LAMBDA CALCULUS INTERPRETER

Authors:

Daniel Ferreiro Villamor - d.ferreiro@udc.es

Eva Maria Arce Ale - eva.arce@udc.es

Introduction

This will be a user guide where will explain in detail how we implemented the lambda interpreter based in Ocaml language, how the code works and the different modules where we implemented the different functionalities, as well as show some self-explanatory examples with code ready to execute and try out.

In the following sections we will be showing all the code improvements made in order to implement the before-mentioned functionalities.

Functionalities

1. Easier writing and introduction to lambda expressions.

    1.2 Multilinear expression recognition.

    In order to implement this section the changes made were on the main.ml document, where we implemented the function **read_command** which processes the lines while checking for the last 2 characters of a line being semicolons, ie: *succ 3;;* , and if so, then ending the acceptance of lines and evaluating.

```
if iszero 3
  then 0
else 1;;
```

2. Extensions on the lambda-calculus language.

    2.1 Addition of a **Fixed-point combinator**, making possible the definition of recursive functions.

    Firstly we implemented the recognition of a new token by the lexer.mll file, this being **LETREC**, posteriorly adding those tokens onto the parser.mly file and adding in the **term :** section support for the let rec term (**TmFix**) and finally adding functionality for this new term in the lambda files, this meaning making the corresponding addition to the **string_of_term, free_vars** & **subst** functions of the TmFix term, which is a pretty straight-forward one.

    For the **eval1** function, where we evaluate the term until a value is found, or in case an abstraction is found, then this one is substituted.

    For the **typeof** function, we need to check that the type of the term is the expected.

    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 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 7;;
```

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

2.2 Implementation of a global definition context that allows binding free-variable names to a value or term, so that it permits the future usage of them.

To implement this functionality a lot of changes were made, we will be explaining them file by file: Starting with the parser file in which in the **s :** section we change it to

- Eval when it is a term followed by the **EOF** and to Bind when **IDV EQ** term **EOF**
- EvalTy to display the content of the identifier and BindTy to associate an identifier with a type. EvalTy it is a string followed by the **EOF** and to BindTy when **IDT EQ** ty **EOF**

In the lambda files is where the majority of the changes where made:

- We added a **contextv** type of (string * term) list and the supporting functions for it **getdef**, **adddef** and **emptydef** .
- Changes were made to the **eval** and **eval1** function to take the context (ctxv) as parameter whenever it is called.
- Added a **give_ctx** function which takes the context and a term and substitutes the name associated with a value for the value itself when evaluating terms.
- Finally we implemented an **execute** function which decides if Eval or Bind or EvalTy or BindTy is needed and also now handles part of the printing to be done.

At last in the main file we changed the loop, now both contexts are passed to the loop, same for the empty contexts. Also now in the try part we now loop the **execute** function mentioned before.

```
x = true;;
id = lambda x : Bool. x;;
id x;;
```

```
fib = 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;;
fib 7;;
```

The type identifier will be useful in the variants (section2.7)

```
N = Nat;;
lambda x : N. x;;
```

2.3 Addition of the String type in order to support character chains as well as a concatenation function.

As with almost every new type to be implemented the first part is to add tokenization for that character, in this case: **STRINGV, CONCAT** & **STRING,** last one for the type, posteriorly in the parser adding the corresponding terms and type, this being **TmConcat** in the **appTerm :** section, **TmString** in the **atomicTerm :** section and **TyString** in of course the **atomicTy :** section.

Finally in the lambda files we added all the support for these new terms and type in the following functions:

string_of_term, free_vars, subst, string_of_ty, isval, typeof & **eval1**

With the only one being relevant enough to mention being the implementation of concat in the eval1 function which receives 2 terms as parameter, evaluating both of them to be Strings and concatenating them with the ^ operator.

```
"";;
"abc";;
concat "para" "sol";;
concat (concat "para" "sol") "es";;
lambda s : String. s;;
(lambda s : String. s) "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;;
```

Additionally, we have implemented another function to retrieve the first character of the string and another one to obtain a copy of the string without its first character (in case the string has only one character or is empty, it would return an empty string)

Addition of tokens STRHEAD and STRTAIL, following that, adding in the parser the corresponding terms this being **TmStrHead** and **TmStrTail** in the **appTerm :** section

```
s = "dlp";;
strhead s;;
strtail s;;
```

2.4 Addition of tuple type and projection based operations.

We decided on tuples over pairs and in order to implement them we followed the same scheme as with any other type: first we added on the lexxer and parser the **LKEY, COMMA** and **RKEY** tokens then we put into Ocaml Lists each element of the tuple making while on the parser file, and finally we get the **TmTuple** and **TyTuple** onto the lambda files.

We update the lambda.mli file and add to the lamda.ml all necessary functions for this new term and type (**string_of_term, free_vars, subst, ...**)

Finally we added a function to access an element of the tuple by position so we added a new term **TmGet** by making use of the **DOT** token, with this in mind, we changed the parser file adding a new access term which will in the future (when implementing records, accept a TmGet with INTV (position for tuple) as well as a IDV (label for records)). Moreover we then make use of the nth function of Ocaml lists to access said element.

```
{1,2};;
t = {{true, 2}, {letrec next : Nat -> Nat = lambda n : Nat. succ n in next,
"abc"}};;
t.1;;
t.1.2;;
t.2.1 0;;
```

2.5 Incorporation of Record type as finite sequences of labeled fields of any type.

Records were the next step after doing tuples and we could make use of a lot of the code built for the tuples whilst having major differences, these being:

For tuples we chopped the element sequence into an Ocaml List, now we do it in a similar way but as an Ocaml tuple List, said tuples would being a pair of field and value

With all this in mind, we again use the **LKEY, COMMA** and **RKEY** token as in tuples adding **EQ** and **COLON** now to the mix for the field > value, field > type relations

We again implement all the necessary functions for the Records (**string_of_term, free_vars, subst, ...**) and finally for the projection function for records we use the before implemented **TmGet** we update the implementation to match the type of the first element given to the get with either tuple or record and in the case of records, we use the Ocaml assoc function in order to get the element corresponding to the given label.

```
{cost=30,partno=5524};;
r = {texto="equis", num={x=1,y=2}, flag=true};;
r.texto;;
r.num;;
r.num.x;;
```

2.7 Incorporation of variant type

For the variant implementations we also could make use of a lot of the code as variants also are some kind of fielded sequence we could use the functions implemented on the parser for the records.

So we created new tokens in the lexer **LVAR** and **RVAR** to signal the beginning and end of the variant term and then, as with records, the **COMMA, EQ** and **COLON** tokens then its the same thing we've done before for the records on all the files remaining adding implementation for all the functions (**string_of_term, free_vars, subst, ...**).

But we're not finished yet since we also have to implement the ascription function and the case of function:

For the **Ascription** function we added the token **AS** and a new **TmAscr.** For this new term constructed as: "term AS ty" we had to match the term to TyVariant first because in the case of it being a variant is when all the big work is done (the tmAscription works for every ty), this being matching the terms type with the type of each of the fields in the ty (ty being a TyVariant of more than 1 field) and if it matches then returning the type of the ty.

Finally the **Case of** function we added the tokens **CASE, OF** and **PIPE** (w.i.p)

```
Int = <pos:Nat, zero:Bool, neg:Nat>;;
p3 = <pos=3> as Int;;
z0 = <zero=true> as Int;;
n5 = <neg=5> as Int;;
1 as Nat;;
```

2.8 Incorporation of lists as finite sequences of elements of the same type, as well as the typical lists operations head, tail, and empty

For the implementation of the lists, we added 8 new tokens in the form of:

- **NIL** for the empty list, returns a new **TmNil**
- **CONS** as a constructor for list, returns a new **TmCons**
- **ISNIL** to call the empty? function, returns a new **TmIsNil**
- **HEAD** to call the head function, returns a new **TmHead**
- **TAIL** to call the tail function, returns a new **TmTail**
- **LIST** for the **TyList** new type
- **OBRACKET**
- **CBRACKET**

These functions all receive the type that the list stores which is surrounded by the OBRACKET and CBRAKET followed by another term, or in the case of the TmCons 2 terms, also TmNil only receives the type.

So after all this was done we added all the support for the given functions in the form of (**string_of_term, free_vars, subst, ...**). the only part worth mentioning, as the rest even though being a lot, were really simple to implement (pattern matching) was the TmCons whose typeof had the difficulty of having to check the type given as well as the type of the head and then recursively checking the rest of the elements typing too (since list have to be homogeneous, all the elements must be of the same typing).

Whilst trying to use the append function for the lists, we also realized that we also needed to check in the TmCons typeof for the tail being of TyList instead of TmCons, in which case we return the type of the list.

```
l1 = cons[Nat] 1 (cons[Nat] 2 (nil[Nat]));;
l2 = cons[Nat] 3 (cons[Nat] 4 (nil[Nat]));;
l3 = cons[String] "a" (nil[String]);;
cons[Nat] 1 l1;;
cons[String] "a" l1;;

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

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;;

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] (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 l2 f;;
```