Práctica 2

INTÉRPRETE DE LAMBDA CÁLCULO

Juan Luis Filgueiras Rilo
Andrés Molares Ulloa
|
Diseño de Lenguajes de Programación – Mención en Computación – UDC
|
Curso 2018-2019

Contenido

Ejercicio 1

FUNCIONALIDADES SOLICITADAS

En este ejercicio se solicita una modificación del código proporcionado con el fin de detectar los errores de tipo que involucren a los tipos Nat y Bool. Para ello será necesario la incorporación al intérprete de los tipos Nat y Bool junto a los mecanismos necesarios para su correcta comprobación.

CAMBIOS REALIZADOS

Para ello, introducimos en el archivo syntax.ml (y en su correspondiente interfaz de módulo, syntax.mli), a modo de definición de tipos, los tipos Nat y Bool en un nuevo tipo de datos llamado (valga la redundancia) '_type':

```
10  open Support.Pervasive
11  open Support.Error
12
13  type _type =
14      TpBool
15    | TpNat
16
17
```

Después, añadimos al archivo core.ml (y a su correspondiente interfaz de módulo core.mli) la función 'gettype' (a la que nos referiremos durante la realización de todos los ejercicios), encargada de realizar las comprobaciones de tipo pertinentes. En este caso, además de comprobar los tipos simples derivados de los términos base (TmTrue, TmFalse y TmZero) de los tipos mencionados, también comprobaremos los tipos de las 'built-in functions' que están en el intérprete que se ha dado (TmSucc, TmPred, TmIsZero, TmIf).

```
(** ----------------------   TYPING  ---------------------- **)
let rec gettype ctx t = match t with
    TmTrue(fi) ->
      TpBool
  | TmFalse(fi) ->
      TpBool
  | TmZero(fi) ->
      TpNat
(*   | t when isnumericval ctx t  ->
      TpNat *)
  | TmIf(fi,t1,t2,t3) ->
      if ((gettype ctx t1) = TpBool)
      then let tpBody = (gettype ctx t2) in
        if tpBody = (gettype ctx t3)
        then tpBody
        else error fi "tipos diferentes en las ramas del if"
      else error fi "la condicion no es de tipo Bool"
  | TmSucc(fi,t1) ->
      let tp = gettype ctx t1 in
        (match tp with
            TpNat -> TpNat
          | _ -> error fi "expected value of type Nat for succ")
  | TmPred(fi,t1) ->
        let tp = gettype ctx t1 in
        (match tp with
            TpNat -> TpNat
          | _ -> error fi "expected value of type Nat for pred")
  | TmIsZero(fi,t1) ->
      let tp = gettype ctx t1 in
        (match tp with
            TpNat -> TpBool
          | _ -> error fi "expected value of type Nat for iszero")
```

Se omite, pero también hemos añadido la interfaz de esta función en el archivo core.mli, para su uso en el intérprete de alto nivel.

Acto seguido, definimos la función de printtype en el archivo syntax.ml para comprobar, finalmente, que la comprobación de tipos la hace correctamente y nos la muestra por

terminal. A mayores, en el archivo 'toplevel.ml', retocamos la función process_command añadiéndole la comprobación de tipos y la llamada al muestreo de tipos por pantalla.

```
369
370  let rec prtype ctx tp = match tp with
371      TpBool -> pr "Bool"
372    | TpNat -> pr "Nat"
```

```
389  let rec printtype ctx term tp_opt = match tp_opt with
390      None -> ()
391    | Some(tp) -> prtype ctx tp
392
```

```
61  (* processes the given command. Used by both methods process_file and process_line.
62     Results are printed a new context is returned *)
63  let rec process_command ctx cmd = match cmd with
64    | Eval(fi,t) ->
65        let tp = gettype ctx t in
66        let t' = eval ctx t in
67        printtm_ATerm true ctx t';
68        force_newline();
69        pr ": ";
70        printtype ctx t (Some(tp));
71        force_newline();
72        ctx
```

Y de esta forma, ya tendríamos nuestro intérprete de tipos básicos para elementos mínimos de los tipos Bool y Nat.

EJEMPLOS DE USO

Para comprobar el correcto funcionamiento de estas funcionalidades, además de anexar comandos de ejemplo que cubrirían todos los posibles casos de uso, mostraremos unos ejemplos y los explicaremos en este apartado, para cada ejercicio.

- Incorporación de TpBool y comprobación de TmIf:

  >> true; (Análogamente, false;)

  true: Bool

  >> if true then true else false;

  true: Bool

  >> if 2 then true else false;

  Error: La condición no es de tipo Bool

  >> if true then 2 else false;

  Error: Tipos diferentes en las ramas del If-Then-Else

- Incorporación de TpNat y comprobación de TmZero, TmIsZero, TmSucc, TmPred:

  >> 2;

  2: Nat

  >> succ 2;

  3: Nat

  >> succ true; (Análogamente, funciona igual para pred).

  Error: Expected value of type Nat for succ

  >> succ (pred 10);

  10: Nat

  >> if true then iszero(succ 0) else iszero(pred 1);

  false: Bool

Ejercicio 2

FUNCIONALIDADES SOLICITADAS

El comportamiento buscado en este ejercicio es que el nuevo intérprete no admita definiciones de funciones cuyos parámetros no vayan tipados de manera explícita.

CAMBIOS REALIZADOS

Para cambiar esto, lo primero que realizamos es un cambio en el lexer.mll, añadiendo los tipos que ya tenemos para poder pasarlos como entrada en la definición de funciones:

```
74      (*Types*)
75      ("Bool", fun i -> Parser.TBOOL i);
76      ("Nat", fun i -> Parser.TNAT i);
```

Asimismo, añadimos en el archivo parser.mly los elementos que reciban los tipos básicos que ya tenemos en el analizador léxico, y añadimos las reglas tanto de definición de tipos como aquellas que sirven para crear las lambda-abstracciones y los tipos de aplicación:

```
46  %token <Support.Error.info> TBOOL
47  %token <Support.Error.info> TNAT
48
```

```
135  Type :
136    AppType
137      { $1 }
138
139  AType :
140    LPAREN Type RPAREN
141      { $2 }
142    | TBOOL
143      {fun ctx -> TpBool}
144    | TNAT
145      {fun ctx -> TpNat}
146
147  AppType :
148    AType ARROW AppType
149      { fun ctx  -> TpApp($1 ctx, $3 ctx)}
150    | AType
151      { $1 }
```

Una vez realizado esto, modificamos las reglas de las lambda-abstracciones:

```
189    | LAMBDA LCID COLON Type DOT Term
190      /* An abstraction matching "lambda word . Term" adds word to the context
191        and returns a TmAbs with the word and term on the new context. */
192        { fun ctx ->
193            let ctx1 = addname ctx $2.v in
194            TmAbs($1, $2.v, $4 ctx, $6 ctx1) }
195    | LAMBDA USCORE COLON Type DOT Term
196      /* An abstraction using underscore (_) is treated like a variable whose name can be disregarded. */
197        { fun ctx ->
198            let ctx1 = addname ctx "_" in
199            TmAbs($1, "_", $4 ctx, $6 ctx1) }
```

Añadiéndole el ':' y el tipo, para definirlo de forma explícita en las funciones.

Después, a mayores, cambiaremos el tipo del binding que hay en 'syntax.ml' para añadirle el tipo a los términos (o a los parámetros de las lambda-abstracciones):

```
53  (* 2 types of binding. The standalone and the one associated with a term *)
54  type binding =
55      NameBind
56    | TmAbbBind of (term option) * (_type option)
57
```

De esta forma, admitimos 4 posibles tipos de binding, según estén término y tipo expresados, alguno de ellos, o ninguno (este último caso no se da en el intérprete, pero se contempla en el resto del código para evitar errores). El binding asociado a la lambda-abstracción solo tendrá el tipo, y aprovecharemos esto para añadirle el tipo a las asignaciones y para la instrucción Let (TmLet).

Para continuar, añadimos las reglas de tipado en nuestra función gettype, y los términos correspondientes a las aplicaciones y a las variables en el archivo 'syntax.ml':

```
21  (** Datatypes **)
22
23  (* Types recognized by the program *)
24  type _type =
25      TpBool
26    | TpNat
27    | TpVar of int * int
28    | TpApp of _type * _type
29
```

```
186   | TmVar(fi,i,_) -> searchFromContextType fi ctx i
187   | TmAbs(fi,x,tpT1,t1) ->
188       let ctx' = addbinding ctx x (TmAbbBind(None,(Some(tpT1)))) in
189       let tpT2 = gettype ctx' t1 in
190         TpApp(tpT1,tpT2)
191   | TmApp(fi,t1,t2) ->
192       let tpT1 = (gettype ctx t1) in
193       let tpT2 = (gettype ctx t2) in
194       (match tpT1 with
195           TpApp(tpT11,tpT12) ->
196             if tpT2 = tpT11 then tpT12
197             else error fi "input parameter doesn't match"
198         | _ -> error fi "expected type for application")
199   | TmLet(fi,x,t1,t2) ->
200       let tpT1 = gettype ctx t1 in
201       let ctx' = addbinding ctx x (TmAbbBind(Some(t1),(Some(tpT1)))) in
202         gettype ctx' t2
```

Para asegurarnos de que el contexto realiza bien estas operaciones, añadiremos una operación análoga al termShifting definida en el archivo 'syntax.ml' pero para los tipos que guardemos en el contexto, manteniendo la consistencia a la hora de definir bindings y evitando perder la referencia a ellos. Se omite debido a que su utilidad es importante pero no aporta nada diferente a lo que se proporciona en el código.

Para poder imprimir correctamente el contenido de los bindings por pantalla, modificamos una función que coge el tipo de los bindings y todos estos cambios los añadimos al 'process_command' del toplevel.ml:

```
393  (* prbinding prints a binding depending on its type. In case it's of type
394     NameBind, nothing is printed and unit is returned. For TmAbbBind type,
395     the equals symbol and the binding's term are printed *)
396  let prbinding ctx b = match b with
397      NameBind -> ()
398    | TmAbbBind(Some(t),tp) -> pr "= "; printtm ctx t; pr ":"; printtype ctx t tp
399    | TmAbbBind(None,_) -> ()
```

Y con esto ya estaría implementado el tipado explícito para la definición de funciones, así como el uso de asignaciones y del término TmLet para manejar el contexto con tipos.

EJEMPLOS DE USO

Para la comprobación de las abstracciones, lo principal es saber si, con la implementación dada, el resto de cambios de nuestro código siguen siendo consistentes con lo que acabamos de introducir. Así pues, entremezclaremos distintos términos de distintas formas para comprobar que todo se realiza correctamente:

- Comprobación del funcionamiento del tipo explícito en las lambda-abstracciones:

  >> lambda x:Bool. x;

  (lambda x.x): Bool->Bool

  >> lambda _:Nat. true;

  (lambda _.true): Nat->True

  >> lambda x:Bool. if x then 10 else 20;

  (lambda x. if x then 10 else 20): Bool -> Nat

  >> lambda x:Bool. (lambda y:Nat. if x then y else succ y);

  (lambda x. (lambda y. if x then y else succ y): Bool->Nat->Nat

- Comprobación del funcionamiento de la aplicación:

  >> (lambda x:Bool. if x then 10 else 20) true;

  10: Nat

  >> (lambda x:Bool. if x then 10 else 20) 10;

  Error: Input parameter doesn't match

  >> 10 20;

  Error: Expected arrow type

  >> (lambda x:Bool. lambda y:Nat. if x then y else succ y) false 20;

  21: Nat

- Comprobación del correcto funcionamiento de la asignación, y de la asignación local mediante la instrucción Let, así como del manejo del contexto;

  >> x = 10;

  x = 10: Nat

  >> x;

  10: Nat

  >> x = lambda x:Bool. if x then 10 else 12;

x = lambda x.if x then 10 else 12: Bool->Nat;

>> x true;

10: Nat

>> let x = 10 in (succ x);

11: Nat

>> let y=12 in let a=10 in lambda x:Nat. lambda a:Nat. if true then (succ a) else (pred y);

lambda x. lambda a. if true then (succ a) else (pred y): Nat->Nat->Nat;

>> fun = let y=12 in let a=10 in lambda x:Nat. lambda a:Nat. if true then (succ a) else (pred y);

fun = lambda x. lambda a. if true then (succ a) else (pred y): Nat->Nat->Nat;

>> fun 10 12;

13: Nat

>> lambda x:Bool. lambda x:Nat. pred x;

lambda x:Bool. lambda x'.pred x': Bool->Nat->Nat

Ejercicio 3

FUNCIONALIDADES SOLICITADAS

El tercer apartado de la práctica es incorporar al intérprete un combinador de punto fijo interno (fix), de tal forma que se puedan declarar este tipo de funciones mediante definiciones recursivas directas.

CAMBIOS REALIZADOS

Para afrontar este ejercicio, comenzamos definiendo TmFix en el archivo 'syntax.ml' para añadir el operador de punto fijo y poderlo utilizar para habilitar la recursión:

```
33  (* Terms recognized by the program *)
34  type term =
35      TmTrue of info                  (* Boolean True *)
36    | TmFalse of info                 (* Boolean False *)
37    | TmIf of info * term * term * term     (* If/then/else *)
38    | TmVar of info * int * int             (* Variable *)
39    | TmAbs of info * string * _type * term     (* Abstraction *)
40    | TmApp of info * term * term           (* Application *)
41    | TmRecord of info * (string * term) list   (* Record *)
42    | TmProj of info * term * string        (* Projection *)
43    | TmFloat of info * float               (* Float *)
44    | TmTimesfloat of info * term * term        (* Product of floats *)
45    | TmString of info * string             (* String *)
46    | TmZero of info                    (* Zero *)
47    | TmSucc of info * term             (* Successor *)
48    | TmPred of info * term             (* Predecessor *)
49    | TmIsZero of info * term           (* IsZero *)
50    | TmLet of info * string * term * term    (* Local variable *)
51    | TmFix of info * term
```

En el archivo 'lexer.mll' añadimos la palabra reservada "letrec" para poder definir funciones recursivas:

```
20  let reservedWords = [
21    (* Keywords *)
22    ("if", fun i -> Parser.IF i);
23    ("then", fun i -> Parser.THEN i);
24    ("else", fun i -> Parser.ELSE i);
25    ("true", fun i -> Parser.TRUE i);
26    ("false", fun i -> Parser.FALSE i);
27    ("lambda", fun i -> Parser.LAMBDA i);
28    ("letrec", fun i -> Parser.LETREC i);
29    ("timesfloat", fun i -> Parser.TIMESFLOAT i);
30    ("succ", fun i -> Parser.SUCC i);
31    ("pred", fun i -> Parser.PRED i);
32    ("iszero", fun i -> Parser.ISZERO i);
33    ("let", fun i -> Parser.LET i);
34    ("in", fun i -> Parser.IN i);
35
```

A continuación, añadimos al parser.mly la regla que nos permitirá la recursión dentro de una abstracción:

```
207     | LETREC LCID COLON Type EQ Term IN Term
208         { fun ctx ->
209             let ctx1 = addname ctx $2.v in
210             TmLet($1, $2.v, TmFix($1, TmAbs($1, $2.v, $4 ctx, $6 ctx1)),
211                 $8 ctx1) }
```

El cual es una combinación de la sentencia Let, junto al uso del término del operador fix que lleva dentro la abstracción a la que le tiene que aplicar la recursión.

Ahora, tendremos que tocar el archivo 'core.ml' para añadir la regla de evaluación de fix y poder también deducir el tipo en la función de obtención de tipo:

Eval:

```
116         | TmFix(fi,t1) ->
117             let t1' = eval1 ctx t1
118             in TmFix(fi,t1')
119         | _ ->
120             raise NoRuleApplies
```

Gettype:

```
214     | TmFix(fi, t1) ->
215         let tpT1 = gettype ctx t1 in
216         (match tpT1 with
217             TpApp(tpT11,tpT12) ->
218               if (tpT12) = (tpT11) then tpT12
219               else error fi "result of body not compatible with domain"
220           | _ -> error fi "arrow type expected")
221
```

EJEMPLOS DE USO

Para comprobar la correcta implementación del combinador de punto fijo, realizaremos una función de recursión de suma mediante letrec, y trataremos de concatenar otra función recursiva dentro de la primera:

- Comprobación del correcto funcionamiento de la instrucción letrec y de TmFix:

  >> letrec sum:Nat -> Nat -> Nat = lambda n:Nat.lambda m:Nat. if iszero n then m else succ(sum (pred n) m) in sum 2 3;

  5: Nat

  >> letrec sum_1:Nat -> Nat -> Nat = lambda n:Nat.lambda m:Nat. if iszero n then m else succ(sum_1 (pred n) m) in letrec sum_2:Nat->Nat->Nat = lambda n:Nat.lambda m:Nat. if iszero n then m else succ (sum_2(pred n) m) in sum_2 (sum_1 4 5) 9;

  18: Nat

Ejercicio 4

FUNCIONALIDADES SOLICITADAS

El último apartado de la práctica consiste en gestionar otros tipos adicionales como pueden ser los tipos String, Float o Record.

Para ello las opciones planteadas son, rechazar estas expresiones, conservar el comportamiento original o bien definir nuevos tipos de datos; En nuestro caso decidimos definir nuevos tipos de datos.

CAMBIOS REALIZADOS

En este apartado comenzamos con los tipos String y Float que son los más similares a los tipos Nat y Bool definidos previamente, para ello definimos los nuevos tipos en el lexer.mll, aceptándolos en el parser.mly y creando los tipos en el syntax.ml:

```
74    (*Types*)
75    ("Bool", fun i -> Parser.TBOOL i);
76    ("Nat", fun i -> Parser.TNAT i);
77    ("String", fun i -> Parser.TSTRING i);
78    ("Float", fun i -> Parser.TFLOAT i);
79
```

```
22
23  (* Types recognized by the program *)
24  type _type =
25      TpBool
26    | TpNat
27    | TpString
28    | TpFloat
29    | TpRecord of (string * _type) list
30    | TpVar of int * int
31    | TpApp of _type * _type
```

Del tipo record hablaremos más adelante, por ser un poco más complejo.

Continuamos añadiendo al core.ml las reglas de obtención de tipos, siendo los tipos básicos nuevos de la misma tónica que los tipos Bool y Nat. A mayores, tenemos en cuenta

la primitiva 'TmTimesFloat', que consiste en una multiplicación de dos TmFloat.

```
82      | TmTimesfloat(fi,TmFloat(_,f1),TmFloat(_,f2)) ->
83          TmFloat(fi, f1 *. f2)
84      | TmTimesfloat(fi,(TmFloat(_,f1) as t1),t2) ->
85          let t2' = eval1 ctx t2 in
86          TmTimesfloat(fi,t1,t2')
87      | TmTimesfloat(fi,t1,t2) ->
88          let t1' = eval1 ctx t1 in
89          TmTimesfloat(fi,t1',t2)
```

Por último, modificamos la función de impresión de tipos en el archivo 'syntax.ml':

```
370   let rec prtype ctx tp = match tp with
371         TpBool -> pr "Bool"
372       | TpNat -> pr "Nat"
373       | TpString -> pr "String"
374       | TpFloat -> pr "Float"
```

Ahora, les toca el turno a los registros (y el correspondiente término de la proyección sobre éste). Para eso definimos el tipo de registro (TpRecord, ya mostrado en la imagen anterior del archivo syntax.ml). Este tipo de registro guarda una lista de tipos, donde cada tipo se refiere a cada uno de los elementos del registro, guardando la etiqueta de cada uno de los campos y su tipo. Ahora, falta añadir la regla de obtención de tipo en nuestra función gettype:

```
203     | TmRecord(fi, fields) ->
204         let fieldtypes =
205         (*Mapeo cada tipo de termino en un tipo registro*)
206           List.map (fun (li,ti) -> (li, gettype ctx ti)) fields in
207         TpRecord(fieldtypes)
208     | TmProj(fi, t1, l) ->
209         (match gettype ctx t1 with
210             TpRecord(fieldtys) ->
211               (try List.assoc l fieldtys
212                with Not_found -> error fi ("label "^l^" not found"))
213           | _ -> error fi "Expected record type")
```

Para el record, utilizamos la función de evaluación elemento a elemento y la mapeamos sobre una lista que contendremos en el TpRecord. Para la proyección, buscaremos la etiqueta en el nombre del registro, y si existe, devuelve el tipo asociado, calculado previamente.

Con esto ya tendríamos completo nuestro intérprete de lambda cálculo con recursividad implementada y todos los tipos que se nos han dado para su uso.

EJEMPLOS DE USO

- Comprobación del correcto funcionamiento del tipo Float y del tipo String:

  >>10.2;

  10.2: Float;

  >>timesfloat 10.0 2.5;

  25.0: Float;

  >>mult = lambda x:Float. lambda y:Float. timesfloat x y;

  mult = (lambda x. lambda y. timesfloat x y): Float-> Float-> Float

  >> timesfloat true 10.0;

  Error: first argument of timesfloat is not a Float

  >> timesfloat 10.0 true;

  Error: second argument of timesfloat is not a Float

  >> “Hola Mundo”;

  “Hola Mundo”: String

  >> let x=”hola” in x;

  “hola”: String

  >>echo = lambda x:String. x;

  echo = (lambda x.x): String -> String

  >> echo “10”;

  “10”: String;

- Comprobación del correcto funcionamiento del tipo Record y del tipo Proj:

  >> {};

  {}: {}

  >> rec = {1,”hola”,10.0,true};

  rec = {1,”hola”,10.0,true}: {Nat, String, Float, Bool}

  >> rec.1;

  1:Nat

  >> rec.etiqueta

Error: label etiqueta not found

>> rec = {nat=1,string="hola",10.0,bool=true};

rec={nat=1, string="hola", 10.0, bool=true}: {nat:Nat ,string:String ,Float, bool:Bool}

>> rec.nat;

1: Nat;

>> rec.1;

Error: label 1 not found;

>> rec.3;

10.: Float

>> rec = {lambda x:Nat. iszero x, let x=10 in succ x};

rec = {lambda x. iszero x, 11}: {Nat->Bool, Nat}

>> rec.1;

(lambda x. iszero x): Nat->Bool

>> rec.1 0;

true: Bool

>> rec = {lambda x:Nat. iszero x, let x=10 in succ x, letrec sum:Nat->Nat->Nat = lambda m:Nat. lambda n:Nat. if iszero n then m else succ (sum (pred n) m) in sum 10 20};

rec = {lambda x. iszero x, 11, 30} : {Nat->Bool, Nat, Nat}