Práctica de Laboratorio

Intérprete de Lambda Cálculo

Grado en Ingeniería Informática (Q7)
Diseño de Lenguajes de Programación
Curso 2022/23

1 Objetivos

Junto con el enunciado de esta práctica se proporcionan varias implementaciones de un intérprete de lambda cálculo escritas en OCaml, las cuales están inspiradas en las implementaciones que Benjamin C. Pierce cita en su libro *Types and Programming Languages*, y ya han sido explicadas en las clases de prácticas. Dichas implementaciones procesan expresiones de lambda cálculo escritas ante el prompt de un lazo interactivo, las evalúan y escriben en pantalla los correspondientes resultados. El propósito de esta práctica es estudiar dichas implementaciones y ampliarlas en los términos que se explican a continuación. Se considerarán tanto mejoras en los propios intérpretes, como extensiones en el lenguaje lambda cálculo que reconocen.

Hemos visto tres versiones del intérprete: `lambda-1`, `lambda-2` y `lambda-3`. Las mejoras y extensiones que consideraremos no afectarán en ningún caso a las versiones `lambda-1` y `lambda-2`. En el primer caso, por ser demasiado básica y sobre todo por la incomodidad de manejo que implica el uso de los términos "Church booleans" y "Church numerals" para la representación de los valores lógicos y de los números naturales. Pero la razón principal, en ambos casos, es el hecho de no incluir lambda cálculo tipado. Así pues, afectarán únicamente a la versión más completa: `lambda-3`.

2 Apartados de la práctica

Las mejoras y extensiones propuestas en esta práctica pueden enumerarse bajo la forma de apartados, tal y como se indica a continuación:

1. Mejoras en la introducción y escritura de las lambda expresiones:

   1.1. Reconocimiento de **expresiones multi-línea**. El objetivo es que una expresión se pueda introducir en varias líneas. Y dado que en este caso el salto de línea por sí solo ya no implicará necesariamente el final de una expresión, quizás sea útil introducir algún nuevo símbolo o símbolos para indicar ese aspecto (por ejemplo, un doble punto y coma).

   Esta mejora puede abordarse mediante la manipulación directa del *string* que teclea el usuario cuando introduce un nuevo término y/o mediante la introducción de nuevos tokens en el analizador léxico y de nuevas reglas gramaticales en el analizador sintáctico (siendo esta última estrategia la más elegante).

   Este apartado es obligatorio.

   1.2. Implementación de un **"*pretty-printer*" más completo**. El objetivo principal aquí es el de intentar minimizar el número de paréntesis a escribir cuando una expresión se transforma en un *string*. Quizás la mejor manera de hacerlo sea mediante una cascada de funciones (guiadas por la gramática del lenguaje), que se van llamando unas a otras según se va profundizando en la estructura interna de la expresión. Existen otros aspectos adicionales que también podrían ser considerados, como por ejemplo la indentación de los diferentes elementos de las expresiones, pero la minimización de paréntesis es el aspecto esencial.

   Este apartado es opcional y aporta hasta 1,5 puntos.

2. Ampliaciones del lenguaje lambda-cálculo:

   2.1. Incorporación de un **combinador de punto fijo interno**, de tal forma que se puedan declarar funciones mediante definiciones recursivas directas. La idea es que en lugar de escribir

```
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
```

   podamos escribir

```
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
```

   Con el fin de verificar más exhaustivamente el correcto comportamiento de esta nueva ampliación, escriba en el fichero `examples.txt` proporcionado y en la memoria final de la práctica tres ejemplos adicionales que involucren recursividad múltiple a partir de la suma: una lambda expresión que calcule el producto de dos números naturales (*prod*), otra que calcule el término n-ésimo de la serie de Fibonacci (*fib*), y otra que calcule el factorial de un número natural (*fact*).

   Este apartado es obligatorio.

   2.2. Incorporación de un **contexto de definiciones globales**, que permita asociar nombres de variables libres con valores o términos, de forma que éstos puedan ser utilizados en lambda expresiones posteriores. La sintaxis debe ser

```
identificador = término
```

   Por ejemplo:

```
x = true
id = lambda x : Bool. x
```

   De esta forma, si después escribimos

```
id x
```

   esta expresión debe ser válida y debe devolver `true`.

   Se recomienda reflexionar sobre cómo debe comportarse esta nueva funcionalidad si al realizar una definición se asocia con un nuevo valor un nombre que ya está en el contexto (es decir, se reutiliza un nombre que ya se utilizó en una definición previa). En función de cómo se maneje este fenómeno, podríamos estar ante un "contexto imperativo" o ante un "contexto funcional". Si estamos implementando un intérprete de lambda cálculo, lo más apropiado es que el contexto sea funcional. Y para ello existen diferentes alternativas de implementación. Se debe elegir una de ellas, explicando cómo se ha programado y explicando también la razón de dicha elección.

   Este apartado es obligatorio.

   2.3. Incorporación del **tipo `String`** para el soporte de cadenas de caracteres, así como de la operación de concatenación de estas cadenas.

   Este apartado es obligatorio.

   2.4. Incorporación de los **pares** (tuplas de dos elementos de cualquier tipo, incluso de tipos diferentes entre ellos), con las operaciones típicas de proyección *first* y *second*.

   Este apartado es obligatorio.

2.5. Incorporación de las **tuplas** de cualquier número de elementos, con las operaciones de proyección basadas en la posición de dichos elementos.

Este apartado es opcional, aporta hasta 1,5 puntos y su realización permite no acometer el apartado anterior (ya que estaría contenido en este, al ser los pares un caso particular de las tuplas).

2.6. Incorporación de los **registros** (secuencias finitas de campos de cualquier tipo etiquetados), con las operaciones típicas de proyección basadas en las etiquetas de los campos.

Este apartado es opcional y aporta hasta 1 punto.

2.7. Incorporación de las **listas** (secuencias finitas de elementos de un mismo tipo), con las operaciones típicas de obtener la cabeza, obtener la cola y consultar si es vacía. Adicionalmente, escriba en el fichero `examples.txt` proporcionado y en la memoria final de la práctica tres lambda expresiones: una que calcule recursivamente en función de la suma la longitud de una lista (*length*), otra que realice la concatenación de dos listas (*append*), y otra que realice la aplicación de una función sobre los elementos de una lista y devuelva la lista de los valores resultantes (*map*).

Este apartado es opcional y aporta hasta 1 punto.

2.8. Incorporación de **subtipado**. Más concretamente, se trataría de escribir una función que implemente el polimorfismo de subtipado para registros y funciones (es decir, una función que compruebe si dos tipos dados verifican dicha relación de subtipado), y se trataría también de re-implementar la función general de tipado de forma que use este tipo de polimorfismo allí donde sea aplicable. Piense en al menos dos lambda expresiones que involucren operaciones de subtipado, y escríbalas también en el fichero `examples.txt` y en la memoria final de la práctica.

Este apartado es opcional y aporta hasta 1 punto.

2.9. Incorporación del **tipo** `Unit`. Y como siempre, cada vez que se incorpora un tipo de dato, lo interesante es añadir también alguna operación que permita usar valores de ese tipo (por ejemplo, en el caso del tipo `String`, añadimos la concatenación). En el caso de `Unit`, podemos ver que el sumario de reglas propone como nuevas formas sintácticas válidas dos cosas: la propia constante `unit`, y la secuencia de expresiones separadas por punto y coma, de forma que `t1; t2` se traduce sintácticamente por `(lambda x:Unit. t2) t1` donde `x` tiene que ser un nombre que no pertenezca al conjunto de variables libres de `t2`.

Este apartado es opcional y aporta hasta 0,75 puntos.

2.10. Incorporación de **operaciones de entrada/salida**. El apartado anterior solo permitiría secuencias del estilo `unit; unit; ...; unit; t`. Si queremos más expresiones que involucren al tipo de dato `Unit`, lo más útil y directo sería la incorporación de operaciones de entrada/salida, al menos para de los tipos básicos `Nat` y `String`, como por ejemplo `print_nat`, `print_string`, `print_newline`, `read_nat` y `read_string`. Que vistas como funciones, serían de tipo `Nat -> Unit`, `String -> Unit`, `Unit -> Unit`, `Unit -> Nat` y `Unit -> String`, respectivamente. Pero no las implementaremos como funciones, sino como nuevos constructores del tipo `term`. O lo que es lo mismo, podemos suponer que son elementos que formarían parte del *kernel* de nuestro lenguaje (como lo serían las operaciones aritméticas, si en lugar de usar una abstracción de los números naturales en función del cero y de sus sucesores, estuviéramos usando los *ints* o los *floats* del *hardware*).

Y teniendo operaciones de entrada/salida, podremos escribir programas más versátiles. Por ejemplo, un programa que pida un número por teclado, calcule su factorial, y lo escriba por pantalla. Diseñe algún otro programa similar, cítelo y descríbalo en la memoria, e inclúyalo con el resto de ficheros de la práctica a la hora de entregarla. Procure, además, que dicho programa incluya varias expresiones (no solo una), y que al menos una de ellas haga uso explícito de una secuencia de sub-expresiones separadas por punto y coma.

De manera adicional, para ejecutar programas de este estilo, usando los mismos módulos que ya tenemos, y adaptando lo que sea necesario, además de generar un ejecutable `top` (para el

modo de trabajo interactivo, expresión a expresión), genere un nuevo ejecutable de nombre `run`, al que se le pueda pasar por línea de comandos un fichero cuyo contenido sea una o varias expresiones. Este ejecutable `run` debe compilar previamente todas las expresiones del fichero (con la función `typeof`) y seguidamente, si no hay errores, las evalúa, pero sin escribir el tipo y el resultado de cada una de ellas (como haría `top`), sino que los únicos efectos que observará el usuario durante la ejecución de este fichero serán los relacionados con las operaciones explícitas de entrada/salida.

En resumen, dispondríamos entonces de dos modos de trabajo o de dos tipos de compilador: uno expresión a expresión, y otro *batch* o por lotes. O bien, estableciendo una similutud con el lenguaje de programación que estamos utilizando, sería como tener por un lado el lazo interactivo `ocaml` y por otro la máquina virtual `ocamlrun`.

Este apartado es opcional y aporta hasta 0,75 puntos.

3. Redacción de una memoria final:

   3.1. Además de entregar el código fuente, debe entregarse también una **memoria** en formato pdf que contenga lo siguiente: un pequeño **manual de usuario** que ilustre (formalmente y con ejemplos de ejecución) las nuevas funcionalidades del intérprete, y también un pequeño **manual técnico** que indique qué módulos de las implementaciones originales han sido modificados y qué tipo de cambios se han realizado sobre ellos para lograr implementar esas nuevas funcionalidades (si bien este segundo manual puede ser sustituido por comentarios en el propio código fuente, siempre y cuando éstos sean suficientemente claros y cumplan la misión anteriormente indicada).

   Este apartado es obligatorio. Y dado que dentro de las competencias de esta asignatura figura el manejo de idiomas extranjeros, existe la opción de redactar la memoria final en inglés. Esta opcionalidad puede aportar hasta 1 punto adicional.

3 Instrucciones de entrega y evaluación

A continuación se enumeran las instrucciones de entrega y evaluación de esta práctica:

- La realización de los apartados obligatorios aporta hasta 5 puntos. Cuando decimos "hasta" nos referimos a que se podrá alcanzar esa cantidad de puntos siempre y cuando los apartados funcionen correctamente y sin anomalías. Si este aspecto no se cumple, se aplicarán penalizaciones. Esto mismo es aplicable a los apartados opcionales.

- De cara a la valoración de cada trabajo entregado, también se tendrá en cuenta, entre otros aspectos, la usabilidad del programa, así como la claridad y calidad (y no necesariamente la extensión) de los comentarios del código. Cabe aclarar, en lo que se refiere a los apartados opcionales, que su realización será valorada de acuerdo con las puntuaciones citadas en cada uno de ellos, aunque su realización no es estrictamente necesaria para aprobar la práctica (eso sí, con una nota máxima de 5 puntos, en caso de que no se realice ninguno).

- La **fecha límite de entrega** de la práctica es el viernes 9 de diciembre de 2022. Los trabajos presentados requerirán sus correspondientes **defensas** ante el profesor de prácticas durante las sesiones de los días 12 y 19, o bien 13 y 20, según el grupo, de diciembre de 2022.