Hoja 1 | Eric Brandwein

1. La semántica del operador Fix original es

$$\text{Fix}(\lambda x:\sigma.M) \to M\{x \leftarrow \text{Fix}(\lambda x:\sigma.M)\} \quad (\text{E-FixBeta})$$

si renombramos a $(\lambda x:\sigma.M)$ como V, nos queda

$$\text{Fix } V \to M\{x \leftarrow \text{Fix } V\}$$

En este caso, $M\{x \leftarrow \text{Fix } V\}$ puede llegar a ser un valor, ya que podría ser, por ejemplo, de la forma $(\lambda y:\tau.(\text{Fix } V)\ y)$.
En cambio, por la semántica operacional de la aplicación, New-Fix nunca puede llegar a ser un valor.
La regla E-App2 es

$$\frac{M_2 \to M_2'}{\underbrace{(\lambda x:\sigma.M)}_{\text{valor } V} M_2 \to (\lambda x:\sigma.M) M_2'}$$

Por lo tanto, New-Fix reduce de la siguiente manera:

$$\text{Fix } V \to V(\text{Fix } V) \to V(V(\text{Fix } V)) \to \dots$$

, ya que $(\text{Fix } V)$ no es un valor y V sí lo es. No sería correcto entonces redefinir la semántica del operador Fix de esta manera.

2. Tomemos el ejemplo

$$Z = \text{if } \underbrace{x\ \emptyset}_{U} \text{ then } \underbrace{x}_{V} \text{ else } \underbrace{2}_{W}$$? 2

El algoritmo de inferencia modificado tendría la sustitución

$$S = MGU\{\sigma \doteq Nat,\ \rho \doteq Bool\}$$

con $W(U) = \Gamma_1 \triangleright M:\rho$ y $W(V) = \Gamma_2 \triangleright P:\sigma$.
$W(x\ \emptyset)$ es igual a

$\{x: Nat \to s\} \triangleright x\ \emptyset : s$, con s una variable fresca.

$W(x)$ es igual a

$\{x:t\} \triangleright x:t$, con t una variable fresca

Nuestro S entonces podrá ser algo como

$$S = MGU\{t \doteq Nat,\ s \doteq Bool\} = \{t \leftarrow Nat,\ s \leftarrow Bool\}$$

$W(Z)$ entonces resultará en

$$S\Gamma_1 \cup S\Gamma_2 \cup S\Gamma_3 \triangleright S(\text{if } x\ 0 \text{ then } x \text{ else } 2) : St =$$

$$= \underbrace{\{x : Nat \to Bool\} \cup \{x : Nat\}}_{\text{esto no está bien formado}} \cup \emptyset \triangleright \text{if } x\ 0 \text{ then } x \text{ else } 2 : Nat$$

Podemos ver que $S\Gamma_1$ y $S\Gamma_2$ tienen definiciones contradictorias para el tipo de x, y por lo tanto el algoritmo es incorrecto.

3. ~~[illegible]~~ Tomemos como ejemplo a

$$\cancel{\lambda x : Float}\ Z = \overbrace{(\lambda x : Nat \to Nat.\ succ(x\ 0))}^{M}\ \overbrace{(\lambda y : Float.\ 2{,}5)}^{N}$$

M es de tipo $(Nat \to Nat) \to Nat$, y N de tipo $Float \to Float$.
Bajo la nueva regla S-Func, $Float \to Float <: Nat \to Nat$, y entonces el término Z debería tipar. Sin embargo, Z reduce a

$$succ((\lambda y : Float.\ 2{,}5)\ 0) \to succ(2{,}5)$$

que, como $2{,}5$ ~~[illegible]~~ es de tipo $Float$, y $Float \not<: Nat$, no tipa.

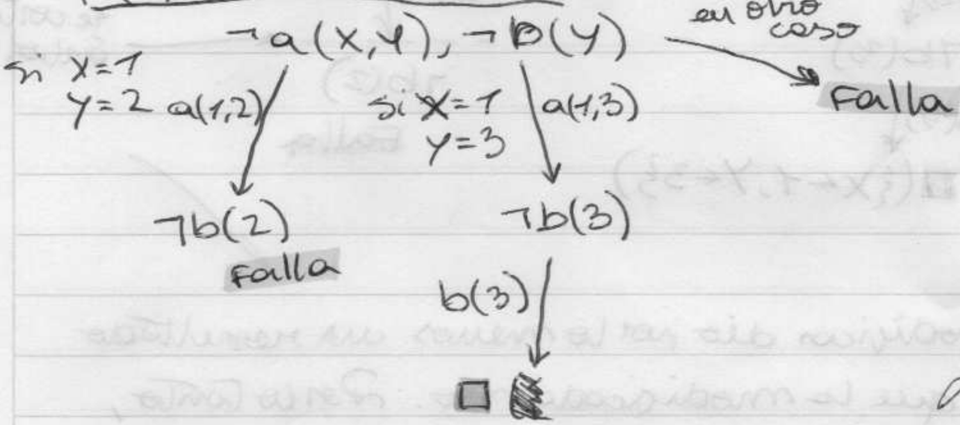
4. a) Para toda consulta ground las variables X e Y estarán ya instanciadas. ~~[illegible]~~ Por lo tanto, X e Y podrán unificar solamente con sus propios valores. Tanto en el programa original como en el modificado, si sus valores son 1 y 3, el predicado será verdadero, y en otro caso será falso.

~~p(X,Y) :- a(X,Y), b(Y)~~
~~si X=1, Y=2: a(1,2)~~
~~:- b(2)~~
~~sino: falla~~

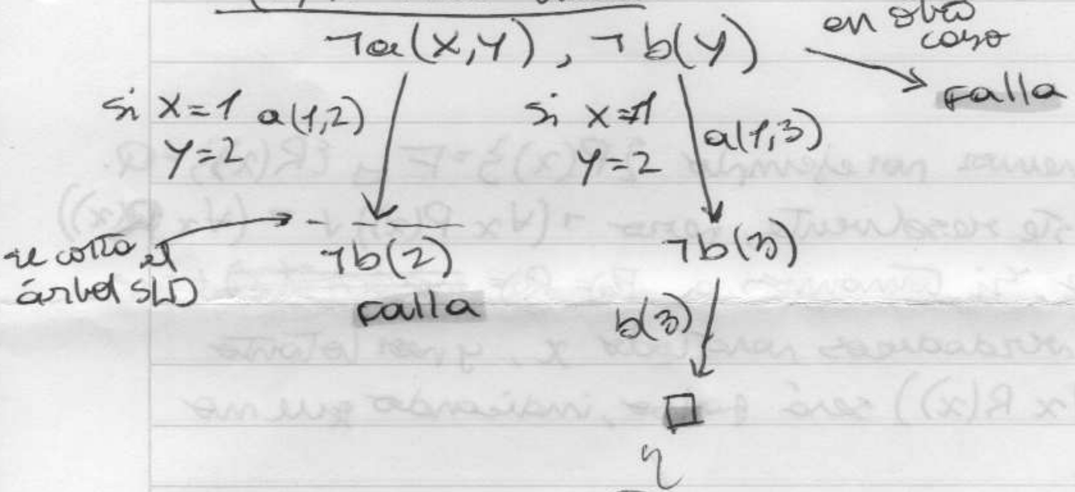
A continuación veremos los árboles SLD de todos casos para ver que llegan a los mismos resultados.

Hoja 2 Eric Brandwein

p(X,Y) sin modificar



P(X,Y) modificado



Arriba se muestran (3) árboles SLD para cada variación del programa, agrupados por conveniencia. Vemos que en todos los casos el resultado es el mismo, y por lo tanto el conjunto de soluciones no se altera.

b) En este caso sí se alteran los resultados. Veamos los árboles SLD.

p(X,Y) sin modificar

$\neg a(X,Y), \neg b(Y)$

- $a(1,2)$: $\neg b(2)$ — falla
- $a(1,3)$: $\neg b(3)$ → $b(3)$ → $\square(\{X \leftarrow 1, Y \leftarrow 3\})$

p(X,Y) modificado

$\neg a(X,Y), \neg b(Y)$

- $a(1,2)$: - - - - ← se corta el árbol SLD
- $\neg b(2)$ — Falla

La variante sin modificar dió por lo menos un resultado positivo, mientras que la modificada no. Por lo tanto, ?-not(p(X,Y)) será fail para el primer caso, pero success para el segundo.

5. a) Falso. Tomemos por ejemplo $\{P(x)\} = F$ y $\{R(x)\} = Q$. Entre ellas no existe resolvente, pero $\neg(\forall x\, P(x)) \vee \neg(\forall x\, R(x))$ no es una tautología. Si tomamos a P(x) = R(x) = ~~mayorA2(x)~~ true, P(x) y R(x) serán verdaderos para todo x, y por lo tanto $\neg(\forall x\, P(x)) \vee \neg(\forall x\, R(x))$ será falso, indicando que no es una tautología.

b) Falso. PROLOG tiene como regla de selección la de seleccionar el átomo de más a la izquierda. El árbol que recorrerá PROLOG ~~pasando~~ llegará primero del goal $\neg p(X,Y), \neg P(Y,Z)$ al goal $\neg q(Y), \neg P(Y,Z)$. De ahí, basándose en lo definido sobre q, unificará Y con un valor, y llegará al goal $\neg P(Y,Z)$. Así, vemos que nunca llegará al goal $\neg q(Y), \neg q(Z)$.

~~$\neg p(X,Y), \neg P(Y,Z)$~~

~~p(X,Y) :- q(Y)~~

Hoja 3 Eric Bronzwein

7. ➜ Prototipado. En JavaScript, todo objeto tiene un atributo __proto__ que determina su prototipo. En el cálculo de objetos, en cambio, no se tienen prototipos, y se usan clases para evitar repetir definiciones.

➜ Funciones Extraibles: En JavaScript, ~~las funciones~~ los métodos de un objeto se pueden extraer del mismo y usarse en otro contexto. En el cálculo de objetos esto no es posible.

✓

➜ Method Dispatch. En Javascript, la selección del método a utilizar no está guiado completamente por la sintaxis, sino que se decide en tiempo de ejecución, basándose en el objeto receptor y toda su familia de prototipos. Esto se llama Method Dispatch Dinámico. En cambio, la sintaxis del cálculo de objetos define inequívocamente a qué método se está llamando. Esto se llama Method Dispatch Estático.

✓

➜ Redefinición de Métodos. En JavaScript, sobreescribir los métodos de un objeto implica cambiar ese objeto en el lugar, tal que futuros usos de ese objeto tendrán en cuenta el nuevo valor. En cambio, ~~sólo~~ asignar un nuevo valor a un método de un objeto del cálculo de objetos genera un nuevo objeto, igual al anterior, pero con el método modificado. Para lograr lo mismo en JavaScript, se debe primero clonar el objeto en cuestión, y luego cambiar su método.

los atributos son dinámicos en JS