Lambda Calculus Interpreter: User's Guide

Daniel Quintillán Quintillán, Jorge Paz Ruza

Introduction

For this assignment, the task was to improve on the given Lambda Calculus interpreter with both usage-oriented features and extensions that are common to most modern programming languages. All of them were implemented over the lambda-3 version of the interpreter, and their usage will be addressed in the same order as they appear on the assignment instructions, providing examples.

Index

1.1 Support for multi-line expressions

Multi-line expressions are supported through the usage of an “expression end” token, which we have chosen for it to be the semicolon (**‘;’**). Of course, this also means that single-line expressions must now also have their end indicated with this token.

As a brief example, the user can enter the interactive loop and write the following expression:

```
>>
let x=
iszero(1)
in if x then 1 else 0;
```

And the program will recognise the semicolon as the end of each expression, returning the output:

```
-: 0 : Nat
>>
```

As expected, the system prompts the user to write another expression after processing and printing the output of the previous one.

1.2 Support for multi-expression lines

Multi-expression lines take advantage of the previously introduced “expression end” **‘;’** token to split lines into different expressions. Multiple expressions can be introduced in the same line if separated with a semicolon. For instance:

```
>>
let x=0 in iszero(x); iszero(1);
```

Will yield the following output:

```
-: true : Bool
-: false : Bool
```

1.3 File Input Support

To evaluate expressions from a text file, the user needs to specify its path using the flag “-f”.

```
$ ./top -f xpr.txt
```

The interactive loop and the file input support can be alternated seamlessly. In fact, the file xpr.txt can contain all of the aforementioned expressions, yielding exactly the same result.

```
$ more xpr.txt && ./top -f xpr.txt

let x=
iszero(1)
in if x then 3 else 4;
let x=0 in iszero(x); iszero(1);

-: 4 : Nat
-: true : Bool
-: false : Bool
```

2.1 Debug mode

The Debug mode prints the intermediate results of the evaluation process preceded by “#”. To enable it, the user just needs to execute the program with the flag -d.

```
$ ./top -d -f examples/1_3.txt

#let x = iszero(1) in if (x) then (3) else (4)
#iszero (1)
#let x = false in if (x) then (3) else (4)
#if (false) then (3) else (4)
#4
#3
#2
#1
#0
-: 4 : Nat
#let x = 0 in iszero (x)
#iszero (0)
#true
-: true : Bool
```

3.1 Fixed point operator

The *Fixed point operator* allows for general recursion in the Lambda Calculus interpreter. Using the convenient “letrec in” notation, the user no longer needs to manually write lengthy operators and their support structures every time they define a recursive function, since it will be implicit.

Since it's a classic example of recursive operations in lambda calculus, an implementation of the integer sum with this notation ought to be representative of its versatility.

```
$ more examples/3_12.txt && ./top -f examples/3_12.txt

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;

-: 55 : Nat
```

3.2 Global Variable Context

A variable context allows users to associate variable names with values or terms, for later usage in other lambda expressions. This association is created with *assignments*, which have the form:

```
identifier = term;
```

This way, the associated term can be used later by naming its identifier. Per example, we could execute the following lambda expressions:

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

And the output would be:

```
-: x=true : Bool
-: id=(lambda x:Bool. x) : (Bool) -> (Bool)
-: true : Bool
```

3.3 Pairs

By introducing pairs into our Lambda Calculus Interpreter, we're extending its usability beyond the concept of syntactic sugar: we introduce a new **term class**, *TmPair*, along with its left and right projections, *TmProj1* and *TmProj2*, as well as a new **type**, *TyProd*, that is, the product type, which allows us to have terms with arbitrary *α*β* type structure, including pairs themselves. This also allows for nested pair construction and projection as seen below:

```
>>
let x= lambda p:{Nat,Bool}.p.1
in x({{1,true},{0,false}}.2);
-: 0 : Nat
>>
```

3.4 Records

By extending our Lambda Calculus to support the declaration and usage of Records, we allow for a more accessible representation of more complex structured data types, as a generalization of pairs and tuples. We therefore again, a new **term class**, *TmRecord*, along with the general projection term *TmProj*, as well as a new **type,** *TyRecord*, which allows to have terms of arbitrary structure of the form *(string*term) list*, including records themselves, and allowing arbitrary nesting of this same type. Therefore, we can construct and use Records and Projections freely as seen in this example:

```
>>
let z= lambda r:{x:Nat,y:Bool}.r.x
in z({x={x=1,y=true},y={x=0,y=false}}.y)
-: 0 : Nat
>>
```

3.5 Subtyping

In our extended Lambda Calculus interpreter, subtyping is supported for functions and records.

For records, subtyping allows the user to apply functions that require record arguments of a certain structure to more specific records, which can differ from the original in both field arity and order. For example, this expression:

```
let func = lambda r:{x:{x:Bool,y:Nat},y:{x:Nat}}.r.x.x in
(func ({x={y=5,x=false,z=4},z=true,y={x=3}}));
```

Would not be valid without subtyping, because the actual argument differs from the expected one. However, it does contain every type of parameter needed by the abstraction, so it would be far more comfortable for the user to be able to evaluate it while changing neither the signature of the abstraction or the structure of the actual parameter.

Notwithstanding, when it comes to abstractions another factor needs to be accounted for. While the return type follows the previously discussed logic, the changes in the arguments need to be contravariant, i.e. the assessment `(s1->s2)<:(t1->t2)` requires that `t1<:s1` and `s2<:t2`. This counterintuitive constraint avoids situations where the subtyped function needs to manipulate information which is absent in the supertype, thus granting the safety of the type system.
An example of this behaviour can be seen below.

```
func2=lambda r:({x:Nat,y:Bool})->Bool.r ({x=1,y=true});
func3=lambda r2:{x:Nat}.iszero(r2.x);
func2 func3;
-: false : Bool
-: func2=(lambda r:({x:Nat, y:Bool}) -> (Bool). (r {x=1, y=true})) :
(({x:Nat, y:Bool}) -> (Bool)) -> (Bool)
-: func3=(lambda r2:{x:Nat}. iszero (r2.x)) : ({x:Nat}) -> (Bool)
-: false : Bool
```