USER MANUAL

Lambda-Calculus Evaluator

Authors: Raquel Gómez Antelo - raquel.antelo1@udc.es
Xoel González Pereira - xoel.gonzalezp@udc.es

Introduction

This manual serves as a small guide to the functionalities implemented in the lambda-calculus evaluator resulting from the work carried out in this assignment. In the following sections we will describe the modifications made to the original code (handed out by the teacher in the beginning) as well as why those changes were made.

Sections of the assignment

Section 1.1 Recognition of multi-line expressions

In order to add the feature of recognizing multi-line expressions versus the one line expressions already admitted by the interpreter, we chose to modify the main process instead of the lexer and the parser. We thought it would be a lot more simple since it only required modifying the amount of times we call the function “read_line()”. This function runs until the character of “end of instruction” appears, in our case the double ;.

Section 2.1 Internal fixed point combiner

We have added the internal fixed point combiner, which will let us declare functions in a directly recursive way without adding auxiliary functions. The parser.mly file has been modified by adding a new token called LETREC along with the necessary grammar rules for its correct functioning. These changes have been implemented in the term field, and lets the interpreter identify that the function being called is recursive. The LETREC implementation can be summarized as calling TmLetIn with one of its arguments being TmFix, a new term defined in order to represent the function calling itself. Once we have implemented this in the parser.mly file, we make changes to the lambda.ml file, focussing on the typeof and eval functions in which we have implemented the rules corresponding to this section. Thus, in the eval function we evaluate the term until a value is found and if the term is an abstraction then it has to be substituted by the given parameter for that abstraction. On the typeof function, the type of the term is checked to make sure that is the expected one.
Also, we added the corresponding implementations in the free_vars and subst functions.

Execution examples:

Multiplication:

```
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;;
```

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 5;;
```

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 5;;
```

Section 2.2 Global definitions context

In this section, we have added new features in our interpreter, letting us associate free variable names to values or terms, which can be used in lambda expressions once they are defined. First, we have changed the lamda.ml implementation by adding the command type, which has two functionalities:

- Eval : to evaluate terms.
- Bind : to bind a name to a term, thus, this will allow access to the variable in future evaluations.

Also, eval and eval1 have been modified, receiving also a context as parameter and the TmVar rule has been added to recover the value assigned to the variable. Also, a new function called apply_ctx has been added, which substitutes the name associated with a value by the value itself when evaluating terms.

At this point, the execute function has been added, which allows the interpreter to differentiate between Eval and Bind, and execute the corresponding instructions.

In the main file, we have to consider the new context and pass it as a parameter to the main loop function.

Section 2.3 Incorporation of type String and Concat

A new type is implemented called TyString and also a new term, TmString. Also, in order to complete the incorporation of String, we made the necessary pattern matching modifications to the functions in the lambda.ml file that required them. So the lexer.mll and parser.mly were able to recognize the String type, we added new rules with the appropriate definitions.

In addition, we have implemented the Concat function. We have created the new term TmConcat, which receives two terms of type String and then concatenates them. To make sure that both terms are of the same type, we use the eval function. In the lexer.mll and parser.mly files we created the CONCAT token, to represent the concatenation symbol (^) and the corresponding rules.

Section 2.4 and 2.5 Incorporation of Pairs and Tuples

At first, we made the pairs implementation, as they are simpler than tuples. Then, we expanded the concept to encompass more elements in tuples. First, we defined the type TyTuple of ty list and also its term, TmTuple of term list. Also, we defined all the implementations for the necessary functions in the lambda.ml file. According to parser.mly, we have created rules in which we identify if the tuple has one element or more. The elements can either be values or terms.

In order to access the elements of tuples, we created TmProj, which computes the projection of the desired position in the tuple. Therefore, it must be a numeric value. This implementation of projections has also been used in records, with a difference in the way the projection is called.

Section 2.6 Incorporation of Records

We have defined the type as TyRecord of (string * ty) list, and the term as TmRecord of (string * term) list. The records are tuples in which elements are associated with labels. We made the corresponding implementations in the lambda.ml file and in the parser.mly we created rules to identify empty records and records with one or more elements. As well as tuples, these elements can either be values or terms.

In order to access the elements of records, we used the TmProj defined previously, with a notorious difference, the elements of records can only be projected by its labels.

Also ,we changed the implementation of appTerm by changing the call to atomicTerm by creating a new rule called pathTerm, which simplifies the projections in records and tuples.

Section 2.7 Incorporation of Lists

We first defined the type of TyList of ty and its term has been divided in TmNil of ty, TmCons of ty * term * term, TmIsNil of ty * term, TmHead of ty * term and TmTail of ty * term. We have implemented the necessary implementations in the lambda.ml file to recognize the head and tail of the list as well as if the list is empty or not. In the lexer.mll file we have implemented a keyword for each list operation. In the parser.mly we added these keywords in rules in order to identify each one of them.

Section 2.8 Incorporation of Subtyping

An auxiliary function has been implemented in the lambda.ml file, called subtypeof, which redefines the implementation of the typeof function, by checking if two given types verify the subtyping relation.

Section 2.9 Incorporation of type Unit

We just added the recognition of the type Unit, for that, we created its type which is TyUnit and also its term, TmUnit. We made the necessary pattern matching modifications to the functions in the lambda.ml file that required them. In the lexer.mll we added keywords that represent the variations of the unit type. In the parser.mly we added the rules that recognize the unit type.