Índice

Tema 1. El lenguaje WHILE

Sintaxis

Tenemos los siguientes conjuntos:

- Numerales: $\mathrm{n} \in \mathrm{Num}$.
- Variables: $\mathrm{x} \in \mathrm{Var}$.
- Expresiones aritméticas: $\mathrm{a} \in \mathrm{Aexp}$.
- Expresiones booleanas: $\mathrm{b} \in \mathrm{Bexp}$.
- Sentencias: $\mathrm{S} \in \mathrm{Stm}$.

Semántica

Numerales

Siendo la sintaxis:

```
n ::= 0 | 1 | n 0 | n 1
```

La función semántica es:

$$\mathcal{N} : \mathrm{Num} \to \mathbb{Z}$$

$$\begin{aligned}\mathcal{N}[\![0]\!] &= 0\\ \mathcal{N}[\![1]\!] &= 1\\ \mathcal{N}[\![n\ 0]\!] &= 2 \cdot \mathcal{N}[\![n]\!]\\ \mathcal{N}[\![n\ 1]\!] &= 2 \cdot \mathcal{N}[\![n]\!] + 1\end{aligned}$$

Composición

Definiciones composicionales

1. Una categoría sintáctica se define dando una *sintaxis abstracta* de unos *elementos básicos* y otros *compuestos*, que se pueden descomponer en sus constituyentes inmediatos.

2. La semántica de esta categoría se define *composicionalmente*. Se da la definición directa de los elementos base y de los compuestos se da en base a la semántica de sus constituyentes.

Inducción estructural

1. Probar que la propiedad se da para los elementos base de la categoría.

2. Asumiendo, *hipótesis de inducción*, que la propiedad se cumple para los constituyentes de un elemento compuesto, probar que también se cumple para el elemento en sí.

Con esto buscamos *homomorfismos de álgebras* usando los constructores como operadores.

Expresiones

Para las expresiones tenemos que definir la función *estado*:

$$\mathrm{State} = \mathrm{Var} \to \mathbb{Z}.$$

Con esto podemos definir la semántica de las expresiones:

$$\begin{aligned}\mathcal{A} &: \mathrm{Aexp} \to (\mathrm{State} \to \mathbb{Z})\\ \mathcal{B} &: \mathrm{Bexp} \to (\mathrm{State} \to \mathbb{T})\end{aligned}$$

que se definen composicionalmente de manera natural.

Propiedades de la semántica

Variables libres

Llamamos *variables libres* de una expresión a aquellas variables que se encuentran en la misma. Se definen composicionalmente sobre los operadores de la expresión de manera natural.

Lema
Sean s y s' dos estados tales que $s\ x = s'\ x,\ \forall x \in \mathrm{FV}\ (a)$. Entonces, $\mathcal{A}[\![a]\!]s = \mathcal{A}[\![a]\!]s'$.

Sustituciones

Una sustitución en una expresión consiste en el reemplazamiento de cada aparición de una variable libre por una expresión. De nuevo, se define composicionalmente de manera natural, pero para el caso base será así:

$$x\,[y \to a_0] = \begin{cases} a_0 & \text{si } x = y \\ x & \text{si } x \neq y \end{cases}$$

Dicho de otra forma, los estados se actualizarán tal que:

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

Tema 2. Semántica operacional del lenguaje WHILE

Semántica de paso largo

Transiciones

Configuraciones

- La *sentencia* S se ejecuta desde el estado s: $\langle S, s\rangle$.
- Estado s' que se alcanza al terminar su ejecución.

Transiciones ($\langle S, s\rangle \to s'$) :

- Asignación:

$$[\mathrm{ass}_{\mathrm{ns}}] := \langle x := a, s\rangle \to s\,[x \mapsto \mathcal{A}[\![a]\!]s]$$

- *Skip*:

$$[\mathrm{skip}_{\mathrm{ns}}] := \langle \mathtt{skip}, s\rangle \to s$$

- Composición:

$$[\text{comp}_{\text{ns}}] := \frac{\langle S_1, s\rangle \to s',\ \langle S_2, s'\rangle \to s''}{\langle S_1; S_2, s\rangle \to s''}$$

- Condicional:
  - Si se cumple:

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

  - Si no se cumple:

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

- Bucle:
  - Si se cumple:

$$\left[\text{while}^{\text{tt}}_{\text{ns}}\right] := \frac{\langle S, s\rangle \to s', \langle \texttt{while } b \texttt{ do } S, s'\rangle \to s''}{\langle \texttt{while } b \texttt{ do } S, s\rangle \to s''},\ \text{ si } \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

  - Si no se cumple:

$$\left[\text{while}^{\text{ff}}_{\text{ns}}\right] := \langle \texttt{while } b \texttt{ do } S, s\rangle \to s,\ \text{ si } \mathcal{B}[\![b]\!]s = \mathbf{ff}$$

Una sentencia S es un estado s tiene dos posibilidades:

- **Terminar** si, y sólo si, $\exists s' \in \mathbf{State}.\langle S, s\rangle \to s'$.
- **Ciclar** si, y sólo si, $\nexists s' \in \mathbf{State}.\langle S, s\rangle \to s'$.

Árboles de derivación

Para todo programa escrito en el lenguaje **WHILE** podemos escribir un árbol de derivación. Este árbol tendrá como raíz el programa entero y sus hijos serán aplicaciones de las reglas de la anterior sección. Las hojas serán *axiomas* (aquellas reglas que no tienen «conclusión»).

Si el programa «cicla» tendrá un árbol infinito, mientras que si termina será finito. En esta versión del **WHILE** para toda transición habrá una sola regla que se pueda aplicar.

Inducción sobre el árbol de derivación

1. Probar que la propiedad se cumple para los *axiomas*.

2. Probar para cada regla compuesta: asumiendo que se cumple para las hipótesis, ver que también es cierta para las conclusiones.

Propiedades

Definición (Equivalencia entre sentencias)

Decimos que dos sentencias S_1 y S_2 son semánticamente equivalentes si, y sólo si, $\forall s \in \mathbf{State}$ se cumple que:

$$\langle S_1, s\rangle \to s' \Leftrightarrow \langle S_2, s\rangle \to s'$$

Definición y proposición (Determinismo)

Decimos que una semántica es *determinista* si dada una sentencia S y cualquier estado s se cumple que:

$$\langle S, s\rangle \to s' \wedge \langle S, s\rangle \to s'' \Rightarrow s' = s''$$

Función semántica

Definición (Significado de una sentencia)

Definimos la semántica operacional de paso largo de las sentencias a través de la siguiente función:

$$\begin{gathered}\mathcal{S}_{ns} : \mathbf{Stm} \to (\mathbf{State} \hookrightarrow \mathbf{State}) \\ \mathcal{S}_{ns}[\![S]\!]s = \begin{cases} s', & \text{si } \langle S, s\rangle \to s' \\ \texttt{indefinido}, & \text{c.c} \end{cases}\end{gathered}$$

Semántica de paso corto

Transición

Primer paso de ejecución de S: $\langle S, s\rangle \Rightarrow \gamma$.

- Si $\gamma = \langle S', s'\rangle$, ejecución *no terminada*.
- Si $\gamma = s'$, ejecución *terminada*.

Si para un $\langle S, s\rangle$ *no existe* ningún $\gamma . \langle S, s\rangle \Rightarrow \gamma$, la configuración estará *bloqueada*.

- Asignación:

$$[\text{ass}_{\text{sos}}] := \langle x := a, s\rangle \Rightarrow s\,[x \mapsto \mathcal{A}[\![a]\!]s]$$

- *Skip*:

$$[\text{skip}_{\text{sos}}] := \langle \texttt{skip}, s\rangle \Rightarrow s$$

- Composición:

$$\begin{aligned}\left[\text{comp}^1_{\text{sos}}\right] &:= \frac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_1'; S_2, s'\rangle} \\ \left[\text{comp}^2_{\text{sos}}\right] &:= \frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}\end{aligned}$$

- Condicional:
  - Si se cumple:

$$\left[\text{if}^{\text{tt}}_{\text{sos}}\right] := \langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, s\rangle \Rightarrow \langle S_1, s\rangle, \;\; \text{si } \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

  - Si no se cumple:

$$\left[\text{if}^{\text{tt}}_{\text{sos}}\right] := \langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, s\rangle \Rightarrow \langle S_2, s\rangle, \;\; \text{si } \mathcal{B}[\![b]\!]s = \mathbf{ff}$$

- Bucle:

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

Las secuencias de derivación pueden *terminar* (con éxito, alcanza un s', o no) o *ciclar*.

Inducción sobre la longitud de secuencia

1. Demostrar la propiedad para las secuencias de longitud 0.

2. Asumiendo que se cumple para las secuencias de longitud k, probarla para secuencias de longitud $k+1$.

Propiedades

Lema

Si tenemos $\langle S_1; S_2, s\rangle \Rightarrow^k s''$, entonces existen $s' \in \mathbf{State}$ y $k_1, k_2 \in \mathbb{N}$ tales que:

$$\langle S_1, s\rangle \Rightarrow^{k_1} s' \wedge \langle S_2, s\rangle \Rightarrow^{k_2} s'',\ \text{con}\ k = k_1 + k_2.$$

Definición (Equivalencia semántica)

Decimos que dos sentencias S_1 y S_2 son semánticamente equivalentes si para todo estado s se cumple que:

- $\langle S_1, s\rangle \Rightarrow^* \gamma \Leftrightarrow \langle S_2, s\rangle \Rightarrow^* \gamma$, para cada γ terminal o bloqueada.
- La secuencia que inicia con $\langle S_1, s\rangle$ es *infinita* si, y sólo si, lo es la que inicia con $\langle S_2, s\rangle$.

Función semántica

Definición (Significado de una sentencia)

Definimos la semántica operacional de paso corto de las sentencias a través de la siguiente función:

$$\mathcal{S}_{sos} : \mathbf{Stm} \to (\mathbf{State} \hookrightarrow \mathbf{State})$$
$$\mathcal{S}_{sos}[\![S]\!]s = \begin{cases} s', & \text{si } \langle S, s\rangle \Rightarrow^* s' \\ \texttt{indefinido}, & \text{c.c} \end{cases}$$

Equivalencia

Teorema

Para toda sentencia $S \in \mathbf{Stm}$ se cumple la siguiente relación:

$$\mathcal{S}_{ns}[\![S]\!] = \mathcal{S}_{sos}[\![S]\!]$$

Es decir, la semántica operacional de paso corto y paso largo con *equivalentes*.

Tema 3. Más sobre semántica operacional

Más allá del determinismo

Abortar cómputos

- Para definir la semántica de la instrucción `abort` *no la definimos*. De esta forma cuando llegamos a ella tenemos `undefined` y el programa alcanza un estado `stuck`.
- Con esta nueva instrucción encontramos una diferencia entre la semántica operacional de paso corto y paso largo. En el paso corto distinguimos entre bucles infinitos y terminaciones forzosas mientras que en la de paso largo solo importa la terminación correcta. En otras palabras, en paso corto `while true do skip` *no* es equivalente a `abort` mientras que en paso largo *sí*.

Indeterminación

- Añadimos una nueva instrucción `S1 or S2` que ejecuta cualquiera (pero solo una) de las dos sentencias de forma no determinista.
- La definición de paso largo es

$$\left[\mathrm{or}^1_{\mathrm{ns}}\right] := \frac{\langle S_1, s\rangle \to s'}{\langle S_1 \ \mathbf{or}\ S_2, s\rangle \to s'} \qquad \left[\mathrm{or}^2_{\mathrm{ns}}\right] := \frac{\langle S_2, s\rangle \to s'}{\langle S_1 \ \mathbf{or}\ S_2, s\rangle \to s'}$$

- Y la de paso corto

$$\left[\mathrm{or}^1_{\mathrm{sos}}\right] := \langle S_1 \ \mathbf{or}\ S_2, s\rangle \Rightarrow \langle S_1, s\rangle \qquad \left[\mathrm{or}^2_{\mathrm{sos}}\right] := \langle S_1 \ \mathbf{or}\ S_2, s\rangle \Rightarrow \langle S_2, s\rangle$$

- Esta instrucción nos aporta una nueva diferencia entre las dos semánticas operacionales: el paso corto *mantiene* los ciclos en presencia de indeterminación mientras que el largo los *elimina*.

Paralelismo

- Extendemos ahora con la instrucción `S1 par S2` que ejecuta paralelamente las dos sentencias, es decir, se intercalan las instrucciones de las dos.
- El paso corto será:

$$\begin{aligned}
\left[\mathrm{par}^1_{\mathrm{sos}}\right] &:= \frac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1 \ \mathbf{par}\ S_2, s\rangle \Rightarrow \langle S_1' \ \mathbf{par}\ S_2, s'\rangle} \\
\left[\mathrm{par}^2_{\mathrm{sos}}\right] &:= \frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1 \ \mathbf{par}\ S_2, s\rangle \Rightarrow \langle S_2, s'\rangle} \\
\left[\mathrm{par}^3_{\mathrm{sos}}\right] &:= \frac{\langle S_2, s\rangle \Rightarrow \langle S_2', s'\rangle}{\langle S_1 \ \mathbf{par}\ S_2, s\rangle \Rightarrow \langle S_1 \ \mathbf{par}\ S_2', s'\rangle} \\
\left[\mathrm{par}^2_{\mathrm{sos}}\right] &:= \frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1 \ \mathbf{par}\ S_2, s\rangle \Rightarrow \langle S_1, s'\rangle}
\end{aligned}$$

- Sin embargo, el paso largo no será capaz de definir esta construcción puesto que solo capta el estado final y no los pasos intermedios, es decir, en el mejor de los casos definirá una construcción en la que una sentencia se ejecuta antes que la otra, pero no más.

Bloques y procedimientos

Bloques

- Extendemos **WHILE** con bloques `begin Dv S end` donde tenemos declaraciones de variables locales: `Dv ::= var x := a; Dv |` . Debemos añadir que $D_V \in \mathbf{Dec}_V$ que es la categoría sintáctica de *declaraciones de variables*. Con esto definimos el conjunto de variables declaradas en `Dv` :

$$\begin{cases} \mathrm{DV}\left(\mathbf{var}\ x := a; D_V\right) & = \{x\} \cup \mathrm{DV}_V \\ \mathrm{DV}\left(\varepsilon\right) & = \emptyset \end{cases}$$

- A la hora de definir la semántica de las declaraciones tenemos que tener en cuenta que *no* son instrucciones. Por esto se tienen que definir de una nueva forma. Además, para recuperar las variables globales que se han redeclarado necesitamos la siguiente función:

$$\left(s'\left[X \mapsto s\right] x\right) := \begin{cases} s\ x, & \text{si } x \in X \\ s'\ x, & \text{si } x \notin X \end{cases}$$

  Intuitivamente, s' es el estado tras abandonar el bloque, s es el de antes de entrar y X es el conjunto de variables globales redeclaradas.

- Definimos ahora la semántica para las declaraciones de variables:

  - Caso base:

$$\left[\mathrm{none}_{\mathrm{ns}}\right] := \langle \varepsilon, s\rangle \to_D s$$

  - Caso recursivo:

$$\left[\mathrm{var}_{\mathrm{ns}}\right] := \frac{\langle D_V, s\left[x \mapsto \mathcal{A}[\![a]\!]s\right]\rangle \to_D s'}{\langle \mathbf{var}\ x := a; D_V, s\rangle \to_D s'}$$

- Y para los bloques:

$$\left[\mathrm{block}_{\mathrm{ns}}\right] := \frac{\langle D_V, s\rangle \to_D s', \langle S, s'\rangle \to s''}{\langle \mathbf{begin}\ D_V\ S\ \mathbf{end}, s\rangle \to s''\left[\mathrm{DV}\left(D_V\right) \mapsto s\right]}.$$

  Es decir, realizamos las declaraciones de las variables locales desde s lo que nos da s', ejecutamos S con s' y obtenemos s'', pero al final tenemos que devolver las variables globales a su estado original con la función auxiliar que hemos visto antes.

Procedimientos

- Presentamos ahora los *procedimientos*, otra nueva construcción que acompaña a los bloques (que no los sustituye, es decir, son los bloques los que generan ámbitos, no los procedimientos).

- Tenemos que cambiar la sintaxis añadiendo una nueva instrucción `call p` y modificando los bloques `begin Dv Dp S end` . Además, tenemos que añadir las declaraciones de los

procedimientos: `Dp ::= proc p is S; Dp |  `. Con esto tenemos que $D_P \in \mathbf{Dec}_P$ que es la categoría sintáctica de *declaraciones de procedimientos* y $p \in \mathbf{Pname}$ que es el conjunto de nombres de procedimientos.

- Para definir la semántica extendida con los procedimientos tenemos tres posibilidades distintas

Variables y procedimientos dinámicos

- **Entorno de procedimientos:**
  Función que asocia los nombres de los procedimientos a sus cuerpos:

$$\mathbf{Env}_P = \mathbf{Pname} \hookrightarrow \mathbf{Stm}$$

  La función no es total para poder hacer llamadas a procedimientos todavía no declarados (por ejemplo, para permitir la recursividad mutua).

- Si ahora tenemos un entorno $env_P \in \mathbf{Env}_P$ tenemos que las transiciones tendrán en cuenta el entorno de la siguiente manera:

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

- Antes de definir la nueva semántica de largo paso, necesitamos una nueva función auxiliar que actualice $\mathbf{Env}_P$ cuando entramos en un bloque para añadir los nuevos procedimientos que se definan:

$$\mathrm{upd}_P : \mathbf{Dec}_P \times \mathbf{Env}_P \to \mathbf{Env}_P$$

  Lo hacemos recursivamente:

  - Caso base:

$$\mathrm{upd}_P\left(\varepsilon, env_P\right) = env_P$$

  - Caso recursivo:

$$\mathrm{udp}_P\left(\texttt{proc}\ p\ \texttt{is}\ S; D_P, env_P\right) = \mathrm{udp}_P\left(D_P, env_P\left[p \mapsto S\right]\right)$$

- Con todo esto, ya podemos actualizar la semántica de **WHILE**:

  - Asignación:

$$\left[\mathrm{ass}_{\mathrm{ns}}\right] := env_P \vdash \langle x := a, s\rangle \to s\left[x \mapsto \mathcal{A}[\![a]\!]s\right]$$

  - *Skip*:

$$\left[\mathrm{skip}_{\mathrm{ns}}\right] := env_P \vdash \langle \texttt{skip}, s\rangle \to s$$

  - Composición:

$$\left[\mathrm{comp}_{\mathrm{ns}}\right] := \frac{env_P \vdash \langle S_1, s\rangle \to s',\ env_P \vdash \langle S_2, s'\rangle \to s''}{env_P \vdash \langle S_1; S_2, s\rangle \to s''}$$

  - Condicional:

- Si se cumple:

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

- Si no se cumple:

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

◦ Bucle:

- Si se cumple:

$$\left[\text{while}^{\text{tt}}_{\text{ns}}\right] := \frac{env_P \vdash \langle S, s\rangle \to s', env_P \vdash \langle \texttt{while } b \texttt{ do } S, s'\rangle \to s''}{env_P \vdash \langle \texttt{while } b \texttt{ do } S, s\rangle \to s''} , \\ \text{si } \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

- Si no se cumple:

$$\left[\text{while}^{\text{ff}}_{\text{ns}}\right] := env_P \vdash \langle \texttt{while } b \texttt{ do } S, s\rangle \to s, \text{ si } \mathcal{B}[\![b]\!]s = \mathbf{ff}$$

◦ Bloque:

$$[\text{block}_{\text{ns}}] := \frac{\langle D_V, s\rangle \to_D s', \ \text{udp}_P\left(D_P, env_P\right) \vdash \langle S, s'\rangle \to s''}{env_P \vdash \langle \texttt{begin } D_V \ D_P \ S \texttt{ end}, s\rangle \to s''\left[\text{DV}\left(D_V\right) \mapsto s\right]}$$

◦ Llamada:

$$[\text{call}^{\text{rec}}_{\text{ns}}] := \frac{env_P \vdash \langle S, s\rangle \to s'}{env_P \vdash \langle \texttt{call } p, s\rangle \to s'}, \text{ si } env_P\ p = S$$

Variables dinámicas y procedimientos estáticos

- **Entorno de procedimientos:**
  Función que asocia los nombres de los procedimientos a sus cuerpos:

$$\mathbf{Env}_P = \mathbf{Pname} \hookrightarrow \mathbf{Stm} \times \mathbf{Env}_P$$

  Necesitamos recordar también los procedimientos *ya declarados* en el momento que se declara uno nuevo (puesto que las llamadas son estáticas). Ahora necesitamos una nueva definición para udp_P que será:

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

  Lo que quiere decir que, cuando actualizamos el entorno para añadir p, guardamos no solo su cuerpo sino también el entorno *en ese momento*.

- Con esto ya podemos actualizar la definición semántica. En general será igual a la vista en la anterior sección a excepción de las llamadas:

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

Sin embargo, en este caso las llamadas recursivas no las tenemos directamente y tenemos que añadir una nueva derivación para el caso de que una función se llame a sí misma (observar que cuando actualizamos el entorno, no p no tiene asociado el entorno en el que ha sido declarada sino *el anterior*).

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

En esta nueva definición estamos añadiendo el valor de p al entorno anterior al que se declaró con lo que podemos realizar la llamada recursiva.

Variables y procedimientos estáticos

- Para tener variables estáticas necesitamos modificar el estado y «separarlo» en dos nuevas funciones distintas. Una guardará la *localización* de la variable en un memoria abstracta mientras que la otra *asignará* un valor a esa dirección. Con esto el estado será la composición de estas dos funciones.

- **Funciones $\mathbf{Env}_V$:**
  Esta funciones mapean variables de $\mathbf{Var}$ a posiciones de una memoria abstracta, $\mathbf{Loc}$:

  $$\mathbf{Env}_V := \mathbf{Var} \to \mathbf{Loc}$$

- Ahora teniendo un “memoria” $\mathbf{Loc}$, necesitamos una forma de obtener la siguiente dirección libre. Para ello usamos new que es una función que dada una dirección, nos devuelve la siguiente libre:

  $$\text{new} : \mathbf{Loc} \to \mathbf{Loc}$$

- **Funciones $\mathbf{Store}$:**
  Estas funciones asignan a las direcciones de $\mathbf{Loc}$ valores que pueden tener las variables (en el caso de **WHILE**, enteros):

  $$\mathbf{Store} = \mathbf{Loc} \cup \{\texttt{next}\} \to \mathbb{Z}$$

  El elemento `next` representa la siguiente dirección de memoria libre.

- Con esto la semántica de las declaraciones de variables tendrá que ser actualizada de la siguiente forma:

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

- Con todo esto ya sí podemos dar las definiciones de la semántica de las declaraciones de variables. De nuevo lo haremos de forma recursiva:

  - Caso base:

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

- Caso recursivo: $[\mathrm{var}_{\mathrm{ns}}] :=$

$$\frac{\langle D_V, env_V\,[x \mapsto l]\,, sto\,[l \mapsto v]\,[\mathtt{next} \mapsto \mathrm{new}\ l]\rangle \to_D (env'_V, sto')}{\langle \mathtt{var}\ x := a;\ D_V, env_V, sto\rangle \to_D (env'_V, sto')}, \quad \text{si } v = \mathcal{A}[\![a]\!]\,(sto \circ env_V)\ \text{ y } l = sto\,\mathtt{next}$$

- De forma intuitiva, la anterior definición nos indica varias cosas:
  - El elemento v nos da el valor de la nueva variable x en el estado anterior a su declaración mientras que l nos indica la posición que ocupará.
  - En la premisa estamos actualizando recursivamente el entorno de las variables añadiendo la posición que ocupará x y el *store* con el valor de x (que se asociará a su posición l) así como con el nuevo **`next`** que lo aportará la función new, que hemos definido antes.
- Con todas estas definiciones ya estamos en disposición de definir el *entorno de procedimientos* para este caso. Claramente será necesario el entorno de variables. Por tanto:

$$\mathbf{Env}_P = \mathbf{Pname} \hookrightarrow \mathbf{Stm} \times \mathbf{Env}_V \times \mathbf{Env}_P$$

  Y, claro, la actualización tendrá que "recordar" también las variables declaradas en ese momento: $\mathrm{udp}_P : \mathbf{Dec}_p \times \mathbf{Env}_V \times \mathbf{Env}_P \to \mathbf{Env}_P$.

$$\begin{cases} \mathrm{udp}_P\,(\varepsilon, env_V, env_P) & = env_P \\ \mathrm{udp}_P\,(\mathtt{proc}\ p\ \mathtt{is}\ S; D_P, env_V, env_P) & = \mathrm{udp}_P\,(D_P, env_V, env_P\,[p \mapsto (S, env_V, env_P)]) \end{cases}$$

- De esta forma las transiciones de la semántica de paso largo serán de la siguiente forma:

$$env_V, env_P \vdash \langle S, sto\rangle \to sto'$$

- Con todo esto, ya podemos actualizar la semántica de **WHILE**:
  - Asignación:

$$[\mathrm{ass}_{\mathrm{ns}}] := env_V, env_P \vdash \langle x := a, sto\rangle \to sto\,[l \mapsto v]\,, \quad \text{si } l = env_V\ x \text{ y } v = \mathcal{A}[\![a]\!]\,(sto \circ env_V)$$

  - *Skip*:

$$[\mathrm{skip}_{\mathrm{ns}}] := env_V, env_P \vdash \langle \mathtt{skip}, sto\rangle \to sto$$

  - Composición:

$$[\mathrm{comp}_{\mathrm{ns}}] := \frac{env_V, env_P \vdash \langle S_1, sto\rangle \to sto',\ env_V, env_P \vdash \langle S_2, sto'\rangle \to sto''}{env_V, env_P \vdash \langle S_1; S_2, sto\rangle \to sto''}$$

  - Condicional:
    - Si se cumple:

$$\left[\mathrm{if}^{\mathtt{tt}}_{\mathrm{ns}}\right] := \frac{env_V, env_P \vdash \langle S_1, sto\rangle \to sto'}{env_V, env_P \vdash \langle \mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2, sto\rangle \to sto'}, \quad \text{si } \mathcal{B}[\![b]\!]\,(sto \circ env_V) = \mathtt{tt}$$

- Si no se cumple:

$$\left[\text{if}^{\text{ff}}_{\text{ns}}\right] := \frac{env_V, env_P \vdash \langle S_2, sto\rangle \to sto'}{env_V, env_P \vdash \langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, sto\rangle \to sto'}, \\ \text{si } \mathcal{B}[\![b]\!]\,(sto \circ env_V) = \textbf{ff}$$

◦ Bucle:

- Si se cumple:

$$\left[\text{while}^{\text{tt}}_{\text{ns}}\right] := \frac{\begin{array}{c} env_V, env_P \vdash \langle S, sto\rangle \to sto', \\ env_V, env_P \vdash \langle \texttt{while } b \texttt{ do } S, sto'\rangle \to sto'' \end{array}}{env_P \vdash \langle \texttt{while } b \texttt{ do } S, sto\rangle \to sto''}, \\ \text{si } \mathcal{B}[\![b]\!]\,(sto \circ env_V) = \texttt{tt}$$

- Si no se cumple:

$$\left[\text{while}^{\text{ff}}_{\text{ns}}\right] := env_V, env_P \vdash \langle \texttt{while } b \texttt{ do } S, sto\rangle \to s, \;\; \text{si } \mathcal{B}[\![b]\!]\,(sto \circ env_V) = \textbf{ff}$$

◦ Bloque:

$$[\text{block}_{\text{ns}}] := \frac{\begin{array}{c} \langle D_V, env_V, sto\rangle \to_D (env'_V, sto'), \\ env'_V, env'_P \vdash \langle S, sto'\rangle \to sto'' \end{array}}{env_V, env_P \vdash \langle \texttt{begin } D_V\ D_P\ S \texttt{ end}, sto\rangle \to sto''}, \\ \text{si } env'_P = \text{udp}_P\,(D_P, env'_V, env_P)$$

Es decir, actualizamos el entorno de las variables para obtener env'_V y sto'. Tras esto, actualizamos el entorno de procedimientos para obtener env'_P. Con todo esto, ya sí ejecutamos S.

◦ Llamada:

- No recursiva:

$$[\text{call}_{\text{ns}}] := \frac{env'_V, env'_P \vdash \langle S, sto\rangle \to sto'}{env_V, env_P \vdash \langle \texttt{call } p, sto\rangle \to sto'}, \;\; \text{si } env_P\ p = (S, env'_V, env'_P)$$

- Recursiva:

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

Tema 4. Implementación correcta

La máquina correcta

La máquina abstracta está compuesta de configuraciones que siguen el siguiente patrón:

$$\langle c, e, s\rangle \in \mathbf{Code} \times \mathbf{Stack} \times \mathbf{State}$$

donde c es una serie de instrucciones ($push, add \ldots$), e es la pila de ejecución y s es el estado del programa. La configuración final es $\langle \varepsilon, e, s\rangle$

Las distintas instrucciones alteran la configuración de la máquina de la siguiente manera:

- "Empujar" un número:

$$\langle \texttt{PUSH}{-}n : c, e, s\rangle \triangleright \langle c, \mathcal{N}[\![n]\!] : e, s\rangle$$

- "Empujar" un booleano:

$$\langle \texttt{TRUE} : c, e, s\rangle \triangleright \langle c, \mathbf{tt} : e, s\rangle$$
$$\langle \texttt{FALSE} : c, e, s\rangle \triangleright \langle c, \mathbf{ff} : e, s\rangle$$

- Operaciones con enteros, $z_1, z_2 \in \mathbb{Z}$:

$$\langle \texttt{ADD} : c, z_1 : z_2 : e, s\rangle \triangleright \langle c, (z_1 + z_2) : e, s\rangle$$
$$\langle \texttt{SUB} : c, z_1 : z_2 : e, s\rangle \triangleright \langle c, (z_1 - z_2) : e, s\rangle$$
$$\langle \texttt{MULT} : c, z_1 : z_2 : e, s\rangle \triangleright \langle c, (z_1 * z_2) : e, s\rangle$$
$$\langle \texttt{EQ} : c, z_1 : z_2 : e, s\rangle \triangleright \langle c, (z_1 = z_2) : e, s\rangle$$
$$\langle \texttt{LE} : c, z_1 : z_2 : e, s\rangle \triangleright \langle c, (z_1 \leq z_2) : e, s\rangle$$

- Operaciones con booleanos, $t, t_1, t_2 \in \mathbf{T}$:

$$\langle \texttt{AND} : c, t_1 : t_2 : e, s\rangle \triangleright \begin{cases} \langle c, \mathbf{tt} : e, s\rangle, & \text{si } t_1 = \mathbf{tt} = t_2 \\ \langle c, \mathbf{ff} : e, s\rangle & \text{c.c} \end{cases}$$
$$\langle \texttt{NEG} : c, t : e, s\rangle \triangleright \begin{cases} \langle c, \mathbf{ff} : e, s\rangle & \text{si } t = \mathbf{tt} \\ \langle c, \mathbf{tt} : e, s\rangle & \text{c.c} \end{cases}$$

- Acceso a memoria:

$$\langle \texttt{FETCH}{-}x : c, e, s\rangle \triangleright \langle c, (s\ x) : e, s\rangle$$
$$\langle \texttt{STORE}{-}x : c, z : e, s\rangle \triangleright \langle c, e, s\,[x \mapsto z]\rangle$$

- Control de "flujo":

$$\langle \texttt{NOOP} : c, e, s\rangle \triangleright \langle c, e, s\rangle$$
$$\langle \texttt{BRACH}\,(c_1, c_2) : c, t : e, s\rangle \triangleright \begin{cases} \langle c_1 : c, e, s\rangle & \text{si } t = \mathbf{tt} \\ \langle c_2 : c, e, s\rangle & \text{c.c} \end{cases}$$
$$\langle \texttt{LOOP}\,(c_1, c_2) : c, e, s\rangle \triangleright \langle c_1 : \texttt{BRANCH}\,(c_2 : \texttt{LOOP}\,(c_1, c_2)\,, \texttt{NOOP}) : c, e, s\rangle$$

  Para la última instrucción tenemos que c_1 sería algo así como la "condición" y c_2 el cuerpo del bucle.

Un programa en esta máquina abstracta puede alcanzar dos tipos de configuraciones: con la cola de instrucciones vacía (con lo que la ejecución ha sido exitosa) o con la cola no vacía (con lo que se ha alcanzado una configuración en la que la máquina no ha podido ejecutar la siguiente instrucción).

Propiedades

Esta máquina abstracta se rige, como acabamos de ver, por unas reglas muy similares a las de la semántica operacional de paso corto. Por esta razón, la mayoría de propiedades de esta última tienen equivalencia aquí (Ver ejercicios).

La función *ejecución*

El significado de una secuencia de instrucciones será una función *parcial* de estado a estado definida de la siguiente manera:

$$\mathcal{M} : \mathbf{Code} \to (\mathbf{State} \hookrightarrow \mathbf{State})$$
$$\mathcal{M}[\![c]\!]s = \begin{cases} s', & \text{si } \langle c, \varepsilon, s\rangle \triangleright^* \langle \varepsilon, e, s'\rangle \\ \texttt{undefinied}, & \text{c.c} \end{cases}.$$

Es decir, que un programa tendrá significado si se pueden ejecutar todas sus instrucciones.

Especificación de la traducción

En esta sección daremos la función que permite «compilar» del lenguaje **While** a las instrucciones de esta máquina abstracta.

Expresiones

Usaremos las siguientes funciones totales:

$$\mathcal{CA} : \mathbf{Aexp} \to \mathbf{Code}$$
$$\mathcal{CB} : \mathbf{Bexp} \to \mathbf{Code}$$

que se definen de manera composicional de la forma natural. Solo destacaré el uso de variables:

$$\mathcal{CA}[\![n]\!] = \texttt{PUSH}{-}n$$
$$\mathcal{CA}[\![x]\!] = \texttt{FETCH}{-}x$$

En los operadores, el operador de la derecha será el que está en la cima, seguido del de la izquierda y del operador en sí, en ese orden.

Instrucciones

Usaremos la siguiente función:

$$\mathcal{CS} : \mathbf{Stm} \to \mathbf{Code}$$

definida composicionalmente de la siguiente manera:

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

Función semántica

Con esta función de compilación ya definida, podemos dar el significado de una sentencia S a través de la siguiente función:

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

definida por composición entre la compilación a sentencias de la máquina abstracta y el significado de las instrucciones de esta misma máquina:

$$\mathcal{S}_{\text{am}}[\![S]\!] = (\mathcal{M} \circ \mathcal{CS})\,[\![S]\!]$$

Corrección

En esta sección buscamos demostrar la equivalencia entre la función semántica para las sentencias de **While** que hemos definido en la anterior sección con la semántica operacional que vimos en los anteriores capítulos.

Expresiones

Lema (Corrección expresiones aritméticas)
Para cualquier expresión aritmética a se cumple que

$$\langle \mathcal{CA}[\![a]\!], \varepsilon, s\rangle \triangleright^* \langle \varepsilon, \mathcal{A}[\![a]\!]s, s\rangle.$$

Además, todas las configuraciones intermedias tendrán una pila de ejecución no vacía.

Claramente, este lema también se da con las expresiones booleanas.

Instrucciones

Se podría ver la equivalencia con la semántica operacional de paso largo o paso corto. Como en este lenguaje ambas son equivalentes, daría lo mismo. En este caso, siguiendo el libro, lo veremos con el paso largo. La demostración será similar a la de la equivalencia entre las dos semánticas operacionales ya que al definir esta nueva función semántica hemos utilizado un significado muy parecido al paso corto.

Teorema (Equivalencia entre semánticas)
Para toda sentencia S del lenguaje **While**, se da la siguiente igualdad:

$$\mathcal{S}_{\text{ns}}[\![S]\!] = \mathcal{S}_{\text{am}}[\![S]\!]$$

Tema 5. Semántica denotacional

La diferencia fundamental de esta nueva semántica con las operacionales es que se consigue definir de manera completamente *composicional*.

Función semántica, valores semánticos y definición composicional

La semántica denotacional asocia significado a las instrucciones de **WHILE** a través de la siguiente función:

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

que se define de la siguiente manera para cada instrucción:

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

Como se puede observar, tenemos dos funciones auxiliares cond y FIX que debemos definir a su vez. Empezamos con cond que tiene como tipo

$$\begin{aligned}
\mathrm{cond} : (\mathbf{State} \to \mathbf{T}) \times (\mathbf{State} \hookrightarrow \mathbf{State}) \times (\mathbf{State} \hookrightarrow \mathbf{State}) \\
\to (\mathbf{State} \hookrightarrow \mathbf{State})
\end{aligned}$$

y se define como

$$\mathrm{cond}\,(p, g_1, g_2)\ s = \begin{cases} g_1\ s, & \text{si } p\ s = \mathbf{tt} \\ g_2\ s, & \text{si } p\ s = \mathbf{ff} \end{cases}.$$

Por otro lado, el tipo de FIX será

$$\mathrm{FIX} : ((\mathbf{State} \hookrightarrow \mathbf{State}) \to (\mathbf{State} \hookrightarrow \mathbf{State})) \to (\mathbf{State} \hookrightarrow \mathbf{State})$$

y su definición será

$$\mathrm{FIX}\ F = g \text{ tal que } F\ g = g.$$

Sin embargo, no todas las funciones tienen punto fijo o no es único por lo que tendremos que hacer un estudio más detallado para refinar la definición.

Teoría de puntos fijos. Dominios semánticos. Funciones continuas

Requerimientos para el punto fijo

Al ejecutar un bucle del lenguaje **WHILE** se nos presentan tres posibles casos:

- El bucle termina correctamente para cualquier posible estado
- El bucle cicla *localmente*, es decir, existen una cantidad *finita* de estados en los que el bucle no termina.
- El bucle cicla *globalmente*, es decir, existen una cantidad *infinita* de estados en los que el bucle no termina.

Los dos primeros casos no dan problemas ya que el punto fijo es único. Sin embargo, en el tercer caso podemos tener múltiples funciones que son punto fijo. Por esta razón, debemos encontrar la función g_0 punto fijo que cumpla la siguiente propiedad:

$$\forall g \text{ punto fijo de } F \Leftrightarrow g_0\ s = s' \Rightarrow g\ s = s', \text{pero no al revés.}$$

O en otras palabras, que g_0 contenga la *mínima cantidad de información* posible (lo que la hace la más general posible).

Conjuntos parcialmente ordenados

Definición: (Orden parcial)
Sean $g_1, g_2 \in \mathbf{State} \hookrightarrow \mathbf{State}$. Decimos que $g_1 \sqsubseteq g_2$ si, y sólo si,

$$\forall s, s' \in \mathbf{State},\ g_1\ s = s' \Rightarrow g_2\ s = s'$$

Es decir, toda la información contenida en g_1, también lo esta en g_2.

***Observación*:**
El conjunto $\mathbf{State} \hookrightarrow \mathbf{State}$ junto con $\sqsubseteq$ es un ejemplo de *conjunto parcialmente ordenado* ya que se cumplen las propiedades *reflexiva*, *transitiva* y *antisimétrica*.
Definición: (Elemento mínimo)
Llamaremos *elemento mínimo*, denotado por $\bot$, de un conjunto parcialmente ordenado $(D, \sqsubseteq)$ a aquel que cumpla que

$$\forall d \in D,\ \bot \sqsubseteq d.$$

Decimos que $\bot$ *no contiene información*.

Proposición: (Unicidad del elemento mínimo)
Si un $(D, \sqsubseteq)$ tiene elemento mínimo, entonces este es único.
Lema: (Elemento mínimo de $\mathbf{State} \hookrightarrow \mathbf{State}$)
El conjunto $(\mathbf{State} \hookrightarrow \mathbf{State}, \sqsubseteq)$ es parcialmente ordenado y la función definida como

$$\bot\ s = \text{undef},\ \forall s \in \mathbf{State}$$

es su elemento mínimo.

Con toda esta nueva información podemos actualizar la definición de FIX para que sea correcta:

- $\mathrm{FIX}\ F$ es un punto fijo, es decir, $F\ (\mathrm{FIX}\ F) = \mathrm{FIX}\ F$.
- $\mathrm{FIX}\ F$ es el punto fijo *mínimo*, es decir,

$$\text{si } F\ g = g \Rightarrow \mathrm{FIX}\ F \sqsubseteq q.$$

Conjuntos parcialmente ordenados completos

Hasta este momento hemos conseguido obtener la unicidad del punto fijo del funcional, sin embargo, todavía no sabemos si existe tal punto fijo. Por esta razón, necesitamos restringir todavía más el conjunto de funcionales con el que estamos tratando.

Definición: (Cotas superiores)
Sea $(D, \sqsubseteq)$ un conjunto parcialmente ordenado, $Y \subset D$ y $d \in D$. Diremos que d es un *cota superior* de Y si

$$\forall d' \in Y,\ d' \sqsubseteq d$$

Y será la cota superior *mínima* si

$$\forall d'' \text{ cota superior de } Y \Rightarrow d \sqsubseteq d''.$$

Esta cota superior mínima (*lub*) se denota, si existe, como $\bigsqcup Y$.

Definición: (Cadenas)
En las condiciones anteriores diremos que Y es una *cadena* si

$$\forall d_1, d_2 \in Y,\ d_1 \sqsubseteq d_2 \text{ ó } d_2 \sqsubseteq d_1.$$

Definición: (Orden y retículo completo por cadenas)

- Decimos que $(D, \sqsubseteq)$ es un *orden completo por cadenas* (*ccpo*) si para toda cadena $Y \subset D$, existe una cota superior mínima $\bigsqcup Y$.
- El conjunto D será un *retículo completo* si todo $Y \subset D$ tiene $\bigsqcup Y$.

Proposición: (Elemento mínimo de un *ccpo*)
Si $(D, \sqsubseteq)$ es un *ccpo*, entonces existe un elemento mínimo tal que

$$\bot = \bigsqcup \emptyset.$$

Proposición:
El conjunto $(\mathbf{State} \hookrightarrow \mathbf{State}, \sqsubseteq)$ es un *ccpo*.

Funciones continuas

En esta sección trataremos con *ccpo*s, por tanto, cada vez que aparezca un conjunto D, D', etc. asumiremos que es tal y sus operadores de orden serán, respectivamente, $\sqsubseteq$, $\sqsubseteq'$, etc.

Definición: (Monotonía)
Diremos que una función $f : D \to D'$ es monótona si

$$\forall d_1, d_2 \in D : d_1 \sqsubseteq d_2 \Rightarrow f\ d_1 \sqsubseteq' f\ d_2.$$

Proposición: (Propiedades monotonía)
Sea $f : D \to D'$ y $f' : D' \to D''$ ambas monótonas, entonces se cumple que

- La composición también es monótona:

$$f' \circ f : D \to D'' \text{ es monótona.}$$

- Se conservan las cadenas. Sea $Y \subset D$ una cadena, entonces $f\ Y = \{f\ d \mid d \in Y\}$ es una cadena de D' y

$$\bigsqcup' f\ Y \sqsubseteq' f\left(\bigsqcup Y\right).$$

***Observación*:**
A pesar de lo que se pueda pensar, la última propiedad de la monotonía no nos permite decir que el elemento mínimo de aplicar la f a todos los puntos de Y sea *igual* a aplicar f al elemento mínimo de Y. Solo nos interesaran las funciones que *sí* lo conserven. De aquí la siguiente definición.
Definición: (Continuidad)
Diremos que $f : D \to D'$ es *continua* si es monótona y para toda cadena no vacía $Y \subset D$ se cumple que

$$\bigsqcup' f\ Y = f\left(\bigsqcup Y\right).$$

Si además se cumple para la vacía, es decir, $\bot = f\ \bot$ diremos que es *estricta*.

Proposición: (Composición continua)
Sea $f : D \to D'$ y $f' : D' \to D''$ ambas continuas, entonces

$$f' \circ f : D \to D''$$

también es continua.

Teorema: (Caracterización del elemento mínimo de f)
Sea $f : D \to D$ una función continua tal que D (*ccpo*) tiene elemento mínimo $\bot$. Entonces

$$\text{FIX}\ f = \bigsqcup \{f^n\ \bot \mid n \in \mathbb{N}\}$$

es un elemento de D y, de hecho, es el punto fijo mínimo de f.

Observaciones

- En el anterior teorema hemos usado como notación $f^0 = \text{id}$ y $f^{n+1} = f \circ f^n$ para $n > 0$.

- Cabe destacar que, como $f^n\ \bot \sqsubseteq f^{n+1}\bot,\ \forall n \in \mathbb{N}$ (al ser continua) el conjunto $\{f^n \bot\}$ es realmente una cadena.

Resumen sección

Para concluir la sección enumeraré las propiedades que deseamos para manejar correctamente los puntos fijos en nuestra semántica:

- Será necesario que el dominio en el que trabajamos sea un *ccpo*.
- Las funciones que tratemos tendrán que ser *continuas* en ese *ccpo*.
- Usaremos los *puntos fijos mínimos* de estas funciones.

Corrección de la definición

Tras construir toda la estructura auxiliar, volvemos al propósito inicial del capítulo, definir *composicionalmente* y *correctamente* la semántica denotacional. Es por esto que ahora debemos asegurarnos que nuestro dominio sea un *ccpo* y que las funciones que utilizamos en la definición de la semántica sean continuas.

Proposición: (Dominio de funciones continuas)
Sean D y D' dos *ccpo*'s, si definimos $(D \to D', \sqsubseteq_F)$ tomando solo las funciones continuas entre D y D' y con

$$f_1 \sqsubseteq_F f_2 := \forall d \in D : (f_1\ d) \sqsubseteq (f_2 d)$$

tenemos que $(D \to D', \sqsubseteq_F)$ es un *ccpo*.

Proposición: (Continuidad de FIX)
El funcional $\mathrm{FIX} : (D \to D) \to D$ es continuo.
Proposición: (Continuidad del condicional)
La función $F : (\mathbf{State} \hookrightarrow \mathbf{State}) \to (\mathbf{State} \hookrightarrow \mathbf{State})$ definida como:

$$F\ g = \mathrm{cond}\,(p, g, g_0)\,,\ g_0 \in \mathbf{State} \hookrightarrow \mathbf{State}$$

es continua.

Proposición: (Continuidad funcional composición)
La función $F : (\mathbf{State} \hookrightarrow \mathbf{State}) \to (\mathbf{State} \hookrightarrow \mathbf{State})$ definida como:

$$F\ g = g \circ g_0,\ g_0 \in \mathbf{State} \hookrightarrow \mathbf{State}$$

es continua.

Y con esto tenemos el resultado que buscábamos.

Proposición: (Corrección semántica denotacional)
La función semántica $\mathcal{S}_{\mathrm{ds}}$ es total, es decir, estar correctamente definida.

Propiedades de la semántica

Definición: (Equivalencia semántica)
Decimos que dos sentencias S_1 y S_2 son semánticamente equivalentes si se cumple que

$$\mathcal{S}_{\mathrm{ds}}[\![S_1]\!] = \mathcal{S}_{\mathrm{ds}}[\![S_2]\!].$$

Equivalencia entre la semántica operacional y la denotacional

Teorema: (Equivalencia entre semántica operacional y denotacional)
Para toda sentencia del lenguaje **WHILE** tenemos que

$$\mathcal{S}_{\mathrm{ds}}[\![S]\!] = \mathcal{S}_{\mathrm{sos}}[\![S]\!].$$

Tema 7. Más sobre semántica denotacional

En este capítulo extendemos el lenguaje **WHILE** con *bloques* y *procedimientos* y daremos la semántica denotacional para esta extensión.

Entornos y memorias

En primer lugar, expandimos el lenguaje con bloques que tienen declaraciones de variables y procedimientos (igual que como vimos en el capítulo 3). A diferencia del capítulo 3 solo daremos el caso de *ámbito estáticos*.

Memoria y direcciones

Definición: (Direcciones)
Decimos que el dominio $\mathbf{Loc}$ representa las posibles localizaciones de las variables.
Definición: (Entorno de variables)
Definimos la siguiente familia de funciones como *entorno de variables* $\mathbf{Env}_V = \mathbf{Var} \to \mathbf{Loc}$ que asignan a cada variable, su dirección.
Definición: (Memoria)
Definimos la siguiente familia de funciones como *memoria* $\mathbf{Store} = \mathbf{Loc} \cup \{\mathrm{next}\} \to \mathbb{Z}$ que asignan a cada dirección, un valor.

De esta forma el estado como lo definíamos anteriormente será la composición de funciones del entorno de variables y la memoria. Esto motiva la siguiente definición.

Definición: (*Lookup*)
Definimos la función *lookup* con tipo $\mathbf{Env}_V \to \mathbf{Store} \to \mathbf{Store}$ a

$$\mathrm{lookup}\ env_V\ sto = sto \circ env_V.$$

Es decir, que dado un entorno de variables y una memoria, obtenemos el estado equivalente.

Con estas nuevas definiciones el tipo de la función semántica denotacional pasará a ser

$$\mathcal{S}'_{\mathrm{ds}} : \mathbf{Stm} \to \mathbf{Env}_V \to (\mathbf{Store} \hookrightarrow \mathbf{Store})$$

y su definición será

$$\begin{aligned}
\mathcal{S}'_{\mathrm{ds}}[\![x := a]\!]\ env_V\ sto &= sto\,[l \mapsto \mathcal{A}[\![a]\!]\,(\mathrm{lookup}\ env_V\ sto)] \\
&\quad \text{donde } l = env_V\ x \\
\mathcal{S}'_{\mathrm{ds}}[\![\texttt{skip}]\!]\ env_V &= \mathrm{id} \\
\mathcal{S}'_{\mathrm{ds}}[\![S_1\ ;\ S_2\ ]\!]\ env_V &= (\mathcal{S}'_{\mathrm{ds}}[\![S_2]\!]\ env_V) \circ (\mathcal{S}'_{\mathrm{ds}}[\![S_1]\!]\ env_V) \\
\mathcal{S}'_{\mathrm{ds}}[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]\ env_V &= \mathrm{cond}\,(\mathcal{B}[\![b]\!] \circ (\mathrm{lookup}\ env_V)\,, \mathcal{S}'_{\mathrm{ds}}[\![S_1]\!]\ env_V, \mathcal{S}'_{\mathrm{ds}}[\![S_2]\!]\ env_V) \\
\mathcal{S}'_{\mathrm{ds}}[\![\texttt{while}\ b\ \texttt{do}\ S]\!]\ env_V &= \mathrm{FIX}\ F \\
&\quad \text{donde } F\ g = \mathrm{cond}\,(\mathcal{B}[\![b]\!] \circ (\mathrm{lookup}\ env_V)\,, g \circ (\mathcal{S}'_{\mathrm{ds}}[\![S]\!]\ env_V)\,, \mathrm{id})
\end{aligned}$$

Declaraciones

Una vez actualizada la definición de las instrucciones básicas de **WHILE**, debemos dar una semántica para las nuevas construcciones, es decir, la declaración de variables y procedimientos así como los bloques y las llamadas. Empecemos por la declaración de variables. En primer lugar el tipo de esta función será

$$\mathcal{D}^V_{\mathrm{ds}} : \mathrm{Dec}_V \to (\mathbf{Env}_V \times \mathbf{Store}) \to (\mathbf{Env}_V \times \mathbf{Store})\,.$$

Es decir, dada una declaración de una variable, actualiza el estado. Su definición es, por tanto,

$$\begin{aligned}
&\mathcal{D}^V_{\mathrm{ds}}[\![\varepsilon]\!] = \mathrm{id} \\
&\mathcal{D}^V_{\mathrm{ds}}[\![\texttt{var}\ x := a;\ D_V]\!]\,(env_V, sto) = \\
&\qquad\qquad \mathcal{D}^V_{\mathrm{ds}}[\![D_V]\!]\,(env_V\,[x \mapsto l]\,, sto\,[l \mapsto v]\,[\mathrm{next} \mapsto \mathrm{new}\ l])\,, \\
&\quad \text{donde } l = sto\ \mathrm{next} \text{ y } v = \mathcal{A}[\![a]\!]\,(\mathrm{lookup}\ env_V\ sto)\,.
\end{aligned}$$

Con lo que al declarar cada variable le asignamos la dirección l (que es la siguiente disponible), le damos el valor correspondiente a esa dirección y actualizamos el siguiente hueco.

Por tanto, la semántica de un bloque sin procedimientos será

$$\begin{aligned}
\mathcal{S}'_{\mathrm{ds}}[\![\texttt{begin}\ D_V\ \varepsilon\ \texttt{end}]\!]\ env_V\ sto &= \mathcal{S}_{\mathrm{d}}[\![S]\!]\ env'_V\ sto' \\
&\quad \text{donde } \mathcal{D}^V_{\mathrm{ds}}[\![D_V]\!]\,(env_V, sto) = (env'_V, sto')\,.
\end{aligned}$$

Veamos, entonces, la declaración de procedimientos. Los entornos de procedimientos serán funciones de tipo

$$\mathbf{Env}_P = \mathbf{Pname} \to (\mathbf{Store} \hookrightarrow \mathbf{Store})$$

Es decir, vamos a asignar a cada procedimiento su *significado* directamente (es conveniente recordar que en el caso operacional, asignábamos a cada procedimiento la sentencia que ejecutaba, pero no su significado). Con esto, la función semántica tendrá tipo

$$\mathcal{D}^P_{\mathrm{ds}} : \mathbf{Dec}_p \to \mathbf{Env}_V \to \mathbf{Env}_P \to \mathbf{Env}_P$$

y su definición recursiva (para procedimientos no recursivos),

$$\begin{aligned}
&\mathcal{D}^P_{\text{ds}}[\![\varepsilon]\!] = \text{id} \\
&\mathcal{D}^P_{\text{ds}}[\![\texttt{proc}\ p\ \texttt{is}\ S;\ D_P]\!]\ env_V\ env_P = \mathcal{D}^P_{\text{ds}}[\![D_P]\!]\ env_V\ (env_P\ [p \mapsto g])\,, \\
&\qquad\qquad\qquad\qquad \text{donde } g = \mathcal{S}_{\text{ds}}[\![S]\!]\ env_V\ env_P.
\end{aligned}$$

Es decir, asignamos al procedimiento p el valor de su sentencia S.

La función semántica $\mathcal{S}_{\text{ds}}$

Por último, veamos como queda al final la función semántica. En primer lugar, su tipo tendrá que tener en cuenta los entornos de variable y de variables. Por tanto,

$$\mathcal{S}_{\text{ds}} : \mathbf{Stm} \to \mathbf{Env}_V \to \mathbf{Env}_P \to (\mathbf{Store} \hookrightarrow \mathbf{Store})$$

Por otro lado, la definición de esta nueva función semántica será igual a la dada por $\mathcal{S}'_{\text{ds}}$ únicamente añadiendo como parámetro el entorno de procedimientos env_P. Sin embargo, los bloques y las llamadas sí debemos especificarlas:

$$\begin{aligned}
&\mathcal{S}_{\text{ds}}[\![\texttt{begin}\ D_V\ D_P\ S\ \texttt{end}]\!]\ env_V\ env_P\ sto = \mathcal{S}_{\text{ds}}[\![S]\!]\ env'_V\ env'_P\ sto' \\
&\qquad\qquad \text{donde } \mathcal{D}^V_{\text{ds}}[\![D_V]\!]\ (env_V, sto) = (env'_V, sto') \\
&\qquad\qquad \text{y } \mathcal{D}^P_{\text{ds}}[\![D_P]\!]\ env'_V\ env_P = env'_P \\
&\mathcal{S}_{\text{ds}}[\![\texttt{call}\ p]\!]\ env_V\ env_P = env_P\ p
\end{aligned}$$

Procedimientos recursivos

Lo último que nos queda por ver es el significado de los procedimientos con llamadas recursivas en sus instrucciones. A diferencia de la forma que utilizábamos para tratar con este caso en la semántica operacional en la que actualizábamos el entorno cada vez que se realizaba una llamada recursiva, en este caso, la recursión será manejada *de una sola vez* cuando se realice la declaración del procedimiento. Esto guarda una clara relación con la iteración de los bucles por lo que, de la misma manera, utilizaremos el *punto fijo* de un funcional. En definitiva,

$$\begin{aligned}
&\mathcal{D}^P_{\text{ds}}[\![\varepsilon]\!] = \text{id} \\
&\mathcal{D}^P_{\text{ds}}[\![\texttt{proc}\ p\ \texttt{is}\ S;\ D_P]\!] env_V, env_P = \mathcal{D}^P_{\text{ds}}[\![D_P]\!]\ env_V\ (env_P\ [p \mapsto \text{FIX}\ F])\,, \\
&\qquad\qquad\qquad\qquad \text{donde } F\ g = \mathcal{S}_{\text{ds}}[\![S]\!]\ env_V\ env_P\ [p \mapsto g]\,.
\end{aligned}$$

Tema 8. Semántica axiomática

Aserciones de corrección parcial

La semántica axiomática se basa en el uso de *aserciciones de corrección parcial* que podemos definir como tuplas con la siguiente estructura

$$\{P\}\, S\, \{Q\}$$

donde P y Q son dos predicados y S una sentencia. El significado de esta tupla es el siguiente: «Si la precondición P se cumple en el *estado inicial* y la sentencia S *acaba* partiendo de ese mismo estado, el *estado final* cumplirá la postcondición Q».

***Observación*:**
Si la sentencia S *no* acaba, no podemos afirmar nada sobre el estado final en relación con la postcondición. Esto es lo que llamamos *corrección parcial* pues no sabemos nada sobre su *terminación*.

Como lenguaje para expresar los predicados podemos seguir dos “filosofías”, la *intensional* o la *extensional*. En este caso, usaremos la extensional en la que los predicados pueden ser considerados como funciones del tipo $\mathbf{State} \to \mathbf{T}$. Por ejemplo, cualquier expresión booleana describe un predicado $\mathcal{B}[\![b]\!]$. Tendremos la siguiente notación:

$$\begin{aligned} P_1 \wedge P_2 &\equiv P, \text{ donde } P\ s = (P_1\ s)\ \text{y}\ (P_2\ s) \\ P_1 \vee P_2 &\equiv P, \text{ donde } P\ s = (P_1\ s)\ \text{ó}\ (P_2\ s) \\ \neg P_1 &\equiv P, \text{ donde } P\ s = \neg\,(P_1\ s) \\ P_1\,[x \mapsto \mathcal{A}[\![a]\!]] &\equiv P, \text{ donde } P\ s = P_1\,(s\,[x \mapsto \mathcal{A}[\![a]\!]s]) \\ P_1 \Rightarrow P_2 &\equiv P_1\ s \Rightarrow P_2\ s,\ \forall s \in \mathbf{State} \end{aligned}$$

Considero necesario dar una breve aclaración sobre el significado del predicado asignación. Simplemente, este predicado será *verdadero* si el predicado sobre el que se aplica es verdadero una vez que se realice la sustitución indicada en la asignación.

Sistema de inferencia

Para describir la semántica de **WHILE** de manera axiomática será necesario dar lo que conocemos como *sistema de inferencia* que, al igual que la semántica operacional, consiste en un conjunto de axiomas y reglas a través de las tuplas que hemos descrito antes. Para cada instrucción del lenguaje tenemos una regla y son las siguientes:

$$\begin{aligned} [\mathrm{ass_p}] &:= \{P\,[x \mapsto \mathcal{A}[\![a]\!]]\}\ x := a\ \{P\} \\ [\mathrm{skip_p}] &:= \{P\}\ \texttt{skip}\ \{P\} \\ [\mathrm{comp_p}] &:= \frac{\{P\}\ S_1\ \{Q\}\,,\ \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1; S_2\ \{R\}} \\ [\mathrm{if_p}] &:= \frac{\{\mathcal{B}[\![b]\!] \wedge P\}\ S_1\ \{Q\}\,,\ \{\neg\mathcal{B}[\![b]\!] \wedge P\}\ S_2\ \{Q\}}{\{P\}\ \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{Q\}} \\ [\mathrm{while_p}] &:= \frac{\{\mathcal{B}[\![b]\!] \wedge P\}\ S\ \{P\}}{\{P\}\ \texttt{while}\ b\ \texttt{do}\ S\ \{\neg\mathcal{B}[\![b]\!] \wedge P\}} \\ [\mathrm{cons_p}] &:= \frac{\{P'\}\ S\ \{Q'\}}{\{P\}\ S\ \{Q\}},\ \text{donde } P' \Rightarrow P \text{ y } Q \Rightarrow Q'. \end{aligned}$$

Daré un par de aclaraciones que considero convenientes:

- La asignación, a primera vista, parece estar al revés. Sin embargo, esto no es así ya que si recordamos la definición de la notación para $P\,[x \mapsto \mathcal{A}[\![a]\!]]$ tenemos que será cierto el predicado si lo es P tras realizar la sustitución. Por lo tanto, lo que nos dice la tupla de la

asignación es que P se cumplirá si sigue siendo cierto al realizar la sustitución.

- La última regla nos permite *fortalecer* las precondiciones y *debilitar* las postcondiciones. Es decir, nos permite dar un caso más concreto que se sigue cumpliendo. De esta forma podemos, por ejemplo, encontrar un predicado que se cumpla para las dos postcondiciones de un `if`.

Notación: (Demostrabilidad de una tupla)
Cuando una tupla quede probada (al realizar todo su *árbol de inferencia*) lo denotaremos por:

$$\vdash_p \{P\}\, S\, \{Q\}$$

Equivalencia axiomática

Definición: (Equivalencia semántica)
Diremos que dos sentencias S_1 y S_2 son *equivalentes demostrablemente* si, $\forall P, Q$ precondiciones y postcondiciones tenemos que:

$$\vdash_p \{P\}\, S_1\, \{Q\} \Leftrightarrow \vdash_p \{P\}\, S_2\, \{Q\}$$

Corrección y completitud de la semántica axiomática

Definición: (Aserciones bajo semántica)
Una aserción de corrección parcial es válida bajo una semántica (por ejemplo, operacional de paso largo) si cumple que

$$\vDash_p \{P\}\, S\, \{Q\} \overset{\text{def}}{\Longleftrightarrow} \forall s \in \mathbf{State}\, [(P\ s = \mathbf{tt} \wedge \exists s' : \langle S, s\rangle \to s') \Rightarrow Q\ s' = \mathbf{tt}]$$

Es decir, que la postcondición se cumple para todos los estados en los que la precondición se cumple y la ejecución de S termina exitosamente.

Teorema:
Para toda aserción de corrección parcial $\{P\}\, S\, \{Q\}$ tenemos que

$$\vDash_p \{P\}\, S\, \{Q\} \Leftrightarrow \vdash_p \{P\}\, S\, \{Q\}\,.$$

O en tras palabras, la semántica axiomática es *completa* y *correcta*.