Advanced Lambda Calculus Interpreter memory

Pablo Pajón Area, Rafael de Almeida Leite

December 6, 2024

Contents

1 Introduction

This document describes a lambda calculus interpreter that extends a simple typed lambda calculus with several advanced features: multi-line expression input, pretty-printing, tuples, records, variants, lists, and subtyping. This interpreter allows a richer type system and more expressive programs, supporting features often found in more advanced functional languages. Besides these, fixed-point combinators, gloal context of definitions and strings were also implemented, but since those were done in class with the assistance of the professor, this document won't go in extensive detail about them.

The document is divided into two main parts:

1. A **user manual** illustrating how to use the new features, including syntax and execution examples.
2. A **technical manual** explaining, section by section, the changes introduced into the original code to implement these features.

2 User Manual

2.1 Multi-line Input

The interpreter now allows typing expressions over multiple lines. Users can continue writing their code until they type a semicolon ';', which signals the end of the expression. For example:

```
>> let x =
       (lambda y:Nat. succ y)
   in x 2;
```

2.2 Pretty Printing

The interpreter's pretty-printer tries to omit unnecessary parentheses and neatly indent complex expressions. This makes the output more readable. For example:

```
>> (lambda x:Nat. lambda y:Nat. x) 3;
- : Nat -> Nat =
lambda y:Nat.
3
```

2.3 Tuples

Tuples are introduced with the syntax `{t1, t2, ...}`. They support arbitrary length. Projection is done via zero-based indexing: The projection was also implemented with the syntax `tuple.n`:

```
>> {3, true, "hello"};
- : {Nat, Bool, String} = {3,true,"hello"}

>> ({3,true,"hello"}.1);
- : Bool = true

>> y = {1, {"string", true}};
y : {Nat,{String,Bool}} =
{1,{"string",true}}

>> y.1.1;
- : Bool =
true
```

2.4 Records

Records are similar to tuples but have labeled fields:

```
>> { x:3, y:true };
- : {x:Nat, y:Bool} = {x: 3, y: true}

>> ({x:3, y:true}.y);
- : Bool = true

>> {num:1, str: "string", tup:{1, "string", true}}.tup.2;
- : Bool =
true
```

2.5 Variants

Variants are tagged unions. They are defined with a variant type and values injected using a tag:

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

>> p3 = <pos=3> as Int;
- : Int = <pos = 3>

>> z0 = <zero=true> as Int;
- : Int = <zero = true>
```

`case` expressions allow pattern matching on variants:

```
>> abs = lambda i:Int.
   case i of
     <pos=p> => <pos=p> as Int
   | <zero=z> => <zero=true> as Int
   | <neg=n> => <pos=n> as Int;

>> abs p3;
- : Int = <pos = 3>
```

2.6 Lists

Lists have type `List[T]` and can be constructed with `cons[T] element tail` or the empty list `nil[T]`:

```
>> cons[Nat] 1 (cons[Nat] 2 (nil[Nat]));
- : List[Nat] = cons[Nat] 1 (cons[Nat] 2 (nil[Nat]))
```

Operations:

- `head[T] list`

- `tail[T] list`
- `isnil[T] list`

2.7 Subtyping

The interpreter supports subtyping for records and functions, allowing more flexible typing. If a record has at least the fields of another record with compatible types, it is considered a subtype. Functions allow contravariance in parameters and covariance in results.

```
>> let
        getx = (lambda r : {x:Nat}. r.x)
   in
        getx {x:0, y:true};

- : Nat =
0

>>let
        f = (lambda g : {x:Nat,y:Bool} -> Nat. g {x:0,y:true})
   in
        f (lambda r : {x:Nat}. r.x);

- : Nat =
0
```

3 Technical Manual

This section explains the key modifications introduced to the original code to support the features described above. For each feature, we detail the changes made to the lexer, parser, type system, evaluation rules, and the underlying data structures.

3.1 Multi-line Input

Goal: Allow the user to enter complex expressions spanning multiple lines, only ending when ‘;‘ is encountered.

Changes Made:

- `main.ml`: Implemented a function `leer_hasta_punto_y_coma` that reads multiple lines until a ‘;‘ character is found. This logic accumulates the input in a buffer.
- By doing this, the parser is given a complete expression at once, regardless of how many lines the user needed to write it.
- No changes to the core language semantics were required. Only the input loop was adapted.

3.2 Pretty Printing

Goal: Improve the readability of printed terms by minimizing unnecessary parentheses and providing a cleaner layout.

Changes Made:

- In `lambda.ml`, the functions `string_of_term` and `string_of_term_at_level` were revised.
- Logic was introduced to:
  - Omit parentheses in obvious contexts.
  - Use indentation levels and line breaks for `if`, `case`, `lambda`, and other constructs.
- No changes to the parser or evaluator. The pretty printer is purely a presentation layer.

3.3 Tuples

Goal: Introduce tuples of arbitrary length and indexed projections.

Changes Made:

- **Type System:** Added `TyTuple` with a list of element types.
- **Term Constructors:** Introduced `TmTuple` as a list of terms for runtime values. `TmProj` handles projection by integer index.
- **Type Checking:** In `typeof`, when encountering `TmTuple`, we iterate over the element list and build a list containing the type of each element contained in the tuple. For `TmProj`, we ensure the term is a tuple and the index is within bounds, if it is we return the type of the element of the position indicated by the index.
- **Evaluation:** In `eval1`, if a TmTuple is received we call eval1 for each of the elements in the tuple. For the projection if the term is fully evaluated as TmTuple, we return the element for the corresponding index, else we continue evaluating the term.
- **Parser/Lexer:** Adjusted grammar rules to parse `{ t1, t2, ... }` into a `TmTuple`. Added rules for ‘term DOT INTV‘ to represent projection and ’proj DOT INTV’ to represent successive projections.

3.4 Records

Goal: Implement records as labeled collections of fields and support label-based projections.

Changes Made:

- **Type System:** Introduced `TyRecord` as a list of pairs of string and types, which maps field names to types.
- **Term Constructors:** `TmRecord` as a list of pairs of strnig as terms for values and `TmRProj` for label projections.
- **Type Checking:** The `typeof` function checks that all fields in a record expression type check consistently. For projection, it checks the record’s fields to find the requested label and returns the type of the term associated with it.

- **Evaluation:** For `TmRProj`, once the record is evaluated, we simply return the appropriate field's value if found. For TmRecord we iterate over the list and evaluate each of its elements.
- **Parser/Lexer:** Added grammar rules to parse records as `{ x: t, y: t', ... }` and label projections as 'term DOT IDV' and 'proj DOT IDV' to for successive projections that can combine record and tuple projections.

3.5 Variants

Goal: Introduce variant types to allow the definition of tagged unions, enabling the representation of data that can take on one of several distinct forms, each potentially carrying its own associated data.

Rationale: Functional languages often rely on variants (also known as tagged unions) to represent data that can have multiple, labeled forms. For instance, a type `Int` could be represented as `<pos:Nat, zero:Bool, neg:Nat>` to distinguish between positive numbers, zero, and negative numbers. Each branch (`pos`, `zero`, `neg`) associates a specific type of data with a unique label. By incorporating variants, the interpreter can model richer data structures and perform pattern matching to handle each form appropriately.

Changes Made:

- **Type System Extension:**
  - Introduced the `TyVariant` constructor to represent variant types. A `TyVariant` takes a list of (*label*, *ty*) pairs, defining each possible form the variant can take.
  - By adding `TyVariant`, we enable users to define new complex data types beyond just `Nat`, `Bool`, `String`, tuples, and records. Now, users can create new variant types that specify multiple labeled alternatives, each with its own type.
- **Term Constructors:**
  - Introduced `TmTag(label, term, ty)` for creating variant values. `TmTag` injects a *term* into a particular variant under a specific *label*.
  - Introduced `TmCase(scrutinee, cases)` for pattern matching. A case expression allows the user to analyze a variant value based on its label and execute a corresponding branch.
- **Type Checking:**
  - On encountering `TmTag(l, t, TyVariant(...))`, the type checker ensures:
    - The provided label `l` exists in the variant's type definition.
    - The term `t` matches the expected type associated with that label.
  - On encountering `TmCase(t0, cases)`, the type checker:
    - Verifies that `t0` is a variant type.
    - Ensures each branch in `cases` corresponds to a label defined in the variant's type.
    - Checks that all branches produce the same resulting type, ensuring type safety in pattern matching.
- **Evaluation:**
  - The evaluation strategy reduces the scrutinee `t0` of a `TmCase` until it becomes a `TmTag(l, v, tyVariant)` form, where `v` is a value.

- Once the label `l` is determined, the interpreter selects the matching branch in the case expression, substitutes the bound variable with `v`, and continues the evaluation of that branch.

- **Parser/Lexer Adjustments:**
  - Extended the grammar to parse variant type declarations such as:

    ```
    Int = <pos:Nat, zero:Bool, neg:Nat>
    ```

  - Extended the grammar to parse tagged values:

    ```
    <pos=3> as Int
    ```

  - Extended the grammar for case expressions:

    ```
    case i of <pos=p> => ...  | <zero=z> => ...  | <neg=n> => ...
    ```

Outcome: By adding `TyVariant` and the corresponding term forms, the interpreter now supports a fundamental feature of many typed functional languages: general, labeled sum types. This allows users to define more abstract and expressive data structures, perform safe pattern matches, and write more robust, polymorphic code.

3.6 Lists

Goal: Introduce lists as sequences of elements of a single type, along with operations `head`, `tail`, and `isnil`.

Changes Made:

- **Type System:** Introduced `TyList ty` for lists of type 'ty'.
- **Term Constructors:**
  - `TmNil ty` for the empty list.
  - `TmCons (ty, head, tail)` for constructing a non-empty list.
  - `TmHead(ty, t)`, `TmTail(ty, t)`, `TmIsNil(ty, t)` for operations.
- **Type Checking:**
  - `TmCons` checks that 'head' is of type 'ty' and 'tail' is `List[ty]`'.
  - `TmHead` and `TmTail` require a list type as input.
  - `TmIsNil` requires a list and returns 'Bool'.
- **Evaluation:**
  - If `TmHead` is applied to `TmCons`, return the head element.
  - If `TmTail` is applied to `TmCons`, return the tail element.
  - `TmIsNil` checks if the given list is 'nil' or 'cons'.
- **Parser/Lexer:** New tokens and grammar rules for `cons[T] head tail` and `nil[T]`.

3.7 Subtyping

Goal: Support subtyping for function and record types, enabling more general and flexible type checking.

Changes Made:

- **Type Relations:**
  - Implemented `is_subtype ctx s t` function. It checks:
    - `Top` and `Bot` rules for universal super-/subtype.
    - Function subtyping: contravariance in parameters, covariance in results.
    - Record subtyping: A record with more fields can be a subtype of a record with fewer fields if their corresponding fields are subtypes.
- **Type Checking:**
  - Updated `typeof` to use `is_subtype` instead of strict equality in places like if-expressions, applications, and fixpoint checks.
  - When checking application `TmApp(t1, t2)`, we now require 'typeof(t2)' to be a subtype of the function parameter type.
  - For conditionals, if `t2` and `t3` differ but one is a subtype of the other, we pick the supertype.