UN SISTEMA PARA LA VERIFICACIÓN DE PROGRAMAS

Alejandro Garcés Calvelo, Pilar Clara Espinosa Soteras, Ianisse Quinzán Suárez

Departamento de Ciencia de la Computación
Universidad de Oriente
Santiago de Cuba 90500
Cuba
agarces@sssn.ciges.inf.cu
pilar@csd.uo.edu.cu
ianisse@csd.uo.edu.cu

RESUMEN

Disponer de Métodos de Especificación Formal en el desarrollo del software ayuda en la representación y comprensión de los requerimientos de los sistemas, permite verificaciones rigurosas de las especificaciones y la implementación del software, a la vez que facilita la transición de la especificación al diseño y la implementación.
En este trabajo, se presenta la construcción de un intérprete y un verificador a partir de un método de especificación formal, para la definición de la semántica de los lenguajes de programación. El método denotacional constructivo descrito, es a la vez abstracto y cercano a la práctica de la programación, y su carácter relacional hace muy sencilla su implementación en el lenguaje de programación PROLOG.
El lenguaje de programación estudiado es extremadamente simple, lo cual responde al valor didáctico que añadido a este trabajo. Sin embargo, ello no resta generalidad a los resultados aquí presentados.

palabras claves: Especificación formal, verificación, semántica denotacional, semántica algebraica.

Introducción

Uno de los problemas principales de la Informática es la poca fiabilidad de los programas, debido a la gran variedad de errores que se puede incurrir en el proceso de desarrollo de programas. Desde la aparición de los modelos gramaticales, el análisis sintáctico ha contado con métodos muy potentes y fáciles de implementar. Sin embargo, la verificación semántica no ha contado con desarrollos similares.

Para reducir el riesgo de errores semánticos y lógicos, es útil tener una especificación formal del lenguaje de desarrollo, la cual no es más que una especificación expresada en un lenguaje cuyo vocabulario, sintaxis y semántica sean formalmente definidos mediante una base matemática, como la lógica formal. Aunque existen diferentes enfoques para la formalización de la semántica de los lenguajes de programación, los principales son los enfoques operacional, axiomático y denotacional.

En el enfoque operacional a cada instrucción del lenguaje se le asocia un conjunto de operaciones de máquina, que describen cambios en el estado de la máquina. El enfoque axiomático se establece sobre una teoría formal con fórmulas que son aserciones o propiedades de los operadores e instrucciones del lenguaje. El enfoque denotacional consiste en asignar al programa un significado, que es una entidad matemática; mediante la cual se establece la dependencia funcional entre el resultado al ejecutar un programa y sus datos de entrada.

La complejidad inherente a los modelos semánticos propuestos hasta la fecha, ha limitado el desarrollo de herramientas para la detección de errores lógicos. En general, las implementaciones de los lenguajes de programación sólo realizan el análisis sintáctico y semántico con el objetivo de lograr la ejecución de un programa, dejando al programador la responsabilidad de que el resultado del mismo se corresponda con sus requerimientos. Por esto, se hace necesario que se les brinden a los programadores herramientas que le permitan desarrollar software de mayor calidad, que lo ayuden en la detección de errores y en la puesta a punto de programas. No pocos esfuerzos han sido realizados, sin embargo el uso de métodos formales, con tal propósito, sigue siendo un terreno poco popular [2].

Entre estas herramientas se distinguen los intérpretes y los verificadores. Los intérpretes ejecutan las instrucciones del programa de una en una, con los que se facilita la ejecución paso a paso del programa, y el programador puede analizar el resultado de cada instrucción aisladamente. Los verificadores son herramientas que analizan el resultado que se obtendría al ejecutar el programa.

Precisamente, en este trabajo se especifican herramientas de desarrollo mediante un enfoque denotacional. El método de formalización descrito, se basa en importantes resultados de investigaciones desarrolladas en el Departamento de Computación de la Universidad de Oriente. Los mismos se originan en el desarrollo de las bases de una metodología denotacional constructiva, propuesta por Iriarte [8] y extendida en trabajos posteriores por Díaz [3] y Alyasin [1], los cuales incluyeron, respectivamente, el tratamiento completo de procedimientos y funciones recursivas (incluyendo parámetros-procedimientos) y la verificación de programas mediante este enfoque. Otras contribuciones de Garcés [4], [5], han establecido aproximaciones a la especificación de sistemas de tipos de datos.

1. Semántica denotacional de los lenguajes de programación.

En esta sección se exponen definiciones de la semántica denotacional, que son necesarias para comprender los resultados principales de este trabajo. Por limitaciones de espacio se han obviado otras importantes definiciones, que sin embargo son utilizadas a lo largo de todo el artículo. Para una mejor comprensión, sugerimos la lectura previa de otros trabajos de Garcés y Espinosa (en particular [5]).

1.1. Conceptos fundamentales del enfoque denotacional.

Los conceptos fundamentales del enfoque denotacional son el ***estado*** y el ***transformador de estados*** [7]. 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.

Se define por estado propio al cuarteto $\theta = < \theta', \tau, \eta, \nu>$, 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, 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:
  $D = Z$, Z conjunto de números enteros
- $\nu = <V', CA>$ es un par, donde V' conjunto de variables declaradas en el estado actual, $V' \subseteq V$. Esta componente es necesaria por el tratamiento local que se dará a las aserciones. CA representa el conjunto de aserciones a cumplir en el estado actual. En el cual cada aserción representa una condición.

Un ***transformador de estado*** es una función $s: \Theta^+ \rightarrow \Theta^+$, donde Θ es el conjunto de todos los estados propios [2] y a Θ^+ ($\Theta^+ = \Theta \cup \{\perp\}$) denota el conjunto de estados. El elemento formal $\perp$ ($\perp \notin \Theta$) representa el estado indefinido, útil para indicar la no terminación. El conjunto de todos los transformadores de estado es denotado 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.

1.2. Sistema de tipos

La semántica denotacional, aunque es mucho menos abstracta que la axiomática, presenta también limitaciones prácticas, las cuales están inducidas por el excesivo rigor matemático, la falta de modularidad y por carecer de formulaciones algorítmicas, que expresen más fielmente los aspectos operacionales. Precisamente, las especificaciones algebraicas surges como un método para especificar estructuras de datos de una forma clara, sencilla, formal y no ambigua [9]. Por demás, su espíritu operacional, le imprime un potente carácter algorítmico.

A continuación se presenta un álgebra Ψ 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 <u>int</u>, el cual constituye el único TDA primitivo en nuestra aproximación.

Definición 1. (*Signatura del álgebra de los Enteros*). La signatura $\Sigma = < S, F>$ del álgebra de los Enteros está compuesta por:

- El conjunto $S = \{S_\infty, bool, int\}$ de géneros de la signatura.
- El conjunto $F = \{ F_{\lambda,S\infty}, F_{\lambda,bool}, F_{\lambda,int}, F_{<int,int>,int}, F_{<int,int>,bool}, F_{<bool,bool>,bool}, F_{bool,bool} \}$ de funciones de la signatura, que es tal que:
  - $F_{\lambda,S\infty} = \{\mu\}$
  - $F_{\lambda,bool} = \{false, true\}$
  - $F_{\lambda,int} = \{m \mid m \in D\}$

- $F_{<int,int>,int} = \{ +, - \}$
- $F_{<int,int>,bool} = \{ =, >, < \}$
- $F_{<bool,bool>,bool} = \{ \wedge, \vee \}$
- $F_{bool,bool} = \{ \neg \}$ □

Fijemos el S-conjunto V [5], el cual se denomina Conjunto de Variables de la signatura. Definimos $Term(\Sigma) = Term_{S\infty}(\Sigma) \cup Term_{bool}(\Sigma) \cup Term_{int}(\Sigma)\}$, al conjunto de términos de la signatura de enteros Σ, fijando el conjunto de variables V. Asumimos que cualquiera de los conjuntos de V, es tal que su intersección con cualquiera de los conjuntos S y F es vacía.

Definición 2.(*Sistema Algebraico de Enteros*). El Sistema Algebraico (o Σ–álgebra) de Enteros Ψ de la signatura $\Sigma = < S, F >$ está dado por el par $\Psi = <D_S, \Psi_F>$, donde D_S y Ψ_F se definen como sigue:

a) El portador D_S es un S-conjunto, dado por $D_S=\{ D_{S\infty}, D_{bool}, D_{int} \}$.
   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 $s \in S$, entonces $m_\Psi( )= m$
c) Si $f \in F_{<int,int>,int}$ 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 $r \in F_{<int,int>,bool}$ 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 en el álgebra tradicional de enteros. En caso contrario, $r_\Psi(m_1, m_2) = false_{\Psi 0}( )$. □

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, que es denotado por int, como el par $<\Sigma_0, \Psi_0>$, el cual describe su interfaz e implementación respectivamente.

1.3. Semántica denotacional del lenguaje de programación.

Una vez definido el comportamiento algebraico de este 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 se tiene en cuenta constructores de nuevos tipos. Sin embargo se permite la introducción de aserciones en la instrucción de declaración de variables. Con las aserciones se facilita la verificación de programas, a la vez que se puede especificar las distintas dependencias entre variables y constantes enteras.

En esta sección, el propósito es construir una función (total) semántica ***Sem*** que para cualquier instrucción S del lenguaje retorne el transformador de estado asociado, el cual conserva 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. Un estudio completo de la semántica denotacional de las estructuras de control de un LP al estilo PASCAL (con sólo un tipo) puede verse en [3].

Semántica denotacional de las instrucciones del Lenguaje de Programación.

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; e es una expresión entera; d un valor entero; α, α' direcciones de memoria.

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

a) Secuencia de instrucciones. S≡ S1; S2;
   - $Sem(S)(\theta) = Sem(S2)(Sem(S1)(\theta))$

b) Iteración. S ≡ while A do S1 endwhile
   - $Sem(S)(\theta) = Sem(S2)(\theta)$ si $\psi_N, \theta \models A$, donde S2 = S1;S
   - $Sem(S)(\theta) = \theta$ en caso contrario

   donde $\psi_N, \theta \models A$ representa que para el álgebra Ψ_0 y el estado θ, es verdadera la fórmula A

c) S≡ call 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). P(x(A1),y(A2)), en el caso más general, denota el esquema, donde "x" es un parámetro formal por referencia, "y" por valor, y A1, A2 son aserciones asociadas a esas variables. Se cumple, que "x" y "z" son del mismo género; "y" y "t" son términos enteros.

   Sean además los transformadores auxiliares $S_{z,t}^{x,y}$ y $\overline{S}$, tal que:

   $S_{z,t}^{x,y}(\theta) = <\theta, \tau\{\tau(z)/x, \alpha/y\}, \eta\{ I_N(t,\theta)/\tau(y), <\{x,y\}, \{A1, A2\}>>$

   si $Vars(A1)=\{x\}$, $y \in Vars(A2)$ y $Vars(A2) \subseteq \{x,y\}$

   $\overline{S}(\theta) = <\pi_1(\pi_1(\theta)), \pi_2(\pi_1(\theta)), \eta, \pi_4(\pi_1(\theta))>$

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

   - $Sem(S)(\theta) = \theta 1$
     si $\forall A \in CA$ tal que $t \in Vars(A)$ y $\forall w \in A$, $\eta(\tau(w)) \neq \mu$, es válido $\psi_N, \theta_1 \models A$

     donde $\theta_1 = \overline{S}(Sem(S_0)(S_{z,t}^{x,y}(\theta)))$

   - $Sem(S)(\theta) = \perp$ en otro caso

   donde
   $\tau\{ \alpha /x_1 \}(x) = \alpha$ si $x_1 = x$
   $\tau\{ \alpha / x_1 \}(x) = \tau(x)$ si $x_1 \neq x$
   $I_N(t,\theta)$ representa el valor de la expresión t.

En esta especificación, están implícitas la declaración de variables y la asignación. Como puede verse, a través de la especificación de este lenguaje simple, el método propuesto se basa en principios algorítmicos bien estructurados. La modularidad de las reglas para la generación de estados hace que su implementación sea sencilla y compacta, tanto si se usan lenguajes orientados a procedimientos o lógicos.

1.4. Interpretación de programas.

Una vez definido los principios que rigen el enfoque denotacional propuesto, es muy simple la construcción de un algoritmo general de interpretación de cualquier programa de ese LP.

Procedimiento 1. Interpretación de programas.

Entrada: Programa P.
Salida: - Éxito si el programa es correcto semánticamente .
- Error si fallo semántico.
Método:
Paso1. Entrar Programa.
Paso2. Calcular $\theta = Sem(P)(\theta_0)$. Donde θ_0 representa el estado inicial.
Paso 3. Si $\theta = \perp$ entonces Error
si no Visualizar θ
Paso4. Fin.

Teorema: El procedimiento 1 cumple con la propiedad de Terminación Total.

2. Verificación de programas.

La metodología denotacional para la especificación de lenguajes de programación, al compararla con otros enfoques tiene la ventaja de la rigurosidad. Al utilizarla el proceso de verificación se convierte en una dependencia funcional entre predicados (conjuntos de estados) que describen las aserciones iniciales, parciales y finales de un programa. Este proceso permite instrumentaciones directas en los lenguajes más ampliamente usados.

2.1. Transformadores de predicados.

Como la ejecución de un programa es representable de forma matemática a través de conjuntos de pares (θ, θ'), que indican el estado inicial y el de terminación de su ejecución para ese estado inicial; se podría usar conjuntos de estados W ($W \subseteq \Theta^+$), a los que llamaremos predicados. Los conjuntos $CT= \{W \mid W \subseteq \Theta^+ \text{ y } \perp \notin W\}$, para definir clases de terminación y $CN=\{W^+ \mid W^+ \subseteq \Theta^+ \text{ y } \perp \in W^+\}$ en el caso de clases de no terminación, las cuales son útiles para la verificación de programas.

Definición 3. (Transformador de predicados directo). Para todo transformador de estados $s : \Theta^+ \rightarrow \Theta^+$, existe un transformador de predicados directo (o simplemente transformador de predicados), el cual se define como sigue:

$$\rho : P(\Theta^+) \rightarrow P(\Theta^+)$$

Donde para todo predicado $W \in P(\Theta^+)$, se tiene que $\rho(W) = \{ s(\theta) \mid \theta \in W\}$. Además, el conjunto de todos los transformadores de predicados se denota por *PT*. □

Definición 4. (Transformador de predicado de instrucciones). Sea S ($S \in Sts$) una instrucción. El transformador de predicados pmd para la instrucción S, está definido como sigue:

$$pmd : Sts \rightarrow PT,$$

que es tal que para todo $W \in P(\Theta^+)$, $pmd(S)(W) = \{ Sem(S)(\theta) \mid \theta \in W\}$. Por definición, se asume que si $W^+ \in CN$, entonces $pmd(S)(W^+) = \{\perp\}$ para toda instrucción $S \in Sts$ del lenguaje de programación. □

2.2. Corrección total de programas.

La validación de la corrección juega un papel primordial en la teoría y la práctica de la programación. Sin embargo, como aquí se ha dicho, ese proceso no es trivial, y además, la ausencia de metodologías de prueba asequibles para la mayoría de los programadores conspiran contra la utilización sistemática de técnicas de verificación.

Definición 5. *(Corrección total).* Sean A1 y A2 fórmulas. El programa P es totalmente correcto para la especificación (A1, A2) si se cumplen las siguientes propiedades:

a) $pmd(P)(W_{A1}) \in CT$, y
b) $pmd(P)(W_{A1}) \subseteq W_{A2}$

Donde $W_A = \{\theta \in \Theta^+ \mid \psi_N, \theta \models A \}$ para toda fórmula A □

En esta definición, la primera propiedad asegura la terminación del programa P para todo estado θ que satisface la precondición A1. La segunda propiedad, establece que el resultado esperado, expresado por la fórmula (postcondición) A2, es un debilitamiento del resultado del comportamiento real de P, que viene dado por $Pmd(P)(W_{A1})$.

Una vez definido los principios que rigen el enfoque denotacional propuesto, es muy simple la construcción de un algoritmo general de prueba de la corrección total de cualquier programa de ese LP. Para garantizar que el problema sea decidible, basta con asumir que el dominio primitivo D_0 es finito.

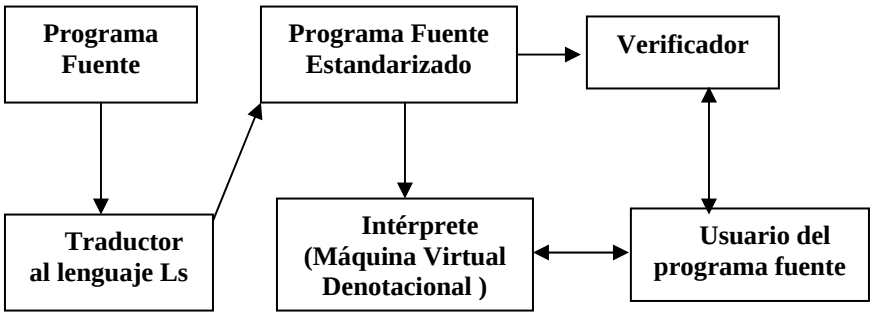
Procedimiento 2. (Algoritmo de corrección total)

Restricción de Entrada: El dominio D_0 es finito.
Entrada: Un programa P sin errores sintácticos, y una especificación (A1, A2).
Salida: Éxito, si se puede probar corrección total.
Método:
Paso 1. computar $W_{A1} = \{\theta \in \Theta^+ \mid \psi, \theta \models A1 \}$
// W_{A1} define los posibles estados de inicio del programa P.
Paso 2. $W_{A2} = \{\theta \in \Theta^+ \mid \psi, \theta \models A2 \}$
// W_{A2} define el comportamiento (conjunto de estados) esperado del // programa P.
Paso 3. computar $W = Pmd(P)(W_{A1})$
// W determina el comportamiento (conjunto de estados) real del programa P.
Paso 4. Si $(W \in CT)$ y $(W \subseteq W_{A2})$ entonces Éxito: El programa P es correcto.
// el comportamiento esperado $W_{A2,}$ es un debilitamiento del comportamiento
// real W del programa P
Paso 5. Fin.

Teorema 2. El procedimiento 2 cumple con la propiedad de terminación total.

3. Plataforma de desarrollo de programas.

La estructura funcional del proceso de interpretación y verificación de programas se muestra a continuación:



La ventaja principal, de este proceso, es la existencia de una Máquina Virtual Denotacional general, que implementa conceptos fundamentales de programación y que por tanto puede ser usada para la construcción de intérpretes de diferentes lenguajes. Basta con tener un traductor particular de cada lenguaje al código intermedio (lenguaje L_s), que es interpretado directamente por la Máquina Virtual Denotacional. Este código intermedio tiene como característica esencial que simplifica las instrucciones de programación, lo cual hace posible una implementación más cómoda y directa. También es importante destacar que a partir de cualquier lenguaje de programación orientado a procedimientos es posible construir un traductor hacia el código intermedio y este puede ser interpretado o verificado según se desee.

El enfoque relacional de la semántica denotacional garantiza que la implementación de esta formalización, sobre una plataforma de programación lógica, sea un proceso natural. Esto permite que algunos de los conceptos formulados teóricamente se vean omitidos o fusionados. Tal es el caso de la relación *variable-dirección* de *memoria- valor*, en cuya implementación se ha evitado el uso de las direcciones de memoria mediante artificios de programación.

La implementación en Prolog es realizada de forma sencilla y cómoda. La información dinámica se mantiene mediante asertos, a través de estos se almacenan las variables y valores de la misma, de esta misma forma se establece la pila de ejecución. Los transformadores de estado, son definidos a través de predicados, que según la instrucción a procesar y el estado, cambia al estado resultante de la ejecución de esa instrucción.

Los predicados fundamentales usados en esta implementación son **sem** y **verif**. A través de **sem** se expresa las reglas semánticas (transformadores de estados) para cada una de las instrucciones del lenguaje. Análogamente, el predicado **verif**, define las reglas de verificación (transformadores de predicados) para las construcciones de programación.

El predicado **sem**, analiza la información dinámica, guardada a través de asertos, y la instrucción a realizar y obtiene un nuevo estado de ejecución. El predicado **verif** se encarga de ejecutar cada posible estado de ejecución, auxiliándose para esto del predicado **sem** en el caso que sea necesario. Los posibles estados de ejecución se muestran a través de un árbol que se va ramificando cuando se generan varios estados de ejecución a partir de un estado; esto ocurre en el caso de entrada de un valor por parte del usuario. Cada nodo del árbol representa las variables cuyos valores son comunes para todos sus hijos, los hijos representan las bifurcaciones realizadas en el árbol; se generan cuando una variable puede tener más de un valor asociado. A partir de las hojas del árbol, se pueden obtener los estados de ejecución.

Se muestra un caso del uso de esos predicados, el caso específico de una declaración de variables

```
sem(int X & I):- atom(X), !, proc_actual(P), asserta(variable(X,0,ind,P,0)),
        asercion_correcta(I,X,P,1), !, asserta(asercion(I,P));
        throw(e('Es incorrecta la aserción ':I)).
verif(int X & I):- atom(X), !, sem(int X & I), proc_actual(P),
        retract(nodo(P,0,-1,L,H)), variable(X,0,ind,P,N,_),
        asserta(nodo(P,0,-1,[variable(X,0,ind,P,N,0)|L],H)).
```

El predicado **sem** se encarga de insertar la variable, luego de comprobar que no ha sido declarada previamente, en la base de datos de Prolog y verificar la corrección de la aserción asociada a esta instrucción.

El predicado **verif** se auxilia del predicado **sem** en la verificación de que puede ser ejecutada con éxito esa instrucción. Posteriormente inserta esa variable en la raíz del árbol pues en ese momento la variable tiene las mismas características para todos los posibles estados de ejecución.

Conclusiones

En este trabajo se ha propuesto una metodología denotacional para la formalización de la semántica de lenguajes de programación, desarrollándose sobre esta base un intérprete y un verificador.

El lenguaje de programación formalizado recoge construcciones fundamentales de programación, y aunque su sistema de tipos es relativamente simple (enteros, la incorporación de otros tipos de datos abstractos primitivos y derivados puede ser realizada usando procedimientos similares (definición de sus signaturas y Σ-álgebra asociadas) sin que provoque cambios sustanciales a los restantes resultados.

En este trabajo se incorpora al lenguaje el uso de aserciones, las cuales amplían la metodología usada. Las aserciones permiten que el programador pueda especificar las condiciones que deben cumplir sus variables. Son útiles pues nos dan una información adicional sobre el propósito de las variables y ayuda en la detección de inconsistencias entre los datos.

Es importante destacar la posibilidad de aplicar el intérprete y el verificador a otros lenguajes de programación con sólo tener un traductor entre el lenguaje y el código intermedio. El código intermedio recoge construcciones fundamentales de los lenguajes de programación.

Bibliografía

1. Alyasin, A. Semántica denotacional de transformadores de predicados como base para el diseño de verificadores de programas. Tesis de doctorado. Universidad de Oriente, Santiago de Cuba, 1994.
2. 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. Disponible en: http://www.dii.unisi.it/~ijcar/Shortpapers/crocker.pdf

3. Díaz, J. Semántica denotacional de procedimientos y funciones, con consideraciones sobre sus ambientes activos. Tesis de doctorado. Universidad de Oriente, Santiago Cuba, 1992.
4. Garcés, A., Matilla, M. Verificación de programas y especificaciones algebraicas. Proceedings of 2nd. International Conference on Automatic Control AUT' 2002. Santiago de Cuba. Ediciones Universidad de Alcalá de Henares, 2002.
5. Garcés, A., Espinosa, P y Matilla, M. Una formalización de la semántica de los lenguajes de programación. Revista Ingeniería Informática. Edición 9. Chile. Agosto 2003.
6. Garcés, A., Espinosa, P y Matilla, M. Verificación de programas: Una aproximación denotacional. Proceedings CISCI' 2003. Orlando, la Florida, 31 de Julio al 2 de Agosto, 2003.
7. Harm, Jörg, Lämmel, Ralf and Riedewald, Günter. LDL-Language Development Laboratory. In: *Nordic Workshop on Program Theory*, University of Oslo, Oslo, December 1996.
8. Iriarte, M. Semántica denotacional de procedimientos, funciones y apuntadores. Tesis de doctorado. San Petersburgo, 1983.
9. Meseguer, J. Conditional rewriting logic as a unified model of concurrency. Theoret. Comput. Sci. 96. 1992, pp. 73-155.
10. Wirth, N. Type Extensions. ACM Transactions on Programming Languages and Systems. Vol 10, No.2. April 1988, pp. 204 - 214.