Lenguajes y Compiladores

Mati Steinberg y Miguel Pagano

5 de abril de 2024

Repaso

Cadena
Dado un poset $(X, \preccurlyeq)$ decimos que una secuencia $x_0, x_1, \ldots, x_n, \ldots$ es una ***cadena*** si $x_i \preccurlyeq x_{i+1}$, para todo $i \in \mathbb{N}$.

Decimos que una cadena es ***interesante*** si tiene infinitos elementos distintos.

Cadena
Dado un poset $(X, \preccurlyeq)$ decimos que una secuencia $x_0, x_1, \ldots, x_n, \ldots$ es una ***cadena*** si $x_i \preccurlyeq x_{i+1}$, para todo $i \in \mathbb{N}$.

Decimos que una cadena es ***interesante*** si tiene infinitos elementos distintos.

Definición
Un poset $(P, \preccurlyeq)$ es un ***predominio*** si toda cadena tiene supremo.

Caracterización
P es un predominio si todas las cadenas interesantes de P tienen supremo.

Corolario: $X_\perp$ es un predominio siempre.

Construcciones sobre predominios

1. Ya vimos que $X_\perp$ es un predominio.
2. Si P es un predominio, entonces $P_\perp$ y $P^\top$ también lo son.
3. Si tanto P como Q son predominios, también lo es $P \times Q$.
4. Si P es un predominio, entonces $A \to P$ también.

Espacio de funciones
Sea P es un predominio y A un conjunto. ¿Cómo es una cadena de funciones en $A \to P$?

Sea f_i una cadena de funciones, entonces podemos definir $\bigsqcup_{A \to P}(f_i)\, x = \bigsqcup_P(\{f_i\, x | i \in \mathbb{N}\})$.

Ejercicio: probar que está bien esa definición.

Dominios

Un predominio P es un ***dominio*** si tiene mínimo. Vamos a empacar la definición en una tupla de cuatro cosas: $D = (D, \sqsubseteq, \sqcup, \perp)$

Lema
Si D es un dominio, entonces $X \rightarrow D$ también lo es:

1. $\perp_{X \rightarrow D}$ está definido como:

$$\begin{aligned} \perp_{X \rightarrow D} \quad &: \quad X \rightarrow D \\ \perp_{X \rightarrow D}\, x \quad &= \quad \perp_D \end{aligned}$$

2. El supremo de una cadena de funciones ya lo definimos.

Funciones monótonas
Dados dos posets $(P, \leqslant)$, $(Q, \sqsubseteq)$ y una función $f : P \to Q$, decimos que f es ***monótona*** si $x \leqslant y$ implica $f\,x \sqsubseteq f\,y$.

Funciones

Funciones monótonas
Dados dos posets $(P, \leqslant)$, $(Q, \sqsubseteq)$ y una función $f : P \to Q$, decimos que f es ***monótona*** si $x \leqslant y$ implica $f\,x \sqsubseteq f\,y$.

Funciones continuas
Dados dos predominios $(P, \sqsubseteq, \sqcup)$, $(P', \sqsubseteq', \sqcup')$ y $f : P \to P'$, decimos que f es ***continua*** si, para toda cadena $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \ldots$, se da $f\,(\bigsqcup x_i) = \bigsqcup' (f\,x_i)$.

Funciones

Funciones monótonas
Dados dos posets $(P, \leqslant)$, $(Q, \sqsubseteq)$ y una función $f : P \to Q$, decimos que f es ***monótona*** si $x \leqslant y$ implica $f\,x \sqsubseteq f\,y$.

Funciones continuas
Dados dos predominios $(P, \sqsubseteq, \sqcup)$, $(P', \sqsubseteq', \sqcup')$ y $f : P \to P'$, decimos que f es ***continua*** si, para toda cadena $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \ldots$, se da $f\,(\bigsqcup x_i) = \bigsqcup'\,(f\,x_i)$.

Funciones estrictas
Dados dos dominios $(D, \sqsubseteq, \sqcup, \perp)$, $(P', \sqsubseteq', \sqcup')\perp'$ y $f : P \to P'$, decimos que f es ***estricta*** si se da $f\,\perp = \perp'$.

Extensión estrictas
Si $f : A \to B$, entonces definimos su ***extensión estricta***, $f_\perp : A_\perp \to B_\perp$

$$f_\perp\, x = \begin{cases} \perp & \text{si } x = \perp \\ f\, a & \text{si } x = a \end{cases}$$

Extensión estrictas
Si $f : A \to B$, entonces definimos su ***extensión estricta***, $f_\perp : A_\perp \to B_\perp$

$$f_\perp\, x = \begin{cases} \perp & \text{si } x = \perp \\ f\, a & \text{si } x = a \end{cases}$$

Extensiones estrictas
Si $f : P \to D$, entonces definimos su ***extensión estricta***, $f_{\perp\!\!\!\perp} : P_\perp \to D$

$$f_{\perp\!\!\!\perp}\, x = \begin{cases} \perp & \text{si } x = \perp \\ f\, a & \text{si } x = a \end{cases}$$

Lenguaje de Programación

Comandos

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

Expresiones

$\langle natconst \rangle ::= \ \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \ldots$

$\langle intexp \rangle ::= \ \langle natconst \rangle$

$\mid \quad \langle var \rangle$

$\mid \quad \text{-} \, \langle intexp \rangle$

$\mid \quad \langle intexp \rangle \oplus \langle intexp \rangle$

$\oplus \in \{+, -, *, /, \%, \mathbf{rem}\}$

$\langle boolconst \rangle ::= \ \mathbf{true} \mid \mathbf{false}$

$\langle boolexp \rangle ::= \ \langle boolconst \rangle$

$\mid \quad \neg \, \langle boolexp \rangle$

$\mid \quad \langle intexp \rangle \oslash \langle intexp \rangle$

$\mid \quad \langle boolexp \rangle \owedge \langle boolexp \rangle$

$\oslash \in \{<, \leqslant, =, \neq, \geqslant, >\}$

$\owedge \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$

Características de LIS

- Hay sólo dos tipos de expresiones: enteras y booleanas

Características de LIS

- Hay sólo dos tipos de expresiones: enteras y booleanas
- Todas las funciones primitivas (incluida la división) son funciones totales.

Características de LIS

- Hay sólo dos tipos de expresiones: enteras y booleanas
- Todas las funciones primitivas (incluida la división) son funciones totales.
- Como en la lógica de predicados, sólo hay variables enteras:

$$\Sigma = \langle var \rangle \to \mathbb{Z}$$

Características de LIS

- Hay sólo dos tipos de expresiones: enteras y booleanas
- Todas las funciones primitivas (incluida la división) son funciones totales.
- Como en la lógica de predicados, sólo hay variables enteras:

$$\Sigma = \langle var \rangle \to \mathbb{Z}$$

- El significado de una expresión entera (booleana), fijado un estado, es un entero (booleano).

Características de LIS

- Hay sólo dos tipos de expresiones: enteras y booleanas
- Todas las funciones primitivas (incluida la división) son funciones totales.
- Como en la lógica de predicados, sólo hay variables enteras:

$$\Sigma = \langle var \rangle \to \mathbb{Z}$$

- El significado de una expresión entera (booleana), fijado un estado, es un entero (booleano).
- El significado de un comando ***que termina***, fijado un estado, es un estado.

Características de LIS

- Hay sólo dos tipos de expresiones: enteras y booleanas
- Todas las funciones primitivas (incluida la división) son funciones totales.
- Como en la lógica de predicados, sólo hay variables enteras:

$$\Sigma = \langle var \rangle \to \mathbb{Z}$$

- El significado de una expresión entera (booleana), fijado un estado, es un entero (booleano).
- El significado de un comando ***que termina***, fijado un estado, es un estado.
- ¿Y el signifcado de un comando que ***NO*** termina?

Funciones semánticas

$$[\![\_]\!]^{intexp} \in \langle intexp \rangle \to (\Sigma \to \mathbb{Z})$$

Significado de los comandos de LIS

Funciones semánticas

$$[\![\_]\!]^{intexp} \in \langle intexp \rangle \to (\Sigma \to \mathbb{Z})$$

$$[\![\_]\!]^{boolexp} \in \langle boolexp \rangle \to (\Sigma \to \{V, F\})$$

Funciones semánticas

$$
\begin{aligned}
[\![\_]\!]^{intexp} &\in \langle intexp \rangle \to (\Sigma \to \mathbb{Z}) \\
[\![\_]\!]^{boolexp} &\in \langle boolexp \rangle \to (\Sigma \to \{V, F\}) \\
[\![\_]\!]^{comm} &\in \langle comm \rangle \to (\Sigma \to \Sigma_{\perp})
\end{aligned}
$$

Significado de los comandos de LIS

Funciones semánticas

$$
\begin{aligned}
[\![\_]\!]^{intexp} &\in \langle intexp \rangle \to (\Sigma \to \mathbb{Z}) \\
[\![\_]\!]^{boolexp} &\in \langle boolexp \rangle \to (\Sigma \to \{V, F\}) \\
[\![\_]\!]^{comm} &\in \langle comm \rangle \to (\Sigma \to \Sigma_{\perp})
\end{aligned}
$$

Observación: en el codominio de $[\![\_]\!]^{comm}$ aparece $\Sigma_{\perp}$ para dar cuenta de la no terminación.

Equivalencia de Expresiones y Comandos

Equivalencia de términos

Dados e, e' en la misma categoría sintáctica, se dice que e es equivalente a e', que escribimos como $e \equiv e'$, si y solo si

$$[\![e]\!] = [\![e']\!]$$

Equivalencia de Expresiones y Comandos

Equivalencia de términos
Dados e, e' en la misma categoría sintáctica, se dice que e es equivalente a e', que escribimos como $e \equiv e'$, si y solo si

$$[\![e]\!] = [\![e']\!]$$

En nuestro caso, es equivalente a decir que para todo $\sigma \in \Sigma$,

$$[\![e]\!]\sigma = [\![e']\!]\sigma$$

Equivalencia de Expresiones y Comandos

Equivalencia de términos
Dados e, e' en la misma categoría sintáctica, se dice que e es equivalente a e', que escribimos como $e \equiv e'$, si y solo si

$$[\![e]\!] = [\![e']\!]$$

En nuestro caso, es equivalente a decir que para todo $\sigma \in \Sigma$,

$$[\![e]\!]\sigma = [\![e']\!]\sigma$$

Por ejemplo:

Equivalencia de Expresiones y Comandos

Equivalencia de términos
Dados e, e' en la misma categoría sintáctica, se dice que e es equivalente a e', que escribimos como $e \equiv e'$, si y solo si

$$[\![e]\!] = [\![e']\!]$$

En nuestro caso, es equivalente a decir que para todo $\sigma \in \Sigma$,

$$[\![e]\!]\sigma = [\![e']\!]\sigma$$

Por ejemplo:

$$\mathbf{true} \equiv x = x$$

Equivalencia de términos
Dados e, e' en la misma categoría sintáctica, se dice que e es equivalente a e', que escribimos como $e \equiv e'$, si y solo si

$$[\![e]\!] = [\![e']\!]$$

En nuestro caso, es equivalente a decir que para todo $\sigma \in \Sigma$,

$$[\![e]\!]\sigma = [\![e']\!]\sigma$$

Por ejemplo:

$$\mathbf{true} \equiv x = x$$

$$2 * x \equiv x + x$$

Equivalencia de Expresiones y Comandos

Equivalencia de términos
Dados e, e' en la misma categoría sintáctica, se dice que e es equivalente a e', que escribimos como $e \equiv e'$, si y solo si

$$[\![e]\!] = [\![e']\!]$$

En nuestro caso, es equivalente a decir que para todo $\sigma \in \Sigma$,

$$[\![e]\!]\sigma = [\![e']\!]\sigma$$

Por ejemplo:

$$\mathbf{true} \equiv x = x$$

$$2 * x \equiv x + x$$

$$x := 0; y := 1 \equiv y := 1; x := 0$$

Ecuaciones Semánticas de Comandos

Comandos sencillos

$$[\![\textbf{skip}]\!]\,\sigma = \sigma$$

Ecuaciones Semánticas de Comandos

Comandos sencillos

$$\llbracket \textbf{skip} \rrbracket \sigma = \sigma$$
$$\llbracket v := e \rrbracket \sigma = [\sigma \,|\, v : \llbracket e \rrbracket \sigma]$$

Comandos sencillos

$$[\![\mathbf{skip}]\!]\,\sigma = \sigma$$

$$[\![v := e]\!]\,\sigma = [\sigma \,|\, v : [\![e]\!]\,\sigma]$$

$$[\![\mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c']\!]\,\sigma = \begin{cases} [\![c]\!]\,\sigma & \text{si } [\![b]\!]\,\sigma \\ [\![c']\!]\,\sigma & \text{si } \neg [\![b]\!]\,\sigma \end{cases}$$

Ecuaciones Semánticas de Comandos

Comandos sencillos

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

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

$$\llbracket \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c' \rrbracket \sigma = \begin{cases} \llbracket c \rrbracket \sigma & \text{si } \llbracket b \rrbracket \sigma \\ \llbracket c' \rrbracket \sigma & \text{si } \neg \llbracket b \rrbracket \sigma \end{cases}$$

Claramente **if** ... es un caso recursivo, pero no problemático.

Ecuaciones Semánticas de Comandos

Composición

La ecuación obvia

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

no tiene en cuenta que c_0 se puede colgar.

Composición

La ecuación obvia

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

no tiene en cuenta que c_0 se puede colgar.

De hecho, nuestro dominio semántico está bien pensado porque hay un error de tipos.

Composición

La ecuación obvia

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

no tiene en cuenta que c_0 se puede colgar.

De hecho, nuestro dominio semántico está bien pensado porque hay un error de tipos.

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

Ecuaciones Semánticas de Comandos

Bloque local

Después de ejecutar el cuerpo hay que restaurar el valor de la variable.

$$
\begin{array}{rcl}
rest_{v,\sigma} & : & \Sigma \to \Sigma \\
rest_{v,\sigma}(\sigma') & = & [\sigma' \,|\, v : \sigma v]
\end{array}
$$

Acá σ y v son parámetros fijos; el argumento es σ'.

Bloque local

Después de ejecutar el cuerpo hay que restaurar el valor de la variable.

$$\begin{array}{rcl} rest_{v,\sigma} & : & \Sigma \to \Sigma \\ rest_{v,\sigma}(\sigma') & = & [\sigma' \,|\, v : \sigma v] \end{array}$$

Acá σ y v son parámetros fijos; el argumento es σ'.

$$[\![\mathbf{newvar}\ v := e\ \mathbf{in}\ c]\!]\,\sigma = (rest_{v,\sigma})_{\perp\!\!\!\perp}([\![c]\!]\,[\sigma|v : [\![e]\!]\,\sigma])$$

Bloque local

Después de ejecutar el cuerpo hay que restaurar el valor de la variable.

$$\begin{array}{rcl} rest_{v,\sigma} & : & \Sigma \to \Sigma \\ rest_{v,\sigma}(\sigma') & = & [\sigma' \,|\, v : \sigma v] \end{array}$$

Acá σ y v son parámetros fijos; el argumento es σ'.

$$[\![\mathbf{newvar}\, v := e \,\mathbf{in}\, c]\!]\, \sigma = (rest_{v,\sigma})_{\perp\!\!\perp}([\![c]\!]\, [\sigma|v : [\![e]\!]\, \sigma])$$

Con notación lambda:

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

Ecuaciones Semánticas de Comandos

Ciclo

Evaluamos la guarda. Si es falsa, se terminó el programa.
Si es verdadera, ejecutamos el cuerpo y luego el ciclo entero.

Ecuaciones Semánticas de Comandos

Ciclo

Evaluamos la guarda. Si es falsa, se terminó el programa.
Si es verdadera, ejecutamos el cuerpo y luego el ciclo entero.

$$[\![\textbf{while}\ b\ \textbf{do}\ c]\!]\,\sigma = \begin{cases} \sigma & \text{si } \neg[\![b]\!]\sigma \\ [\![\textbf{while}\ b\ \textbf{do}\ c]\!]_{\perp\!\!\perp}([\![c]\!]\,\sigma) & \text{si } [\![b]\!]\sigma \end{cases}$$

Ciclo

Evaluamos la guarda. Si es falsa, se terminó el programa.
Si es verdadera, ejecutamos el cuerpo y luego el ciclo entero.

$$[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!]\,\sigma = \begin{cases} \sigma & \text{si } \neg[\![b]\!]\sigma \\ [\![\mathbf{while}\ b\ \mathbf{do}\ c]\!]_{\perp\!\!\perp}([\![c]\!]\,\sigma) & \text{si } [\![b]\!]\sigma \end{cases}$$

Problema: no es dirigida por sintaxis. Solución: definir

$$F_{b,c} : (\Sigma \to \Sigma_\perp) \to (\Sigma \to \Sigma_\perp)$$

y encontrar su menor punto fijo.

Ciclo

Evaluamos la guarda. Si es falsa, se terminó el programa.
Si es verdadera, ejecutamos el cuerpo y luego el ciclo entero.

$$[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!]\,\sigma = \begin{cases} \sigma & \text{si } \neg[\![b]\!]\sigma \\ [\![\mathbf{while}\ b\ \mathbf{do}\ c]\!]_{\perp\!\!\!\perp}([\![c]\!]\,\sigma) & \text{si } [\![b]\!]\sigma \end{cases}$$

Problema: no es dirigida por sintaxis. Solución: definir

$$F_{b,c} : (\Sigma \to \Sigma_\perp) \to (\Sigma \to \Sigma_\perp)$$

y encontrar su menor punto fijo.

$$F_{b,c}\, f\, \sigma = \begin{cases} \sigma & \text{si } \neg[\![b]\!]\sigma \\ f_{\perp\!\!\!\perp}([\![c]\!]\,\sigma) & \text{si } [\![b]\!]\sigma \end{cases}$$

Ecuación semántica para el ciclo

La ecuación

Sea $b \in \langle boolexp \rangle$ y $c \in \langle comm \rangle$.

$$
\begin{array}{rcl}
F_{b,c} & : & (\Sigma \to \Sigma_\perp) \to (\Sigma \to \Sigma_\perp) \\
F_{b,c}\, f\, \sigma & = & \begin{cases} \sigma & \text{si } \neg [\![b]\!] \sigma \\ f_{\perp\!\!\!\perp}([\![c]\!]\, \sigma) & \text{si } [\![b]\!] \sigma \end{cases}
\end{array}
$$

La ecuación

Sea $b \in \langle \mathit{boolexp} \rangle$ y $c \in \langle \mathit{comm} \rangle$.

$$\begin{array}{rcl} F_{b,c} & : & (\Sigma \to \Sigma_\perp) \to (\Sigma \to \Sigma_\perp) \\ F_{b,c}\, f\, \sigma & = & \begin{cases} \sigma & \text{si } \neg [\![b]\!] \sigma \\ f_{\perp\!\!\perp}([\![c]\!]\, \sigma) & \text{si } [\![b]\!] \sigma \end{cases} \end{array}$$

$$[\![\textbf{while}\ b\ \textbf{do}\ c]\!]\, \sigma = \left(\bigsqcup_{i \in \mathbb{N}} F^i \perp \right) \sigma$$

La intuición
El elemento k-ésimo de la cadena $F^i \perp$ representa la idea de ***desenrrollar*** k veces el while:

$$
\begin{aligned}
F^0 \perp &: \quad p_0 = \textbf{while true do skip} \\
F^{i+1} \perp &: \quad p_{i+1} = \textbf{if } b \textbf{ then } (c; p_i) \textbf{ else skip}
\end{aligned}
$$

Ecuación semántica para el ciclo

La intuición

El elemento k-ésimo de la cadena $F^i \perp$ representa la idea de ***desenrrollar*** k veces el while:

$$\begin{aligned} F^0 \perp &\quad : \quad p_0 = \textbf{while true do skip} \\ F^{i+1} \perp &\quad : \quad p_{i+1} = \textbf{if } b \textbf{ then } (c; p_i) \textbf{ else skip} \end{aligned}$$

Si con esas k veces ya llegamos al resultado, la cadena se mantiene estable.

Ecuación semántica para el ciclo

La intuición
El elemento k-ésimo de la cadena $F^i \perp$ representa la idea de ***desenrrollar*** k veces el while:

$$\begin{aligned} F^0 \perp &\quad : \quad p_0 = \textbf{while true do skip} \\ F^{i+1} \perp &\quad : \quad p_{i+1} = \textbf{if } b \textbf{ then } (c; p_i) \textbf{ else skip} \end{aligned}$$

Si con esas k veces ya llegamos al resultado, la cadena se mantiene estable.

Pero si las k veces no fueron suficientes, tenemos un resultado indefinido todavía.