Interpreter of lambda calculus

Authors

Juan Villaverde

Carlos Rodriguez rojo - carlos.rojo@udc.es

User manual

1.1 Implementation of multi-line expression recognition.

The interpreter recognizes expressions until it encounters two consecutive semicolons '**;;**'.

2.1 Incorporation of an internal fixed-point combinator.

Recursive functions are possible using the keyword **letrec**, so insted of writing:

```
let fix = lambda f.(lambda x. f (lambda y. x x y)) (lambda x. f (lambda y. x x y)) in
let sumaux =
lambda f. (lambda n. (lambda m. if (iszero n) then m else succ (f (pred n) m))) in
let sum = fix sumaux in
sum 21 34
```

You can write:

```
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
```

2.2 Addition of a global definition context

Allows associating variable names with values or terms as well as creating type aliases:

```
identificador = termino
x = false;;
N = Nat;;
lambda x: N. x;;
```

2.3 Addition of the String type

Character strings can be formalized using double quotes as follows:

```
"srt";;
"Hola Mundo";;
```

You can also use the keyword **concat** to realize the concatenation of two strings:

```
concat "hola " "mundo";;
```

2.4 Addition of the tuple type

Using brakets, the tuple type can be defined as **{}**, representing the empty tuple. Its elemnts can also be projected using a dot and an integer starting from 0:

```
{1,"hola mundo", if true then false else true} : {Nat, String, Bool}
{3,4}.1 --> devolveria 4
```

2.5 Addition of the record type

To define a record we use brackets as in tuples but with the help of a unique label associated to each term inside the record:

```
{hola=2, mundo="srt",adios=true} : {hola:Nat, mundo:String, adios:Bool};;
```

We use its label for projection:

```
{hola=2, mundo="srt"}.hola ---> 2
```

The empty record is not considered, **{}** still represents an empty tuple.

2.7 Addition of variants

2.8 Addition of lists

Manual tecnico

Cambios realizados en los modulos con cada incorporacion:

1.1 multi-line expression recognition

Module: main.ml

line:8

```
let read_command () =
  let rec read acc =
    try
      let line = read_line () in
      if String.ends_with ~suffix:";;" line
        then String.concat " " (List.rev (String.sub line 0 (String.length line - 2)::acc))
        else read(line::acc)
    with End_of_file ->
      String.concat " " (List.rev acc)
    in read []
```

line:39

```
let tm = s token (from_string (read_command ())) in
```

2.1 Incorporation of an internal fixed-point combinator

Module:lambda.mli

line:25

```
| TmFix of term
```

Module:lambda.ml

line:27

```
| TmFix of term
```

line:129

```
| TmFix t1 ->
          let tyT1 = typeof ctx t1 in
          (match tyT1 with
              TyArr (tyT11, tyT12) ->
                  if tyT11 = tyT12 then tyT12
                  else raise (Type_error "result of body not compatible with domain")
          |  _ -> raise (Type_error "arrow type expected"))
```

line:179

```
| TmFix t ->
      "(fix " ^ string_of_term t ^ ")"
```

line:220

```
| TmFix t ->
      free_vars t
```

line:265

```
| TmFix t ->
      TmFix (subst x s t)
```

line:360

```
| TmFix (TmAbs (x, _, t2)) ->
      subst x tm t2
```

line:364

```
| TmFix t1 ->
    let t1' = eval1 t1 in
    TmFix t1'
```

Module lexer.mll:

line:20

```
| "letrec"   { LETREC }
```

Module parser.mly:

line:16

```
%token LETREC
```

line:53

```
| LETREC IDV COLON ty EQ term IN term
      { TmLetIn ($2, TmFix (TmAbs($2, $4, $6)), $8) }
```

2.2 Addition of a global definition context

Module lexer.mll

line:33 Addition of 'A'-'Z'

```
| ['a'-'z''A'-'Z']['a'-'z' '_' '0'-'9']*
```

Module parser.mly

line:59 Addition of

```
tyTerms :
    atomicTyTerms
      { $1 }
  | atomicTyTerms ARROW tyTerms
      { TmTyArr ($1, $3) }

atomicTyTerms :
    LPAREN tyTerms RPAREN
      { $2 }
  | BOOL
      { TmTyBool }
  | NAT
      { TmTyNat }
  | STRING
      { TmTyString }
  | IDV
      { TmVar $1 }
```

line:90

```
  | tyTerms
    { $1 }
```

line:125

```
| IDV
      { TyVar $1 }
```

Module lambda.mli

line:7

```
  | TyVar of string
```

line:30

```
  | TmDef of string * term
  | TmTyBool
  | TmTyNat
  | TmTyArr of term * term
  | TmTyString
```

line:40

```
type contextTerm =
  (string * term) list
;;
```

line: 48

```
val emptyctxTerms : contextTerm;;
val addbindingTerms : contextTerm -> string -> term -> contextTerm;;
val getbindingTerms : contextTerm -> string -> term;;
```

Module lambda.ml

Same headers as in lambda.mli

Also...

line:47

```
let emptyctxTerms =
  []
;;

let addbindingTerms ctx x bind =
  (x, bind):: ctx
;;

let getbindingTerms ctx x =
  List.assoc x ctx
;;
```

line:82

```
| TyVar t -> t
```

line:122

```
| TmDef (t1, t2) ->
      t1 ^ " = " ^ string_of_term t2
  | TmTyArr (t1,t2) ->
      string_of_term t1 ^ "->" ^ string_of_term t2
  | TmTyBool ->
    "Bool"
  | TmTyNat ->
    "Nat"
  | TmTyString ->
    "String"
```

Global term context added to typeof header

line:186

```
| TmAbs (x, tyT1, t2) ->
let typesCtx' = addbinding typesCtx x tyT1 in
      let termsCtx' = addbindingTerms termsCtx x t2 in
      let tyT2 = typeof typesCtx' termsCtx' t2 in
      let tyT1' =
      (match tyT1 with
        | TyVar t -> typeof typesCtx' termsCtx' ((getbindingTerms termsCtx (string_of_ty(tyT1))) )
        | _ -> tyT1) in
      TyArr (tyT1', tyT2)
```

line:231

```
| TmDef (x, t1) ->
      let tyT1 = typeof typesCtx termsCtx t1 in
      let ctx' = addbinding typesCtx x tyT1 in
      let termsCtx' = addbindingTerms termsCtx x t1 in
      typeof ctx' termsCtx' t1

  | TmTyArr (t1,t2) ->
    let tyT1 = typeof typesCtx termsCtx t1 in
    let tyT2 = typeof typesCtx termsCtx t2 in
    TyArr (tyT1,tyT2)

  | TmTyBool ->
    TyBool

  | TmTyNat -> TyNat

  | TmTyString -> TyString
```

line:291

```
| _ ->  []
```

line 339

```
| _ -> tm
```

line:362

```
let esAbstraccion termsCtx = function
  | TmAbs (_,_,_) -> true
  | _ -> false
let devolverAbstraccion termsCtx typesCtx (TmAbs(y,ty,t12)) =
  match (ty,t12) with
  | (TyVar t1, TmVar t) -> TmAbs (y, (getbinding typesCtx (string_of_ty(ty))), (getbindingTerms termsCtx (string_of_term(t12))))
  | (_, TmVar t) -> TmAbs (y, ty, (getbindingTerms termsCtx (string_of_term(t12))))
  | (TyVar t1, _) -> TmAbs (y, (getbinding typesCtx (string_of_ty(ty))), t12)
  | (_,_) -> (TmAbs(y,ty,t12))

let esArrowType termsCtx = function
  | TyArr _ -> true
  | _ -> false
```

line:503

```
| TmDef (x, t1) when isval t1->
    print_endline("PASA POR AQUI 11");
    print_endline(string_of_term(t1));
    if esAbstraccion termsCtx t1 then devolverAbstraccion termsCtx typesCtx t1 else t1

  | TmDef (x, t1) ->
    print_endline("PASA POR AQUI 10");
    let t1' = eval1 termsCtx typesCtx t1 in TmDef (x, t1')
```

Addition of termsCtx typesCtx to eval1 and eval headers

Module main.ml

Inside main loop:

```
let tm = s token (from_string (read_command ())) in

      if esDefinicion tm
        then  let tyTm = typeof typesCtx termsCtx tm in

              let nombreVar = String.split_on_char ' ' (string_of_term(tm)) in
              if comienza_con_mayuscula (List.nth nombreVar 0)
                then print_endline("type " ^ (List.nth nombreVar 0) ^ " = " ^ string_of_ty tyTm)
                else print_endline((List.nth nombreVar 0) ^ " : " ^ string_of_ty tyTm ^ " = " ^ string_of_term (eval termsCtx typesC


              loop (addbinding typesCtx (List.nth nombreVar 0) tyTm) (addbindingTerms termsCtx (List.nth nombreVar 0) (eval termsCtx
        else  let tyTm = typeof typesCtx termsCtx tm in
              let nombreVar = String.split_on_char ' ' (string_of_term(tm)) in
              if comienza_con_mayuscula (List.nth nombreVar 0)
                then print_endline("type " ^ (List.nth nombreVar 0) ^ " = " ^ string_of_ty tyTm)
                else print_endline("-: " ^ string_of_ty tyTm ^ " = " ^ string_of_term (eval termsCtx typesCtx tm));

              loop typesCtx termsCtx
```

```
let esDefinicion = function
  | TmDef (_,_) -> true
  | _ -> false


let comienza_con_mayuscula (cadena : string) : bool =
  let patron = Str.regexp "^[A-Z]" in
  try
    ignore (Str.search_forward patron cadena 0);
    true
  with Not_found -> false
```

```
| Not_found ->
        print_endline "Otro error";
        loop typesCtx termsCtx
```

Addition of emptyCtxTerms to main header

Makefile

Added str.cma in line 3: ocamlc str.cma -o top lambda.cmo parser.cmo lexer.cmo main.cmo

2.3 Addition of the string type.

Module lambda.mli:

line:6

```
| TyString
```

line:26

```
| TmString of string
| TmConcat of term * term
```

Module lambda.ml

line:8

```
| TyString
```

line:28

```
  | TmString of string
  | TmConcat of term * term
```

line:57

```
| TyString ->
      "String"
```

line:138

```
  | TmString _->
      TyString
```

line:142

```
  | TmConcat (t1, t2) ->
      if typeof ctx t1 = TyString && typeof ctx t2 = TyString then TyString
      else raise (Type_error "argument of concat is not a string")
```

line:181

```
| TmString s ->
    "\"" ^ s ^ "\""
```

line:183

```
| TmConcat (t1, t2) ->
    "concat " ^ "(" ^ string_of_term t1 ^ ")" ^ " " ^ "(" ^ string_of_term t2 ^ ")"
```

line:222

```
  | TmString _ ->
   []
  | TmConcat (t1, t2) ->
   lunion (free_vars t1) (free_vars t2)
```

line:267

```
  | TmString st ->
    TmString st
  | TmConcat (t1, t2) ->
    TmConcat (subst x s t1, subst x s t2)
```

line:283

```
| TmString _ -> true
```

line:369

```
  | TmConcat (TmString s1, TmString s2) ->
    TmString (s1 ^ s2)
```

line:373

```
  | TmConcat (TmString s1, t2) ->
    let t2' = eval1 t2 in
    TmConcat (TmString s1, t2')
```

line:378

```
  | TmConcat (t1, s2) ->
    let t1' = eval1 t1 in
    TmConcat (t1', s2)
```

Module lexer.mll:

line:25

```
  | "String"    { STRING }
```

line:35

```
  | '"'[^ '"' ';' '\n']* '"'
              { let s = Lexing.lexeme lexbuf in
                STRINGV (String.sub s 1 (String.length s - 2))}
```

Module parser.mly:

line:21

```
%token STRING
```

line:33

```
%token <string> STRINGV
```

line:65

```
  | CONCAT atomicTerm atomicTerm
     { TmConcat ($2, $3) }
```

line:84

```
  | STRINGV
     { TmString $1}
```

line:100

```
  | STRING
     { TyString }
```

2.4 Addition of the tuple type.

Module: lexer.mll

line 42

```
  | '{'        { LCORCH }
 | '}'        { RCORCH }
 | ','        { COMA }
```

Module: parser.mly

line 29

```
%token LCORCH
%token RCORCH
%token COMA
```

```
    | LCORCH algo RCORCH
        { $2 }
```

```
algo:
  | term COMA tupla
    { TmTuple ([$1] @ $3)}
  | term
    { TmTuple [$1]}
  | /*Tupla vacia*/
    { TmTuple []}
```

```
        | term DOT INTV
            { TmTProj ($1, $3)}
```

Module: lambda.mli

line 2:

```
  | TyTuple of ty list
```

line 36

```
  | TmTuple of term list
 | TmTProj of term * int
```

Module: lambda.ml

line 10:

```
  | TyTuple of ty list
```

line 38

```
  | TmTuple of term list
 | TmTProj of term * int
```

line 97

```
| TyTuple l ->
      let rec aux str=function
        | [] -> "{" ^ str ^ "}"
        | [h] -> aux (str ^ string_of_ty h) []
        | h::t -> aux (str ^ string_of_ty h ^ ", ") t
      in aux "" l
```

line 164

```
| TmTuple l ->
    let rec aux str=function
      | [] -> "{" ^ str ^ "}"
      | [h] -> aux (str ^ string_of_term h) []
      | h::t -> aux (str ^ string_of_term h ^ ", ") t
    in aux "" l
  | TmTProj (t, idx) ->
      (match t with
        TmTuple l -> string_of_term (List.nth l idx)
        | _ -> string_of_term t)
```

line 349

```
| TmTuple t1 ->
    let rec axu res= function
      | [] -> TyTuple (List.rev res)
      | h::t -> axu (typeof typesCtx termsCtx h::res) t
    in axu [] t1
| TmTProj (t, idx) ->
      (match t with
        | TmTuple l -> typeof typesCtx termsCtx (List.nth l idx)
        | TmVar y -> (try
                      match (getbinding typesCtx y) with
                        TyTuple l -> List.nth l idx
                        | _ -> raise (Type_error "Incompatible types")
                    with
                      _ -> raise (Type_error ("no binding type for variable " ^ y)))
        | _ -> raise (Type_error "Projecting from not project type"))
```

line 470

```
| TmTuple (t1) ->
    let rec aux res = function
      | [] -> res
      | h::t -> aux (lunion (free_vars h) res) t
    in aux [] t1
  | TmTProj (t1, idx) ->
    (match t1 with
      | TmTuple l -> free_vars (List.nth l idx)
      | _ -> free_vars t1)
```

line 558

```
| TmTuple t ->
     TmTuple t
| TmTProj (t, id) ->
   (match t with
     TmVar y -> if y = x
                then TmTProj(s, id)
                else failwith "Let in without TmVar"
     | _ -> failwith "Didn't match TmVar")
```

line 600

```
| TmTuple l -> let rec axu = function
                  | [] -> true
                  | h::t -> if isval h
                              then axu t
                              else false
                in axu l
```

line 770

```
| TmTProj (t1, idx) ->
   let t1' = eval1 termsCtx typesCtx t1 in
   TmTProj(t1', idx)
   (* E-Tuple *)
  | TmTuple (t1) when not (isval (TmTuple t1))->
   let rec axu res = function
     | [] -> TmTuple (List.rev res)
     | h::t -> if isval h
                then axu (h::res) t
                else axu (eval1 termsCtx typesCtx h::res) t
   in axu [] t1
```

2.5 Addition of the record type.

Module parser.mly

line 78:

```
| term DOT IDV
          { TmRProj ($1, $3)}
```

line 91:

```
| IDV EQ term reg
   { TmReg ([($1,$3)] @ $4) }
```

line 100:

```
reg:
  | COMA IDV EQ term reg
   { [($2,$4)] @ $5 }
  | /**/
   { [] }
```

Module lambda.mli

line 9:

```
| TyReg of (string * ty) list
```

line 38:

```
  | TmReg of (string * term) list
 | TmRProj of term * string
```

Module lambda.ml

line 11:

```
  | TyReg of (string * ty) list
```

line 38:

```
| TmTProj of term * int
| TmReg of (string * term) lis
```

line 103:

```
| TyReg l ->
   let rec aux srt = function
     | [] -> "{" ^ srt ^ "}"
     | [(key, t)] -> aux (srt ^ key ^ ":" ^ string_of_ty t) []
     | (key, t1)::t -> aux (srt ^ key ^ ":" ^ string_of_ty t1 ^ ", ") t
   in aux "" l
```

line 174:

```
| TmReg l ->
     let rec aux srt = function
       | [] -> "{" ^ srt ^ "}"
       | [(key, t1)] -> aux (srt ^ key ^ "=" ^ string_of_term t1) []
       | (key, t1)::t -> aux (srt ^ key ^ "=" ^ string_of_term t1 ^ ", ") t
     in aux "" l
  | TmRProj (t, et) ->
     (match t with
       | TmReg l -> string_of_term (List.assoc et l)
       | _ -> string_of_term t)
```

line 368:

```
| TmReg l ->
   let rec aux res = function
     | [] -> TyReg (List.rev res)
     | (key, t1)::t -> let t1' = typeof typesCtx termsCtx t1
                       in aux ((key, t1')::res) t
   in aux [] l

   (* T-Record-Proj *)
  | TmRProj (t, et) ->
     (match t with
       | TmReg l -> typeof  typesCtx termsCtx (List.assoc et l)
       | TmVar y -> (try
                       match (getbinding typesCtx y) with
                         TyReg l -> List.assoc et l
                         | _ -> raise (Type_error "incompatible types")
                     with
                       _ -> raise (Type_error ("no binding type for variable " ^ y)))
       | _ -> raise (Type_error "Projecting from not project type"))
```

line 479

```
| TmReg l ->
    let rec aux res = function
      | [] -> res
      | (key, t1)::t -> aux (lunion (free_vars t1) res) t
    in aux [] l
  | TmRProj (t, key) ->
    (match t with
      | TmReg l -> free_vars (List.assoc key l)
      | _ -> free_vars t)
```

line 566:

```
| TmReg l ->
      TmReg l
  | TmRProj (t, key) ->
      (match t with
        TmVar y -> if y = x
                      then TmRProj(s, key)
                      else failwith "Let in without TmVar"
        | _ -> failwith "Didn't match TmVar")
```

line 606:

```
| TmReg l -> let rec axu = function
                 | [] -> true
                 | (key,t1)::t -> if isval t1
                                   then axu t
                                   else false
               in axu l
```

line 782

```
| TmRProj (TmReg l, key) when isval (TmReg l)->
      List.assoc key l

  | TmRProj (t1, key) ->
      let t1' = eval1 termsCtx typesCtx t1 in
      TmRProj (t1', key)

  | TmReg l when not (isval (TmReg l)) ->
      let rec aux res= function
        | [] -> TmReg (List.rev res)
        | (key, t1)::t -> if isval t1
                          then aux ((key,t1)::res) t
                          else let t1' = eval1 termsCtx typesCtx t1
                            in aux ((key,t1')::res) t
      in aux [] l
```