Programación Funcional

Ingeniería Informática 2005

Práctica #2 (λ-Cálculo con tipos)

Esta práctica consiste en la implementación de un interprete/compilador de términos en el λ-cálculo con tipos, usando enteros, booleanos y expresiones condicionales. Notas a tener en cuenta para la realización de la práctica:

- Los ficheros de la práctica se enxcuentran en la direccion web: `http://www.lfcia.org/~quintela/programacion_f`
- La documentación a entregar consistirá en la práctica que compile con todos los ficheros.
- Un fichero de texto (`lambda.txt`) describiendo la labor realizada, junto con ejemplos de uso y cualquier explicación que se considere pertinente. Si se modificase algún fichero además de `tipado.ml` y `lambda.ml` comentese en este fichero que ficheros han cambiado y la razón de los cambios.
- En los ficheros entregados para la realización de la práctica están definidos todos los tipos/constructores necesarios para las partes opcionales de la práctica. Si se lee un constructor que no se contempla habrá que lanzar una excepción con un mensaje oportuno. (Ver los ejemplos de implementación que se dan).

Los tipos están definidos en el módulo `Tipos`.

```
type variable = string
type booleano = F | V
type tipo = Bool
          | Int
          | Flecha of tipo * tipo
          | Variable of variable
type op2 = Sum | Res | Mul | Div | Mayor | Igual | Menor
type term = CBool of booleano
          | Op2 of op2 * term * term
          | Var of variable
          | Abs of variable * tipo * term
          | App of term * term
          | Cond of term * term * term
```

Para leer y presentar los términos use las siguientes funciones implementadas ya:

```
val Pretty.term : Tipos.term -> string
val Pretty.tipo : Tipos.tipo -> string
val Lector.term : string -> Tipos.term
val Lector.fichero : string -> Tipos.term
```

donde `Lector.fichero` toma como argumento el nombre de un fichero y devuelve el término que se encuentra en ese fichero. La sintaxis de los términos en el λ-cálculo con tipos que entiende el Lector es:

termino ::= **true**
| **false**
| *numero*
| *termino op2 termino*
| *variable*
| **/** *variable* **.** *termino*
| **/** *variable* **:** *tipo* **.** *termino*
| *termino termino*
| **if** *termino* **then** *termino* **else** *termino*

numero ::= [**0**–**9**]+

variable ::= [**a-z**] [**a-z0**–**9_**]*

op2 ::= + | **-** | ***** | / | = | > | <

tipo ::= **Bool**
| **Int**
| *tipo* `->` *tipo*

La sintaxis que entiende el lector de términos para las diferentes definiciones:

let *VAR1* := *termino1* ;;
let *VAR2* := *termino2* ;;
...
let *VARn* := *terminon* ;;

un ejemplo de uso es:

```
# let e1 = Lector.term "let x := /x:Int.x = 5 + 3;; let main := x 3";;
val e1 : Tipos.term =
  App
   (Abs ("x", Int, Op2 (Igual, Var "x", Op2 (Sum, CInt 5, CInt 3))),
    CInt 3)
# Pretty.term e1;; Pretty.term e1;;
- : string = "((/x:Int.(x=(5+3))) 3)"
```

Puede evaluarse un término a partir de un fichero usando el comando `lambda`. A continuación se presenta un ejemplo de uso:

```
quintela$ cat fact
let pfact := / g:Int->Int./ n:Int. if n =0 then 1 else n * g (n - 1);;
let fact  := Y pfact;;
let main  := fact 5;;
quintela$ ./lambda fact
valor = 120
tipo = Int
```

Ejercicio 1 (**Evaluador**) Construya un evaluador *call-by-need* `eval` para el λ-cálculo con constantes. Esta función se implementará en el fichero `lambda.ml` usando el esqueleto proporcionado a tal efecto en ese fichero.

```
eval: Tipos.term -> Tipos.term
```

Ejercicio 2 (**Sistema de Tipos**) Defina las reglas que definen un sistema de tipos para los valores `term`. Basándose en dichas reglas, implemente la función `tipo_de` que permite determinar el tipo de un valor `term` y rechazar términos mal tipados. Esta función se definirá en el fichero `tipado.ml` usando el esqueleto de función dado a tal efecto en dicho fichero.

```
tipo_de: Tipos.term -> Tipos.tipo
```

Ejercicio 3 (**Evaluador + Sistema de Tipos**) Combinando `tipo_de` y `eval` construya una función de evaluación libre de errores de tipos en tiempo de ejecución. Esta función se implementará en el fichero `lambda.ml` usando el esqueleto proporcionado a tal efecto en dicho fichero.

```
eval_seguro: term -> (term * tipo)
```

Ejercicio 4 (**Extensiones a la práctica (Opcionales)**)

- Definir la función `Pretty.term` de forma que minimice el número de paréntesis.
- Introducir las cadenas de caracteres en el lenguaje. Las cadenas se representarán por el tipo `String`. Las operaciones a implementar son la concatenación, (la sintaxis es el operador binario `^`) y la función que devuelve la longitud de la cadena.:

```
type tipo = ...
          | String
type op1 = Lengthc
type op2  = ...  | Conc
type term = ...
          | Cadena of string
          | Op1 of op1 * term
```

  donde la sintaxis ha aumentado de la siguiente forma:

termino	::=	...
	|	*cadena*
	|	*op1 termino*
op1	::=	**lengthc**
op2	::=	...
	|	`^`
tipo	::=	...
	|	**String**

- Definir el operador menos unario (`-`)para los enteros.

```
type op1 = ...
         | Menos
```

  con la sintaxis:

op1	::=	`-`

- Definir la construcción *let... in* en el lenguaje.

```
type term = ...
          | LetIn of variable * term * term
```

  con la sintaxis:

termino	::=	...
	|	**let** *variable* = *termino* **in** *termino*

- Definir el operador punto fijo (Y) en el lenguaje para poder implementar términos recursivos:

```
type term = ...
          | Y term
```

con la sintaxis:

termino	::=	...
	|	**Y** *termino*

- Definir la evaluación y el tipado para el tipo producto dentro del lenguaje.

```
type tipo = ...
          | Producto of tipo * tipo
```

```
type term = ...
          | Par of term * term
          | Fst of term
          | Snd of term
```

con la sintaxis:

termino	::=	...
	|	(*termino* **,** *termino*)
	|	**fst** *termino*
	|	**snd** *termino*

- Definir la evaluación y el tipado para las listas de enteros:

```
type term = ...
          | Nil
          | ConsInt of term * term
          | Hd of term
          | Tl of term
          | Null of term
```

con la sintaxis:

termino	::=	...
	|	[]
	|	*termino* **::** *termino*
	|	[*elementos*]
	|	**hd** *termino*
	|	**tl** *termino*
	|	**null** *termino*
elementos	::=	*termino*
	|	*termino* **,** *elementos*

- Definir la inferencia de tipos. El único cambio es que las abstracciones no necesitan tener tipo explicito.
- **dificil** Cambiar toda la estructura del sistema para permitir polimorfismo.