Apuntes de Teoría de la Programación

26 de abril de 2021

Contenidos

1 Introducción

1.1. Un ejemplo de la lógica

La pregunta que motiva todo lo siguiente es: ¿qué es el significado? O, más concretamente, ¿cuál es la relación entre la sintaxis y la semántica? Aunque aquí nos centraremos principalmente en especificar el comportamiento de programas, parece conveniente presentar un ejemplo relacionado con la lógica. Recordemos que, en lógica de primer orden, disponíamos de un método para asignar un valor semántico a cada expresión. En este caso, el valor semántico que nos interesa es la *denotación*, es decir, que por ejemplo $\varphi \vee \psi$ denota lo verdadero en función de φ y ψ. El método consistía en:

- Asumir que las fórmulas atómicas tienen una denotación fija, es decir, podemos determinar previamente si es V o F.
- Las denotaciones de $\varphi \vee \psi$, $\neg\varphi$ quedan determinadas por las tablas de verdad correspondientes y el paso anterior.
- La denotación de $\forall x.\varphi$ es V si y solo si, para cada a posible, la denotación de $\varphi[a/x]$ es V.

Con esto ya sabríamos responder a la primera pregunta que nos hicimos para la lógica de primer orden. Sin embargo, esta aproximación no es la única. Pensemos en el concepto de 'prueba' en un sentido computacional. En vez de enfocar el valor semántico hacia la denotación, preguntémonos cuándo un enunciado 'tiene una prueba'. Así obtenemos un método alternativo:

- Asumir que, para las fórmulas atómicas, conocemos lo que significa una prueba. Por ejemplo, la prueba de que $2 + 2 = 4$ consiste en operar con lápiz y papel.
- Una prueba de $\varphi \wedge \psi$ es un par (r, s) donde r es una prueba de φ y donde s es una prueba de ψ.
- Una prueba de $\varphi \vee \psi$ es un par (k, r), donde o bien $r = 0$ y r es prueba de φ o bien $k = 1$ y r es prueba de ψ.
- Una prueba de $\varphi \to \psi$ es una función que lleva pruebas de φ en pruebas de ψ.
- Una prueba de $\neg\varphi$ es una prueba de $\varphi \to \bot$, donde $\bot$ no admite prueba.
- Una prueba de $\forall x.\varphi$ es una función que lleva cada elemento posible a en una prueba de $\varphi[a/x]$.
- Una prueba de $\exists x.\varphi$ es un par (a, r) donde a es un elemento posible y r es una prueba de $\varphi[a/x]$.

¿Qué ha ocurrido aquí? Hemos dado un valor semántico diferente a la sintaxis lógica usual. Nos encontraremos con situaciones similares a ésta a lo largo del curso pero, en vez de hablar de 'proposiciones' y 'fórmulas' trataremos 'programas'.

1.2. Métodos de descripción semántica

Consideremos un programa como $\mathtt{z} := \mathtt{x}; \mathtt{x} := \mathtt{y}; \mathtt{y} := \mathtt{z}$. Un análisis sintáctico nos dice que tenemos tres expresiones separadas por ';' y que cada una tiene la forma de una variable separada de otra por ':='. El análisis semántico depende en gran medida del sintáctico: será entonces conveniente asumir programas que estén sintácticamente bien escritos, como en este ejemplo. La asignación de un valor semántico para tal expresión se tiene que realizar necesariamente en dos pasos:

(i) Dar un significado expresiones separadas por ';'.

(ii) Dar un significado a expresiones formadas por una variable seguida de ':=' y una expresión.

Nosotros nos centraremos en lo sucesivo en tres enfoques distintos (aunque complementarios) para dar significado a expresiones bien formadas.

1.2.1. Semántica operacional

Aquí el valor semántico recae sobre el efecto que tiene el programa sobre la máquina en la que se ejecuta, es decir, una descripción operacional os dirá cómo ejecutar el programa que presentamos antes:

(i) Para ejecutar una serie de expresiones separadas por ';', las ejecutamos una a una de izquierda a derecha.

(ii) Para ejecutar una expresión formada por una variable seguida de ':=' y una variable, determinamos el valor de la segunda variable y se lo damos a la primera.

Por tanto, si ejecutamos el programa $\mathtt{z} := \mathtt{x}; \mathtt{x} := \mathtt{y}; \mathtt{y} := \mathtt{z}$ suponiendo que tenemos las asignaciones iniciales $[\mathtt{x} \mapsto 5, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 0]$, tenemos:

$$\begin{aligned}\langle \mathtt{z} := \mathtt{x}; \mathtt{x} := \mathtt{y}; \mathtt{y} := \mathtt{z}, [\mathtt{x} \mapsto 5, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 0]\rangle &\to \\ \langle \mathtt{x} := \mathtt{y}; \mathtt{y} := \mathtt{z}, [\mathtt{x} \mapsto 5, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 5]\rangle &\to \\ \langle \mathtt{y} := \mathtt{z}, [\mathtt{x} \mapsto 7, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 5]\rangle &\to \\ [\mathtt{x} \mapsto 7, \mathtt{y} \mapsto 5, \mathtt{z} \mapsto 5]&\end{aligned}$$

Esto es lo que se denomina *semántica operacional estructural*. Pero podemos seguir un procedimiento distinto que muestra menos pasos del proceso anterior:

$$\dfrac{\dfrac{\langle \mathtt{z} := \mathtt{x}, s_0\rangle \to s_1 \qquad \langle \mathtt{x} := \mathtt{y}, s_1\rangle \to s_2}{\langle \mathtt{z} := \mathtt{x}; \mathtt{x} := \mathtt{y}, s_0\rangle \to s_2} \qquad \langle \mathtt{y} := \mathtt{z}, s_2\rangle \to s_3}{\langle \mathtt{z} := \mathtt{x}; \mathtt{x} := \mathtt{y}; \mathtt{y} := \mathtt{z}, s_0\rangle \to s_3}$$

Siendo:

$$s_0 := [\mathtt{x} \mapsto 5, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 0], s_1 := [\mathtt{x} \mapsto 5, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 5],$$
$$s_2 := [\mathtt{x} \mapsto 7, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 5], s_3 := [\mathtt{x} \mapsto 5, \mathtt{y} \mapsto 5, \mathtt{z} \mapsto 5].$$

Es decir, aquí hemos resumido toda la información en $\langle e, s\rangle \to t$, que simboliza el hecho de que, al ejecutar la expresión e en el estado s, pasamos al estado t.

1.2.2. Semántica denotacional

En este punto de vista, el valor semántico se encuentra en el efecto de cómo se ejecutan los programas. En el caso que tratamos:

(i) El efecto de una serie de expresiones separadas por ';' consiste en la composición de los efectos de las expresiones.

(ii) El efecto de una expresión formada por una variable seguida por ':=' y otra variable es una función que lleva un estado en uno nuevo, formado a partir del original haciendo que el valor de la primera variable sea el de la segunda.

Es decir, tenemos:

$$\mathcal{S}[\![\mathtt{z} := \mathtt{x}; \mathtt{x} := \mathtt{y}; \mathtt{y} := \mathtt{z}]\!] = \mathcal{S}[\![\mathtt{y} := \mathtt{z}]\!] \circ \mathcal{S}[\![\mathtt{x} := \mathtt{y}]\!] \circ \mathcal{S}[\![\mathtt{z} := \mathtt{x}]\!]$$

Y por tanto,

$$\begin{aligned}
&\mathcal{S}[\![\mathtt{z} := \mathtt{x}; \mathtt{x} := \mathtt{y}; \mathtt{y} := \mathtt{z}]\!]([\mathtt{x} \mapsto 5, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 0]) \\
=&\mathcal{S}[\![\mathtt{y} := \mathtt{z}]\!](\mathcal{S}[\![\mathtt{x} := \mathtt{y}]\!](\mathcal{S}[\![\mathtt{z} := \mathtt{x}]\!]([\mathtt{x} \mapsto 5, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 0]))) \\
=&\mathcal{S}[\![\mathtt{y} := \mathtt{z}]\!](\mathcal{S}[\![\mathtt{x} := \mathtt{y}]\!]([\mathtt{x} \mapsto 5, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 5])) \\
=&\mathcal{S}[\![\mathtt{y} := \mathtt{z}]\!]([\mathtt{x} \mapsto 7, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 5]) \\
=&[\mathtt{x} \mapsto 7, \mathtt{y} \mapsto 5, \mathtt{z} \mapsto 5].
\end{aligned}$$

Nótese lo que hemos conseguido aquí: hemos traducido el funcionamiento del programa a una serie de objetos matemáticos. Esto será de ayuda en el futuro.

1.2.3. Semántica axiomática

Dado un programa, decimos que es *parcialmente correcto* respecto de una premisa y una consecuencia si, cuando el estado inicial verifica la premisa y el programa termina, el estado final verifica la consecuencia. En nuestro caso, tenemos la propiedad:

$$\{\mathtt{x} = \mathtt{n} \wedge \mathtt{y} = \mathtt{m}\}\mathtt{z} := \mathtt{x}; \mathtt{x} := \mathtt{y}; \mathtt{y} := \mathtt{z}\{\mathtt{y} = \mathtt{n} \wedge \mathtt{x} = \mathtt{m}\}$$

Nótese que esta propiedad no asegura que el programa termine. Desde este punto de vista, lo que queremos es construir un sistema lógico para demostrar la corrección parcial de un programa. En cambio, como veremos, ciertos aspectos de los programas no serán tenidos en cuenta. La deducción de la corrección parcial del programa de ejemplo es la siguiente:

$$\dfrac{\dfrac{\{p_0\}\mathtt{z} := \mathtt{x}\{p_1\} \qquad \{p_1\}\mathtt{x} := \mathtt{y}\{p_2\}}{\{p_0\}\mathtt{z} := \mathtt{x}; \mathtt{x} := \mathtt{y}\{p_2\}} \qquad \{p_2\}\mathtt{y} := \mathtt{z}\{p_3\}}{\{p_0\}\mathtt{z} := \mathtt{x}; \mathtt{x} := \mathtt{y}; \mathtt{y} := \mathtt{z}\{p_3\}}$$

Donde

$$p_0 := \mathtt{x} = \mathtt{n} \wedge \mathtt{y} = \mathtt{m}, p_1 := \mathtt{z} = \mathtt{n} \wedge \mathtt{y} = \mathtt{m},$$

$$p_2 := \mathtt{z} = \mathtt{n} \wedge \mathtt{x} = \mathtt{m}, p_3 := \mathtt{y} = \mathtt{n} \wedge \mathtt{x} = \mathtt{m}.$$

Aunque se pueda observar cierta similitud con el enfoque operacional, la diferencia es que aquí trabajamos con aserciones que no tienen en cuenta el funcionamiento del programa. La ventaja de esto consiste en que podemos describir fácilmente determinadas propiedades de tal programa.

1.3. El lenguaje While

Veamos a continuación un ejemplo que iremos desarrollando durante el curso, el lenguaje **While**. Para especificar las *categorías sintácticas*, especificamos una *metavariable* específica que toma valores en los elementos de cada una[1]:

- Numerales: $n \in \mathbf{Num}$.
- Variables: $x \in \mathbf{Var}$.

[1] Formalmente, trataremos a estas colecciones como *tipos básicos*. En el futuro trabajaremos tanto con estos tipos como con los *tipos funcionales*, es decir, de aplicaciones de un tipo en otro.

- Expresiones aritméticas: $a \in \mathbf{Aexp}$.
- Expresiones booleanas: $b \in \mathbf{Bexp}$.
- Sentencias: $S \in \mathbf{Stm}$.

En caso de que hiciera falta emplear más de una metavariable, emplearemos, por ejemplo, $n', n'', \ldots$ y $n_1, n_2, \ldots$. Supondremos que las categorías **Num** y **Var** se construyen de manera natural. Las categorías sintácticas **Aexp**, **Bexp** y **Stm** se construyen, respectivamente, de la siguiente forma:

$$\begin{aligned} a &::= n \mid x \mid a_1 + a_2 \mid a_1 \times a_2 \mid a_1 - a_2 \\ b &::= \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \\ S &::= x := a \mid \texttt{skip} \mid S_1; S_2 \mid \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \mid \texttt{while } b \texttt{ do } S \end{aligned}$$

Si volvemos al ejemplo de antes, la expresión $\texttt{z} := \texttt{x}; \texttt{x} := \texttt{y}; \texttt{y} := \texttt{z}$, podemos definir una *sintaxis concreta*, en el sentido de que, de los diferentes árboles de derivación para tal expresión, debemos escoger uno. En cambio, lo que acabamos de definir arriba es un ejemplo de *sintaxis abstracta*, y los árboles de derivación posibles son todos distintos elementos de la categoría **Stm**. De hecho, en caso de que queramos expresar la prioridad de las operaciones, lo haremos mediante el uso de paréntesis.

1.4. Semántica para expresiones

Veamos, a modo de ejemplo, cómo dar significado a los numerales. Expresaremos la gramática de **Num** por

$$n ::= \texttt{0} \mid \texttt{1} \mid n\ \texttt{0} \mid n\ \texttt{1}$$

Intepretaremos los numerales como si fuesen la expresión binaria de un número natural. Es decir, se tendrá una *aplicación semántica* $\mathcal{N} : \mathbf{Num} \to \mathbb{Z}$ dada recursivamente por:

$$\begin{aligned} \mathcal{N}[\![\texttt{0}]\!] &= 0 \\ \mathcal{N}[\![\texttt{1}]\!] &= 1 \\ \mathcal{N}[\![n\texttt{0}]\!] &= 2 \otimes \mathcal{N}[\![n]\!] \\ \mathcal{N}[\![n\texttt{1}]\!] &= 2 \otimes \mathcal{N}[\![n]\!] \oplus 1 \end{aligned}$$

Donde $\oplus, \otimes$ expresan las operaciones de suma y producto en $\mathbb{Z}$ y $0, 1, 2$ los enteros correspondientes: la distinción entre objetos sintácticos y semánticos tiene que ser cuidadosa. Las igualdades anteriores se llaman *ecuaciones semánticas*, nos indican cómo asociar un objeto matemático a un símbolo. Además, tenemos aquí un ejemplo de lo que se llama el *principio de composición*, es decir, construimos el significado de una expresión (*elemento compuesto*) en función del de sus componentes (*elementos base*). Ésto facilita aplicar el método de demostración general que seguiremos, la *inducción estructural*, que consiste en primero demostrar una propiedad para cada elemento base y después demostrarla para los compuestos empleando la hipótesis de inducción. Veamos un ejemplo a continuación.

Primero recordemos que una función $f : A \to B$ es *parcial* si hay elementos $a \in A$ en los que no está definida. En caso contrario, se denomina *función total*.

Proposición 1.1. *La función $\mathcal{N} : \mathbf{Num} \to \mathbb{Z}$ es total.*

Demostración. Por inducción. De los siguientes casos, $(a), (b)$ son los casos base y $(c), (d)$ los *inductivos*:

(a) Si $n = \texttt{0}$, $\mathcal{N}[\![n]\!] = 0$.

(b) Si $n = \texttt{1}$, $\mathcal{N}[\![n]\!] = 1$.

(c) Si $n = m\texttt{0}$, $\mathcal{N}[\![n]\!] = \mathcal{N}[\![m\texttt{0}]\!] = 2 \otimes \mathcal{N}[\![m]\!]$, y por hipótesis de inducción tenemos el resultado.

(d) Análogo a (c).

□

Ejemplo 1.2. Definamos una gramática y una semántica asociada para interpretar los numerales (binarios), de modo que el primer caracter de cada cadena represente el signo (positivo o negativo) del número representado por el resto de la misma. Una gramática posible es:

$$n ::= \mathtt{0} \mid \mathtt{1} \mid \mathtt{0}n \mid \mathtt{1}n$$

Definiremos su semántica atendiendo a que las cadenas $\mathtt{0}$ y $\mathtt{1}$ representan el número 0, pues se compondrían solo del signo, sin acompañar ningún número.

La función semántica es $\mathcal{S} : \mathbf{Num} \to \mathbb{Z}$, definida por:

$$\begin{aligned}
\mathcal{S}[\![\mathtt{0}]\!] &= 0 \\
\mathcal{S}[\![\mathtt{1}]\!] &= 0 \\
\mathcal{S}[\![\mathtt{0}n]\!] &= \mathcal{N}[\![n]\!] \\
\mathcal{S}[\![\mathtt{1}n]\!] &= -\mathcal{N}[\![n]\!]
\end{aligned}$$

Notemos el siguiente detalle: en principio, la función $\mathcal{N}$, como tal, no puede tomar valores del modo anterior. Sin embargo, se puede probar que, como la gramática que dimos en la definición de $\mathcal{N}$ y la que hemos escrito arriba generan las mismas cadenas, la función $\mathcal{N}$ se puede identificar fácilmente con la función que buscamos.

Pese a ello es posible definir dicha función de la siguiente forma

$$\begin{aligned}
\mathcal{N}[\![\mathtt{0}]\!] &= 0 \\
\mathcal{N}[\![\mathtt{1}]\!] &= 1 \\
\mathcal{N}[\![\mathtt{0}n]\!] &= \mathcal{AUX}[\![n, 0]\!] \\
\mathcal{N}[\![\mathtt{1}n]\!] &= \mathcal{AUX}[\![n, 1]\!]
\end{aligned}$$

y la función $\mathcal{AUX} : \mathbf{Num} \times \mathbb{N} \to \mathbb{N}$

$$\begin{aligned}
\mathcal{AUX}[\![\mathtt{0}, x]\!] &= 2 \otimes x \\
\mathcal{AUX}[\![\mathtt{1}, x]\!] &= 2 \otimes x \oplus 1 \\
\mathcal{AUX}[\![\mathtt{0}n, x]\!] &= \mathcal{AUX}[\![n, 2 \otimes x]\!] \\
\mathcal{AUX}[\![\mathtt{1}n, x]\!] &= \mathcal{AUX}[\![n, 2 \otimes x \oplus 1]\!]
\end{aligned}$$

donde el segundo parámetro representa un acumulador. Por construcción de la gramática la lectura de la cadena se hace de izquierda a derecha, dificultando un poco la construcción de la semántica.

Desde la perspectiva de la semántica denotacional, el significado de una expresión está determinado por los valores que toman en ella las variables. Esto motiva el concepto de *estado*, definido en nuestro caso como un elemento del conjunto $\mathbf{State} := \mathbf{Var} \to \mathbb{Z}$, es decir, como una función que lleva una variable en su valor (un entero positivo). Por tanto, el significado de la expresión viene dado por una función auxiliar $\mathcal{A} : \mathbf{Aexp} \to (\mathbf{State} \to \mathbb{Z})$, donde $\mathcal{A}$ toma una exprexión aritmética y un estado s[2]:

$$\begin{aligned}
\mathcal{A}[\![n]\!]s &= \mathcal{N}[\![n]\!] \\
\mathcal{A}[\![x]\!]s &= s\ x \\
\mathcal{A}[\![a_1 + a_2]\!]s &= \mathcal{A}[\![a_1]\!]s \oplus \mathcal{A}[\![a_2]\!]s \\
\mathcal{A}[\![a_1 \times a_2]\!]s &= \mathcal{A}[\![a_1]\!]s \otimes \mathcal{A}[\![a_2]\!]s \\
\mathcal{A}[\![a_1 - a_2]\!]s &= \mathcal{A}[\![a_1]\!]s \ominus \mathcal{A}[\![a_2]\!]s
\end{aligned}$$

Ejemplo 1.3. Podemos añadir a la definición la ecuación semántica

$$\mathcal{A}[\![-a]\!]s = 0 \ominus \mathcal{A}[\![a]\!]s$$

Incluso podemos prescindir del 0 por cómo está definida $\ominus$. En cambio,

$$\mathcal{A}[\![-a]\!]s = \mathcal{A}[\![\mathtt{0} - a]\!]s$$

no está definida de forma composicional.

[2]Nótese que $\mathcal{A}$ nos lleva a a una función $\mathcal{A}[\![a]\!]$ y aplicamos tal función a s escribiendo $\mathcal{A}[\![a]\!]s$. Por otro lado, con $s\,x$ nos referimos a s aplicado a x.

Se puede repetir el procedimiento anterior para definir una función semántica para los booleanos, $\mathcal{B} : \mathbf{Bexp} \to (\mathbf{State} \to \mathit{Bool})$, siendo $\mathit{Bool} := \{\mathbf{tt}, \mathbf{ff}\}$, definida por:

$$\mathcal{B}[\![\texttt{true}]\!]s = \mathbf{tt}$$
$$\mathcal{B}[\![\texttt{false}]\!]s = \mathbf{ff}$$
$$\mathcal{B}[\![a_1 = a_2]\!]s = \begin{cases} \mathbf{tt}, \text{ si } \mathcal{A}[\![a_1]\!]s \text{ es igual a } \mathcal{A}[\![a_2]\!]s \\ \mathbf{ff}, \text{ en otro caso} \end{cases}$$
$$\mathcal{B}[\![a_1 \leq a_2]\!]s = \begin{cases} \mathbf{tt}, \text{ si } \mathcal{A}[\![a_1]\!]s \text{ es menor que } \mathcal{A}[\![a_2]\!]s \\ \mathbf{ff}, \text{ en otro caso} \end{cases}$$
$$\mathcal{B}[\![\neg b]\!]s = \begin{cases} \mathbf{tt}, \text{ si } \mathcal{B}[\![b]\!]s \text{ es } \mathbf{ff} \\ \mathbf{ff}, \text{ en otro caso} \end{cases}$$
$$\mathcal{B}[\![b_1 \wedge b_2]\!]s = \begin{cases} \mathbf{tt}, \text{ si } \mathcal{B}[\![b_1]\!]s \text{ es } \mathbf{tt} \text{ y } \mathcal{B}[\![b_2]\!]s \text{ es } \mathbf{tt} \\ \mathbf{ff}, \text{ en otro caso} \end{cases}$$

De nuevo, por inducción estructural, es fácil demostrar el siguiente resultado, que es análogo al que vimos para los numerales:

Proposición 1.4. *La función $\mathcal{B} : \mathbf{Bexp} \to (\mathbf{State} \to \mathit{Bool})$ es total.*

El siguiente ejemplo ilustra cómo podemos extender una categoría sintáctica (de forma cuidadosa):

Ejemplo 1.5. Consideremos la extensión $\mathbf{Bexp}'$ de **Bexp**:

$$b ::= \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \neq a_2 \mid a_1 \leq a_2 \mid a_1 \geq a_2 \mid a_1 < a_2 \mid a_1 > a_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid b_1 \Rightarrow b_2 \mid b_1 \Leftrightarrow b_2$$

Dos expresiones booleanas b_1, b_2 se dicen *equivalentes* si, para cada estado s, $\mathcal{B}[\![b_1]\!]s = \mathcal{B}[\![b_2]\!]s$. Veamos que, dada una expresión $b' \in \mathbf{Bexp}'$, existe $b \in \mathbf{Bexp}$ equivalente a b'. La demostración consiste en dos pasos: (i) Dar un valor semántico a cada expresión de la extensión, (ii) Comprobar que podemos expresar el valor semántico de b' mediante b, empleando las igualdades sintácticas naturales.

- Si b' es una expresión de **Bexp**, $b := b'$.
- Si b' es de la forma $a_1 \neq a_2$, tomamos b como $\neg(a_1 = a_2)$.
- Si b' es de la forma $a_1 \geq a_2$, tomamos b como $a_2 \leq a_1$.
- Si b' es de la forma $a_1 < a_2$, tomamos b como $(a_1 \leq a_2) \wedge \neg(a_1 = a_2)$.
- Si b' es de la forma $a_1 > a_2$, tomamos b como $(a_2 \leq a_1) \wedge \neg(a_1 = a_2)$.
- Si b' es de la forma $b_1 \vee b_2$, tomamos b como $\neg(\neg b_1 \wedge \neg b_2)$.
- Si b' es de la forma $b_1 \Rightarrow b_2$, tomamos b como $\neg(b_1 \wedge \neg b_2)$.
- Si b' es de la forma $b_1 \Leftrightarrow b_2$, tomamos b como $\neg(b_1 \wedge \neg b_2) \wedge \neg(b_2 \wedge \neg b_1)$.

Notemos que podríamos haber razonado inductivamente, pero para mayor claridad hemos indicado cuál es la traducción concreta de cada expresión.

1.5. Propiedades semánticas

En esta sección introducimos dos conceptos fundamentales que son comunes a la lógica.

1.5.1. Variables libres

Dada una expresión aritmética a, su conjunto de *variables libres*, $\mathrm{FV}(a) \subseteq \mathbf{Var}$, se define composicionalmente como:

$$\begin{aligned}
\mathrm{FV}(n) &= \emptyset \\
\mathrm{FV}(x) &= \{x\} \\
\mathrm{FV}(a_1 + a_2) &= \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2) \\
\mathrm{FV}(a_1 \times a_2) &= \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2) \\
\mathrm{FV}(a_1 - a_2) &= \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2)
\end{aligned}$$

El siguiente resultado nos dice que $\mathrm{FV}(a)$ determina el valor semántico de a:

Lema 1.6. *Sea $a \in$ **Aexp**. Sean $s, s' \in$ **State** tales que, para cada $x \in \mathrm{FV}(a)$, $s\ x = s'\ x$. Entonces $\mathcal{A}[\![a]\!]s = \mathcal{A}[\![a]\!]s'$.*

Demostración. Veamos los casos base:

- Si $a := n$, sabemos que $\mathcal{A}[\![a]\!]s := \mathcal{N}[\![n]\!] =: \mathcal{A}[\![a]\!]s'$.
- Si $a := x$, entonces, como $x \in \mathrm{FV}(a)$, por hipótesis tenemos que $\mathcal{A}[\![a]\!]s := s\ x = s'\ x := \mathcal{A}[\![n]\!]s'$.

Los casos inductivos son:

- Si a es de la forma $a_1 + a_2$, $\mathcal{A}[\![a]\!]s := \mathcal{A}[\![a_1]\!]s + \mathcal{A}[\![a_2]\!]s$ y $\mathcal{A}[\![a]\!]s' := \mathcal{A}[\![a_1]\!]s' + \mathcal{A}[\![a_2]\!]s'$. Como $\mathrm{FV}(a_i) \subseteq \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2) = \mathrm{FV}(a_1 + a_2)$, por la hipótesis de inducción aplicada a a_i, tenemos que $\mathcal{A}[\![a_i]\!]s = \mathcal{A}[\![a_i]\!]s'$, para $i = 1, 2$. Entonces,

  $$\mathcal{A}[\![a_1 + a_2]\!]s = \mathcal{A}[\![a_1]\!]s + \mathcal{A}[\![a_2]\!]s = \mathcal{A}[\![a_1]\!]s' + \mathcal{A}[\![a_2]\!]s' = \mathcal{A}[\![a_1 + a_2]\!]s',$$

  como queríamos.
- Para $a_1 * a_2$ y $a_1 - a_2$ basta repetir lo anterior (ya que el conjunto de variables libres es el mismo).

□

De la misma forma, para expresiones booleanas, tenemos:

$$\begin{aligned}
\mathrm{FV}(\texttt{true}) &= \emptyset \\
\mathrm{FV}(\texttt{true}) &= \emptyset \\
\mathrm{FV}(a_1 = a_2) &= \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2) \\
\mathrm{FV}(a_1 \leq a_2) &= \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2) \\
\mathrm{FV}(\neg b) &= \mathrm{FV}(b) \\
\mathrm{FV}(b_1 \wedge b_2) &= \mathrm{FV}(b_1) \cup \mathrm{FV}(b_2)
\end{aligned}$$

La demostración del anterior lema se puede repetir de nuevo:

Lema 1.7. *Sea $b \in$ **Bexp**. Sean $s, s' \in$ **State** tales que, para cada $x \in \mathrm{FV}(b)$, $s\ x = s'\ x$. Entonces $\mathcal{B}[\![b]\!]s = \mathcal{B}[\![b]\!]s'$.*

Demostración. Casos base:

- Si $b :=$ `true`, $\mathcal{B}[\![b]\!]s := V =: \mathcal{B}[\![b]\!]s'$ y análogamente con `false`.
- Si b es de la forma $a_1 = a_2$, con $a_1, a_2 \in$ **Aexp**, sabemos que

  $$\mathcal{B}[\![a_1 = a_2]\!]s = \begin{cases} V, \text{ si } \mathcal{A}[\![a_1]\!]s \text{ es igual a } \mathcal{A}[\![a_2]\!]s \\ F, \text{ en otro caso} \end{cases} \quad \text{y que } \mathcal{B}[\![a_1 = a_2]\!]s' = \begin{cases} V, \text{ si } \mathcal{A}[\![a_1]\!]s' \text{ es igual a } \mathcal{A}[\![a_2]\!]s' \\ F, \text{ en otro caso} \end{cases}$$

  Como suponemos que para cada $x \in \mathrm{FV}(b)$, $s\ x = s'\ x$ y $\mathrm{FV}(a_1), \mathrm{FV}(a_2) \subseteq \mathrm{FV}(b)$, se sigue que se verifican las hipótesis del lema anterior, y que por tanto $\mathcal{A}[\![a_i]\!]s = \mathcal{A}[\![a_i]\!]s'$, para $i = 1, 2$. Ahora bien, $\mathcal{B}[\![b]\!]s$ es V si y solo si $\mathcal{A}[\![a_1]\!]s$ es igual a $\mathcal{A}[\![a_2]\!]s$ y, por lo que acabamos de decir, esto es cierto si y solo si $\mathcal{A}[\![a_1]\!]s'$ es igual a $\mathcal{A}[\![a_2]\!]s'$, que es precisamente equivalente a que $\mathcal{B}[\![b]\!]s'$ sea V.

- Si b es de la forma $a_1 \leq a_2$, con con $a_1, a_2 \in \mathbf{Aexp}$, el procedimiento es análogo al anterior.

Para los casos inductivos tenemos:

- Si b es de la forma $\neg b'$, para cierta $b' \in$ Bexp, sabemos que entonces $\mathrm{FV}(b) = \mathrm{FV}(b')$. Como suponemos que, para cada $x \in \mathrm{FV}(b)$, $s\ x = s'\ x$, podemos aplicar la hipótesis de inducción, y entonces obtenemos que $\mathcal{B}[\![b]\!]s$ es V si y solo si $\mathcal{B}[\![b']\!]s = \mathcal{B}[\![b']\!]s'$ es V, que es equivalente a que $\mathcal{B}[\![b]\!]s'$ sea V. Por tanto, $\mathcal{B}[\![b]\!]s = \mathcal{B}[\![b]\!]s'$.
- Si b es de la forma $b_1 \wedge b_2$, para ciertas $b_1, b_2 \in$ Bexp, sabemos que entonces $\mathrm{FV}(b) = \mathrm{FV}(b_1) \cup \mathrm{FV}(b_2)$ y, siguiendo los razonamientos que ya hemos hecho antes, podemos aplicar la hipótesis de inducción sobre b_1 y b_2. Entonces $\mathcal{B}[\![b]\!]s$ es V si y solo si $\mathcal{B}[\![b_1]\!]s$ es V y $\mathcal{B}[\![b_2]\!]s$ es v, que equivale a que $\mathcal{B}[\![b_1]\!]s'$ sea V y $\mathcal{B}[\![b_2]\!]s'$ sea V, que es cierto si y solo si $\mathcal{B}[\![b]\!]s'$ es V y, por tanto, $\mathcal{B}[\![b]\!]s = \mathcal{B}[\![b]\!]s'$.

□

1.5.2. Sustitución

Si tenemos dos expresiones aritméticas a, a_0 y $x \in \mathrm{FV}(a)$, entonces denotamos por $a[x \mapsto a_0]$ a la expresión obtenida al *sustituir* cada ocurrencia de x en a por a_0. Se define composicionalmente como:

$$\begin{aligned}
n[x \mapsto a_0] &= n \\
y[x \mapsto a_0] &= \begin{cases} a_0, \text{ si } x = y \\ y, \text{ si } x \neq y \end{cases} \\
(a_1 + a_2)[x \mapsto a_0] &= a_1[x \mapsto a_0] + a_2[x \mapsto a_0] \\
(a_1 \times a_2)[x \mapsto a_0] &= a_1[x \mapsto a_0] \times a_2[x \mapsto a_0] \\
(a_1 - a_2)[x \mapsto a_0] &= a_1[x \mapsto a_0] - a_2[x \mapsto a_0]
\end{aligned}$$

También podemos definir la sustitución en relación con los estados:

$$(s[y \mapsto v])x := \begin{cases} v, \text{ si } x = y \\ s\ x, \text{ si } x \neq y \end{cases}$$

La relación entre ambos conceptos se muestra en el siguiente resultado:

Lema 1.8. *Dadas $a, a_0 \in$ **Aexp**, para todo $s \in$ **State** se cumple que*

$$\mathcal{A}[\![a[y \mapsto a_0]]\!]s = \mathcal{A}[\![a]\!](s[y \mapsto \mathcal{A}[\![a_0]\!]s]).$$

Demostración. De nuevo, una demostración rutinaria por inducción estructural. Los casos base son los siguientes:

- Si $a := n$, $\mathcal{A}[\![a[y \mapsto a_0]]\!]s = \mathcal{A}[\![n]\!]s = \mathcal{N}[\![n]\!] = \mathcal{A}[\![a]\!]([s \mapsto \mathcal{A}[\![a_0]\!]s])$.
- Si $a := x$, entonces

$$\mathcal{A}[\![a[y \mapsto a_0]]\!]s = \begin{cases} \mathcal{A}[\![a_0]\!]s & x = y \\ \mathcal{A}[\![x]\!]s & x \neq y \end{cases}$$

$$\mathcal{A}[\![a]\!](s[y \mapsto \mathcal{A}[\![a_0]\!]s]) = (s[y \mapsto \mathcal{A}[\![a_0]\!]s])\ x = \begin{cases} \mathcal{A}[\![a_0]\!]s & x = y \\ s\ x & x \neq y \end{cases}$$

- Si $a := a_1 + a_2$ con a_1, a_2 cumpliendo la proposición. Se tiene

$$\mathcal{A}[\![a_i[y \mapsto a_0]]\!]s = \mathcal{A}[\![a_i]\!](s[y \mapsto \mathcal{A}[\![a_0]\!]s]) = \mathcal{A}[\![a_i]\!]s'$$

para $i \in \{1, 2\}$. Se denota $s' := (s[y \mapsto \mathcal{A}[\![a_0]\!]s])$. Entonces

$$\begin{aligned}
\mathcal{A}[\![(a_1 + a_2)[y \mapsto a_0]]\!]s &= \mathcal{A}[\![a_1[y \mapsto a_0] + a_2[y \mapsto a_0]]\!]s \\
&= \mathcal{A}[\![a_1[y \mapsto a_0]]\!]s \oplus \mathcal{A}[\![a_2[y \mapsto a_0]]\!]s \\
&\overset{hip.ind.}{=} \mathcal{A}[\![a_1]\!]s' \oplus \mathcal{A}[\![a_2]\!]s' \\
&= \mathcal{A}[\![a_1 + a_2]\!]
\end{aligned}$$

- El caso $a := a_1 \times a_2$ es análogo.

□

En general, si tratamos con conjuntos de variables, necesitaremos lo siguiente. Sea $X \subseteq \mathbf{Var}$ y $s, s' \in \mathbf{State}$. Dada $\mathtt{x} \in \mathbf{Var}$, definimos la *sustitución múltiple en X* como:

$$(s'[X \mapsto s])\ x := \begin{cases} s\ x, \text{ si } x \in X \\ s'\ x, \text{ en otro caso} \end{cases}$$

Todo lo anterior justifica una noción que será importante a lo largo del curso. Dada una categoría sintáctica **Cat**, dos expresiones $b_1, b_2 \in \mathbf{Cat}$ y una función semántica $\mathcal{C} : \mathbf{Cat} \to C$, se dice que b_1, b_2 son *semánticamente equivalentes* si para todo $s \in \mathbf{State}$ se tiene

$$\mathcal{C}[\![b_1]\!]s = \mathcal{C}[\![b_2]\!]s.$$

De todos modos, veremos que esta noción depende en gran medida del conjunto de reglas que empleemos.

2 | Semántica operacional

En el anterior capítulo hemos visto cómo dar un valor semántico al lenguaje While mediante el punto de vista de la semántica denotacional. Centrémonos ahora en la semántica operacional. La distinción fundamental que hacíamos de este enfoque es la siguiente:

- Semántica operacional *natural*, que describe cómo se han obtenido los resultados generales de las ejecuciones.
- Semántica operacional *estructural*, que describe cómo se ha obtenido cada paso en la ejecución.

Para ambos tipos de semántica operacional, el valor semántico de cada expresión será especificado por un *sistema de transiciones*, compuesto de dos configuraciones distintas:

$\langle S, s\rangle$, que denota que la expresión S se ejecutará desde el estado s.

s, que denota un estado terminal. Las *configuraciones terminales* tendrán esta forma.

Finalmente, es necesaria una *relación de transición* que describa cómo tiene lugar la ejecución. La diferencia entre las dos semánticas se encuentra principalmente en ésta. De hecho, veremos que ambos tipos de semántica son, en cierto sentido, equivalentes.

2.1. Semántica operacional natural

2.1.1. Sistema de transiciones

La relación de transición $\langle S, s\rangle \to s'$ se puede leer como que, la ejecución de S desde el estado s terminará y el nuevo estado será s'. Está determinada por las siguientes reglas[1]:

Sistema (While_{ns})**.**

[ass_{ns}]

$$\frac{}{\langle x := a, s\rangle \to s[x \mapsto \mathcal{A}[\![a]\!]s]}$$

[skip_{ns}]

$$\frac{}{\langle \texttt{skip}, s\rangle \to s}$$

[comp_{ns}]

$$\frac{\langle S_1, s\rangle \to s' \qquad \langle S_2, s'\rangle \to s''}{\langle S_1; S_2, s\rangle \to s''}$$

[1] Nótese que las variables S_1, S_2, s, s', s'', etc. son libres.

$[\text{if}^{\text{tt}}_{\text{ns}}]$

$$\frac{\langle S_1, s\rangle \to s'}{\langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, s\rangle \to s'} \quad \text{si } \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

$[\text{if}^{\text{ff}}_{\text{ns}}]$

$$\frac{\langle S_2, s\rangle \to s'}{\langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, s\rangle \to s'} \quad \text{si } \mathcal{B}[\![b]\!]s = \mathbf{ff}$$

$[\text{while}^{\text{tt}}_{\text{ns}}]$

$$\frac{\langle S, s\rangle \to s' \qquad \langle \texttt{while } b \texttt{ do } S, s'\rangle \to s''}{\langle \texttt{while } b \texttt{ do } S, s\rangle \to s''} \quad \text{si } \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

$[\text{while}^{\text{ff}}_{\text{ns}}]$

$$\frac{}{\langle \texttt{while } b \texttt{ do } S, s\rangle \to s} \quad \text{si } \mathcal{B}[\![b]\!]s = \mathbf{ff}$$

Aclaremos un poco la terminología:

Definición 2.1. Una *regla* en general tiene la forma general

$$\frac{\langle S_1, s_1\rangle \to s_1' \ldots \langle S_n, s_n\rangle \to s_n'}{\langle S, s\rangle \to s'} \quad \text{si } \varphi$$

donde los términos que aparecen encima y bajo la línea son, respectivamente, las *premisas* y la *conclusión*, y donde φ es la *condición*. Cuando empleamos las reglas anteriores para obtener una transición $\langle S, s\rangle \to s'$, obtenemos un *árbol de derivación*. Una regla sin premisas se llama *axioma*.

Consideremos el problema de construir un árbol de derivación para una expresión S y un estado s. El método general consiste en partir de la 'raíz' y encontrar las 'hojas', es decir, el paso inicial consiste en buscar una regla de modo que su conclusión tenga la ejecución $\langle S, s\rangle$ en su parte izquierda. Los pasos inductivos son:

- Si la regla encontrada es un axioma, entonces podemos determinar el estado terminal y terminamos.
- Si la regla encontrada no es un axioma, entonces el siguiente paso consiste en buscar un árbol de derivación para sus premisas.

Nótese que, en cada paso, las condiciones para aplicar cada regla tienen que ser verificadas. En el futuro demostraremos algo que parece falso a primera vista: que en el lenguaje While hay a lo sumo un árbol de derivación posible para cada ejecución $\langle S, s\rangle$.

Definición 2.2. Decimos que una ejecución de la expresión S desde el estado s, $\langle S, s\rangle$, *termina* si existe un estado s' tal que $\langle S, s\rangle \to s'$. Si tal estado no existe entonces decimos que la ejecución *cicla*. Para una expresión S, decimos que *siempre termina* si $\langle S, s\rangle$ termina para cada elección de s y que *siempre cicla* si $\langle S, s\rangle$ cicla para cada elección de s.

Ejemplo 2.3. Podemos tratar de determinar si las siguientes expresiones terminan o ciclan siempre:

1. $\texttt{while } \neg(\texttt{x} = 1) \texttt{ do } (\texttt{y} := \texttt{y} \times \texttt{x}; \texttt{x} := \texttt{x} - 1)$.
2. $\texttt{while } 1 \leq \texttt{x} \texttt{ do } (\texttt{y} := \texttt{y} \times \texttt{x}; \texttt{x} := \texttt{x} - 1)$.
3. `while true do skip`.

La primera para si se inicializa x con un valor mayor o igual que 1 y cicla en caso contrario: nótese que si $\mathtt{x} < 1$ decrecerá continuamente, luego es imposible que la condición del bucle llegue a no cumplirse.

La segunda para siempre. Es parecida a la anterior salvo por el hecho de que la condición del bucle hace que, para $\mathtt{x} < 1$, se pare.

La tercera dejaría intacta la configuración inicial. Sin embargo, la ejecución cicla pese a no realizar ninguna acción. Esto es porque la condición del bucle es siempre cierta.

2.1.2. Propiedades

El sistema de transición nos da un entorno en el que estudiar las propiedades de las expresiones. Veamos a continuación una definición precisa de un concepto que introdujimos al final de la introducción:

Definición 2.4. Dos expresiones S_1, S_2 se dicen *semánticamente equivalentes* si para cada par $s, s' \in \mathbf{State}$,

$$\langle S_1, s\rangle \to s' \text{ si y solo si } \langle S_2, s\rangle \to s'.$$

Lema 2.5. $\mathtt{while}\ b\ \mathtt{do}\ S$ *es semánticamente equivalente a* $\mathtt{if}\ b\ \mathtt{then}\ (S; \mathtt{while}\ b\ \mathtt{do}\ S)\ \mathtt{else\ skip}$.

Demostración. Dividimos la prueba en dos implicaciones:
Parte 1. Supongamos que se cumple $\langle \mathtt{while}\ b\ \mathtt{do}\ S, s\rangle \to s''$. Entonces existe un árbol de derivación para él, T. T puede tener dos formas en función de la regla que hayamos aplicado: o bien hemos aplicado la regla o el axioma $[\text{while}^{\text{ff}}_{\text{ns}}]$. Veamos cada caso:

(a) Si hemos aplicado la regla $[\text{while}^{\text{tt}}_{\text{ns}}]$, T es de la forma:

$$[\text{while}^{\text{tt}}_{\text{ns}}]\ \frac{\overset{\cdots}{\langle S, s\rangle \to s'} \qquad \overset{\cdots}{\langle \mathtt{while}\ b\ \mathtt{do}\ S, s'\rangle \to s''}}{\langle \mathtt{while}\ \ b\ \ \mathtt{do}\ \ S, s\rangle \to s''}$$

con $\mathcal{B}[\![b]\!]s = \mathbf{tt}$. Ahora bien, notemos que:

$$[\text{comp}_{\text{ns}}]\ \frac{\overset{\cdots}{\langle S, s\rangle \to s'} \qquad \overset{\cdots}{\langle \mathtt{while}\ b\ \mathtt{do}\ S, s'\rangle \to s''}}{\langle S; \mathtt{while}\ \ b\ \ \mathtt{do}\ \ S, s\rangle \to s''}$$

Usando que $\mathcal{B}[\![b]\!]s = \mathbf{tt}$, podemos aplicar:

$$[\text{if}^{\text{ff}}_{\text{ns}}]\ \frac{\langle S; \mathtt{while}\ \ b\ \ \mathtt{do}\ \ S, s\rangle \to s''}{\langle \mathtt{if}\ b\ \mathtt{then}\ (S; \mathtt{while}\ \ b\ \ \mathtt{do}\ \ S)\ \mathtt{else\ skip}, s\rangle \to s''}$$

Y por tanto, $\langle \mathtt{if}\ b\ \mathtt{then}\ (S; \mathtt{while}\ \ b\ \ \mathtt{do}\ \ S)\ \mathtt{else\ skip}, s\rangle \to s''$.

(b) Si hemos aplicado la regla $[\text{while}^{\text{ff}}_{\text{ns}}]$, T es de la forma:

$$\frac{}{\langle \mathtt{while}\ b\ \mathtt{do}\ S, s\rangle \to s}$$

es decir, necesariamente $s = s''$ y $\mathcal{B}[\![b]\!]s = \mathbf{ff}$. Usando el axioma $[\text{skip}_{\text{ns}}]$, directamente obtenemos que

$$[\text{skip}_{\text{ns}}]\ \frac{}{\langle \mathtt{skip}, s\rangle \to s''}$$

Pero entonces,

$$[\text{if}^{\text{ff}}_{\text{ns}}]\ \frac{\langle \mathtt{while}\ b\ \mathtt{do}\ S, s\rangle \to s \qquad \langle \mathtt{skip}, s\rangle \to s''}{\langle \mathtt{if}\ b\ \mathtt{then}\ (S; \mathtt{while}\ \ b\ \ \mathtt{do}\ \ S)\ \mathtt{else\ skip}, s\rangle \to s''}$$

y por tanto obtenemos el resultado.

Parte 2. Supongamos ahora que se cumple $\langle \texttt{if } b \texttt{ then } (S; \texttt{while } b \texttt{ do } S) \texttt{ else skip}, s\rangle \to s''$. Entonces, tenemos un árbol de derivación T y, de nuevo, podemos distinguir qué forma tendrá según las reglas que hayamos aplicado:

(a) Si hemos aplicado la regla $[\text{if}^{\text{tt}}_{\text{ns}}]$, T es de la forma:

$$[\text{if}^{\text{tt}}_{\text{ns}}]\ \frac{\dfrac{\dots}{\langle S; \texttt{while } b \texttt{ do } s, s\rangle \to s''}}{\langle \texttt{if } b \texttt{ then } (S; \texttt{while } b \texttt{ do } S) \texttt{ else skip}, s\rangle \to s''}$$

y con $\mathcal{B}[\![b]\!]s = \mathbf{tt}$. Ahora bien, solo hemos podido obtener la premisa anterior mediante $[\text{comp}_{\text{ns}}]$, por tener una expresión de la forma $S_1; S_2$ en la ejecución. Entonces deducimos que T es:

$$[\text{comp}_{\text{ns}}]\ \frac{\dfrac{\dots}{\langle S, s\rangle \to s'} \qquad \dfrac{\dots}{\langle \texttt{while } b \texttt{ do } S, s'\rangle \to s''}}{\langle S; \texttt{while } b \texttt{ do } s, s\rangle \to s''}$$

Pero entonces notemos que, usando la hipótesis $\mathcal{B}[\![b]\!]s = \mathbf{tt}$:

$$[\text{while}^{\text{tt}}_{\text{ns}}]\ \frac{\dfrac{\dots}{\langle S, s\rangle \to s'} \qquad \dfrac{\dots}{\langle \texttt{while } b \texttt{ do } S, s'\rangle \to s''}}{\langle \texttt{while } b \texttt{ do } S, s\rangle \to s''}$$

y obtenemos el resultado.

(b) Si hemos usado la regla $[\text{if}^{\text{ff}}_{\text{ns}}]$, deducimos que $\mathcal{B}[\![b]\!]s = \mathbf{ff}$ y que por tanto tenemos un árbol de derivación para $\langle \texttt{skip}, s\rangle \to s''$ y, por tanto, que $s = s''$. Pero usando $[\text{while}^{\text{ff}}_{\text{ns}}]$, tenemos el resultado (el razonamiento ha sido análogo al apartado (b) de la Parte 1).

□

Ejemplo 2.6. Veamos que $S_1; (S_2; S_3)$ y $(S_1; S_2); S_3$ son semánticamente equivalentes. Si suponemos que $\langle S_1; (S_2; S_3), s\rangle \to s'$, entonces es porque en su árbol de derivación hemos empleado $[\text{comp}_{\text{ns}}]$ a las premisas $\langle S_1, s\rangle \to s''$ y $\langle S_2; S_3, s''\rangle \to s'$. A su vez, la segunda premisa proviene del mismo modo de las premisas $\langle S_2, s''\rangle \to t$ y $\langle S_3, t\rangle \to s'$. Es decir, tenemos las siguientes hojas:

(a) $\langle S_1, s\rangle \to s''$.

(b) $\langle S_2, s''\rangle \to t$.

(c) $\langle S_3, t\rangle \to s'$.

Ahora, combinando (a) y (b) con $[\text{comp}_{\text{ns}}]$, obtenemos $\langle S_1; S_2, s\rangle \to t$ y, combinando esto con (c) de la misma forma, obtenemos que $\langle (S_1; S_2); S_3, s\rangle \to s'$, como queríamos ver. La otra implicación es análoga.

Notemos, por otro lado, que en general $S_1; S_2$ y $S_2; S_1$ no son semánticamente equivalentes: si tratásemos de hacer lo mismo que antes, obtendríamos las hojas $\langle S_1, s\rangle \to s''$ y $\langle S_2, s''\rangle \to s'$ por un lado y $\langle S_2, s\rangle \to s''$ y $\langle S_1, s''\rangle \to s'$ por otro, y en general no hay forma de combinar cada par de premisas para obtener la conclusión deseada.

Ejemplo 2.7. Podemos expandir el sistema While_{ns} el siguiente modo: añadimos dos reglas que permitan dar una semántica de la expresión $\texttt{for } x := a_1 \texttt{ to } a_2 \texttt{ do } S$, es decir,

$$[\text{for}^{\text{tt}}_{\text{ns}}]\quad \frac{\langle x := a_1; S, s\rangle \to s' \qquad \langle \texttt{for x} := \texttt{x} + \texttt{1 to } a_2 \texttt{ do } S, s'\rangle \to s''}{\langle \texttt{for x} := a_1 \texttt{ to } a_2 \texttt{ do } S, s\rangle \to s''} \quad \text{si } \mathcal{B}[\![a_1 \le a_2]\!]s = \mathbf{tt}$$

$$[\text{for}^{\text{ff}}_{\text{ns}}]\quad \frac{}{\langle \texttt{for x} := a_1 \texttt{ to } a_2 \texttt{ do } S, s\rangle \to s[\texttt{x} \mapsto \mathcal{A}[\![a_1]\!]]s} \quad \text{si } \mathcal{B}[\![a_1 \le a_2]\!]s = \mathbf{ff}$$

Pero debemos tener un especial cuidado con este tipo de reglas, por ejemplo, podemos descuidar que en a_1 aparezca la variable $\texttt{y}$, a saber, que a_1 contenga $\texttt{y} + 3$, y que por otro lado en S tengamos $\texttt{y} = 5$. Del mismo modo, podríamos tener que la variable $\texttt{x}$ ya aparece del mismo modo como $\texttt{x} = 4$, por ejemplo. Si $\texttt{x}$ apareciera en a_2 entonces también tendríamos este problema.

Aunque no lo demostraremos, se puede observar que el sistema While_{ns} es *Turing-completo*, es decir, en él podemos simular cualquier computación posible en una máquina de Turing. Por tanto, se podía pensar que podemos introducir reglas para ciertas expresiones en función de su correlato en While_{ns} (que existe, por lo anterior). Sin embargo, si quisiéramos introducir `for` $x := a_1$ `to` a_2 `do` S como un bucle `while ... do ...`, acabaríamos teniendo apariciones de `while ... do ...` en las reglas asociadas a `for` $x := a_1$ `to` a_2 `do` S, lo que difiere de la semántica operacional que hemos visto hasta ahora.

Ejemplo 2.8. Podríamos extender el lenguaje While con dos reglas para la expresión `repeat` S `until` b:

$$[\text{repeat}^{\text{tt}}_{\text{ns}}] \quad \frac{\langle S, s\rangle \to s'}{\langle \texttt{repeat } S \texttt{ until } b, s\rangle \to s'} \qquad \text{si } \mathcal{B}[\![b]\!]s' = \mathbf{tt}$$

$$[\text{repeat}^{\text{ff}}_{\text{ns}}] \quad \frac{\langle S, s\rangle \to s' \qquad \langle \texttt{repeat } S \texttt{ until } b, s'\rangle \to s''}{\langle \texttt{repeat } S \texttt{ until } b, s\rangle \to s''} \qquad \text{si } \mathcal{B}[\![b]\!]s' = \mathbf{ff}$$

Proposición 2.9. *Son semánticamente equivalentes:*

- *`repeat` S `until` b.*
- *S;`if` b `then skip else` (`repeat` S `until` b).*

Demostración. Parte 1. Supongamos que $\langle \texttt{repeat } S \texttt{ until } b, s\rangle \to s'$. Solo tenemos las siguientes posibilidades:

(a) Si hemos aplicado la regla $[\text{repeat}^{\text{tt}}_{\text{ns}}]$, tenemos:

$$[\text{repeat}^{\text{tt}}_{\text{ns}}] \quad \frac{\langle S, s\rangle \to s'}{\langle \texttt{repeat } S \texttt{ until } b, s\rangle \to s'} \qquad \text{si } \mathcal{B}[\![b]\!]s' = \mathbf{tt}$$

Ahora bien, por otro lado, podemos aplicar el axioma $[\text{skip}_{\text{ns}}]$ para obtener directamente que $\langle \texttt{skip}, s'\rangle \to s'$. Ahora, como si $\mathcal{B}[\![b]\!]s' = \mathbf{tt}$, podemos aplicar la regla $[\text{if}^{\text{tt}}_{\text{ns}}]$:

$$[\text{if}^{\text{tt}}_{\text{ns}}] \ \frac{\langle \texttt{skip}, s'\rangle \to s'}{\langle \texttt{if } b \texttt{ then skip else (repeat } S \texttt{ until } b), s'\rangle \to s'}$$

Y, entonces,

$$[\text{comp}_{\text{ns}}] \ \frac{\langle S, s\rangle \to s' \qquad \langle \texttt{if } b \texttt{ then skip else (repeat } S \texttt{ until } b), s'\rangle \to s'}{\langle S; \texttt{if } b \texttt{ then skip else (repeat } S \texttt{ until } b), s\rangle \to s'}$$

Luego obtenemos el resultado.

(b) Si hemos aplicado la regla $[\text{repeat}^{\text{ff}}_{\text{ns}}]$, tenemos:

$$[\text{repeat}^{\text{ff}}_{\text{ns}}] \quad \frac{\langle S, s\rangle \to s'' \qquad \langle \texttt{repeat } S \texttt{ until } b, s''\rangle \to s'}{\langle \texttt{repeat } S \texttt{ until } b, s\rangle \to s'} \qquad \text{si } \mathcal{B}[\![b]\!]s' = \mathbf{ff}$$

Ahora, usando que si $\mathcal{B}[\![b]\!]s' = \mathbf{ff}$,

$$[\text{if}^{\text{ff}}_{\text{ns}}] \ \frac{\langle \texttt{repeat } S \texttt{ until } b, s''\rangle \to s'}{\langle \texttt{if } b \texttt{ then skip else (repeat } S \texttt{ until } b), s''\rangle \to s'}$$

Pero entonces,

$$[\text{comp}_{\text{ns}}] \ \frac{\langle S, s\rangle \to s'' \qquad \langle \texttt{if } b \texttt{ then skip else (repeat } S \texttt{ until } b), s''\rangle \to s'}{\langle S; \texttt{if } b \texttt{ then skip else (repeat } S \texttt{ until } b), s\rangle \to s'}$$

Parte 2. Supongamos que $\langle S;\ \texttt{if } b \texttt{ then skip else repeat } S \texttt{ until } b, s\rangle \to s'$. La única posibilidad es haber aplicado la regla $[\text{comp}_{\text{ns}}]$

$$\frac{\langle S, s\rangle \to s_0 \qquad \langle \texttt{if } b \texttt{ then skip else (repeat } S \texttt{ until } b), s_0\rangle \to s'}{\langle S; \texttt{if } b \texttt{ then skip else repeat } S \texttt{ until } b, s\rangle \to s'}$$

para algún $s_0 \in \mathbf{State}$. Para la transición $\langle \texttt{if } b \texttt{ then skip else } (\texttt{repeat } S \texttt{ until } b), s_0\rangle \to s'$ tenemos dos posibilidades

(a) Si $\mathcal{B}[\![b]\!]s_0 = \mathbf{tt}$ entonces únicamente existe la posibilidad de que se haya derivado de $[\text{if}^{\text{tt}}_{\text{ns}}]$:

$$\frac{\langle skip, s_0\rangle \to s'}{\langle \texttt{if } b \texttt{ then skip else } (\texttt{repeat } S \texttt{ until } b), s_0\rangle \to s'}$$

y la única forma de que sea cierto $\langle skip, s_0\rangle \to s'$ es que $s_0 = s'$. Como se verifica $\langle S, s\rangle \to s_0$ entonces se verifica $\langle S, s\rangle \to s'$ y se puede aplicar la regla $[\text{repeat}^{\text{tt}}_{\text{ns}}]$:

$$\frac{\langle S, s\rangle \to s'}{\langle \texttt{repeat } S \texttt{ until } b, s\rangle \to s'}$$

obteniendo el resultado pues ya se sabe que $\langle S, s\rangle \to s_0$.

(b) Si $\mathcal{B}[\![b]\!]s_0 = \mathbf{ff}$ entonces solo cabe la posibilidad de que haya partido de $[\text{if}^{\text{ff}}_{\text{ns}}]$:

$$\frac{\langle \texttt{repeat } S \texttt{ until } b, s_0\rangle \to s'}{\langle \texttt{if } b \texttt{ then skip else } (\texttt{repeat } S \texttt{ until } b), s_0\rangle \to s'}$$

teniendo así $\langle \texttt{repeat } S \texttt{ until } b, s_0\rangle \to s'$ y entonces se puede deducir

$$\frac{\langle S, s\rangle \to s_0 \qquad \langle \texttt{repeat } S \texttt{ until } b, s_0\rangle \to s'}{\langle \texttt{repeat } S \texttt{ until } b, s\rangle \to s'}$$

mediante $[\text{repeat}^{\text{ff}}_{\text{ns}}]$ pues $\mathcal{B}[\![b]\!]s_0 = \mathbf{ff}$.

□

Para poder demostrar que una propiedad como la anterior se verifica en árboles sencillos y compuestos, emplearemos la demostración por *inducción sobre reglas*, que se compone de dos pasos:

1. Primero comprobamos que la propiedad se verifica para los axiomas del sistema.
2. Para cada regla, suponiendo que las premisas verifican la propiedad , comprobamos que también se cumple para la conclusión (siempre y cuando se verifiquen las condiciones de la regla).

El siguiente resultado nos dice que, en general, hay *una* manera de deducir una configuración mediante las reglas del sistema de transición While_{ns}:

Teorema 2.10. *El sistema de transiciones* While_{ns} *es determinista, es decir, para cada* $S \in \mathbf{Stm}$, $s, s', s'' \in \mathbf{State}$,

$$\langle S, s\rangle \to s' \text{ y } \langle S, s\rangle \to s'' \text{ implica que } s' = s''.$$

Demostración. Para simplificar la demostración, vamos a definir una propiedad sintáctica de las reglas del sistema While_{ns}. Decimos que dos reglas son *independientes entre sí* cuando no es posible obtener una mediante la aplicación de la otra. Notemos que este es el caso de nuestro sistema: las reglas $[\text{while}^{\text{tt}}_{\text{ns}}]$ y $[\text{while}^{\text{ff}}_{\text{ns}}]$ son independientes entre sí porque ambas tienen premisas distintas (suponemos que $\mathbf{tt}$ y $\mathbf{ff}$ son distintos). Entonces, como cada regla es independiente de la otra (y evidentemente, cada regla es determinista), deducimos que, en caso de que tengamos $\langle S, s\rangle \to s'$ y $\langle S, s\rangle \to s''$, necesariamente tendremos que haber aplicado la misma única regla posible en los dos casos para llegar a las respectivas configuraciones. Es fácil convencerse entonces de que, por inducción sobre las reglas, la propiedad deseada se cumple[2]. □

Ejemplo 2.11. Podemos añadir una semántica $\texttt{forVar } x \texttt{ do } S$ que ejecute la sentencia S siempre que x sea distinto de 0 y lo incremente en 1 en cada iteración. Veamos que sería semánticamente equivalente a $\texttt{while } \neg(x = 0) \texttt{ do } (S; x := x + 1)$.

Primero, definimos la semantica de $\texttt{forVar } x \texttt{ do } S$:

[2]Obviamente, esa demostración no es intercambiable con la demostración formal por inducción estructural. Podría demostrarse el caso general del que hemos hablado, a saber, formalizando lo que significa precisamente la independencia de dos reglas.

$$[\text{for}^0]\ \frac{}{\langle \texttt{forVar}\ x\ \texttt{do}\ S, s\rangle \to s}\ \text{si } \mathcal{A}[\![x]\!]s = 0$$

$$[\text{for}^{\neq 0}]\ \frac{\langle S; x := x+1, s\rangle \to s' \qquad \langle \texttt{forVar}\ x\ \texttt{do}\ S, s'\rangle \to s_1}{\langle \texttt{forVar}\ x\ \texttt{do}\ S, s\rangle \to s_1}\ \text{si } \mathcal{A}[\![x]\!]s \neq 0$$

Veamos que $\langle \texttt{forVar}\ x\ \texttt{do}\ S, s\rangle \to s_1$ implica $\langle \texttt{while}\ \neg(x = 0)\ \texttt{do}\ (S; x := x + 1), s\rangle \to s_1$.
Para empezar, sabemos que $\mathcal{A}[\![x]\!]s = 0$ si y solo si $\mathcal{B}[\![\neg(x = 0)]\!]s = \mathbf{ff}$, dividimos la demostración en dos pasos:

1. Si $x = 0$ tenemos por $[\text{for}^0]$ que:

   $$\langle \texttt{forVar}\ x\ \texttt{do}\ S, s\rangle \to s$$

   Como $\mathcal{B}[\![\neg(x = 0)]\!]s = \mathbf{ff}$ por la regla $[\text{while}^{\text{ff}}_{\text{ns}}]$ sabemos que:

   $$\langle \texttt{while}\ \neg(x = 0)\ \texttt{do}\ (S; x := x + 1), s\rangle \to s$$

2. Si $x \neq 0$, entonces suponemos ciertas las siguientes premisas:

   a) $\langle S; x := x + 1, s\rangle \to s_2$
   b) $\langle \texttt{forVar}\ x\ \texttt{do}\ S, s_2\rangle \to s'$

   pues la transición $\langle \texttt{forVar}\ x\ \texttt{do}\ S, s\rangle \to s'$ solo puede haber provenido de:

   $$[\text{for}^{\neq 0}]\ \frac{\langle S; x := x+1, s\rangle \to s_2 \qquad \langle \texttt{forVar}\ x\ \texttt{do}\ S, s_2\rangle \to s'}{\langle \texttt{forVar}\ x\ \texttt{do}\ S, s\rangle \to s'}$$

   Podemos aplicar la hipótesis de inducción sobre $\langle \texttt{forVar}\ x\ \texttt{do}\ S, s_2\rangle \to s'$ y por lo tanto tenemos que $\langle \texttt{forVar}\ x\ \texttt{do}\ S, s_2\rangle \to s'$ implica que $\langle \texttt{while}\ \neg(x = 0)\ \texttt{do}\ (S; x := x + 1), s_2\rangle \to s'$, luego podemos costruir el siguiente árbol de derivación:

   $$[\text{while}^{\text{tt}}_{\text{ns}}]\ \frac{\langle S; x := x+1, s\rangle \to s_2 \qquad \langle \texttt{while}\ \neg(x = 0)\ \texttt{do}\ (S; x := x + 1), s_2\rangle \to s'}{\langle \texttt{while}\ \neg(x = 0)\ \texttt{do}\ (S; x := x + 1), s\rangle \to s'}$$

Supongamos ahora que $\langle \texttt{while}\ \neg(x = 0)\ \texttt{do}\ (S; x := x + 1), s\rangle \to s'$. Entonces, distinguimos los siguientes casos:

1. Si hemos aplicado $[\text{while}^{\text{ff}}_{\text{ns}}]$, entonces

   $$[\text{while}^{\text{ff}}_{\text{ns}}]\ \frac{}{\langle \texttt{while}\ \neg(x = 0)\ \texttt{do}\ (S; x := x + 1), s\rangle \to s'}$$

   y además deducimos que $s = s'$ y que $x \neq 0$. Pero entonces tenemos que, directamente:

   $$[\text{for}^0]\ \frac{}{\langle \texttt{forVar}\ x\ \texttt{do}\ S, s\rangle \to s}$$

   es decir, obtenemos la implicación deseada.

2. Si hemos aplicado $[\text{while}^{\text{tt}}_{\text{ns}}]$,

   $$[\text{while}^{\text{tt}}_{\text{ns}}]\ \frac{\langle (S; x := x+1), s\rangle \to s' \qquad \langle \texttt{while}\ \neg(x = 0)\ \texttt{do}\ (S; x := x + 1), s'\rangle \to s''}{\langle \texttt{while}\ \neg(x = 0)\ \texttt{do}\ (S; x := x + 1), s\rangle \to s''}$$

   y además deducimos que $x \neq 0$. Si aplicamos hipótesis de inducción sobre $\langle \texttt{while}\ \neg(x = 0)\ \texttt{do}\ (S; x := x + 1), s'\rangle \to s''$, obtenemos que $\langle \texttt{forVar}\ x\ \texttt{do}\ S, s'\rangle \to s''$. Pero entonces, juntando las premisas anteriores,

   $$[\text{for}^{\neq 0}]\ \frac{\langle S; x := x+1, s\rangle \to s' \qquad \langle \texttt{forVar}\ x\ \texttt{do}\ S, s'\rangle \to s''}{\langle \texttt{forVar}\ x\ \texttt{do}\ S, s\rangle \to s''}$$

   luego obtenemos el resultado.

2.1.3. Expresiones

Finalmente, podemos definir el valor semántico de cada $S \in \mathbf{Stm}$ mediante una aplicación $\mathcal{S}_{\text{ns}} : \mathbf{Stm} \to (\mathbf{State} \hookrightarrow \mathbf{State})$, donde

$$\mathcal{S}_{\text{ns}}[[S]] : \mathbf{State} \hookrightarrow \mathbf{State}$$
$$s \mapsto \begin{cases} s', \text{ si } \langle S, s\rangle \to s' \\ \text{indefinido, en otro caso} \end{cases}$$

El determinismo de While_{ns} implica que está bien definida. Además, es parcial porque, como vimos, la expresión `while true do skip` siempre entra en bucle, es decir, $\mathcal{S}_{\text{ns}}[[\texttt{while true do skip}]]s = \text{indefinido}$, para cada $s \in \mathbf{State}$.

Ejemplo 2.12. Podemos definir, por ejemplo, una semántica de paso largo para **Aexp** mediante la relación de transición $\langle a, s\rangle \to_A z$, donde $\langle a, s\rangle$ significa que $a \in \mathbf{Aexp}$ se evalúa en $s \in \mathbf{State}$ y $z \in \mathbb{Z}$ es un estado final:

Sistema ($\mathbf{Aexp}_{\text{ns}}$)**.**

$$\frac{}{\langle n, s\rangle \to_A \mathcal{N}[\![n]\!]} \text{ si } n \in \mathbf{Num}$$

$$\frac{}{\langle x, s\rangle \to_A sx} \text{ si } x \in \mathbf{Var}$$

$$\frac{\langle a_1, s\rangle \to_A z_1 \qquad \langle a_2, s\rangle \to_A z_2}{\langle a_1 \ \texttt{op} \ a_2, s\rangle \to_A z_1 * z_2}$$

donde `op` se refiere a $\oplus, \ominus, \otimes$ y $*$ a $+, -, \times$.

Proposición 2.13. *Sean $a \in \mathbf{Aexp}$, $s \in \mathbf{State}$, $z \in \mathbb{Z}$. Entonces $\langle a, s\rangle \to_A z$ si y solo si $\mathcal{A}[\![a]\!]s = z$.*

Demostración. La demostración es por inducción estructural. Los casos base son:

- Si $a = n$, entonces $\langle a, s\rangle \to_A \mathcal{N}[\![n]\!] = \mathcal{A}[\![n]\!]s$.
- Si $a = x$, entonces $\langle a, s\rangle \to_A sx = \mathcal{A}[\![x]\!]s$.

Resumimos los casos inductivos en:

- Si $a = a_1 \ \texttt{op} \ a_2$, entonces, empleando la última regla de $\mathbf{Aexp}_{\text{ns}}$, tenemos que $\langle a_i, s\rangle \to_A z_i$ si y solo si (por hipótesis de inducción) $\mathcal{A}[\![a_i]\!]s = z_i$, con $i = 1, 2$. Pero entonces sabemos que $\mathcal{A}[\![a_1 \ \texttt{op} \ a_2]\!]S = \mathcal{A}[\![a_1]\!]s * \mathcal{A}[\![a_2]\!]s = z_1 * z_2 = z$.

□

Ejemplo 2.14. Siguiendo el ejemplo anterior, podemos definir un sistema de transiciones para expresiones booleanas como sigue. Definimos $\langle b, s\rangle \to_B X$, donde $\langle b, s\rangle$ indica que b se evalúa en el estado s y donde $X \in \mathit{Bool}$.

Sistema ($\mathbf{Bexp}_{\text{ns}}$)**.**

$$\frac{}{\langle \texttt{true}, s\rangle \to_B \mathbf{tt}}$$

$$\frac{}{\langle \texttt{false}, s\rangle \to_B \mathbf{ff}}$$

$$\frac{\langle a_1, s\rangle \to_A z_1 \qquad \langle a_2, s\rangle \to_A z_2}{\langle a_1 = a_2, s\rangle \to_A X}$$

donde X es el booleano correspondiente (véase la definición de $\mathcal{B}[\![\cdot]\!]$).

$$\frac{\langle a_1, s\rangle \to_A z_1 \qquad \langle a_2, s\rangle \to_A z_2}{\langle a_1 \leq sa_2, s\rangle \to_A X}$$

donde X es el booleano correspondiente (de nuevo, empleando la definición de $\mathcal{B}[\![\cdot]\!]$).

$$\frac{\langle a_1, s\rangle \to_A z_1 \qquad \langle a_2, s\rangle \to_A z_2}{\langle a_1 \leq sa_2, s\rangle \to_A X}$$

donde X es el booleano correspondiente.

$$\frac{\langle b, s\rangle \to_B X}{\langle \neg b, s\rangle \to_B X'}$$

donde X' es el booleano correspondiente (negación de X).

$$\frac{\langle b_1, s\rangle \to_B X \qquad \langle b_2, s\rangle \to_B Y}{\langle b_1 \wedge b_2, s\rangle \to_B Z}$$

donde Z es el booleano correspondiente (conjunción de X e Y).

El siguiente resultado es análogo al último que dimos antes:

Proposición 2.15. *Sean $b \in$ **Bexp**, $s \in$ **State** y $X \in Bool$. Entonces $\langle b, s\rangle \to_B X$ si y solo si $\mathcal{B}[\![b]\!]s = X$.*

Demostración. Por inducción estructural. Véase la demostración del anterior teorema y la definición de $\mathcal{B}[\![\cdot]\!]$. La demostración es análoga. □

2.2. Semántica operacional estructural

2.2.1. Sistema de transiciones

Ahora nos centramos en los pasos concretos de la ejecución de un programa. Para ello, definimos una relación de transición $\langle S, s\rangle \Rightarrow \gamma$ como:

- Si γ es de la forma $\langle S', s'\rangle$, entonces la ejecución de S desde s no se completa y sigue en $\langle S', s'\rangle$.
- Si γ es de la forma s', entonces la ejecución finaliza en el estado s'.

La nueva relación de transición queda determinada por el conjunto de reglas:

Sistema ($\text{While}_{\text{sos}}$)**.**

[ass_{sos}]

$$\frac{}{\langle x := a, s\rangle \Rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s]}$$

[skip_{sos}]

$$\frac{}{\langle \texttt{skip}, s\rangle \Rightarrow s}$$

[$\text{comp}^1_{\text{sos}}$]

$$\frac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_1'; S_2, s'\rangle}$$

[$\text{comp}^2_{\text{sos}}$]

$$\frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}$$

[$\text{if}^{\text{tt}}_{\text{sos}}$]

$$\frac{}{\langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, s\rangle \Rightarrow \langle S_1, s\rangle} \text{ si } \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

[$\text{if}^{\text{ff}}_{\text{sos}}$]

$$\frac{}{\langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2; s, s\rangle \Rightarrow \langle S_2, s\rangle} \text{ si } \mathcal{B}[\![b]\!]s = \mathbf{ff}$$

[$\text{while}_{\text{sos}}$]

$$\frac{}{\langle \texttt{while } b \texttt{ do } S, s\rangle \Rightarrow \langle \texttt{if } b \texttt{ then } (S; \texttt{while } b \texttt{ do } S) \texttt{ else skip}, s\rangle}$$

Notemos que podríamos haber incluido, por ejemplo, dos reglas para la semántica de `while` b `do` S:

$$[\text{while}^{\text{ff}}_{\text{sos}}] \ \frac{}{\langle \texttt{while } b \texttt{ do } S, s\rangle \Rightarrow s} \text{ si } \mathcal{B}[\![s]\!] = \mathbf{ff}$$

y

$$[\text{while}^{\text{tt}}_{\text{sos}}] \ \frac{}{\langle \texttt{while } b \texttt{ do } S, s\rangle \Rightarrow \langle S; \texttt{while } b \texttt{ do } S, s\rangle} \text{ si } \mathcal{B}[\![s]\!] = \mathbf{tt}$$

Definición 2.16. Se dirá que $\langle S, s\rangle$ está *bloqueada* si no existe γ tal que $\langle S, s\rangle \Rightarrow \gamma$. Una secuencia de derivación es finita cuando llega a un bloqueo o a un estado final:

$$\gamma_0 \Rightarrow \gamma_1 \Rightarrow ... \Rightarrow \gamma_k$$

donde $\gamma_0 = \langle S, s\rangle$, $\gamma_i \Rightarrow \gamma_{i+1}$ para $i \in \{0, ..., k-1\}$ y γ_k es una configuración bloqueada.

Normalmente escribiremos $\gamma_0 \Rightarrow^i \gamma$ si hay i pasos en la ejecución de γ_0 a γ. Si hay finitos pasos, denotamos $\gamma_0 \Rightarrow^* \gamma$. $\gamma_0 \Rightarrow^i \gamma$ y $\gamma_0 \Rightarrow^* \gamma$ no tiene por qué ser secuencias de derivación, solo si γ es configuración final o de bloqueo.

Definición 2.17. La ejecución $\langle S, s\rangle$ de la expresión S en un estado s:

1. *Termina* si existe una única secuencia de derivación finita comenzando en $\langle S, s\rangle$.
2. *Termina con éxito* si $\langle S_1, s\rangle \Rightarrow^* s'$ para algún estado s'.
3. *Cicla* si existe una secuencia de derivación infinita comenzando en $\langle S, s\rangle$.

Nótese que estas definiciones son mutuamente excluyentes si y solo si las secuencias de derivacion son únicas. Por comodidad, las definimos de este modo porque, si extendemos el lenguaje, no nos tendremos que preocupar.

Ejemplo 2.18. Supongamos que queremos extender $\text{While}_{\text{sos}}$ con la expresión `repeat` S `until` b. Podemos añadir la regla:

$$[\text{repeat}_{\text{sos}}] \ \frac{}{\langle \texttt{repeat } S \texttt{ until } b, s\rangle \Rightarrow \langle S; \texttt{if } b \texttt{ then skip else } (\texttt{repeat } S \texttt{ until } b), s\rangle}$$

La idea es que la expresión `repeat` S `until` b sea equivalente a S; `while` $\neg b$ `do` S. Se definirá posteriormente el concepto de equivalencia semántica y se demostrará este resultado.

2.2.2. Propiedades

El método de demostración principal consiste en hacer *inducción sobre la longitud de las secuencias de derivación* (finitas) que se estudian, es decir, si queremos demostrar una propiedad acerca de nuestro sistema de transiciones:

- Demostramos que la propiedad se cumple para secuencias de derivación de longitud 0 (en ocasiones nos encontraremos que se cumple la propiedad de forma vacía).
- Demostramos que si la propiedad se cumple para secuencias de longitud (a lo sumo) k, entonces se cumple para secuencias de longitud $k+1$.

A modo de ejemplo de este método, veamos el siguiente resultado:

Lema 2.19. *Si* $\langle S_1; S_2, s\rangle \Rightarrow^k s''$*, entonces existen* $s' \in \mathbf{State}$*,* $k_1, k_2 \in \mathbb{N}$ *tales que* $k = k_1 + k_2$ *y*

$$\langle S_1, s\rangle \Rightarrow^{k_1} s' \quad y \quad \langle S_2, s'\rangle \Rightarrow^{k_2} s''.$$

Demostración. Si $k = 0$, entonces $\langle S_1; S_2, s\rangle \Rightarrow^0 s''$ implica (vacuamente) el resultado, porque $\langle S_1; S_2, s\rangle$ y s'' son distintos. Supongamos que el resultado se cumple para longitudes menores o iguales que k. Veamos que se sigue para $k+1$. Por tanto, tenemos la premisa $\langle S_1; S_2, s\rangle \Rightarrow^{k+1} s''$, es decir, que existe una configuración γ tal que

$$\langle S_1; S_2, s\rangle \Rightarrow \gamma \Rightarrow^k s''$$

Por tanto, distinguimos dos casos según la regla que hemos aplicado a $\langle S_1; S_2, s\rangle$ para llegar a γ:

(a) Si hemos aplicado $[\mathrm{comp}^1_{\mathrm{sos}}]$, tenemos que

$$[\mathrm{comp}^1_{\mathrm{sos}}]\ \frac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_1'; S_2, s'\rangle = \gamma}$$

luego $\langle S_1'; S_2, s'\rangle \Rightarrow^k s''$. Entonces, como esta derivación es de longitud k, podemos aplicar hipótesis de inducción, esto es, existen $s_0 \in \mathbf{State}$ y $k_1, k_2 \in \mathbb{N}$ con $k = k_1 + k_2$ y

$$\langle S_1', s'\rangle \Rightarrow^{k_1} s_0 \quad \text{y} \quad \langle S_2, s_0\rangle \Rightarrow^{k_2} s''.$$

Ahora bien, como tenemos la premisa $\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle$ y $\langle S_1', s'\rangle \Rightarrow^{k_1} s_0$, entonces tenemos que $\langle S_1, s\rangle \Rightarrow^{k_1+1} s_0$. Por otro lado, también tenemos $\langle S_2, s_0\rangle \Rightarrow^{k_2} s''$ y que $(k_1+1)+k_2 = (k_1+k_2)+1 = k+1$. Es decir, hemos obtenido la conclusión deseada. Por tanto, hemos probado el resultado para este caso.

(b) Si hemos aplicado $[\mathrm{comp}^2_{\mathrm{sos}}]$, tenemos que

$$[\mathrm{comp}^2_{\mathrm{sos}}]\ \frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_2, s'\rangle = \gamma}$$

Entonces deducimos que $\langle S_2, s'\rangle \Rightarrow^k s''$. Simplemente tomando $k_1 := 1$ y $k_2 := k$ vemos que $k_1 + k_2 = k+1$ y que tenemos el resultado.

□

Ejemplo 2.20. Por otro lado, $\langle S_1; S_2, s\rangle \Rightarrow^* \langle S_2, s'\rangle$ no implica necesariamente que $\langle S_1, s\rangle \Rightarrow^* s'$. Por ejemplo, podemos tomar $S_1 := \texttt{skip}$, $S_2 := \texttt{while}\ \neg(\texttt{x} = 1)\ \texttt{do}\ \texttt{x} := \texttt{x} + 1$ y $s\texttt{x} = 3$, $s'\texttt{x} = s[\texttt{x} \mapsto 2]$.

El siguiente lema viene a decir que la ejecución de una expresión es independiente de cualquier enunciado que se ejecute después:

Lema 2.21. *Si* $\langle S_1, s\rangle \Rightarrow^k s'$*, entonces* $\langle S_1; S_2, s\rangle \Rightarrow^k \langle S_2, s'\rangle$.

Demostración. Por inducción sobre la longitud de las derivaciones. En caso de $k = 0$, la premisa es falsa y el resultado se tiene directamente. Supongamos que se cumple el resultado para longitudes $\leq k$ y veámoslo para $k+1$. Nuestra suposición es que $\langle S_1, s\rangle \Rightarrow^{k+1} s'$. Entonces tenemos que hay cierta configuración γ con

$$\langle S_1, s\rangle \Rightarrow \gamma \Rightarrow^{k} s'$$

y además, notemos que $\gamma = \langle S, s''\rangle$ porque $k \leq 1$. Pero entonces, aplicando la hipótesis de inducción a $\langle S, s''\rangle \Rightarrow^{k} s'$, tenemos que $\langle S; S_2, s''\rangle \Rightarrow^{k} \langle S_2, s'\rangle$.

Por otro lado, de $\langle S_1, s\rangle \Rightarrow \langle S, s''\rangle$ podemos deducir que:

$$[\mathrm{comp}^1_{\mathrm{sos}}]\ \frac{\langle S_1, s\rangle \Rightarrow \langle S, s''\rangle}{\langle S_1; S_2, s\rangle \Rightarrow \langle S; S_2, s''\rangle}$$

Es decir, sabemos que $\langle S_1; S_2, s\rangle \Rightarrow \langle S; S_2, s''\rangle$ y que $\langle S; S_2, s''\rangle \Rightarrow^{k} \langle S_2, s'\rangle$. Basta componer ambas derivaciones para ver que $\langle S_1; S_2, s\rangle \Rightarrow^{k+1} \langle S_2, s'\rangle$, como queríamos. □

Teorema 2.22. *El sistema de transiciones* $\mathrm{While}_{\mathrm{sos}}$ *es determinista, es decir, para cualesquiera* S, s, γ, γ' *tenemos que*

$$\langle S, s\rangle \Rightarrow \gamma \ y\ \langle S, s\rangle \Rightarrow \gamma' \ \textit{implica que}\ \gamma = \gamma'$$

Demostración. Véase la demostración del Teorema 2.10. □

Definición 2.23. Dos expresiones S_1, S_2 se dicen *semánticamente equivalentes* si, para cada $s \in$ **State**,

- Si γ es estado final o bloqueado, entonces $\langle S_1, s\rangle \Rightarrow^* \gamma$ si y solo si $\langle S_2, s\rangle \Rightarrow^* \gamma$. Nótese que las longitudes de las derivaciones no tienen por qué coincidir.
- La[3] secuencia de derivación empezando en $\langle S_1, s\rangle$ es infinita si y solo si lo es la que empieza en $\langle S_2, s\rangle$.

Ejemplo 2.24. Veamos que S y $S;\texttt{skip}$ son semánticamente equivalentes. Supongamos que $\langle S, s\rangle \Rightarrow^* s'$, es decir, existe $k \in \mathbb{N}$ tal que $\langle S, s\rangle \Rightarrow^* s'$. Entonces, el Lema 2.21 nos dice que dado $\langle S, s\rangle \Rightarrow^{k} s'$ se tiene

$$\langle S; \texttt{skip}, s\rangle \Rightarrow^{k} \langle \texttt{skip}, s'\rangle$$

Por la regla $[\mathrm{skip}_{\mathrm{sos}}]$ se deduce $\langle \texttt{skip}, s'\rangle \Rightarrow s'$ y entonces

$$\langle S; \texttt{skip}, s\rangle \Rightarrow^* s'$$

Supongamos ahora que $\langle S; \texttt{skip}, s\rangle \Rightarrow^* s''$, entonces existe $k \in \mathbb{N}$ tal que $\langle S; \texttt{skip}, s\rangle \Rightarrow^{k} s''$. Por el Lema 2.19 se tiene la existencia de $s' \in$ **State** y $k_1, k_2 \in \mathbb{N}$ tal que $k = k_1 + k_2$ y

$$\langle S, s\rangle \Rightarrow^{k_1} s' \quad y \quad \langle \texttt{skip}, s'\rangle \Rightarrow^{k_2} s''$$

La secuencia $\langle \texttt{skip}, s'\rangle \Rightarrow^{k_2} s''$ es cierta si y solo si $k_2 = 1$ y $s' = s''$ pues solo se puede aplicar $[\mathrm{skip}_{\mathrm{sos}}]$, se deduce entonces

$$\langle S, s\rangle \Rightarrow^{k_1} s''$$

es decir, $\langle S, s\rangle \Rightarrow^* s''$.

Ejemplo 2.25. Definimos en 2.18 la sentencia $\texttt{repeat}\ S\ \texttt{until}\ b$ y dejamos por hacer la demostración de su equivalencia con $S;\ \texttt{while}\ \neg b\ \texttt{do}\ S$. Hay que probar que para cada $s, s' \in$ **State** sucede que

$$\langle \texttt{repeat}\ S\ \texttt{until}\ b, s\rangle \Rightarrow^* s' \Longleftrightarrow \langle S;\ \texttt{while}\ \neg b\ \texttt{do}\ S, s\rangle \Rightarrow^* s'$$

$\Longrightarrow$) Supongamos que

$$\langle \texttt{repeat}\ S\ \texttt{until}\ b, s\rangle \Rightarrow^* s'$$

Sabemos por la regla $[\mathrm{repeat}_{\mathrm{sos}}]$

$$\langle \texttt{repeat}\ S\ \texttt{until}\ b, s\rangle \Rightarrow^* s' \implies \langle S;\ \texttt{if}\ b\ \texttt{then skip else}\ (\texttt{repeat}\ S\ \texttt{until}\ b), s\rangle \Rightarrow^* s'$$

[3] La unicidad viene dada por el determinismo de $\mathrm{While}_{\mathrm{sos}}$.

y por el lema 2.19 existen $k_1, k_2 \in \mathbb{N}$ tal que

$$\langle S, s\rangle \Rightarrow^{k_1} s''$$

y

$$\langle \texttt{if } b \texttt{ then skip else (repeat } S \texttt{ until } b), s''\rangle \Rightarrow^{k_2} s'$$

Dividimos en casos en función del valor booleano de b:

- $\mathcal{B}[\![b]\!]s'' = \mathbf{tt}$. Usando el axioma $[\text{if}^{\text{tt}}_{sos}]$ seguido de $[\text{skip}_{\text{sos}}]$ se tiene

$$\langle \texttt{if } b \texttt{ then skip else (repeat } S \texttt{ until } b), s''\rangle \overset{[\text{if}^{\text{tt}}_{sos}]}{\Rightarrow} \langle \texttt{skip}, s''\rangle \overset{[\text{skip}_{\text{sos}}]}{\Rightarrow} s''$$

  de lo que se deduce, por determinismo, que $s' = s''$. Desarrollemos ahora $\langle S;\ \texttt{while } \neg b \texttt{ do } S, s\rangle$. Por el lema 2.21, como sabemos que $\langle S, s\rangle \Rightarrow^{k_1} s''$, deducimos que

$$\langle S;\ \texttt{while } \neg b \texttt{ do } S, s\rangle \Rightarrow^{k_1} \langle \texttt{while } \neg b \texttt{ do } S, s''\rangle$$

  Aplicando $[\text{while}_{sos}]$ deducimos $\langle \texttt{if } \neg b \texttt{ then } S;\ \texttt{while } \neg b \texttt{ do } S \texttt{ else skip}, s''\rangle$ y como $\mathcal{B}[\![\neg b]\!] = \mathbf{ff}$ entonces

$$\langle \texttt{if } \neg b \texttt{ then } S;\ \texttt{while } \neg b \texttt{ do } S \texttt{ else skip}, s''\rangle \Rightarrow \overset{[\text{if}^{\text{ff}}_{sos}]}{\Longrightarrow} \langle \texttt{skip}, s''\rangle \overset{[\text{skip}_{\text{sos}}]}{\Longrightarrow} s''$$

  concluyendo que $\langle S;\ \texttt{while } \neg b \texttt{ do } S, s\rangle \Rightarrow^* s'$, como queríamos probar.

- $\mathcal{B}[\![b]\!]s'' = \mathbf{ff}$. Aplicando $[\text{if}_{\text{ff}}]$ se tiene

$$\langle \texttt{if } b \texttt{ then skip else (repeat } S \texttt{ until } b), s''\rangle \overset{[\text{if}^{\text{tt}}_{sos}]}{\Rightarrow} \langle \texttt{repeat } S \texttt{ until } b, s''\rangle$$

  y esta última sentencia $\Rightarrow^* s'$, por determinismo. Por hipótesis de inducción se deduce de

$$\langle \texttt{repeat } S \texttt{ until } b, s''\rangle \Rightarrow^* s'$$

  que

$$\langle S;\ \texttt{while } \neg b \texttt{ do } S, s''\rangle \Rightarrow^* s'$$

  y por el lema 2.21 y aplicando $[\text{while}_{\text{sos}}]$

  $\langle S;\ \texttt{while } \neg b \texttt{ do } S, s\rangle \Rightarrow^* \langle\ \texttt{while } \neg b \texttt{ do } S, s''\rangle \Rightarrow \langle \texttt{if } \neg b \texttt{ then } S;\ \texttt{while } \neg b \texttt{ do } S \texttt{ else skip}, s''\rangle$

  Dado que $\mathcal{B}[\![\neg b]\!]s'' = \mathbf{tt}$ se tiene

$$\langle \texttt{if } \neg b \texttt{ then } S;\ \texttt{while } \neg b \texttt{ do } S \texttt{ else skip}, s''\rangle \Rightarrow \langle S;\ \texttt{while } \neg b \texttt{ do } S, s''\rangle \Rightarrow^* s'$$

  cocluyendo el resultado $\langle S;\ \texttt{while } \neg b \texttt{ do } S, s\rangle \Rightarrow^* s'$.

$\Longleftarrow$) Suponemos $\langle S;\ \texttt{while } \neg b \texttt{ do } S, s\rangle \Rightarrow^* s'$. Nuevamente por el lema 2.19 se tienen $k_1, k_2 \in \mathbb{N}$ y $s'' \in$ **State** tal que

$$\langle S, s\rangle \Rightarrow^{k_1} s'' \quad y \quad \langle \texttt{while } \neg b \texttt{ do } S, s''\rangle \Rightarrow^{k_2} s'$$

De $[\text{while}_{\text{sos}}]$ se tiene

$$\langle \texttt{while } \neg b \texttt{ do } S, s''\rangle \Rightarrow \langle \texttt{if } \neg b \texttt{ then } S;\ \texttt{while } \neg b \texttt{ do } S \texttt{ else skip}, s''\rangle$$

Se divide en casos según $\mathcal{B}[\![b]\!]s''$:

- Si $\mathcal{B}[\![b]\!]s'' = \mathbf{tt}$ ($\mathcal{B}[\![\neg b]\!]s'' = \mathbf{ff}$) entonces se aplica $[\text{if}^{\text{ff}}_{\text{sos}}]$ y $[\text{skip}_{\text{sos}}]$ para deducir

$$\langle \texttt{if } \neg b \texttt{ then } S;\ \texttt{while } \neg b \texttt{ do } S \texttt{ else skip}, s'' \rangle \Rightarrow \langle \texttt{skip}, s'' \rangle \Rightarrow s''$$

y como $\langle \texttt{while } \neg b \texttt{ do } S, s'' \rangle \Rightarrow_1^k s'$, por determinismo debe suceder $s' = s''$. Veamos que la sentencia del `repeat` lleva al estado s'. Aplicamos la regla $[\text{repeat}_{\text{sos}}]$ para obtener

$$\langle \texttt{repeat } S \texttt{ until } b, s \rangle \Rightarrow \langle S; \texttt{if } b \texttt{ then skip else (repeat } S \texttt{ until } b), s \rangle$$

Ahora como $\langle S, s \rangle \Rightarrow^{k_1} s''$ aplicando el lema 2.21 se consigue que

$$\langle S; \texttt{if } b \texttt{ then skip else (repeat } S \texttt{ until } b), s \rangle \Rightarrow^{k_1} \langle \texttt{if } b \texttt{ then skip else (repeat } S \texttt{ until } b), s'' \rangle$$

Y ahora como $\mathcal{B}[\![b]\!]s'' = \mathbf{tt}$ se consigue mediante $[\text{if}^{\text{tt}}_{\text{sos}}]$ y $[\text{skip}_{\text{sos}}]$

$$\langle \texttt{if } b \texttt{ then skip else (repeat } S \texttt{ until } b), s'' \rangle \Rightarrow \langle \texttt{skip}, s'' \rangle \Rightarrow s''$$

Dado que $s' = s''$ se consigue finalmente

$$\langle \texttt{repeat } S \texttt{ until } b, s \rangle \Rightarrow^* s'$$

- Si $\mathcal{B}[\![b]\!]s'' = \mathbf{ff}$ ($\mathcal{B}[\![\neg b]\!]s'' = \mathbf{tt}$) entonces toca aplicar la regla $[\text{if}^{\text{tt}}_{\text{sos}}]$

$$\begin{aligned} \langle \texttt{if } \neg b \texttt{ then } S;\ \texttt{while } \neg b \texttt{ do } S \texttt{ else skip}, s'' \rangle &\Rightarrow \langle S; \texttt{while } \neg b \texttt{ do } S, s'' \rangle \\ &\Rightarrow^* s' \end{aligned}$$

Donde en el último paso se ha aplicado determinismo. Volvemos a aplicar lo mismo de antes para el `repeat`

$$\begin{aligned} \langle \texttt{repeat } S \texttt{ until } b, s \rangle &\overset{[\text{repeat}_{\text{sos}}]}{\Rightarrow} \langle S; \texttt{if } b \texttt{ then skip else (repeat } S \texttt{ until } b), s \rangle \\ &\overset{2,21}{\Rightarrow} \langle \texttt{if } b \texttt{ then skip else (repeat } S \texttt{ until } b), s'' \rangle \\ &\overset{[\text{if}^{\text{ff}}_{\text{sos}}]}{\Rightarrow} \langle \texttt{repeat } S \texttt{ until } b, s'' \rangle \end{aligned}$$

Por hipótesis de inducción se tiene que

$$\langle \texttt{repeat } S \texttt{ until } b, s'' \rangle \Rightarrow^* s' \iff \langle S; \texttt{while } \neg b \texttt{ do } S, s'' \rangle \Rightarrow^* s'$$

de donde se deduce que $\langle \texttt{repeat } S \texttt{ until } b, s'' \rangle \Rightarrow^* s'$ y finalmente

$$\langle \texttt{repeat } S \texttt{ until } b, s \rangle \Rightarrow^* s'$$

2.2.3. Expresiones

Análogamente a como hicimos en la semántica de paso largo, podemos definir el valor semántico de las expresiones mediante una función parcial $\mathcal{S}_{\text{sos}} : \mathbf{Stm} \to (\mathbf{State} \hookrightarrow \mathbf{State})$, donde

$$\mathcal{S}_{\text{sos}}[[S]] : \mathbf{State} \hookrightarrow \mathbf{State}$$

$$s \mapsto \begin{cases} s', \text{ si } \langle S, s \rangle \Rightarrow^* s' \\ \texttt{indefinido}, \text{ en otro caso} \end{cases}$$

Notemos que esta función está bien definida precisamente por el determinismo que vimos en el anterior apartado. La ejecución de $\langle S, s \rangle$ para una expresión S dada una configuración inicial $s \in \mathbf{State}$ puede dar lugar a tres casos

- Que termine, y entonces existe $s' \in \mathbf{State}$ tal que

$$s \in \mathbf{State} \Rightarrow^* s'$$

y entonces $\mathcal{S}_{\text{sos}}[[S]]s = s'$

- Que se quede bloqueada, y entonces no queda otra que $\mathcal{S}_{\text{sos}}[[S]]s = \texttt{indefinido}$.
- Que cicle, ocurriendo nuevamente que $\mathcal{S}_{\text{sos}}[[S]]s = \texttt{indefinido}$.

La equivalencia semántica coincide $\mathcal{S}_{\text{sos}}$ en el sentido de la siguiente proposición:

Proposición 2.26. *Si S_1 y S_2 son semánticamente equivalente entonces*

$$\mathcal{S}_{\text{sos}}[[S_1]] = \mathcal{S}_{\text{sos}}[[S_2]]$$

Demostración. Hay que probarlo particularizando en cada $s \in \textbf{State}$

Supongamos que S_1 y S_2 son semánticamente equivalentes. Se distinguen dos casos

- Existe un estado final $s' \in \textbf{State}$ tal que

  $$\langle S_1, s\rangle \Rightarrow^* s' \quad y \quad \langle S_2, s\rangle \Rightarrow^* s'$$

  y trivialmente

  $$\mathcal{S}_{\text{sos}}[[S_1]]s = s' = \mathcal{S}_{\text{sos}}[[S_2]]s$$

- Ambas secuencias son infinitas, es decir

  $$\langle S_1, s\rangle \text{ es infinita si y solo si } \langle S_2, s\rangle \text{ es infinita}$$

  y entonces

  $$\mathcal{S}_{\text{sos}}[[S_1]]s = \texttt{indefinido} = \mathcal{S}_{\text{sos}}[[S_2]]s$$

□

La implicación contraria no es cierta, pues si S_1 es una sentencia que cicla y S_2 es una sentencia que se bloquea, $\mathcal{S}_{\text{sos}}[[S_1]] = \texttt{indefinido} = \mathcal{S}_{\text{sos}}[[S_2]]$ pero S_1 y S_2 no son semánticamente equivalentes.

2.3. Teorema de equivalencia

Hasta ahora hemos presentado por separado los dos sistemas de transiciones para While, y hemos visto que tienen comportamientos, en general, distintos. Sin embargo, si vemos la semántica de paso largo como una extensión de la de paso corto, podemos convencernos que son esencialmente los mismo. De hecho, tienen el mismo poder expresivo. Más precisamente, el resultado clave de esta sección es:

Teorema 2.27 (De equivalencia)**.** *Consideremos el lenguaje* While. *Para cada $S \in \textbf{Stm}$, $\mathcal{S}_{\text{ns}}[[S]] = \mathcal{S}_{\text{sos}}[[S]]$.*

Demostración. La demostración se divide en dos pasos, cada uno consiste en demostrar uno de los siguientes lemas:

- Para cada $S \in \textbf{Stm}$, dados $s, s' \in \textbf{State}$, $\langle S, s\rangle \to s'$ implica que $\langle S, s\rangle \Rightarrow^* s'$.

  Se demuestra por inducción en los árboles de derivación.

- Para cada $S \in \textbf{Stm}$, dados $s, s' \in \textbf{State}$ y $k \in \mathbb{N}$, $\langle S, s\rangle \Rightarrow^k s'$ implica que $\langle S, s\rangle \to s'$.

  Se demuestra por inducción en la longitud de las secuencias de derivación.

Entonces, dados $S \in \textbf{Stm}$, $s \in \textbf{State}$, por los dos lemas se tiene que $\mathcal{S}_{\text{ns}}[[S]] = s'$ si y solo si $\mathcal{S}_{\text{sos}}[[S]] = s'$. Pero entonces obtenemos que, si una de las dos funciones está definida en un estado o no, la otra lo estará también y coincidirá con ella o no estará definida, respectivamente. Es decir, ambas funciones coinciden. □

3 | Más semántica operacional

La elección de una semántica operacional u otra para el lenguaje While es, por el Teorema de equivalencia, arbitraria. En cambio, para otros lenguajes puede ocurrir que una aproximación sea natural y la otra completamente impracticable. En esta sección estudiaremos una serie de conceptos que mostrarán la debilidad de las semánticas que hemos visto hasta ahora.

3.1. Construcciones no secuenciales

3.1.1. `abort`

La intención que tenemos al introducir la expresión `abort` es la siguiente: queremos que, cuando se ejecute, pare la ejecución de todo el programa. Notemos que no podríamos, a simple vista, construir una expresión con el mismo comportamiento en While. A saber, `while true do skip` solo consigue hacer un bucle, y `skip` permite que un programa se pueda ejecutar más tarde. Entonces, la sintaxis de las expresiones (o sentencias) queda del siguiente modo:

$$S ::= x := a \mid \texttt{skip} \mid S_1; S_2 \mid \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \mid \texttt{while } b \texttt{ do } S \mid \texttt{abort}$$

Por otro lado, aunque debemos modificar ahora el comportamiento de las relaciones de transición que vimos en el anterior capítulo, no es necesario alterar los sistemas de transiciones asociados, porque tan solo queremos que cualquier configuración de la forma $\langle \texttt{abort}, s\rangle$ quede atascada, y esto no modifica el comportamiento de (las reglas asociadas a) las otras expresiones. Es decir, la expresión `abort` no necesita de ninguna regla que determine su interpretación.

Ahora bien, no es cierto que While_{ns} y $\text{While}_{\text{sos}}$ se comporten, en general, del mismo modo. De hecho, `abort` distingue entre ambos sistemas de semántica operacional. Esto se debe a que en While_{ns} solo nos interesan las ejecuciones que terminan correctamente y por tanto no distinguimos entre bucles o configuraciones atascadas, mientras que en $\text{While}_{\text{sos}}$ podemos definir bucles (secuencias de derivación infinitas) y ejecuciones que terminan incorrectamente (secuencias de derivación finitas que terminan en una configuración atascada). Así, aunque `abort` no sea semánticamente equivalente a `skip` en While_{ns}, sí que lo es `while true do skip`. Por otro lado, vemos que `abort` no puede ser semánticamente equivalente en $\text{While}_{\text{sos}}$ a `while true do skip`, porque a partir de $\langle \texttt{while true do skip}, s\rangle$ hay una secuencia de derivación infinita y a partir de $\langle \texttt{abort}, s\rangle$ no. Tampoco puede serlo `skip`, porque $\langle \texttt{skip}, s\rangle \Rightarrow s$ es la única secuencia de derivación posible empezando en `skip` y $\langle \texttt{abort}, s\rangle$ es la correspondiente a `abort`.

Ejemplo 3.1. Podemos extender While con la expresión `assert` b `before` S. La idea es que, si b es cierto, entonces ejecutamos S y, si es falso, entonces la ejecución del programa se aborta. Para semántica operacional natural bastaría incluir la regla

$$\frac{\langle S, s\rangle \to s'}{\langle \texttt{assert } b \texttt{ before}, s\rangle \to s'} \text{ si } \mathcal{B}[\![b]\!]s = \texttt{tt}$$

y análogamente para la semántica operacional estructural

$$\frac{}{\langle \texttt{assert } b \texttt{ before}, s\rangle \Rightarrow \langle S, s\rangle} \text{ si } \mathcal{B}[\![b]\!]s = \texttt{tt}$$

Como se ha mencionado anteriormente no se añade ninguna regla para el caso $\mathcal{B}[\![b]\!]s = \texttt{ff}$ pues la idea es que en esta situación actúe como un `abort`.

3.1.2. `or`

Otra posible extensión de las expresiones de While consiste en añadir nuevas posibilidades de ejecución, es decir, forzar un tipo de no determinismo. Para ello incluimos la expresión `or`:

$$S ::= x := a \mid \texttt{skip} \mid S_1; S_2 \mid \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \mid \texttt{while } b \texttt{ do } S \mid S_1 \texttt{ or } S_2$$

Así, si por ejemplos ejecutamos `x` $:= 1$ `or x` $:= 2$, entonces generamos dos caminos distintos: uno en el que `x` pasa a tener el valor 1 y otro en el que toma el valor 2. Como siempre, en caso de que sea preciso, emplearemos paréntesis para indicar de manera precisa las distintas opciones.

A diferencia de la expresión `abort`, debemos distinguir la nueva ampliación de While en función de cada semántica operacional:

Sistema (While_{ns} con `or`)**.** A las reglas de While_{ns} añadimos:

$$[\text{or}^1_{\text{ns}}] \ \frac{\langle S_1, s\rangle \to s'}{\langle S_1 \texttt{ or } S_2, s\rangle \to s'}$$

$$[\text{or}^2_{\text{ns}}] \ \frac{\langle S_2, s\rangle \to s'}{\langle S_1 \texttt{ or } S_2, s\rangle \to s'}$$

Sistema ($\text{While}_{\text{sos}}$ con `or`)**.** A las reglas de $\text{While}_{\text{sos}}$ añadimos:

$$[\text{or}^1_{\text{sos}}] \ \frac{}{\langle S_1 \texttt{ or } S_2, s\rangle \Rightarrow \langle S_1, s\rangle}$$

$$[\text{or}^2_{\text{sos}}] \ \frac{}{\langle S_1 \texttt{ or } S_2, s\rangle \Rightarrow \langle S_2, s\rangle}$$

De nuevo, como cabía esperar, ambos sistemas de transiciones funcionan de maneras distintas respecto del no determinismo que hemos introducido. En el caso de While_{ns} tenemos que el no determinismo omite la posibilidad de que haya bucles infinitos, mientras que en $\text{While}_{\text{sos}}$ esto no ocurre. Recurramos al ejemplo de antes.

En While_{ns}, a la configuración $\langle \texttt{x} := 1 \texttt{ or x} := 2, s\rangle$ corresponden dos árboles de derivación, cada uno asociado a las siguientes transiciones:

$$\langle \texttt{x} := 1 \texttt{ or x} := 2, s\rangle \to s[\texttt{x} \mapsto 1]$$

$$\langle \texttt{x} := 1 \texttt{ or x} := 2, s\rangle \to s[\texttt{x} \mapsto 2]$$

Además, si tuviésemos $\langle (\texttt{while true do skip}) \texttt{ or x} := 2, s\rangle$, entonces solo habría un árbol de derivación posible, a saber, el que no corresponde al bucle, es decir,

$$\langle (\texttt{while true do skip}) \texttt{ or x} := 2, s\rangle \to s[\texttt{x} \mapsto 2]$$

En cambio, en $\text{While}_{\text{sos}}$, las secuencias de derivación para $\langle \texttt{x} := 1 \texttt{ or x} := 2, s\rangle$ serían

$$\langle \texttt{x} := 1 \texttt{ or x} := 2, s\rangle \Rightarrow^* s[\texttt{x} \mapsto 1]$$

$$\langle \texttt{x} := 1 \texttt{ or x} := 2, s\rangle \Rightarrow^* s[\texttt{x} \mapsto 2]$$

y para $\langle (\texttt{while true do skip}) \texttt{ or x} := 2, s\rangle$,

$$\langle (\texttt{while true do skip}) \texttt{ or x} := 2, s\rangle \Rightarrow^* s[\texttt{x} \mapsto 2]$$

$$\langle (\texttt{while true do skip}) \texttt{ or x} := 2, s\rangle \Rightarrow \langle \texttt{while true do skip}, s\rangle \Rightarrow \dots$$

de donde se deduce lo que dijimos arriba.

Ejemplo 3.2. Podemos extender While con la expresión `random(x)`. [HACER ESTO]

3.1.3. `par`

Una manera de profundizar en el no determinismo que vimos antes consiste en que, por ejemplo, se dé la posibilidad de ejecutar dos expresiones, pero no de forma excluyente, es decir, que ambas ejecuciones se puedan ejecutar intercalándose. Esta es la idea detrás de la expresión `par`:

$$S ::= x := a \mid \texttt{skip} \mid S_1; S_2 \mid \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \mid \texttt{while } b \texttt{ do } S \mid S_1 \texttt{ par } S_2$$

En un caso sencillo como $\texttt{x} := 1 \texttt{ par } \texttt{x} := 2$, tenemos solo los resultados posibles 1 y 2, correspondientes a ejecutar primero $\texttt{x} := 1$ y después $\texttt{x} := 2$ y viceversa. Si por ejemplo tuviésemos $\texttt{x} := 1 \texttt{ par } (\texttt{x} := 2; \texttt{x} := \texttt{x} + 2)$, entonces habría tres posibles resultados:

- Si ejecutamos $\texttt{x} := 1$ y luego $\texttt{x} := 2; \texttt{x} := \texttt{x} + 2$, obtenemos 4.
- Si ejecutamos $\texttt{x} := 2; \texttt{x} := \texttt{x} + 2$ y luego $\texttt{x} := 1$, obtenemos 1.
- Si ejecutamos $\texttt{x} := 2$, después $\texttt{x} := 1$ y por último $\texttt{x} := \texttt{x} + 2$, obtenemos 3.

Veamos cómo queda el lenguaje While ampliado con esta nueva expresión. En el caso de $\text{While}_{\text{sos}}$ tenemos:

Sistema ($\text{While}_{\text{sos}}$ con `par`)**.** A las reglas de $\text{While}_{\text{sos}}$ añadimos[1]:

$$[\text{par}^1_{\text{sos}}] \frac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1 \texttt{ par } S_2, s\rangle \Rightarrow \langle S_1' \texttt{ par } S_2, s'\rangle}$$

$$[\text{par}^2_{\text{sos}}] \frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1 \texttt{ par } S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}$$

$$[\text{par}^3_{\text{sos}}] \frac{\langle S_2, s\rangle \Rightarrow \langle S_2', s'\rangle}{\langle S_1 \texttt{ par } S_2, s\rangle \Rightarrow \langle S_1 \texttt{ par } S_2', s'\rangle}$$

$$[\text{par}^4_{\text{sos}}] \frac{\langle S_2, s\rangle \Rightarrow s'}{\langle S_1 \texttt{ par } S_2, s\rangle \Rightarrow \langle S_1, s'\rangle}$$

Si intentásemos ampliar While_{ns} de la misma manera, podríamos comenzar escribiendo las siguientes reglas:

$$\frac{\langle S_2, s'\rangle \to s''}{\langle S_1 \texttt{ par } S_2, s\rangle \to \langle S_1, s''\rangle}$$

$$\frac{\langle S_1, s'\rangle \to s''}{\langle S_1 \texttt{ par } S_2, s\rangle \to \langle S_1, s''\rangle}$$

Pero entonces observamos que estas reglas únicamente expresan que S_1 se ejecuta antes o después de S_2, es decir, no parecen traducir correctamente el valor semántico que tenemos en mente para `par`. Esto se debe a que en While_{ns} nos interesa solo el desarrollo general de la ejecución y, por tanto, la ejecución de una expresión no se puede descomponer en ejecuciones intermedias.

Ejemplo 3.3. Podemos desarrollar una sentencia que fuerce a que un conjunto de sentencias se ejecute de forma atómica. Por ejemplo

$$(x := 1) \texttt{ par } (x := 2; x := x + 2)$$

podría ejecutarse hasta de 3 formas, pudiendo intercalar las sentencias de la derecha. Para evitarlo podemos forzar a que la segunda parte se ejecute de forma atómica. Se define la sintaxis para `protect`

$$\texttt{protect } S \texttt{ end}$$

Para la semántica operacional natural se podría definir su semántica como

[1] Se hace notar que las reglas no son mutuamente excluyentes, es decir, podemos aplicar distintas reglas indistintamente, como hemos visto en los ejemplos anteriores.

$$[\text{protect}_{\text{ns}}]\ \frac{\langle S, s\rangle \to s'}{\langle \texttt{protect}\ \ S\ \ \texttt{end}, s\rangle \to s'}$$

aunque sería algo ambiguo pues en While_{ns} no es posible realizar paralelismo, como mencionamos anteriormente.

Sí se puede definir análogamente para la semántica estructural:

$$[\text{protect}_{\text{sos}}]\ \frac{\langle S, s\rangle \Rightarrow^* s'}{\langle \texttt{protect}\ \ S\ \ \texttt{end}, s\rangle \Rightarrow s'}$$

Se toma la premisa $\langle S, s\rangle \Rightarrow^* s'$ para resaltar el estado final de S. Nótese que se utiliza el asterisco para recalcar que ésta puede darse en una cantidad de pasos superior a uno.

3.2. Bloques y declaración de variables

Si bien en la anterior sección introdujimos nuevas expresiones que capturaran nuevos tipos de ejecuciones, lo que nos interesa ahora es añadir entornos para variables y procedimientos locales.

3.2.1. Bloques y declaraciones simples

La primera extensión de While que consideraremos es el lenguaje Block. Éste consiste en la siguiente extensión de la sintaxis de las expresiones de While:

$$S ::= \texttt{x} := a \mid \texttt{skip} \mid S_1; S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ b\ \texttt{do}\ S \mid \texttt{begin}\ \ D_V\ \ S\ \ \texttt{end}$$

donde D_V es una metavariable de la nueva categoría sintáctica $\mathbf{Dec}_V$ de *declaraciones de variables*:

$$D_V ::= \texttt{var}\ \ \texttt{x} := a; D_V \mid \varepsilon$$

siendo ε la *declaración vacía*. El conjunto de todas las variables declaradas en D_V, $DVar(D_V)$, se define como sigue:

$$DVar(\varepsilon) := \emptyset, \qquad DVar(\texttt{var}\ \ \texttt{x} := a; D_V) := \{\texttt{x}\} \cup DVar(D_V)$$

Queremos que esta nueva clase de expresiones se interprete como la ejecución de otra expresión cualquiera bajo la declaración de variables de D_V, y que éstas variables vuelvan al estado anterior después de finalizar. Es decir, la idea es que las variables declaradas dentro del *bloque* $\texttt{begin}\ \ D_V\ \ S\ \ \texttt{end}$ sean *locales*, a diferencia de las que quedan fuera, que llamamos *globales*.

Ejemplo 3.4. Consideremos la expresión

$$\texttt{begin var y} := 1; (\texttt{x} := 1; \texttt{begin var x} := 2; \texttt{y} := \texttt{x} + 1\ \ \texttt{end}; \texttt{x} := \texttt{x} + \texttt{y})\ \ \texttt{end}$$

El símbolo $\texttt{x}$ en $\texttt{y} := \texttt{x} + 1$ se refiere a la variable local introducida en $\texttt{var x} := 2$, mientras que en $\texttt{x} := \texttt{x} + \texttt{y}$ se refiere a la variable global $\texttt{x}$ en $\texttt{x} := 1$. La variable $\texttt{y}$ es global. Por tanto, $\texttt{y} := \texttt{x} + 1$ nos da el valor $2 + 1 = 3$ para $\texttt{y}$ y, en $\texttt{x} := \texttt{x} + \texttt{y}$, $\texttt{x}$ toma el valor $1 + 3 = 4$.

Nos podemos centrar en ampliar While_{ns} (hacerlo con $\text{While}_{\text{sos}}$ es considerablemente más difícil). Tendremos que distinguir un sistema de transiciones para cada una de las dos categorías sintácticas que hemos introducido antes. Las configuraciones asociadas a $\mathbf{Dec}_V$ son del tipo $\langle D_V, s\rangle$ o s, con el significado usual, y denotamos por $\langle D_V, s\rangle \to_D s'$ a la función de transición asociada. Entonces tenemos:

Sistema (Block_{ns})**.** A las reglas de While_{ns} añadimos:

$$[\text{var}_{\text{ns}}]\ \frac{\langle D_V, s[x \to \mathcal{A}[\![a]\!]s]\rangle \to_D s'}{\langle \texttt{var}\ \ x := a; D_V, s\rangle \to_D s'}$$

$$[\text{none}_{\text{ns}}]\ \frac{}{\langle \varepsilon, s\rangle \to_D s}$$

$$[\text{block}_{\text{ns}}]\ \frac{\langle D_V, s\rangle \to_D s' \qquad \langle S, s'\rangle \to s''}{\langle \texttt{begin}\ \ D_V\ \ S\ \texttt{end}, s\rangle \to s''[\text{DV}(D_v) \mapsto s]}$$

Fíjemonos en que el objetivo de la última regla es ejecutar primero la sentencia S con la declaración de variables local D_V y que, después, aquellas variables dentro de D_V vuelvan al estado previo al `begin`[2].

Ejemplo 3.5. Podríamos crear una función (sintáctica) que permita sustituir las expresiones de $\mathbf{Dec}_V$ por otras de **Stm**:

$$\begin{array}{rcl} \delta : \mathbf{Dec}_V & \longrightarrow & \mathbf{Stm} \\ \texttt{var x} := a; Dv & \mapsto & \texttt{x} := a; \delta(D_V) \\ \varepsilon & \mapsto & \texttt{skip} \end{array}$$

Entonces podríamos reinterpretar las reglas empleando la semántica que ya conocemos para While_{ns}. Por ejemplo, la regla block_{bs} podría expresarse como:

$$\frac{\langle \delta(D_V), s\rangle \to s' \qquad \langle S, s'\rangle \to s''}{\langle \texttt{begin}\ \ D_V\ \ S\ \texttt{end}, s\rangle \to s''[\text{DVar}(D_V) \mapsto s]}$$

es decir, sin emplear la transición $\to_D$. Las otras dos reglas se seguirían de las que ya vimos en While_{ns}.

3.2.2. Procedimientos

En gran parte de los lenguajes de programación, ciertos conjuntos de instrucciones se repiten constantemente. Esto motiva el siguiente concepto: un *procedimiento* será un conjunto de sentencias a las cuales les damos un nombre para, cuando queramos ejecutarlas de nuevo, sólamente debamos invocar tal nombre. Hay varias razones prácticas para hacer ésto.

Para formalizar esta idea, consideramos una ampliación de Block, Proc:

$$\begin{array}{rl} S ::= & \texttt{x} := a \mid \texttt{skip} \mid S_1; S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ b\ \texttt{do}\ S \mid \\ & \texttt{begin}\ \ D_V\ \ D_P\ \ S\ \ \texttt{end} \mid \texttt{call}\ \ p \\ D_V ::= & \texttt{var x} := a; D_V \mid \varepsilon \\ D_P ::= & \texttt{proc}\ \ p\ \ \texttt{is}\ \ S; D_P \mid \varepsilon \end{array}$$

donde $p \in \mathbf{Pname}$ es la categoría sintáctica de los *nombres de procedimientos* y $D_P \in \mathbf{Dec}_P$ es la de *declaraciones de procedimientos*. Notemos que hemos modificado `begin` y se ha añadido la sentencia `call`. $\varepsilon \in \mathbf{Dec}_P$ es la *declaración vacía*.

Para las variables hemos utilizado **State** para guardar sus valores. Se hace más complicado replicar lo mismo para los procesos. Además hemos utilizado en la sentencia `begin` la idea de variables locales y globales, según estuvieran definidas dentro de la sentencia `begin` o no. Ampliaremos este concepto para hablar de variables y procedimientos según el entorno en el que estén definidos. Por ejemplo, una función generalmente modifica variables internamente sin cambiar su valor fuera de ellas. Por tanto, como ya hicimos con las variables, distinguimos entre procedimientos *locales* y *globales*. Los nombres de las variables y los procedimientos, según el entorno en el que estén, se referirán a cosas diferentes. Vamos a distinguir dos tipos de declaración de variables y procedimientos: el *ámbito dinámico* y el *estático*. Según los apliquemos a un tipo u otro de datos, tenemos las siguientes posibilidades:

- Ámbito dinámico para variables y procedimientos
- Ámbito dinámico para variables y estático para procedimientos.
- Ámbito estático para variables y procedimientos.

[2] Véase la sección 1.5.2 para la definición de sustitución múltiple.

A ellas nos dedicaremos en los siguientes apartados.

Ejemplo 3.6. Sea la expresión[3]:

```
begin var x:= 0;
    proc p is x:= x*2;
    proc q is call p;
    begin var x:=5;
        proc p is x := x+1;
        call q; y:=x
    end
end
```

Imaginemos cada `begin` como una altura de un árbol. El proceso `p` está definido en dos profundidades: en la primera (línea 2) significa multiplicar por dos y en la segunda (línea 5) significa sumar 1 a la variable `x`. El procedimiento `q` estaría definido únicamente a profundidad 1.

Para hacernos a la idea, en un ámbito estático aquello definido a profundidad n tomará como referencias internas solamente cosas a profundidad $\leq n$, independientemente de que se ejecute a una profundidad mayor. En un ámbito dinámico utilizará las últimas según a qué profundidad se haya ejecutado. Veamos el ejemplo con las distintas casuísticas:

- Si se utiliza **ámbito estático para variables y procedimientos** la sentencia `call q` (profundidad 2) hará referencia a la `q` definida en la línea 3. Este procedimiento está definido a la profundidad 1, por lo que la referencia interna de la `p` que utilizará solo podrá estar a profunidad menor o igual. Es decir, el la invocación de `q` se referirá a la `p` de la línea 2 y no a la de la línea 5 (pues esta última está definida a profundidad 2).

  Es decir, la ejecución de `q` se toma a profundidad 2, pero hace referencia a un procedimiento a profundidad 1, luego no puede utilizar internamente la definición de `p` creada en la línea 5. Cuando el ámbito es estático la referencia se fija en tiempo de compilación y no de ejecución.

  Como el ámbito es estático también para variables, la invocación de la `p` definida en la segunda línea se referirá a la variable $x := 0$ definida en el primer `begin` (sin alterar, de esta forma, el valor de la x del segundo `begin`).

  En conclusión, el estado final en este caso será de $x := 0$ (se multiplica por dos, pero quedaría igual). De la misma forma, la variable y definida localmente en el segundo `begin` se quedará con el valor de 5, pues la x local no se modifica.

- Supongamos que se utiliza **ámbito estático para procedimientos** y **dinámico para variables**. Como ocurría antes la invocación de `q` en la línea 6 hará referencia a la `p` de la segunda línea. Sin embargo, dado que ahora estamos suponiendo un ámbito dinámico para variables, la referencia a x dentro de `p` será la última que esté disponible en tiempo de ejecución (es decir, aunque `p` venga dado a profundidad 1, como se ejecuta realmente a profundidad 2, podrá utilizar las variables a esta altura, pues es dinámico). Entonces la variable que se modificará será $x := 5$, definida en la línea 4, y la multiplicará por 2, quedando localmente $x = 10$ e $y = x = 10$. La variable x de profundidad 1 se quedará tal cual estaba.

- Si se da el caso en el que **procedimientos y variables están en ámbito dinámico** la invocación de `q` hará referencia a la última `p` referenciada en ejecución (es decir, la de la línea 5). A su vez, este procedimiento hará referencia a la última definición de la variable x. Quedará $x = 5 + 1 = 6$ e $y = 6$ localmente, con la variable $x = 0$ a profundidad 1.

[3]Para mayor comodidad, empezaremos a escribir las expresiones como pseudocódigo.

3.2.3. Ámbito dinámico para variables y procedimientos

Como hemos visto en el anterior ejemplo, la idea es que, para ejecutar `call p`, debemos ejecutar todo el procedimiento, es decir, tenemos que identificar cada procedimiento con su nombre. Para ello, definimos un *entorno de procedimiento*, que es una aplicación $env_P \in \mathbf{Env}_P := \mathbf{Pname} \hookrightarrow \mathbf{Stm}$, es decir, que nos lleva cada nombre de procedimiento a tal procedimiento.

Comencemos extendiendo While_{ns} con tal entorno. Debemos extender ahora las transiciones $\langle S, s\rangle \to s'$ usuales. Mediante la notación

$$env_P \vdash \langle S, s\rangle \to s'$$

nos referimos a que podemos acceder al entorno y por tanto hacer uso de los procedimientos declarados correspondientes, es decir, dado un entorno de procedimientos, hay una transición usual entre dos estados. Por otro lado, reflexionemos sobre el significado de `begin` $D_V\ D_P\ S$ `end`: queremos interpretarla para que, al ejecutar S, tengamos disponibles los procedimientos declarados en D_P. Para ello es necesario actualizar el entorno que hemos definido. Más concretamente, se define una aplicación que define recursivamente un entorno para procedimientos como $\text{upd}_P : \mathbf{Dec}_P \times \mathbf{Env}_P \to \mathbf{Env}_P$ dada por

$$\begin{array}{rcl} \text{udp}_P(\texttt{proc}\ p\ \texttt{is}\ S; D_P, env_P) & := & \text{udp}_P(D_P, env_P[p \mapsto S]) \\ \text{upd}_P(\varepsilon, env_P) & := & env_P \end{array}$$

Por tanto, las reglas de Proc quedan como:

Sistema (Proc, ámbito dinámico para variables y procedimientos)**.**

[ass_{ns}]

$$\frac{}{env_P \vdash \langle x := a, s\rangle \to s[x \mapsto \mathcal{A}[\![a]\!]s]}$$

[skip_{ns}]

$$\frac{}{env_P \vdash \langle \texttt{skip}, s\rangle \to s}$$

[comp_{ns}]

$$\frac{env_P \vdash \langle S_1, s\rangle \to s' \qquad env_P \vdash \langle S_2, s'\rangle \to s''}{env_P \vdash \langle S_1; S_2, s\rangle \to s''}$$

[$\text{if}^{\text{tt}}_{\text{ns}}$]

$$\frac{env_P \vdash \langle S_1, s\rangle \to s'}{env_P \vdash \langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \to s'} \quad \text{si } \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

[$\text{if}^{\text{ff}}_{\text{ns}}$]

$$\frac{env_P \vdash \langle S_2, s\rangle \to s'}{env_P \vdash \langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \to s'} \quad \text{si } \mathcal{B}[\![b]\!]s = \mathbf{ff}$$

[$\text{while}^{\text{tt}}_{\text{ns}}$]

$$\frac{env_P \vdash \langle S, s\rangle \to s' \qquad env_P \vdash \langle \texttt{while}\ b\ \texttt{do}\ S, s'\rangle \to s''}{env_P \vdash \langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \to s''} \quad \text{si } \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

[$\text{while}^{\text{ff}}_{\text{ns}}$]

$$\frac{}{env_P \vdash \langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \to s} \quad \text{si } \mathcal{B}[\![b]\!]s = \mathbf{ff}$$

[block_{ns}]

$$[\text{block}_{\text{ns}}]\ \frac{\langle D_V, s\rangle \to_D s' \qquad \text{upd}_P(D_P, env_P) \vdash \langle S, s'\rangle \to s''}{env_P \vdash \langle \texttt{begin}\ \ D_V\ \ S\ \texttt{end}, s\rangle \to s''[\text{DV}(D_v) \mapsto s]}$$

[$\text{call}^{\text{rec}}_{\text{ns}}$]

$$\frac{env_P \vdash \langle S, s\rangle \to s'}{env_P \vdash \langle \texttt{call p}, s\rangle \to s'} \quad \text{si } env_P(\texttt{p}) = S$$

Notemos que las variables no deben en principio acceder al entorno de procedimientos, luego no es necesario modificar las transiciones del tipo $\langle D_V, s\rangle \to_D s'$ que ya vimos en Block. Es importante notar que permitimos que los procedimientos sean recursivos y que, si llamamos a un procedimiento inexistente, entonces el programa abortará.

3.2.4. Ámbito estático para procedimientos

Para modificar Block siguiendo este enfoque, debemos asegurarnos que cada procedimiento que llamemos sea capaz de 'recordar' los procedimientos que ya se habían introducido antes de que fuese declarado. Intuitivamente, debemos asociar cada nombre con el procedimiento al que se refiere, junto con el entorno de procedimientos que había en el momento de declararlo. La manera formal de hacer ésto consiste en extender env_P como una función de $\mathbf{Env}_P := \mathbf{Pname} \hookrightarrow \mathbf{Stm} \times \mathbf{Env}_P$. Este tipo está bien definido, precisamente porque definiremos el entorno de cada procedimiento en función de otros menores, hasta llegar al *entorno de procedimientos vacío*:

$$\begin{aligned} \text{udp}_P(\texttt{proc}\ p\ \texttt{is}\ S; D_P, env_P) &:= \text{udp}_P(D_P, env_P[p \mapsto (S, env_P)]) \\ \text{upd}_P(\varepsilon, env_P) &:= env_P \end{aligned}$$

Tampoco tendremos que cambiar esta vez las declaraciones de variables. Solo tenemos que cambiar las reglas para llamadas de procedimientos. Tenemos dos casos:

- Si asumimos que los procedimientos de Proc son no recursivos, entonces solo tenemos que conocer el entorno del procedimiento para determinarlo junto con su entorno en el momento de la declaración.
- Si asumimos que los procedimientos de Proc son recursivos, tenemos que asegurar que las apariciones de `call` p en el cuerpo de p se refieren al procedimiento mismo, es decir, debemos modificar el entorno del procedimiento para que contenga tal información.

Por tanto, tenemos:

Sistema (Proc, ámbito estático para procedimientos no recursivos)**.** En las reglas de Proc, intercambiamos [$\text{call}^{\text{rec}}_{\text{ns}}$] por:

$$[\text{call}_{\text{ns}}]\ \frac{env'_P \vdash \langle S, s\rangle \to s'}{env_P \vdash \langle \texttt{call p}, s\rangle \to s'} \quad \text{si } env_P(\texttt{p}) = (S, env'_P)$$

Y

Sistema (Proc, ámbito estático para procedimientos recursivos)**.** En las reglas de Proc, intercambiamos [$\text{call}^{\text{rec}}_{\text{ns}}$] por[4]:

$$[\text{call}^{\text{rec}}_{\text{ns}}]\ \frac{env'_P[p \mapsto (S, env'_P)] \vdash \langle S, s\rangle \to s'}{env_P \vdash \langle \texttt{call p}, s\rangle \to s'} \quad \text{si } env_P(\texttt{p}) = (S, env'_P)$$

[4]Por env'_P nos referimos a otra metavariable distinta de env_P. Podríamos haber escogido letras como nombres, pero env_P nos recuerda inmediatamente lo que simboliza.

3.2.5. Ámbito estático para variables y procedimientos

Ahora, siguiendo las modificaciones que acabamos de ver, debemos cambiar el funcionamiento de los estados. Es decir, reemplazaremos los estados con dos aplicaciones. La primera consiste en un *entorno de variable*, que lleva cada variable en una *ubicación*, $env_V \in \mathbf{Env}_V : \mathbf{Var} \to \mathbf{Loc}$. Conviene pensar en las ubicaciones como celdas en una memoria abstracta que se van rellenando con datos.

El siguiente tipo de aplicación, llamada *función de almacenamiento*, asocia cada ubicación con un entero, $sto \in \mathbf{Store} : \mathbf{Loc} \cup \{\texttt{next}\} \to \mathbb{Z}$, donde `next` es una expresión que sirve para guardar un valor en la siguiente ubicación libre. Para que ésto quede bien formalizado, necesitamos, finalmente, $new : \mathbf{Loc} \to \mathbf{Loc}$, que lleva cada ubicación en la siguiente que esté vacía. Para *actualizar* el entorno de variable y la función de almacenamiento, empleamos las declaraciones de variables.

Nosotros tomaremos $\mathbf{Loc} := \mathbb{Z}$, es decir, iremos asociando variables a ubicaciones etiquetadas con enteros. Por tanto, en nuestro caso, la aplicación *new* es simplemente la función sucesor. Aunque sean el mismo conjunto, denotaremos el tipo de las ubicaciones como **Loc** y a los enteros de la forma usual para mayor claridad.

Dijimos que queríamos reemplazar los estados con estas dos aplicaciones que hemos definido. La forma de hacerlo es la siguiente: $s := sto \circ env_V$. De este modo, tenemos que:

- Para *determinar* el valor de $x \in \mathbf{Var}$, debemos determinar primero una ubicación $l := env_V(x)$ y después un valor $sto(l) \in \mathbb{Z}$.
- Para *asignar* un valor $v \in \mathbb{Z}$ a $x \in \mathbf{Var}$, debemos determinar primero la ubicación $l := env_V(x)$ y después actualizar (el sentido que dijimos antes) **Store** hasta que $sto(l) = v$.

Debemos considerar el entorno de variables y el almacenamiento inicial, es decir, debemos fijar unos valores antes de la ejecución del programa. Podríamos, por ejemplo, hacer que el entorno de variables inicial llevase todas las variables a la ubicación etiquetada por 0 y que la función de almacenamiento inicial llevase `next` en la ubicación etiquetada por 1 (desde 0).

En resumen, las configuraciones para declaraciones de variables no dependen ahora de los estados sino de las funciones env_V y sto (¡lo que cambia son estas funciones, no los estados!). Ahora, en vez de tener un objeto como $\mathcal{A}[\![a]\!]s$, por ejemplo, tendremos una función $\mathcal{A}[\![a]\!](sto \circ env_V)$. Más precisamente, sustituimos los estados por pares del tipo (env_V, sto).

Conviene comparar la siguiente función de transición con la que dimos al principio de 3.2.1:

$$\langle D_V, (env_V, sto)\rangle \to_D (env'_V, sto')$$

A continuación resumimos su semántica asociada:

$$[\text{var}_{\text{ns}}]\ \frac{\langle D_V, (env_V[x \mapsto l], sto[l \mapsto v][\texttt{next} \mapsto new(l)])\rangle \to_D (env'_V, sto')}{\langle \texttt{var}\ \ x := a; D_V, (env_V, sto)\rangle \to_D (env'_V, sto')}$$

donde $v := \mathcal{A}[\![a]\!](sto \circ env_V)$ y $l := sto(\texttt{next})$.

$$[\text{none}_{\text{ns}}]\ \frac{}{\langle \varepsilon, env_V, sto\rangle \to_D (env_V, sto)}$$

El siguiente paso consiste en extender el entorno de procedimientos para guardar el entorno de cada variable en el momento de la declaración. Por tanto, consideraremos funciones $env_P \in \mathbf{Env}_P := \mathbf{Pname} \hookrightarrow \mathbf{Stm} \times \mathbf{Env}_V \times \mathbf{Env}_P$. Por otro lado, definimos:

$$\begin{aligned} \text{udp}_P(\texttt{proc}\ p\ \texttt{is}\ S; D_P, env_V, env_P) &:= \text{udp}_P(D_P, env_V, env_P[p \mapsto (S, env_V, env_P)]) \\ \text{upd}_P(\varepsilon, env_V, env_P) &:= env_P \end{aligned}$$

Las transiciones tendrán entonces la forma $env_V, env_P \vdash \langle S, sto\rangle \to sto'$, es decir, dados dos entornos respectivos de variable y procedimiento, hay una transición entre dos ubicaciones, una inicial y otra final.

Sistema (Proc, ámbito estático para variables)**.**

[ass_{ns}]

$$\frac{}{env_V, env_P \vdash \langle x := a, sto\rangle \to sto[l \mapsto v]}$$

donde $l := env_V(x)$ y $v := \mathcal{A}[\![a]\!](sto \circ env_V)$.

[skip_{ns}]

$$\frac{}{env_V, env_P \vdash \langle \texttt{skip}, sto\rangle \to sto}$$

[comp_{ns}]

$$\frac{env_V, env_P \vdash \langle S_1, sto\rangle \to sto' \qquad env_V, env_P \vdash \langle S_2, sto'\rangle \to sto''}{env_V, env_P \vdash \langle S_1; S_2, sto\rangle \to sto''}$$

[$\text{if}^{\text{tt}}_{\text{ns}}$]

$$\frac{env_V, env_P \vdash \langle S_1, sto\rangle \to sto'}{env_V, env_P \vdash \langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, sto\rangle \to sto'}$$

si $\mathcal{B}[\![b]\!](sto \circ env_V) = \mathbf{tt}$

[$\text{if}^{\text{ff}}_{\text{ns}}$]

$$\frac{env_V, env_P \vdash \langle S_2, sto\rangle \to sto'}{env_V, env_P \vdash \langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, sto\rangle \to sto'}$$

si $\mathcal{B}[\![b]\!](sto \circ env_V) = \mathbf{ff}$

[$\text{while}^{\text{tt}}_{\text{ns}}$]

$$\frac{env_V, env_P \vdash \langle S, sto\rangle \to sto' \qquad env_V, env_P \vdash \langle \texttt{while } b \texttt{ do } S, sto'\rangle \to sto''}{env_V, env_P \vdash \langle \texttt{while } b \texttt{ do } S, sto\rangle \to sto''}$$

si $\mathcal{B}[\![b]\!](sto \circ env_V) = \mathbf{tt}$

[$\text{while}^{\text{ff}}_{\text{ns}}$]

$$\frac{}{env_V, env_P \vdash \langle \texttt{while } b \texttt{ do } S, sto\rangle \to sto} \qquad \text{si } \mathcal{B}[\![b]\!](sto \circ env_V) = \mathbf{ff}$$

[block_{ns}]

$$\frac{\langle D_V, (env_V, sto)\rangle \to_D (env'_V, sto') \qquad env'_V, env'_P \vdash \langle S, sto'\rangle \to sto''}{env_V, env_P \vdash \langle \texttt{begin } D_V\ D_P\ S \texttt{ end}, sto\rangle \to sto''}$$

donde $env'_P := \text{upd}_P(D_P, env'_V, env_P)$.

[call_{ns}]

$$\frac{env'_V, env'_P \vdash \langle S, sto\rangle \to sto'}{env_V, env_P \vdash \langle \texttt{call p}, sto\rangle \to sto'}$$

donde $env_P(p) := (S, env'_V, env'_P)$.

$[\text{call}^{\text{rec}}_{\text{ns}}]$

$$\frac{env'_V, env'_P[p \mapsto (S, env'_V, env'_P)] \vdash \langle S, sto\rangle \to sto'}{env_V, env_P \vdash \langle \texttt{call p}, sto\rangle \to sto'}$$

donde $env_P(p) := (S, env'_V, env'_P)$.

Notemos que no puede haber uan expresión parecida a $s''[DV(D_V) \mapsto s]$, porque las variables solo toman valores accediendo a su entorno asociado.

4 | Implementación correcta

Una pregunta a realizarse sería cómo implementar un lenguaje de programación para que funcione en un ordenador. En este capítulo se mostrará, *grosso modo*, cómo realizar una implementación basándonos en cómo funciona Java. Para ello, estudiaremos los siguientes conceptos:

- Una máquina abstracta con unas instrucciones básicas.
- Traducción de **Stm** a código de la máquina abstracta.
- Corrección.
- Bisimulación.

La idea es que definiremos el comportamiento semántico de las instrucciones de la máquina abstracta mediante semántica operacional y después estudiaremos traducciones que lleven elementos de While en secuencias de estas instrucciones. Con *corrección* nos referimos a que, si traducimos una expresión en *código* y lo ejecutamos, entonces obtendremos el mismo resultado que el que ya describimos mediante $\mathcal{S}_{\text{ns}}$ o $\mathcal{S}_{\text{sos}}$. Finalmente, veremos de pasada la bisimulación.

4.1. Máquina abstracta

4.1.1. Configuraciones e introducciones

Una *configuración* de la máquina abstracta (MA) consta de:

- Una secuencia de instrucciones a ejecutar (o código), s.
- Una pila de evaluación, e.
- Un almacén, s,

Denotaremos por **Code** a la categoría sintáctica de las *secuencias de instrucciones*. La pila de evaluación servirá para evaluar expresiones aritméticas y booleanas. Formalmente será una lista de valores:

$$\textbf{Stack} := (\mathbb{Z} \cup T)^*$$

De esta forma, una configuración será una terna $\langle c, e, s\rangle \in \textbf{Code} \times \textbf{Stack} \times \textbf{State}$. Para mayor simplicidad, se considerá el almacén de forma parecida a como consideramos los elementos de **State** para almacenar variables.

La gramática de las instrucciones de la máquina abstracta es de la siguiente forma:

$$\begin{array}{rcl} inst & ::= & \texttt{PUSH} - n \mid \texttt{ADD} \mid \texttt{MULT} \mid \texttt{SUB} \\ & \mid & \texttt{TRUE} \mid \texttt{FALSE} \mid \texttt{EQ} \mid \texttt{NEG} \mid \texttt{AND} \\ & \mid & \texttt{FETCH} - x \mid \texttt{STORE} - x \\ & \mid & \texttt{NOOP} \mid \texttt{BRANCH}(c, c) \mid \texttt{LOOP}(c, c) \\ c & ::= & \varepsilon \mid inst : c \end{array}$$

con $c \in \mathbf{Code}$.

Se denomina *configuración final* a las ternas de la forma $\langle \varepsilon, e, s\rangle$, es decir, con el código vacío. La idea va a ser que, a partir de una configuración no final, haya una transición que vaya eliminando código y modificando la pila de evaluación y el almacén.

Pasemos a describir el sistema de semántica operacional asociado a la MA. Definimos una relación de transición:

$$\langle c, e, s\rangle \rhd \langle c', e', s'\rangle$$

que significa lo siguiente: la configuración $\langle c', e', s'\rangle$ se obtiene de $\langle c, e, s\rangle$ *en un paso de ejecución*. Nótese la similitud con la semántica operacional de paso corto. Esta relación viene determinada por el siguiente sistema de transiciones:

Sistema (Semántica operacional para la MA)**.**

$$\begin{array}{rcl}
\langle \texttt{PUSH} - n : c, e, s\rangle & \rhd & \langle c, \mathcal{N}[\![n]\!] : e, s\rangle \\
\langle \texttt{ADD} : c, z_1 : z_2 : e, s\rangle & \rhd & \langle c, (z_1 \oplus z_2) : e, s\rangle \quad si\ z_1, z_2 \in \mathbf{Z} \\
\langle \texttt{MULT} : c, z_1 : z_2 : e, s\rangle & \rhd & \langle c, (z_1 \otimes z_2) : e, s\rangle \quad si\ z_1, z_2 \in \mathbf{Z} \\
\langle \texttt{SUB} : c, z_1 : z_2 : e, s\rangle & \rhd & \langle c, (z_1 \ominus z_2) : e, s\rangle \quad si\ z_1, z_2 \in \mathbf{Z} \\
\langle \texttt{TRUE} : c, e, s\rangle & \rhd & \langle c, \mathbf{tt} : e, s\rangle \\
\langle \texttt{FALSE} : c, e, s\rangle & \rhd & \langle c, \mathbf{ff} : e, s\rangle \\
\langle \texttt{EQ} : c, z_1 : z_2 : e, s\rangle & \rhd & \langle c, (z_1 z_2) : e, s\rangle \quad si\ z_1, z_2 \in \mathbf{Z} \\
\langle \texttt{LE} : c, z_1 : z_2 : e, s\rangle & \rhd & \langle c, (z_1 \leq z_2) : e, s\rangle \quad si\ z_1, z_2 \in \mathbf{Z} \\
\langle \texttt{AND} : c, t_1 : t_2 : e, s\rangle & \rhd & \begin{cases} \langle c, \mathbf{tt} : e, s\rangle & \text{si } t_1 = t_2 = \mathbf{tt} \\ \langle c, \mathbf{ff} : e, s\rangle & \text{caso contrario} \end{cases} \\
\langle \texttt{NEG} : c, t : e, s\rangle & \rhd & \begin{cases} \langle c, \mathbf{ff} : e, s\rangle & \text{si } t = \mathbf{tt} \\ \langle c, \mathbf{tt} : e, s\rangle & \text{si } t = \mathbf{ff} \end{cases} \\
\langle \texttt{FETCH-x} : c, x, e\rangle & \rhd & \langle c, s(x) : e, s\rangle \\
\langle \texttt{STORE-x} : c, z : e, s\rangle & \rhd & \langle c, e, s[x \to z]\rangle \quad si\ z \in \mathbf{Z} \\
\langle \texttt{NOOP} : c, e, s\rangle & \rhd & \langle c, e, s\rangle \\
\langle \texttt{BRANCH}(c_1, c_2) : c, t : e, s\rangle & \rhd & \begin{cases} \langle c_1 : c, e, s\rangle & \text{si } t = \mathbf{tt} \\ \langle c_2 : c, e, s\rangle & \text{si } t = \mathbf{ff} \end{cases} \\
\langle \texttt{LOOP}(c_1, c_2) : c, e, s\rangle & \rhd & \langle c_1 : \texttt{BRANCH}(c_2, \texttt{LOOP}(c_1, c_2), \texttt{NOOP} : c, e, s\rangle
\end{array}$$

Donde hemos usado ':' con doble sentido: por un lado, nos referimos a la concatenación de dos secuencias de isntrucciones (algo similar a ';' para **Stm**) y, por otro, a la operación de anteponer un elemento a una tal secuencia. También conviene aclarar algunos de los códigos. El código $\texttt{LOOP}(c_1, c_2)$ es el más complejo y es el que servirá más adelante para traducir la sentencia `While`. El código $\texttt{BRANCH}(c_1, c_2)$ toma la idea de actuar como un `if`, siendo la condición la que esté en la cima de la pila.

4.1.2. Propiedades semánticas

Como ya hemos dicho, el sistema de transiciones anterior no es más que un sistema de semántica operacional estructural. Por tanto, podemos reescribir los conceptos que ya vimos anteriomente para esta semántica. Por ejemplo, las secuencias de derivación se denominan *secuencias de cómputo*. Una secuencia de cómputo para un código c y un almacén s puede ser *finita*, es decir, de la forma

$$\gamma_0 \rhd \gamma_1 \rhd ... \rhd \gamma_k$$

siendo $\gamma_i \in \mathbf{Code} \times \mathbf{Stack} \times \mathbf{State}$ para todo $i \leq k$ y cumpliendo $\gamma_i \rhd \gamma_{i+1}$ para $i < k$. Por otro lado, denotamos una secuencia de cómputo *infinita* como

$$\gamma_0 \rhd \gamma_1 \rhd ...$$

con $\gamma_i \rhd \gamma_{i+1}$ para $i \geq 0$. En ambos casos, la configuración inicial será de la forma $\gamma_0 := \langle c, \varepsilon, s\rangle$, es decir, las secuencias empiezan con la pila vacía. Además, decimos que una secuencia de cómputo:

- *Termina* si y solo si es finita.
- *Cicla* si y solo si es infinita.

Nótese que, en el primer caso, se puede terminar con una configuración final o con una *configuración atascada*, como por ejemplo $\langle \texttt{ADD}, \varepsilon, s\rangle$.

Para no repetir las demostraciones que ya hemos visto, enunciaremos las diferentes propiedades básicas del sistema que acabamos de presentar:

Lema 4.1. *Si* $\langle c_1, e_1, s\rangle \rhd^k \langle c', e', s'\rangle$, *entonces* $\langle c_1 : c_2, e_1 : e_2, s\rangle \rhd^k \langle c' : c_2, e' : e_2, s'\rangle$.

Demostración. Véase la demostración del Lema 2.21. □

Lema 4.2. *Si* $\langle c_1 : c_2, e, s\rangle \rhd^k \langle \varepsilon, e'', s''\rangle$, *entonces existe una configuración* $\langle \varepsilon, e', s'\rangle$ *y* $k_1, k_2 \in \mathbb{N}$ *tales que* $k = k_1 + k_2$ *y*

$$\langle c_1, e, s\rangle \rhd^{k_1} \langle \varepsilon, e', s'\rangle \quad y \quad \langle c_2, e', s'\rangle \rhd^{k_2} \langle \varepsilon, e'', s''\rangle.$$

Demostración. Véase la demostración del Lema 2.19. □

Teorema 4.3. *El sistema de transiciones para el código de la máquina abstracta es determinista, es decir,*

$$\gamma \rhd \gamma' \text{ y } \gamma \rhd \gamma'' \quad \textit{implica} \quad \gamma' = \gamma''$$

Demostración. Véase la demostración del Teorema 2.22. □

4.1.3. Función de ejecución

De nuevo, por analogía sobre lo que ya comentamos en el capítulo de semántica operacional estructural, aparece la idea de una función que permita definir el valor semántico de cada expresión (aquí, secuencia de instrucciones). Más concretamente, definimos la función parcial $\mathcal{M} : \textbf{Code} \to (\textbf{State} \hookrightarrow \textbf{State})$:

$$\mathcal{M}[\![c]\!]s := \begin{cases} s', \text{ si } \langle c, \varepsilon, s\rangle \rhd^* \langle \varepsilon, e, s'\rangle \\ \texttt{INDEFINIDO}, \text{ en otro caso.} \end{cases}$$

Que $\mathcal{M}$ esté bien definida se sigue por determinismo. Conviene hacer notar que esta definición requiere que la componente del código, en la configuración final, sea vacía.

4.2. Traducción

Ahora estudiamos cómo generar código para la máquina abstracta. Para ello, debemos precisar el comportamiento de expresiones y sentencias, así como la relación con `While`.

4.2.1. Expresiones y sentencias

Para traducir expresiones (aritméticas y booleanas) debemos proceder composicionalmente. Recordemos que esto era relevante a la hora de especificar una propiedad o (como en nuestro caso) una aplicación sobre elementos sintácticos. Conseguimos la composicionalidad con las siguientes definiciones:

$$\begin{aligned} \mathcal{CA} : \textbf{Aexp} &\longrightarrow \textbf{Code} \\ n &\mapsto \texttt{PUSH} - n \\ x &\mapsto \texttt{FETCH} - x \\ a_1 + a_2 &\mapsto \mathcal{CA}[\![a_2]\!] : \mathcal{CA}[\![a_1]\!] : \texttt{ADD} \\ a_1 * a_2 &\mapsto \mathcal{CA}[\![a_2]\!] : \mathcal{CA}[\![a_1]\!] : \texttt{MULT} \\ a_1 - a_2 &\mapsto \mathcal{CA}[\![a_2]\!] : \mathcal{CA}[\![a_1]\!] : \texttt{SUB} \end{aligned}$$

$$
\begin{aligned}
\mathcal{CB} : \mathbf{Bexp} &\longrightarrow \mathbf{Code} \\
\texttt{true} &\mapsto \texttt{TRUE} \\
\texttt{false} &\mapsto \texttt{FALSE} \\
a_1 = a_2 &\mapsto \mathcal{CA}[\![a_2]\!] : \mathcal{CA}[\![a_1]\!] : \texttt{EQ} \\
a_1 \leq a_2 &\mapsto \mathcal{CA}[\![a_2]\!] : \mathcal{CA}[\![a_1]\!] : \texttt{LE} \\
\neg b &\mapsto \mathcal{CB}[\![b]\!] : \texttt{NEG} \\
b_1 \wedge b_2 &\mapsto \mathcal{CB}[\![b_2]\!] : \mathcal{CB}[\![b_1]\!] : \texttt{AND}
\end{aligned}
$$

Se deben distinguir, evidentemente, los casos base y los inductivos. El orden que seguimos para el código no es arbitrario: véase el sistema de transiciones asociado a la máquina abstracta.

Ejemplo 4.4. $\mathcal{CA}[\![\texttt{x} + 1]\!] = \mathcal{CA}[\![1]\!] : \mathcal{CA}[\![\texttt{x}]\!] : \texttt{ADD} = \texttt{PUSH} - 1 : \texttt{FETCH} - x : \texttt{ADD}$.

Conviene indicar que, precisamente por el hecho de que mediante las aplicaciones $\mathcal{CA}$ y $\mathcal{CB}$ vamos 'apilando' información, se puede dar el caso de que dos expresiones que son iguales en While no lo son en el código. Por ejemplo, $\mathcal{A}[\![a_1 + (a_2 + a_3)]\!] = \mathcal{A}[\![(a_1 + a_2) + a_3]\!]$ pero $\mathcal{CA}[\![a_1 + (a_2 + a_3)]\!] \neq \mathcal{CA}[\![(a_1 + a_2) + a_3]\!]$. Únicamente podemos decir que ambas expresiones se 'comportan' de la misma forma.

Siguiendo un razonamiento similar al anterior, definimos la aplicación:

$$
\begin{aligned}
\mathcal{CB} : \mathbf{Stm} &\longrightarrow \mathbf{Code} \\
x := a &\mapsto \mathcal{CA}[\![a]\!] : \texttt{STORE} - x \\
\texttt{skip} &\mapsto \texttt{NOOP} \\
S_1; S_2 &\mapsto \mathcal{CA}[\![S_2]\!] : \mathcal{CA}[\![S_1]\!] \\
\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 &\mapsto \mathcal{CB}[\![b]\!] : \texttt{BRANCH}(\mathcal{CS}[\![S_1]\!], \mathcal{CS}[\![S_1]\!]) \\
\texttt{while } b \texttt{ do } S &\mapsto \texttt{LOOP}(\mathcal{CB}[\![b]\!], \mathcal{CS}[\![S]\!])
\end{aligned}
$$

Notemos que, de nuevo, esta definición es composicional. Además, no se genera nunca código vacío.

Ejemplo 4.5. Supongamos que queremos generar código para la expresión `repeat` S `until` b. Podríamos tratar de definir (composicionalmente) una secuencia de instrucciones a partir de esta expresión. Una alternativa consiste en emplear que esta expresión es semánticamente a $S; \texttt{while } \neg b \texttt{ do } S$ y simplemente traducirla usando las aplicaciones que acabamos de introducir.

4.2.2. La función semántica para la máquina abstracta

Tras haber convertido las expresiones aritméticas, booleanas y las sentencias del lenguaje While se puede proceder a definir una función $\mathcal{S}_{\text{am}}[[\cdot]] : \mathbf{Stm} \to (\mathbf{State} \hookrightarrow \mathbf{State})$ que primero traduzca a código de la máquina y después aplique la función de ejecución. Matemáticamente se puede definir como

$$\mathcal{S}_{\text{am}} := \mathcal{M} \circ \mathcal{CS}$$

Ejemplo 4.6. Veamos por ejemplo el resultado de aplicar $\mathcal{S}_{\text{am}}$ a la sentencia $x := 1; y := x + 1$. Se traduce a código máquina

$$
\begin{aligned}
\mathcal{CS}[\![x := 1; y := x + 1]\!] &= \mathcal{CS}[\![x := 1]\!] : \mathcal{CS}[\![y := x + 1]\!] \\
&= \mathcal{CA}[\![1]\!] : \texttt{STORE} - x : \mathcal{CS}[\![y := x + 1]\!] \\
&= \mathcal{CA}[\![1]\!] : \texttt{STORE} - x : \mathcal{CA}[\![x + 1]\!] : \texttt{STORE} - y \\
&= \texttt{PUSH} - 1 : \texttt{STORE} - x : \mathcal{CA}[\![x]\!] : \mathcal{CA}[\![1]\!] : \texttt{ADD} : \texttt{STORE} - y \\
&= \texttt{PUSH} - 1 : \texttt{STORE} - x : \texttt{FETCH} - x : \texttt{PUSH} - 1 : \texttt{ADD} : \texttt{STORE} - y
\end{aligned}
$$

y ahora se pasa dicho código por ejecución

$$\begin{aligned}
& \langle \texttt{PUSH} - 1 : \texttt{STORE} - x : \texttt{FETCH} - x : \texttt{PUSH} - 1 : \texttt{ADD} : \texttt{STORE} - y, \varepsilon, s\rangle \\
\rhd\ & \langle \texttt{STORE} - x : \texttt{FETCH} - x : \texttt{PUSH} - 1 : \texttt{ADD} : \texttt{STORE} - y, 1, s\rangle \\
\rhd\ & \langle \texttt{FETCH} - x : \texttt{PUSH} - 1 : \texttt{ADD} : \texttt{STORE} - y, \varepsilon, s[x \mapsto 1]\rangle \\
\rhd\ & \langle \texttt{PUSH} - 1 : \texttt{ADD} : \texttt{STORE} - y, s[x \mapsto 1](x), s[x \mapsto 1]\rangle \\
\rhd\ & \langle \texttt{PUSH} - 1 : \texttt{ADD} : \texttt{STORE} - y, 1, s[x \mapsto 1]\rangle \\
\rhd\ & \langle \texttt{ADD} : \texttt{STORE} - y, 2 : 1, s[x \mapsto 1]\rangle \\
\rhd\ & \langle \texttt{STORE} - y, 3, s[x \mapsto 1]\rangle \\
\rhd\ & \langle \varepsilon, \varepsilon, (s[x \mapsto 1])[y \mapsto 3]\rangle
\end{aligned}$$

luego $\mathcal{S}_{\mathrm{am}}[[x := 1; y := x + 1]] = (s[x \mapsto 1])[y \mapsto 3]$.

4.3. Corrección

Recordemos que el objetivo ahora consiste en comprobar que, si ejecutamos un código para una expresión de While en **Code**, el resultado es el mismo que obtendríamos siguiendo la semántica operacional de While. Por tanto, la forma de proceder vuelver a ser composicional, es decir, debemos realizar un razonamiento que siga cierta inducción estructural.

Para expresiones, tenemos unos resultados precisos:

Lema 4.7. *Sea $a \in$ **Aexp**. Entonces $\langle \mathcal{CA}[\![a]\!], \varepsilon, s\rangle \rhd^* \langle \varepsilon, \mathcal{A}[\![a]\!]s, s\rangle$ y cada configuración intermedia de esta secuencia de derivación tendrá pila no vacía.*

Lema 4.8. *Sea $b \in$ **Bexp**. Entonces $\langle \mathcal{CB}[\![b]\!], \varepsilon, s\rangle \rhd^* \langle \varepsilon, \mathcal{B}[\![b]\!]s, s\rangle$ y cada configuración intermedia de esta secuencia de derivación tendrá pila no vacía.*

Una pregunta natural es: ¿qué semántica de While emplear para comprobar la corrección de la traducción? Nosotros usaremos la semántica natural. Sin embargo, en la siguiente sección discutiremos una alternativa a este enfoque. El enunciado preciso es el siguiente:

Teorema 4.9. *Sea $S \in$ **Stm**. Entonces, $\mathcal{S}_{\mathrm{ns}}[[S]] = \mathcal{S}_{\mathrm{am}}[[S]]$.*

Demostración. La demostración consiste en dos lemas:

Lema 4.10. *Dados $S \in$ **Stm** y $s, s' \in$ **State**, $\langle S, s\rangle \to s'$ implica que $\langle \mathcal{CS}[\![S]\!], \varepsilon, s\rangle \rhd^* \langle \varepsilon, \varepsilon, s'\rangle$.*

En este caso, la demostración se realiza por inducción sobre la forma del árbol de derivación (recordemos que trabajamos con semántica natural).

Lema 4.11. *Dados $S \in$ **Stm** y $s, s' \in$ **State**, $\langle \mathcal{CS}[\![S]\!], \varepsilon, s\rangle \rhd^k \langle \varepsilon, e, s'\rangle$ implica que $\langle S, s\rangle \to s'$ y $e = \varepsilon$. Es decir, si la ejecución de un código para S desde el almacén s termina, entonces la semántica natural de S desde s terminará en un estado igual que el almacén de la configuración terminal correspondiente.*

Aquí se emplea inducción sobre la longitud de las secuencias de computación (derivación) de la máquina abstracta. Se podría consultar el Teorema 2.27 para tomar una idea del procedimiento general. □

Notemos que el teorema anterior implica dos propiedades. Por un lado, tenemos que, si la ejecución de una sentencia S termina en uno de los dos sistemas de transiciones, entonces también termina en el otro, y los resultados coinciden. Por otro lado, también se sigue que, si la ejecución de S en uno de estos sistemas entra en bucle, ocurrirá lo mismo en el otro.

4.4. Bisimulación

Del teorema previo, junto con el teorema de equivalencia, podemos concluir que $\mathcal{S}_{\text{sos}}[[S]] = \mathcal{S}_{\text{am}}[[S]]$. En cambio, podemos dar un procedemiento mucho más directo para demostrar esto y, de hecho, un resultado que emplee la semántica operacional estructural de While será más natural, dado que, como ya dijimos, la semántica de la máquina abstracta también es de paso corto.

De este modo, podríamos haber definido una relación de *bisimulación* entre las configuraciones de While y la máquina abstracta como:

$$\langle S, s\rangle \approx \langle \mathcal{CS}[\![S]\!], \varepsilon, s\rangle$$
$$s \approx \langle \varepsilon, \varepsilon, s\rangle$$

Notemos que ésta fue la motivación informal que dimos en su momento para comprender el funcionamiento de la máquina abstracta. Lo siguiente consiste en demostrar que la bisimulación enlaza caminos paralelos, es decir, siempre que una configuración de la semántica operacional estructural cambie en un paso, existe una secuencia de pasos en la semántica de la AM que hará un cambio similar en la configuración correspondiente. Más concretamente:

Lema 4.12. *Si $\gamma_{\text{sos}} \approx \gamma_{\text{am}}$ y $\gamma_{\text{sos}} \Rightarrow \delta_{\text{sos}}$, entonces existe una configuración δ_{am} tal que $\gamma_{\text{am}} \rhd^+ \delta_{\text{am}} \approx \delta_{\text{sos}}$.*
De hecho, si $\langle S, s\rangle \Rightarrow^* s'$, entonces $\langle \mathcal{CS}[\![S]\!], \varepsilon, s\rangle \rhd^* \langle \varepsilon, \varepsilon, s'\rangle$.

Recíprocamente, hay que demostrar que, si la máquina abstracta realiza una serie de movimientos con la pila vacía (es decir, cambia de una configuración a otra y ambas tienen la pila vacía, esto es, son de la forma $\langle c, \varepsilon, s\rangle$) entonces se pueden realizar otros movimientos paralelos en la semántica operacional estructural. Notemos que en la semántica operacional las variables se evalúan en un paso mientras que, en la máquina abstracta, esto puede llevar más de uno. Tenemos entonces:

Lema 4.13. *Sean $\gamma_{\text{sos}} \approx \gamma^1_{\text{am}} \rhd \cdots \rhd \gamma^k_{\text{am}}$, con $k > 1$ y de modo que solo γ^1_{am} y γ^k_{am} tienen la pila vacía. Entonces, existe una configuración δ_{sos} tal que $\gamma_{\text{sos}} \Rightarrow \delta_{\text{sos}} \approx \gamma^k_{\text{am}}$.*
De hecho, si $\langle \mathcal{CS}[\![S]\!], \varepsilon, s\rangle \rhd^* \langle \varepsilon, \varepsilon, s'\rangle$, entonces $\langle S, s\rangle \Rightarrow^* s'$.

Por tanto, obtenemos:

Teorema 4.14. *Dado $S \in \mathbf{Stm}$, $\mathcal{S}_{\text{sos}}[[S]] = \mathcal{S}_{\text{am}}[[S]]$.*

En el fondo, hemos podido demostrar este resultado de este modo por el siguiente motivo: es posible encontrar configuraciones en secuencias de derivaciones paralelas que se corresponden entre ellos (esta es la definición de bisimulación). Esta idea no se aplica, naturalmente, a la semántica operacional natural: deja de haber pasos intermedios en una de las dos derivaciones paralelas.

En general, la *demostración por bisimulación* (para la corrección) consiste en dos pasos:

(i) Asociamos pasos individuales en la semántica operacional estructural a procedimientos de la máquina abstracta y después extendemos esta asociación a secuencias de pasos.

(ii) Demostramos que secuencias especiales de la máquina abstracta pueden ser asociadas con secuencias de la semántica correspondiente y después demostramos a secuencias generales.

Ejemplo 4.15. Supongamos que, en el sistema $\text{While}_{\text{sos}}$, cambiamos la regla

$$[\text{while}_{\text{sos}}]\ \frac{}{\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \Rightarrow \langle \texttt{if}\ b\ \texttt{then}\ (S; \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else skip}, s\rangle}$$

Por las dos reglas:

$$[\text{while}'^{\text{tt}}_{\text{sos}}]\ \frac{}{\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \Rightarrow' \langle S; \texttt{while}\ b\ \texttt{do}\ S, s\rangle}\ \text{si } \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

$$[\text{while}'^{\text{ff}}_{\text{sos}}]\ \frac{}{\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \Rightarrow' s}\ \text{si } \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

Podríamos preguntarnos si, una vez que tenemos la función semántica modificada, $\mathcal{S}'_{\text{sos}}$, se verifica que, para cada $S \in \mathbf{Stm}$,

$$\mathcal{S}_{\text{sos}}[[S]] = \mathcal{S}'_{\text{sos}}[[S]].$$

Podríamos aplicar el método de demostración que acabamos de ver, es decir, podemos asociar $\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle$ y s en cada sistema y, por otro lado, tendríamos que $\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \Rightarrow^* \langle S; \texttt{while}\ b\ \texttt{do}\ S, s\rangle$ mientras que $\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \Rightarrow' \langle S; \texttt{while}\ b\ \texttt{do}\ S, s\rangle$, si si $\mathcal{B}[\![b]\!]s = \mathbf{tt}$, y $\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \Rightarrow^* s$ mientras que $\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \Rightarrow' s$. Con esto podemos convencernos de la afirmación anterior.

Ejemplo 4.16. Hemos realizado una modificación de la máquina abstracta de tal forma que ahora el código es fijo y disponemos de un contador de programa pc que indica la instrucción en ejecución (tal y como aparece en la máquina $\mathbf{AM}_2$). También se han reemplazado las instrucciones `BRANCH` y `LOOP` por las siguientes: $\texttt{JUMP} - z$ y $\texttt{JUMPFALSE} - z$. Pero se ha mantenido el estado. La configuración es una cuaterna definida de la siguiente forma:

$$\langle pc, cs, e, s\rangle \in \mathbb{N} \times \mathbf{Code} \times \mathbf{Stack} \times \mathbf{State}$$

Se presentan las nuevas instrucciones:

$$\begin{array}{rl}
\langle pc, cs, e, s\rangle \rhd \langle pc+1, cs, \mathcal{N}[\![n]\!] : e, s\rangle & \textbf{si } cs[pc] = \texttt{PUSH} - n \\
\langle pc, cs, e, s\rangle \rhd \langle pc+z, cs, \mathcal{N}[\![n]\!] : e, s\rangle & \textbf{si } cs[pc] = \texttt{JUMP} - z \\
\langle pc, cs, \texttt{ff} : e, s\rangle \rhd \langle pc+z, cs, \mathcal{N}[\![n]\!] : e, s\rangle & \textbf{si } cs[pc] = \texttt{JUMPFALSE} - z \\
\langle pc, cs, \texttt{tt} : e, s\rangle \rhd \langle pc+1, cs, \mathcal{N}[\![n]\!] : e, s\rangle & \textbf{si } cs[pc] = \texttt{JUMPFALSE} - z
\end{array}$$

Los nuevos esquemas de tradución son los mismos salvo `if`, `while`:

$$\mathcal{CS}[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] = \mathcal{CS}[\![b]\!] : \texttt{JUMPFALSE-}(z_1+1) : \mathcal{CS}[\![S_1]\!] : \texttt{JUMP-}z_2 : \mathcal{CS}[\![S_1]\!]$$

$$\text{donde la longitud de } \mathcal{CB}[\![S_1]\!], \mathcal{CB}[\![S_2]\!] = z_1, z_2$$

$$\mathcal{CS}[\![\texttt{while}\ b\ \texttt{do}\ S]\!] = \mathcal{CS}[\![b]\!] : \texttt{JUMPFALSE-}(z_1+1) : \mathcal{CS}[\![S]\!] : \texttt{JUMP-}(-(z_0+z_1+1)) : \mathcal{CS}[\![S_1]\!]$$

$$\text{donde la longitud de } \mathcal{CB}[\![b]\!], \mathcal{CB}[\![S]\!] = z_0, z_1$$

Demostremos la equivalencia de esta nueva máquina con la anterior

5 | Semántica denotacional

Hasta ahora, lo que nos ha interesado de los programas era su ejecución. En cambio, como ya dijimos en la introducción, lo que queremos estudiar ahora es el efecto de esta ejecución, es decir, la relación entre los estados iniciales y los finales asociados a ella. Para ello, lo que haremos es definir una función semántica para cada categoría sintáctica, en el sentido de que cada construcción sintáctica será interpretada mediante un objeto matemático, el 'efecto' de ejecutar tal construcción.

Este enfoque no es del todo desconocido para nosotros. Por ejemplo, las funciones semánticas $\mathcal{A}$ y $\mathcal{B}$ que ya hemos empleado son un ejemplo típico de definición denotacional: a cada expresión aritmética asociamos un objeto abstracto, más concretamente, una función de tipo $\mathbf{State} \to \mathbf{Z}$. Y lo mismo ocurre con las expresiones booleanas y las correspondientes funciones de $\mathbf{State} \to \mathbf{Bool}$. Recordemos que una característica importante de esta definiciones era la composicionalidad (véase la introducción). Por tanto, las funciones $\mathcal{S}_{\text{ns}}$ y $\mathcal{S}_{\text{sos}}$ no se corresponden con la idea de definición denotacional que hemos descrito.

5.1. Sistema para While

5.1.1. Función semántica

Las sentencias modifican el estado en el que se encuentran. Por tanto, en el enfoque denotacional, se define una aplicación que, dada una sentencia, devuelve una función (parcial) que transforma estados en estados:

$$\mathcal{S}_{\text{ds}} : \mathbf{Stm} \to (\mathbf{State} \hookrightarrow \mathbf{State})$$

Notemos que $\mathcal{S}_{\text{ds}}$ es parcial precisamente por la sentencia `while true do skip`. Veamos, caso por caso, qué función es exactamente la que asignamos.

Si reflexionamos en lo que implica la ejecución de $x := a$, parecerá natural que le corresponda una aplicación que modifica, en cada estado, el valor que toma x. Más precisamente, una definición de este tipo sería:

$$\mathcal{S}_{\text{ds}}[[x := a]] : s \mapsto s[x \mapsto \mathcal{A}[\![a]\!]s]$$

es decir, dado un estado s, lo convertiremos a un estado en el cual la variable x toma el valor $\mathcal{A}[\![a]\!]s$.

En el caso de `skip` queremos que no se ejecute ningún cambio en el estado. Lo reflejamos mediante la identidad $id : \mathbf{State} \to \mathbf{State}$, dada por $id(s) := s$[1]:

$$\mathcal{S}_{\text{ds}}[[\texttt{skip}]] := id$$

Para la composición de dos sentencias, $S_1; S_2$, simplemente queremos recibir el estado al que se llega tras ejecutar S_1 y después aplicar S_2. Esto lo podemos hacer mediante la composición:

$$\mathcal{S}_{\text{ds}}[[S_1; S_2]] := \mathcal{S}_{\text{ds}}[[S2]] \circ \mathcal{S}_{\text{ds}}[[S_1]]$$

[1] Se hace notar que podríamos dar definiciones de las aplicaciones recurriendo a imágenes de estados arbitrarios. En este caso no se ha utilizado la variable estado s, porque no la necesitamos para modificar el estado posterior (en el caso anterior necesitamos s para saber el valor de la expresión aritmética a). Se podría haber definido, de forma equivalente, $\mathcal{S}_{\text{ds}}[[\texttt{skip}]]s := s$, para todo $s \in \mathbf{State}$.

Es muy importante tener en cuenta que $\mathcal{S}_{\text{ds}}[[S_1]]$ se evalúa antes, pasando su resultado a $\mathcal{S}_{\text{ds}}[[S_2]]$.

La idea para el condicional es la siguiente:

$$\mathcal{S}_{\text{ds}}[[\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]]s := \begin{cases} \mathcal{S}_{\text{ds}}[[S1]]s, \text{ si } \mathcal{B}[\![b]\!]s = \mathbf{tt} \\ \mathcal{S}_{\text{ds}}[[S2]]s, \text{ si } \mathcal{B}[\![b]\!]s = \mathbf{ff} \end{cases}$$

donde sencillamente se ha utilizado el valor $\mathcal{B}[\![b]\!]s$ para separar en dos casos y, según el resultado, aplicar una sentencia u otra. Pero, como dijimos antes, suele ser más cómodo emplear notación funcional. Podemos definir entonces una aplicación auxiliar $\texttt{cond} : (\mathbf{State} \to \mathbf{Bool}) \times (\mathbf{State} \hookrightarrow \mathbf{State}) \times (\mathbf{State} \hookrightarrow \mathbf{State}) \to (\mathbf{State} \hookrightarrow \mathbf{State})$, bajo la idea de que el primer parámetro será la evaluación de la expresión booleana (en función del estado) y las otras dos serán las evaluaciones de las sentencias S_1 y S_2[2]. Para ello, según el valor que retorne el booleano, se devolverá el primer parámetro o el segundo, es decir:

$$\texttt{cond}(f_b, f_1, f_2)s := \begin{cases} f_1 s, \text{ si } f_b s = \mathbf{tt} \\ f_2 s, \text{ si } f_b s = \mathbf{ff} \end{cases}$$

Por tanto, la definición deseada es:

$$\mathcal{S}_{\text{ds}}[[\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]] := \texttt{cond}(\mathcal{B}[\![b]\!], \mathcal{S}_{\text{ds}}[[S_1]], \mathcal{S}_{\text{ds}}[[S_2]])$$

Esto es, si b es cierto en el estado actual, se toma la primera sentencia y, en caso contrario, la segunda.

Hemos visto que ha sido posible definir las expresiones básicas mediante aplicaciones relativamente sencillas. No ocurrirá lo mismo con la sentencia $\texttt{while } b \texttt{ do } S$. Podríamos intentar usar lo anterior, junto con la equivaencia semántica con la sentencia $\texttt{if } b \texttt{ then } (S; \texttt{while } b \texttt{ do } S) \texttt{ else skip}$, y así obtendríamos:

$$\mathcal{S}_{\text{ds}}[[\texttt{while } b \texttt{ do } S]] := \texttt{cond}(\mathcal{B}[\![b]\!], \mathcal{S}_{\text{ds}}[[\texttt{while } b \texttt{ do } S]] \circ \mathcal{S}_{\text{ds}}[[S]], id)$$

Pero, ¿acaso es esto una definición aceptable? Entre otras cosas, no es composicional, y por tanto no podemos aceptarla. La igualdad anterior sí que significa, por otro lado, algo, a saber: que $\mathcal{S}_{\text{ds}}[[\texttt{while } b \texttt{ do } S]]$ es *punto fijo* de la función F dada por $F(g) := \texttt{cond}(\mathcal{B}[\![b]\!], g \circ \mathcal{S}_{\text{ds}}[[S]], id)$. De hecho, F sí que admite una definición composicional porque en lo anterior solo aparecen construcciones estructuralmente más sencillas que $\texttt{while } b \texttt{ do } S$. Por tanto, la siguiente definición es más satisfactoria:

$$\mathcal{S}_{\text{ds}}[[\texttt{while } b \texttt{ do } S]] := \texttt{fix}(F)$$

donde $\texttt{fix} : ((\mathbf{State} \hookrightarrow \mathbf{State}) \to (\mathbf{State} \hookrightarrow \mathbf{State})) \to (\mathbf{State} \hookrightarrow \mathbf{State})$ nos lleva una aplicación (de apliaciones) a su punto fijo.

Ejemplo 5.1. Sea $\texttt{while } \neg(\texttt{x} = 0) \texttt{ do skip}$. El funcional asociado es, en este caso:

$$G(g)s := \begin{cases} gs, \text{ si } s\texttt{x} \neq 0 \\ s, \text{ en otro caso} \end{cases}$$

¿Hay algún punto fijo para G? Probemos con

$$g_1 s := \begin{cases} \texttt{INDEFINIDO}, \text{ si } s\texttt{x} \neq 0 \\ s, \text{ en otro caso} \end{cases}$$

En efecto,

$$G(g_1)s = \begin{cases} g_1 s, \text{ si } s\texttt{x} \neq 0 \\ s, \text{ en otro caso} \end{cases} = \begin{cases} \texttt{INDEFINIDO}, \text{ si } s\texttt{x} \neq 0 \\ s, \text{ en otro caso} \end{cases} = g_1 s$$

Luego $g_1 = \texttt{fix}(G)$. Por otro lado, $g_2 \equiv \texttt{INDEFINIDO}$ no puede ser punto fijo de G porque, si $s \in \mathbf{State}$ verifica $sx = 0$, entonces $G(g_2)s = s \neq \texttt{INDEFINIDO} = g_2 s$.

[2]Hemos empleado el producto de tipos para mayor simplicidad. También podríamos haber escrito $\texttt{cond} : (\mathbf{State} \to \mathbf{Bool}) \to (\mathbf{State} \hookrightarrow \mathbf{State}) \to (\mathbf{State} \hookrightarrow \mathbf{State}) \to (\mathbf{State} \hookrightarrow \mathbf{State})$ y por tanto prescindir de paréntesis. Puede que, en un aspecto puramente formal, esta diferencia no sea relevante. En cambio, sí que es importante señalarla desde el punto de vista de los tipos de datos que emplea cond. También podríamos haber sustituido el tipo de funciones $\mathbf{State} \to \mathbf{Bool}$ por el tipo simple **Bool** pero, de nuevo, recordemos que es preferible trabajar con funciones.

Entonces, de momento, tenemos las siguientes reglas:

Sistema (Semántica denotacional para While)**.**

$$\begin{aligned}
\mathcal{S}_{\mathrm{ds}}[[x := a]]s &:= s[x \mapsto \mathcal{A}[\![a]\!]s] \\
\mathcal{S}_{\mathrm{ds}}[[\texttt{skip}]] &:= id \\
\mathcal{S}_{\mathrm{ds}}[[S_1; S_2]] &:= \mathcal{S}_{\mathrm{ds}}[[S2]] \circ \mathcal{S}_{\mathrm{ds}}[[S_1]] \\
\mathcal{S}_{\mathrm{ds}}[[\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]]s &= \texttt{cond}(\mathcal{B}[\![b]\!], \mathcal{S}_{\mathrm{ds}}[[S_1]], \mathcal{S}_{\mathrm{ds}}[[S_2]]) \\
\mathcal{S}_{\mathrm{ds}}[[\texttt{while } b \texttt{ do } S]] &:= \texttt{fix}(F)
\end{aligned}$$

donde $F(g) := \texttt{cond}(\mathcal{B}[\![b]\!], g \circ \mathcal{S}_{\mathrm{ds}}[[S]], id)$.

Notemos que podríamos haber intentado definir

$$\mathcal{S}_{ds}[[\texttt{while } b \texttt{ do } S]]s := \mathcal{S}_{ds}^{\delta(b,S)s}s$$

con

$$\delta(b,S)s := \begin{cases} 0, \text{ si } \mathcal{B}[\![b]\!]s = \mathbf{ff} \\ 1 + \delta(b,S)s', \text{ en otro caso} \end{cases}$$

Sería una definición composicional pero, en cambio, la aplicación $\delta(b,S)$ no está bien definida como función recursiva, porque podría darse el caso de que no devolviera ningún valor bien definido.

5.1.2. Requisitos de punto fijo

Ahora bien, si examinamos la definición que hemos dado para $\mathcal{S}_{\mathrm{ds}}[[\texttt{while } b \texttt{ do } S]]$, veremos que volvemos a tener problemas. Esta vez el obstáculo se encuentra en que hemos supuesto que cada aplicación asociada a una sentencia de la forma $\texttt{while } b \texttt{ do } S$ tiene *un* punto fijo. La realidad es bien distinta: puede haber varios puntos fijos o incluso ninguno, como ilustran los siguientes ejemplos.

Ejemplo 5.2. Cualquier elemento $f \in \mathbf{State} \hookrightarrow \mathbf{State}$, tal que $fs = s$ si $sx = 0$, es punto fijo de la aplicación G del anterior ejemplo.

Ejemplo 5.3. Sean $g_1, g_2 \in \mathbf{State} \hookrightarrow \mathbf{State}$. La aplicación

$$H(g) := \begin{cases} g_1, \text{ si } g = g_2 \\ g_2, \text{ en otro caso} \end{cases}$$

no admite ningún punto fijo si $g_1 \neq g_2$ porque, si $H(g) = g$, entonces, o bien $g \neq g_2$ y $H(g) = g_2$, o bien $g = g_2$ y $H(g) = g_2 \neq g_1 = g$.

Por tanto, como no podemos admitir que `fix` devuelva un conjunto de candidatos a puntos fijos, parece irremediable dar una serie de requerimientos para seleccionar *el* que nos interesa. La tarea, entonces, es doble, porque debemos probar que al menos y a lo sumo uno de los posibles puntos fijos verifica tales condiciones. ¿Qué condiciones dar? No entraremos mucho en detalle aquí, pero conviene tener una idea general de por qué exigimos determinadas restricciones. Podemos comenzar estudiando las diferentes configuraciones que se obtienen al ejecutar $\texttt{while } b \texttt{ do } S$ desde un estado s_0. Sea F la aplicación asociada. Tenemos los siguientes casos:

(a) La ejecución termina. Por ejemplo, con $\texttt{while } 0 \leq \texttt{x do x} := \texttt{x} - 1$ desde cualquier estado en el que $\texttt{x} \geq 0$.

(b) La ejecución entra en un bucle local, es decir, hay alguna subexpresión en S que entra en bucle. Por ejemplo, en $\texttt{while } 0 \leq \texttt{x do (if x} = 0 \texttt{ then (while true do skip) else x} := \texttt{x} - 1)$ desde cualquier estado con $\texttt{x} \geq 0$.

(c) La ejecución entra en un bucle global, es decir, se entra en bucle al ejecutar la expresión completa. Por ejemplo, con `while` $\neg(\mathtt{x} = 0)$ `do skip` desde cualquier estado con $\mathtt{x} \neq 0$.

Más concretamente,

(a) Hay una secuencia de estados $s_1, \ldots, s_n$ tales que $\mathcal{B}[\![b]\!]$ toma el valor **ff** únicamente en s_n y con $\mathcal{S}_{\mathrm{ds}}[[S]]s_i = s_{i+1}$ para $i < n$. Si $F(g) = g$, entonces

$$gs_i = F(g)s_i = \mathtt{cond}(\mathcal{B}[\![b]\!], g \circ \mathcal{S}_{\mathrm{ds}}[[S]], id)$$

y entonces es claro que $gs_i = s_{i+1}$, si $i < n$, y $gs_i = s_i$ si $i = n$. Pero entonces $gs_0 = s_n$.

(b) Hay una secuencia de estados $s_1, \ldots, s_n$ tales que $\mathcal{B}[\![b]\!]$ toma el valor **tt** para cada $i \leq n$ y tales que $\mathcal{S}_{\mathrm{ds}}[[S]]s_i = s_{i+1}$, si $i < n$, y $\mathcal{S}_{\mathrm{ds}}[[S]]s_i = \mathtt{INDEFINIDO}$, si $i = n$. Por un razonamiento análogo al anterior, podemos determinar que todo $g = F(g)$ verifica que $gs_0 = \mathtt{INDEFINIDO}$.

(c) Existe una secuencia infinita de estados $s_1, s_2, \ldots$ tales que $\mathcal{B}[\![b]\!]s_i = \mathbf{tt}$ y $\mathcal{S}_{\mathrm{ds}}[[S]]s_i = s_{i+1}$, para cada $i = 1, 2, \ldots$. Entonces, es fácil ver que, si $F(g) = g$,

$$gs_i = gs_{i+1}, \text{ para todo } i \in \mathbb{N}$$

Pero entonces $gs_0 = gs_i$, para cada $i \in \mathbb{N}$. Por tanto, siguiendo este método no podemos determinar el valor de gs_0.

Precisamente, esta anomalía del caso (c) es lo que nos indica que los posibles puntos fijos de F pueden diferir entre sí. Para mayor claridad, centrémonos en el siguiente ejemplo:

Ejemplo 5.4. Consideremos `while` $\neg(\mathtt{x} = 0)$ `do skip`, ejecutado desde un estado s_0 con $\mathtt{x} \neq 0$. En este caso,

$$F(g)s := \begin{cases} gs, \text{ si } s\mathtt{x} \neq 0 \\ s, \text{ en otro caso} \end{cases}$$

Pero ya vimos que cualquier $g \in \mathbf{State} \hookrightarrow \mathbf{State}$ con $gs = s$ si $s\mathtt{x} = 0$ era punto fijo de F. Por otro lado, está claro que la intuición nos dice que sería deseable tener

$$\mathcal{S}_{\mathrm{ds}}[[\mathtt{while}\ \neg(\mathtt{x} = 0)\ \mathtt{do\ skip}]]s_0 := \begin{cases} \mathtt{INDEFINIDO}, \text{ si } s_0\mathtt{x} = 0 \\ s_0, \text{ en otro caso} \end{cases}$$

Pero entonces, si tomamos ahora

$$hs := \begin{cases} \mathtt{INDEFINIDO}, \text{ si } s\mathtt{x} = 0 \\ s, \text{ en otro caso} \end{cases}$$

es claro que $F(h) = h$ y, *además*, que $hs = s'$ implica $h's = s'$, para cualquier otro punto fijo h' de F, pero no al contrario.

Este ejemplo muestra que, en resumen, el candidato a $\mathtt{fix}(F)$ debe ser cierto $h \in \mathbf{State} \hookrightarrow \mathbf{State}$ tal que:

(1) $F(h) = h$.

(2) Sean $s, s' \in \mathbf{State}$. Si $g \in \mathbf{State} \hookrightarrow \mathbf{State}$ verifica (1), entonces

$$hs = s' \text{ implica que } gs = s'$$

Nótese que, si $hs = \mathtt{INDEFINIDO}$ entonces no se verifica (2).

Ejemplo 5.5 (Ejercicio 5.2)**.** El funcional F asociado con la sentencia

$$\texttt{while } \neg(x = 0) \texttt{ do } x := x - 1$$

es, por definición $F(g) = \texttt{cond}(\mathcal{B}[\![\neg(x = 0)]\!], g \circ \mathcal{S}_{\text{ds}}[[x := x - 1]], \texttt{id})$, que aplicado a un estado s puede verse de la forma

$$\begin{aligned} F\ g(s) &= \begin{cases} g \circ \mathcal{S}_{\text{ds}}[[x := x - 1]], \texttt{id})(s) & \mathcal{B}[\![\neg(x = 0)]\!]s = \mathbf{tt} \\ s & c.c \end{cases} \\ &= \begin{cases} g(s[x \mapsto (s\ x) - 1]) & s\ x \neq 0 \\ s & s\ x = 0 \end{cases} \end{aligned}$$

Ahora vamos a ver varios ejemplos de función g y comprobaremos si son puntos fijos por F:

1. La aplicación
$$g_1\ s := \texttt{INDEFINIDO}, \quad \forall s \in \mathbf{State}$$
no es un punto fijo. Tomemos por ejemplo un estado $s \in \mathbf{State}$ tal que $s\ x = 1$, entonces
$$\begin{aligned} s' &:= (F\ g_1)\ s \\ &= \begin{cases} g_1\ s[x \mapsto 0] & x \neq 0 \\ s & c.c \end{cases} \\ &= g_1\ s[x \mapsto 0] \end{aligned}$$
luego $s'\ x = \texttt{INDEFINIDO} \neq 1 = s\ x$, por lo que g no es un punto fijo.

2. La aplicación
$$g_2\ s = \begin{cases} s[x \mapsto 0] & s\ x \geq 0 \\ \texttt{INDEFINIDO} & s\ x < 0 \end{cases}$$
sí es un punto fijo. Sea $s \in \mathbf{State}$, supongamos $s\ x > 0$:
$$\begin{aligned} s' &:= (F\ g_2)\ s \\ &= \begin{cases} g_2\ s[x \mapsto s\ x - 1] & s\ x \neq 0 \\ s & s\ x = 0 \end{cases} \\ &= g_2\ s[x \mapsto s\ x - 1] \\ &= s[[x \mapsto s\ x - 1]][x \mapsto 0] \\ &= s[x \mapsto 0] = g_2\ s \end{aligned}$$
Para el caso $s\ x = 0$ sucede que
$$(F\ g_2)\ s = s = s[x \mapsto 0] = g_2\ s$$
Y finalmente si $x < 0$, informalmente el bucle sería infinito:
$$(F\ g_2)\ s = \texttt{INDEFINIDO} = g_2\ s$$
Por lo que se deduce que $F\ g_2 = g_2$, luego es un punto fijo.

3. Consideremos la aplicación
$$g_3\ s = \begin{cases} s[x \mapsto 0] & s\ x \geq 0 \\ s & s\ x < 0 \end{cases}$$
Tomemos $s\ x = -1$ entonces
$$\begin{aligned} (F\ g_3)\ s &= g_3\ s[x \mapsto -2] \\ &= g_3\ s[x \mapsto -2] \neq s = g_3\ s \end{aligned}$$
Por lo que $F\ g_3 \neq g_3$ y entonces g_3 no es punto fijo.

4. Definimos ahora $g_4\ s = s[x \mapsto 0]$ para todo $s \in \mathbf{State}$. Esta función es punto fijo:

$$\begin{array}{rcl}(F\ g_4)\ s & = & \left\{\begin{array}{ll} g_4\ s[x \mapsto s\ x - 1] & s\ x \neq 0 \\ s & s\ x = 0 \end{array}\right. \\ & = & \left\{\begin{array}{ll} s[x \mapsto 0] & x \neq 0 \\ s & s\ x = 0 \end{array}\right. \\ & = & s[x \mapsto 0] = g_4\ s \end{array}$$

5. La aplicación identidad $g_5\ s = s$ no es un punto fijo. Sea $s \in \mathbf{State}$ tal que $s\ x = 1$ entonces

$$(F\ g_5)\ s \quad = \quad g_5\ s[x \mapsto 0] \neq s = g_5\ s$$

de lo que se deduce que $F\ g_5 \neq g_5$.

Ejemplo 5.6 (Ejercicio 5.3)**.** Consideramos ahora la sentencia

$$\texttt{while}\ \neg(x = 1)\ \texttt{do}\ (y := y * x; x := x - 1)$$

Tiene por funcional F:

$$\begin{array}{rcl}(F\ g)s & = & \left\{\begin{array}{ll} g \circ \mathcal{S}_{\mathrm{ds}}[[y := y * x; x := x - 1]], \mathtt{id})(s) & \mathcal{B}[\![\neg(x = 1)]\!]s = \mathbf{tt} \\ s & c.c \end{array}\right. \\ & = & \left\{\begin{array}{ll} g\ s[y \mapsto (s\ y) * (s\ x)][x \mapsto (s\ x) - 1] & s\ x \neq 1 \\ s & s\ x = 1 \end{array}\right. \end{array}$$

Este bucle hace un cálculo de $y \cdot x!$, por lo que es previsible que el punto fijo represente este hecho.

Un ejemplo de punto fijo es la aplicación

$$f_1\ s = \left\{\begin{array}{ll} s[y \mapsto (s\ y) * (s\ x)!][x \mapsto 1] & si\ s\ x \geq 1 \\ \texttt{INDEFINIDO} & \text{caso contrario} \end{array}\right.$$

pues

$$\begin{array}{rcl}(F\ f_1)\ s & = & \left\{\begin{array}{ll} f_1\ s[y \mapsto (s\ y) * (s\ x)][x \mapsto (s\ x) - 1] & s\ x \neq 1 \\ s & s\ x = 1 \end{array}\right. \\ & = & \left\{\begin{array}{ll} f_1\ s'[y \mapsto (s'\ y) * (s'\ x)][x \mapsto 1] & s\ x \neq 1 \wedge s' \geq 1 \\ \texttt{INDEFINIDO} & s\ x \neq 1 \wedge s' < 1 \\ s & s\ x = 1 \end{array}\right. \\ & = & \left\{\begin{array}{ll} f_1\ s[y \mapsto (s\ y) * (s\ x)!][x \mapsto 1] & s\ x \geq 2 \\ \texttt{INDEFINIDO} & s\ x < 1 \\ s & s\ x = 1 \end{array}\right. \\ & = & f_1\ s \end{array}$$

donde $s' := s[y \mapsto (s\ y) * (s\ x)][x \mapsto (s\ x) - 1]$- También es un punto fijo

$$f_2\ s = s[y \mapsto (s\ y) * (s\ x)!][x \mapsto 1]$$

pues es igual a f_1 salvo en los puntos en los que la primera función está indefinida.

5.2. Conjuntos parcialmente ordenados

Antes de formalizar todas las ideas anteriores, conviene repasar algunos resultados básicos sobre conjuntos parcialmente ordenados. Posteriormente, haremos referencia a éstos, aplicados a un caso especial que describiremos más adelante.

Definición 5.7. Un par $(D, \sqsubseteq)$ formado por un conjunto D y una relación $\sqsubseteq \subseteq D^2$ se dice *conjunto parcialmente ordenado* si $\sqsubseteq$ es reflexiva, antisimétrica y transitiva.

Recordemos también que un elemento $d \in D$ se dice $\sqsubseteq$-*mínimo* si, para cada $d' \in D$, $d \sqsubseteq d'$.

Lema 5.8. *Si un conjunto parcialmente ordenado $(D, \sqsubseteq)$ tiene un elemento mínimo, entonces es único.*

Demostración. En otro caso, si hubiera dos mínimos, bastaría aplicar la definición de elemento mínimo y la antisimetría del orden parcial para comprobar que son iguales. □

Por tanto, si existe, denotaremos a tal elemento mínimo como $\bot$.

Definición 5.9. Dados $(D, \sqsubseteq)$ parcialmente ordenado y $Y \subset D$, se dice que $d \in D$ es una *cota superior* de Y si, para cada $d' \in Y$, $d' \sqsubseteq d$.

Llamaremos *supremo* de $Y \subseteq D$ a la menor cota superior de Y (respecto de $\sqsubseteq$).

Lema 5.10. *Si $Y \subseteq D$ tiene un supremo respecto de un orden parcial, entonces es único.*

Demostración. Basta repetir la demostración del lema anterior, esta vez para cotas superiores. □

Si existe tal supremo en Y, lo denotamos por $\bigsqcup Y$. Pasemos ahora a un concepto que será relevante más adelante. Recordemos que un conjunto A se dice *totalmente ordenado* por una relación de orden $<$ si, dados arbitrarios $x, y \in A$, $x < y$ o $y < x$.

Definición 5.11. Un subconjunto C de un conjunto parcialmente ordenado $(D, \sqsubseteq)$ se llama *cadena* si es totalmente ordenado.

Definición 5.12. Un conjunto parcialmente ordenado $(D, \sqsubseteq)$ se llama *completo* si, dada una cadena cualquiera C, existe $\bigsqcup C$. Si lo mismo ocurre para todos los subconjuntos de D, lo denominamos *retículo completo*.

Proposición 5.13. *Sea $(D, \sqsubseteq)$ es un conjunto parcialmente ordenado y completo. Entonces tiene un elemento mínimo $\bot := \bigsqcup \emptyset$.*

Demostración. Claramente, $\emptyset$ es cadena (verifica la afirmación de forma vacía) y, como D es completo por hipótesis, existe $\bigsqcup \emptyset$. Ahora bien, dado $d \in D$, por la definición de $\bigsqcup \emptyset$ es directo que se verifica $\bigsqcup \emptyset \sqsubseteq d$, luego obtenemos el resultado. □

Definición 5.14. Sean $(D, \leq)$ y $(E, \preccurlyeq)$ conjuntos ordenados. Una aplicación $f : D \to E$ se dice *monótona* si $d \leq d'$ implica que $f(d) \preccurlyeq f(d')$, para cualesquiera $d, d' \in D$.

Es directo comprobar que la composición de aplicaciones monótonas es monótona.

Lema 5.15. *Sean $(D, \sqsubseteq), (E, \preccurlyeq)$ conjuntos parcialmente ordenados completos y $f : D \to E$ monótona. Entonces, si $C \subseteq D$ es cadena, $f(C) \subseteq E$ también lo es. Además*[3]*,*

$$\bigsqcup f(C) \preccurlyeq f\left(\bigsqcup C\right).$$

Demostración. Si $C = \emptyset$, entonces $f(\emptyset) = \emptyset$ y, como $\bot_E \preccurlyeq f(\bot_D)$, tenemos el resultado. Por tanto, supongamos que $C \neq \emptyset$. Sean $e_1, e_2 \in f(C)$. Entonces existen $d_1, d_2 \in C$ con $f(d_1) = e_1$ y $f(d_2) = e_2$. Por ser C cadena, $d_1 \sqsubseteq d_2$ o $d_2 \sqsubseteq d_1$ y, por monotonía de f, $e_1 \preccurlyeq e_2$ o $e_2 \preccurlyeq e_1$, luego $f(C)$ es efectivamente cadena.

Para lo segundo, sea $d \in C$. Entonces, por definición, $d \sqsubseteq \bigsqcup C$ y, por monotonía, $f(d) \preccurlyeq f(\bigsqcup C)$. Es decir, $f(\bigsqcup C)$ es cota superior de $f(C)$. Pero entonces, por definición, $\bigsqcup f(C) \preccurlyeq f(\bigsqcup C)$. □

Aunque, en general, no es cierto que las aplicaciones monótonas preserven supremos, nos centraremos en ellas a continuación:

[3]Nótese que, en la expresión que aparece en el teorema, nos referimos, primero a un $\preccurlyeq$-supremo, y después a un $\sqsubseteq$-supremo. Confiamos en el lector para que esté atento al uso en cada caso.

Definición 5.16. Sean $(D, \sqsubseteq), (E, \preccurlyeq)$ conjuntos parcialmente ordenados completos y $f : D \to E$ monótona. Decimos que f es *continua* si $\bigsqcup f(C) = f(\bigsqcup C)$ para cada cadena $C \neq \emptyset$. Si, además, $f(\bot_D) = \bot_E$, decimos que f es *estricta*.

Es fácil comprobar que, si $f : D \to E$ es continua, entonces es monótona. También es fácil ver que composición de funciones continuas es continua, y que lo mismo ocurre para las funciones estrictas.

Llegamos de este modo al concepto que ha motivado toda la sección:

Teorema 5.17. *Sea $(D, \sqsubseteq)$ conjunto parcialmente ordenado completo. Sea $f : D \to D$ continua. Entonces,*

$$\mathit{fix}(f) := \bigsqcup \{f^m(\bot) \mid m \in \mathbb{N}\}$$

define un elemento de D y es precisamente el menor punto fijo de f[4]*.*

Demostración. Veamos que $\mathtt{fix}(f)$ está bien definido. Primero, $id(\bot) = \bot$ y, claramente, $\bot \sqsubseteq d$ para cada $d \in D$. Supongamos que $f^n(\bot) \sqsubseteq f^n(d)$ para cada $d \in D$ y cierto $n > 0$. Entonces, por monotonía, $f^{n+1}(\bot) = f(f^n \bot) \sqsubseteq f(f^n(d)) = f^{n+1}(d)$. Entonces, es fácil comprobar que $f^n(\bot) \sqsubseteq f^m(\bot)$ si $n \leq m$. Pero entonces, $C := \{f^n(\bot) \mid n \in \mathbb{N}\}$ es cadena no vacía en D y, por completitud, existe $\mathtt{fix}(f)$.

Veamos que $f(\mathtt{fix}(f)) = \mathtt{fix}(f)$:

$$f(\mathtt{fix}(f)) = f\left(\bigsqcup \{f^n(\bot) \mid n \geq 0\}\right) = \bigsqcup \{f^{n+1}(\bot) \mid n \geq 0\} = \bigsqcup \{f^n(\bot) \mid n \geq 1\}$$

porque f es continua. Ahora, usando que $\bigsqcup(C \cup \{\bot\}) = \bigsqcup C$, para cada cadena C, y que $\bot = f^0(\bot)$,

$$\bigsqcup \{f^n(\bot) \mid n \geq 1\} = \bigsqcup (\{f^n(\bot) \mid n \geq 1\} \cup \{\bot\}) = \bigsqcup \{f^n(\bot) \mid n \geq 0\} = \mathtt{fix}(f).$$

Finalmente, veamos que es el menor punto fijo. Sea d otro punto fijo de f. Como $\bot \sqsubseteq d$, por monotonía se tiene que $f^n(\bot) \sqsubseteq f^n(d)$, para cada $n \geq 0$. Como d es punto fijo, $f^n(\bot) \sqsubseteq d$, para cada $n \geq 0$, es decir, d es cota superior de la cadena $\{f^n(d) \mid n \geq 0\}$. Pero, por definición, $\mathtt{fix}(f) \sqsubseteq d$. □

Proposición 5.18. *Sea $f : D \to D$ continua con $(D, \sqsubseteq)$ parcialmente ordenado y completo. Sea $d \in D$ tal que $f(d) \sqsubseteq d$. Entonces, $\mathit{fix}(f) \sqsubseteq d$.*

Demostración. Para empezar, por monotonía, deducimos que $f^n(d) \sqsubseteq \cdots \sqsubseteq f(d) \sqsubseteq d$ y que $f^n(\bot) \sqsubseteq f^n(d)$, para cada $n \in N$. Pero, entonces,

$$\mathtt{fix}(f) := \bigsqcup \{f^n(\bot) \mid n \geq 0\} \sqsubseteq \bigsqcup \{f^n(d) \mid n \geq 0\} \sqsubseteq d.$$

□

5.3. Estudio del punto fijo

5.3.1. Descripción

Regresemos entonces a nuestro problema original. Para empezar, vamos a definir un orden parcial en **State** $\hookrightarrow$ **State**, recordando los requisitos de punto fijo que ya expusimos anteriormente:

Definición 5.19. Dadas $g_1, g_2 \in$ **State** $\hookrightarrow$ **State**, decimos que g_1 *está menos definida que* g_2 si, para arbitrarios $s, s' \in$ **State**, $g_1 s = s'$ implica que $g_2 s = s'$. Simbolizaremos esta relación como $\sqsubseteq$.

Evidentemente, la anterior definición se puede reformular mediante pares, de modo que dice que $\mathrm{G}(g_1) \subseteq G(g_2)$, donde $G(f)$ denota el *grafo* de f, es decir, el conjunto de los pares de los que se compone f.

[4] Aquí empleamos la notación $f^0 := id$, $f^{n+1} := f^n \circ f$.

Lema 5.20. (**State** $\hookrightarrow$ **State**, $\sqsubseteq$) *es conjunto parcialmente ordenado con elemento mínimo dado por* $\bot \equiv$ `INDEFINIDO`.

Demostración. Las propiedades reflexiva y transitiva son directas. Veamos la antisimétrica. Sean $g_1 \sqsubseteq g_2 \sqsubseteq g_1$. Si $g_1 s = s'$, entonces evidentemente se sigue que $g_2 s = s'$. Si $g_1 s =$ `INDEFINIDO`, entonces $g_2 s =$ `INDEFINIDO` porque, en otro caso, digamos que $g_2 s = s'$, tendríamos que $g_1 s = s'$, lo que es imposible.

Veamos lo segundo. Evidentemente, $\bot \in$ **State** $\hookrightarrow$ **State**. Además, $\bot s = s'$ implica de forma vacía que $gs = s'$, para cada $g \in$ **State** $\hookrightarrow$ **State**. □

Ejemplo 5.21. Sean $g_1, g_2, g_3, g_4 \in$ **State** $\hookrightarrow$ **State**:

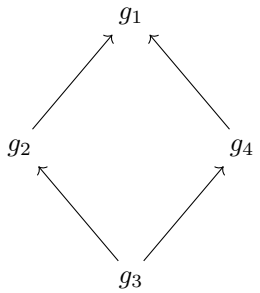
$$g_1 := id$$

$$g_2 s := \begin{cases} s, \text{ si } sx \geq 0 \\ \texttt{INDEFINIDO}, \text{ en otro caso} \end{cases}$$

$$g_3 s := \begin{cases} s, \text{ si } sx = 0 \\ \texttt{INDEFINIDO}, \text{ en otro caso} \end{cases}$$

$$g_4 s := \begin{cases} s, \text{ si } sx \leq 0 \\ \texttt{INDEFINIDO}, \text{ en otro caso} \end{cases}$$

Entonces es claro que tenemos el siguiente diagrama de Hasse:



Donde cada arista indica que se da la relación de orden entre los nodos (en el sentido especificado). No aparecen las aristas obtenidas mediante transitividad (dicho de otro modo, la transitividad viene dada por llegar de un nodo a otro mediante un camino válido).

Entonces, las propiedades que debía cumplir un punto fijo para un funcional F quedan como:

(a) $F(\texttt{fix}(F)) = \texttt{fix}(F)$.

(b) Si g es punto fijo de F, $\texttt{fix}(F) \sqsubseteq g$.

Además, es claro que, si F tiene al menos un $\sqsubseteq$-menor punto fijo, entonces es único, por la antisimetría del orden.

El siguiente ejemplo muestra cómo podemos hablar de cadenas en **State** $\hookrightarrow$ **State**:

Ejemplo 5.22. Consideremos la sucesión $(g_n) \subseteq$ **State** $\hookrightarrow$ **State** dada por

$$g_n\ s := \begin{cases} \texttt{INDEFINIDO}, \text{ si } s\ x > n \\ s[\texttt{x} \mapsto -1], \text{ si } 0 \leq s\ x < n \\ s, \text{ si } s\ x < 0 \end{cases}$$

Notemos que $g_n \sqsubseteq g_m$ si $n \leq m$ porque g_n está indefinida para más casos que g_m. Entonces, $C : \{g_n \mid n \in \mathbb{N}\}$ es cadena. C está acotada superiormente por

$$g\ s := \begin{cases} s[\mathtt{x} \mapsto -1], \text{ si } 0 \leq s\ x \\ s, \text{ si } s\ x < 0 \end{cases}$$

y, de hecho, g es supremo de C.

Tenemos el siguiente resultado:

Lema 5.23. $(\mathbf{State} \hookrightarrow \mathbf{State}, \sqsubseteq)$ *es conjunto parcialmente ordenado completo. Si C es una cadena en* $\mathbf{State} \hookrightarrow \mathbf{State}$*, entonces*[5]

$$G\left(\bigsqcup C\right) = \bigcup\{G(g) \mid g \in C\}.$$

Demostración. Veamos que $\bigcup\{G(g) \mid g \in C\}$ es grafo de una función parcial en $\mathbf{State} \hookrightarrow \mathbf{State}$. Sean $(s, s'), (s, s'')$ elementos de este conjunto. Entonces existen $g, h \in C$ tales que $gs = s'$ y $hs = s''$. Como C es cadena, h y g son comparables, luego obtenemos $s' = gs = hs = s''$, como queríamos.

Sea entonces g_0 tal función parcial con $G(g_0) = \bigcup\{G(g) \mid g \in C\}$. Veamos que g_0 es cota superior de C: como es obvio que $G(g) \subseteq G(g_0)$ para cada $g \in C$, tenemos que $g \sqsubseteq g_0$.

Finalmente, veamos que g_0 es el supremo de C. En efecto, si g_1 es otra cota superior de C, tendremos que $G(g) \subseteq G(g_1)$, para cada $g \in C$. Tomando la unión en cada lado, tenemos que

$$G(g_0) = \bigcup\{G(g) \mid g \in C\} \subseteq G(g_1)$$

es decir, $g_0 \sqsubseteq g_1$, como queríamos. □

Con el siguiente ejemplo vemos qué aspecto tiene una aplicación continua en el caso especial que estudiamos:

Ejemplo 5.24. Recordemos el siguiente funcional asociado a `while`$\neg(\mathtt{x} = 0)$`do skip`:

$$F(g)s := \begin{cases} gs, \text{ si } s\mathtt{x} \neq 0 \\ s, \text{ en otro caso} \end{cases}$$

Veamos que es continuo. Tenemos que ver que F es monótona y que $F(\bigsqcup C) = \bigsqcup F(C)$ para una cadena arbitraria $C \neq \emptyset$. Sean $g \sqsubseteq h$ cualesquiera. Entonces, basta ver la definición para notar que la relación de $F(g)$ y $F(h)$ depende únicamente de la que ya había entre g y h, porque, si $F(g)s = s'$, entonces tenemos que $F(g)s$ puede ser $gs = s' = hs$ o bien puede ser s, es decir, $F(g) \sqsubseteq F(h)$.

Por otro lado, sea $C \neq \emptyset$ cadena y sea $\bigsqcup C$. Consideremos $F(\bigsqcup C)$. Sean a

5.3.2. Existencia

El Teorema 5.17 aplicado al orden $\sqsubseteq$ nos dice que, bajo determinadas condiciones, $\mathtt{fix}(\cdot)$ permite definir una función. En esta sección queremos estudiar estas condiciones. Para empezar, recordemos que definíamos

$$\mathcal{S}_{\mathrm{ds}}[[\mathtt{while}\ b\ \mathtt{do}\ S]] := \mathtt{fix}(F),$$

siendo $F(g) := \mathtt{cond}(\mathcal{B}[\![b]\!], g \circ \mathcal{S}_{\mathrm{ds}}[[S]], id)$. Por tanto, debemos verificar que F verifica las condiciones de las que hablamos. El siguiente Lema nos da un resultado en esta dirección:

Lema 5.25. *Sea $p \in \mathbf{State} \to \mathbf{Bool}$. La aplicación $\Phi(g, h) := \mathtt{cond}(p, g, h)$ es continua en las dos variables g, h, es decir, tanto*

$$F(g) := \mathtt{cond}(p, g, g_0) \quad \textit{como} \quad H(g) := \mathtt{cond}(p, g_0, g)$$

son continuas, siendo $g_0 \in \mathbf{State} \hookrightarrow \mathbf{State}$.

[5] Nótese que esto quiere decir que $(\bigsqcup C)s = s'$ si y solo si $gs = s'$, para cierta $g \in C$.

Demostración. Vamos a demostrar la continuidad de F, el otro caso es análogo. Primero veamos la monotonía de F. Sean $g_1 \sqsubseteq g_2$. Supongamos que $F(g_1)s = s'$. Si $ps = \mathbf{tt}$, $F(g_1)s = g_1 s = s'$ implica $s' = g_2 s = F(g_2)s$. Si $ps = \mathbf{ff}$, $F(g_1)s = g_0 s = F(g_2)s$ y el resultado es trivial.

Veamos ahora la continuidad. Basta ver que, si $C \subseteq \mathbf{State} \hookrightarrow \mathbf{State}$ es una cadena arbitraria no vacía, $F(\bigsqcup C) \sqsubseteq \bigsqcup F(C)$ (la otra dirección viene dada, en general, por monotonía). Por lo que ya vimos antes, podemos ver que $G(F(\bigsqcup C)) \subseteq \bigcup\{G(F(g)) \mid g \in C\}$ para tener el resultado. Sea $(s, s') \in G(F(\bigsqcup C))$. Si $ps = \mathbf{tt}$, $F(\bigsqcup C)s = g_0 s = s'$. Pero entonces, para cada $g \in C$, $F(g)s = g_0 s = s'$. Si $ps = \mathbf{tt}$, $F(\bigsqcup C)s = (\bigsqcup C)s = s'$, es decir, $(s, s') \in G(\bigsqcup C) = \bigcup\{G(g) \mid g \in C\}$, luego existe cierta $h \in C$ con $hs = s'$, es decir, $F(h)s = s'$, lo que nos da el resultado. □

Lema 5.26. *La aplicación $\Psi(g, h) := g \circ h$ es continua en las dos variables, es decir, tanto*

$$F(g) := g \circ g_0 \quad \textit{como} \quad H(g) := g_0 \circ g$$

son continuas, siendo $g_0 \in \mathbf{State} \hookrightarrow \mathbf{State}$.

Demostración. Veamos el caso F. F es monótona: $g_1 \sqsubseteq g_2$ implica $G(g_1) \subseteq G(g_2)$, luego

$$G(g_0) \circ G(g_1) \subseteq G(g_0) \circ G(g_2)$$

que prueba que $F(g) := g_0 \circ g_1 \sqsubseteq g_0 \circ g_2 =: F(g_2)$.
F es continua: sea C cadena no vacía cualquiera. Entonces,

$$G\left(F\left(\bigsqcup C\right)\right) = G\left(\bigsqcup C \circ g_0\right) = G(g_0) \circ G\left(\bigsqcup C\right)$$

donde hemos empleado $\circ$ en el segundo miembro como la composición de relaciones[6]. Ahora bien, lo anterior es igual a

$$G(g_0) \circ \bigcup\{G(g) \mid g \in C\} = \bigcup\{G(g_0) \circ G(g) \mid g \in C\} = G\left(\bigsqcup F(C)\right).$$

□

Teorema 5.27. *El sistema de semántica denotacional para* `While` *define una función total $\mathcal{S}_{\mathrm{ds}} : \mathbf{Stm} \to (\mathbf{State} \hookrightarrow \mathbf{State})$.*

Demostración. Como tenemos definiciones composicionales, la demostración es por inducción estructural sobre la metavariable $S \in \mathbf{Stm}$. Los casos base son:

(i) Si S es de la forma $x := a$, entonces basta comprobar que la aplicación $s \mapsto s[x \mapsto \mathcal{A}[\![a]\!]s]$ está bien definida. Pero esto es directo.

(ii) Si $S =$ `skip`, la aplicación asociada es la identidad, que está bien definida.

Los casos inductivos son:

(i) Si $S = S_1; S_2$, entonces $\mathcal{S}_{\mathrm{ds}}[[S]] = \mathcal{S}_{\mathrm{ds}}[[S_2]] \circ \mathcal{S}_{\mathrm{ds}}[[S_1]]$, que es composición, por hipótesis de inducción, de funciones bien definidas.

(ii) Si $S =$ `if` b `then` S_1 `else` S_2, de nuevo tenemos que $\mathcal{S}_{\mathrm{ds}}[[S_1]]$ y $\mathcal{S}_{\mathrm{ds}}[[S_2]]$ están bien definidas por hipótesis. Pero es directo comprobar que esto se mantiene por la aplicación `cond`.

[6] Sean $R \subseteq A \times B, S \subseteq B \times C$. La *composición* de R y S se define como

$$R \circ S := \{(x, z) \in A \times C \mid \exists y \in B.(xRy \wedge ySz)\}$$

Entonces, notemos que el orden de composición difiere del que tenemos para las funciones, es decir,

$$G(f \circ g) = G(g) \circ G(f).$$

(iii) Si $S = \texttt{while}\ b\ \texttt{do}\ S_1$, sabemos que $\mathcal{S}_{\text{ds}}[[S_1]]$ está bien definida. Pero las aplicaciones

$$F_1(g) := \texttt{cond}(\mathcal{B}[\![b]\!], g, id), \quad F_2(g) := g \circ \mathcal{S}_{\text{ds}}[[S_1]]$$

son continuas por los lemas previos. Como sabemos que composición de continuas es continua, tenemos que $F(g) := F_1(F_2(g))$ es continua y, por el Teorema 5.17, $\texttt{fix}(F)$ está bien definido, como queríamos ver.

□

Ejemplo 5.28. Veamos cómo aplica $\mathcal{S}_{\text{ds}}$ a la sentencia

$$S \equiv \texttt{y} := \texttt{1}; \texttt{while}\ \neg(\texttt{x} = \texttt{1})\ \texttt{do}\ (\texttt{y} := \texttt{y} * \texttt{x}; \texttt{x} := \texttt{x} - \texttt{1})$$

y cómo obtener el punto fijo para su bucle. Aplicamos la reglas del sistema denotacional para While (5.1.1) y nos sale

$$\mathcal{S}_{\text{ds}}[[S]]s_0 = \texttt{fix}(F)\ s_0[\texttt{y} \mapsto \texttt{1}]$$

donde

$$(F\ g)\ s = \begin{cases} g\ (\mathcal{S}_{\text{ds}}[[\texttt{y} * \texttt{x}; \texttt{x} := \texttt{x} - \texttt{1}]]s) & \mathcal{B}[\![\neg(\texttt{x} = \texttt{1})]\!]s = \mathbf{tt} \\ s & \mathcal{B}[\![\neg(\texttt{x} = \texttt{1})]\!]s = \mathbf{ff} \end{cases}$$

Aplicando nuevamente las reglas de $\mathcal{S}_{\text{ds}}$ se puede reescribir como

$$(F\ g)\ s = \begin{cases} g\ (s[\texttt{y} \mapsto (s\ \texttt{y}) * (s\ \texttt{x})][\texttt{x} \mapsto (s\ \texttt{x}) - \texttt{1}]) & s\ \texttt{x} \neq \texttt{1} \\ s & s\ \texttt{x} = \texttt{1} \end{cases}$$

Ahora computamos $F^n\ \bot$ para obtener el menor punto fijo

$$\begin{aligned} (F^0\bot)\ s &= \texttt{INDEFINIDO} \\ (F^1\bot)\ s &= \begin{cases} \texttt{INDEFINIDO} & s\ \texttt{x} \neq 1 \\ s & s\ \texttt{x} = 1 \end{cases} \\ (F^2\bot)\ s &= \begin{cases} \texttt{INDEFINIDO} & s\ \texttt{x} \neq 1 \wedge s\ \texttt{x} \neq 2 \\ s[\texttt{y} \mapsto (s\ \texttt{y}) * 2][x \mapsto 1] & s\ \texttt{x} = 2 \\ s & s\ \texttt{x} = 1 \end{cases} \end{aligned}$$

Cada evaluación de F representa el cómputo del cuerpo del bucle tantas veces como la condición booleana se cumpla. La fórmula general es

$$(F^n\bot)\ s = \begin{cases} \texttt{INDEFINIDO} & s\ \texttt{x} < 1 \vee s\ \texttt{x} > \texttt{n} \\ s[\texttt{y} \mapsto (s\ \texttt{y}) * j * ... * 2 * 1][x \mapsto 1] & s\ \texttt{x} = j \wedge 1 \leq j \wedge j \leq n \end{cases}$$

y se tiene que el punto fijo es

$$(\texttt{fix}\ F)\ s = \begin{cases} \texttt{INDEFINIDO} & s\ \texttt{x} < 1 \\ s[\texttt{y} \mapsto (s\ \texttt{y}) * (s\ \texttt{x})!][x \mapsto 1] & s\ \texttt{x} \geq 1 \end{cases}$$

Si se toma por ejemplo un estado $s_0 \in \mathbf{State}$ tal que $s_0\ \texttt{x} = 3$ se tendría

$$(\texttt{fix}(F)\ s_0[\texttt{y} \mapsto \texttt{1}])\ \texttt{y} = 1 * 3 * 2 * 1 = 6$$

5.3.3. Equivalencia semántica

Ahora que ya disponemos de un sistema de semántica denotacional, parece lógico repetirse las mismas preguntas que ya nos hicimos en su momento con la semántica operacional.

Definición 5.29. Dos sentencias $S_1, S_2 \in \mathbf{Stm}$ se dicen *semánticamente equivalentes* si $\mathcal{S}_{\text{ds}}[[S_1]] = \mathcal{S}_{\text{ds}}[[S_2]]$.

Ejemplo 5.30. EJERCICIO 5.53

5.4. Teorema de equivalencia

Como siempre, cuando introducimos un nuevo sistema de semántica, nos interesa estudiar si se encuentra relacionado con otros que ya hemos presentado antes. Más concretamente, nos interesa el siguiente enunciado:

Teorema 5.31. *Para cada* $S \in \mathbf{Stm}$, $\mathcal{S}_{\text{sos}}[[S]] = \mathcal{S}_{\text{ds}}[[S]]$.

Notemos que, tanto $\mathcal{S}_{\text{sos}}[[S]]$ como $\mathcal{S}_{\text{ds}}[[S]]$ son elementos de $\mathbf{State} \hookrightarrow \mathbf{State}$, que es un conjunto ordenado por $\sqsubseteq$, luego basta demostrar que $\mathcal{S}_{\text{sos}}[[S]] \sqsubseteq \mathcal{S}_{\text{ds}}[[S]]$ y $\mathcal{S}_{\text{ds}}[[S]] \sqsubseteq \mathcal{S}_{\text{sos}}[[S]]$, para $S \in \mathbf{Stm}$ arbitrario.