Technical and User Manual

Extended Lambda Calculus Interpreter

David Manuel Barbosa Rodríguez, Álvaro Silva Silva

December 2, 2025

Contents

1 Introduction

The main objective of this project is the implementation of an evaluator and interpreter for a functional programming language based on the **Lambda Calculus with Simple Types (STLC)**. Developed in **OCaml**, this system extends the base calculus by incorporating features found in modern programming languages:

- **Base Types:** Naturals, Booleans, and Strings.
- **Data Structures:** Tuples, Records, Variants (Sum Types), and Lists.
- **Recursion:** Native support via fixed-point combinators.
- **Static Typing:** Rigorous type checking prior to execution.

The explanation of the examples and code types is based on this memory and on the comments added in the code.

2 System Architecture

The project design is modular, favoring the separation of concerns into four key components:

A. Lexer (`lexer.mll`)
Generated using `ocamllex`. It is responsible for lexical analysis; it transforms the input character stream into a sequence of valid tokens (e.g., `LAMBDA`, `IF`, `IN`, `STRINGV`, `INTV`).

B. Parser (`parser.mly`)
Generated using `ocamlyacc`. It performs syntactic analysis by consuming tokens to build the **Abstract Syntax Tree (AST)**. It also manages the syntactic translation of complex constructions like `letrec`.

C. Core (`lambda.ml`)
The heart of the interpreter. It defines:

- Algebraic data types for Types (`ty`) and Terms (`term`).
- The evaluation context.
- Type inference and checking logic (`typeof`).
- Step-by-step evaluation semantics (`eval1`, `eval`).

D. Main Interface (`main.ml`)
Implement the Read-Eval-Print Loop (**REPL**). It manages data input (including support for multi-line blocks) and formats result output.

3 Language Definition

The language strictly distinguishes between Types and Terms.

3.1 Types (`ty`)

- **Atomic:** `Bool`, `Nat`, `String`, `Unit`.
- **Constructors:**
  - `Arrow (->)`: Functions.
  - `List T`: Homogeneous lists.
  - `Tuple {T1, ...}`: Ordered sequences.
  - `Record {l:T, ...}`: Labeled fields.
  - `Variant <l:T, ...>`: Sum types (tagged unions).

3.2 Terms (`term`)

- **Control:** `if-then-else`, `case-of` (pattern matching).
- **Functions:** `lambda x:T. t`, application `t1 t2`, `let x=t1 in t2`.
- **Recursion:** `fix t`, and the derived `letrec`.
- **Data Creation:** `{v1, v2}` (tuples), `{l=v}` (records), `<l=v> as T` (variants), `cons/nil` (lists).
- **Data Access:** `t.i` (tuple projection), `t.l` (record projection), `head/tail` (lists).

4 Technical Features and Semantics

4.1 Recursion and `letrec`

To allow recursive function definitions, we implemented the fixed-point combinator `TmFix`.

- **Syntactic Sugar:** The user command `letrec f = lambda x:type. body` is automatically parsed into `let f = fix (lambda f:type. lambda x:type. body)`. This simplifies the AST while providing a user-friendly syntax.

4.2 Global Context

The interpreter maintains a persistent environment.

- **Values:** Users can bind terms to variables using `x = term;;`.
- **Types:** Users can define type aliases using `T = Type;;`. The system includes cycle detection in the alias resolution logic to prevent infinite loops (e.g., $A = B, B = A$).

4.3 Tuples, Records, and Unit

Tuples and Records share similar projection logic but differ in access method (index vs. label).

- **The Unit Type:** The empty tuple `{}` is treated as the `Unit` type. In our subtyping system, we treat `Unit` as a supertype for records, allowing generic handling of structured data.

- **Visual Clarity:** To avoid ambiguity, the pretty-printer renders `{}` specifically as `unit` (value) or `Unit` (type), reserving braces `{...}` for non-empty structures.

4.4 Subtyping (<:)

We implemented a robust subtyping relation to increase language flexibility:

- **Records:** Width subtyping (extra fields are allowed) and Permutation subtyping (field order is irrelevant).
- **Variants:** A variant value is valid if its payload type is a subtype of the expected variant field type.
- **Functions:** Contravariant in argument types and covariant in return types.

The language strictly distinguishes between **Types** (metadata describing values) and **Terms** (computable expressions).

Syntax Note: Instructions must end with a double semicolon (`;;`) to be processed by the interpreter.

4.5 Data Types (AST - Type `ty`)

Types must begin with an uppercase letter. Type aliases can be created using `BindTy`.

A. Primitive Types

- **TyBool:** Booleans.

```
Aliasbool = Bool;;
```

- **TyNat:** Natural Numbers.

```
Aliasnat = Nat;;
```

- **TyString:** Character Strings.

```
Text = String;;
```

B. Compound Types

- **TyArr:** Functions (Arrows).

```
Operation = Nat -> Nat;;
```

- **TyTuple:** Tuples (Ordered sequence of types).

```
Pair = {Nat, Bool};;
```

- **TyRcd:** Records (Labeled fields).

```
Point = {x:Nat, y:Nat};;
```

- **TyVariant:** Variants (Labeled unions).

```
Integer = <pos:Nat, zero:Bool, neg:Nat>;;
```

- **TyList:** Homogeneous lists.

```
Numlist = List Nat;;
```

C. Type Variables

- **TyVar:** Previously defined aliases.

```
Coordinate = 3;;
Point3d = {Coordinate, Coordinate, Coordinate};;
```

4.6 Terms (AST - Type `term`)

Terms represent the program logic. Primitive values and keywords are usually written in lowercase.

A. Control Flow

- **Conditionals** (`TmIf`)**:**

```
if true then 5 else 0;;
```

- **Pattern Matching** (`TmCase`)**:** Requires a defined variant type.

```
Int = <pos:Nat, zero:Bool, neg:Nat>;;
abs = L i : Int.
  case i of
    <pos=p> => (<pos=p> as Int)
  | <zero=z> => (<zero=true> as Int)
  | <neg=n> => (<pos=n> as Int);;
```

B. Arithmetic and Logic

- **Booleans** (`TmTrue`, `TmFalse`)**:**

```
true;;
false;;
```

- **Naturals** (`TmZero`, `TmSucc`)**:** Literal numbers are internally converted to successors.

```
0;;
succ (succ 0);; (* Equivalent to 2 *)
```

- **Operators** (`TmPred`, `TmIsZero`)**:**

```
pred 5;;
iszero 0;;
```

C. Functions and Recursion

- **Abstraction** (`TmAbs`):

```
lambda x:Nat. succ x;;
```

- **Application** (`TmApp`):

```
(lambda x:Nat. succ x) 5;;
```

- **Local Binding** (`TmLetIn`):

```
let x = 5 in succ x;;
```

- **Recursion** (`TmFix` / `letrec`):

```
letrec f : Nat -> Nat =
  lambda x : Nat . if iszero x then 0 else f (pred x)
in f 5;;
```

D. Data Structures

- **Tuples** (`TmTuple`) **and Projection:**

```
t = {10, true, "hello"};;
t.1;;
```

- **Records** (`TmRcd`) **and Projection:**

```
p = {x=5, y=10};;
p.x;;
```

- **Variants** (`TmVariant`):

```
Status = <ok:Nat, error:String>;;
res = <ok=200> as Status;;
```

- **Lists** (`TmCons`, `TmNil`):

```
l = cons [Nat] 1 (cons [Nat] 2 (nil [Nat]));;
head [Nat] l;;
isnil [Nat] l;;
```

- **Strings** (`TmString`):

```
concat "Hello " "World";;
```

4.7 System Commands

Top-level instructions that interact with the global environment.

1. **Pretty Printer**

```
>> let x =4
in succ(x);;
(* x : Nat = 5;; *)
```

2. **BindTy** (`Type = ty`)**:** Defines a global type alias.

```
>> Coordinate = {Nat, Nat};;
(* Coordinate = {Nat, Nat} *)
```

3. **Eval** (`term`)**:** Evaluates an expression and prints the result without saving it.

```
>> if true then 1 else 0;;
(* 1 : Nat *)
```

4. **Quit** (`quit`)**:** Closes the interpreter.

```
>> quit;;
```

5 Static Semantics (Typing)

The system employs strong, static typing. The `typeof` function verifies correctness before executing any calculation.

- **Subtyping:** Implements the inclusion relation $S <: T$. It is fundamental for flexible record handling.
  - *Width Rule:* A record with more fields (e.g., `{x:Nat, y:Nat}`) is a subtype of one with fewer fields (e.g., `{x:Nat}`). This allows passing complex objects to functions expecting simple interfaces.
- **Alias Resolution:** Before validating types, the system recursively resolves all user-defined `TyVar`. It includes cycle detection (e.g., $A = B$ and $B = A$), raising a `Type_alias_loop` exception.
- **Critical Validations:**
  - **If-Else:** Both branches must unify to the same type.
  - **Case:** All destruction branches of a variant must return exactly the same type.
  - **Lists:** Must be strictly homogeneous (all elements of the same type).

6 Dynamic Semantics (Evaluation)

The interpreter uses a **small-step semantics** strategy.

1. **Substitution and Renaming:** The `subst` function is the engine of computation. It implements *alpha-conversion* (variable renaming) to avoid accidental capture of free variables in closures or nested functions.
2. **Recursion (Fixed Point):** Implemented via the `TmFix` term. The reduction rule `E-FixBeta` substitutes the recursive term inside the function body only when necessary, enabling logical loops.

3. **Structure Evaluation:**
   - **Lists:** `cons` constructors are strict (they evaluate the head before building the list).
   - **Variants:** The `case` expression evaluates the term to a label and dynamically selects the corresponding code branch.
   - **Strings:** Uses the host language's native concatenation after reducing operands to values.

7 Installation and Usage Guide

7.1 Compilation

The project includes a `Makefile` to automate the process.

1. Open a terminal in the project root.
2. Run the command:

   ```
   $ make
   ```

   This will generate the binary executable named `top`.

7.2 Execution

To start the interactive environment:

```
$ ./top
```

The system will display the prompt ». You can enter multi-line code. To execute the block, end with `;;` and press Enter.

8 Usage Examples

8.1 String Manipulation

```
>> greeting = "Hello";;
>> name = "User";;
>> concat greeting (concat " " name);;
(* "Hello User" : String *)
```

8.2 Lists of Naturals

```
>> l = cons [Nat] 10 (cons [Nat] 20 (nil [Nat]));;
>> head [Nat] l;;
(* 10 : Nat *)
>> tail [Nat] l;;
(* (cons [Nat] 20 nil [Nat]) : List Nat *)
```

8.3 Pattern Matching with Variants

```
>>> typesensor = < temp : Nat , error : String , off : Bool >;;

valor = L i : typesensor.
  case i of
      < temp = t >       => ( < temp = 999 > as typesensor )
    | < error = msg >    => ( < error = msg > as typesensor )
    | < off = b >        => ( < off = b > as typesensor );;
```

8.4 Recursive Function (Factorial)

```
>> add =
  letrec add_aux : Nat -> Nat -> Nat =
    lambda n: Nat. lambda m: Nat.
      if iszero n then m else succ (add_aux (pred n) m)
  in add_aux 3 4;;
24 : Nat
```

9 Usage Examples

All the exercises mentioned in the practice sections are done in examples.md