DEMOSTRANDO QUE F ES CONTINUA

Esto no lo hicimos en clase (porque pronto vamos a demostrar la continuidad de otra F parecida).

Para comprobar que dicha F es continua (y por lo tanto, confirmar que estamos utilizando bien el teorema del menor punto fijo), primero veamos que es monótona. Sean f y g funciones en el dominio Z -> Z_⊥ tales que f ≤' g, es decir, ∀n∈Z. f n ≤ g n. Veamos que F f ≤' F g, es decir, que ∀n∈Z. F f n ≤ F g n.
Tenemos que comparar:

$$\begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ f\ (n-2) & \text{c.c.} \end{cases}$$

con

$$\begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ g\ (n-2) & \text{c.c.} \end{cases}$$

Ambas son iguales, salvo en el caso n ∉ {0, 1} en que F f n = f (n-2) ≤ g (n-2) = F g n.
Por lo tanto F es monótona.

Por el corolario de Prop 2, para demostrar que es continua basta con demostrar que para toda cadena interesante f0 ≤' f1 ≤' f2 ≤' f3 ≤' ... de funciones en el dominio Z -> Z_⊥, F (sup'(fi)) ≤' sup'(F fi); es decir que ∀n∈Z. F (sup'(fi)) n = sup'(F fi) n.

Recordemos que supremo de una cadena de funciones es el supremo punto a punto, es decir, sup'(F fi) n = sup(F fi n). Debemos comparar F (sup'(fi)) n con sup(F fi n)

$$F\ \text{sup'(fi)}\ n = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ \text{sup'(fi)}\ (n-2) & \text{c.c.} \end{cases}$$

$$F\ \text{fi}\ n = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ \text{fi}\ (n-2) & \text{c.c.} \end{cases}$$

El sup(F fi n) depende de n:
sup(F fi 0) = sup({0, 0, 0, ...}) = sup(0) = 0 = F sup'(fi) 0.
sup(F fi 1) = sup({1, 1, 1, ...}) = sup(1) = 1 = F sup'(fi) 1.
Si n ∉ {0, 1}, sup(F fi n) = sup({f0 (n-2), f1 (n-2), ...}) = sup(fi (n-2)) = sup'(fi) (n-2).

Por lo tanto, F es continua.

ESPACIO DE FUNCIONES

Vimos que:

1) Si Q es un orden parcial, P -> Q también lo es, donde el orden entre funciones se define punto a punto.
2) Las cadenas de P -> Q son secuencias de funciones f0 ≤' f1 ≤' f2 ≤' ... ≤' fn ≤' ... tales que, tomadas punto a punto son cadenas, es decir, para todo p ∈ P, f0 p ≤ f1 p ≤ f2 p ≤ ... ≤ fn p ≤ ...
3) Si Q es un predominio, P -> Q también lo es, donde el supremo de una cadena de funciones se define punto a punto.
4) Si Q es un dominio, P -> Q también lo es, donde el mínimo se define punto a punto.

Si P y Q son predominios (resp. dominios) y tomamos P -c-> Q = {f ∈ P -> Q | f es continua}, entonces P -c-> Q también es un predominio (resp. dominio). La demostración es la misma ya que es el mismo orden, el mismo supremo y el mismo mínimo. Sólo que hay que demostrar que 5) el supremo de una cadena de funciones continuas también es una función continua. En el caso de dominios, también hay que demostrar que 6) la función constante ⊥ también es continua (trivial, ya que todas las funciones constantes son continuas).
Notación, en el libro (y acá), no se escribe P -c-> Q sino simplemente P -> Q, es decir, la misma notación que veníamos utilizando para el espacio de todas las funciones (continuas o no).

Demostraciones:

1) Sea Q con ≤ un orden parcial, se puede ver fácilmente que ≤' sobre P -> Q definida por

f ≤' g sii para todo x ∈ P. f x ≤ g x es un orden parcial (reflexivo, antisimétrico y transitivo).

2) Se deja como ejercicio.

3) Sea f0 ≤' f1 ≤' f2 ≤' f3 ≤' ... ∈ P -> Q una cadena de funciones. Se puede comprobar que f definido por f x = sup(fi x) es el supremo de f. En efecto, f es cota: para todo j y todo x ∈ P, fj x ≤ sup(fi x) = f x. Por lo tanto fj ≤' f para todo j. Además, f es la menor tal cota: sea g tal que para todo j, fj ≤' g. Sea x ∈ P. Para todo j, fj x ≤ g x. Por lo tanto, g x es cota de fj x, por lo tanto, f x = sup(fi x) ≤ g x. Por ello, f ≤' g. Conclusión, f = sup'(fi). O sea, sup'(fi) x = sup(fi x)

4) Sea ⊥' ∈ P -> Q definida por ⊥' x = ⊥ para todo x ∈ P. Claramente ⊥' ≤' f para toda función f, ya que para todo x ∈ P, ⊥' x = ⊥ ≤ f x.

5) Sea f0 ≤' f1 ≤' f2 ≤' f3 ≤' ... ∈ P -> Q una cadena de funciones continuas. Queremos demostrar que sup'(fi) es continua. Haremos explícito el índice sobre el que se toma supremo: quiero ver que sup'_i(fi) es continua. Para ver que es continua, sea x0 ≤" x1 ≤" x2 ≤" x3 ≤" ... una cadena de elementos de P. Debo ver que

sup'_i(fi) sup"_j(xj) = sup_j(sup'_i(fi) xj)

Comprobémoslo:

```
sup'_i(fi) sup"_j(xj) = sup_i(fi sup"_j(xj))    (por definición de sup', punto 2)
                      = sup_i(sup_j(fi xj))     (porque para todo i, fi es continua)
                      = sup_j(sup_i(fi xj))     (por lema)
                      = sup_j(sup'_i(fi) xj)    (por definición de sup', punto 2)
```

que es lo queríamos demostrar.

Lema: En un orden parcial, si sup_i(sup_j(xij)) existe, entonces sup_j(sup_i(xij)) también y son iguales.
Dem: sup_i(sup_j(xij)) es el elemento x caracterizado por
1) (es cota) para todo i, sup_j(xij) ≤ x
2) (es la menor de ellas) si para todo i, sup_j(xij) ≤ c, entonces x ≤ c
Estas condiciones son equivalentes a
1') para todo i, para todo j, xij ≤ x
2') si para todo i y para todo j, xij ≤ c, entonces x ≤ c
Si hiciéramos lo mismo con sup_j(sup_i(xij)) llegaríamos a que es el elemento y caracterizado por
3') para todo j, para todo i, xij ≤ y
4') si para todo j y para todo i, xij ≤ c, entonces y ≤ c
Es decir, que la única diferencia entre la caracterización de sup_i(sup_j(xij)) y sup_j(sup_i(xij)) es en el orden entre para todo i y para todo j. Como ∀i j es lo mismo que ∀j i, ambas caracterizaciones son equivalentes. Por ello, un supremo existe sii el otro existe, y son iguales.

LENGUAJE IMPERATIVO SIMPLE

Presentamos un lenguaje imperativo. Las expresiones enteras son como antes:

```
<intexp> ::= 0 | 1 | 2 | ...
           | <var>
           | -<intexp>
           | <intexp> + <intexp> | <intexp> - <intexp>
           | <intexp> * <intexp> | <intexp> ÷ <intexp> | <intexp> % <intexp>
```

Las expresiones booleanas son como las de <assert>, pero sin cuantificadores.

```
<boolexp> ::= true | false
            | <intexp> = <intexp> | <intexp> ≠ <intexp> | <intexp> ≤ <intexp>
            | <intexp> ≥ <intexp> | <intexp> < <intexp> | <intexp> > <intexp>
            | ¬ <boolexp>
            | <boolexp> ∧ <boolexp> | <boolexp> ∨ <boolexp>
            | <boolexp> ⇒ <boolexp> | <boolexp> ⇔ <boolexp>
```

La novedad son los comandos. Tenemos skip, asignación, secuencia, condicional, ciclos, y declaraciones de variables locales:

```
<comm> ::= skip | <var>:= <intexp>
         | <comm> ; <comm>
```

```
        | if <boolexp> then <comm> else <comm>
        | while <boolexp> do <comm>
        | newvar <var>:= <intexp> in <comm>
```

DOMINIOS SEMÁNTICOS

Para las expresiones enteras y booleanas podemos proceder como antes:

```
[[_]] ∈ <intexp> -> Σ -> Z
[[_]] ∈ <boolexp> -> Σ -> B
```

donde Σ = <var> -> Z

Ahora necesitamos un tercer dominio semántico para las frases de tipo <comm>. Es natural pensar que una tal frase puede interpretarse como un transformador del estado, es decir, una función de Σ en Σ:

```
[[_]] ∈ <comm> -> Σ -> Σ
```

Esta idea es razonable, pero no contempla el hecho de que un comando puede no terminar dado que hay ciclos. Para ello, agregamos ⊥ al posible resultado. Un comando puede transformar el estado, o puede que en un estado dado no termine:

```
[[_]] ∈ <comm> -> Σ -> Σ_⊥
```

Σ_⊥ es un dominio llano, muy parecido a Z_⊥ o B_⊥, salvo que en vez de enteros o booleanos, se trata de estados los que son todos incomparables entre sí (parece más complicado porque los estados son funciones, pero no lo es). Gráficamente:

```
       ... s s' s" s0 s1 s2 s3 ...
                 \ | /
                  ⊥
```

Los diferentes estados (por ejemplo, s, s' ∈ Σ = <var> -> Z) son incomparables entre sí porque Z en la definición de Σ tiene el orden discreto.

ECUACIONES SEMÁNTICAS

Las ecuaciones correspondientes a las expresiones enteras y booleanas son idénticas a las presentadas para la lógica de predicados. Presentamos las ecuaciones correspondientes a los comandos:

```
[[v:= e]]s = [s | v : [[e]]s]
```

Por ejemplo:

```
[[x:= x-1]]s = [s | x : [[x-1]]s]
             = [s | x : s x - 1]

[[y:= y+x]]s = [s | y : [[y+x]]s]
             = [s | y : s y + s x]

[[skip]]s = s
```

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

Por ejemplo:

$$[[\text{if x > y then x:= x-1 else y:= y+x}]]s = \begin{cases} [[x:= x-1]]s & \text{si } [[x>y]]s \\ [[y:= y+x]]s & \text{c.c.} \end{cases}$$

$$= \begin{cases} [s \mid x : s\ x - 1] & \text{si } s\ x > s\ y \\ [s \mid y : s\ y + s\ x] & \text{c.c.} \end{cases}$$

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

Acá nos hemos encontrado con una dificultad: no podemos escribir directamente [[c1]]([[c0]]s) porque el argumento de [[c1]] debe ser un estado, y [[c0]]s puede ser ⊥. Es decir, [[c1]] ∈ Σ -> Σ_⊥, y [[c0]]s ∈ Σ_⊥. No puedo aplicar directamente. Dada una f ∈ P -> D, donde P es un predominio y D un dominio, denotamos por f_⊥⊥ la única extensión estricta de f a P_⊥. Así, f_⊥⊥ ∈ P_⊥ -> D definida por:

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

En el caso del c0;c1, esto significa que si [[c0]]s = ⊥ (no termina), [[c1]]_⊥⊥ propaga ese comportamiento, lo que corresponde a la intuición nuestra: si c0 en el estado s no termina, entonces c0;c1 en ese mismo estado tampoco puede terminar.

Ejemplo:

```
[[x:= x-1; y:= y+x]]s = [[y:= y+x]]_⊥⊥ ([[x:= x-1]]s)
                      = [[y:= y+x]]_⊥⊥ [s | x : s x - 1]
                      = [[y:= y+x]] [s | x : s x - 1]
                      = (*)
```

Para continuar, sea s' = [s | x : s x - 1]

```
(*) = [[y:= y+x]] s'
    = [s' | y : s' y + s' x]
    = [s | x : s x - 1 | y : s y + s x - 1]
```

Semántica del while

Intuitivamente, escribiríamos:

[[while b do c]]s = [[if b then c ; while b do c else skip]]s

$$= \begin{cases} [[c;\text{while b do c}]]s & \text{si } [[b]]s \\ s & \text{c.c.} \end{cases}$$

Así como está, no es dirigido por sintaxis (recordemos que esto es importante porque garantiza existencia de solución única y composicionalidad, que a su vez garantiza que podemos reemplazar subfrases por otras de igual significado sin que el significado de la frase cambie).

Quizá si simplificamos ... :

$$(*) \quad [[\text{while b do c}]]s = \begin{cases} [[\text{while b do c}]]_{\perp\perp}\ ([[c]]s) & \text{si } [[b]]s \\ s & \text{c.c.} \end{cases}$$

Y no podemos simplificar más. Esto no es bueno porque nos quedó una ecuación que no es dirigida por sintaxis.

Vamos a manipular un poco esto. En realidad lo que nos dice (*) es que

[[while b do c]] = w,

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

¡Hagamos como la clase pasada!

Llamemos

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

Buscamos w tal que w s = F w s, o sea, w = F w.
Sabemos cómo hacer eso:

[[while b do c]] = sup'(F^i ⊥')

Suponiendo que F ∈ (Σ -> Σ_⊥) -> (Σ -> Σ_⊥) es continua (lo es). Esto nos da una ecuación dirigida por sintaxis ya que sólo usa la definición de F, que a su vez sólo utiliza la semántica de las subfrases de while b do c.

Veremos la clase que viene que efectivamente F es continua y daremos la ecuación para el newvar.