## LENGUAJES Y COMPILADORES

**Repaso.** La clase pasada demostramos el teorema del menor punto fijo

*Teorema.* Sea $D$ un dominio, y $F \in D \to D$ continua. Entonces $\sup(F^i \perp)$ existe y es el menor punto fijo de $F$.

y vimos que nos permite encontrar la menor solución a ecuaciones como

$$f\ n = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ f\ (n-2) & \text{si } n \notin \{0,1\} \end{cases} \tag{1}$$

Incluso vimos que dicha solución, en este ejemplo, es justamente la función $mod2$:

$$mod2\ n = \begin{cases} n\%2 & \text{si } n \geq 0 \\ \perp & \text{caso contrario} \end{cases}$$

También habíamos mencionado que si $Q$ es un orden parcial, predominio o dominio, $P \to Q$ también lo es con el orden, supremo o mínimo definido punto a punto. Si $P$ y $Q$ son predominios, podemos considerar el conjunto de funciones **continuas** de $P$ en $Q$, que es un subconjunto de $P \to Q$. Resulta que con la misma definición de supremo, este subconjunto de $P \to Q$ también es un predominio. Para ello, es necesario comprobar que si $f_0 \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq \ldots$ es una cadena de funciones continuas, el supremo $\bigsqcup_{i=0}^{\infty} f_i$ como lo hemos definido (punto a punto) es también una función continua.

Más aún, si $Q$ es un dominio, la función $x \mapsto \perp_Q$ es continua también (todas las funciones constantes lo son) y por lo tanto el conjunto de funciones continuas de $P$ en $Q$ es un dominio en ese caso.

Lamentablemente, la misma notación $P \to Q$ suele usarse para ambos conjuntos: para el conjunto de todas las funciones de $P$ en $Q$; o para el conjunto de todas las funciones **continuas** de $P$ en $Q$.

### Lenguaje imperativo simple

Finalmente consideramos un lenguaje de programación. Le llamamos lenguaje imperativo simple, y su sintaxis está dada por:

<intexp>::= 0 | 1 | 2 | . . .
| <var>
| -<intexp>
| <intexp> + <intexp>| <intexp> - <intexp>
| <intexp> * <intexp>| <intexp> ÷ <intexp>| <intexp> % <intexp>
<boolexp>::= **true**| **false**
| <intexp> = <intexp>| <intexp> ≠ <intexp>| <intexp>≤ <intexp>
| <intexp> ≥ <intexp>| <intexp> <<intexp>| <intexp>><intexp>
| ¬ <boolexp>

| <boolexp>∧ <boolexp>| <boolexp>∨ <boolexp>
| <boolexp>⇒ <boolexp>| <boolexp>⇔ <boolexp>
<comm>::= **skip**
| <var> := <intexp>
| <comm>;<comm>
| **if** <boolexp> **then** <comm> **else** <comm>
| **newvar** <var>:= <intexp> **in** <comm>
| **while** <boolexp> **do** <comm>

Es decir, las expresiones enteras son las mismas que en la lógica de predicados, y las expresiones booleanas también salvo que ahora, como es habitual en los lenguajes de programación, no hay cuantificadores, y por eso usamos un nombre diferente: <boolexp> en vez de <assert>.

La novedad principal está dada por la existencia de **comandos**: el vacío (**skip**), la asignación a una variable del valor de una expresión entera, la secuencia de comandos, el condicional, la variable local que introduce el **newvar**, y el ciclo **while**.

**Semántica denotacional.** Habiendo tres categorías semánticas (<intexp>, <boolexp> y <comm>), para cada una debemos encontrar el conjunto de denotaciones. Como en el caso de la lógica de predicados:

$$\begin{array}{rcl} [\![\ ]\!] & \in & \text{<intexp>} \to \Sigma \to \mathbb{Z} \\ [\![\ ]\!] & \in & \text{<boolexp>} \to \Sigma \to \mathbb{B} \end{array}$$

Y las ecuaciones se repiten textualmente.

$$\begin{array}{rcl} [\![\ ]\!] & \in & \text{<intexp>} \to \Sigma \to \mathbb{Z} \\ [\![0]\!]\sigma & = & 0 \\ [\![v]\!]\sigma & = & \sigma\ v \\ [\![-e]\!]\sigma & = & -[\![e]\!]\sigma \\ [\![e_0 + e_1]\!]\sigma & = & [\![e_0]\!]\sigma + [\![e_1]\!]\sigma \\ & \ldots & \end{array}$$

y

$$\begin{array}{rcl} [\![\mathbf{true}]\!]\sigma & = & V \\ [\![e_0 = e_1]\!]\sigma & = & [\![e_0]\!]\sigma = [\![e_1]\!]\sigma \\ & \ldots & \\ [\![\neg b]\!]\sigma & = & \neg[\![b]\!]\sigma \\ [\![b_0 \wedge b_1]\!]\sigma & = & [\![b_0]\!]\sigma \wedge [\![b_1]\!]\sigma \\ & \ldots & \end{array}$$

Las expresiones enteras siguen interpretandose como funciones que dado un estado devuelven un entero, y las expresiones booleanas como funciones que dado un estado devuelve un valor booleano.

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 $\Sigma$ en $\Sigma$:

$$[\![\ ]\!] \in \text{<comm>} \to \Sigma \to \Sigma$$

Esta idea es razonable, pero no contempla el hecho de que un comando puede no terminar dado que hay ciclos. Para ello, agregamos $\perp$ al conjunto de posibles resultados. Un comando puede transformar el estado, o puede que en un estado dado no termine:

$$[\![\ ]\!] \in <\mathrm{comm}> \to \Sigma \to \Sigma_\perp$$

donde $\Sigma_\perp$ es un dominio llano, muy parecido a $\mathbb{Z}_\perp$ o $\mathbb{B}_\perp$, salvo que en vez de enteros o booleanos, se trata de estados los que son todos incomparables entre sí. Puede parecer más complicado porque los estados son funciones y porque no son una cantidad numerable. Pero en realidad como dominio sigue siendo sencillo. Gráficamente:

$$\begin{array}{ccccccccc} \dots & \sigma & \sigma' & \sigma'' & \sigma_0 & \sigma_1 & \sigma_2 & \sigma_3 & \dots \\ & & \dots & \backslash & | & / & \dots & & \\ & & & & \perp & & & & \end{array}$$

Es decir, los diferentes estados (por ejemplo, $\sigma, \sigma' \in \Sigma = <\mathrm{var}> \to \mathbb{Z}$) son incomparables entre sí porque $\mathbb{Z}$ en la definición de $\Sigma$ tiene el orden discreto.

**Ecuaciones para comandos.** Presentamos las ecuaciones correspondientes a los comandos. El comando **skip** no modifica el estado. La asignación, en cambio, lo modifica de la manera obvia:

$$\begin{array}{rcl} [\![\ ]\!] & \in & <\mathrm{comm}> \to \Sigma \to \Sigma_\perp \\ [\![\mathbf{skip}]\!]\sigma & = & \sigma \\ [\![v := e]\!]\sigma & = & [\sigma | v : [\![e]\!]\sigma] \end{array}$$

Por ejemplo:

$$\begin{array}{rcl} [\![x := x - 1]\!]\sigma & = & [\sigma | x : [\![x - 1]\!]\sigma] \\ & = & [\sigma | x : [\![x]\!]\sigma - [\![1]\!]\sigma] \\ & = & [\sigma | x : \sigma\ x - 1] \\ [\![y := y + x]\!]\sigma & = & [\sigma | y : [\![y + x]\!]\sigma] \\ & = & [\sigma | y : [\![y]\!]\sigma + [\![x]\!]\sigma] \\ & = & [\sigma | y : \sigma\ y + \sigma\ x] \end{array}$$

El condicional modifica el estado de una u otra forma según sea verdadera o falsa la condición, evaluada en el estado inicial:

$$[\![\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1]\!]\sigma = \begin{cases} [\![c_0]\!]\sigma & \text{si } [\![b]\!]\sigma \\ [\![c_1]\!]\sigma & \text{si no} \end{cases}$$

Por ejemplo

$$\begin{array}{rcl} [\![\mathbf{if}\ x > y\ \mathbf{then}\ x := x - 1\ \mathbf{else}\ y := y + x]\!]\sigma & = & \begin{cases} [\![x := x - 1]\!]\sigma & \text{si } [\![x > y]\!]\sigma \\ [\![y := y + x]\!]\sigma & \text{si no} \end{cases} \\ & = & \begin{cases} [\sigma | x : \sigma\ x - 1] & \text{si } \sigma\ x > \sigma\ y \\ [\sigma | y : \sigma\ y + \sigma\ x] & \text{si no} \end{cases} \end{array}$$

La secuencia de comandos $c_0; c_1$ modifica sucesivamente el estado, esto es, el comando $c_1$ se ejecuta en el estado resultante de haber ejecutado $c_0$, o dicho de otra manera, el estado inicial de $c_1$ es el estado final de $c_0$. Esto podría expresarse con la ecuación

$$[\![c_0; c_1]\!]\sigma = [\![c_1]\!]([\![c_0]\!]\sigma)$$

pero la misma es incorrecta porque $[\![c_0]\!]\sigma$ no necesariamente es un estado, también puede ser $\perp$, en caso de que el comando $c_0$ no termine en el estado $\sigma$. Si revisamos el tipo de $[\![\ ]\!]$ notamos que $[\![c_1]\!] \in \Sigma \to \Sigma_\perp$ pero $[\![c_0]\!]\sigma \in \Sigma_\perp$, por lo que la aplicación del lado derecho de la ecuación no está bien tipada. Corregimos entonces

$$[\![c_0; c_1]\!]\sigma = \begin{cases} \perp & \text{si } [\![c_0]\!]\sigma = \perp \\ [\![c_1]\!]([\![c_0]\!]\sigma) & \text{si no} \end{cases}$$

Por conveniencia, dada una $f \in P \to D$, donde $P$ es un predominio y $D$ un dominio, se denota por $f_{\perp\!\!\!\perp}$ la única extensión estricta de $f$ a $P_\perp$. Así, $f_{\perp\!\!\!\perp} \in P_\perp \to D$ está definida por:

$$f_{\perp\!\!\!\perp}\ x = \begin{cases} \perp & \text{si } x = \perp \\ f\ x & \text{si no} \end{cases}$$

Ahora sí podemos escribir la ecuación definitiva para $[\![c_0; c_1]\!]$:

$$[\![c_0; c_1]\!]\sigma = [\![c_1]\!]_{\perp\!\!\!\perp}([\![c_0]\!]\sigma)$$

Se puede comprobar que si $[\![c_0]\!]\sigma = \perp$ (no termina), $[\![c_1]\!]_{\perp\!\!\!\perp}$ propaga ese comportamiento, lo que corresponde a la intuición nuestra: si $c_0$ en el estado $\sigma$ no termina, entonces $c_0; c_1$ en ese mismo estado tampoco puede terminar. Ejemplo de secuencia de comandos:

$$\begin{array}{rcl} [\![x := x - 1; y := y + x]\!]\sigma & = & [\![y := y + x]\!]_{\perp\!\!\!\perp}([\![x := x - 1]\!]\sigma) \\ & = & [\![y := y + x]\!]_{\perp\!\!\!\perp}[\sigma|x : \sigma\ x - 1] \\ & = & [\![y := y + x]\!][\sigma|x : \sigma\ x - 1] \\ & = & [\![y := y + x]\!]\overbrace{[\sigma|x : \sigma\ x - 1]}^{\sigma'} \\ & = & [\![y := y + x]\!]\sigma' \\ & = & [\sigma'|y : \sigma'\ y + \sigma'\ x] \\ & = & [\sigma|x : \sigma\ x - 1|y : \sigma\ y + \sigma\ x - 1] \end{array}$$

La ecuación correspondiente **newvar** v:= e **in** c debe expresar que el comando $c$ se ejecuta en el estado que resulta de inicializar la variable $v$ con el valor de la expresión $e$ en el estado inicial. Pero para garantizar que la variable $v$ es local, al finalizar debe restaurarse el valor de la variable global $v$:

$$[\![\textbf{newvar}\ \text{v:= e}\ \textbf{in}\ \text{c}]\!]\sigma = \begin{cases} \perp & \text{si } [\![c]\!][\sigma|v : [\![e]\!]\sigma] = \perp \\ [[\![c]\!][\sigma|v : [\![e]\!]\sigma]|v : \sigma\ v] & \text{si no} \end{cases}$$

Efectivamente, en el caso en que $[\![c]\!][\sigma|v : [\![e]\!]\sigma] = \perp$, el comando $c$ no termina por lo que el comando entero **newvar** v:= e **in** c tampoco puede terminar. En cambio, si $[\![c]\!][\sigma|v : [\![e]\!]\sigma] \neq \perp$, significa que $[\![c]\!][\sigma|v : [\![e]\!]\sigma] \in \Sigma$ es un estado, por lo tanto a ese estado, llamemosle $\sigma'$, debe restaurarsele el valor de la variable global $v$ escribiendo $[\sigma'|v : \sigma\ v]$. Esta ecuación también se puede escribir

$$[\![\textbf{newvar}\ \text{v:= e}\ \textbf{in}\ \text{c}]\!]\sigma = (\lambda\sigma' \in \Sigma.\ [\sigma'|v : \sigma\ v])_{\perp\!\!\!\perp}([\![c]\!][\sigma|v : [\![e]\!]\sigma])$$

usando la notación lambda $(\lambda)$.A la función $\lambda\sigma' \in \Sigma.\ [\sigma'|v : \sigma\ v]$ se lo puede llamar **operador de restauración**. La ecuación dice, entonces, que si el comando $c$ termina en un estado final se aplica el operador de restauración a dicho estado.