Lambda Calculus Interpreter

Estevo Aldea Arias
Alejandro Fernández Otero

Practice Group: 2.11

Contents

1 Introduction

Throughout this memory it will be explained how to properly implement the different points of the practice, complemented with a couple of examples for every point.

1.1 User Manual

You can compile the code following these steps:

1. Compile the code with `make all`
2. Execute the program with `./top`
3. Use the program (finishing sentences with `;;`)
4. Use `Quit;;` to exit the program
5. Use `make clean` to erase the files that aren't needed

2 Implementation

2.1 Improvements

2.1.1 Multi-line recognition

To implement this functionality, the main loop was modified. The main change is that, while the input does not end with ";;", it is stored in a buffer. Finally, when the input ends with ";;", the buffer with the command is returned.

```
(* Example *)
x
:
Nat
=
4
;;
```

2.1.2 Pretty-printer

This section aims to improve the printing functionality, mainly by reducing the number of parentheses in the output. This is achieved with the usage of the Ocaml Format Module. The usage of this module can be seen in the "string_of_term" function. Precedence levels are defined for the different operators of the interpreter. These levels are used to decide when parenthesis are necessary and when they are not.

```
(* Example: This produce parenthesis*)
letrec sum : Nat -> Nat -> Nat =
    Ln:Nat. Lm:Nat.
        if iszero n then
            m
        else
            succ(sum(pred n)m)
in sum;;

(* Example: This does not produce parenthesis *)
iszero (pred (succ (succ 0)));;
```

2.2 Extensions of the lambda calculus language

2.2.1 Internal fixed-point combiner incorporation

The internal fixed-point combiner is implemented using the **letrec** term.

letrec is just an alias of the structure **let in fix**. First of all, the **parser** and the **lexer** were modified to accept the new syntax.

Following this, **lambda.ml** and its interface file were modified to support two new terms, **TmFix** and **TmAbs**.

TmAbs represents an abstraction of a function. It is used in combination with **TmFix** to achieve the goal of this section.

TmFix support the fix function behaviour in lambda-calculus.

```
(* Examples:*)
letrec sum : Nat -> Nat -> Nat =
    lambda n : Nat. lambda m : Nat.
        if iszero n then
            m
        else
            succ (sum (pred n) m)
in sum 3 5;;

letrec prod : Nat -> Nat -> Nat =
  lambda n : Nat. lambda m : Nat.
    if iszero n
    then 0
    else
      letrec sum : Nat -> Nat -> Nat =
        lambda x : Nat. lambda y : Nat.
          if iszero x
          then y
          else succ (sum (pred x) y)
```

```
    in sum m (prod (pred n) m)
in prod 3 5;;
```

```
letrec fib : Nat -> Nat =
  lambda n : Nat.
    if iszero n then 0
    else if iszero (pred n) then 1
    else
        letrec sum : Nat -> Nat -> Nat =
            lambda x : Nat. lambda y : Nat.
            if iszero x
            then y
            else succ (sum (pred x) y)
        in
        let n1 = pred n in
        let n2 = pred n1 in
        sum (fib n1) (fib n2)
in fib 7;;

letrec fact : Nat -> Nat =
  lambda n : Nat.
    if iszero n then
        succ 0
    else
        letrec add : Nat -> Nat -> Nat =
        lambda x : Nat. lambda y : Nat.
            if iszero x
            then y
            else succ (add (pred x) y)
        in
        letrec mul : Nat -> Nat -> Nat =
            lambda a : Nat. lambda b : Nat.
                if iszero a
                then 0
                else add b (mul (pred a) b)
            in
        mul n (fact (pred n))
in fact 5;;
```

2.2.2 Context for global bindings

We extended the commands so that users can introduce global bindings. The grammar now accepts three forms:

- `<id> = <term>` (type is inferred)
- `<id> : <Type> = <term>` (type-annotated binding)
- `<IdType> = <Type>` (type alias)

To accomplish this, we applied some core changes:

- **Parser** (`parser.mly`):
  - `IDV EQ term` → `BindInfer`
  - `IDV COLON ty EQ term` → `Bind`
  - `IDV EQ ty` → `TypeBind`
- **lambda.ml**:
  - `Bind`: expands type aliases (`expand_ty`), typechecks with subtyping, evaluates the term, prints, and stores `(name, type, value)` via `addvbinding`.
  - `BindInfer`: infers `typeof ctx term`, evaluates, prints, and stores the binding.
  - `TypeBind`: expands the type, prints `type N = Nat`, and stores it in the type context (`addtbinding`); later lookups are resolved by `expand_ty` and handled in `typeof`.
- **Context pipeline**: bindings are stored in an immutable list (functional context). Redefinitions shadow previous ones without mutation, preserving referential transparency.

```
(* Example: Global value binding with inferred type *)
x = true;;

(* Example: Annotated binding and later use *)
id : Bool -> Bool = lambda x:Bool. x;;
id x;;

(* Example: Type alias and use *)
N = Nat;;
lambda y:N. y;;

(* Example: Functional shadowing (new binding overrides the old) *)
x = 0;;
id x;;
(* >> Type error: parameter type mismatch *)
```

2.2.3 String type

String support was added by updating the **parser** and **lexer**, and including basic cases in the lambda module for this new type.

Concat function was implemented by adding a new term **TmConcat**.

```
"Hello, World!";;
concat "Hello," " World!";;
```

2.2.4 Tuple type

Tuples were implemented by adding a new type (**TyTuple**), which is a list of types, and a new term (**TmTuple**), which is a list of terms. Similarly, a term for the projection operation (**TmProj**) was implemented. It receives a tuple and returns the n-th element of the tuple. The lexer and the parser were also modified to properly support the new functionality.

```
(* Example of a Nat x Bool tuple *)
{1, false};;

(* Example of a tuple assignation *)
x : {Nat, String} = {1, "Hello, World!"};;

(* Example of projection *)
x.2;;
{1,false,"hi",4}.3;;
```

2.2.5 Register type

Register's logic is similar to the tuples one. First, a new type was created, **TyRecord**. After that, terms **TmRecord**, a list of String * term; and **TmRProj** were implemented. TmRProjc receives a TmRecord and a String and returns the term associated with that String.

```
(* Example of a record *)
{greeting="Hi!", meaning=42};;

(* Example of a record assignation *)
x : {greeting : String, meaning : Nat} = {greeting = "Hi!", meaning =
    42};;

(* Example of projection *)
x.greeting;;
x.meaning;;
```

2.2.6 Variants type

We added algebraic variants with explicit labels and pattern matching.

- **Syntax**: new type constructor `TyVariant of (label * ty) list`, and terms `TmVariant (label, term, ty)` for values like `<left = 1> as <left:Nat, right:Bool>` and `TmCase` for pattern matching `case v of <lbl = x> => e | ...`. Tokens `case`, `of`, `|`, `=>`, `as`, and delimiters `< >` were added to lexer/parser.
- **Typing** (`lambda.ml`):
  - `TmVariant`: the annotation must be a `TyVariant`; the label must exist and the payload's type must be a subtype of the declared one; the whole term has the annotated variant type.
  - `TmCase`: the scrutinee must be a `TyVariant`; each branch label must belong to it, and the branch body is typed under a context extended with the payload type. All branches must return the same type (checked via mutual subtyping).
- **Evaluation**: a variant is a value when its payload is. `case` on a variant value selects the matching branch and substitutes the binder; otherwise the it's evaluated from left to right. Pretty-printing shows `<lbl = v> as <...>` and formats the `case` with labeled branches.

```
(* Pattern match on a two-way variant *)
let v = <right = 1> as <left:Nat, right:Bool> in
case v of
  <left = x> => x
| <right = y> => if y then 1 else 0;;

(* Passing variants to functions *)
let chooser = lambda w: <a:Nat, b:Nat>.
  case w of
    <a = x> => x
  | <b = y> => succ y
in chooser (<b = 2> as <a:Nat, b:Nat>);;
```

```
(*Type definition:*)
Int = <pos:Nat, zero:Bool, neg:Nat>;;

(*Helper functions*)
sum : Nat -> Nat -> 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;;

sub : Nat -> Nat -> Nat =
  letrec sub : Nat -> Nat -> Nat =
    lambda n:Nat. lambda m:Nat.
      if iszero m then n
      else if iszero n then 0
      else sub (pred n) (pred m)
  in sub;;

leq : Nat -> Nat -> Bool =
  letrec leq : Nat -> Nat -> Bool =
    lambda n:Nat. lambda m:Nat.
      if iszero n then true
      else if iszero m then false
      else leq (pred n) (pred m)
  in leq;;

(*Add function*)
add : Int -> Int -> Int =
  lambda x:Int. lambda y:Int.
    case x of
      <pos = p> =>
        (case y of
           <pos = q> => (<pos = sum p q> as Int)
         | <zero = z> => (<pos = p> as Int)
         | <neg = n> =>
             if leq p n then (<neg = sub n p> as Int)
             else (<pos = sub p n> as Int))
    | <zero = z> => y
    | <neg = n> =>
        (case y of
           <neg = m> => (<neg = sum n m> as Int)
         | <zero = z> => (<neg = n> as Int)
         | <pos = p> =>
             if leq p n then (<neg = sub n p> as Int)
             else (<pos = sub p n> as Int));;

(*Examples*)
add (<pos = 3> as Int) (<neg = 5> as Int) ;; --> <neg = 2>
add (<pos = 3> as Int) (<pos = 5> as Int) ;; --> <pos = 8>
```

```
add (<neg = 3> as Int) (<pos = 5> as Int) ;; --> <pos = 2>
add (<neg = 3> as Int) (<neg = 5> as Int) ;; --> <neg = 8>
add (<zero = true> as Int) (<pos = 5> as Int) ;; --> <pos = 5>
```

2.2.7 Lists type

We added a list type and the usual functions to build and deconstruct lists

- **Syntax**: new type constructor `TyList` and terms `nil`, `cons`, `isnil`, `head`, `tail` (`TmNil`, `TmCons`, `TmIsNil`, `TmHead`, `TmTail`). Tokens `List`, `nil`, `cons`, `isnil`, `head`, `tail` were added to lexer and parser.
- **Typing** (`lambda.ml`):
  - `nil T` has type `List T`.
  - `cons h t` requires `typeof h = U` and `typeof t = List U`.
  - `isnil` expects a list and returns `Bool`
  - `head` returns the element type
  - `tail` returns `List U`.
- **Evaluation**: lists are values when both head and tail are values; `isnil nil` → `true`, `isnil (cons ...)` → `false`; `head (cons v w)` → `v`; `tail (cons v w)` → `w`. Otherwise, arguments are evaluated left-to-right. The pretty-printer detects lists and prints them as `[a; b]`.

```
(* Primitives *)
nil Nat;;
cons 1 (nil Nat);;

(* length: List Nat -> Nat *)
letrec length : List Nat -> Nat =
  lambda l: List Nat.
    if isnil l then 0 else succ (length (tail l))
in length (cons 1 (cons 2 (nil Nat)));;

(* append: List Nat -> List Nat -> List Nat *)
letrec append : List Nat -> List Nat -> List Nat =
  lambda l1: List Nat. lambda l2: List Nat.
    if isnil l1 then l2 else cons (head l1) (append (tail l1) l2)
in append (cons 1 (cons 2 (nil Nat))) (cons 3 (nil Nat));;

(* map: (Nat -> Nat) -> List Nat -> List Nat *)
letrec map : (Nat -> Nat) -> List Nat -> List Nat =
  lambda f: Nat -> Nat. lambda l: List Nat.
    if isnil l then nil Nat else cons (f (head l)) (map f (tail l))
in map (lambda x:Nat. succ x) (cons 1 (cons 2 (nil Nat)));;
```

2.2.8 Subtyping

We introduced subtyping for functions and tuples.

- **Relation** (`lambda.ml`): a recursive `subtype` checks:
  - **Functions**: S_1`->`S_2 `<:` T_1`->`T_2 if T_1 `<:` S_1 (contravariant arguments) and S_2 `<:` T_2 (covariant returns).
  - **Tuples**: width+depth subtyping; a longer tuple is subtype of a shorter one if its prefix components are pairwise subtypes.
  - Otherwise only reflexivity applies.
- **Typing** uses `subtype` instead of strict equality:
  - Application: argument type must be a subtype of the parameter type.
  - `fix`: result type must be a subtype of the domain.
  - Toplevel `Bind`: stored value may be a subtype of the annotated type.

```
(* Width on tuples: {Nat,Bool} <: {Nat} *)
let f = lambda t: {Nat}. t.1 in f {1, true};;

(* Contravariance on functions: ({Nat,Bool})->Nat <: {Nat}->Nat *)
let use = lambda g: ({Nat, Bool}) -> Nat. g {0, true}
in let short = lambda t:{Nat}. t.1 in use short;;
```