

Estructura de la materia a grandes rasgos:

Primera Parte: Lenguaje imperativo

Segunda Parte: Lenguaje aplicativo puro, y lenguaje aplicativo con referencias y asignación

Ejes de contenidos de la primer parte

- 1 Introducción a la sintaxis y la semántica de lenguajes
- 2 El problema de dar significado a la recursión e iteración
- 3 Un Lenguaje Imperativo Simple

Un Lenguaje Imperativo Simple

LIS

$\langle comm \rangle ::= \mathbf{skip}$
 $\langle var \rangle := \langle intexp \rangle$
 $\langle comm \rangle ; \langle comm \rangle$
 $\mathbf{if} \langle boolexp \rangle \mathbf{then} \langle comm \rangle \mathbf{else} \langle comm \rangle$
 $\mathbf{newvar} \langle var \rangle := \langle intexp \rangle \mathbf{in} \langle comm \rangle$
 $\mathbf{while} \langle boolexp \rangle \mathbf{do} \langle comm \rangle$

$\langle \text{intexp} \rangle ::= \langle \text{natconst} \rangle$	$\langle \text{boolexp} \rangle ::= \langle \text{boolconst} \rangle$
$\langle \text{var} \rangle$	$\langle \text{intexp} \rangle = \langle \text{intexp} \rangle$
$\langle \text{intexp} \rangle + \langle \text{intexp} \rangle$	$\langle \text{intexp} \rangle < \langle \text{intexp} \rangle$
$\langle \text{intexp} \rangle * \langle \text{intexp} \rangle$	$\langle \text{intexp} \rangle \leq \langle \text{intexp} \rangle$
$\langle \text{intexp} \rangle - \langle \text{intexp} \rangle$	$\langle \text{intexp} \rangle > \langle \text{intexp} \rangle$
$\langle \text{intexp} \rangle / \langle \text{intexp} \rangle$	$\langle \text{intexp} \rangle \geq \langle \text{intexp} \rangle$
$\langle \text{intexp} \rangle \% \langle \text{intexp} \rangle$	$\neg \langle \text{boolexp} \rangle$
$-\langle \text{intexp} \rangle$	$\langle \text{boolexp} \rangle \vee \langle \text{boolexp} \rangle$
	$\langle \text{boolexp} \rangle \wedge \langle \text{boolexp} \rangle$
$\langle \text{natconst} \rangle ::= 0 \mid 1 \mid 2 \mid \dots$	
$\langle \text{boolconst} \rangle ::= \mathbf{true} \mid \mathbf{false}$	

Características de LIS

- Hay sólo dos tipos de expresiones: enteras y booleanas
- Las expresiones enteras siempre se pueden evaluar, y su resultado es un entero. Todas las funciones primitivas (incluida la división) son funciones totales.
- Sólo posee variables que adoptan valores enteros. Luego la noción de **estado** se refleja en la siguiente definición:

Conjunto de estados: $\Sigma = \langle var \rangle \rightarrow \mathbf{Z}$

(la memoria posee infinitos lugares que siempre alojan un número entero).

Significado de los comandos de LIS

Funciones semánticas:

$$\llbracket _ \rrbracket^{intexp} \in \langle intexp \rangle \rightarrow \Sigma \rightarrow \mathbf{Z}$$

$$\llbracket _ \rrbracket^{boolexp} \in \langle boolexp \rangle \rightarrow \Sigma \rightarrow \{V, F\}$$

$$\llbracket _ \rrbracket^{comm} \in \langle comm \rangle \rightarrow \Sigma \rightarrow \Sigma_{\perp}$$

Primeras ecuaciones semánticas

$$\llbracket \text{skip} \rrbracket \sigma = \sigma$$

$$\llbracket v := e \rrbracket \sigma = [\sigma | v : \llbracket e \rrbracket \sigma]$$

$$\llbracket \text{if } e \text{ then } c \text{ else } c' \rrbracket \sigma = \text{if } \llbracket e \rrbracket \sigma \text{ then } \llbracket c \rrbracket \sigma \text{ else } \llbracket c' \rrbracket \sigma$$

Ejemplo:

$$\begin{aligned} \llbracket x := x - 1 \rrbracket \sigma &= [\sigma | x : \llbracket x - 1 \rrbracket \sigma] \\ &= [\sigma | x : \llbracket x \rrbracket \sigma - \llbracket 1 \rrbracket \sigma] \\ &= [\sigma | x : \sigma x - 1] \end{aligned}$$

Semántica de la composición de comandos

La ecuación $\llbracket c_0; c_1 \rrbracket \sigma = \llbracket c_1 \rrbracket (\llbracket c_0 \rrbracket \sigma)$ no está bien tipada.

Esto se soluciona acudiendo a funciones auxiliares de transferencia de control.

Si $f \in \Sigma \rightarrow \Sigma_{\perp}$, entonces definimos una nueva función $f_{\perp} \in \Sigma_{\perp} \rightarrow \Sigma_{\perp}$ de la siguiente manera:

$$f_{\perp} \sigma = f \sigma \qquad f_{\perp} \perp = \perp$$

Ecuación semántica para la composición:

$$\llbracket c_0; c_1 \rrbracket \sigma = \llbracket c_1 \rrbracket_{\perp} (\llbracket c_0 \rrbracket \sigma)$$

Semántica de **newvar**

Función “restauración de v según σ ”:

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

$$\llbracket \text{newvar } v := e \text{ in } c \rrbracket \sigma = (f_{v,\sigma})_{\perp} (\llbracket c \rrbracket [\sigma | v : \llbracket e \rrbracket \sigma])$$

Notación lambda para la función restauración:

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

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

Revisión: ¿Qué función computa haskell?

```
g :: Int -> Int
g n = if n == 0 then 0
      else if n == 1 then 1
           else g (n-2)
```

Es la menor solución en $\mathbb{Z} \rightarrow \mathbb{Z}_{\perp}$ de la ecuación recursiva:

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

La ecuación ER define una familia de funciones.

Una función es solución de ER si y sólo si es punto fijo de:

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

El menor punto fijo es:

$$\sup_{\mathbb{Z} \rightarrow \mathbb{Z}_\perp} (\{F^i \perp_{\mathbb{Z} \rightarrow \mathbb{Z}_\perp} \mid i \geq 0\})$$

donde $\perp_{\mathbb{Z} \rightarrow \mathbb{Z}_\perp}$ es el elemento mínimo de $\mathbb{Z} \rightarrow \mathbb{Z}_\perp$, es decir, la función que devuelve siempre \perp .

Fin de la revisión

Dificultades para dar significado a la iteración

El comando

while b **do** c

debe satisfacer la propiedad:

$$\begin{aligned} \llbracket \mathbf{while} \ b \ \mathbf{do} \ c \rrbracket \sigma &= \text{if } \llbracket b \rrbracket \sigma \text{ then } \llbracket c; \mathbf{while} \ b \ \mathbf{do} \ c \rrbracket \sigma \text{ else } \llbracket \mathbf{skip} \rrbracket \sigma \\ &= \begin{cases} \llbracket \mathbf{while} \ b \ \mathbf{do} \ c \rrbracket_{\perp} (\llbracket c \rrbracket \sigma) & \text{si } \llbracket b \rrbracket \sigma \\ \sigma & \text{si } \neg \llbracket b \rrbracket \sigma \end{cases} \end{aligned}$$

Semántica del **while**

Dificultad principal:

$$\llbracket \mathbf{while} \ b \ \mathbf{do} \ c \rrbracket \sigma = \begin{cases} \llbracket \mathbf{while} \ b \ \mathbf{do} \ c \rrbracket \perp \perp (\llbracket c \rrbracket \sigma) & \text{si } \llbracket b \rrbracket \sigma \\ \sigma & \text{si } \neg \llbracket b \rrbracket \sigma \end{cases} \quad (W)$$

no es dirigida por sintaxis

La propiedad W no es dirigida por sintaxis

La propiedad del **while** no puede ser tomada como definición.

Pero podemos tener la certeza de:

- el significado de $\llbracket \mathbf{while} \ b \ \mathbf{do} \ c \rrbracket$ es una función $\omega \in \Sigma \rightarrow \Sigma_{\perp}$
- tal función ω satisface la siguiente "ecuación funcional":

$$\omega \sigma = \begin{cases} \omega_{\perp}(\llbracket c \rrbracket \sigma) & \text{si } \llbracket b \rrbracket \sigma \\ \sigma & \text{si } \neg \llbracket b \rrbracket \sigma \end{cases} \quad (W)$$

Obtenemos entonces una ecuación similar a (ER)

Semántica del **while**

Sea

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

entonces, la ecuación del **while** (W) queda

$$\llbracket \mathbf{while} \ b \ \mathbf{do} \ c \rrbracket \sigma = F \llbracket \mathbf{while} \ b \ \mathbf{do} \ c \rrbracket \sigma$$

Aquí:

$$\llbracket \mathbf{while} \ b \ \mathbf{do} \ c \rrbracket \in \Sigma \rightarrow \Sigma_{\perp}$$

$$F \in (\Sigma \rightarrow \Sigma_{\perp}) \rightarrow (\Sigma \rightarrow \Sigma_{\perp})$$

Luego la ecuación también puede escribirse

$$\llbracket \mathbf{while} \ b \ \mathbf{do} \ c \rrbracket = F \llbracket \mathbf{while} \ b \ \mathbf{do} \ c \rrbracket$$

Semántica del **while** usando el TMPF

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

$$\llbracket \text{while } b \text{ do } c \rrbracket = \bigsqcup_{i=0}^{\infty} F^i \perp_{\Sigma \rightarrow \Sigma} \perp$$

para

$$F w \sigma = \begin{cases} w \perp \perp (\llbracket c \rrbracket \sigma) & \text{si } \llbracket b \rrbracket \sigma \\ \sigma & \text{si no} \end{cases}$$

Esta definición sí es dirigida por sintaxis.

Semántica denotacional de LIS

$$\llbracket \cdot \rrbracket \in \langle \text{comm} \rangle \rightarrow \Sigma \rightarrow \Sigma_{\perp}$$

$$\llbracket \text{skip} \rrbracket \sigma = \sigma$$

$$\llbracket v := e \rrbracket \sigma = [\sigma | v : \llbracket e \rrbracket \sigma]$$

$$\llbracket c_0; c_1 \rrbracket \sigma = \llbracket c_1 \rrbracket_{\perp} (\llbracket c_0 \rrbracket \sigma)$$

$$\llbracket \text{if } e \text{ then } c \text{ else } c' \rrbracket \sigma = \text{if } \llbracket e \rrbracket \sigma \text{ then } \llbracket c \rrbracket \sigma \text{ else } \llbracket c' \rrbracket \sigma$$

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

Semántica denotacional de LIS

$$\llbracket \text{while } b \text{ do } c \rrbracket = \bigsqcup_{i=0}^{\infty} F^i \perp_{\Sigma \rightarrow \Sigma} \perp$$

donde

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

¿Cómo calcular el menor punto fijo en $\Sigma \rightarrow \Sigma_{\perp}$?

Si w es la solución buscada (el menor punto fijo), entonces obtenemos el valor de $w \sigma$ de la siguiente manera:

- Si la cadena

$$F^0 \perp_{\Sigma \rightarrow \Sigma_{\perp}} \sigma, F^1 \perp_{\Sigma \rightarrow \Sigma_{\perp}} \sigma, F^2 \perp_{\Sigma \rightarrow \Sigma_{\perp}} \sigma \dots$$

adopta algún valor distinto de \perp , ese será el valor de $w \sigma$

- Si la cadena mencionada es siempre \perp , entonces $w \sigma = \perp$

Ejemplo

while $x \neq 0 \wedge x \neq 1$ **do** $x := x - 2$.

$$\llbracket \text{while } x \neq 0 \wedge x \neq 1 \text{ do } x := x - 2 \rrbracket = \bigsqcup_{i=0}^{\infty} F^i \perp_{\Sigma \rightarrow \Sigma} \perp$$

donde

$$F w \sigma = \begin{cases} w \perp \perp (\llbracket x := x - 2 \rrbracket \sigma) & \text{si } \llbracket x \neq 0 \wedge x \neq 1 \rrbracket \sigma \\ \sigma & \text{si no} \end{cases}$$

Entonces:

$$F w \sigma = \begin{cases} \sigma & \text{si } \sigma x \in \{0, 1\} \\ w [\sigma | x : \sigma x - 2] & \text{si no} \end{cases}$$

Se puede comprobar que

$$F^i \perp_{\Sigma \rightarrow \Sigma} \sigma = \begin{cases} [\sigma | x : \sigma x \% 2] & \text{si } \sigma x \in \{0, 1, \dots, 2 * i - 1\} \\ \perp & \text{si } \sigma x \notin \{0, 1, \dots, 2 * i - 1\} \end{cases}$$

Luego

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

Variables Libres

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

Variables asignables

$FA(\mathbf{skip})$	$= \emptyset$
$FA(v := e)$	$= \{v\}$
$FA(c_0; c_1)$	$= FA(c_0) \cup FA(c_1)$
$FA(\mathbf{if } b \mathbf{ then } c_0 \mathbf{ else } c_1)$	$= FA(c_0) \cup FA(c_1)$
$FA(\mathbf{while } b \mathbf{ do } c)$	$= FA(c)$
$FA(\mathbf{newvar } v := e \mathbf{ in } c)$	$= FA(c) - \{v\}$

Propiedades del LIS

Teorema de Coincidencia (TC)

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

¿Es correcto enunciarlo como sigue?

$$(\forall w \in FV(c). \sigma w = \sigma' w) \Rightarrow \llbracket c \rrbracket \sigma = \llbracket c \rrbracket \sigma'$$

Teorema de Coincidencia (TC)

1 $(\forall w \in FV(c). \sigma w = \sigma' w)$ implica,

o bien $\llbracket c \rrbracket \sigma = \perp = \llbracket c \rrbracket \sigma'$,

o bien $\llbracket c \rrbracket \sigma \neq \perp \neq \llbracket c \rrbracket \sigma'$ y

$$\forall w \in FV(c). \llbracket c \rrbracket \sigma w = \llbracket c \rrbracket \sigma' w$$

2 Si $\llbracket c \rrbracket \sigma \neq \perp$, entonces $\forall w \notin FA(c). \llbracket c \rrbracket \sigma w = \sigma w$.

Teorema de Renombre (TR)

No importa el nombre de las variables utilizadas en las declaraciones de variables locales (o sea las ligadas):

$$u \notin FV(c) - \{v\} \Rightarrow$$

$$\llbracket \text{newvar } u := e \text{ in } c/v \rightarrow u \rrbracket = \llbracket \text{newvar } v := e \text{ in } c \rrbracket$$

Sustituciones

Conjunto de las sustituciones:

$$\Delta = \langle var \rangle \rightarrow \langle var \rangle$$

Operador Sustitución:

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

$$\mathbf{skip}/\delta = \mathbf{skip}$$

$$(v := e)/\delta = (\delta v) := (e/\delta)$$

$$(c_0; c_1)/\delta = (c_0/\delta); (c_1/\delta)$$

$$(\mathbf{if } b \mathbf{ then } c_0 \mathbf{ else } c_1)/\delta = \mathbf{if } b/\delta \mathbf{ then } c_0/\delta \mathbf{ else } c_1/\delta$$

$$(\mathbf{while } b \mathbf{ do } c)/\delta = \mathbf{while } b/\delta \mathbf{ do } c/\delta$$

Operador sustitución para **newvar**

$(\mathbf{newvar} \ v := e \ \mathbf{in} \ c) / \delta =$

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

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

¿Cómo enunciar el Teorema de Sustitución para comandos?

¿si aplico la sustitución δ a c y luego evalúo en el estado σ , puedo obtener el mismo resultado a partir de c sin sustituir si evalúo en un estado que hace el trabajo de δ y de σ (en las variables libres de c)?

O sea, vale

$$(\forall w \in FV(c). \sigma(\delta w) = \sigma'w) \Rightarrow \llbracket c/\delta \rrbracket \sigma = \llbracket c \rrbracket \sigma'?$$

Los estados originales adoptarán eventualmente valores distintos en las variables que no ocurren libres en c , y en consecuencia los estados finales diferirán en las mismas.

Los estados originales adoptarán eventualmente valores distintos en las variables que no ocurren libres en c , y en consecuencia los estados finales diferirán en las mismas.

Debemos comparar $\llbracket c/\delta \rrbracket \sigma(\delta w)$ con $\llbracket c \rrbracket \sigma' w$.

Los estados originales adoptarán eventualmente valores distintos en las variables que no ocurren libres en c , y en consecuencia los estados finales diferirán en las mismas.

Debemos comparar $\llbracket c/\delta \rrbracket \sigma(\delta w)$ con $\llbracket c \rrbracket \sigma' w$.

Por ejemplo, si al programa $c = (x := x - 1; y := 2 * y)$ se le sustituye x por u e y por v (sustitución δ), entonces al ejecutar c/δ en un estado σ deberíamos obtener en u y v los valores que se obtienen al ejecutar c en un estado en donde x e y adopten los valores de σu y σv resp.

Problema del alias

Consideremos el programa $x := x - 1; y := 2 * y$,

y la sustitución $\delta x = \delta y = z$

El programa resultante será $z := z - 1; z := 2 * z$

Aunque $\sigma'x = \sigma z = \sigma'y$, se tiene

$$\llbracket z := z - 1; z := 2 * z \rrbracket_{\sigma z} \neq \llbracket x := x - 1; y := 2 * y \rrbracket_{\sigma'x}$$

$$\llbracket z := z - 1; z := 2 * z \rrbracket_{\sigma z} \neq \llbracket x := x - 1; y := 2 * y \rrbracket_{\sigma'y}$$

El problema surge porque δ no es inyectiva.

Teorema de Sustitución

Si δ es inyectiva sobre $FV(c)$ y

$\forall w \in FV(c). \sigma(\delta w) = \sigma' w$, entonces

o bien $\llbracket c/\delta \rrbracket \sigma = \perp = \llbracket c \rrbracket \sigma'$,

o bien $\llbracket c/\delta \rrbracket \sigma \neq \perp \neq \llbracket c \rrbracket \sigma'$ y

$$\forall w \in FV(c). \llbracket c/\delta \rrbracket \sigma(\delta w) = \llbracket c \rrbracket \sigma' w.$$