Lambda Calculus Interpreter

Programming Languages Dessign

2021-2022

Diego Roca Rodriguez

1. User Manual

1.1 Using the Lambda Calculus Interpreter

To use the Lambda Calculus Interpreter we must first compile it with the included makefile using

```
# make all
```

After that to run it we just have to execute the binary file obtained. We can run the Interpreter in File Mode or in Interactive Mode. To use the interpreter in File Mode we just have to pass as argument the path to the file that contains the lambda expressions that the program will parse. To use the interpreter in Interactive mode we just call the binary from the command line.

Whenever we are running the interpreter in File Mode and it finds a error during the parsing the program will input a exception and the line number where the error was found.

File Mode:

```
# ./top file_path
```

Interactive Mode:

```
#./top
```

The Interpreter will only parse a expression that ends with a double semicolon (;;), allowing for multiline expressions

1.1 Types in Lambda Calculus Interpreter

The following types and type-related functions are implemented and perfectly usable in our Lambda Calculus Interpreter. Details about the implementation will be explained in the Technical Manual section.

Natural numbers: Simply the numbers belonging to the set of the Naturals, this means, numbers ranging from Zero to Positive Infinite. Are defined simply by inputing the desired number.

The operations implemented with Natural Numbers are getting the predecessor, getting the successor and checking if the number is zero.

```
>> 1;;
 - : Nat = 1
>> succ 1;;
 - : Nat = 2
>> pred 1;;
```

```
 - : Nat = 0
>> iszero 1;;
 - : Bool = false
>> iszero 0;;
 - : Bool = true
>> succ (pred 1);;
 - : Nat = 1
```

Booleans: Truth value types, used mostly in conditional sentences. Are defined by the keywords true or false (capitalization is relevant).

```
>> true;;
 - : Bool = true
>> false;;
 - : Bool = false
>> if true then 1 else 0;;
 - : Nat = 1
>> if false then 1 else 0;;
 - : Nat = 0
```

Strings: Arrays of characters used to represent words or sentences. Are defined by inputed any array of characters between quotation marks.

The operation implemented with Strings is the concatenation.

```
>> "hello";;
 - : String = hello
>> "world";;
 - : String = world
>> "hello"@"world";;
- : String = helloworld
```

Pairs: Tuples of two elements of any type. Are defined by inputing the elements that form the tuple between parenthesis and separated by a comma.

The operations implemented with Pairs are the ones that deal with getting the first or second element of the pair.

```
>> (1,2);;
 - : (Nat,Nat) = (1,2)
>> ("hello","world");;
 - : (String,String) = (hello,world)
>> ("hello",("brave",("new","world")));;
 - : (String,(String,(String,String))) = (hello,
(brave,(new,world)))
>> ("hello","world").1;;
 - : String = hello
>> ("hello","world").2;;
 - : String = world
>> ("hello",("brave",("new","world"))).2;;
```

```
 - : (String,(String,String)) = (brave,
(new,world))
>> ("hello",("brave",("new","world"))).2.2;;
- : (String,String) = (new,world)
>> (L x:Nat.succ x, 1);;
 - : Pair((Nat) -> (Nat),Nat) = ((lambda x:Nat.
succ (x)),1)
```

Records: Set of named elements of any type. Are defined by inputing the elements that form the set between braces and separated by a comma. A element of the set will be formed by the element name and the element itself separated by a colon.

```
>> {number:1};;
 - : {number:Nat} = {number:1}
>> {number:1,string:"hello world!"};;
 - : {number:Nat, string:String} = {number:1,
string:hello world!}
>> {number:1,string:"hello world!"}.number;;
 - : Nat = 1
>> {number:1,string:"hello world!"}.string;;
 - : String = hello world!
>> {lambda_function:L x:Nat. succ x,number:1};;
 - : Record{lambda_function:(Nat) -> (Nat),
number:Nat} = {lambda_function:(lambda x:Nat.
succ (x)), number:1}
```

The operation implemented with record is the projection, this means, getting a element of the record using its name.

Lists: Array of n elements of the same type. Are defined by inputing the elements of the list between brackets and separated by commas. A list can be empty if no element is given, thus being defined by only two brackets.

The operations implemented with lists are head and tail, in order to get first and last element and also a operation to check if a given list is an empty list;

```
>> [1,2,3];;
list List[1,2,3]
 - : Nat list  = List[1,2,3]
>> ["a","b","c"];;
list List[a,b,c]
 - : String list  = List[a,b,c]
>> [{id:"1"},{id:"2"},{id:"3"}];;
list List[Record{id:1},Record{id:2},Record{id:3}]
 - : Record{id:String} list  =
List[Record{id:1},Record{id:2},Record{id:3}]
>> hd [1,2,3];;
list List[1,2,3]
 - : Nat = 1
>> tl [1,2,3];;
list List[1,2,3]
```

```
 - : Nat = 3
>> []
;;
 - : Empty List = Empty List
>> isempty [];;
 - : Bool = true
>> isempty [1,2,3];;
- : Bool = false
```

1.3 Functions in Lambda Calculus Interpreter

The functions in Lambda Calculus Interpreter are declared as Lambda Calculus Abstractions. To do so we must use the lambda (or 'l' 'L') keyword.

The structure of a function declaration will be:

L <v_name>:<v_type>. <body>

In order to do a multi-variable function we need to use a technique known as currying, that allows for multiple variables to be binded in the same function body.

```
>> L x:Nat. succ x;;
 - : (Nat) -> (Nat) = (lambda x:Nat. succ (x))
>> L x:Str. x@"hola";;
- : (String) -> (String) = (lambda x:String.
Xhola)
>>  L x:(Nat,Nat). x;;
 - : (Pair(Nat,Nat)) -> (Pair(Nat,Nat)) = (lambda
x:Pair(Nat,Nat). x)
>> L x:{id:Nat}.x;;
 - : (Record{id:Nat}) -> (Record{id:Nat}) =
(lambda x:Record{id:Nat}. x)
```

1.3 Variable scope and Let In in Lambda Calculus Interpreter

A alternative to currying in declaring functions with several variables is the use of "Let In". This allows us to declare a name for a term inside the scope of the current operation.

The structure of a let in declaration is the following:

let <term_name>=<term_body> in <term>

```
>> let next=(lambda x:Nat. succ (x)) in next 2;;
 - : Nat = 3
>> let first=(lambda x:(Nat,Nat). x.1) in first
(1,2);;
- : Nat = 1
```

1.3 Conditional Sentences in Lambda Calculus Interpreter

A powerful tool in programming languages is the use of flow control sentences, like the conditional sentences. A conditional sentence is formed by a conditional guard, a “then” block and a “else” block.

In Lamda Calculus Interpreter we have flow control sentences in the form of a if-then-else term. The conditional must be of type bool (or derive into a type bool) and the ‘then’ and ‘else’ block must be of the same type

```
>> if (true) then 1 else 0;;
 - : Nat = 1
>> if (false) then "hello" else "world";;
 - : String = world
>> if (iszero 0) then 2 else 3;;
 - : Nat = 2
>> if (isempty []) then 2 else 3;;
- : Nat = 2
```

1.3 Recursivity in Lambda Calculus Interpreter

Sometimes we would want to declare a function that could call itself in order to do some task or simply to improve computation efficiency. This is known as revursivity.

Lambda Calculus uses a combinator called ‘fix’ combinator to achieve this goal, however is really tedious to declare this term every time we want to do a recursion.

In order to declare recursive functions in Lambda Calculus Interpreter we just input the keyword ‘letrec’ before the declaration of the function to apply recursion to it.

```
>> 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;;
- : Nat = 55
```

1.2 Variable declaration in Lambda Calculus Interpreter

The Lamda Calculus Interpreter is able to deal with a context of named variables, allowing us to declare them and then use them elsewhere. Any term of the language can be a variable, ranging from functions to any of the previously mentioned types.

To declare a variable we input: var_name = element.

To use the element contained in the variable we just put the var_name wherever we want to use that element.

```
>> a=2;;
 - : Nat = 2
>> b=3;;
 - : Nat = 3
```

```
>> r={a:"str"};;
 - : Record{a:String} = Record{a:str}
>> r.a;;
 - : String = str
>> succ a;;
 - : Nat = 3
>> pred b;;
 - : Nat = 2
>> sum = L x:Nat. L y:Nat. 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) -> ((Nat) -> (Nat)) = (lambda x:Nat.
(lambda y:Nat. let sum = (fix (lambda sum:(Nat) -
> ((Nat) -> (Nat)). (lambda n:Nat. (lambda m:Nat.
Conditional stament: if (iszero (Variable: n))
then (Variable: m) else (succ (((Variable: sum
pred (Variable: n)) Variable: m))))))) in
((Variable: sum Variable: x) Variable: y)))
```

2. Technical Manual

2.1 Code structure

Lambda Calculus Interpreter code is divided in several files

→ main.ml: contains the main loop of the interpreter and the file parser.
→ lexer.ml: deals with the lexical analysis, binding tokens to the words that are inputed and sends the token to the parser.
→ parser.mly: deals with the syntactic analysis, stablishes what combination of tokens are valid and sends the data inputed to the interpreter.
→ lambda.ml: core part of the interpreter, deals with the evaluation of terms and types in lambda calculus
→ lambda.mli: interface for lambda.ml

2.2 Main Execution Loop and File Parsing Implementation

The main loop is contained in the file main.ml. It consits of two parts, reading the user input and send it to the Interpreter. It also deals with discerning between Interactive Mode or File Mode.

In Interactive mode the program will read from the stdin until a separator, in this case “;;” is found. This allows us to input multi-line expressions and make the code we write a little bit more readable.

To develop the file parser a function to parse a file and split the sentences included in it was needed. To do this, we parse a file line by line and mark a lambda expression as complete whenever we find the separator, in this case, “;;”. To acomplish this the use of regular expressions and the function str.split was needed.

Once the lambda expressions are taken from the file they are parsed in the same way that the interactive interpreter, but with the difference that in the loop the program keep track of the line number, so whenever an error is found in the file the user can be informed in what line it was found.

2.3 Lambda Calculus Extensions

The main extensions implemented to the Lambda Calculus Interpreter are the following ones

Fixed Point Combinator: In Lambda Calculus is needed to use a fix point combinator in order to implement recursion. In order to avoid writing all the time the fixed point combinator it was implemented a recursive term.

The recursion is inputed into the interpreter whenever the parser finds a expression of the following shape:

```
LETREC STRINGV COLON ty EQ term IN term
```

where stringv is defined as any combination of at least 1 character or digit.

To develop the fixed combinator it was needed to add a TmFix term that deals with the inclusion of the fixed combinator whenever it is evaluated.

Strings: The strings are inputed into the interpreter whenever the parser finds a expression of the following shape:

```
termString:
“ STRING “

termConcat:
<term> @ <term>
```

where string is defined in the lexer as any combination of at least 1 characters, digits or spaces

To develop the string term and its related operations it was needed to include the following types:

```
tyString
```

and the following terms:

```
tmString of string
tmConcat of tm*tm
```

For the evaluation of strings and its related operations it was done this way

```
tmString(<string>) → tmString(<string>)
tmConcat tm1 tm2 → if (tm1 is tmString(s1) and
tm2 is tmString(s2)) then tmString(s1^s2)
```

Pairs: The pairs and their operations are inputed into the interpreter whenever the parser finds a expression of the following shape:

```
termPair:
( <term> , <term> )

termSndFst:
<term> . FST
<term> . SND
```

To develop the pair term and its related operations it was needed to include the following types:

```
tyPair of ty*ty
```

and the following terms:

```
tmPair of tm*tm
tmFst of tm
tmSnd of tm
```

For the evaluation of pair and its related operations it was done this way

```
tmPair(term_1, term_2) → tmPair(eval(term_1),
eval(term_2))
tmFst of tm → if (tm is tmPair(term_1, term_2)
then eval(term_1))
tmSnd of tm → if (tm is tmPair(term_1, term_2)
then eval(term_2))
```

Records: The records and their operations are inputed into the interpreter whenever the parser finds a expression of the following shape:

```
termRecordContent:
VSTRING : <term>
VSTRING : <term>, <termRecordContent>

termRecord:
{ <termRecordContent> }

termProyection:
<term> . VSTRING
```

To develop the record term and its related operations it was needed to include the following types:

```
tyRecord of list(string*ty)
```

and the following terms:

```
tmRecord of list(string*tm)
tmProjection of term*string
```

For the evaluation of pair and its related operations it was done this way

```
tmRecord(list(string*tm)) → foreach element in
list(string*tm) build_list (string*eval (tm))
tmProjection(term, string) → if (term is
tmRecord(list(string*tm))) then eval (assoc
string list(string*tm)
```

Lists: The lists and their operations are inputed into the interpreter whenever the parser finds a expression of the following shape:

```
termListContent:
<term>
<term> , <termListContent>

termList:
[ <termListContent> ]
[ ]

termIsEmpty
IS_EMPTY <term>

termHdTl
HD <term>
TL <term>
```

To develop the list term and its related operations it was needed to include the following types:

```
tyList of ty
tyEmptyList
```

and the following terms:

```
tmList of list(tm)
tmHead of tm
tmTail of tm
tmEmptyList
tmIsEmptyList of tm
```

For the evaluation of pair and its related operations it was done this way

```
tmList of list(tm) → foreach element in list(tm)
build_list eval(element)
tmHead of tm → if (tm is a list) then eval(hd
list)
tmTail of tm → if (tm is a list) then eval (tl
list)
tmEmptyList → tmEmptyList
tmIsEmptyList of tm → if (tm is tmEmptyList) then
true else if (tm is tmList(tm)) then false
```

Context of variables: To implement the context of variables it was needed to split the operations allowed in the interpreter between evaluation and binding. To achieve this a type operation was created:

```
type operation =
          Eval term
          Bind string * term
```

Also the type context was modified in order to store also strings that correspond to the name of the variables:

```
context = (string * (term * type))
```

In order to perform the variable substitution a function that allows us to evaluate the context and replace the variables with their corresponding term it was created. The way it works is similar as the already developed function eval1 works, but with the variable context.

That function is called after eval1, that is, whenever eval1 finds no more rules to apply we try to substitute the variables by its terms and try to evaluate again.

3. Summary of the implemented tasks and known errors

3.1 Implemented tasks

The following tasks of the assigned ones were implemented:

→ Multiline expressions recoginition
→ Recognition of expressions from files
→ Fixed-Point combinator implementation
→ Variable Context implementation
→ Implementation of String Type
→ Implementation of Pair Type
→ Implementation of Record Type
→ Implementation of List Type
→ Memory

3.2 Known errors

Whenever we try to apply a function stored inside a variable to another two variables the interpreter displays the definition of the function instead of actually applying it. However, the type displayed is the correct one. Probably the mistake happens during the variable context evaluation.

3.3 Notes

All the examples shown in the memory are inside examples.txt