DLP PRACTICE MANUAL 2021-2022

The purpose of this memory is to recopilate all the changes and features applied to the originally given code of evaluation of lambda-calculus. The scheme of this memory will be represented as it follows:

-For each change applied, we will represent it as a point, in which we will talk about:

a) The objective of the modification/agregation of the code

b) The modifications applied, as well as a brief explanantion about why it was made that way

c) Specific examples about the functioning of the code after the implementation

Point 1): Improvements in writting and lecturing of Lambda-calculus: Recognition of multi-line expressions

a) Our main goal is the modification of the lecturing method of our code, in a way it will be able to read not just the firts line of what we introduce, but to recognize all the arguments given until we reach a certain point, delimited by an special character (we will use two ";;"). This will help us to not to be restricted to a one-line writting, but to be able to scatter the lines in different rows, facilitating its disposition and lecture.

b) For this purpose, we will modify our main.ml file, in such a way it will only stop reading when it detects the special set of characters ";;". This is obtained through the impementation of a new function called "read", that works as it follows:

```
let rec read ()=
   let rec auxread = function
           h::(""::_)-> h
           | h::[]-> h ^ " " ^ read()
           | h::t->h ^ (auxread t)
           | [] -> read()
   in
   auxread (String.split_on_char ';' (read_line () ) )
```

As we can see, it is applied to what we got in the split_on_char function to the readed introducted line, in such a way that it will proccess the command until it detects a void string (that will mark the split of the char, with tell us that there were a ;; there and we must stop) or it doesn have anything more to process. In this second case, the auxred function will sumon the read function again with the next reading line, appending what we got in our first read. This makes that the function only stopt when either it gets the end of the introduced command, or a ";;" characters

c) Now, whenever we introduce a command with a ";;" in the middle, it will only procees it to that point. For example:

```
>> true;; this is clearly a syntax error

   true : Bool


>> if true
       then 1
   else
       0;;

   1 : Nat


>> if false then 1;; else 0

   syntax error
```

Point 2) Arrangement of an internal fix point combinator, in order to be able to write direct recursive functions

a) In this point, we will try to define a new combinator to define recursive functions in a easier way. Initially, we will use the key word "letrec", followed by the type of the function we will built, and its implementation. It should end up like the example above:

```
letrec sum : Nat -> Nat -> Nat =
lambda n : Nat. lambda m : Nat. if iszero n then m else succ (sum (pred n) m)
in sum 21 34
```

b) For this purpose, the following files have been updated:

-lexer.mll: We included "letrec" as the keyword in order to apply recursivity

-parser.mly: We added lines refering to the application of recursive functions in the term application appart.

-lambda.ml: We had to include a new kind of term based on the application of the fix function (our applied-method for recursivity) from lambda-calculus. These changes are:

-"TmFix" has been added as a new kind of term

-The case "TmFix" has been added to the "typeof" function in order for the program to know how to proceed with this kind of terms. Same with "string_of_term", "free_vars", and "subst" functions

-lambda.mli: "TmFix" has been added to the list of existing terms.

c) Now, with just applyng the writting told in sub-appart a), we are able to express recursivity in a way easier form. For example, the result of applying the function talked about before:

```
letrec sum : Nat -> Nat -> Nat =
        lambda n : Nat. lambda m : Nat. if iszero n then m else succ (sum (pred n) m)
in sum 21 3
```

Point 3): Include of a variable CONTEXT

a) In this part we are looking for include a variable context that allow us to save values in vars and use them in next operations. For this we are using the syntax x = expresion, where x is the name of the var and the solution of the expresion is the value that we are going to save in it. Also, we must know that we are usin a functional implementation of var context because as in Lambda calculus we use short expresions that work similar to mathematical functions using a imperative type will only make our encoding more difficult.

b) In order to achive this we will subdivide what we did in 3 steps, one fore each file we changed:

- The parser will not more return a type Term, it will now return a new type called variable with two posible creators: varValue of term: wich just requests for the evaluation for that Term. varAsignation of string * term: wich requests to save the value of the avaluation of the term in a var with the string as name.

- In main.ml we will separate the flow of the function in two, one for each new types. In the case of varValue is pretty similar to the original but for the varAsignation we will have to add the type of var that we want to include in the context and also have a list with their values after evaluation.

- For all of this have sense we need to finally replace the vars in the expresions, and for that we added a new argument for the function eval wich is the list of vars and changed it as it follows:

```
let rec use_vars tm = function
        (x,s)::t-> use_vars (subst x s tm) t
        | []-> tm;;


let rec eval_loop tm = try
        let tm' = eval1 tm in
        eval_loop tm' with
                NoRuleApplies -> tm;;

let eval tm vars=
        let tm'=eval_loop tm in
        eval_loop (use_vars tm' vars);;
```

This new code will just make a first atempt to evaluate the expresion directly, search and replace vars after that and last but not less, a final evaluation.

c) Now we can save values as easy as this:

```
>> x=0;;

   x = 0 : Nat

>> f = Lx:Nat.(succ x);;

   f = (lambda x:Nat. succ (x)) : (Nat) -> (Nat)

>> f x;;

   1 : Nat
```

Point 4) Inclusion of a string type, plus implementation of a concat functions

a) For this point, we will try to add a string type to our program, in order to give a basic expresions support to the user of the evalutor. Besides, we will also create a concatenation function in order to append two string values, which can be really useful in many situations.

b) The implementation of this objective has required changes in several files:

-lexer.mll: A STRING, STR and CONCAT tokens have been added.

-parser.mly: The token CONCAT has been added in appTerm, the STRING token has been added into atomicTerm and STR token in atomicTy.

-lamba.ml: Inclusion of string in the list of terms and types. Implementation of concat case in "typeof", "string_of_term", "eval1" and "subs" function.

c) The inclusion of the string type allow us to operate with this new type in the next form:

```
>> "a";;

     "a" : Str

>> x="abc";;

     x = "abc" : Str

>> concat "a" "b";;

    "ab" : Str

>> f=Lx:Str.concat x "fin";;

     f = (lambda x:Str. concat(x) ("fin")) : (Str) -> (Str)

>> f "del";;

    "delfin" : Str
```

Point 5) Adding pair type

a) We will try to add a pair type in order to represent tuples of values in easy form, as well as implement "first" and "second" operation to obtain the value of a certain element of a tuple in a direct way. This elemnts will have the forma: (a,b). Where "a" is the first element of the pair, and "b" the second, both separated through a coma and gathered between parenthesis

b) The arrangement of code made to obtain this new type has been at it is followed:

-lexer.mll: A COMA token has been added

-parser.mly: The token FST and SCN has been added in appTerm, pair form has been added in atomicTerm

-lamba.ml: Inclusion of TyPair and TermPair, as well as their use in evaluation, typing, printing, and substitution functions (typeof, string_of_ty, string_of_term)

c) Examples of pair evaluation comes as it follows:

```
>> p=(0,1);;

     p = ((0),(1)) : ((Nat),(Nat))


>> p2=("Hola",true);;

    p2 = (("Hola"),(true)) : ((Str),(Bool))

>> first p;;

     0 : Nat

>> second p2;;

    true : Bool
```

Point 6) Adding list typeç

a) The objective of this point is the inclusion of a list type, in a way that we would be able to create a ordered set of elements of the same type, simulating the ocaml lists they could be written inside "[]" ean separated with "," or conect a head and it's tail with "::". Also we will need to add a ":" followed by the type that the elements are going to have for avoiding problems with the empty list. Besides, we will also add utilities functions, such as head (get the first element of a list), tail (get all the list except the first element) andisEmpty (check if the given list is empty or not). Finally, we will create two example functions in order to probe the correct functioning of this whole point.A length function, which will return the number of elements that are inside a given list, and a map function, which will apply a certain given function to all the elements of a given list

b) In order to reach this goal, the next modifications have been applied:

-lexer.mll: "LIST", "RCORCH", "LCORCH", "HEAD", "TAIL" and "ISEMPTY" tokens have been added in order to represent the proper list type plus the utilities functions

-parser.mly: TmHead, TmTail and TmIsEmpty have been added to the appTerm collection, as well as LCORH and RCORH to the atomicTerm collection. At last, we have also added the type LIST to the ty set

-lambda.ml: TmList, TmHead, TmTail and TmIsEmpty added to term type set. TyList, TmList, TmHead, TmTail and TmIsEmpty, added to the different management evaluation (string_of_term, free_vars) functions

-lambda.mli: Inclusion of TmList, TmHead, TmTail and TmIsEmpty in type term. List and EmptyList types added in type ty.

c) Examples of list evaluation are shown below:

```
>> x=[1,2,3]:Nat;;
    x = [1,2,3]:Nat : (Nat list)
>> s="a"::["b"]:Str;;
    s = ["a","b"]:Str : (Str list)
>> tail x;;
    [2,3]:Nat : (Nat list)
>> head x;;
    1 : Nat
>> (head x)::(tail x);;
    [1,2,3]:Nat : (Nat list)
>> isempty x;;
    false : Bool
>> isempty []:Bool;;
    true : Bool
```

Point 7) Adding records

a) Our objective in this point is the inclusion of record fields. We will try to introduce records themselves, as well as the project function, which will allow us to get a certain value of one of the elements of the selected record

b) To reach this objective, the next features have been applied:

-lexer.mll: "RBRAC", "LBRAC", and "PROJECT" tokens have been added in order to represent the proper record type plus the utilities functions

-parser.mly: TmProject have been added to the appTerm collection, as well as LBRAC and RBRAC notation to the atomicTerm and ty collections.

-lambda.ml: TmRec and TmProject added to term type set. TyRec added to the different management evaluation (string_of_term, free_vars)functions

-lambda.mli: Inclusion of TmRec and TmProject in type term. TyRec type added in type ty.

c) Some examples of records use:

```
>> reg={a=1,b="string",c=true,f=Lx:Nat.succ x};;

    reg = {a=1,b="string",c=true,f=(lambda x:Nat. succ (x))} : {a:Nat,b:Str,c:Bool,f:
    (Nat) -> (Nat)}

>> project a reg;;

    1 : Nat

>> project f reg;;

     (lambda x:Nat. succ (x)) : (Nat) -> (Nat)

>> (project f reg) 3;;

    4 : Nat
```

Point 8) Adding sub-TYPING

a) The goal in this apart is the inclusion of subtypes in such away we will be able to allow polymorphism in records and functions in order to make comppatible some set of values that they weren't before

b) To achieve this objective, the following changes have been applied to the code:

-lambda.ml: We created a new pair of functions:

```
let rec rec_subtype l = function
        (name,ty)::t ->  (try
        let ty' = List.assoc name l in
        if (ty=ty') then rec_subtype l t
        else false
        with
        Not_found -> false)
        |[]-> true


let subtype t1 t2 = match t1,t2 with
        (TyRec l1, TyRec l2) -> rec_subtype l1 l2
        |_-> t1=t2
```

Which allow us to analyze the type of the given parameters, in order to see if they match either for a direct equivalent (they bouth have the same type), either because of
one of them include the other.

c) Here we have some examples of subtyping:

```
>> f= L x:{a:Nat}.x;;

    f = (lambda x:{a:Nat}. x) : ({a:Nat}) -> ({a:Nat})

>> f {a=1,b="b"};;

   {a=1,b="b"} : {a:Nat}
```

Alejandro Mayorga Redondo
Brais Garcia Brenlla