Memoria P1

Manuel Couto Pintos

Índice

1 Introducción

En este manual se documentan las distintas funcionalidades iniciales del intérprete lambda-3 junto con las extensiones realizadas sobre el mismo. Así pues, a la hora de mostrar la sintaxis de cada elemento se emplearán los rombos ”<termino>” para indicar un término; “:Tipo” para indicar un tipo, que podrá ser de tipo genérico T o Ti / i [1..n], o un tipo concreto como Nat, Bool o cualquiera de los definidos posteriormente. Los corchetes “[]” y barra “|” permitirán listar las opciones. Además, se obviará el separador final en las definiciones de la sintaxis de la funcionalidad.

2 Compilación y Ejecución

Para compilar y ejecutar el programa se emplean comandos definidos en el archivo Makefile, que permite lanzar órdenes con la siguiente sintaxis:

make [all | lambda | parser | lexer | main | clean | exec | Debug]

all **->** Compila todos los archivos del proyecto generando un ejecutable ./**top**

lambda **->** Compila el módulo Lambda

parser **->** Compila el módulo Parser

lexer **->** Compila el módulo Lexer

main **->** Compila el main.ml

clean **->**Elimina los archivos binarios generados durante la compilación.

exec **->** Compila todo y ejecuta **rlwrap** ./**top**

debug **->** Compila todo y ejecuta **rlwrap ./top -d**

(¡**Requisito!)** Deben estar instalados en el sistema **ocaml** y **rlwrap** para poder ejecutar los comandos definidos en el Makefile.

3 Términos Previamente Implementados

3.1 Sentencia Condicional

La sentencia condicional permiten controlar el flujo de ejecución de las diferentes instrucciones. La sintaxis es la siguiente:

if <term:Bool> then <term:T > else <term:T>: T

Un ejemplo de ejecución:

if false then true else false;;

```
>> if false then true else false;;
false : Bool
```

3.2 Abstracciones

Las abstracciones permiten la asociación de valores de parámetros de entrada de las funciones a variables libres definidas en el cuerpo de la función. La sintaxis es la siguiente:

[L | lambda] x:T. <term> :(T) ->(<term>:T)

Algunos ejemplos:

lambda x:Nat. x;;

L x:Nat. x;;

```
>> lambda x:Nat. x;;
(lambda x:Nat. x) : (Nat) -> (Nat)
>> L x:Nat. x;;
(lambda x:Nat. x) : (Nat) -> (Nat)
```

3.3 Aplicaciones

Las aplicaciones permiten evaluar abstracciones aplicadas sobre términos con tipos compatibles mediante la aplicación de betareducciones. La sintaxis general es:

<abstraction:(T1)->(T2)><term:T1>:T2

Un ejemplo de esto:

x= L x:Nat x;;

x 1;;

```
>> x = L x:Nat. x;;
x = (lambda x:Nat. x) : (Nat) -> (Nat)
>> x 1;;
1 : Nat
```

3.3.1 Instrucción Let

La herramienta también cuenta con la instrucción **let** a modo de simplificar el proceso de definición de una abstracción y su aplicación sobre un determinado término, tiene la sintaxis:

let <var> = <abstraction:(T1)->(T2)> in <application> <term:T1> : T2

Un ejemplo:

let id_nat = L x:Nat. x in id_nat 1;;

```
>> let id_nat = L x:Nat. x in id_nat 1;;
1 : Nat
```

Existen otras aplicaciones específicas a determinados tipos que están definidas internamente, las cuales se detallarán en el apartado 3.4.3, enfocado en las aplicaciones asociadas a tipos.

3.4 Términos asociados a tipos

Algunos términos están definidos internamente en la herramienta y pueden tener un tipo específico o aplicarse únicamente a un tipo específico.

3.4.1 Booleano

[true|false]: Bool

Un ejemplo de uso:

true;;

false;;

3.4.2 Natural

Hay un límite en el entero máximo que puede procesar la herramienta debido a las limitaciones de ocaml y la forma que está implementado *TmSucc* en la función *eval1*, que deja muchas operaciones pendientes antes de llegar al elemento *TmZero* y provoca una excepción *Stack_overflow*.

<1..MaxInt>:Nat

Un ejemplo:

1;;

232;;

3.4.3 Aplicaciones asociadas a tipos

1. Sucesor

succ <term: Nat> : Nat

Un ejemplo:

succ 1;;

```
>> succ 1;;
2 : Nat
```

2. Predecesor

pred <term: Nat> : Nat

Un ejemplo:

pred 9;;

pred 0;;

```
>> pred 9;;
8 : Nat
>> pred 0;;
0 : Nat
```

3. Es cero

iszero <term:Nat> : Bool

Un ejemplo:

iszero 0;;

iszero 1;;

```
>> iszero 0;;
true : Bool
>> iszero 1;;
false : Bool
```

4 Mejoras en la introducción y escritura de las lambda expresiones

4.1 Reconocimiento de expresiones muti-línea

El sistema es capaz de reconocer expresiones multi-línea, utilizándose un separador **“;;”** para indicar el fin de una instrucción.

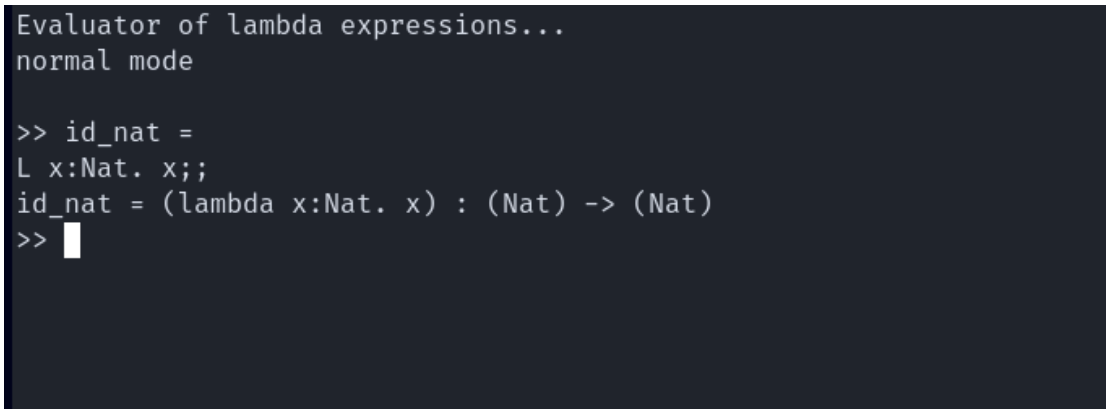
< instruc-

tion>;;

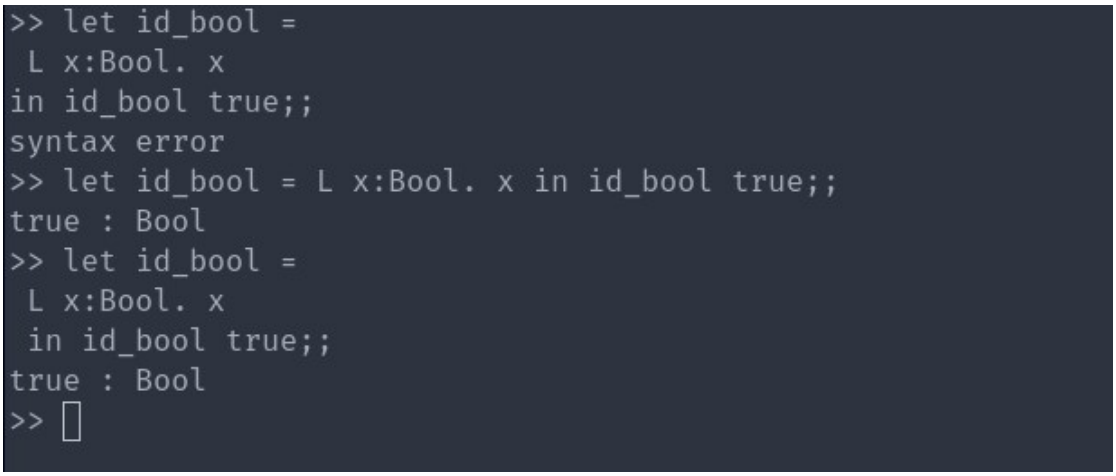
A continuación se muestra un ejemplo de ejecución:

id_nat =

L x:Nat. X;;



Además, es importante mantener los espacios en blanco aunque estén en líneas distintas:



4.2 Reconocimiento de líneas multi-expresión.

El sistema reconoce secuencias de instrucciones en una misma línea, pero deben estar separadas por el separador **“;;”**.

<inst1>;;<inst2>;; <*eol*>

Un ejemplo:

x={l=1, r=2, t=4};; x.l;;

```
>> x = {l=1, r=2, t=4 };; x.l;;
x = {l=1,r=2,t=4} : {l:Nat,r:Nat,t:Nat}
1 : Nat
>>
```

4.3 Reconocimiento de expresiones desde fichero

El sistema tiene una palabra reservada para la importación de instrucciones desde fichero, esta palabra reservada es **#open**, su sintaxis es la siguiente:

#open <Path> ;;

Ejemplo abriendo archivo integracion.tdd :

#open integracion.tdd;;

```
>> #open integracion.tdd;;
1;;
1.1;;
"String";;
'n';;

{1,2};;
```

También se puede abrir un archivo y ejecutar instrucciones del archivo en la misma línea:

#open integracion.tdd;; head x3;;

```
>> #open integracion.tdd;; head x3;;
1;;
1.1;;
"String";;
'n';;

{1,2};;
{1,2,3};; {1,2}.1;; {1,2}.2;;

x = {1,2.003};; x.1;; x.2;;

x = {1,"String",'c'};; x.3;; x.2;; x.1;;
```

5 Mejoras en la evaluación del lambda-cálculo

5.1 Incorporación de modo debug

El modo debug puede activarse usando el flag **-d**. De esta forma se ven los sucesivos términos que genera la función *eval1*, además del contenido del contexto de tipos y de términos.

./top -d

También existe un comando en el fichero make para compilar y ejecutar con y sin modo debug:

make <exec/debug>

Un ejemplo:

make debug

let id_bool = L x:Bool. x in id_bool true;;

```
manux2  master  …  practicas  master  DLP  130  make debug
ocamlc -c lambda.mli lambda.ml
ocamlyacc parser.mly
ocamlc -c parser.mli parser.ml
ocamllex lexer.mll
122 states, 3912 transitions, table size 16380 bytes
ocamlc -c lexer.ml
ocamlc -c main.ml
ocamlc -o top lambda.cmo parser.cmo lexer.cmo main.cmo
rlwrap ./top -d
Evaluator of lambda expressions...
debugmode active

>> let id_bool = L x:Bool. X in id_bool true;;
lexical error
>> let id_bool = L x:Bool. x in id_bool true;;

TOKENS:
TmLet(id_bool,TmAbs(x, TyBool, TmVar x),TmApp(TmVar id_bool,TmTrue))

EVAL:
let id_bool = (lambda x:Bool. x) in (id_bool true)

TOKENS:
TmApp(TmAbs(x, TyBool, TmVar x),TmTrue)

EVAL:
((lambda x:Bool. x) true)

TOKENS:
TmTrue

EVAL:
true
true : Bool
Context terms:
Context Types:
id_bool : (Bool) -> (Bool)
x : Bool
>>
```

6 Aplicaciones del lenguaje lambda-cálculo

6.1 Combinador de punto fijo interno

El sistema implementa la recursividad. Se emplea la palabra reservada **letrec** que extiende directamente la funcionalidad de la directiva **let,** permitiendo la aplicación cíclica del primer **término** sobre sí mismo.

La sintaxis empleada:

letrec <var>: <(T1)->(T2)> = <abstraction:(T1)->(T2)> in <application> <term:T1> : T2

Un ejemplo básico:

letrec sum: Nat -> Nat -> Nat =

L n :Nat. L m: Nat.

if iszero n then m else succ (sum (pred n) m) in sum 23 55;;

```
letrec sum : Nat -> Nat -> Nat =
lambda n : Nat. lambda m : Nat.
if iszero n then m else succ (sum (pred n) m) in sum 23 55;;
78 : Nat
>>
```

6.2 Incorporación de contexto de variables libres

El sistema tiene incorporado un contexto de variables libres, la sintaxis necesaria es la siguiente:

<var> = <term:T>:T

Un ejemplo de uso:

x =1;; x2= 3;;

letrec sum: Nat -> Nat -> Nat =

L n :Nat. L m: Nat.

if iszero n then m else succ (sum (pred n) m) in sum x x2;;

```
>> x =1;; x2= 3;;
x = 1 : Nat
x2 = 3 : Nat
letrec sum: Nat -> Nat -> Nat =
L n :Nat. L m: Nat.
if iszero n then m else succ (sum (pred n) m) in sum x x2;;
4 : Nat
```

6.3 Incorporación de más clases de términos

6.3.1 Tuplas / Pairs

Sintaxis tuplas/pairs:

{<term1:T1>,<term2:T2>, ...<term3:Tn>}:{<<T1> * <T2> * ... * <Tn>>}

Un ejemplo de tupla y pair:t

{1, "Test"};;

{1, "Test", 'c'};;

```
>> {1,"Test"};;
{1,"Test"} : Nat * String
>> {1,"Test",'c'};;
{1,"Test",'c'} : {Nat * String * Char}
```

Las operaciones sobre tuplas permiten acceder a los diferentes valores almacenados. Para ello se hace uso de la operación **projección**, que tiene la siguiente sintaxis:

<term:Pair/Tuple>.<term:Nat>:<Tn>

Un ejemplo de proyección:

{1,"Test"}.2;; {1,"Test",'c'} .3;;

```
>> {1,"Test"}.2;;{1,"Test",'c'}.3;;
"Test" : String
'c' : Char
```

6.3.2 Records

Los records permiten almacenar téminos asociados a variables, la sintaxis es la siguiente:

{<var k1>=<term1:T1>, <var k2>=<term2:T2>, ... , <var kn>=<termn:Tn>}: {<var k1>:T1, <var k2>:T2, ... <var kn>:Tn}

Un ejemplo de una tupla:

{t=1, l=true, s=2.0094};;

```
>> {t=1, l=true, s=2.0094};;
{t=1,l=true,s=2.0094} : {t:Nat,l:Bool,s:Real}
```

Para acceder a los valores de los registros se usa la operación **proyección**, la sintaxis es la siguiente:

{<var k1>=<term1:T1>, <var k2>=<term2:T2>, ... , <var kn>=<termn:Tn>}.<var ki>:Ti

Un ejemplo:

{t=1, l=true, s=2.0094}.s;;

```
>> {t=1, l=true, s=2.0094}.s;;
2.0094 : Real
```

6.3.3 Listas

Sintaxis de las listas:

[<term1:T>,<term2:T>, ... , <termn:T>]:[T]: List T

Un ejemplo:

[true, false, true]:[Bool];; [1,2,3,4]:[Nat];; [1.23, 0.234, 1.431]:[Real];;

['s', '9', 't']:[Char];; ["as", "asdkk", "asdasd"]:[String];;

```
>> [true, false, true]:[Bool];; [1,2,3,4]:[Nat];; [1.23, 0.234, 1.431]:[Real];;
[true,false,true] : List Bool
[1,2,3,4] : List Nat
[1.23,0.234,1.431] : List Real
>> ['s', '9', 't']:[Char];; ["as", "asdkk", "asdasd"]:[String];;
['s','9','t'] : List Char
["as","asdkk","asdasd"] : List String
```

Un ejemplo de la sintaxis para definir una lista **vacía**:

[T]: List T

Un ejemplo:

[Char];; [Bool];; [Nat];; [Real];;

```
>> [Char];; [Bool];; [Nat];; [Real];;
[ ] : List Char
[ ] : List Bool
[ ] : List Nat
[ ] : List Real
```

Están implementadas las operaciones sobre listas, **tail**, **head**, **construct:**

Sintaxis **tail**:

tail <termL:List T>: List T

Un ejemplo:

tail ['s', '9', 't']:[Char];;

```
>> tail ['s', '9', 't']:[Char];;
['9','t'] : List Char
```

Sintaxis **head**:

head <termL:List T>: T

Un ejemplo:

head ['s', '9', 't']:[Char];;

```
>> head ['s', '9', 't']:[Char];;
's' : Char
```

Sintaxis **construct**:

<term:T>::<termL:List T> : List T

Un ejemplo:

'a'::['s', '9', 't']:[Char];;

```
>> 'a'::['s', '9', 't']:[Char];;
['a','s','9','t'] : List Char
```

6.4 Incorporación de nuevos tipos

6.4.1 Tipo *Char*

Se ha incorporado el tipo ***Char*** con la sintaxis :

‘<char>’: Char

Un ejemplo:

‘c’;;

```
>> 'c';;
'c' : Char
```

6.4.2 Tipo String

Se ha incorporado el tipo ***String***, con la sintaxis:

"<String>": String

Un ejemplo:

"String";;

```
>> "String";;
"String" : String
```

6.4.3 Tipo *Real / Float* Point

Se ha incorporado el tipo ***Real***, con la sintaxis:

<parte entera>.<parte decimal>: Real

Un ejemplo:

4.003;;

```
>> 4.003;;
4.003 : Real
```

7 Quit

Se puede salir del programa con el tomando quit que tiene la siguiente sintaxis:

#quit;;

8 Fichero Integracion.tdd

El fichero integración es un archivo que se ha usado durante el desarrollo para asegurar que las nuevas funcionalidades no afectaran al correcto funcionamiento de la herramienta. Las instrucciones de ejemplo de la documentación están incluidas al final en este documento y se pueden ejecutar con la instrucción:

#open integracion.tdd

9 Aviso

Si se copian los comandos directamente desde este fichero, la letra cursiva hace que algunos caracteres como la comilla simple o doble den fallos en la aplicación