SEMÁNTICA DE CONTINUACIONES O INVERSA

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]

$$\{\{\text{if b then c0 else c1}\}\}kjs = \begin{cases} \{\{c0\}\}kjs & \text{si } [[b]]s \\ \{\{c1\}\}kjs & \text{c.c.} \end{cases}$$

{{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

$$F\ w\ s = \begin{cases} \{\{c\}\}wjs & \text{si } [[b]]s \\ k\ s & \text{c.c.} \end{cases}$$

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.

FALLAS CON RÓTULOS

En el parcial se pidió modificar la semántica denotacional (directa) de modo de incluir los comandos

<comm> ::= fail <label> | catch <label> in <comm> with <comm>

donde <label> puede ser lo mismo que <var>.

Para ello, en vez de

$$\Sigma' = \Sigma \cup \{abort\} \times \Sigma$$

se define

$$\Sigma' = \Sigma \cup \langle label \rangle \times \Sigma$$

Tras lo cual las ecuaciones se reescriben como sigue:

[[skip]]s = s
[[fail l]]s = <l, s>
[[v:= e]]s = [s | v : [[e]]s]

$$[[\text{if b then c0 else c1}]]s = \begin{cases} [[c0]]s & \text{si } [[b]]s \\ [[c1]]s & \text{c.c.} \end{cases}$$

[[c0;c1]]s = [[c1]]_* ([[c0]]s)
donde, dada una f ∈ Σ -> Σ'_⊥, denotamos por f_* la siguiente extensión de f a Σ'_⊥. Así, f_* ∈ Σ'_⊥ -> Σ'_⊥ definida por:

$$f_* x = \begin{cases} f\ x & \text{si } x \in \Sigma \\ x & \text{c.c.} \end{cases}$$

[[catch l in c0 with c1]]s = [[c1]]_♣ ([[c0]]s)

donde, dada una f ∈ Σ -> Σ'_⊥, denotamos por f_♣ la siguiente extensión de f a Σ'_⊥. Así, f_♣ ∈ Σ'_⊥ -> Σ'_⊥ definida por:

$$f_\clubsuit\ x = \begin{cases} f\ s & \text{si } x = \langle l, s\rangle \\ \langle l', s\rangle & \text{si } x = \langle l', s\rangle \text{ con } l' \neq l \\ x & \text{si } x \in \Sigma \\ \bot & \text{si } x = \bot \end{cases}$$

que puede simplificarse

$$f_\clubsuit\ x = \begin{cases} f\ s & \text{si } x = \langle l, s\rangle \\ x & \text{c.c.} \end{cases}$$

[[while b do c]] = sup'(F^i ⊥')

$$\text{donde } F\ w\ s = \begin{cases} w_*\ ([[c]]s) & \text{si } [[b]]s \\ s & \text{c.c.} \end{cases}$$

[[newvar v:= e in c]]s = (λs'∈Σ. [s' | v : s v])_† ([[c]][s | v : [[e]]s])
donde, dada una f ∈ Σ -> Σ, denotamos por f_† la siguiente extensión de f a Σ'_⊥. Así, f_† ∈ Σ'_⊥ -> Σ'_⊥ definida por:

$$f_\dagger\ x = \begin{cases} \langle l, f\ s\rangle & \text{si } x = \langle l, s\rangle \\ f\ x & \text{si } x \in \Sigma \\ \bot & \text{si } x = \bot \end{cases}$$

CONTINUACIONES

La misma extensión puede hacerse usando continuaciones. Primero cambiamos el tipo de la segunda continuación de modo de que reciba también el rótulo:

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

En realidad, es como si recibiera una continuación para cada rótulo.

{{skip}}kjs = k s
{{fail l}}kjs = j l s

Acá hemos seleccionado la continuación correspondiente al rótulo l.

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

$$\{\{\text{if b then c0 else c1}\}\}kjs = \begin{cases} \{\{c0\}\}kjs & \text{si } [[b]]s \\ \{\{c1\}\}kjs & \text{c.c.} \end{cases}$$

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

{{catch l in c0 with c1}}kjs = {{c0}}k[j|l:{{c1}}kj]s

dice que si c0 falla con rótulo l, entonces se ejecuta c1, pero si falla con otro rótulo hay que utilizar la continuación correspondiente a ese rótulo.

{{while b do c}}kj = Y_{Σ -> X_⊥} F

donde

$$F\ w\ s = \begin{cases} \{\{c\}\}wjs & \text{si } [[b]]s \\ k\ s & \text{c.c.} \end{cases}$$

En el caso del newvar, el valor global de la variable debe restaurarse en todos los casos: para todas las posibles formas de terminal mal (es decir, cualquiera sea el rótulo con el que termine mal)

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

Recordemos que los estados pueden omitirse en algunas ecuaciones:

```
{{skip}}kj = k
{{fail l}}kj = j l
{{c0;c1}}kj = {{c0}}({{c1}}kj)j
{{catch l in c0 with c1}}kj = {{c0}}k[j|l:{{c1}}kj]
{{while b do c}}kj = Y_{Σ -> X_⊥} F
```

donde

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

BREAK Y CONTINUE

Para notar el poder expresivo de la semántica de continuaciones, extendemos el lenguaje con dos nuevos comandos:

```
<comm> ::= break | continue
```

Cuando se ejecuta dentro de un ciclo, el comando break termina la ejecución del mismo. El comando continue, en cambio, termina con la iteración actual, pero vuelve a chequear la condición del ciclo y si se cumple, continúa ejecutando las subsiguientes iteraciones. No es importante señalar qué ocurre cuando estos comandos se ejecutan fuera de un ciclo, podría considerarse que el programa falla en ese caso.

Una manera de interpretar esta extensión es agregando dos continuaciones más: una para el break y otra para el continue

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

{{skip}}kjih = k
{{fail l}}kjih = j l
{{break}}kjih = i
{{continue}}kjih = h
{{v:=e}}kjihs = k [s | v : [[e]]s]
                                ⎧{{c0}}kjihs         si [[b]]s
{{if b then c0 else c1}}kjihs = ⎨
                                ⎩{{c1}}kjihs         c.c.
{{c0;c1}}kjih = {{c0}}({{c1}}kjih)jih
{{catch l in c0 with c1}}kjih = {{c0}}k[j|l:{{c1}}kjih]ih

{{while b do c}}kjih = Y_{Σ -> X_⊥} F
```

donde

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

Observar que acá se pone como continuación k, en caso de que haya un break (k es la continuación que espera el estado final del while, poner k como continuación asociada al break equivale dar por terminado el while cuando se encuentra con un break). Observar tmabién que se pone w en caso de que haya un continue, eso provocaría continuar con el while en caso de que encuentre el comando continue.

```
{{newvar v:= e in c}}kjihs = {{c}}(λs'∈Σ.k [s'|v:sv])
                                  (λl∈<label>.λs'∈Σ.j l [s'|v:sv])
                                  (λs'∈Σ.i [s'|v:sv])
                                  (λs'∈Σ.h [s'|v:sv])
                                  [s|v:[[e]]s]
```

Otra forma de resolverlo, para evitar esta explosión de continuaciones, sería juntar todas las continuaciones en una sola continuación:

```
Cont = (Σ -> X_⊥)
Conts = Index -> Cont
```

Index = {Ok, Break, Continue} + <label>

{{_}} ∈ <comm> -> Conts -> Cont

{{skip}}k = k (L0 Ok)
{{fail l}}k = k (L1 l)
{{break}}k = k (L0 Break)
{{continue}}k = k (L0 Continue)

{{v:=e}}k = k (L0 Ok) [s | v : [[e]]s]

$$\{\{\text{if b then c0 else c1}\}\}ks = \begin{cases} \{\{c0\}\}ks & \text{si } [[b]]s \\ \{\{c1\}\}ks & \text{c.c.} \end{cases}$$

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

donde updOk ∈ Conts -> Cont -> Conts se define así:

$$\text{updOk k kok x} = \begin{cases} kok & \text{si } x = L0\ Ok \\ k\ x & \text{c.c.} \end{cases}$$

O sea, updOk k kok = [k | L0 Ok : kok]

De manera análoga pueden definirse updBreak, updContinue, updAbort (en éste último caso es un poco diferente).

{{catch l in c0 with c1}}k = {{c0}}(updAbort k l ({{c1}}k))

{{while b do c}}k = Y_{Σ -> X_⊥} F

donde

$$F\ w\ s = \begin{cases} \{\{c\}\}(\text{contsWhile k w (k (L0 Ok))})s & \text{si } [[b]]s \\ k\ s & \text{c.c.} \end{cases}$$

donde contsWhile ∈ Conts -> Cont -> Cont -> Conts se define así:

$$\text{contsWhile j w k x} = \begin{cases} w & \text{si } x = L0\ Ok \vee x = L0\ Continue \\ k & \text{si } x = L0\ Break \\ j\ x & \text{c.c.} \end{cases}$$

o también contsWhile j w k = updBreak (updContinue (updOk j w) w) k

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

Las operaciones sobre Conts deberían implementarse en un módulo aparte.