UNA FORMALIZACIÓN DE LA SEMÁNTICA DE LOS LENGUAJES DE PROGRAMACION

Alejandro Garcés Calvelo[1]

Pilar Clara Espinosa Soteras[2]

Magali Matilla Belett[3]

Departamento de Ciencia de la Computación
Universidad de Oriente
Santiago de Cuba 90500
Cuba
[1]agarces_cu@yahoo.es
[2]pilar@csd.uo.edu.cu
[3]magali@csd.uo.edu.cu

Resumen

La mayoría de las implementaciones de lenguajes de programación, a partir de especificaciones formales, se basan en el enfoque operacional, el cual por su naturaleza algorítmica está muy cercano al desarrollo de prototipos. Sin embargo, este método de definición, incorpora una gran cantidad de detalles irrelevantes, lo cual lo hace inapropiado para la demostración de propiedades de programas.

En este trabajo, se presenta un método matemático para la definición de la semántica de los lenguajes de programación, el cual es a la vez abstracto y cercano a la práctica de la programación. Esta estrategia extiende una variante denotacional introducida por Iriarte, incorporando sistemas algebraicos para la especificación de las abstracciones de datos.

Palabras claves: Métodos formales, Especificación, Semántica Denotacional, Semántica Algebraica.

1. Introducción.

La especificación formal de los lenguajes de programación tiene una singular importancia para la comunidad de investigadores de la Ciencia de la Computación. A través de la misma se expone con rigurosidad una serie de detalles que, para el programador común, son transparentes o carecen de significado. Una definición caracteriza la instrumentación de cualquier rasgo del lenguaje, de forma tal que pueda ser usada en la prueba de propiedades de programas como la corrección, terminación, eficiencia y equivalencias entre ellos. La definición de cualquier lenguaje debe reunir ciertas cualidades, como son:

- Debe ser total o completa, en el sentido de que los aspectos esenciales del lenguaje estén rigurosamente definidos.
- Debe ser natural, clara y legible.
- Debe emplear una notación adecuada, comprensible y factible de ser implementada.

A partir de la década del 60 esto ha sido tarea primordial de los científicos de esta ciencia. En cuanto a la definición de la sintaxis se cuenta con métodos de definición ampliamente aceptados (ej. La notación BNF), para los cuales existen algoritmos cuyas implementaciones son muy eficientes. En cuanto a la semántica, sin embargo, no es hasta la década del 70 que adquiere un impulso trascendente.

La formalización de la semántica de los lenguajes de programación ha sido estudiada y elaborada desde posiciones diferentes. Fundamentalmente se distinguen tres enfoques: el operacional, el axiomático y el denotacional.

1.1. Enfoque operacional

En el enfoque operacional, el más cercano a la construcción de compiladores, la semántica del lenguaje se define en términos de una máquina abstracta. Es decir, a cada instrucción del lenguaje se le asocia un conjunto de operaciones de máquina.

Las operaciones describen cambios en el estado de la máquina caracterizado por el conjunto de valores de las variables y en el acto de control. Mediante el control se define el flujo de ejecución del programa, incluyendo la terminación o interrupción de su trabajo. A partir del estado donde termina el programa se determina el resultado de su ejecución. En la medida de cuan satisfactoria sea la abstracción del hardware, será su valor como guía para la estructuración general e instrumentación de compiladores.

En la actualidad existen dos tendencias principales de la semántica operacional:

- **La aproximación Máquina Abstracta**. Se especifica una vía de implementación del lenguaje sobre alguna computadora (idealizada) de bajo nivel, o en su defecto la traducción a un lenguaje de bajo nivel para el cual se tiene una semántica operacional. La ventaja de este enfoque es su aproximación a la implementación. Su gran desventaja es que lleva implícito una gran cantidad de detalles irrelevantes.

- **La Semántica Operacional Estructural**. Fue introducida por Plotkin a principios de la década de los ochenta [14]. Parte de la construcción de reglas de generación, para definir todas las relaciones relevantes. Estas reglas simbólicas trabajan directamente con la sintaxis del lenguaje. Las mismas, especifican las transiciones elementales de un programa, por inducción sobre su estructura. La Semántica Operacional Estructurada es un poco más abstracta y clara que la aproximación Máquina Abstracta. Una introducción más amplia, puede encontrarse en [10].

1.2. Enfoque Axiomático

La semántica axiomática descansa en una teoría formal cuyas fórmulas son ciertas aserciones o propiedades de los operadores e instrucciones del lenguaje. Los axiomas y reglas de deducción se utilizan para la demostración de propiedades de programas, por lo que constituyen el basamento fundamental de la construcción de verificadores [2].

Un conjunto de axiomas, junto con una descripción formal del programa, pueden permitir la deducción del resultado de la ejecución del programa en cualquier ambiente dado, así como inferir las propiedades más generales, tales como la corrección, terminación y equivalencia entre programas.

Este enfoque, por estar orientado a las necesidades de demostración de propiedades de programas, no constituye un camino natural ni directo a algún método de implementación. Existen numerosos libros en los que se describe con más nivel de detalle (Un buen ejemplo lo brinda Balcazar en [2]).

1.3. Enfoque denotacional.

El enfoque denotacional, que constituye el eslabón de enlace entre los enfoques anteriores, opera asignándole un significado (entidad matemática) al programa. A través de esta entidad se describe la dependencia funcional entre el resultado de la ejecución de un programa y sus datos iniciales. En particular se utiliza fuertemente la teoría de conjuntos, donde los programas se expresan como funciones donde la máquina no está presente. Este enfoque, también llamado matemático o funcional, es libre de los detalles de implementación.

La semántica denotacional, a diferencia de otros métodos, parte de considerar la interpretación como un conjunto de funciones totales incluyendo funciones que están bien definidas sobre toda función admisible como argumento (una clase de funcionales) y que aun pueden aplicarse a ellas mismas. Los inicios de este enfoque se enmarcan en los trabajos de Scott y Strachey [16].

Uno de los mayores retos, para los estudiosos del enfoque denotacional, es buscar variantes constructivas que permitan la implementación directa de prototipos de los lenguajes de programación. Precisamente, los trabajos con el enfoque de Iriarte [11][4], establecieron una estrategia singular, pues en ellos se establece la formalización de los conceptos partiendo de sus aspectos operacionales. Este trabajo extiende esta variante, incorporando especificaciones algebraicas para la definición de las abstracciones de datos.

2. Fundamentos de la semántica denotacional.

La principal idea del enfoque denotacional, es que el significado de un programa puede describirse en términos de la aplicación de funciones a sus argumentos. Así, no será necesario basar la semántica en máquinas de bajo nivel, ni siquiera en máquinas virtuales, tal como se hace en la semántica operacional. Además, con el uso de conjuntos y funciones, se podrá modelar la mayoría de los conceptos, tales como memoria, variables, tipos, etc. Esta modelación, permite alcanzar tan alto nivel de abstracción como el método axiomático.

2.1. Preliminares.

En este epígrafe se exponen un conjunto de notaciones y definiciones matemáticas, las cuales son necesarias para comprender los resultados principales de este trabajo.

2.1.1. Relaciones

Si $A_1 = A_2 = ... = A_n$ con $(n \geq 1)$, entonces $A_1 \times A_2 \times ... \times A_n$ se denotará por A^n. También se denotara por A^0 al conjunto $\{ <> \}$. Se define por A^* al conjunto $\bigcup_{n \geq 0} A^n$.

De igual forma, a^* que pertenece a A^*, denotará a una tupla ordenada de elementos de A.

Definición 1. (*Función*). Una función parcial (o simplemente función) f de A en B es una relación en $A \times B$, tal que si $<a,b>$ y $<a,c>$ pertenecen a f, entonces $b = c$. En este caso, se escribirá $f : A \rightarrow B$. A los conjuntos A y B se le llaman dominio (Dom(f)) y Codominio (Cdm(f)) de f respectivamente. ■

Definición 2. (*Función total*). Si $f : A \rightarrow B$ es una función definida para toda $a \in A$, es decir para toda a de A, existe un b de B tal que f(a) = b, entonces se dice que f es total.■

2.1.2. Sistemas algebraicos.

Definición 3. (*S-Conjunto*). Un S-conjunto C es una familia de conjuntos indexada por S, $C = \{C_s\}_{s \in S}$.■

Definición 4. (*Signatura Simple*). El par ordenado $\Sigma = <S, F>$ se denomina signatura simple, si se cumplen las siguientes condiciones:

i) Los conjuntos S y F no tienen elementos comunes.
ii) F es un $S^* \times S$ — conjunto, $F = \{F_{w,s}\}_{<w,s> \in S^* \times S}$.

S se denominan conjunto de géneros y los elementos del conjunto F son los símbolos de operaciones o de funciones. ■

Sea $w = w_1 \times ... \times w_n$, y $w_1, ..., w_n, s \in S$. Si $f \in F_{w,s}$, decimos que f es un símbolo de operación o función de dominio $w_1 \times ... \times w_n$ y género s. Si $f \in F_{w,s}$, también podrá denotarse por f: $w_1 \times ... \times w_n \rightarrow s$.

Sea $\Sigma = <S, F>$ una signatura simple. Definimos por $F_{\lambda,s} = \{f \mid f:\rightarrow s\}$, al conjunto de las operaciones constantes (o simplemente constantes) de género s.

Definición 5. (*Conjunto de términos de un género en una Signatura*). Sea $\Sigma=<S,F>$ una signatura simple y fijemos V como un S-conjunto de variables. El conjunto $Term_s(\Sigma)$ de términos (o expresiones) del género s en la signatura Σ, se define recurrentemente como sigue:

a) Toda variable x, tal que $x \in V_s$, es un término del género s en la signatura Σ (Es decir, $x \in Term_s(\Sigma)$).

b) Toda constante m, tal que $m \in F_{\lambda,s}$, es un término del género s en la signatura Σ.

c) Si $f \in F_{w1,\ldots,wn,s}$, y $e_1 \in Term_{w1}(\Sigma)$, … , $e_n \in Term_{wn}(\Sigma)$, entonces $f(e_1, \ldots, e_n)$ es un término del género s en la signatura Σ.

d) Todo término del género s en la signatura Σ, se obtiene solamente como resultado de la aplicación un número finito de veces de los pasos a), b) y c). ■

Note, que cuando se fija un conjunto vacío de variables del género s ($V_s=\varnothing$), entonces el conjunto $Term_s(\Sigma)$ sólo está compuesto por términos básicos.

Definición 6. (*Conjunto de términos de una signatura*). El conjunto de términos de la signatura $\Sigma=<S, F>$ está definido por $Term(\Sigma) = \bigcup_{s \in S} Term_s(\Sigma)$ ■

Definición 7. (*Enriquecimiento de una Signatura:*). Sean dos signaturas simples $\Sigma_1=<S_1,F_1>$ y $\Sigma_2=<S_2,F_2>$. Se dice que Σ_2 es un enriquecimiento de Σ_1, y lo denotamos por $\Sigma_1 \subseteq \Sigma_2$, si $S_1 \subseteq S_2$ y $F_1 \subseteq F_2$.

Definición 8. (*Sistema Algebraico*). El par ordenado $\Psi = <D_S, \Psi_F>$, es un sistema algebraico de la signatura $\Sigma=<S, F>$ (o simplemente una Σ- álgebra), si se cumplen las siguientes condiciones:

i. El portador D_S, es un S-conjunto. Es decir, $D_S = \{D_s\}_{s \in S}$.

ii. $\Psi_F = \{f_\Psi\}_{f \in F}$, y es tal que para todo $n \geq 0$ y $f \in F_{w1,\ldots, wn,s}$, se tiene la interpretación f_Ψ de f en Ψ, definida como f_Ψ: $D_{w1} \times \ldots \times D_{wn} \rightarrow D_s$. ■

2.2. Conceptos básicos del enfoque denotacional.

Los conceptos fundamentales del enfoque denotacional son el ***estado*** y el ***transformador de estados*** [11]. La información dinámica de cada concepto del lenguaje de programación (lo que cambia en ejecución) está caracterizada en el concepto de ***estado***. La información estática (lo que no cambia en tiempo de ejecución) se considera en la descripción de los ***transformadores de estado***. Estos dos conceptos permiten expresar las reglas semánticas de los lenguajes de programación.

El estado debe abarcar aquellos conceptos de máquina que cambian durante la ejecución de un programa (dirección de memoria, valor almacenado, etc.) y deben ser sólo los imprescindibles para describir completamente la semántica del lenguaje de programación.

Se define por estado propio [11] a la terna $\theta = < \theta', \tau, \eta >$, donde:

- θ' es el estado viejo (útil para formalizar la estructuración en módulos del programa),
- $\tau : V \rightarrow MA$ es el ambiente de variables, donde $V=\{x,x_1,...\}$ es el conjunto de identificadores de variables y MA el conjunto linealmente ordenado de direcciones de memoria. Si una variable x no está activa en el estado θ, entonces $\tau(x) = \gamma$.
- $\eta : MA \rightarrow D^+$, donde $D^+= D \cup \{\mu\} \cup \{false, true\}$y D es el conjunto de datos representables en la máquina. El símbolo μ ($\mu \notin D$) representa al valor indefinido. Por definición, se cumple que $\eta(\gamma)=\mu$. En lo adelante, sin perdida de generalidad, el conjunto no vacío D representará al conjunto definido de la siguiente forma:

  $D_0 = Z$, Z representa el conjunto de números enteros

  $D_n = \{<d_1, d_2, \ldots, d_n> \mid d_i \in D_0, \text{ para toda i } (1 \leq i \leq n)\}$

  $D = \bigcup_{n \geq 0} D_n$

Un ***transformador de estado*** es una función $s: \Theta^+ \rightarrow \Theta^+$, donde Θ es el conjunto de todos los estados propios [5] y a Θ^+ ($\Theta^+ = \Theta \cup \{\perp\}$) lo llamaremos el conjunto de estados. El elemento formal $\perp$ ($\perp \notin \Theta$) representa el estado indefinido, útil para indicar la no terminación. Al conjunto de todos los transformadores de estado lo denotaremos por *ST*. Se cumple por definición que si $s \in ST$, entonces $s(\perp) = \perp$. El elemento $\omega \in ST$, con la propiedad $\omega(\theta) = \perp$ para todo $\theta \in \Theta$, es un elemento distinguido en ST.

2.3. Significado de las expresiones predefinidas.

La especificación algebraica surge como un lenguaje para especificar estructuras de datos de una forma clara, sencilla, formal y no ambigua. A continuación se presenta un álgebra Ψ_0 que caracteriza, de manera general, los principios de construcción y evaluación de las expresiones enteras. El álgebra Ψ_0 corresponde a la realización del Tipo de Datos Abstracto (TDA) entero int, el cual constituye el único TDA primitivo en nuestra aproximación. La manipulación de diferentes tipos de datos abstractos primitivos no será abordada, pues no aporta información trascendente para los resultados que aquí se presentan. Descripciones más amplias, sobre la formalización de lenguajes con múltiples tipos, se pueden encontrar en otros trabajos de Garcés ([5], [6]).

Definición 9. (*Signatura del álgebra de los Enteros*). La signatura $\Sigma_0 = < S_0, F_0>$ del álgebra de los Enteros está compuesta por:

- El conjunto $S_0 = \{S_\infty, bool, int\}$ de géneros de la signatura.
- Sea $int^2= int \times int$. El conjunto $F_0 = \{ F_{\lambda,S\infty}, F_{\lambda,bool}, F_{\lambda,int}, F_{int^2,int}, F_{int^2,bool} \}$ de funciones de la signatura, que es tal que:
  - $F_{\lambda,S\infty} = \{\mu\}$
  - $F_{\lambda,bool} = \{false, true\}$
  - $F_{\lambda,int} = \{m \mid m \in D_0\}$, ($D_0$ definido como en epígrafe 2.2)
  - $F_{int^2,int} = \{ +, - \}$
  - $F_{int^2,bool} = \{ =, >, < \}$ ■

Fijemos el S_0-conjunto V_0, el cual se denomina Conjunto de Variables de la signatura del TDA entero. Definimos por $Term(\Sigma_0)=Term_{S\infty}(\Sigma_0) \cup Term_{bool}(\Sigma_0) \cup Term_{int}(\Sigma_0)$, al conjunto de términos de la signatura de enteros Σ_0, fijando el conjunto de variables V_0. Asumimos que cualquiera de los elementos de $V_0 = V_{0int} \cup V_{0bool}$, es tal que no pertenece a ninguno de los conjuntos S_0 y F_0.

Definición 10.(*Sistema Algebraico de Enteros*). El Sistema Algebraico (o Σ–álgebra) de Enteros Ψ_0 de la signatura $\Sigma_0 = < S_0, F_0 >$ está dado por el par $\Psi_0 = <D_{S0}, \Psi_{F0}>$, donde D_{S0} y Ψ_{F0} se definen como sigue:

a) El portador D_{S0} es un S_0-conjunto, dado por $D_{S0}=\{ D_{S\infty}, D_{bool}, D_{int} \}$. Donde, $D_{S\infty}= F_{\lambda,S\infty}$, $D_{bool} =F_{\lambda,bool}$ y $D_{int} =F_{\lambda,int}$.

b) Si $m \in F_{\lambda,s}$, con $s \in S_0$, entonces $m_\Psi( )= m$

c) Si $f \in F_{int}^2{}_{,int}$ y $m_1, m_2 \in F_{\lambda,int}$, entonces $f_\Psi(m_1, m_2) = m_{\Psi 1}( ) \otimes m_{\Psi 2}( )$.
Donde ⊗ es la interpretación de f en el álgebra tradicional de enteros.

d) Si $r \in F_{int}^2{}_{,bool}$ y $m_1, m_2 \in F_{\lambda,int}$, entonces $r_\Psi(m_1, m_2)= true_\Psi( )$, si $<m_1, m_2> \in ®$, donde ® es la interpretación de la relación r sobre los enteros. En caso contrario, $r_\Psi(m_1, m_2) = false_\Psi( )$. ■

Note que se ha hecho coincidir las representaciones sintácticas y semánticas. Esto, no resta generalidad a los resultados presentados. En lo adelante, también se denotará a la constante m( ) simplemente por m.

Definimos el Tipo de Datos Abstracto entero, denotado por <u>int</u>, como el par $<\Sigma_0, \Psi_0>$, el cual describe su interfaz e implementación respectivamente.

Definición 11. (R-valor *de expresiones enteros*). Sea el Sistema Algebraico de los Enteros $\Psi_0 = <D_{S0}, \Psi_{F0}>$ de la signatura $\Sigma_0 = < S_0, F_0 >$, el S_0-conjunto de variables V_0 y el estado θ ($\theta \in \Theta^+$), entonces el R-valor $I_0(e, \theta) \in D_{int} \cup D_{S\infty}$ de la expresión e ($e \in Term_{int}(\Sigma_0)$) en Ψ_0 para el estado θ, se define como sigue:

a) $I_0(e,\perp)= \mu$ para toda $e \in Term_{int}(\Sigma_0)$

b) $I_0(m,\theta)= m_\Psi( )$ para toda $m \in F_{\lambda,int}$ y todo estado propio $\theta \in \Theta$

c) $I_0(x,\theta)= \eta(\tau(x))$ para toda $x \in V_{0int}$ y todo estado propio $\theta = < \theta', \tau, \eta> \in \Theta$.

d) $I_0(f(e1,e2),\theta)= f_\Psi(I_0(e_1,\theta), I_0(e_2,\theta))$ para todo $f \in F_{int}^2{}_{,int}$ y $e_1,e_2 \in Term_{int}(\Sigma_0)$ y $\theta \neq \perp$ ■

Proposición 1. La definición de la interpretación de expresiones I_0 describe una función total $I_0 : Term_{int}(\Sigma_0) \times \Theta^+ \rightarrow D_0^+$. ■

2.4. Significado de las fórmulas.

El concepto de fórmula juega un papel fundamental en la formalización de las especificaciones y las instrucciones protegidas, destacándose entre estas últimas las alternativas y la selección por casos. Estas instrucciones se basan en la evaluación de una protección, cuyo resultado determina si la instrucción está abierta (si la protección se satisface) o está cerrada (si no se satisface).

Definición 12. (*Conjunto de fórmulas de* Σ_0). El conjunto Form(Σ_0) de fórmulas de la signatura Σ_0 se define recurrentemente como sigue:

a) Si $A \in Term_{bool}(\Sigma_0)$, entoces A es una fórmulas de la signatura Σ_0.
b) Si A_1 y A_2 son fórmulas de la signatura Σ_0, entonces $A_1 \wedge A_2$; $A_1 \vee A_2$; y $\neg A_1$ son fórmulas de la signatura Σ_0.
c) Toda fórmula se obtiene solamente como resultado de la aplicación un número finito de veces de los pasos a) y b). ■

Definición 13. (*Veracidad de fórmulas en el Sistema Algebraico de Enteros*). Sea el Sistema Algebraico de los Enteros $\Psi_0 = <D_{S0}, \Psi_{F0}>$ de la signatura $\Sigma_0 = < S_0, F_0 >$, el estado propio θ ($\theta \in \Theta$), el S_0-conjunto de variables V_0, la función R-valor I_0 y la fórmula $A \in Form(\Sigma_0)$, definimos la relación $\psi_0, \theta \models A$ (se lee "para el álgebra Ψ_0 y el estado θ, es verdadero A") como sigue:

a) $\psi_0, \theta \models$ true es válido.
b) $\psi_0, \theta \models$ false no es válido.
c) $\psi_0, \theta \models x$ es válido, si $\eta(\varepsilon(x)) =$ true.
d) $\psi_0, \theta \models x$ no es válido, si $\eta(\varepsilon(x)) \neq$ true.
e) Si $r \in F_{int^2,bool}$ y $e_1, e_2 \in Term_{int}(\Sigma_0)$, entonces
   - $\psi_0, \theta \models r(e_1, e_2)$ es válido si, y sólo si, $r_\Psi(I_0(e_1,\theta), I_0(e_2,\theta)) = true_\Psi( )$.
   - En caso contrario $\psi_0, \theta \models r(e_1, e_2)$ no es válido.
f) $\psi_0, \theta \models \neg A$ si, y sólo si, no es válido $\psi_0, \theta \models A$.
g) $\psi_0, \theta \models A1 \wedge A2$ si, y sólo si, son válidos $\psi_0, \theta \models A1$ y $\psi_0, \theta \models A2$.
h) $\psi_0, \theta \models A1 \vee A2$ si, y sólo si, es válida $\psi_0, \theta \models A1$ o $\psi_0, \theta \models A2$.
i) $\psi_0, \perp \models A$ no es válido. ■

2.5. Tipo de datos abstracto Vector.

En las secciones anteriores se ha definido completamente el Tipo de Datos Abstracto Primitivo **int**. En este epígrafe, se da la definición (por extensión) del Tipo de Datos Abstracto Genérico **Vector^N**, el cual caracteriza vectores (de dimensión variable) de enteros. El supraíndice N (N>0) define la dimensión del tipo vector. El polimorfismo paramétrico que se presenta, es relativamente simple y evita relaciones de subtipos. El principio escogido, para la construcción de abstracciones de datos, se basa en el concepto de Extensiones de Tipos de Wirth[17]. Algunos trabajos más recientes, pudieran servir de base a la generalización de esta clase de polimorfismo, destacándose entre ellos los de Abadi[1] y Leavens[13].

Sea $F_{Vector^N,int} = \{\pi_i \mid 1 \leq i \leq N\}$, que describe el conjunto de símbolos de proyecciones (operaciones), aplicables a cualquier tupla de un género $Vector^N$.

Definición 14. (*Signatura del TDA* Vector^N). La signatura $\Sigma_N = < S_N, F_N>$ del TDA **Vector^N** es un enriquecimiento de la signatura de entero Σ_0, y está definida como sigue:

a) El conjunto $S_N = S_0 \cup \{$ **Vector^N**$\}$.
b) El conjunto $F_N = F_0^+ \cup F_{\lambda,Vector^N} \cup F_{Vector^N,int}$.
   Donde, $F_{\lambda,Vector^N} = \{m^* \mid m^* \in D_N\}$, ($D_N$ definido como en epígrafe 2.2). ■

Fijemos el S_N-conjunto V_N (Conjunto de Variables de la signaturas del TDA Vector^N). Sea $Term(\Sigma_N)= Term_{S\infty}(\Sigma_N) \cup Term_{bool}(\Sigma_N) \cup Term_{int}(\Sigma_N) \cup Term_{Vector^N}(\Sigma_N)$, el conjunto de términos de la signatura Σ_N del TDA **Vector^N**, fijando el conjunto de variables V_N. Asumimos, igualmente, que los elemento de V_N, no son miembros de ninguno de los conjuntos S_N y F_N.

Definición 15. (*Sistema Algebraico del TDA* Vector^N). El Sistema Algebraico Ψ_N de la signatura $\Sigma_N = <S_N, F_N>$, para el TDA **Vector^N**, está dado por el par $\Psi_N = <D_{SN}, \Psi_{FN}>$, donde D_{SN} y Ψ_{FN} se definen como sigue:

a) El portador D_{SN} es un S_N-conjunto, dado por $D_{SN}=\{D_{S\infty}, D_{bool}, D_{int}, D_{Vector^N}\}$. Donde, $D_{S\infty}= F_{\lambda,S\infty}$, $D_{bool} =F_{\lambda,bool}$, $D_{int} =F_{\lambda,int}$ y $D_{Vector^N} = F_{\lambda,Vector^N}$.

b) Si $m \in F_{\lambda,s}$, con $s \in S_N$, entonces $m_\Psi( )= m$

c) Si $f \in F_{int^2,int}$ y $m_1, m_2 \in F_{\lambda,int}$, entonces $f_\Psi(m_1, m_2) = m_{\Psi 1}( ) \otimes m_{\Psi 2}( )$. Donde $\otimes$ es la interpretación de f en el álgebra tradicional de enteros.

d) Si $r \in F_{int^2,bool}$ y $m_1, m_2 \in F_{\lambda,int}$, entonces $r_\Psi(m_1, m_2)= true_\Psi( )$, si $<m_1, m_2> \in ®$, donde ® es la interpretación de la relación r sobre los enteros. En caso contrario, $r_\Psi(m_1, m_2) = false_\Psi( )$.

e) Si $\pi_i \in F_{Vector^N,int}$ y $m^*=<m_1,\ldots,m_N>$(con $m^* \in F_{\lambda,Vector^N}$), entonces $\pi_{i\Psi}(m^*)=m_i$. ■

Definición 16. (*Generalización del R-valor de expresiones enteros*). Sea el Sistema Algebraico $\Psi_N = <D_{SN}, \Psi_{FN}>$ de la signatura $\Sigma_N=<S_N, F_N>$ del TDA **Vector^N**, el estado θ ($\theta \in \Theta^+$) y el S_N-conjunto de variables V_N, entonces la generalización del R-valor, denotado por $I_N(e, \theta) \in D_{int} \cup D_{S\infty}$ de la expresión e ($e \in Term_{int}(\Sigma_N)$ en Ψ_N para el estado θ, se define como sigue:

a) $I_N(e,\theta)= \pi_{i\Psi}(m^*)$ si $e \equiv \pi_i(m^*)$ y $m^* \in F_{\lambda,Vector^N}$ y $\pi_i \in F_{Vector^N,int}$ y $\theta \in \Theta$.

b) $I_N(e,\theta)= \pi_{i\Psi}(\eta(\tau(x_N)))$ si $e \equiv \pi_i(x_N)$ y $x_N \in V_{NVector^N}$ y $\pi_i \in F_{Vector^N,int}$ y $\theta \in \Theta$.

c) $I_N(e,\theta)= I_0(e,\theta)$ en otro caso. ■

Proposición 2. La definición de la generalización del R-valor de expresiones I_N describe una función total $I_N : Term_{int}(\Sigma_N) \times \Theta^+ \rightarrow D_N^+$. ■

Definición 17. (*Extensión de la Interpretación (R-valor) de expresiones*). La extensión de la interpretación de expresiones I_N, la cual denotamos por I^+, se define como sigue:

a) $I^+(e, \theta) = I_N(E, \theta)$ si $e \in Term_{int}(\Sigma_N)$ y $\theta \in \Theta^+$

b) $I^+(e, \theta) = \mu$ en caso contrario ■

Definición 18. (*Conjunto de fórmulas de* Σ_N). El conjunto Form(Σ_n) de fórmulas de la signatura Σ_N se define recurrentemente como sigue:

a) Si A $\in$ $Term_{bool}(\Sigma_N)$, entoces A es una fórmulas de la signatura Σ_N.
b) Si A_1 y A_2 son fórmulas de la signatura Σ_N, entonces $A_1 \wedge A_2$; $A_1 \vee A_2$; y $\neg A_1$ son fórmulas de la signatura Σ_N.
c) Toda fórmula se obtiene solamente como resultado de la aplicación un número finito de veces de los pasos a) y b). ■

Note que chequear la veracidad de una fórmula A$\in$ Form(Σ_N) en un estado θ se reduce a probar que la relación ψ_N, θ |= A se satisface.

2.6. Significado de las construcciones de programación.

Una vez definido el comportamiento algebraico de nuestro modelo de datos, definir la semántica denotacional de un lenguaje de programación, radica en la construcción de las funciones (transformadores de estados) que dan significado a cada una de sus instrucciones. Para simplificar no tendremos en cuenta constructores de nuevos tipos. En esta sección, nuestro propósito es construir una función (total) semántica ***Sem*** que para cualquier instrucción S del lenguaje retorne el transformador de estado asociado, el cual conserve las propiedades algebraicas para los enteros. Más precisamente, se definirá la función:

$$Sem : Sts \rightarrow ST$$

Donde *Sts* es el conjunto de instrucciones del lenguaje. Es decir, para cada S $\in$ *Sts* se tiene que *Sem*(S) $\in$ *ST*.

La definición de la semántica mediante este enfoque imprime un alto contenido constructivo, lo cual hace que sea preferida por muchos autores para la implementación (por ejemplo, Reynolds[15]). Un estudio completo de la semántica denotacional de las estructuras de control de un Lenguaje de Programación al estilo PASCAL (con sólo un tipo) puede verse en [4]. En este trabajo sólo se presenta, por razones didácticas, la semántica denotacional de un conjunto reducido de instrucciones.

La sintaxis del conjunto de instrucciones del Lenguaje de Programación objeto de estudio está dada por:

S::= x:=e | x_N:= e_N | S_1; S_2 | <P(x,y); S_0> | P(z,t) | if A then S_1 else S_2

Donde, x y e son términos del género int; x_N y e_N son términos del género Vector; S, S_1, S_2 son instrucciones; A es una fórmula; <P(x,y); S_0> denota la definición del procedimiento P (no recursivo) con parámetros formales x (por referencia) e y (por valor); P(z,t) es el llamado al procedimiento P, donde z es una variable y t es una expresión.

2.6.1. Semántica denotacional de las instrucciones del Lenguaje de Programación.

La semántica denotacional de los lenguajes orientados a procedimientos ha sido abordada, en trabajos anteriores, desde diferentes puntos de vista (Ver [4], [5], [12], [15]). En esta sección, para facilitar la comprensión del método que se propone en este trabajo, mostramos la especificación formal sólo de algunas construcciones de programación.

En cada definición S, S1, S2 son instrucciones; A es una fórmula; x una variable entera; x_N es una variable vector; e es una expresión entera; e_N representa un vector, ya sea una variable o una constante vector; d un valor entero; α, α' direcciones de memoria.

a) Secuencia de instrucciones. S≡ S1; S2.

$$Sem(S)(\theta) = Sem(S2)(sem(S1)(\theta))$$

b) Asignación entera. S≡ x := e.

Para todo estado propio $\theta = < \theta', \tau, \eta> \in \Theta$

- $Sem(S)(\theta) = < \theta', \tau, \eta\{ I^+(e,\theta) / \tau(x) \}>$ si $I^+(e,\theta) \neq \mu$
- $Sem(S)(\theta) = \perp$ si $I^+(e,\theta) = \mu$

Donde:

- $\eta\{ d / \alpha' \}(\alpha) = d$ si $\alpha' = \alpha$
- $\eta\{ d / \alpha' \}(\alpha) = \eta(\alpha)$ si $\alpha' \neq \alpha$

c) Asignación de vectores. S≡ $x_N := e_N$.

Para todo estado propio $\theta = < \theta', \tau, \eta> \in \Theta$

i. Si $e_N \equiv z$ y (existe N, tal que $x_N, z \in V_{N\text{Vector}N}$), entonces

$Sem(S)(\theta) = < \theta', \tau, \eta\{ \eta(\tau(z)) / \tau(x_N) \}>$ si $x_N \neq \gamma$ y $\eta(\tau(z)) \neq \mu$
$Sem(S)(\theta) = \perp$ en caso contrario

ii. Si $e_N \equiv m^*$ y (existe N, tal que $x_N \in V_N$ y $m^* \in F_{\lambda,\text{Vector}N}$), entonces

$Sem(S)(\theta) = < \theta', \tau, \eta\{ m_\Psi( )/ \tau(x_N) \}>$ si $\tau(x_N) \neq \gamma$
$Sem(S)(\theta) = \perp$ en caso contrario

iii. En cualquier otro caso, $Sem(S)(\theta) = \perp$

d) Alternativa simple. S≡ if A then S1 else S2.

- $Sem(S)(\theta) = Sem(S1)(\theta)$ si $\psi_N, \theta \models A$
- $Sem(S)(\theta) = Sem(S2)(\theta)$ en caso contrario

e) S≡ P(z,t). Donde z es una variable y t una expresión.

Sea <P(x,y);S_0> la declaración del procedimiento P (no recursivo); "x" es un parámetro formal por referencia e "y" por valor; "x" y "z" son del mismo género; "y" y "t" son términos enteros.

Sean además los transformadores auxiliares $S^{x,y}_{z,t}$ y $\overline{S}$, tal que $\forall \theta \in \Theta$

$S^{x,y}_{z,t}(\theta) = <\theta, \tau\{\tau(z)/x, \alpha/y\}, \eta\{ I_N(t,\theta)/\tau(y)>$

$\overline{S}(\theta) = <\pi_1(\pi_1(\theta)), \pi_2(\pi_1(\theta)), \eta>$

Entonces la semántica denotacional del llamado S≡ P(t,z) al procedimiento <P(x,y);S_0> viene dada por:

$$Sem(S)(\theta) = \overline{S}(Sem(S_0)(S^{x,y}_{z,t}(\theta)))$$

Conclusiones

Sin lugar a dudas, la principal ventaja de la semántica denotacional, con respecto a otros enfoques, es la rigurosidad. La definición de los lenguajes de programación, con este enfoque, tiene en cuenta tanto la estructura lógica, su almacenamiento en memoria y la propia sintaxis del programa.

La metodología denotacional para la especificación de lenguajes de programación, presentada en este trabajo, es una muestra elocuente de ello. En la misma, el proceso de interpretación se establece a través de la dependencia funcional entre los estados, los cuales caracterizan la ejecución del programa en cada instante. Este proceso, por su alto carácter constructivo, permite instrumentaciones directas en los lenguajes más ampliamente usados.

Todos los conceptos tratados se definen exhaustivamente y con suficiente rigor. Se garantiza el carácter total de las funciones definidas. Esta última propiedad resulta de mucha utilidad para garantizar la terminación de las implementaciones de los algoritmos definidos.

La mayor contribución, de este trabajo, es su valor metodológico, por lo cual puede servir de guía a aquellos que se inicien en la difícil tarea de la especificación formal de los lenguajes de programación. Al igual que otras metodologías del enfoque denotacional, su mayor deficiencia radica en el nivel de detalles de especificación, lo cual puede afectar la eficiencia de las implementaciones computacionales de los diferentes algoritmos de chequeo semántico. Sin embargo, los avances en las tecnologías de la programación y el bajo costo para obtener hardware con muy altas prestaciones, hacen creer que el desarrollo de prototipos a partir de especificaciones formales, cada vez más es una posibilidad práctica.

Por otra parte, con el impacto de las tecnologías de información, quizás los puntos principales de atención en la formalización de lenguajes y sistemas, estén relacionados con la modelación basada en componentes[13][9], los sistemas multiagentes y sus lenguajes de comunicación[8].

Referencias

[1] Abadi, M.; Cardelle, L. and Burien, P.L. Formal parametric polymorphism. Theoretical Computer Science 121(1993), pp. 9 -58.

[2] Balcázar, J. L. Programación metódica. McGraw-Hill. 1994.

[3] Crocker, D. Making Formal Methods popular through Automated Verification. In International Joint Conference on Automated Reasoning (Short Papers). Siena, Italy. June 2001. pp. 21-24. http://www.dii.unisi.it/~ijcar/Shortpapers/crocker.pdf

[4] Díaz, J. Semántica denotacional de procedimientos y funciones, con consideraciones sobre sus ambientes activos. Tesis de doctorado. Universidad de Oriente. Cuba. 1992.

[5] Garcés, A. Una aproximación matemática a los tipos de datos. Revista Ciencias Matemáticas, Vol. 16, No.2 (1998), pp. 153-168.

[6] Garcés, A.; Matilla, M. Una metodología de validación exhaustiva basada en restricciones. Revista Ciencias Matemáticas, Vol. 19, No. 1 (2001), pp. 80-88.

[7] Garcés, A.; Matilla, M. Verificación de programas y especificaciones algebraicas. Proceedings of 2nd. International Conference on Automatic Control AUT' 2002. Santiago de Cuba. 2002. ISBN 84-699-9025-X

[8] Guerin, F. and Pitt, J. Denotational Semantic for Agent Communication Languages. In Proceedings of Fifth International Conference on Autonomous Agents AA´2001. Montreal. ACM Press. Pp. 497 – 504.

[9] Henderson, P. and Walters, B. Behavioral Analysis of Component-Based Systems. Information and Software Technology, Vol 43, 3 (2001), pp.161-169

[10] Hennesy, M. The semantics of programming Languages, John Wiley & Sons, New York. 1990.

[11] Iriarte, M. Semántica denotacional de procedimientos, funciones y apuntadores. Tesis de doctorado. San Petersburgo. 1983.

[12] Labra, J.; Cueva, J.; Luengo, M.; Cernuda, A. Y Joyanes, L. Comparación de Técnicas de Especificación Semántica de Lenguajes de Programación. In Proceedings of SISOFT 2001. Simposio Americano de Sistemas de Información e Ingeniería de Software en la Sociedad del Conocimiento. Santa Fé de Bogota. Agosto 2002. ACTAS ISBN 95897030-3-8.

[13] Leavens, G. and Dhara, K. Concepts of behavioral Subtyping and Sketch of their Extension to Component-Based Systems. In Gary Leavens and Murali Sitarman (editors), Foundations of Component-Based Systems. Cambridge University Press (2000). Chapter 6, pp. 113-135.

[14] Plotkin, G.D. A structural approach to operational semantics. Computer Science Dept., Aarthus Univesity, 1981.

[15] Reynolds, J. A literate, executable, denotational semantics of simple C++ declarations. Department of Computer Science. Iowa State University. TR# 93-15. 1993.

[16] Scott, D y Strachey, C. Towards a Mathematical Semantics for computer languages. In: Proceedings of the symposium on Computer and Automata (J.Fox ed.), Polytechnic Institute of Brooklyn, 1971.

[17] Wirth, N. Type Extensions. ACM Transactions on Programming Languages and Systems. Vol 10, No.2 (April 1988), pp. 204 - 214.