Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Escuela de Ciencias Exactas y Naturales
Departamento de Ciencias de la Computación
R-422 — Compiladores

Apunte

La Representación Locally Nameless

1. El λ-cálculo con nombres (Fully Named)

La simplicidad del λ-cálculo viene, en parte, de que su única regla de computación es la β-reducción:

$$(\lambda x.P)Q \to P[Q/x]$$

Sin embargo, hay muchas cosas que esa regla no dice, en particular no dice nada acerca de cómo se comporta la sustitución. Primero, más trivialmente, la sustitución no reemplaza todas las ocurrencias de x en P, sino sólo aquellas que ocurran libres. Es decir, no substituye bajo otro binder para x (i.e. λx). Segundo, la sustitución debe evitar capturar variables libres. Si tenemos $P = \lambda y.x$, y $Q = y$, el reemplazo de x por y en P realizado de forma literal nos da $\lambda y.y$, donde y fue capturada (antes era libre y ahora no lo es)[1]. Para evitar esto (cuando trabajamos en papel) simplemente reemplazamos la variable y (en P) por alguna otra variable que no ocurra libre en Q, ya que consideramos, informalmente, que todos los términos α-equivalentes son iguales. Si leemos que "$(\lambda x.F(xx))(\lambda x.F(xx))$ es un punto fijo de F", naturalmente entendemos que x no ocurre libre en F, o que se renombra la variable de ligadura x si es así.

Cuando implementamos un programa que manipula λ-términos, no podemos ser tan laxos y debemos asegurar que la sustitución sustituye las variables correctas, sin ocasionar capturas. Lograr esto puede ser ineficiente, porque para chequear que no ocurran capturas se debe recorrer todo Q para conseguir sus variables libres.

Aunque algunos esfuerzos heroicos demuestran que *se puede* trabajar de esta manera [2], en general es una mala idea usar una representación con *fully named* para una implementación. Analizamos algunas alternativas.

2. Índices de de Bruijn

Una estrategia alternativa para representar nombres, debida a Nicolaas de Bruijn [3], es renunciar completamente al uso de nombres y representar las variables por su *índice de de Bruijn*, que es simplemente la "distancia" a la que está una determinada variable ligada de su binder. Los binders (en el caso del λ-cálculo sólo λ) no tienen ningún nombre asociado, ya que no lo necesitan[2]. Por ejemplo, el término *fully named* $M = (\lambda x.x(\lambda y.x(\lambda x.xx)))$ se representa como:

$$\lambda 0(\lambda 1(\lambda 00))$$

Un natural n representa a la variable que fue ligada "n binders atrás". Por ejemplo, el primer 0 representa a la variable que fue ligada en el λ que inmediatamente cubre esta ocurrencia, es decir el de más a la izquierda. El 1 indica que hay que "saltar" un λ para llegar al binder relevante, y también "apunta" al λ

[1] El λ-cálculo exhibe estos problemas en su forma más pura, pero ocurren siempre que hay variables ligadas. Por ejemplo, considere: $\left(\int_0^1 x\ dx\right) \times x = \left(\int_0^1 x^2\ dx\right)$.

[2] En implementaciones reales se suele almacenar un nombre para poder imprimir términos más agradables, pero sin ningún significado semántico.

de más a la izquierda. Notar como una misma variable ligada puede aparecer como dos índices distintos en un mismo término.

La gran ventaja de la notación de de Bruijn es que *identifica* (hace iguales) sintácticamente a los λ-términos α-equivalentes: si dos términos son α-equivalentes, su representación con índices de de Bruijn es sintácticamente igual.

El precio a pagar es que se torna más complicado generar términos, manipularlos programáticamente, y leerlos cuando los imprimimos. Al descender en el cuerpo de un λ-término (por ejemplo, en la implementación de un typechecker), algunas ocurrencias de variables empiezan a "escapar" al término (es decir que su índice "apunta" fuera del término mismo). Entonces si, por ejemplo, queremos escribir una función que implemente la η-expansión ($M \to (\lambda x.Mx)$), no alcanza con convertir M a $\lambda M0$, porque M puede tener índices que escapen (considerar $M = 0$, o $M = \lambda 2$). Para implementar correctamente la expansión, hay que "deslizar" o "correr" a todos los índices que escapen en M, mientras que en la representación con nombres, alcanzaría con tomar un x fresco.

En resumen, la notación con índices de de Bruijn nos da ventajas para la manipulación de términos con binders en forma automática al identificar los términos α-equivalentes, pero dificulta razonar con los mismos cuando debemos explorar el cuerpo de un binder y se aleja de nuestra forma de trabajar en papel.

Ejercicios

Ej. 1. ¿A qué términos fully named corresponden los siguientes términos con índices de de Bruijn?

a) $\lambda 0$

b) $\lambda 0 \lambda 1\ 0$

c) $\lambda\lambda\lambda 2\ 0\ 1$

d) $\lambda(\lambda 0)(\lambda 1)$

Ej. 2. Expresar los siguientes términos fully named con índices de de Bruijn.

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

b) $\lambda f.\, \lambda x.\, f\ x$

c) $\lambda x.\, x\ x$

d) $\lambda x.\, \lambda y.\, (\lambda z.\, x\ z)\ y$

3. La representación Locally Nameless (LN)

Una tercera opción, que combina las ventajas de ambas opciones previas, es usar nombres *sólo cuando convenga*, teniendo dos tipos ocurrencias de variables: los índices $(0, 1, ...)$ y los nombres locales. Los binders (λ) siguen sin tener un nombre asociado.

La idea general es que en la representación de un término t usaremos índices para las variables que estén ligadas en él, y nombres para las variables libres. Entonces, sólo consideraremos términos *localmente cerrados (LC)*, es decir aquellos cuyos índices no escapan a sus binders, pero sin restricción sobre sus nombres. Por ejemplo: x y $\lambda 0$ son LC, pero $(\lambda 1 z)$ y 0 no lo son.

Cuando queremos descender bajo un binder, por ejemplo en el cuerpo de λP, debemos primero "abrir" P para convertirlo en un término LC. El proceso de apertura reemplaza el índice 0 (que escaparía) por un nombre. Por ejemplo, abrir el cuerpo de M (de la §2) con el nombre z resulta en:

$$\mathsf{open}\ z\ (0(\lambda 1(\lambda 00))) = z(\lambda z(\lambda 00))$$

La operación $\mathsf{open}\ x\ t$ reemplaza todas las ocurrencias de la (única) variable que escapa en t por el nombre x. Para que esta operación sea correcta, x debe ser fresco en t, es decir no debe haber otra variable (nombrada) con el mismo nombre.Una de las ventajas de la representación LN es que sólo hay que tener

cuidado de tener una colisión con otros nombres: los índices no pueden colisionar. Si trabajamos con términos LC, los únicos nombres que pueden aparecer en t son los que fueron abiertos en él, entonces sólo debemos evitar abrir dos veces el mismo nombre en un término, lo cual no require recorrer el término.

Al tener nombres, es un poco más fácil manipular los términos. Por ejemplo, si queremos contar cuántas veces una abstracción λM usa su argumento, podemos abrir M con un nombre fresco x y contar las ocurrencias de x. No hace falta llevar ninguna cuenta de índices ni de profundidad.

Debido a que trabajamos con términos LC, la η-expansión de M sí puede implementarse como $\lambda M 0$. Ninguna variable ligada de M puede escapar a M, por lo cual este nuevo λ no las afecta, ni tampoco afecta a los nombres locales.

La sustitución también se simplifica. Por ejemplo, para la β-reducción de $(\lambda P)Q$ (siempre asumiendo que los términos son LC) podemos abrir P con un nombre fresco x, y luego reemplazar todas las ocurrencias de x por Q. Ninguna captura puede ocurrir, y no necesitamos deslizar ningún índice. Estas dos operaciones pueden incluso hacerse a la vez y evitar así la necesidad de un nombre fresco.

La operación inversa a open, llamada close, transforma un nombre x en un término t al índice 0. El resultado debe ser envuelto en un binder (λ) para recuperar un término LC. Por ejemplo, si tenemos el término:

$$Q = x(\lambda 0 x)$$

podemos cerrar x para obtener:

$$\mathsf{close}\ x\ Q = 0(\lambda 01)$$

y luego armar el término LC:

$$\lambda(\mathsf{close}\ x\ Q) = \lambda 0(\lambda 01)$$

Usando close podemos implementar una traducción de términos *fully named* a LN de manera muy directa:

$$\begin{array}{rcl} [\![x]\!] & = & x \\ [\![MN]\!] & = & [\![M]\!][\![N]\!] \\ [\![\lambda x.M]\!] & = & \lambda(\mathsf{close}\ x\ [\![M]\!]) \end{array}$$

Ejercicios

Ej. 3. ¿Cuál es el resultado de las siguientes aplicaciones de open y close?

a) open x 0

b) open x ($\lambda\lambda 2$ 0 1)

c) open x ($\lambda 0$)

d) open x ($0\lambda 0$ $1\lambda 1$ 2)

e) close x 0

f) close x (x ($\lambda 0$ x))

g) close x ($\lambda\lambda 0$ 1)

h) close x λ(close y (λy x))

4. Nuestra implementación

En el intérprete básico de FD4 comenzamos con una representación fully named al momento de parsear un término, pero luego los *elaboramos* a locally nameless. Los términos "superficiales" (fully named) están representados por un tipo `Lang.STm`. Los términos internos (LN) están representados por el tipo `Lang.Tm`. Ambos están parametrizados por 1) el tipo `var` de variables y 2) un tipo `info` que es usado para llevar "metadatos" en cada nodo del AST, e.g. un tipo o la posición fuente donde algo fue parseado. Los términos superficiales además están parametrizados por `ty`, un tipo que representa a los tipos de FD4. Ahora mismo, `STm` y `Tm` son prácticamente iguales. Luego, para implementar algunas extensiones, veremos la utilidad de haberlos separado.

Para los términos fully named (`Lang.STerm`), tomamos simplemente `var = Name = String`. Para los términos LN (`Lang.Term`), tomamos `var = Var`, un tipo que representa la unión disjunta de índices de de Bruijn, nombres locales (nombres que surgen de abrir un término), y nombres globales (nombres de definiciones top-level). La elaboración (`Elab.elab :: STerm -> Term`) es esencialmente la traducción $[\![-]\!]$ de arriba.

En el módulo `Subst` están las implementaciones de `open`, `close`, y `subst` (`subst P Q` sustituye el primer índice libre de P por Q, y no requiere un nombre fresco.). También van a encontrar implementadas `open2`, `close2` y `subst2`, que son las operaciones análogas para dos variables a la vez. Necesitamos estas variantes únicamente porque fix bindea a dos variables a la vez, y entonces debemos abrirlas y cerrarlas juntas para mantenernos en términos LC. Notar que en cada construcción de términos que liga variables (`Lam`, `Fix`, `Let`), el cuerpo usa un tipo específico (`Scope` o `Scope2`) para indicar la cantidad de variables libres. De esta forma, podemos evitar algunos errores estáticamente, dado que el tipo nos indica cuál función debemos usar para abrir/cerrar/sustituir.

¿No era que no había que contar índices...? Las implementaciones de las operaciones en `Subst` sí cuentan índices para saber qué índices sustituir, pero notar que no hay lógica para evitar las capturas: están imposibilitadas por la representación. Además, ¡esta complejidad queda totalmente *contenida* en este módulo! El resto del compilador usa nombres a medida que sea necesario, y siempre manipula términos LC. Por esto mismo, **fuera del módulo `Subst` podemos (y debemos) tratar a `Scope` y `Scope2` como tipos abstractos**, que sólo se manipulan con las operaciones exportadas por el módulo `Subst`.

Una trampa. En algunos módulos del compilador, hacemos open x t sin chequear que x haya sido fresca para el término t. Esto puede hacerse si sabemos con certeza que t es un término elaborado con la traducción de arriba, dado que open x (close x t) $= t$, sin necesidad de frescura. Es importante sólo hacer esto con términos que fueron elaborados: si t es el resultado de una normalización o cualquier otra transformación, esto puede ser incorrecto. El typechecker usa esta trampa, porque opera sobre los términos escritos en el programa (parseados y elaborados), pero el pretty-printer no puede usarla, porque debe imprimir correctamente términos que fueron normalizados y sustituidos.

Para más información: Una descripción mucho más detallada de la representación LN puede encontrarse en [1]. Una implementación en Haskell de esta representación fue descrita en [4].

Referencias

[1] Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 49:363–408, 10 2012.

[2] Ernesto Copello, Álvaro Tasistro, Nora Szasz, Ana Bove, and Maribel Fernández. Alpha-structural induction and recursion for the lambda calculus in constructive type theory. *Electronic Notes in Theoretical Computer Science*, 323:109–124, 07 2016.

[3] N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381 – 392, 1972.

[4] Conor McBride and James McKinna. Functional Pearl: I am not a Number-I am a Free Variable. In Henrik Nilsson, editor, *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2004, Snowbird, UT, USA, September 22-22, 2004*, pages 1–9. ACM, 2004.