SEMÁNTICA DE CONTINUACIONES O INVERSA

Como vimos, sucesivas extensiones al lenguaje forzaron redefiniciones de f_*, f_♣ y f_†. Esto se debe a que (en el caso de f_*) la semántica de la secuencia está definida así:

```
[[c0;c1]]s = [[c1]]_*([[c0]]s)
```

que es en cierta forma artificial, ya que [[c1]] parece estar aplicada al resultado de [[c0]] y por lo tanto hay que forzar la propagación de lo que [[c0]] produce, con la única excepción del estado final de [[c0]] que es en realidad lo único que [[c1]] debería recibir.

Para evitar esto se propone otra forma de definir la semántica: denotemos a esta nueva semántica {{c}} en vez de [[c]] que va a seguir denotando la semántica que hemos estado viendo (se llama semántica directa). La idea de {{c}} es que en vez de devolver el estado final, dice "decime qué querés hacer con el estado final y yo lo hago por vos". En el caso del lenguaje imperativo simple (sin ninguna de las extensiones) teníamos

```
[[_]] ∈ <comm> -> Σ -> Σ_⊥
```

La frase "decime qué querés hacer con el estado final y yo lo hago por vos" se podría escribir así:

```
(*)  {{c}}ks = k_⊥⊥ ([[c]]s)
```

donde k es "lo que se quiere hacer con el estado final". La ecuación muestra que {{c}} recibe "lo que se quiere hacer con el estado final", y también recibe el estado inicial s. No devuelve el estado final [[c]]s sino que le aplica k a dicho resultado.

En la ecuación (*) se ve que {{_}} recibe 3 argumentos en vez de 2 que recibe [[_]]. El argumento nuevo, k, es "lo que se quiere hacer con el estado final", se suele llamar "continuación". Como se ve en (*), k debe poder aplicarse a un estado (al estado final):

```
{{_}} ∈ <comm> -> (Σ -> X_⊥) -> Σ -> X_⊥
```

donde A puede ser cualquier cosa, se agrega ⊥ ya que en (*) se ve que el resultado final puede ser ⊥ (k_⊥⊥ propaga ⊥ y por lo tanto puede devolver ⊥). Como se ve, las continuaciones son funciones ∈ Σ -> X_⊥.

No debe interpretarse (*) como una definición de {{_}}, sólo es una propiedad que se desea que tenga. La intención es justamente definir {{_}} desde cero, a través de ecuaciones, como hemos definido [[_]]. El propósito es resolver las molestias causadas por ecuaciones como

```
[[c0;c1]]s = [[c1]]_*([[c0]]s)
```

que fuerzan a propagar siempre el comportamiento de [[c0]], salvo su estado final.

Definamos entonces {{_}} teniendo en cuenta que recibe una continuación k, y que no devuelve el estado final sino que aplica k a dicho estado:

```
{{skip}}ks = k s
{{v:=e}}ks = k [s | v : [[e]]s]
                                          ⎧{{c0}}ks        si [[b]]s
{{if b then c0 else c1}}ks = ⎨
                                          ⎩{{c1}}ks        c.c.
{{c0;c1}}ks = {{c0}}({{c1}}k)s
```

Seguramente ésta es la ecuación más difícil de entender: {{c0}} tiene 2 argumentos, el primero es {{c1}}k y el segundo es el estado inicial s. Primero que nada habría que confirmar que esto está bien escrito, que {{c1}}k es una continuación (si no lo fuera no podría ser el primer argumento de {{c0}}). Efectivamente, por el tipo de {{_}}, {{c1}} recibe una continuación y devuelve una continuación, o sea que, aplicado a k es una continuación. Tenemos entonces que {{c1}}k es una continuación.

Lo que la ecuación dice, entonces, es dado que k es "lo que se quiere hacer con el estado final de c0;c1", se evalúa c0 y se le pasa como argumento "lo que se quiere hacer con el estado final de c0". ¿Y qué se quiere hacer con el estado final de c0? Se lo quiere usar para evaluar c1. Eso explica que la continuación de {{c0}} sea de la forma {{c1}}... Para explicar que el argumento de {{c1}} sea k, hay que preguntarse "qué va a querer hacerse con el estado final de c1". ¿Y qué va a querer hacerse con él? Justamente ése va a ser el estado final de c0;c1 y habíamos dicho que lo que quería hacerse con él estaba dado por k.

Para terminar de convencerse uno podría intentar demostrar la propiedad (*) por inducción en la estructura de c:

```
c = skip
{{skip}}ks = k s = k_⊥⊥ s = k_⊥⊥ ([[skip]]s)

c = v:= e
{{v:= e}}ks = k [s | v : [[e]]s] = k_⊥⊥ [s | v : [[e]]s] = k_⊥⊥ ([[v:= e]]s)

c = if b then c0 else c1, caso [[b]]s verdadero
{{if b then c0 else c1}}ks = {{c0}}ks
                           = k_⊥⊥ ([[c0]]s)   por hipótesis inductiva
                           = k_⊥⊥ ([[if b then c0 else c1]]s)
caso [[b]]s falso es similar

c = c0;c1
primero, observar que la propiedad (*) también se puede escribir {{c}}k = k_⊥⊥ . [[c]] donde . es
composición de funciones.

{{c0;c1}}ks = {{c0}}({{c1}}k)s
            = ({{c1}}k)_⊥⊥ ([[c0]]s)              por hipótesis inductiva en c0
            = (k_⊥⊥ . [[c1]])_⊥⊥ ([[c0]]s)       por hipótesis inductiva en c1
            = (k_⊥⊥ . [[c1]]_⊥⊥) ([[c0]]s)       por (f_⊥⊥ . g)_⊥⊥ = f_⊥⊥ . g_⊥⊥
            = k_⊥⊥ ([[c1]]_⊥⊥([[c0]]s))
            = k_⊥⊥ ([[c0;c1]]s)

Esto termina de demostrar la propiedad (*) para skip, asignación, condicional y secuencia.

La propiedad (f_⊥⊥ . g)_⊥⊥ = f_⊥⊥ . g_⊥⊥ que utilizamos puede comprobarse así:

Sean

f ∈ Q -> D
g ∈ P -> Q_⊥

entonces

f_⊥⊥ ∈ Q_⊥ -> D
f_⊥⊥ . g ∈ P -> D
(f_⊥⊥ . g)_⊥⊥ ∈ P_⊥ -> D
g_⊥⊥ ∈ P_⊥ -> Q_⊥
f_⊥⊥ . g_⊥⊥ ∈ P_⊥ -> D

Para comparar (f_⊥⊥ . g)_⊥⊥ y f_⊥⊥ . g_⊥⊥ lo hacemos punto a punto. Sea x ∈ P_⊥. Si x = ⊥, entonces

(f_⊥⊥ . g)_⊥⊥ x = (f_⊥⊥ . g)_⊥⊥ ⊥
                = ⊥
                = f_⊥⊥ ⊥
                = f_⊥⊥ (g_⊥⊥ ⊥)
                = (f_⊥⊥ . g_⊥⊥) ⊥
                = (f_⊥⊥ . g_⊥⊥) x

Si x ≠ ⊥

(f_⊥⊥ . g)_⊥⊥ x = (f_⊥⊥ . g) x
                = f_⊥⊥ (g x)
                = f_⊥⊥ (g_⊥⊥ x)
                = (f_⊥⊥ . g_⊥⊥) x

Continuamos con la definición de {{_}} para los casos restantes: while y newvar.

{{while b do c}}ks = {{if b then c;while b do c else skip}}ks
                   = ⎧{{c;while b do c}}ks         si [[b]]s
                   = ⎩{{skip}}ks                   c.c.
                   = ⎧{{c}}({{while b do c}}k)s    si [[b]]s
                   = ⎩k s                          c.c.

por lo tanto

{{while b do c}}k = w

donde
```

```
          ⎧{{c}}ws    si [[b]]s
w s = ⎨
          ⎩k s        c.c.
```

o, como reformulamos en similares situaciones,

{{while b do c}}k = Y_{Σ -> X_⊥} F

donde

```
              ⎧{{c}}ws    si [[b]]s
F w s = ⎨
              ⎩k s        c.c.
```

Finalmente,

{{newvar v:= e in c}}ks = {{c}}(λs'∈Σ.k [s'|v:sv])[s|v:[[e]]s]

Se asume que k es lo que se quiere hacer con el estado final del newvar. El estado en que se evalúa c es [s|v:[[e]]s], es decir, uno en el que v ya se inicializó. El argumento (λs'∈Σ.k [s'|v:sv]) dice "qué se quiere hacer con el estado final de c". Lo que se quiere hacer es dárselo a k, salvo que k debe recibir el estado final de todo el newvar, es decir, sin el valor local de v. Por ello se restaura el valor global antes de dárselo a k.

NOTACION

Al dar la semántica de continuaciones, es habitual suprimir mencionar el estado en las ecuaciones en que no es necesario. En este caso solo se puede suprimier en skip y secuencia:

{{skip}}k = k
{{c0;c1}}k = {{c0}}({{c1}}k)

FALLAS

Para dar la semántica de continuaciones para un lenguaje que tiene fallas, hacen falta 2 continuaciones: una es la que dice "lo que se quiere hacer con el estado final de c si c termina bien" y otra que dice "lo que se quiere hacer con el estado final de c si c termina mal".

{{_}} ∈ <comm> -> (Σ -> X_⊥) -> (Σ -> X_⊥) -> Σ -> X_⊥

{{skip}}kjs = k s
{{fail}}kjs = j s
{{v:=e}}kjs = k [s | v : [[e]]s]

```
                                          ⎧{{c0}}kjs         si [[b]]s
{{if b then c0 else c1}}kjs = ⎨
                                          ⎩{{c1}}kjs         c.c.
```

{{c0;c1}}kjs = {{c0}}({{c1}}kj)js

Esta ecuación es interesante, dice que la continuación en caso de que termine mal c0;c1, o c0 o c1 es la misma. Es lógico ya que si c0 termina mal, c0;c1 termina mal; y si c1 termina mal, también.

{{catchin c0 with c1}}kjs = {{c0}}k({{c1}}kj)s

En el caso del catchin los roles de k y j se invierten. La dualidad entre skip y ; por un lado y fail y catchin por el otro, es notoria.

{{while b do c}}kj = Y_{Σ -> X_⊥} F

donde

```
              ⎧{{c}}wjs    si [[b]]s
F w s = ⎨
              ⎩k s         c.c.
```

En el caso del newvar, el valor global de la variable debe restaurarse en ambos casos: termine bien o mal:

{{newvar v:= e in c}}kjs = {{c}}(λs'∈Σ.k [s'|v:sv])(λs'∈Σ.j [s'|v:sv])[s|v:[[e]]s]

ENTRADA-SALIDA

Para agregar entrada y salida, conviene determinar el tipo X_⊥ que hasta ahora no requirió definición concreta: será el dominio Ω definido en clases anteriores por el isomorfismo

Ω ≈ (Σ' + Z x Ω + (Z -> Ω))_⊥

La semántica tendrá ahora el siguiente tipo

{{_}} ∈ <comm> -> (Σ -> Ω) -> (Σ -> Ω) -> Σ -> Ω

Todas las ecuaciones que hemos dado siguen intactas (de hecho estaban definidas para cualquier X_⊥). Sea agregan las correspondientes a entrada y salida:

{{!e}}kjs = Lout <[[e]]s, k s>
{{?v}}kjs = Lin (λn∈Z.k [s | v : n])

NOTACION

Como antes, a veces pueden suprimirse los estados en las ecuaciones:

{{skip}}kj = k
{{fail}}kj = j
{{c0;c1}}kj = {{c0}}({{c1}}kj)j
{{catchin c0 with c1}}kj = {{c0}}k({{c1}}kj)

RELACION CON LA SEMÁNTICA DIRECTA

La semántica directa puede obtenerse a partir de la semántica de continuaciones pasando continuaciones que "no hacen nada con el estado final":

[[c]]s = {{c}} Lterm Labort s

Lterm y Labort son continuaciones que no hacen nada con el estado final de c salvo meterlo en Ω. Observar que Lterm y Labort pertenecen a Σ -> Ω, o sea que son continuaciones.

Se puede formular una propiedad similar a la que se mencionó al comienzo (*):

(**) {{c}}kjs = (k,j)_* ([[c]]s)

donde (k,j)_* es la única función continua de Ω -> Ω que satisface:

$$(k,j)_* x = \begin{cases} \bot_\Omega & \text{si } x = \bot_\Omega \\ k\ s & \text{si } x = \text{Lterm } s \\ j\ s & \text{si } x = \text{Labort } s \\ \text{Lout } \langle n,(k,j)_*\ w\rangle & \text{si } x = \text{Lout } \langle n,w\rangle \\ \text{Lin } ((k,j)_* \cdot g) & \text{si } x = \text{Lin } g \end{cases}$$

Observar que los únicos casos interesantes son el 2do y 3ro, los demás no hacen más que propagar. Por eso, si k = Lterm y j = Labort, (k,j)_* resulta la identidad. Por lo tanto, (**) en este caso dice

{{c}} Lterm Labort s = (Lterm,Labort)_* ([[c]]s)
= [[c]]s

obteniéndose la semántica directa a partir de la de continuaciones.