DLP Practicum

Pablo García López - pablo.glopez@udc.es
David Javier Montes Fernández - davidj.montes@udc.es

Contents

1 User Manual

1.1 The REPL and basic usage

The entry point of the interpreter is a read–eval–print loop (REPL). At the prompt `>>` the user can type either:

- A term to be evaluated and type-checked.
- A global definition (term or type alias).
- A command such as `clear` to wipe terminal or `quit` to exit.

Expressions are terminated with `;;`, which allows multi-line input: line breaks do not automatically finish an expression. The REPL collects lines until it sees `;;` and then sends the whole block to the lexer and parser.

Example:

```
>> lambda x:{Nat,Nat}.x.1;;
- : {Nat * Nat} -> Nat = (lambda x:{Nat * Nat}.x.1)
```

1.2 The fixed-point combinator: `letrec`

Recursion is supported directly via `letrec`, without the need to encode a fixed-point combinator manually. A recursive definition has the form:

```
letrec f : A -> B =
lambda x : A. ... f ...
in
f;;
```

The following recursive functions are provided as examples.

Addition on natural numbers

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

Product

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

Fibonacci numbers

```
fib =
  letrec fib : Nat -> Nat =
    lambda n : Nat.
      if iszero n then
        0
      else if iszero (pred n) then
        1
      else
        sum (fib (pred n)) (fib (pred (pred n)))
  in fib
;;
```

Factorial

```
fact =
  letrec fact : Nat -> Nat =
    lambda x : Nat.
      if iszero x then 1 else prod x (fact (pred x))
  in fact
;;
```

These definitions show that `letrec` behaves as expected for typical recursive patterns over `Nat`.

1.3 Global term definitions

Global term definitions have the form:

```
identifier = term;;
```

For example:

```
>> x = true;;
x : Bool = true

>> id = lambda x : Bool. x;;
id : Bool -> Bool = <fun>

>> id x;;
- : Bool = true
```

Once defined, a name can be reused in later expressions, and the most recent binding shadows previous ones (more on that in the technical section).

1.4 Global type aliases

The interpreter also supports global type aliases, with syntax:

```
Identifier = Type;;
```

For instance:

```
>> N = Nat;;
type N = Nat

>> lambda x : N. x;;
- : N -> N = <fun>
```

The alias `N` is a user-level name. Internally, the type checker resolves `N` to the actual type `Nat` before performing any checks. This behaviour is symmetric for more complex aliases (e.g. lists, tuples, variants).

1.5 Strings and concatenation

The language includes a base type `String` and two term constructs:

- String literals: `"hello"` of type `String`.
- Concatenation: `t1 ^ t2` where both `t1` and `t2` have type `String`.

Evaluation:

- String literals are values.
- Concatenation evaluates its arguments and then joins the underlying OCaml strings.

Example:

```
>> "Hello" ^ " " ^ "world";;
- : String = "Hello world"
```

1.6 Tuples and positional projection

The language supports tuples of arbitrary arity and heterogeneous types. Intuitively, a tuple is written as:

```
{t1, t2, ..., tn}
```

and its type as:

```
{T1, T2, ..., Tn}
```

Projection is positional and uses an index, for instance:

`t.1`, `t.2`, etc.

Some examples (informal, the exact concrete syntax follows the parser):

```
>> let p = {true, 3} in p.1;;
- : Bool = true

>> let p = {true, 3} in p.2;;
- : Nat = 3
```

The type checker enforces that the index is within bounds and returns the corresponding component type.

1.7 Records and labeled projection

Records are like tuples with named fields. A record value is written as:

```
{ label1 = t1, label2 = t2, ... }
```

and its type as:

```
{ label1 : T1, label2 : T2, ... }
```

Projection is by label:

```
t.label
```

Example:

```
>> let r = {x = 3, y = true} in r.x;;
- : Nat = 3

>> let r = {x = 3, y = true} in r.y;;
- : Bool = true
```

Records interact naturally with other features (functions, lists, variants). In the extended type system, they can also be used as the basis for subtyping.

1.8 Variants and integers

Variants allow representing tagged unions of heterogeneous values. A variant type is written as:

```
<label1 : T1, label2 : T2, ..., labeln : Tn>
```

Values of that type are constructed using a single label:

```
<label = t> as VariantType
```

Pattern matching uses a `case` expression over all labels.

Example: integer type `Int`

We define a variant type `Int` which represents signed integers via a sign and a natural magnitude:

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

Some values:

```
p3 = <pos = 3> as Int;;
z0 = <zero = true> as Int;;
n5 = <neg = 5> as Int;;
```

An absolute value function:

```
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);;
```

Example evaluations:

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

>> abs z0;;
- : Int = <zero = true>

>> abs n5;;
- : Int = <pos = 5>
```

Equality, comparison and arithmetic on `Int`

First, we define some auxiliary functions over `Nat`.

Equality on `Nat`.

```
eq =
  letrec eq : Nat -> Nat -> Bool =
    lambda x : Nat. lambda y : Nat.
      if iszero x then iszero y
      else if iszero y then false
      else eq (pred x) (pred y)
  in
    eq
;;
```

Greater-or-equal on `Nat`.

```
ge =
  letrec ge : Nat -> Nat -> Bool =
    lambda a : Nat. lambda b : Nat.
      if iszero b then
        true
      else if iszero a then
        false
      else
        ge (pred a) (pred b)
  in
    ge
;;
```

Subtraction on `Nat`.

```
sub =
  letrec sub : Nat -> Nat -> Nat =
    lambda a : Nat. lambda b : Nat.
      if iszero b then
        a
      else
        sub (pred a) (pred b)
  in
    sub
;;
```

Addition on `Nat`.

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

Now we define integer addition on `Int`:

```
add =
  letrec sum : Nat -> Nat -> Nat =
    lambda n : Nat. lambda m : Nat.
      if iszero n then m
      else succ (sum (pred n) m)
  in
  letrec add : Int -> Int -> Int =
    lambda x : Int. lambda y : Int.
      (case x of
        <pos = p> =>
          (case y of
             (* Positive + Positive *)
             <pos = q> => (<pos = sum p q> as Int)
           (* Positive + Zero *)
           | <zero = z> => x
           (* Positive + Negative *)
           | <neg = q> =>
               if eq p q then (<zero = true> as Int)
               else if ge p q then (<pos = sub p q> as Int)
               else (<neg = sub q p> as Int))
      | <zero = z> =>
          (* Zero + anything *)
          y
      | <neg = p> =>
          (case y of
             (* Negative + Positive *)
             <pos = q> =>
               if eq p q then (<zero = true> as Int)
               else if ge p q then (<neg = sub p q> as Int)
               else (<pos = sub q p> as Int)
           (* Negative + Zero *)
           | <zero = z2> => x
           (* Negative + Negative *)
           | <neg = q> => (<neg = sum p q> as Int)))
  in add
```

```
;;
```

Informally, the behaviour is:

- **Positive + Positive** ⇒ add magnitudes: `sum p q`.
- **Positive + Zero** ⇒ return the positive number.
- **Positive + Negative** ⇒ subtract magnitudes, sign depends on comparison.
- **Zero + Anything** ⇒ return the other operand.
- **Negative + Positive** ⇒ symmetric to the previous case.
- **Negative + Zero** ⇒ return the negative number.
- **Negative + Negative** ⇒ add magnitudes and keep sign negative.

1.9 Lists

Lists are finite sequences of elements of the same type. The language provides:

- A type constructor `List[T]`.
- Constructors `nil[T]` and `cons[T]`.
- Selectors `head[T]`, `tail[T]` and a predicate `isnil[T]`.

Basic list construction

```
l1 = cons[Nat] 1 (cons[Nat] 2 (nil[Nat]));
l2 = cons[Nat] 3 (cons[Nat] 4 (nil[Nat]));
```

Length of a list

```
length =
  letrec length : List[Nat] -> Nat =
    lambda l : List[Nat].
      if isnil[Nat] l then 0
      else sum 1 (length (tail[Nat] l))
  in
    length
;;
```

Append two lists

```
append =
  letrec appen : List[Nat] -> List[Nat] -> List[Nat] =
    lambda xs : List[Nat]. lambda ys : List[Nat].
      if isnil[Nat] xs then ys
      else cons[Nat] (head[Nat] xs) (appen (tail[Nat] xs) ys)
  in appen
;;
```

Map a function over a list

```
map =
  letrec map : (Nat -> Nat) -> List[Nat] -> List[Nat] =
    lambda f : Nat -> Nat. lambda xs : List[Nat].
      if isnil[Nat] xs then nil[Nat]
      else cons[Nat] (f (head[Nat] xs)) (map f (tail[Nat] xs))
```

```
  in map
;;

double = lambda x : Nat. sum x x;;

map double l1;;
```

These examples implement typical list processing patterns without relying on OCaml's own lists, as required by the assignment.

2 Technical Manual

2.1 Project structure

The project is organised into several OCaml modules:

- **`main.ml`**: entry point. Implements the REPL, reads user input, passes it to the parser and invokes `execute`.
- **`lexer.mll`**: lexical analyser. Turns raw text into tokens (keywords, identifiers, symbols, string literals, etc.).
- **`parser.mly`**: parser. Defines the grammar and builds commands and terms (AST).
- **`lambda.ml` / `lambda.mli`**: defines the `command` type (`Eval`, `Bind`, `TypeBind`, `Quit`, `Clear`, ...) and the function `execute`, which orchestrates type checking and evaluation.
- **`types.ml`**: representation and pretty-printing of types.
- **`typing.ml`**: type system implementation, including `typeof` and `resolve_type`.
- **`terms.ml`**: abstract syntax for terms and basic syntactic operations (`isval`, substitution, free variables, pretty-printing).
- **`eval.ml`**: small-step evaluation (`eval1`) and multi-step evaluation (`eval`).
- **`context.ml`**: context representation and operations for variable, term and type bindings.

2.2 Global type aliases: `TyName` and `resolve_type`

Changes in `types.ml`

A new constructor is added to the type representation:

```
type ty =
  ...
  | TyName of string
```

`TyName s` represents a user-defined type alias name `s`, as introduced by a global type definition:

```
N = Nat;;
ListNat = List[Nat];;
```

Changes in `typing.ml`

A central function `resolve_type` is implemented. Its responsibilities are:

- Take a type that may contain `TyName` nodes.
- Look up each name in the context using `getTypeBinding`.
- Recursively expand each alias to its corresponding concrete type.
- Return a fully resolved type, free of `TyName`, before any comparison.

The function `typeof` calls `resolve_type` in all places where types are compared, stored in the context, or appear in function abstractions, applications, let-bindings, tuples, and concatenation.

As a consequence, two types that are syntactically different but equal modulo aliases are treated as equal. For example, `N -> N` and `Nat -> Nat` are equivalent once resolution is applied.

Changes in `lambda.ml`

When printing types in the REPL, `resolve_type` is applied so that the user sees resolved, canonical types where appropriate. This ensures coherence between the type checker and the printed output.

2.3 Global term definitions and functional context

Term and type aliases are stored in a global context. The chosen design is *purely functional*: the context is an immutable structure, and every new binding produces a fresh context. Lookup always returns the most recent binding, effectively implementing shadowing.

In more detail:

- `context.ml` represents the context as a list of bindings.
- Operations such as `addTermBinding` and `addTypeBinding` extend the list by *prefixing* new bindings.
- Lookup functions traverse the list from the front, ensuring that newer bindings shadow older ones.

> We chose a purely functional context model: every new binding produces a fresh context by prefixing the new association, and lookup always returns the most recent binding (shadowing). This approach avoids mutation, preserves referential transparency, and matches the formal treatment of contexts in typed λ-calculus. It also ensures coherence between evaluation and typing, and enables the implementation of global term and type aliases in a clean, mathematically sound way.

2.4 Fixed-point combinator and `letrec`

Recursive definitions are supported at the surface language level via `letrec`. Syntactically, `letrec` introduces a binding where the defined name is in scope inside its own right-hand side.

Parser and lexer

- `lexer.mll` introduces a `LETREC` token.
- `parser.mly` defines the grammar rule for `letrec`, producing an AST node (e.g. `TmLetRec`) that carries the function name, its type, its argument, and body.

Typing

In the implementation, the context is extended with `f : A -> B` before checking the body of `f`, and the type of the whole expression is the type of the body of the `in`-part.

Evaluation

Operationally, `letrec` is translated to a suitable fixed-point construction, or directly implemented using an environment-based semantics. The interpreter ensures that the recursive function can call itself during evaluation, as witnessed by the examples `sum`, `prod`, `fib` and `fact` presented in the user manual.

2.5 Strings and concatenation

Changes in `terms.ml`

New term constructors are introduced:

```
type term =
  ...
  | TmString of string
  | TmConcat of term * term
```

The following functions are updated:

- `isval`: a `TmString` is a value.
- `string_of_term`: prints string literals and concatenations.
- `free_vars`, `subst`: traverse the new constructors.

Changes in `types.ml` and `typing.ml`

A new base type `TyString` is added, and `typeof` assigns type `String` to `TmString` and checks that both operands of `TmConcat` have type `String`.

Changes in `eval.ml`

`eval1` includes rules for concatenation, evaluating left and right arguments and finally concatenating their underlying OCaml strings when both are values.

Changes in `lexer.mll` and `parser.mly`

- `lexer.mll`: a rule is added to recognise string literals (between quotes) and a token for the concatenation operator `^`.
- `parser.mly`: introduces tokens `STRING` and `CONCAT` and corresponding grammar rules to build `TmString` and `TmConcat`.

2.6 Tuples and records

Tuples (`TyTupl`, `TmTupl`, `TmProj`)

Types. A new type constructor:

```
| TyTupl of ty list
```

represents tuples of arbitrary arity.

Terms. New term constructors:

```
| TmTupl of term list
| TmProj of term * int
```

represent tuple construction and positional projection, respectively.

Typing and evaluation. The type checker ensures that:

- A tuple `{t1, ..., tn}` has type `{T1, ..., Tn}` if each component `ti` has type `Ti`.
- A projection `t.i` is well-typed only when `t` has a tuple type of sufficient arity; the result type is the `i`-th component.

`eval1` evaluates tuples component-wise and reduces a projection once its argument is a value tuple.

Lexer and parser. Tokens for curly braces and commas are added (`LCURLYB`, `RCURLYB`, `COMMA`), and the grammar rules for `atomicTerm` and `appterm` are extended to support tuple syntax and projection.

Records (`TyRecord`, `TmRecord`, `TmProjRecord`)

Records are similar to tuples but with labels:

```
type ty =
  ...
  | TyRecord of (string * ty) list

type term =
  ...
  | TmRecord of (string * term) list
  | TmProjRecord of term * string
```

The type checker verifies that the labels and types in a record literal match the declared record type, and that projections by label are well-typed.

In `eval.ml`, the evaluation strategy for records mirrors that of tuples: fields are evaluated from left to right, and projections are reduced once the record is a value.

Grammar rules for `atomicTerm` and `appterm` are updated to distinguish tuple and record syntax and to support projections by label.

2.7 Variants and lists

Variants

- A type constructor for variants (e.g. `TyVariant of (string * ty) list`).
- Term constructors for injection into a variant and for case analysis.
- Typing rules that ensure exhaustiveness and consistent types across branches.
- Evaluation rules that reduce the scrutinee and then select the appropriate branch.

The example `Int` and the functions `abs` and `add` serve both as tests and as documentation of the behaviour.

Lists

Lists are implemented as an algebraic data type at the object-language level, with explicit type parameter `T`. The type constructor `List[T]` is supported by the type system, and the operations:

- `nil[T]`
- `cons[T]`
- `isnil[T]`
- `head[T]`
- `tail[T]`

are given appropriate typing and evaluation rules. The examples `length`, `append` and `map` show how to build typical recursive list-processing functions without using OCaml lists.