DLP

LAMBDA CALCULUS INTERPRETER

Diego Suárez Ramos (diego.suarez.ramos@udc.es)
Pablo Fernández Pérez (pablo.fperez@udc.es)

December 7, 2023

Contents

1 USER MANUAL

(All the examples provided in this manual are also available in the file examples.txt)

1.1 MULTI-LINE EXPRESSIONS

Now, an expression can be introduced across several lines. As the newline alone no longer necessarily implies the end of an expression, a new symbol has been introduced to address this aspect—the semicolon.

Example:

```
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
;;
```

1.2 PRETTY-PRINTER

The number of parentheses required when an expression is transformed into a string has been reduced. Additionally, some line breaks and indentation have been added to enhance understanding.

Example:

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

1.3 INTERNAL FIXED-POINT COMBINATOR

Now functions can be declared through direct recursive definitions.

Examples:

```
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 21 34
;;
```

PROD:

```
letrec sum : Nat -> Nat -> Nat =
lambda n: Nat. lambda m : Nat.
if iszero n then
m
else succ (sum (pred n) m)
in
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 12 5;;
```

FIBONACCI:

```
letrec sum : Nat -> Nat -> Nat =
lambda n : Nat. lambda m : Nat.
if iszero n then
m
else
succ (sum (pred n) m)
in
letrec fib: Nat -  Nat =
lambda n : Nat.
if iszero n then
0
else
if iszero (pred n) then
1
else
sum(fib (pred (pred n))) (fib (pred n))
in fib 5;;
```

FACTORIAL:

```
letrec sum : Nat -> Nat -> Nat =
lambda n : Nat. lambda m : Nat.
if iszero n then
m
else
succ (sum (pred n) m)
in
letrec prod : Nat -> Nat -> Nat =
lambda n : Nat. lambda m : Nat.
if iszero n then
0
else
sum (prod (pred n) m) m
in
letrec fact: Nat -> Nat =
lambda n : Nat.
if iszero n then
1
else
prod n (fact (pred n))
in fact 5;;
```

1.4 GLOBAL DEFINITIONS CONTEXT

Now it is allowed to associate names of free variables with values or terms, so that these can be used in subsequent lambda expressions.

Examples:

```
x = 5;;

succ x;;

f = lambda y : Nat.x;;

f 3;;

x = 7;;

f 4;;

x;;
```

```
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
;;

sum x 3;;

N = Nat;;

N2 = N -> N;;

N3 = N -> N2;;
```

1.5 STRING TYPE

The String type has been incorporated for support of character strings, along with string concatenation, length, and comparison operations.

- concat: Concatenates two strings.
- length: Returns the length of a string.
- compare: Compares two strings, returning 0 if they have the same length, 1 if the first is shorter, and 2 if the first is shorter.

Examples:

```
””;;

”abc”;;

concat ”para” ”sol”;;

concat (concat ”para” ”sol”) ”es”;

lambda s : String. s;;

(lambda s : String.s) ”abc”;;

letrec replicate : String -> Nat -> String =
lambda s : String. lambda n : Nat.
if iszero n then ”” else concat s (replicate s (pred n))
in
replicate ”abc” 3
;;

concat ((lambda s : String.s) ”abc”) ”de”;;
```

```
let s = letrec replicate : String -> Nat -> String =
lambda s : String. lambda n : Nat.
if iszero n then "" else concat s (replicate s (pred n))
in
replicate "abc" 3
in concat s s
;;

length "12345";;

compare "hola" "hola";;

compare "hola1" "hola";;

compare "hola" "hola1";;
```

1.6 TUPLES

Tuples of any number of elements have been introduced, along with projection operations based on the position of these elements.

Examples:

```
{5, true};;

{5, true, "abc"};;

{5, true, "abc"}.1;;

{5, true, "abc"}.2;;

{5, true, "abc"}.3;;

{5, true, "abc"}.4;;

{5, {true, "abc"}};;

{5, {true, "abc"}}.1;;

{5, {true, "abc"}}.2;;

{5, {true, "abc"}}.2.1;;

{5, {true, "abc"}}.2.2;;

t = {5, {true, "abc"}};;
```

```
t.1;;

t.2;;

t.2.1;;

t.2.2;;
```

1.7 RECORDS

Records (finite sequences of fields of any labeled type) have been incorporated, along with typical projection operations based on the labels of the fields.

Examples:

```
{x=2, y=5, z=0};;

{x=2, y=5, z=0}.x;;

{x=2, y=5, z=0}.y;;

{x=2, y=5, z=0}.z;;

{x=2, y=5, z=0}.1;;

{x=2, y=5, z=0}.h;;

p = {na={”luis”, ”vidal”}, e=28};;

p.na;;

p.na.1;;

p.na.2;;

p.e;;
```

1.8 VARIANTS

Variants have been introduced, which constitute a labeled generalization of binary sum types.

Examples:

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

p3 = <pos=3> as Int;;
```

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

n5 = <neg=5> as Int;;

is_zero = L i : Int.
case i of
<pos=p> => false
| <zero=z> => true
| <neg=n> => false
;;

is_zero p3;;

is_zero z0;;

is_zero n5;;

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);;

abs p3;;

abs z0;;

abs n5;;
```

FUNCIÓN ADD

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

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

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

let rec add : Int -> Int -> Int =
lambda x : Int. lambda y : Int.
case x of
<pos=p1> =>
case y of
<pos=p2> => <pos=(sum p1 p2)> as Int
| <zero=z2> => <pos=p1> as Int
| <neg=n2> =>
if (is_greater p1 n2) then <pos=(sub p1 n2)> as Int
else <neg=(sub p1 n2)> as Int
| <zero=z1> =>
case y of
<pos=p2> => <pos=p2> as Int
| <zero=z2> => <zero=true> as Int
| <neg=n2> => <neg=n2> as Int
| <neg=n1> =>
case y of
<pos=p2> =>
if (is_greater p2 n1) then <pos=(sub p2 n1)> as Int
else <neg=(sub p2 n1)> as Int
| <zero=z2> => <neg=n1> as Int
| <neg=n2> => <neg=(sum n1 n2)> as Int
;;
```

1.9 LISTS

Lists (finite sequences of elements of the same type) have been incorporated, along with typical operations to obtain the head, obtain the tail, and check if it is empty.

Examples:

```
nil[Nat];;

cons[Nat] 3 nil[Nat];;

cons[Nat] 5 (cons[Nat] 3 nil[Nat]);;

l = cons[Nat] 8 (cons[Nat] 5 (cons[Nat] 7 nil[Nat]));;
```

```
isnil[Nat] l;;

head[Nat] l;;

tail[Nat] l;;

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
;;

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
;;

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

len l;;

N3 = Nat -> Nat -> Nat;;

nil[N3];;

cons[N3] sum nil[N3];;

cons[N3] prod (cons[N3] sum nil[N3]);;

len (cons[N3] prod (cons[N3] sum nil[N3]));;

(head[N3] (cons[N3] prod (cons[N3] sum nil[N3]))) 12 5;;

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

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

```
map l f;;

append = letrec append: List[Nat] -> List[Nat] -> List[Nat] =
lambda l1: List[Nat]. lambda l2: List[Nat].
if isnil[Nat] l1 then
l2
else
cons[Nat] (head[Nat] l1) (append (tail[Nat] l1) l2)
in append;;

l2 = (cons[Nat] 5 (cons[Nat] 3 nil[Nat]));;

append l l2;;
```

1.10 SUBTYPING

Subtyping has been incorporated.

Examples:

```
{};;
```

In records:

```
idr = lambda r : {}. r;;

idr {x=0, y=1};;

(lambda r : {x : Nat}. r.x) {x=0, y=1};;

a = {x=1, y=1, z={x=1}};;

b = {x=1, y=1, z=a};;

(lambda r : {z : {x : Nat}}. (r.z).x) b;;

(lambda r : {z : {x : Nat}}. (r.z).x) a;;
```

In lists:

```
l1 = (cons[{x : Nat}] x=2, y=3 nil[{x : Nat}]);;

idr2 = lambda l : List[{x : Nat}]. l;;

l2 = (cons[{x : Nat, y : Nat}] {x=2, y=3} nil[{x : Nat, y : Nat}]);;

idr2 l2;;
```

In variants:

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

idr3 = lambda v : Int. v;;

idr3 (<pos = 6> as Int);;

2 TECHNICAL MANUAL

2.1 MULTI-LINE EXPRESSIONS

To enable multi-line expressions, we have modified main.ml to add a function read_command() that continuously executes read_line() until it detects a ';'. At that point, it returns all the read lines concatenated. This function is subsequently used in the main function of the main.ml file.

2.2 PRETTY-PRINTER

New functions have been created: pretty_print and pretty_print_ty, inspired by the previous ones. In these new functions, only the strictly necessary parentheses are included.

2.3 INTERNAL FIXED-POINT COMBINATOR

In lexer.mll, when a letrec is encountered, a LETREC token is passed to the parser, and when a fix is encountered, a FIX token is passed. In the parser, we define rules that allow recognizing the structure of a letrec or fix, triggering corresponding actions to return the associated term. In the case of letrec, a term TmLetIn is returned, with one of its arguments being a new term, TmFix. In lambda.ml, eval1 has been modified to add new cases to handle the new term, TmFix. Similarly, the typeof, subst, and free_vars functions have also been modified to include the new type.

2.4 GLOBAL DEFINITIONS CONTEXT

Firstly, in lambda.mli and lambda.ml, we added the command type, which is used to represent the actions that the lambda interpreter can perform. Specifically, commands for evaluation (Eval) can be issued to evaluate a term in the current context, or assignment commands (Bind) can be used to bind a name to a term. The execute function has also been added to differentiate between these commands and execute the corresponding action. Now we have a context for definitions and another for types, so the types of functions in lambda.mli have been modified accordingly. Additionally, in lambda.ml, the apply_ctx function has been added to substitute the name of the definition with its value. The eval and eval1 functions have also been modified to include the context and a case for TmVar to retrieve the value associated with the definition. main.ml has been modified to include the new context. In the parser, the grammar axiom now distinguishes between whether it is an assignment or simply a term for proper analysis.

2.5 STRING TYPE

Incorporated a new type named TyString and introduced a corresponding term, TmString. To seamlessly integrate the String type, essential pattern matching adjustments were made to functions in the lambda.ml file. To ensure recognition of the String type by the lexer.mll and parser.mly, new rules were added along with their appropriate definitions. Furthermore, three new functions—concat, length, and compare—have been implemented. For these functions, corresponding terms TmConcat, TmLength, and TmCompare were created. Consequently, lambda.ml was modified to accommodate these changes. In eval1, the following implementations were added:
- concat: Concatenates two strings.
- length: Returns the length of a string.
- compare: Compares two strings, returning 0 if they have the same length, 1 if the first is shorter, and 2 if the first is shorter.

2.6 TUPLES

Initially, in lambda.ml, we established the TyTuple type of ty list along with its corresponding term, TmTuple of term list. Subsequently, corresponding cases were added in various functions. As outlined in the parser.mly, we devised rules to discern whether a tuple comprises a singular element or multiple elements. These elements have the flexibility to be either values or terms. To facilitate the extraction of tuple elements, we introduced TmProjection, a construct designed to compute the projection of a specified position within the tuple. It's imperative to note that the position must be a numeric value.

2.7 RECORDS

We've specified the type as TyRecord, consisting of a list of pairs (string * ty), and the corresponding term as TmRecord, composed of a list of pairs (string * term). Records, in this context, are essentially tuples where each element is associated with a label. The lambda.ml file and the parser.mly file have been updated to reflect these definitions, and rules have been established in parser.mly to recognize both empty records and records with one or more elements. Similar to tuples, these elements within records can manifest as either values or terms. To access the elements within records, we utilized the previously defined TmProjection. However, a notable distinction exists—elements in records can only be projected by their respective labels. Furthermore, we've introduced a modification in the implementation of appTerm. This involves replacing the call to atomicTerm with a new rule named pathTerm in order to streamline projections within records and tuples.

2.8 VARIANTS

In the lambda.ml file, we added a term TmVariant and a type tyVariant. In the execute function, we added two new cases: one for having global types variables and another for type ascriptions. To do the latter, we also added the type tyName of string. We also added TmCase for when a case of statement is used. Corresponding cases were added in various functions. In the lexer, we added the necessary tokens, and in the parser, we included rules for these cases in the grammar. In the typeof function, we added auxiliary functions to convert tyName.

2.9 LISTS

Firstly, we defined the TyList type for ty, and its term has been decomposed into TmNil of ty, TmCons of ty * term * term, TmIsNil of ty * term, TmHead of ty * term, and TmTail of ty * term. We have implemented the necessary functionalities in the lambda.ml file to recognize the head and tail of the list, as well as to check if the list is empty or not. In the lexer.mll file, we have introduced a token for each list operation. In the parser.mly, we have added these tokens in the rules to identify each of them.

2.10 SUBTYPING

In the lambda.ml file, we have introduced an auxiliary function called subtypeof. This function is designed to redefine the implementation of the typeof function by assessing whether two given types adhere to the subtyping relation. In this function, subtyping rules are added, including rules for lists, records, and variants. Furthermore, necessary modifications have been made in typeof to incorporate subtyping, so that where it was previously required for two types to be equal, it is now required that one be a subtype of the other. This impacts TmApp, TmFix, and TmCons.