1. Un lenguaje imperativo simple

Primera aproximación. Vamos a introducir los conceptos centrales desde un fragmento (llamado LIS), que refleja las construcciones más significativas de este tipo de lenguajes.

La gramática abstracta que define LIS es la siguiente:

$$
\begin{array}{rcl}
\langle comm \rangle & ::= & \textbf{skip} \\
 & & \langle var \rangle := \langle intexp \rangle \\
 & & \langle comm \rangle \ ; \ \langle comm \rangle \\
 & & \textbf{if } \langle boolexp \rangle \textbf{ then } \langle comm \rangle \textbf{ else } \langle comm \rangle \\
 & & \textbf{newvar } \langle var \rangle := \langle intexp \rangle \textbf{ in } \langle comm \rangle \\
 & & \textbf{while } \langle boolexp \rangle \textbf{ do } \langle comm \rangle
\end{array}
$$

$$
\begin{array}{rcl}
\langle intexp \rangle & ::= & \langle natconst \rangle \\
 & & \langle var \rangle \\
 & & \langle intexp \rangle + \langle intexp \rangle \\
 & & \langle intexp \rangle * \langle intexp \rangle \\
 & & \langle intexp \rangle - \langle intexp \rangle \\
 & & \langle intexp \rangle \,/\, \langle intexp \rangle \\
 & & \langle intexp \rangle \,\%\, \langle intexp \rangle \\
 & & -\langle intexp \rangle
\end{array}
\qquad
\begin{array}{rcl}
\langle boolexp \rangle & ::= & \langle boolconst \rangle \\
 & & \langle intexp \rangle = \langle intexp \rangle \\
 & & \langle intexp \rangle < \langle intexp \rangle \\
 & & \langle intexp \rangle \leq \langle intexp \rangle \\
 & & \langle intexp \rangle > \langle intexp \rangle \\
 & & \langle intexp \rangle \geq \langle intexp \rangle \\
 & & \neg \langle boolexp \rangle \\
 & & \langle boolexp \rangle \vee \langle boolexp \rangle \\
 & & \langle boolexp \rangle \wedge \langle boolexp \rangle
\end{array}
$$

$$
\begin{array}{rcl}
\langle natconst \rangle & ::= & 0 \mid 1 \mid 2 \mid .... \\
\langle boolconst \rangle & ::= & \textbf{true} \mid \textbf{false}
\end{array}
$$

Note que 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: $\langle boolexp \rangle$ en vez de $\langle assert \rangle$.

El lenguaje LIS, en su búsqueda de simpleza posee varios aspectos reduccionistas. El primero que podemos mencionar (además de haber sólo dos tipos de expresiones: enteras y booleanas) es que las expresiones enteras siempre se pueden evaluar, y su resultado es un entero. Al igual que en el Cálculo de Predicados, seguimos asumiendo que todas las funciones primitivas (incluida la división) son funciones totales.

Para comenzar, excluimos de nuestra consideración al comando de iteración **while**. Veremos ahora como daríamos significado al lenguaje sin iteración, es decir un lenguaje en el cual todos los programas terminan (ni se cuelgan ni hay errores de operación).

LIS sólo posee variables que adoptan valores enteros. Luego la noción de estado se refleja en la siguiente definición:

$$\textbf{Conjunto de estados:} \qquad \Sigma \ = \ \langle var \rangle \to \mathbf{Z}$$

Esto es, el conjunto de estados Σ no es otra cosa que la familia de funciones totales entre el conjunto de variables y los enteros. Aquí se refleja otra característica reduccionista de LIS: la memoria posee infinitos lugares que siempre alojan un número entero.

El significado de los comandos y las expresiones del lenguaje LIS sin iteración estará dado por las siguientes funciones semánticas:

$[\![\_]\!]^{intexp} \in \langle intexp \rangle \to \Sigma \to \mathbf{Z}$
$[\![\_]\!]^{boolexp} \in \langle boolexp \rangle \to \Sigma \to \{V, F\}$
$[\![\_]\!]^{comm} \in \langle comm \rangle \to \Sigma \to \Sigma$

Debemos notar que dado que no estamos considerando la iteración en LIS, todos los comandos terminan en su ejecución, y su "resultado" es un nuevo estado.

Las ecuaciones semánticas para las frases de tipo $\langle intexp \rangle$ y $\langle boolexp \rangle$ son similares a las dadas para el lenguaje del Cálculo de Predicados. Las ecuaciones semánticas para las frases de tipo $\langle comm \rangle$ son las siguientes. Las primeras cuatro no merecen mayores comentarios:

$$\begin{array}{lcl}
[\![\textbf{skip}]\!]\sigma & = & \sigma \\
[\![v := e]\!]\sigma & = & [\sigma | v : [\![e]\!]\sigma] \\
[\![c_0; c_1]\!]\sigma & = & [\![c_1]\!]([\![c_0]\!]\sigma) \\
[\![\textbf{if } e \textbf{ then } c \textbf{ else } c']\!]\sigma & = & \text{if } [\![e]\!]\sigma \text{ then } [\![c]\!]\sigma \text{ else } [\![c']\!]\sigma
\end{array}$$

Por ejemplo:

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

Para definir el significado de las variables locales debemos expresar el hecho de que el valor original del estado en la variable local se restaura al terminar la ejecución de su alcance. (Definimos a c como el alcance de **newvar** $v := e$ **in** c.) Este hecho lo expresamos mediante la aplicación de la función "restauración de v según σ":

$$f_{v,\sigma}\sigma' = [\sigma' | v : \sigma v]$$

De esta manera podemos expresar el significado de la construcción **newvar**:

$$[\![\textbf{newvar } v := e \textbf{ in } c]\!]\sigma \quad = \quad f_{v,\sigma}([\![c]\!][\sigma | v : [\![e]\!]\sigma])$$

Dado que estas funciones auxiliares serán frecuentemente utilizadas, es conveniente incorporar un mecanismo que nos permita expresar su efecto sin necesidad de introducir nuevos símbolos de función (en este caso $f_{v,\sigma}$). Este mecanismo es la notación lambda, que

por ahora será para nosotros una herramienta del metalenguaje apropiada para describir funciones (ver descripción abajo). La función "restauración" adopta la siguiente forma:

$$f_{v,\sigma} \;=\; \lambda\sigma' \in \Sigma.\ [\sigma'|v : \sigma v]$$

De esta manera podemos expresar en el metalenguaje esta operación sin necesidad de introducir funciones auxiliares:

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

La notación lambda. Explicaremos ahora brevemente el uso de esta notación. Podemos decir que el resultado de la función entera $f(x) = 2x + 5$ "aplicada" a 3 es $2 * 3 + 5$, por que el mismo se obtiene producto de la sustitución en la expresión $2x + 5$ de x por 3. La notación lambda nos provee de una forma adecuada de definir funciones de manera que la operatoria con las mismas que involucran "aplicaciones" de funciones a otras expresiones se efectúe apropiadamente. La misma consiste en señalar en una expresión que variable "determina" la función, producto precisamente, de su variación. Mediante la expresión

$$\lambda x.\ 2x + 5$$

denotamos la función f (sin necesidad de introducir un nuevo símbolo). Podemos operar por ejemplo de la siguiente manera:

$$(\lambda x.\ 2x + 5)(4z) = 8z + 5$$

Note que la expresión $\lambda x.\ 2x + 5$ tiene un significado que no depende de x, de hecho podríamos escribir equivalentemente $\lambda w.\ 2w + 5$.

1.1. **Dificultades para dar significado a la iteración.** En los lenguajes imperativos, la estructura típica de recursión es la iteración, que tiene la forma del comando:

$$\textbf{while}\ b\ \textbf{do}\ c$$

El significado operacional usual establece que se pregunta si b es verdadero, y en ese caso se ejecuta c, y luego se continua en el estado corriente ejecutando nuevamente el while. En nuestro marco de significado, escribiríamos este hecho de la siguiente manera:

$$\begin{aligned}[\![\textbf{while}\ b\ \textbf{do}\ c]\!]\sigma \;&=\; \text{if } [\![b]\!]\sigma \text{ then } [\![c; \textbf{while}\ b\ \textbf{do}\ c]\!]\sigma \text{ else } [\![\textbf{skip}]\!]\sigma \\ &=\; \begin{cases} [\![\textbf{while}\ b\ \textbf{do}\ c]\!]([\![c]\!]\sigma) & \text{si } [\![b]\!]\sigma \\ \sigma & \text{si } \neg[\![b]\!]\sigma \end{cases}\end{aligned}$$

Surgen varias dificultades al pretender tomar esto como una definición. La primera dificultad es que, como en el caso de las funciones recursivas, el dominio semántico es inadecuado. Al incluir la iteración se incorpora otra situación de ejecución atípica: la no terminación del programa.

Incorporaremos entonces el símbolo de "no terminación" $\perp$ al dominio de resultados posibles. De manera que nuestro nuevo "espacio de significado" será

$$\Sigma_\perp \;=\; \Sigma \cup \{\perp\}$$

$\Sigma_\perp$ representa el conjunto de resultados posibles de la ejecución de un programa imperativo. El mismo cuenta por supuesto con una copia de Σ, representando a los resultados de programas que terminan en un estado. Además de la no terminación, en el futuro incorporará la posibilidad de representar otros comportamientos, como los errores de ejecución, y los inputs y outputs, por ejemplo.

Si volvemos ahora a la definición de arriba, la expresión

$$[\![\textbf{while } b \textbf{ do } c]\!]([\![c]\!]\sigma) \qquad (2)$$

tiene el problema de que $[\![c]\!]\sigma$ puede ser $\perp$, que no pertenece al dominio de $[\![\textbf{while } b \textbf{ do } c]\!]$. Esto se soluciona acudiendo a nuestras funciones auxiliares de transferencia de control. Si $f \in \Sigma \to \Sigma_\perp$, entonces definimos una nueva función $f_{\perp\!\!\perp} \in \Sigma_\perp \to \Sigma_\perp$ de la siguiente manera:

$$f_{\perp\!\!\perp}\sigma \;=\; \sigma \qquad\qquad f_{\perp\!\!\perp}\perp = \perp$$

Luego la expresión (2) puede ser corregida poniendo

$$[\![\textbf{while } b \textbf{ do } c]\!]_{\perp\!\!\perp}([\![c]\!]\sigma)$$

Pero el problema principal de querer tomar (1) como definición es la no dirección por sintaxis: la definición de $[\![\textbf{while } b \textbf{ do } c]\!]$ no se da exclusivamente en términos de $[\![b]\!]$ y $[\![c]\!]$. Más aún, la definición es circular: se usa $[\![\textbf{while } b \textbf{ do } c]\!]$ para definir $[\![\textbf{while } b \textbf{ do } c]\!]$, cayendo en un confuso estilo de definición autoreferente, develando así la naturaleza recursiva del while, y con ella los problemas que estuvimos viendo para las funciones recursivas enteras.

El análisis hecho para las funciones recursivas enteras de Haskell es útil para entender ahora el problema de la iteración:

- el significado de $[\![\textbf{while } b \textbf{ do } c]\!]$ es una función $\omega \in \Sigma \to \Sigma_\perp$
- tal función ω satisface la siguiente "ecuación funcional":

$$\omega\sigma \;=\; \begin{cases} \omega_{\perp\!\!\perp}([\![c]\!]\sigma) & \text{si } [\![b]\!]\sigma \\ \sigma & \text{si } \neg[\![b]\!]\sigma \end{cases} \qquad (W)$$

Obtenemos entonces una ecuación similar a la ecuación (ER). Nuevamente, describir el conjunto de soluciones de la ecuación (W) de manera genérica es matemáticamente complicado ya que puede haber infinitas funciones. Pero al igual que para las funciones recursivas nos interesa la solución "menos informativa" (o sea la menor). El Teorema del Menor Punto Fijo nos dará entonces la respuesta.

Un caso extremo de solución "menos informativa" lo constituye el significado de

while true do skip,

que es claramente la función idénticamente bottom:

$$[\![\textbf{while true do skip}]\!] = \omega \qquad\qquad \omega\sigma = \perp \quad \text{(para todo } \sigma \in \Sigma)$$

Sin embargo la ecuación (W) para este caso es:

$$\begin{aligned} \omega\sigma &= \begin{cases} \omega_{\perp\!\!\perp}([\![\mathbf{skip}]\!]\sigma) & \text{si } [\![\mathbf{true}]\!]\sigma \\ \sigma & \text{si } \neg[\![\mathbf{true}]\!]\sigma \end{cases} \\ &= \omega_{\perp\!\!\perp}\sigma \\ &= \omega\sigma \end{aligned}$$

Claramente cualquier función en $\Sigma \to \Sigma_\perp$ satisface la ecuación $\omega\sigma = \omega\sigma$. Luego el significado esperado (idénticamente bottom) no está determinado a priori por la ecuación (W). Vemos que se reproduce el mismo problema que el observado para las funciones recursivas enteras.

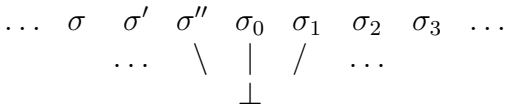
Preguntas.

(1) ¿Cuál es el significado de **while false do** c? Debe dar una función ω perteneciente a $\Sigma \to \Sigma_\perp$.
(2) ¿Cuál es el significado de **while** $x < 0$ **do skip**? Debe dar una función ω perteneciente a $\Sigma \to \Sigma_\perp$.
(3) Según lo visto hasta el momento, ¿qué puede decir sobre el significado de **while** b **do** c?

1.2. **Semántica denotacional de LIS.** Un comando puede transformar el estado, o puede que en un estado dado no termine:

$$[\![\ ]\!] \in \langle comm \rangle \to \Sigma \to \Sigma_\perp$$

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:



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

Ecuaciones para comandos. Los comandos **skip**, asignación y alternativa no presentan ninguna modificación en relación a los dados al principio. La ecuación

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

es ahora 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 σ. 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

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

Recordemos que 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 σ 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:

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

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 σ', debe restaurarsele el valor de la variable global v escribiendo $[\sigma'|v : \sigma\ v]$. Esta ecuación también se puede escribir

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

usando la notación lambda (λ).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.

Volvamos ahora al **while** para poder establecer su ecuación semámtica. Mencionamos anteriormente que desde la ecuación intuitiva:

$$\begin{array}{rcl} [\![\textbf{while } b \textbf{ do } c]\!]\sigma & = & [\![\textbf{if } b \textbf{ then } c; \textbf{while } b \textbf{ do } c \textbf{ else skip}]\!]\sigma \\ & = & \begin{cases} [\![\textbf{while } b \textbf{ do } c]\!]_{\perp\!\!\perp}([\![c]\!]\sigma) & \text{si } [\![b]\!]\sigma \\ \sigma & \text{si no} \end{cases} \end{array}$$

resulta una ecuación que no es dirigida por sintaxis, pero que por tener el formato de una ecuación recursiva podemos recurrir al Teorema del Menor Punto Fijo para caracterizar la solución que buscamos. Definamos

$$F\ w\ \sigma = \begin{cases} w_{\perp\!\!\perp}([\![c]\!]\sigma) & \text{si } [\![b]\!]\sigma \\ \sigma & \text{si no} \end{cases}$$

con esta definición, la ecuación del **while** queda

$$[\![\textbf{while } b \textbf{ do } c]\!]\sigma = F\ [\![\textbf{while } b \textbf{ do } c]\!]\ \sigma$$

Note que estamos planteando una igualdad de funciones. Es fácil comprobar que

$[\![\textbf{while } b \textbf{ do } c]\!] \in \Sigma \to \Sigma_\perp$ y $F \in (\Sigma \to \Sigma_\perp) \to (\Sigma \to \Sigma_\perp)$

Por ello, $F\ [\![\textbf{while } b \textbf{ do } c]\!] \in \Sigma \to \Sigma_\perp$, luego la ecuación también puede escribirse

$$[\![\textbf{while } b \textbf{ do } c]\!] = F\ [\![\textbf{while } b \textbf{ do } c]\!]$$

que simplemente dice que $[\![\textbf{while } b \textbf{ do } c]\!]$ es punto fijo de F. Recordemos entonces 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.

Recordemos que nos interesa el menor punto fijo de F porque otros puntos fijos contienen información que no se encuentra en la ecuación a resolver.

Sabemos que $F \in (\Sigma \to \Sigma_\perp) \to (\Sigma \to \Sigma_\perp)$, es decir $F \in D \to D$ con $D = \Sigma \to \Sigma_\perp$, que es un dominio porque $\Sigma_\perp$ lo es.

Para poder utilizar el teorema, deberíamos asegurarnos de que F es continua. En caso de serlo, la semántica de **while** b **do** c será

$$[\![\textbf{while } b \textbf{ do } c]\!] = \bigsqcup_{i=0}^{\infty} F^i\ \perp_{\Sigma\to\Sigma_\perp}$$

para

$$F\ w\ \sigma = \begin{cases} w_{\perp\!\!\perp}([\![c]\!]\sigma) & \text{si } [\![b]\!]\sigma \\ \sigma & \text{si no} \end{cases}$$

Esta definición sí es dirigida por sintaxis.

Veamos que F es continua. Para ello, primero vemos que es monótona: sean $w \sqsubseteq w' \in \Sigma \to \Sigma_\perp$. Si $[\![b]\!]\sigma$ y $[\![c]\!]\sigma = \perp$, entonces

$$\begin{array}{rcll} F\ w\ \sigma & = & w_{\perp\!\!\perp}([\![c]\!]\sigma) & \text{definición de } F \\ & = & \perp & [\![c]\!]\sigma = \perp \\ & \leq & F\ w'\ \sigma & \perp \text{ es el mínimo} \end{array}$$

En cambio, si $[\![b]\!]\sigma$ pero $[\![c]\!]\sigma \neq \bot$

$$\begin{array}{lcll} F\ w\ \sigma & = & w_{\perp\!\!\perp}([\![c]\!]\sigma) & \text{definición de } F \\ & = & w\ ([\![c]\!]\sigma) & [\![c]\!]\sigma \neq \bot \\ & \leq & w'\ ([\![c]\!]\sigma) & w \sqsubseteq w' \\ & = & w'_{\perp\!\!\perp}([\![c]\!]\sigma) & [\![c]\!]\sigma \neq \bot \\ & = & F\ w'\ \sigma & \text{definición de } F \end{array}$$

Por último, si $\neg[\![b]\!]\sigma$

$$\begin{array}{lcl} F\ w\ \sigma & = & \sigma \\ & = & F\ w'\ \sigma \end{array}$$

Por lo tanto, F es monótona. Por las propiedades ya demostradas, para comprobar continuidad, alcanza con demostrar que para toda cadena $w_0 \sqsubseteq w_1 \sqsubseteq w_2 \sqsubseteq \ldots$ de funciones de $\Sigma \to \Sigma_\perp$, $F\ (\bigsqcup_{i=0}^{\infty} w_i) \sqsubseteq \bigsqcup_{i=0}^{\infty} F\ w_i$.

Nuevamente consideramos tres casos. Si $[\![b]\!]\sigma$ y $[\![c]\!]\sigma = \bot$, entonces

$$\begin{array}{lcll} F\ (\bigsqcup_{i=0}^{\infty} w_i)\ \sigma & = & (\bigsqcup_{i=0}^{\infty} w_i)_{\perp\!\!\perp}([\![c]\!]\sigma) & \text{definición de } F \\ & = & \bot & [\![c]\!]\sigma = \bot \\ & \leq & \bigsqcup_{i=0}^{\infty} F\ w_i\ \sigma & \bot \text{ es el mínimo} \end{array}$$

Ahora, si $[\![b]\!]\sigma$ pero $[\![c]\!]\sigma \neq \bot$

$$\begin{array}{lcll} F\ (\bigsqcup_{i=0}^{\infty} w_i)\ \sigma & = & (\bigsqcup_{i=0}^{\infty} w_i)_{\perp\!\!\perp}([\![c]\!]\sigma) & \text{definición de } F \\ & = & (\bigsqcup_{i=0}^{\infty} w_i)\ ([\![c]\!]\sigma) & [\![c]\!]\sigma \neq \bot \\ & = & \bigcup_{i=0}^{\infty} w_i\ ([\![c]\!]\sigma) & \text{supremo de cadena de funciones} \\ & = & \bigcup_{i=0}^{\infty} w_{i\,\perp\!\!\perp}\ ([\![c]\!]\sigma) & [\![c]\!]\sigma \neq \bot \\ & = & \bigcup_{i=0}^{\infty} F\ w_i\ \sigma & \text{definición de } F \\ & = & (\bigsqcup_{i=0}^{\infty} F\ w_i)\ \sigma & \text{supremo de cadena de funciones} \end{array}$$

Finalmente, si $\neg[\![b]\!]\sigma$,

$$\begin{array}{lcll} F\ (\bigsqcup_{i=0}^{\infty} w_i)\ \sigma & = & \sigma & \text{definición de } F \\ & = & \bigcup_{i=0}^{\infty}\ \sigma & \text{supremo de cadena constante} \\ & = & \bigcup_{i=0}^{\infty}\ F\ w_i\ \sigma & \text{definición de } F \\ & = & (\bigsqcup_{i=0}^{\infty} F\ w_i)\ \sigma & \text{supremo de cadena de funciones} \end{array}$$

Por lo tanto F es continua y la definición dada para la semántica del **while** está bien definida.

1.3. **Ejemplo.** Podemos considerar algunos ejemplos de cálculo de semántica de programas con **while**. Revisemos primero el programa **while true do skip** que es el caso más sencillo de un programa que no termina en ningún estado, y por ello, su semántica debe ser $\bot_{\Sigma \to \Sigma_\perp}$.

Sea $F \in (\Sigma \to \Sigma_\perp) \to (\Sigma \to \Sigma_\perp)$ definida por

$$\begin{array}{rcl} F\ w\ \sigma & = & \begin{cases} w_{\perp\!\!\perp}([\![\mathbf{skip}]\!]\sigma) & \text{si } [\![\mathbf{true}]\!]\sigma \\ \sigma & \text{si no} \end{cases} \\ & = & w_{\perp\!\!\perp}([\![\mathbf{skip}]\!]\sigma) \\ & = & w_{\perp\!\!\perp}\ \sigma \\ & = & w\ \sigma \end{array}$$

es decir, $F\ w = w$. Calculemos $F^i\ \perp_{\Sigma\to\Sigma_\perp}$ para algunos $i \in \mathbb{N}$:

$$\begin{array}{rcl} F^0\ \perp_{\Sigma\to\Sigma_\perp} & = & \perp_{\Sigma\to\Sigma_\perp} \\ F^1\ \perp_{\Sigma\to\Sigma_\perp} & = & F\ (F^0\ \perp_{\Sigma\to\Sigma_\perp}) \\ & = & F\ \perp_{\Sigma\to\Sigma_\perp} \\ & = & \perp_{\Sigma\to\Sigma_\perp} \end{array}$$

Tenemos entonces $F^1\ \perp_{\Sigma\to\Sigma_\perp} = F^0\ \perp_{\Sigma\to\Sigma_\perp}$. En general, si tenemos $F^{i+1}\ \perp_D = F^i\ \perp_D$ podemos concluir que $F^i\ \perp_D = F^{i+1}\ \perp_D = F^{i+2}\ \perp_D = \ldots$ y la cadena resulta no interesante y su supremo es $F^i\ \perp_D$.

En el caso de **while true do skip** obtenemos, $\bigsqcup_{i=0}^{\infty} F^i\ \perp_{\Sigma\to\Sigma_\perp} = \perp_{\Sigma\to\Sigma_\perp}$ que es lo que habíamos intuido.

Podíamos haber observado también, al obtener $F\ w = w$, que cualquier $w \in \Sigma \to \Sigma_\perp$ es punto fijo de F ya que F es la función identidad. El menor de ellos es el mínimo de $\Sigma \to \Sigma_\perp$, o sea, $\perp_{\Sigma\to\Sigma_\perp}$.

Un ejemplo no trivial es el de **while** $x \neq 0 \wedge x \neq 1$ **do** $x := x - 2$. Intuitivamente, este programa reemplaza el valor de x en el estado por el del módulo 2 de dicho valor si el mismo es no negativo. Si es negativo, no termina. Comparar con el caso considerado anteriormente de una función recursiva que tenía a $mod2$ como solución.

Tenemos

$$[\![\mathbf{while}\ x \neq 0 \wedge x \neq 1\ \mathbf{do}\ x := x - 2]\!] = \bigsqcup_{i=0}^{\infty} F^i\ \perp_{\Sigma\to\Sigma_\perp}$$

para

$$\begin{array}{rcl} F\ w\ \sigma & = & \begin{cases} w_{\perp\!\!\perp}([\![x := x - 2]\!]\sigma) & \text{si } [\![x \neq 0 \wedge x \neq 1]\!]\sigma \\ \sigma & \text{si no} \end{cases} \\ & = & \begin{cases} w_{\perp\!\!\perp}([\sigma | x : \sigma\ x - 2]) & \text{si } \sigma\ x \notin \{0,1\} \\ \sigma & \text{si no} \end{cases} \\ & = & \begin{cases} w\ ([\sigma | x : \sigma\ x - 2]) & \text{si } \sigma\ x \notin \{0,1\} \\ \sigma & \text{si no} \end{cases} \\ & = & \begin{cases} \sigma & \text{si } \sigma\ x \in \{0,1\} \\ w\ [\sigma | x : \sigma\ x - 2] & \text{si no} \end{cases} \end{array}$$

Calculemos $F^i\ \bot_{\Sigma\to\Sigma_\bot}$ para algunos $i \in \mathbb{N}$:

$$
\begin{array}{rcl}
F^0\ \bot_{\Sigma\to\Sigma_\bot} & = & \bot_{\Sigma\to\Sigma_\bot} \\
F^1\ \bot_{\Sigma\to\Sigma_\bot} & = & F\ (F^0\ \bot_{\Sigma\to\Sigma_\bot}) \\
& = & F\ \bot_{\Sigma\to\Sigma_\bot} \\
& = & \sigma \mapsto \begin{cases} \sigma & \text{si } \sigma\ x \in \{0,1\} \\ \bot_{\Sigma\to\Sigma_\bot}\ [\sigma|x : \sigma\ x-2] & \text{si no} \end{cases} \\
& = & \sigma \mapsto \begin{cases} \sigma & \text{si } \sigma\ x \in \{0,1\} \\ \bot & \text{si no} \end{cases} \\
F^2\ \bot_{\Sigma\to\Sigma_\bot} & = & F\ (F^1\ \bot_{\Sigma\to\Sigma_\bot}) \\
& = & \sigma \mapsto \begin{cases} \sigma & \text{si } \sigma\ x \in \{0,1\} \\ F^1\ \bot_{\Sigma\to\Sigma_\bot}\ [\sigma|x : \sigma\ x-2] & \text{si no} \end{cases} \\
& = & \sigma \mapsto \begin{cases} \sigma & \text{si } \sigma\ x \in \{0,1\} \\ F^1\ \bot_{\Sigma\to\Sigma_\bot}\ \overbrace{[\sigma|x : \sigma\ x-2]}^{\sigma'} & \text{si no} \end{cases} \\
& = & \sigma \mapsto \begin{cases} \sigma & \text{si } \sigma\ x \in \{0,1\} \\ \sigma' & \text{si } \sigma\ x \notin \{0,1\} \wedge \sigma'\ x \in \{0,1\} \\ \bot & \text{si } \sigma\ x \notin \{0,1\} \wedge \sigma'\ x \notin \{0,1\} \end{cases} \\
& = & \sigma \mapsto \begin{cases} \sigma & \text{si } \sigma\ x \in \{0,1\} \\ [\sigma|x : \sigma\ x-2] & \text{si } \sigma\ x \notin \{2,3\} \\ \bot & \text{si } \sigma\ x \notin \{0,1,2,3\} \end{cases} \\
& = & \sigma \mapsto \begin{cases} [\sigma|x : \sigma\ x\%2] & \text{si } \sigma\ x \notin \{0,1,2,3\} \\ \bot & \text{si } \sigma\ x \notin \{0,1,2,3\} \end{cases}
\end{array}
$$

$$
\begin{array}{rcl}
F^3\ \bot_{\Sigma\to\Sigma_\bot} & = & F\ (F^2\ \bot_{\Sigma\to\Sigma_\bot}) \\
& = & \sigma \mapsto \begin{cases} \sigma & \text{si } \sigma\ x \in \{0,1\} \\ F^2\ \bot_{\Sigma\to\Sigma_\bot}\ \overbrace{[\sigma|x : \sigma\ x-2]}^{\sigma'} & \text{si no} \end{cases} \\
& = & \sigma \mapsto \begin{cases} \sigma & \text{si } \sigma\ x \in \{0,1\} \\ [\sigma'|x : \sigma'\ x\%2] & \text{si } \sigma\ x \notin \{0,1\} \wedge \sigma'\ x \in \{0,1,2,3\} \\ \bot & \text{si } \sigma\ x \notin \{0,1\} \wedge \sigma'\ x \notin \{0,1,2,3\} \end{cases} \\
& = & \sigma \mapsto \begin{cases} \sigma & \text{si } \sigma\ x \in \{0,1\} \\ [\sigma'|x : \sigma'\ x\%2] & \text{si } \sigma\ x \in \{2,3,4,5\} \\ \bot & \text{si } \sigma\ x \notin \{0,1,2,3,4,5\} \end{cases} \\
& = & \sigma \mapsto \begin{cases} [\sigma|x : \sigma\ x\%2] & \text{si } \sigma\ x \in \{0,1,2,3,4,5\} \\ \bot & \text{si } \sigma\ x \notin \{0,1,2,3,4,5\} \end{cases}
\end{array}
$$

Se puede comprobar que

$$
F^i\ \bot_{\Sigma\to\Sigma_\bot} = \sigma \mapsto \begin{cases} [\sigma|x : \sigma\ x\%2] & \text{si } \sigma\ x \in \{0,1,\ldots,2*i-1\} \\ \bot & \text{si } \sigma\ x \notin \{0,1,\ldots,2*i-1\} \end{cases}
$$

Observar que $F^i\ \bot_{\Sigma\to\Sigma_\bot}$ es la función que, aplicada a un estado para el que alcanzarían con (a lo sumo) $i-1$ iteraciones del ciclo para terminar, devuelve el estado final del **while**,

y aplicada a un estado para el que no alcanzarían, devuelve $\bot$. En el límite esto daría la función que aplicada a un estado para el que alcanza con una cantidad finita de iteraciones, devuelve el estado final, y aplicada a un estado para el que no termina, devuelve $\bot$:

$$\bigsqcup_{i=0}^{\infty} F^i \ \bot_{\Sigma \to \Sigma_\bot} = \sigma \mapsto \begin{cases} [\sigma | x : \sigma \ x\%2] & \text{si } \sigma \ x \geq 0 \\ \bot & \text{si no} \end{cases}$$

que es lo que habíamos anticipado originariamente.

2. Propiedades de la semántica

Variables Libres, Variables asignables. Similar a los predicados, el conjunto de variables libres de un comando se define con las siguientes ecuaciones.

$$\begin{array}{lcl} FV(\textbf{skip}) & = & \emptyset \\ FV(v := e) & = & \{v\} \cup FV(e) \\ FV(c_0; c_1) & = & FV(c_0) \cup FV(c_1) \\ FV(\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1) & = & FV(b) \cup FV(c_0) \cup FV(c_1) \\ FV(\textbf{while } b \textbf{ do } c) & = & FV(b) \cup FV(c) \\ FV(\textbf{newvar v:= e in c}) & = & FV(e) \cup (FV(c) - \{v\}) \end{array}$$

El conjunto de las variables que eventualmente modifican su estado en la ejecución de un comando se define con las siguientes ecuaciones.

$$\begin{array}{lcl} FA(\textbf{skip}) & = & \emptyset \\ FA(v := e) & = & \{v\} \\ FA(c_0; c_1) & = & FA(c_0) \cup FA(c_1) \\ FA(\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1) & = & FA(c_0) \cup FA(c_1) \\ FA(\textbf{while } b \textbf{ do } c) & = & FA(c) \\ FA(\textbf{newvar v:= e in c}) & = & FV(c) - \{v\} \end{array}$$

Claramente $FA(c) \subseteq FV(c)$, como puede comprobarse ecuación por ecuación.

Teorema de Coincidencia (TC). Si dos estados σ y σ' coinciden en las variables libres de c, entonces da lo mismo evaluar c en σ o σ'.

Note que debemos tener cuidado en como esta proposición se escribe, ya que si enunciamos el hecho de manera similar a los predicados tenemos una proposición falsa:

$$(\forall w \in FV(c).\ \sigma w = \sigma' w) \Rightarrow [\![c]\!]\sigma = [\![c]\!]\sigma'$$

Esto no vale si c es un comando, ya que devuelven estados finales que pueden diferir en las $w \notin FV(c)$ (porque tales diferencias podían existir previamente). Además, podría ocurrir que uno de ellos no termine, en cuyo caso ni siquiera devolvería un estado. Puede verse que si uno de ellos se cuelga, el otro también.

El enunciado correcto del **Teorema de Coincidencia** es:

(1) $(\forall w \in FV(c).\ \sigma w = \sigma' w)$ implica,
   o bien $[\![c]\!]\sigma = \bot = [\![c]\!]\sigma'$,
   o bien $[\![c]\!]\sigma \neq \bot \neq [\![c]\!]\sigma'$ y $\forall w \in FV(c).\ [\![c]\!]\sigma w = [\![c]\!]\sigma' w$

(2) Si $[\![c]\!]\sigma \neq \perp$, entonces $\forall w \notin FA(c).\ [\![c]\!]\sigma w = \sigma w$.

Teorema de Renombre (TR). Otra propiedad que cobra importancia en el lenguaje imperativo es la que establece que no importa el nombre de las variables utilizadas en las declaraciones de variables locales (o sea las ligadas):

$$\mathsf{u} \notin FV(c) - \{\mathsf{v}\} \Rightarrow [\![\textbf{newvar } \mathsf{u} := e \textbf{ in } c/\mathsf{v} \to \mathsf{u}]\!] = [\![\textbf{newvar } \text{v:= e } \textbf{in } \text{c}]\!]$$

La demostración del TR surge inmediatamente del Teorema de Sustitución, que ahora abordaremos.

Sustituciones. Por culpa de la asignación $v := e$, no podemos reemplazar una variable por una expresión entera arbitraria en un comando. Sólo podemos reemplazarla por otra variable. Luego consideraremos como el conjunto de las sustituciones al siguiente

$$\Delta = \langle var \rangle \to \langle var \rangle$$

A pesar de este cambio, las ecuaciones que la definían para la lógica de predicados siguen valiendo (salvo el caso de los cuantificadores, que ahora no tenemos). Ahora se agregan ecuaciones para los comandos.

$$\_/\_ \in \langle comm \rangle \times \Delta \to \langle comm \rangle$$

$$\begin{aligned}
\textbf{skip}/\delta &= \textbf{skip} \\
(v := e)/\delta &= (\delta v) := (e/\delta) \\
(c_0; c_1)/\delta &= (c_0/\delta); (c_1/\delta) \\
(\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1)/\delta &= \textbf{if } b/\delta \textbf{ then } c_0/\delta \textbf{ else } c_1/\delta \\
(\textbf{while } b \textbf{ do } c)/\delta &= \textbf{while } b/\delta \textbf{ do } c/\delta
\end{aligned}$$

El caso del newvar tiene los mismos inconvenientes que los cunatificadores, ahora es un poco más simple ya que δw son siempre variables.

$$(\textbf{newvar } v := e \textbf{ in } c)/\delta = \textbf{newvar } v_{new} := e/\delta \textbf{ in } (c/[\delta | v : v_{new}])$$

donde $v_{new} \notin \{\delta w | w \in FV(c) - \{v\}\}$

El enunciado del Teorema de sustitución se complica bastante en relación al correspondiente Teorema del cálculo de predicados. La cuestión es: ¿si aplico la sustitución δ a c y luego evalúo en el estado σ, puedo obtener el mismo resultado a partir de c sin sustituir si evalúo en un estado que hace el trabajo de δ y de σ (en las variables libres de c)?

O sea, vale

$$(\forall w \in FV(c).\ \sigma(\delta w) = \sigma' w) \Rightarrow [\![c/\delta]\!]\sigma = [\![c]\!]\sigma'?$$

Por los problemas mencionados para el TC, esta propiedad debería reescribirse atentos a la posibilidad de que los estados originales adopten valores distintos en las variables que no ocurren libres en c, y en consecuencia los estados finales difieran en las mismas. Deberíamos comparar entonces $[\![c/\delta]\!]\sigma(\delta w)$ con $[\![c]\!]\sigma' w$. Por ejemplo, si al programa $c = (\mathsf{x} := \mathsf{x} - 1; \mathsf{y} := 2 * \mathsf{y})$ se le sustituye x por u e y por v (sustitución δ), entonces al ejecutar

c/δ en un estado σ deberíamos obtener en u y v los valores que se obtienen al ejecutar c en un estado en donde x e y adopten los valores de σu y σv resp.

Pero ni siquiera así vale el TS. El problema ocurre cuando δ manda dos variables a una misma variable. Por ejemplo, consideremos el programa $\mathsf{x} := \mathsf{x} - 1; \mathsf{y} := 2 * \mathsf{y}$. Si tenemos la mala suerte de que δ reemplace x e y por la misma variable z, entonces el programa resultante será $\mathsf{z} := \mathsf{z} - 1; \mathsf{z} := 2 * \mathsf{z}$. Luego no será posible encontrar un estado σ' que efectúe el trabajo de δ y de σ en las variables libres: ninguna variable al ejecutar c en σ' experimentará el cambio que sufre z al ejecutar c/δ. O sea, aunque se satisfaga $\sigma' x = \sigma z = \sigma' y$, se tiene

$$[\![\mathsf{z} := \mathsf{z} - 1; \mathsf{z} := 2 * \mathsf{z}]\!]\sigma z \neq [\![\mathsf{x} := \mathsf{x} - 1; \mathsf{y} := 2 * \mathsf{y}]\!]\sigma' x$$

$$[\![\mathsf{z} := \mathsf{z} - 1; \mathsf{z} := 2 * \mathsf{z}]\!]\sigma z \neq [\![\mathsf{x} := \mathsf{x} - 1; \mathsf{y} := 2 * \mathsf{y}]\!]\sigma' y$$

El problema surge porque δ no es inyectiva.

Ahora sí:

Teorema de Sustitución. Si δ es inyectiva sobre $FV(c)$ y $\forall w \in FV(c).\ \sigma(\delta w) = \sigma' w$, entonces

o bien $[\![c/\delta]\!]\sigma = \bot = [\![c]\!]\sigma'$,

o bien $[\![c/\delta]\!]\sigma \neq \bot \neq [\![c]\!]\sigma'$ y $\forall w \in FV(c).\ [\![c/\delta]\!]\sigma(\delta w) = [\![c]\!]\sigma' w$.

La demostración del TS requiere de una generalización:

Lema de Sustitución (LS). Sea $FV(c) \subseteq V \subseteq \langle var \rangle$ tal que δ es inyectiva sobre V y $\forall w \in V.\ \sigma(\delta w) = \sigma' w$. Entonces,

o bien $[\![c/\delta]\!]\sigma = \bot = [\![c]\!]\sigma'$,

o bien $[\![c/\delta]\!]\sigma \neq \bot \neq [\![c]\!]\sigma'$ y $\forall w \in V.\ [\![c/\delta]\!]\sigma(\delta w) = [\![c]\!]\sigma' w$.

Resumen: Lenguaje imperativo simple (LIS)

Hemos presentado el lenguaje imperativo simple y dado las siguientes ecuaciones semánticas:

$$\begin{array}{rcll} [\![\ ]\!] & \in & \langle intexp \rangle \to \Sigma \to \mathbb{Z} & \\ [\![0]\!]\sigma & = & 0 & \\ [\![-e]\!]\sigma & = & -[\![e]\!]\sigma & \\ [\![e_0 + e_1]\!]\sigma & = & [\![e_0]\!]\sigma + [\![e_1]\!]\sigma & \text{similarmente para } -, *, \text{ etc.} \\ [\![v]\!]\sigma & = & \sigma\ v & \end{array}$$

$$
\begin{array}{rcll}
[\![\ ]\!] & \in & \langle boolexp\rangle \to \Sigma \to \mathbb{B} & \\
[\![\mathbf{true}]\!]\sigma & = & V & \\
[\![\mathbf{false}]\!]\sigma & = & F & \\
[\![e_0 = e_1]\!]\sigma & = & [\![e_0]\!]\sigma = [\![e_1]\!]\sigma & \text{similarmente para } <,\ >, \text{ etc.} \\
[\![\neg b]\!]\sigma & = & \neg[\![b]\!]\sigma & \\
[\![b_0 \wedge b_1]\!]\sigma & = & [\![b_0]\!]\sigma \wedge [\![b_1]\!]\sigma & \text{similarmente para } <,\ >,\ \vee, \text{ etc.}
\end{array}
$$

$$
\begin{array}{rcl}
[\![\ ]\!] & \in & \langle comm\rangle \to \Sigma \to \Sigma_\perp \\
[\![\mathbf{skip}]\!]\sigma & = & \sigma \\
[\![v := e]\!]\sigma & = & [\sigma | v : [\![e]\!]\sigma] \\
[\![\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1]\!]\sigma & = & \left\{ \begin{array}{ll} [\![c_0]\!]\sigma & \text{si } [\![b]\!]\sigma \\ [\![c_1]\!]\sigma & \text{si no} \end{array} \right. \\
[\![c_0; c_1]\!]\sigma & = & [\![c_1]\!]_{\perp\!\!\perp}([\![c_0]\!]\sigma) \\
[\![\mathbf{newvar}\ \text{v:= e}\ \mathbf{in}\ \text{c}]\!]\sigma & = & (\lambda\sigma' \in \Sigma.\ [\sigma' | v : \sigma\ v])_{\perp\!\!\perp}([\![c]\!][\sigma | v : [\![e]\!]\sigma]) \\
[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] & = & \bigsqcup_{i=0}^{\infty} F^i\ \perp_{\Sigma \to \Sigma_\perp}
\end{array}
$$

donde

$$
\begin{array}{rcl}
f_{\perp\!\!\perp}\ x & = & \left\{ \begin{array}{ll} \perp & \text{si } x = \perp \\ f\ x & \text{si no} \end{array} \right. \\
F\ w\ \sigma & = & \left\{ \begin{array}{ll} w_{\perp\!\!\perp}([\![c]\!]\sigma) & \text{si } [\![b]\!]\sigma \\ \sigma & \text{si no} \end{array} \right.
\end{array}
$$

Observación. La ecuación del **while** es expresada normalmente sin explicitar el estado σ, pero también puede explicitarse

$$[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!]\sigma = \bigsqcup_{i=0}^{\infty} F^i\ \perp_{\Sigma \to \Sigma_\perp}\ \sigma$$

donde ahora el supremo es el del dominio llano $\Sigma_\perp$, a diferencia del supremo de la definición anterior que es el del dominio $\Sigma \to \Sigma_\perp$. La equivalencia entre ambas definiciones es consecuencia de la definición de supremo de una cadena de funciones, que es el supremo punto a punto:

$$(\bigsqcup_{i=0}^{\infty^{\Sigma \to \Sigma_\perp}} F^i\ \perp_{\Sigma \to \Sigma_\perp})\ \sigma = \bigsqcup_{i=0}^{\infty^{\Sigma_\perp}} F^i\ \perp_{\Sigma \to \Sigma_\perp}\ \sigma$$

3. Fallas en el lenguaje imperativo

Para la incorporación de la posibilidad de transferencia de control por fallas, se agregan excepciones al lenguaje:

$$\langle comm \rangle ::= \textbf{fail} \mid \textbf{catchin}\ \langle comm \rangle\ \textbf{with}\ \langle comm \rangle$$

Ahora un programa tiene 3 comportamientos posibles:

(1) da un estado final
(2) aborta y da un estado final
(3) no termina

Sea $\Sigma' = \Sigma\ \cup\ \{\textbf{abort}\} \times \Sigma$ con el orden discreto. La función semántica tendrá ahora tipo:

$$[\![\_]\!] \in \langle comm \rangle \to \Sigma \to \Sigma'_\perp$$

En $\Sigma'_\perp$ seguimos teniendo un dominio llano. Las ecuaciones semánticas son:

$$\begin{aligned}
[\![\textbf{skip}]\!]\sigma &= \sigma \\
[\![\textbf{fail}]\!]\sigma &= \langle \textbf{abort}, \sigma \rangle \\
[\![v := e]\!]\sigma &= [\sigma | v : [\![e]\!]\sigma] \\
[\![\textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1]\!]\sigma &= \begin{cases} [\![c_0]\!]\sigma & \text{si } [\![b]\!]\sigma \\ [\![c_1]\!]\sigma & \text{si no} \end{cases}
\end{aligned}$$

Para establecer las ecuaciones semánticas de las restantes construcciones, es necesario disponer de tres operadores de transferencia de control similar al operador $(\_)_{\perp\!\!\perp}$. Pasamos ahora a definirlos.

Dada $f \in \Sigma \to \Sigma'_\perp$, denotamos por f_* la siguiente extensión de f a $\Sigma'_\perp$:

$$f_* \in \Sigma'_\perp \to \Sigma'_\perp$$

$$f_* x = \begin{cases} f\sigma & \text{si } x = \sigma \in \Sigma \\ x & \text{si no} \end{cases}$$

En este caso, la presencia de una situación abortiva determina que no se transfiere el control a f. Servirá para describir el significado de $c_0; c_1$, ya que si ocurre una situación de excepción al ejecutar c_0, el control no es tranferido a c_1.

Por otro lado, dado $f \in \Sigma \to \Sigma'_\perp$, denotamos por f_+ la siguiente extensión de f a $\Sigma'_\perp$:

$$f_+ \in \Sigma'_\perp \to \Sigma'_\perp$$

$$f_+ x = \begin{cases} f\sigma & \text{si } x = \langle \textbf{abort}, \sigma \rangle \in \{\textbf{abort}\} \times \Sigma \\ x & \text{si no} \end{cases}$$

En una clara dualidad con la definición de f_*, la definición de f_+ determina lo contrario: se tranfiere el control a f sólo en caso de excepción. Esto corresponderá a **catchin** c **with** c'.

Finalmente, dado $f \in \Sigma \to \Sigma$, denotamos por $f_\dagger$ la siguiente extensión de f a $\Sigma'_\perp$:

$$f_\dagger \in \Sigma'_\perp \to \Sigma'_\perp$$

$$f_\dagger x = \begin{cases} \langle \mathbf{abort}, f\sigma \rangle & x = \langle \mathbf{abort}, \sigma \rangle \\ fx & x \in \Sigma \\ \bot & x = \bot \end{cases}$$

Note que aquí hay una transferencia de control a f en cualquier situación (abortiva o no). Servirá para restaurar el valor de las variables locales.

El sentido completo de estas funciones quedará claro al analizar las ecuaciones semánticas que siguen.

$$\begin{aligned} [\![c_0; c_1]\!]\sigma &= [\![c_1]\!]_*([\![c_0]\!]\sigma) \\ [\![\mathbf{catchin}\ c_0\ \mathbf{with}\ c_1]\!]\sigma &= [\![c_1]\!]_+([\![c_0]\!]\sigma) \\ [\![\mathbf{newvar}\ \text{v:= e}\ \mathbf{in}\ \text{c}]\!]\sigma &= (\lambda\sigma' \in \Sigma.\ [\sigma'|v : \sigma\ v])_\dagger([\![c]\!][\sigma|v : [\![e]\!]\sigma]) \\ [\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] &= \bigsqcup_{i=0}^{\infty} F^i\ \bot_{\Sigma \to \Sigma_\bot} \end{aligned}$$

donde

$$F\ w\ \sigma = \begin{cases} w_*([\![c]\!]\sigma) & \text{si } [\![b]\!]\sigma \\ \sigma & \text{si no} \end{cases}$$

Ejercicio. Reformular el teorema de coincidencia y el de sustitución para que tenga en cuenta los posibles nuevos comportamientos.

4. Output

Agregamos el comando

$$\langle comm \rangle ::= !\,\langle intexp \rangle$$

donde $!e$ es el comando que escribe el valor de e. Claramente este comando puede encontrarse dentro de un ciclo o en cualquier fragmento de programa, eventualmente un programa puede generar una cantidad finita o infinita de output.

Los comportamientos posibles de un programa en un estado dado son ahora los siguientes:

- se genera una cantidad finita de output y luego "se cuelga"
- se genera una cantidad finita de output y luego termina
- se genera una cantidad finita de output y luego falla
- se genera una cantidad infinita de output

Sea Ω = conjunto de estos comportamientos. El mismo será la unión de las siguientes familias. Las mismas corresponden en orden a las situaciones de arriba.

- $\{\langle n_1, .., n_k \rangle :\ n_i \in \mathbf{Z}\}$
- $\{\langle n_1, .., n_k, \sigma \rangle :\ n_i \in \mathbf{Z},\ \sigma \in \Sigma\}$
- $\{\langle n_1, .., n_k, \langle \mathbf{abort}, \sigma \rangle \rangle :\ n_i \in \mathbf{Z},\ \sigma \in \Sigma\}$
- $\{\langle n_1, .., n_k, ... \rangle :\ n_i \in \mathbf{Z}\}$

En Ω se define la relación $\omega \leq \omega'$ cuando ω es segmento inicial de ω'. Con esta relación Ω es un dominio, donde el mínimo es la secuencia vacía $\langle\rangle$. Las cadenas interesantes tienen supremo de la forma $\{\langle n_1, .., n_k, ...\rangle\}$.

Las ecuaciones que definen la función semántica

$$[\![\_]\!] \in \langle comm \rangle \to \Sigma \to \Omega$$

son las siguientes. La primera define la nueva construcción:

$$[\![!e]\!]\sigma = \langle [\![e]\!]\sigma, \sigma \rangle$$

Las restantes se mantienen invariantes respecto del lenguaje sin output, sólo que se deben redefinir los operadores de transferencia de control.

$$\begin{array}{rcl}
[\![\textbf{skip}]\!]\sigma & = & \langle \sigma \rangle \\
[\![\textbf{fail}]\!]\sigma & = & \langle \langle \textbf{abort}, \sigma \rangle \rangle \\
[\![v := e]\!]\sigma & = & \langle [\sigma | v : [\![e]\!]\sigma] \rangle \\
[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!]\sigma & = & \begin{cases} [\![c_0]\!]\sigma & \text{si } [\![b]\!]\sigma \\ [\![c_1]\!]\sigma & \text{si no} \end{cases} \\
[\![c_0; c_1]\!]\sigma & = & [\![c_1]\!]_*([\![c_0]\!]\sigma) \\
[\![\textbf{catchin } c_0 \textbf{ with } c_1]\!]\sigma & = & [\![c_1]\!]_+([\![c_0]\!]\sigma) \\
[\![\textbf{newvar v:= e in c}]\!]\sigma & = & (\lambda\sigma' \in \Sigma.\ [\sigma' | v : \sigma\ v])_\dagger([\![c]\!][\sigma | v : [\![e]\!]\sigma]) \\
[\![\textbf{while } b \textbf{ do } c]\!] & = & \bigsqcup_{i=0}^{\infty} F^i\ \bot_{\Sigma \to \Omega}
\end{array}$$

donde

$$F\ w\ \sigma \ = \ \begin{cases} w_*([\![c]\!]\sigma) & \text{si } [\![b]\!]\sigma \\ \langle \sigma \rangle & \text{si no} \end{cases}$$

Los nuevos operadores de transferencia de control deben considerar los distintos tipos de comportamiento esperado. Sea $f \in \Sigma \to \Omega$, entonces las extensiones

$$f_*,\ f_+ \in \Omega \to \Omega$$

se definen de la siguiente manera. El operador $(\_)_*$ transfiere el control a la función sólo en caso de terminación normal:

$$f_* x = \begin{cases} \langle n_1, ..., n_k \rangle & x = \langle n_1, ..., n_k \rangle \\ \langle n_1, ..., n_k, f\sigma \rangle & x = \langle n_1, ..., n_k, \sigma \rangle \\ \langle n_1, ..., n_k, \langle \textbf{abort}, \sigma \rangle \rangle & x = \langle n_1, ..., n_k, \langle \textbf{abort}, \sigma \rangle \rangle \\ \langle n_1, ..., n_k, ... \rangle & x = \langle n_1, ..., n_k, ... \rangle \end{cases}$$

Note que una forma de definir $(\_)_*$ de manera más compacta es:

$$f_* x = \begin{cases} \langle\rangle & x = \langle\rangle \\ f\sigma & x = \langle \sigma \rangle \\ \langle \textbf{abort}, \sigma \rangle & x = \langle \textbf{abort}, \sigma \rangle \\ \langle n \rangle +\!\!+ f_*\omega & x = \langle n \rangle +\!\!+ \omega \end{cases}$$

De la misma manera definimos:

$$f_+ x = \begin{cases} \langle\rangle & x = \langle\rangle \\ \langle\sigma\rangle & x = \langle\sigma\rangle \\ f\sigma & x = \langle \mathbf{abort}, \sigma\rangle \\ \langle n\rangle +\!\!+ f_+\omega & x = \langle n\rangle +\!\!+ \omega \end{cases}$$

Sea $f \in \Sigma \to \Sigma$, entonces la extensión

$$f_\dagger \in \Omega \to \Omega$$

se define

$$f_\dagger x = \begin{cases} \langle\rangle & x = \langle\rangle \\ \langle f\sigma\rangle & x = \langle\sigma\rangle \\ \langle \mathbf{abort}, f\sigma\rangle & x = \langle \mathbf{abort}, \sigma\rangle \\ \langle n\rangle +\!\!+ f_\dagger\omega & x = \langle n\rangle +\!\!+ \omega \end{cases}$$

En las siguientes secciones abordaremos herramientas que nos permitirán definir dominios de manera apropiada, atendiendo a la complejidad que los mismos comienzan a evidenciar cuando avanzamos sobre las extensiones de LIS.

5. Más sobre dominios

Producto Cartesiano.

Producto de cpos. Si $P_0, P_1, ..., P_{n-1}$ son órdenes parciales, entonces $P_0 \times P_1, ... \times P_{n-1}$ también lo es, donde el orden entre tuplas se define componente a componente.

Cadenas. Una cadena de tuplas es una secuencia

$\langle p_0^1, p_1^1, ..., p_{n-1}^1\rangle \leq$
$\langle p_0^2, p_1^2, ..., p_{n-1}^2\rangle \leq$
$\vdots$
$\langle p_0^n, p_1^n, ..., p_{n-1}^1\rangle \leq$
$\vdots$

tal que si tomamos componente a componente formamos cadenas en los respectivos órdenes:

$p_0^1 \leq p_0^2 \leq ... \leq p_0^n \leq ...$
$p_1^1 \leq p_1^2 \leq ... \leq p_1^n \leq ...$
$\vdots$

Producto de predominios. Si $P_0, P_1, ..., P_{n-1}$ son predominios, entonces $P_0 \times P_1, ... \times P_{n-1}$ también lo es, donde el supremo de una cadena de tuplas se define componente a componente.

Producto de dominios. Si $P_0, P_1, ..., P_{n-1}$ son dominios, entonces $P_0 \times P_1, ... \times P_{n-1}$ también lo es, donde el mínimo es la tupla que consiste del mínimo de cada uno de los dominios.

Todas las funciones sencillas usuales (proyecciones, constructor de tuplas, etc) son trivialmente continuas.

Uniones Disjuntas. Dados conjuntos $P_0, P_1, ..., P_{n-1}$ se define la unión disjunta

$$P_0 + P_1 + ... + P_{n-1} = \{\langle i, p\rangle : p \in P_i\}.$$

Se definen las inyecciones $\iota_i \in P_i \to P_0 + P1 + ... + P_{n-1}$ mediante $\iota_i p = \langle i, p\rangle$

Uniones Disjuntas de cpos. Dados órdenes parciales $P_0, P_1, ..., P_{n-1}$, entonces $P_0 + P_1 + ... + P_{n-1}$ es un órden parcial, donde el orden "no mezcla" los órdenes de los diferentes conjuntos dados:

$$\langle i, p\rangle \leq \langle j, q\rangle \Longleftrightarrow i = j \ \wedge \ p \leq_i q$$

Cadenas. Una cadena de $P_0 + P_1 + ... + P_{n-1}$ es una que proviene enteramente de uno de los P_i:

$$\langle i, p_1\rangle \leq \langle i, p_2\rangle \leq ... \leq \langle i, p_n\rangle \leq ...$$

donde los p_j son todos elementos de P_i.

Unión de predominios. Dados predominios $P_0, P_1, ..., P_{n-1}$, entonces $P_0+P_1+...+P_{n-1}$ es un predominio, donde el supremo de una cadena es simplemente el supremo de la componente de donde la cadena proviene:

$$\bigsqcup_{j=1}^{\infty} \langle i, p_j\rangle = \langle i, \bigsqcup_{j=1}^{\infty} p_j\rangle$$

Unión de dominios. Dados dominios $P_0, P_1, ..., P_{n-1}$, entonces $P_0+P_1+...+P_{n-1}$ en general **no** es un dominio (sólo lo es en el caso trivial $n = 1$). ¿Cuál sería el mínimo, si los mínimos de los diferentes P_i no se comparan entre sí?

Todas las funciones sencillas usuales (inyecciones, análisis por casos, etc) son trivialmente continuas.

Dominios recursivos. El domino semántico para LIS con fallas y output puede ser definido a través de una **ecuación recursiva de dominos** que exprese el hecho de que un objeto ω puede ser visto como el tramo final de otros objetos pertenecientes a Ω: para todo k entero, $k +\!\!+ \omega$ es también un objeto de Ω. Esto es:

$$\Omega \ \approx \ (\Sigma' + \mathbf{Z} \times \Omega)_\perp$$

El símbolo $\approx$ significa isomorfismo, el cual está dado por las funciones continuas (una inversa de la otra):

$$\phi \ \in \ \Omega \to (\Sigma' + \mathbf{Z} \times \Omega)_\perp \qquad \psi \ \in \ (\Sigma' + \mathbf{Z} \times \Omega)_\perp \to \Omega$$

definidas mediante:

$$\phi x = \begin{cases} \bot & x = \langle\rangle \\ \iota_\bot(\iota_0 y) & x = \langle y\rangle \text{ con } y \in \Sigma' \\ \iota_\bot(\iota_1\langle n, \phi\omega\rangle) & x = \langle n\rangle +\!\!+ \omega \end{cases}$$

$$\psi x = \begin{cases} \langle\rangle & x = \bot \\ \langle y\rangle & x = \iota_\bot(\iota_0 y) \text{ con } y \in \Sigma' \\ \langle n\rangle +\!\!+ \psi\omega & x = \iota_\bot(\iota_1\langle n, \omega\rangle) \end{cases}$$

Notación. Para expresar las ecuaciones semánticas en adelante serán útiles las siguientes composiciones.

$$\begin{array}{lcl} \iota_{term} & = & \psi \cdot \iota_\bot \cdot \iota_0 \cdot \iota_{norm} \ \in \Sigma \to \Omega \\ \iota_{abort} & = & \psi \cdot \iota_\bot \cdot \iota_0 \cdot \iota_{abnorm} \ \in \Sigma \to \Omega \\ \iota_{out} & = & \psi \cdot \iota_\bot \cdot \iota_1 \ \in \mathbf{Z} \times \Sigma \to \Omega \\ \bot_\Omega & = & \psi(\bot) \in \Omega \end{array}$$

Las mismas pueden expresarse en el siguiente gráfico:

$$\begin{array}{ccccccc} \Sigma & \overset{\iota_{norm}}{\longrightarrow} & & \overset{\iota_0}{} & & & \\ & & \Sigma' & \longrightarrow & & & \\ \Sigma & \underset{\iota_{abnorm}}{\longrightarrow} & & & \Sigma' + \mathbf{Z} \times \Omega & \overset{\iota_\bot}{\longrightarrow} (\Sigma' + \mathbf{Z} \times \Omega)_\bot & \overset{\psi}{\longrightarrow} \Omega \\ & & \mathbf{Z} \times \Omega & \overset{\iota_1}{\longrightarrow} & & & \end{array}$$

Utilizando esta nueva definición matemática del domino semántico (dominio recursivo), podemos reformular las ecuaciones semánticas dadas con anterioridad de la siguiente manera.

$$\begin{array}{rcl} [\![\textbf{skip}]\!]\sigma & = & \iota_{term}\sigma \\ [\![\textbf{fail}]\!]\sigma & = & \iota_{abort}\sigma \\ [\![v := e]\!]\sigma & = & \iota_{term}[\sigma | v : [\![e]\!]\sigma] \\ [\![!e]\!]\sigma & = & \iota_{out}([\![e]\!]\sigma, \iota_{term}\sigma) \\ [\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!]\sigma & = & \begin{cases} [\![c_0]\!]\sigma & \text{si } [\![b]\!]\sigma \\ [\![c_1]\!]\sigma & \text{si no} \end{cases} \\ [\![c_0; c_1]\!]\sigma & = & [\![c_1]\!]_*([\![c_0]\!]\sigma) \\ [\![\textbf{catchin } c_0 \textbf{ with } c_1]\!]\sigma & = & [\![c_1]\!]_+([\![c_0]\!]\sigma) \\ [\![\textbf{newvar v:= e in c}]\!]\sigma & = & (\lambda\sigma' \in \Sigma.\ [\sigma' | v : \sigma\ v])_\dagger([\![c]\!][\sigma | v : [\![e]\!]\sigma]) \\ [\![\textbf{while } b \textbf{ do } c]\!] & = & \bigsqcup_{i=0}^{\infty} F^i\ \bot_{\Sigma\to\Omega} \end{array}$$

donde

$$F\ w\ \sigma \ = \ \begin{cases} w_*([\![c]\!]\sigma) & \text{si } [\![b]\!]\sigma \\ \iota_{term}\sigma & \text{si no} \end{cases}$$

Los operadores de transferencia de control pueden también ser redefinidos.

$$f_* x = \begin{cases} \bot_\Omega & x = \bot_\Omega \\ f\sigma & x = \iota_{term}\sigma \\ \iota_{abort}\sigma & x = \iota_{abort}\sigma \\ \iota_{out}(n, f_*\omega) & x = \iota_{out}(n,\omega) \end{cases}$$

$$f_+ x = \begin{cases} \bot_\Omega & x = \bot_\Omega \\ \iota_{term}\sigma & x = \iota_{term}\sigma \\ f\sigma & x = \iota_{abort}\sigma \\ \iota_{out}(n, f_+\omega) & x = \iota_{out}(n,\omega) \end{cases}$$

$$f_\dagger x = \begin{cases} \bot_\Omega & x = \bot_\Omega \\ \iota_{term}(f\sigma) & x = \iota_{term}\sigma \\ \iota_{abort}(f\sigma) & x = \iota_{abort}\sigma \\ \iota_{out}(n, f_\dagger\omega) & x = \iota_{out}(n,\omega) \end{cases}$$

6. Input

Para poder expresar la acción de alojar en una variable un entero dado como input, agregamos el comando:

$$\langle comm \rangle \ ::= ? \langle var \rangle$$

Para poder establecer el significado de esta construcción debemos enriquecer el domino semántico. Lo hacemos planteando una ecuación recursiva, y asumiendo que posee solución. La misma contempla (a través de la noción de función) la posibilidad de que el comportamiento de un programa dependa de un entero dado (input) .

$$\Omega \ \approx \ (\Sigma' + \mathbf{Z} \times \Omega + \mathbf{Z} \to \Omega)_\bot$$

Como antes, el símbolo $\approx$ significa isomorfismo, el cual está dado por las funciones continuas (una inversa de la otra):

$$\phi \ \in \ \Omega \to (\Sigma' + \mathbf{Z} \times \Omega + \mathbf{Z} \to \Omega)_\bot \qquad \psi \ \in \ (\Sigma' + \mathbf{Z} \times \Omega + \mathbf{Z} \to \Omega)_\bot \to \Omega$$

Notación. Para expresar las ecuaciones semánticas, actualizamos la lista de composiciones:

$$\begin{array}{lcl} \iota_{term} & = & \psi \cdot \iota_\bot \cdot \iota_0 \cdot \iota_{norm} \ \in \Sigma \to \Omega \\ \iota_{abort} & = & \psi \cdot \iota_\bot \cdot \iota_0 \cdot \iota_{abnorm} \ \in \Sigma \to \Omega \\ \iota_{out} & = & \psi \cdot \iota_\bot \cdot \iota_1 \ \in \mathbf{Z} \times \Sigma \to \Omega \\ \iota_{in} & = & \psi \cdot \iota_\bot \cdot \iota_2 \ \in (\mathbf{Z} \to \Sigma) \to \Omega \\ \bot_\Omega & = & \psi(\bot) \in \Omega \end{array}$$

Las mismas pueden expresarse en un gráfico similar al dado para output:

$$\begin{array}{lll} & \overset{\iota_{norm}}{\longrightarrow} & \\ \Sigma & & \overset{\iota_0}{\longrightarrow} \\ & & \Sigma' \\ \Sigma & \longrightarrow & \\ & \underset{\iota_{abnorm}}{\longrightarrow} & \end{array} \qquad \begin{array}{lll} & \overset{\iota_1}{\longrightarrow} & \\ \mathbf{Z} \times \Omega & & \overset{\iota_2}{\longrightarrow} \\ & & \mathbf{Z} \to \Omega \end{array} \qquad \Sigma' + \mathbf{Z} \times \Omega \overset{\iota_\perp}{\longrightarrow} (\Sigma' + \mathbf{Z} \times \Omega)_\perp \overset{\psi}{\longrightarrow} \Omega$$

Dar las ecuaciones semánticas para el lenguaje completo requiere sólo agregar la ecuación del constructor $?v$, y actualizar las definiciones de los operadores de transferencia de control.

$$[\![?v]\!]\sigma \quad = \quad \iota_{in}(\lambda n \in \mathbf{Z}.\ \iota_{term}[\sigma | v : n])$$

$$f_* x = \begin{cases} \perp_\Omega & x = \perp_\Omega \\ f\sigma & x = \iota_{term}\sigma \\ \iota_{abort}\sigma & x = \iota_{abort}\sigma \\ \iota_{out}(n, f_*\omega) & x = \iota_{out}(n, \omega) \\ \iota_{in}(f_* \cdot g) & x = \iota_{in} g \end{cases}$$

$$f_+ x = \begin{cases} \perp_\Omega & x = \perp_\Omega \\ \iota_{term}\sigma & x = \iota_{term}\sigma \\ f\sigma & x = \iota_{abort}\sigma \\ \iota_{out}(n, f_+\omega) & x = \iota_{out}(n, \omega) \\ \iota_{in}(f_+ \cdot g) & x = \iota_{in} g \end{cases}$$

$$f_\dagger x = \begin{cases} \perp_\Omega & x = \perp_\Omega \\ \iota_{term}(f\sigma) & x = \iota_{term}\sigma \\ \iota_{abort}(f\sigma) & x = \iota_{abort}\sigma \\ \iota_{out}(n, f_\dagger\omega) & x = \iota_{out}(n, \omega) \\ \iota_{in}(f_\dagger \cdot g) & x = \iota_{in} g \end{cases}$$

7. Semántica Operacional

Cuando hablamos de semántica operacional (o de transiciones), en vez de asociar significado a los programas a través de una función de los programas en los significados, se describe cómo se realiza el cómputo. Por ejemplo:

$$\langle \mathsf{x} := 1; \mathsf{y} := 2 * \mathsf{y}, \sigma \rangle \to \langle \mathsf{y} := 2 * \mathsf{y}, [\sigma | \mathsf{x} : 1] \rangle \to [\sigma | \mathsf{x} : 1 | \mathsf{y} : 2 * \sigma \mathsf{y}]$$

La relación $\to$ describe un paso de ejecución. Por eso se la llama *small-step semantics*. En cada paso se pasa de una *configuración* a otra. El conjunto de configuraciones que describe las etapas de cómputo será denotado mediante Γ. El mismo está formado por configuraciones *terminales* (que reflejan el resultado de un cómputo), y por configuraciones *no terminales* (que reflejan una etapa intermedia del mismo).

$$\Gamma = \Gamma_t \cup \Gamma_n$$

Cada uno de estos conjuntos está determinado por:

$$\begin{aligned} \Gamma_t &= \Sigma \\ \Gamma_n &= \langle comm \rangle \times \Sigma \end{aligned}$$

El paso de ejecución $\to$, que lleva una configuración no terminal en una configuración (terminal o no), se define matemáticamente como una relación:

$$\to \subseteq \ \Gamma_n \times \Gamma$$

La definición de una relación de este tipo (formada por objetos que proceden de una sintaxis abstracta) suele hacerse de manera axiomática. Vamos a mostrar ahora los axiomas que definen la misma.

Los primeros obedecen a una clara intuición de cómputo, y representan casos de obtención de configuraciones terminales:

$$\frac{}{\langle \mathbf{skip}, \sigma \rangle \to \sigma}$$

$$\frac{}{\langle v := e, \sigma \rangle \to [\sigma | v : [\![e]\!]\sigma]}$$

La posibilidad de obtener una secuencia encadenada de transiciones se representa axiomáticamente mediante:

$$\frac{\langle c_0, \sigma \rangle \to \sigma'}{\langle c_0; c_1, \sigma \rangle \to \langle c_1, \sigma' \rangle}$$

$$\frac{\langle c_0, \sigma \rangle \to \langle c_0', \sigma' \rangle}{\langle c_0; c_1, \sigma \rangle \to \langle c_0'; c_1, \sigma' \rangle}$$

Note como estas reglas permiten construir un paso intermedio de ejecución, como por ejemplo el primer paso de la ejecución de arriba:

$$\frac{\langle \mathsf{x} := 1, \sigma\rangle \to [\sigma|\mathsf{x} : 1]}{\langle \mathsf{x} := 1; \mathsf{y} := 2 * \mathsf{y}, \sigma\rangle \to \langle \mathsf{y} := 2 * \mathsf{y}, [\sigma|\mathsf{x} : 1]\rangle}$$

Las reglas para el comando if y el camando while también obedecen a su intuición operacional:

$$\frac{(\llbracket e \rrbracket \sigma = V)}{\langle \textbf{if } e \textbf{ then } c \textbf{ else } c', \sigma\rangle \to \langle c, \sigma\rangle}$$

$$\frac{(\llbracket e \rrbracket \sigma = F)}{\langle \textbf{if } e \textbf{ then } c \textbf{ else } c', \sigma\rangle \to \langle c', \sigma\rangle}$$

$$\frac{(\llbracket e \rrbracket \sigma = F)}{\langle \textbf{while } e \textbf{ do } c, \sigma\rangle \to \sigma}$$

$$\frac{(\llbracket e \rrbracket \sigma = T)}{\langle \textbf{while } e \textbf{ do } c, \sigma\rangle \to \langle c ; \textbf{ while } e \textbf{ do } c, \sigma\rangle}$$

Finalmente debemos dar reglas para el comando newvar. Una posible regla que podría considerarse natural es:

$$\frac{}{\langle \textbf{newvar } v := e \textbf{ in } c, \sigma\rangle \to \langle c; v := \sigma v, [\sigma|v : \llbracket e \rrbracket \sigma]\rangle}$$

La dificultad de esta regla es la pérdida de la localía de v en c, que tendría inconvenientes en una eventual extensión del lenguaje con concurrencia. En consistencia con el Reynolds vamos a adoptar:

$$\frac{\langle c, [\sigma|v : \llbracket e \rrbracket \sigma]\rangle \to \sigma'}{\langle \textbf{newvar } v := e \textbf{ in } c, \sigma\rangle \to [\sigma'|v : \sigma v]}$$

$$\frac{\langle c, [\sigma|v : \llbracket e \rrbracket \sigma]\rangle \to \langle c', \sigma'\rangle}{\langle \textbf{newvar } v := e \textbf{ in } c, \sigma\rangle \to \langle \textbf{newvar } v := \sigma' v \textbf{ in } c', [\sigma'|v : \sigma v]\rangle}$$

8. Corrección de la semántica operacional

Note que la relación $\to$ así definida es una función: en efecto, ninguna configuración no terminal puede mover (en un sólo paso) hacia más de una configuración. Esto significa que la relación es determinística. Además ninguna configuración no terminal puede mover hacia menos de una configuración (no se traba).

En efecto, sea $\langle c, \sigma\rangle$ una configuración no terminal. Entonces, si

- si $c = \mathbf{skip}$, hay una única transición posible, a σ.
- si $c = \ (v := e)$, hay una única transición posible, a $[\sigma | v : [\![e]\!]\sigma]$.
- lo mismo se puede establecer para el **if** y el **while**, dependiendo de el valor de verdad de la guarda.
- si $c = c0; c1$, entonces, por hipótesis inductiva desde $\langle c_0, \sigma\rangle$ hay una única transición posible. Si esa transición es a una configuración de la forma σ', entonces hay una única transición posible desde $\langle c, \sigma\rangle$, es a $\langle c_1, \sigma'\rangle$. Si en cambio esa transición es a una configuración de la forma $\langle c_0', \sigma\rangle$, entonces hay una única transición posible desde $\langle c, \sigma\rangle$, es a $\langle c_0'; c_1, \sigma'\rangle$.

Como $\to$ es una función, para toda configuración existe una única ejecución. Por *ejecución* entendemos una secuencia $c_0 \to c_1 \to c_2 \to ...$ maximal, esto es, que no puede prolongarse más de lo que está. Dicha ejecución es infinita o termina en una configuración terminal σ. Si la ejecución es infinita decimos que c_0 *diverge* y escribimos $c_0 \uparrow$. Se puede entonces definir:

$$\{\!\{c\}\!\}\sigma = \begin{cases} \bot & \text{si } \langle c, \sigma\rangle \uparrow \\ \sigma' & \text{si existe } \sigma' \text{ tal que } \langle c, \sigma\rangle \to^* \sigma' \end{cases}$$

donde $\to^*$ es la clausura reflexiva y transitiva definida a continuación.

$$\frac{\gamma \to \gamma'}{\gamma \to^* \gamma'}$$

$$\frac{}{\gamma \to^* \gamma}$$

$$\frac{\gamma \to^* \gamma' \qquad \gamma' \to^* \gamma''}{\gamma \to^* \gamma''}$$

Se puede demostrar que $\{\!\{c\}\!\} = [\![c]\!]$. Damos ahora una serie de resultados que están involucrados en la prueba.

Lema 1.

(1) Si $\langle c_0, \sigma\rangle \to^* \sigma'$, entonces $\langle c_0; c_1, \sigma\rangle \to^* \langle c_1, \sigma'\rangle$
(2) Si $\langle c, [\sigma | v : [\![e]\!]\sigma]\rangle \to^* \sigma'$, entonces

$$\langle \mathbf{newvar}\ v := e\ \mathbf{in}\ c, \sigma\rangle \to^* [\sigma' | v : \sigma v].$$

(3) Si $\langle c, [\sigma | v : [\![e]\!]\sigma] \rangle \to^* \langle c', \sigma' \rangle$, entonces

$$\langle \textbf{newvar } v := e \textbf{ in } c, \sigma \rangle \to^* \langle \textbf{newvar } v := \sigma' x \textbf{ in } c', [\sigma' | v : \sigma v] \rangle$$

Lema 2.

(1) $\langle c, \sigma \rangle \to \sigma' \Longrightarrow [\![c]\!]\sigma = \sigma'$.
(2) $\langle c, \sigma \rangle \to \langle c', \sigma' \rangle \implies [\![c]\!]\sigma = [\![c']\!]\sigma'$.

Lema 3. $[\![c]\!]\sigma = \sigma' \Longrightarrow \langle c, \sigma \rangle \to^* \sigma'$

Teorema. Para todo comando c se tiene $\{\!\{c\}\!\} = [\![c]\!]$.

8.1. **Semántica operacional de las fallas.** Para dar la semántica de transiciones de LIS con fallas, modificamos el conjunto de configuraciones terminales:

$$\Gamma_t \;=\; \Sigma \;\cup\; \{\textbf{abort}\} \times \Sigma$$

Corresponde además agregar reglas que involucren las nuevas configuraciones terminales:

$$\frac{}{\langle \textbf{fail}, \sigma \rangle \to \langle \textbf{abort}, \sigma \rangle}$$

$$\frac{\langle c_0, \sigma \rangle \to \langle \textbf{abort}, \sigma' \rangle}{\langle c_0; c_1, \sigma \rangle \to \langle \textbf{abort}, \sigma' \rangle}$$

El catchin se define por tres reglas

$$\frac{\langle c_0, \sigma \rangle \to \sigma'}{\langle \textbf{catchin } c_0 \textbf{ with } c_1, \sigma \rangle \to \sigma'}$$

$$\frac{\langle c_0, \sigma \rangle \to \langle \textbf{abort}, \sigma' \rangle}{\langle \textbf{catchin } c_0 \textbf{ with } c_1, \sigma \rangle \to \langle c_1, \sigma' \rangle}$$

$$\frac{\langle c_0, \sigma \rangle \to \langle c_0', \sigma' \rangle}{\langle \textbf{catchin } c_0 \textbf{ with } c_1, \sigma \rangle \to \langle \textbf{catchin } c_0' \textbf{ with } c_1, \sigma' \rangle}$$

Por último, al newvar se le agrega un caso más

$$\frac{\langle c, \sigma \rangle \to \langle \textbf{abort}, \sigma' \rangle}{\langle \textbf{newvar } v := e \textbf{ in } c, \sigma \rangle \to \langle \textbf{abort}, [\sigma' | v : \sigma v] \rangle}$$

El if y el while habían sido definidos con suficiente generalidad para que no requieran revisión.

La relación $\rightarrow$ sigue siendo una función en el lenguaje con fallas, toda configuración γ tiene una única ejecución que puede ser infinita (γ diverge) o terminar en una configuración terminal que puede ser de la forma σ o $\langle \mathbf{abort}, \sigma \rangle$.

Se puede definir:

$$\{\!\{c\}\!\}\sigma = \begin{cases} \bot & \text{si } \langle c, \sigma \rangle \uparrow \\ \sigma' & \text{si existe } \sigma' \text{ tal que } \langle c, \sigma \rangle \rightarrow^* \sigma' \\ \langle \mathbf{abort}, \sigma' \rangle & \text{si existe } \sigma' \text{ tal que } \langle c, \sigma \rangle \rightarrow^* \langle \mathbf{abort}, \sigma' \rangle \end{cases}$$

y obtendremos de manera similar a LIS que para todo comando c se tiene $\{\!\{c\}\!\} = [\![c]\!]$.