISTS

LEN:

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

MAP:

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

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 list1 list2;;
```

## 3.3 VARIANTS

```
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 then
                  if 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
;;
```