Intérprete de cálculo lambda en OCaml

Cursos de Verano 2023: "Técnicas avanzadas de programación: el paradigma funcional''

Fernando Peña Bes

Agosto de 2023, Universidad de Zaragoza

Resumen

OCaml, y en especial sus tipos algebraicos, permiten representar y procesar de forma elegante los árboles sintácticos utilizados en la creación de lenguajes de programación. Por este motivo he decidido implementar un intérprete para uno de los lenguajes de programación más simples que existen, el cálculo lambda sin tipos. Es un sistema formal capaz de expresar computación utilizando únicamente la aplicación de funciones y es la base multitud de lenguajes de programación funcionales (entre ellos OCaml).

Introducción

El cálculo lambda fue desarrollado por Alonzo Church en los años 30 y es un modelo de computación universal, con la misma capacidad expresiva que una máquina de Turing. El lenguaje se construye de forma recursiva mediante las siguientes reglas:

- Una variable x es un término lambda.
- Si t es un término lambda y x es una variable, entonces $(\lambda x.t)$ es un término lambda llamado abstracción.
- Si t y s son términos lambda, entonces $(t\ s)$ es un término lambda llamado aplicación.

Una abstracción $(\lambda x.t)$ representa una función anónima con entrada x. Decimos que la variable x está *ligada* en el término t. Si una variable no aparece ligada en un término se llama *libre*. Una aplicación $(t\ s)$ representa la aplicación de de un término t (normalmente una abstracción) a un término s que actúa como argumento.

Estos términos se pueden transformar en otros nuevos mediante operaciones de reducción, que expresan la noción de computación:

- β-reducción. Consiste reducir una aplicación sustituyendo la variable ligada dentro cuerpo de la abstracción de la izquierda por el término de la derecha: $(\lambda x.t)s \to t[x := s]$.
- α-conversión. Consiste en renombrar las variables ligadas en una abstracción: $\lambda x.t[x] \to \lambda y.t[y]$. Esta reducción suele usar para evitar colisiones de nombres antes de realizar una β-reducción.

La β-reducción se puede repetir el número de veces necesario hasta que no queden aplicaciones por reducir. Si este proceso termina, el término que se alcanza se dice que está en *forma normal beta*.

Analizador sintáctico

Los términos del cálculo lambda tal y como se han definido anteriormente se pueden reconocer con una gramática muy sencilla:

```
lambda_term: NAME | function | application | "(" lambda_term ")"
function: "λ" NAME "." lambda_term
application: lambda_term lambda_term
```

Sin embargo, para facilitar la escritura de términos más complicados y reducir la cantidad de paréntesis se suelen utilizar las siguientes convenciones:

- Para representar abstracciones con varias entradas, una abstración puede contener varias variables separadas por espacios. Estas variables quedan ligadas en un mismo término. El término $\lambda x\ y.z$ es equivalente a $\lambda x.(\lambda y.z)$.

- El cuerpo de una abstracción se extiende lo máximo posible hacia la derecha. El término $\lambda x.\lambda y.z$ es equivalente a $\lambda x.(\lambda y.z)$.
- La aplicación es asociativa a la izquierda. El término $\lambda x.abc$ es equivalente a $\lambda x.(ab)c$.

Un ejemplo que incluye todas las reglas anteriores es $\lambda xy.xy\lambda z.x$, que equivale a $\lambda x.(\lambda y.(((xy)z)(\lambda z.x)))$.

Una posible gramática capaz de reconocer estos término es la siguiente:

```
lambda_term: abstraction | application
simple_term: NAME | "(" lambda_term ")"
abstraction: "λ" binders "." lambda_term
binders: NAME | binders NAME
application: simple_term | application simple_term | application abstraction
```

Esta es la gramática que he implementado para analizar la entrada del intérprete. He utilizado las bibliotecas `ocamllex` y `ocamlyacc` para implementar el *lexer* y el *parser* en los ficheros `lib/lexer.mll` y `lib/parser.mly` respectivamente.

El resultado del analizador sintáctico es un término lambda representado mediante un tipo algebraico en OCaml:

```
type term =
| Var of string
| Abs of string * term
| App of term * term
```

Implementación de β-reducción

La β-reducción es el reemplazo de una variable que está ligada en el cuerpo de una abstracción por un argumento. De forma general, el término $(\lambda x.t)s$ se reduce a $t[x := s]$. Este es el término t donde todas las instancias libres de x se han reemplazado por el término s. La β-reducción elimina las abstracciones generando un término concreto.

Cuando en un término contiene varias aplicaciones, existen diferentes estrategias de reducción según el orden el que se realicen las reducciones:

- En el orden normal un término se evalúa reduciendo primero la parte de la izquierda. Si el argumento contiene aplicaciones, estas no se evalúan hasta que es necesario.
- En el orden aplicativo las aplicaciones del argumento se evalúan antes de sustituirlo en la parte de la izquierda.

El orden aplicativo es más eficiente en el sentido de que las expresiones resultantes tras las reducciones son más pequeñas. Este es el orden de evaluación más usado en lenguajes de programación. Sin embargo, tiene la desventaja de que la evaluación puede no terminar si algún argumento contiene un bucle infinito. Por ejemplo, el término

$$(\lambda x\ y.x)(x)((\lambda x.(x\ x))(\lambda x.(x\ x)))$$

se reducirá a x al evaluarlo aplicando orden normal, pero la evaluación no terminará si se usa orden aplicativo ya que la evaluación de $(\lambda x.(x\ x))(\lambda x.(x\ x))$, llamado combinador ω, no termina. Hay lenguajes como Lisp que aunque tienen evaluación estricta permiten que el programador decida el orden de evaluación de los argumentos mediante la definición de macros.

El orden normal garantiza siempre se llega a la forma normal beta si es posible. Pero, de cualquier manera, el teorema de Church-Rosser indica que si un método de evaluación permite alcanzar la forma normal beta de un término, esta será la misma independientemente del orden de las reducciones.

Por desgracia, realizar una β-reducción no es tan sencillo como sustituir simplemente el argumento en el cuerpo de la abstracción, ya que es posible que se produzcan colisiones en los nombre de las variables. El caso problemático ocurre cuando una variable libre en el argumento pasa a estar ligada al sustituirse en el cuerpo. Por ejemplo, la reducción $(\lambda y.\lambda x.x\ y)x \to \lambda x.x\ x$ es incorrecta. La solución es realizar una α-reducción para renombrar la variable ligada en esa abstracción, por ejemplo $(\lambda y.\lambda a.a\ y)x \to \lambda a.a\ x$. Antes de realizar esta sustitución, es necesario asegurarse de que el nuevo nombre no esté presente entre las variables libres del argumento ni del cuerpo de la abstracción.

En el intérprete he implementado únicamente la evaluación en orden normal. Para que la secuencia de reducciones sea más fácil de entender, el argumento se sustituye directamente en el cuerpo de la abstracción de la izquierda. Una política de evaluación más eficiente consistiría en reducir al completo el cuerpo de la abstracción antes de sustituir el argumento. Los detalles de implementación de la β-reducción se pueden consultar la función `beta` del fichero `lib/term.ml`.

Comprobación de α-equivalencia

Dos términos lambda se consideran equivalentes si es posible renombrar las variables ligadas del primer término de forma que sea igual al segundo. Por ejemplo $\lambda x.x$ equivale a $\lambda y.y$, pero $\lambda x.ax$ no equivale a $\lambda y.by$. Este tipo de equivalencia se conoce como α-equivalencia. Al igual que en la β-reducción, renombrar las variables ligadas presenta ciertas dificultades, así que una estrategia diferente para comprobar la α-equivalencia usar una representación que no contenga nombres, como los índices de De Bruijn. Esta representación sustituye los nombres de las variables ligadas por números que indican el número de abstracciones que hay entre la abstracción donde se ha ligado esa variable y el lugar donde se encuentra.

Para tratar las variables libres se puede crear un diccionario que asigne el mismo número a las variables libres con el mismo nombre (empezando a partir del número máximo asociado a una variable ligada en el término), pero para no complicar demasiado la implementación he decidido mantener el nombre de las variables libres en la representación. Por ejemplo, el término $(\lambda x.x\lambda y.x\ y\ z)$ se representaría como $\lambda 1(\lambda 2\ 1\ z)$.

He implementado la transformación de un término a índices de De Bruijn mediante una función recursiva que utiliza una lista asociativa en la que se relaciona el nombre de cada variable con el número de abstracciones anidadas hasta llegar a la abstracción donde se define. Cada vez que se entra en una abstracción, se añade la variable ligada correspondiente con nivel 1 y se aumenta el nivel de todas las demás. Al encontrar una variable, se sustituye por su nivel almacenado la lista asociativa. Si no se encuentra en dicha lista, es una variable libre, por lo que se mantiene su nombre.

OCaml permite comparar la equivalencia estructural entre valores de tipos algebraicos, por lo que una vez que dos términos se encuentran representados con índices de De Bruijn, su α-equivalencia se puede comprobar comparando directamente ambos valores con el operador =.

La representación con índices de De Bruijn se puede utilizar en la β-reducción para evitar los problemas de colisión de nombres de variables ligadas, pero las reducciones son menos legibles.

Estructura del intérprete

La función básica del intérprete consiste en leer un término lambda desde la entrada estándar y evaluarlo hasta obtener su forma normal beta. Para facilitar la introducción de términos más complicados, he añadido la posibilidad de crear definiciones con la sintaxis `nombre := término`. La gramática del intérprete es la siguiente:

```
main: definition "\n" | lambda_term "\n"
definition: NAME ":=" lambda_term
```

Las definiciones se almacenan en un entorno (una lista asociativa) y se reemplazan en los siguientes términos lambda que se introducen antes de evaluarlos. El proceso de sustitución de constantes es una extensión del cálculo lambda básico y se llama δ-reducción. Durante la evaluación, de un término el intérprete muestra el resultado de la δ-reducción, la secuencia de β-reducciones durante la evaluación y finalmente, si el término resultado es α-equivalente a alguna de las definiciones del entorno, muestra los nombres de estas definiciones.

Ejecución

El intérprete se puede compilar y ejecutar con `dune` utilizando el comando

```
rlwrap dune exec lambda
```

Hay una serie de tests que se pueden lanzar con

```
dune test
```

Ejemplos

A continuación, se muestra una interacción con el intérprete en la que se definen los números `1` y `2` con la codificación de Church y la operación de suma para representar la expresión `1 + 1`, que es equivalente a `2`:

```
# sum := lambda m n f x. m f (n f x)
λ > (λm.(λn.(λf.(λx.((m f)((n f)x))))))
α > sum
# 1 := (lambda f x. f x)
λ > (λf.(λx.(f x)))
α > 1
# 2 := (lambda f x. f(f x))
λ > (λf.(λx.(f(f x))))
α > 2
# sum 1 1
δ > (((λm.(λn.(λf.(λx.((m f)((n f)x))))))(λf.(λx.(f x))))(λf.(λx.(f x))))
β > ((λn.(λf.(λx.(((λf.(λx.(f x)))f)((n f)x)))))(λf.(λx.(f x))))
β > (λf.(λx.(((λf.(λx.(f x)))f)(((λf.(λx.(f x)))f)x))))
β > (λf.(λx.((λx.(f x))((λx.(f x))x))))
β > (λf.(λx.(f((λx.(f x))x))))
β > (λf.(λx.(f(f x))))
λ > (λf.(λx.(f(f x))))
α > 2
```

En el entorno por defecto he añadido la definición del combinador Y, que permite que una abstracción se aplique a si misma para evaluar funciones recursivas como el factorial de un número:

```
# (Y G) 3
δ > (((λg.((λx.(g(x x)))(λx.(g(x x)))))(λr.(λn.((((λp.(λx.(λy.((p x)y))))((λn.
    ((n(λx.(λx.(λy.y))))(λx.(λy.x))))n))(λf.(λx.(f x))))(((λm.(λn.(λf.(m(n f))
    )))n)(r((λn.(λf.(λx.(((n(λg.(λh.(h(g f)))))(λu.x))(λu.u)))))n)))))))(λf.(λx.
    (f(f(f x))))))
β > (((λx.((λr.(λn.((((λp.(λx.(λy.((p x)y))))((λn.((n(λx.(λx.(λy.y))))(λx.(λy.x))))
    n))(λf.(λx.(f x))))))((λm.(λn.(λf.(m(n f)))))n)(r((λn.(λf.(λx.(((n(λg.(λh.(h(g f)
    ))))(λu.x))(λu.u)))))n))))))(x x)))(λx.((λr.(λn.(((  (λp.(λx.(λy.((p x)y))))))((λn.
    ((n(λx.(λx.(λy.y))))(λx.(λy.x))))n))(λf.(λx.(f x))))(((λm.(λn.(λf.(m(n f)))))n)
    (r((λn.(λf.(λx.(((n(λg.(λh.(h(g f)))))(λu.x))(λu.u)))))n))))))(x x))))(λf.(λx.
    (f(f(f x))))))
...
λ > (λf.(λx.(f(f(f(f(f(f x)))))))
α > 6
```

Siguientes pasos

Por falta de tiempo no he introducido tratamiento de errores en el analizador sintáctico. Si una expresión de entrada no está bien formada, el intérprete se detiene. Lo mismo ocurre al intentar salir del intérprete, el fin de la entrada produce un error en el analizador. En futuras iteraciones se podrían mejorar estos aspectos. También se podría dar la opción de controlar el nivel de información que muestra cada vez que se evalúa un término.

Fuentes

- Raúl Rojas. A tutorial Introduction to the Lambda Calculus, 2015
- Benjamin C. Pierce. Types and Programming Languages, 2002
- Wikipedia – Lambda calculus
- Ben Lynn – Lambda calculus implementation