Comparación de Técnicas de Especificación Semántica de Lenguajes de Programación

J. E. Labra Gayo[1], J. M. Cueva Lovelle[1], M. C. Luengo Díez[1], A. Cernuda del Río[1], and L. Joyanes Aguilar[2]

[1] Departamento de Informática, Universidad de Oviedo
C/ Calvo Sotelo S/N, 3307, Oviedo, Spain {labra,cueva,candi,guti}@lsi.uniovi.es
[2] Universidad Pontificia de Salamanca (Campus de Madrid)
Madrid-España, 28040
ljoyanes@dlsi.fpablovi.org

Abstract We compare the main approaches for the semantic specification of programming languages specifying the same set of three languages in the different formalisms and assessing the ideal requirements of non ambiguity, modularity, reusability, automatic prototype generation, proof facilities, flexibility and experience.

Keywords Programming Language Semantics, Modularity, Reusability, Proof, Specification

Resumen Se comparan las principales técnicas de especificación semántica de lenguajes de programación desarrollando la especificación de un mismo conjunto de tres lenguajes en los diferentes formalismos y valorando su comportamiento respecto a los requisitos ideales de no ambigüedad, legibilidad, modularidad, reusabilidad, generación automática de prototipos, facilidades para la demostración de propiedades, flexibilidad y experiencia.

Palabras clave Semántica de Lenguajes de Programación, Modularidad, Reusabilidad, Demostración, Especificación

1 Introducción

Lamentablemente, la especificación de la mayoría de los lenguajes de programación, desde Fortran [3] hasta Java [9], se realiza en lenguaje natural. Las descripciones en lenguaje natural acarrean gran cantidad de problemas como la falta de rigurosidad y la ambigüedad. Si el lenguaje de programación no está bien especificado es imposible demostrar propiedades básicas de los programas escritos en él. Tampoco es posible garantizar si un procesador de un lenguaje implementa dicho lenguaje ni obtener de forma automática dicho procesador a partir de la especificación.

Mientras que para la especificación sintáctica, la notación BNF se utiliza de forma prácticamente universal. En el caso de la especificación semántica existen diversas técnicas formales sin que ninguna de ellas haya alcanzado un alto grado de utilización. En este artículo se compara el comportamiento de las principales técnicas respecto a unos requisitos que se consideran fundamentales:

- No ambigüedad. La técnica debe facilitar la creación de descripciones rigurosas.
- *Demostración.* La herramienta debe tener una base matemática que permita la posterior demostración de propiedades de los programas escritos en el lenguaje especificado.
- *Prototipo.* Debería ser posible obtener prototipos ejecutables de los lenguajes que se diseñan de forma automática.
- Modularidad semántica. La descripción de un lenguaje debe poder realizarse de forma incremental. Al añadir nuevas características computacionales a un lenguaje, no debe ser necesario retocar las definiciones que no dependan de dichas características. Por ejemplo, al añadir variables a un lenguaje de expresiones aritméticas, las especificaciones realizadas para las expresiones aritméticas no deberían verse afectadas.

– *Reusabilidad*. La herramienta debe facilitar la reutilización de descripciones en diferentes lenguajes. Debería ser posible disponer de una librería de descripciones semánticas de características que pudiesen añadirse o eliminarse a un lenguaje de forma sencilla.
  Un ejemplo sería la descripción semántica de expresiones aritméticas. Existen multitud de lenguajes que admiten este tipo de expresiones y la descripción de estos lenguajes debería permitir incorporar tal componente de forma sencilla.
– *Legibilidad*. Las especificaciones deben ser legibles por personas con una formación heterogénea. La descripción del comportamiento de un lenguaje afecta a todas las personas que intervienen en el proceso de desarrollo de software y lo ideal es que todas esas personas pudiesen comprender la especificación.
– *Flexibilidad*. La herramienta descriptiva debe adaptarse a la gran variedad de lenguajes y familias de lenguajes existentes.
– *Experiencia*. La herramienta debe ser capaz de describir lenguajes reales. Algunas de las técnicas existentes son aplicables a lenguajes sencillos pero se resienten al ser aplicadas a algunos de los complejos lenguajes existentes en la actualidad. Debe existir, por tanto, cierta experiencia en la aplicación de la técnica de especificación a lenguajes de programación reales.

El fracaso de las técnicas existentes puede deberse a que no cumplen algunas de las características anteriores. En este artículo se repasan las principales técnicas y se realiza una valoración de su comportamiento respecto a los criterios mencionados.

La comparación utiliza como ejemplo la descripción de un sencillo lenguaje de forma incremental. Se parte de un lenguaje de expresiones aritméticas básicas, se añaden variables y posteriormente se incluyen órdenes imperativas y asignaciones.

El objetivo es ofrecer un marco común que permite observar las diferencias entre las diferentes técnicas. Por motivos de espacio, no será posible justificar las descripciones semánticas presentadas ni las valoraciones realizadas. En [15] presentamos este mismo estudio con mayor grado de detalle justificando las valoraciones tomadas.

2 Lenguaje Natural

A continuación se describen en lenguaje natural los tres lenguajes que se tomarán como ejemplo en el resto del artículo.

– El lenguaje $\mathcal{L}_1$ consiste únicamente en expresiones aritméticas simples. La sintaxis abstracta viene dada por:

  $expr$: *Const Integer* — constante numérica
  | $expr$ + $expr$ — suma

  $\mathcal{L}_1$ tiene un único tipo primitivo que representa los números enteros. Las expresiones de $\mathcal{L}_1$ denotan valores enteros.
– El lenguaje $\mathcal{L}_2$ añade variables al lenguaje anterior con la siguiente sintaxis

  $expr$: ... — definiciones anteriores de $\mathcal{L}_1$
  | *Var ident* — identificador

  Semánticamente, la evaluación de una expresión con identificadores depende del valor que el identificador x tenga en el momento de la evaluación. Aunque la modelización del entorno difiere según el formalismo semántico utilizado, para unificar las diferentes presentaciones, ς denotará un valor de tipo *State* que representará el contexto en el que se evalúan los identificadores. También se supone que se han definido las siguientes funciones:
  - $upd : State \to ident \to Value \to State$, $upd\ \varsigma\ x\ v$ devuelve el estado resultante de asignar a x el valor v en el estado ς.

- $lkp : State \to ident \to Value$, $lkp\ \varsigma\ x$ devuelve el valor de v en ς

– $\mathcal{L}_3$ incorpora órdenes formadas por secuencias y asignaciones

$$\begin{array}{lll} comm & : ident := expr & \text{— asignación} \\ & \mid comm\ ;\ comm & \text{— secuencia} \\ & \mid skip & \text{— comando vacío} \end{array}$$

3 Semántica Operacional

En 1981, G. Plotkin [20] desarrolla la *semántica operacional estructurada* en la cual se especifican las transiciones elementales de un programa mediante reglas de inferencia definidas por inducción sobre su estructura.

La especificación de $\mathcal{L}_1$ consta únicamente de expresiones aritméticas sencillas cuya evaluación producirá un valor entero. La semántica operacional define, una función de evaluación $\overset{eval}{\rightsquigarrow}: Expr \to \mathbb{N}$ que relaciona una expresión con el valor que denota. Las reglas de inferencia son

$$\frac{}{\langle Const\ n\rangle \overset{eval}{\rightsquigarrow} n} \qquad \frac{\langle e_1\rangle \overset{eval}{\rightsquigarrow} v_1 \quad \langle e_2\rangle \overset{eval}{\rightsquigarrow} v_2}{\langle e_1 + e_2\rangle \overset{eval}{\rightsquigarrow} v_1 + v_2}$$

Al añadir variables, la función de evaluación debe modificarse para poder consultar su valor en un contexto ς. El nuevo tipo será $\overset{eval}{\rightsquigarrow}: Expr \to State \to \mathbb{N}$. Es decir, el significado de una expresión ya no será un valor, sino una función que depende de un contexto $\varsigma \in State$. La regla de inferencia será

$$\frac{}{\langle Var\ v, \varsigma\rangle \overset{eval}{\rightsquigarrow} lkp\ v\ \varsigma}$$

Sin embargo, la modificación realizada afecta a las definiciones realizadas hasta el momento. Así la definición de la evaluación de expresiones aritméticas debe modificarse para que tenga la forma

$$\frac{\langle e_1, \varsigma\rangle \overset{eval}{\rightsquigarrow} v_1 \quad \langle e_2, \varsigma\rangle \overset{eval}{\rightsquigarrow} v_2}{\langle e_1 + e_2, \varsigma\rangle \overset{eval}{\rightsquigarrow} v_1 + v_2}$$

En $\mathcal{L}_3$ se añaden órdenes imperativas que pueden modificar un estado global. El significado de una orden imperativa toma una porción de programa y un estado y devuelve el resto de programa a ejecutar junto con el nuevo estado.

$$\overset{step}{\rightsquigarrow}: (Comm \times State) \to (Comm \times State)$$

Las reglas de inferencia serán

$$\frac{\langle e, \varsigma\rangle \overset{eval}{\rightsquigarrow} v}{\langle x := e, \varsigma\rangle \overset{step}{\rightsquigarrow} \langle skip, upd\ \varsigma\ x\ v\rangle} \qquad \frac{}{\langle \mathsf{skip}; c, \varsigma\rangle \overset{step}{\rightsquigarrow} \langle c, \varsigma\rangle} \qquad \frac{\langle c_1, \varsigma\rangle \overset{step}{\rightsquigarrow} \langle c_1', \varsigma'\rangle}{\langle c_1; c_2, \varsigma\rangle \overset{step}{\rightsquigarrow} \langle c_1'; c_2, \varsigma'\rangle}$$

La semántica operacional anterior se denomina también semántica de paso simple (*single-step semantics*) ya que se especifica la traza de ejecución de cada comando paso a paso. La semántica global de un comando puede obtenerse mediante el cierre transitivo de $\overset{step}{\rightsquigarrow}$.

La semántica natural [11] aparece como una variación de la semántica operacional que define transiciones globales en lugar de transiciones elementales. También se denomina *big-step semantics* o *semántica de gran paso*. Sin embargo, este formalismo tiene los mismos problemas de modularidad y reusabilidad. Recientemente, P. Mosses, inspirando por la semántica monádica modular, ha propuesto una semántica operacional estructurada modular [18]

4 Semántica Denotacional

En semántica denotacional el comportamiento del programa se describe modelizando los significados mediante entidades matemáticas básicas. El estudio denotacional de lenguajes de programación fue propuesto originalmente por C. Strachey en 1967 [21] utilizando λ-cálculo. Posteriormente D. Scott y C. Strachey colaboran en 1969 para establecer las bases teóricas utilizando teoría de deminios. La descripción del lenguaje $\mathcal{L}_1$ se realizará mediante la función $\mathcal{E} : Expr \to \mathbb{N}$.

$$\mathcal{E}[\![Const\ \ n]\!] = n \qquad \mathcal{E}[\![e_1 + e_2]\!] = \mathcal{E}[\![e_1]\!] + \mathcal{E}[\![e_2]\!]$$

En el caso de $\mathcal{L}_2$, al añadir variables al lenguaje, es necesario modificar la función de evaluación para que el valor de las variables se obtenga del contexto ς. La función de evaluación será $\mathcal{E} : Expr \to State \to \mathbb{N}$.

$$\mathcal{E}[\![Var\ v]\!]\varsigma = lkp\ v\ \varsigma$$

El cambio anterior obliga a modificar todas las definiciones anteriores para que tengan en cuenta el nuevo argumento:

$$\mathcal{E}[\![Const\ \ n]\!]\varsigma = n \qquad \mathcal{E}[\![e_1 + e_2]\!]\varsigma = \mathcal{E}[\![e_1]\!]\varsigma + \mathcal{E}[\![e_2]\!]\varsigma$$

Para capturar la semántica denotacional de las órdenes imperativas se utilizará una nueva función semántica $\mathcal{C} : Comm \to State \to State$. Es decir, la ejecución de un comando denota una función que, dado un estado inicial, devuelve un estado final.

$$\mathcal{C}[\![\mathsf{skip}]\!]\varsigma = \varsigma \quad \mathcal{C}[\![x := e]\!]\varsigma = upd\ \varsigma\ x\ (\mathcal{E}[\![e]\!]\varsigma) \quad \mathcal{C}[\![c_1; c_2]\!]\varsigma = \mathcal{C}[\![c_2]\!](\mathcal{C}[\![c_1]\!]\varsigma)$$

5 Semántica Axiomática

La semántica axiomática [8] consiste en definir una serie de reglas de inferencia que caracterizan las propiedades de las diferentes construcciones del lenguaje. Ha alcanzado gran popularidad en medios académicos, existiendo numerosos libros de texto en los que se describe con mayor detalle [6,1] La utilización de semántica axiomática para la descripción de lenguajes de programación se presenta en [24,10].

Se utilizan aserciones y ternas de Hoare basadas en precondiciones y postcondiciones. Una *aserción* es una fórmula P de lógica de predicados que toma un valor booleano en un contexto ς, el cual se denota como $[\![\mathrm{P}]\!](\varsigma)$ Una *terna de Hoare* es una expresión de la forma $\{P\}C\{Q\}$ donde P y Q son aserciones y C es una sentencia. P se denomina *precondición* y Q se denomina *postcondición*. Normalmente, este formalismo se emplea a posteriori, es decir, una vez definido el lenguaje, se estudia su semántica axiomática.

A continuación se exponen las reglas de inferencia de las órdenes imperativas.

$$\frac{}{\{P\}\mathsf{skip}\{P\}} \qquad \frac{}{\{P(x/e)\}\mathsf{x} := \mathsf{e}\{P\}} \qquad \frac{\{P\}c_1\{Q\} \quad \{Q\}c_2\{R\}}{\{P\}c_1; c_2\{R\}}$$

Con este formalismo se facilita la demostración de propiedades de los programas, además de la verificación a posteriori.

6 Semántica Algebraica

La semántica algebraica aparece a finales de los años 70, a partir de los trabajos desarrollados para la especificación algebraica de tipos de datos. Matemáticamente, se basa en el álgebra universal, también llamada lógica ecuacional. Una *especificación algebraica* permite definir una estructura matemática de forma abstracta junto con las propiedades que debe cumplir. Existen múltiples lenguajes de especificación algebraica como ASL, Larch, ACT ONE, Clear, OBJ [5], etc. Una especificación algebraica consiste en dos partes:

– La *signatura* define los géneros (*sorts*) que se están especificando, así como los símbolos de operaciones sobre dichos géneros y sus funcionalidades.
– Los *axiomas* son sentencias lógicas que definen el comportamiento de las operaciones. Habitualmente se utilizan una serie de ecuaciones, estableciendo una lógica ecuacional.

El siguiente módulo define una específicación algebraica de valores enteros.

obj *Int* **is**
  sort *Int*
  op $0 : \rightarrow Int$
  op *succ* _ : $Int \rightarrow Int$
  op _ + _ : $Int\ Int \rightarrow Int$
  vars x, y
  eq $x + 0 = x$
  eq $x + succ\ y = succ\ (x + y)$
endo

La especificación algebraica del lenguaje formado por espresiones aritméticas es similar a la semántica denotacional.

obj *Expr* **is pr** *Int* — **pr***Int* importa la especificación de enteros
  subsort *Int* $<$ *Expr* — Indica que un entero es una expresión
  op _ + _ : $Expr\ Expr \rightarrow Expr$
  op *eval* _ : $Expr \rightarrow Int$
  var $n\ e_1\ e_2 : Expr$
  eq $eval\ n = n$
  eq $eval\ (e_1 + e_2) = eval\ e_1 + eval\ e_2$
endo

La modelización de variables requiere una especificación algebraica del contexto de evaluación. La especificación de expresiones con variables será entonces

obj *VExpr* **is pr** *State*, **pr** *Expr*
  subsort $Var < Expr$
  op $eval : State\ Var \rightarrow Int$
  var $x : Var$, **var** $\varsigma : State$
  eq $eval\ \varsigma\ x = lkp\ \varsigma\ x$
endo

Las funciones de evaluación anteriores ya no sirven, puesto que no tenían en cuenta el estado y deben modificarse. La inclusión de órdenes imperativas en $\mathcal{L}_3$ se modeliza de la siguiente forma

obj *Comm* **is pr** *Expr*, **pr** *State*
  sort *Comm*
  op _ := _ : $Var\ Expr \rightarrow Comm$
  op _ ; _ : $Comm\ Comm \rightarrow Comm$
  op $exec : State\ Comm \rightarrow State$
  var $\varsigma : State$, **vars** $c_1\ c_2 : Comm$, **var** $x : Var$, **var** $e : Expr$
  eq $exec\ \varsigma\ (x := e) = upd\ \varsigma\ x\ (eval\ \varsigma\ e)$
  eq $exec\ \varsigma\ (c_1; c_2) = exec\ (exec\ \varsigma\ c_1)\ c_2$
endo

7 Máquinas de Estado Abstracto

Las máquinas de estado abstracto o *Abstract State Machines* (ASM) definen un algoritmo mediante una abstracción del estado sobre el que se trabaja y una serie de reglas de transición entre elementos de dicho estado. Originalmente, se denominaban *evolving algebras* y fueron desarrolladas por Y. Gurevich [7]. Las máquinas de estado abstracto pueden considerarse una evolución de la semántica operacional que trabajan con un nivel de abstracción cercano al algoritmo que se está definiendo.

Para la descripción semántica del lenguaje imperativo, se toma como modelo la especificación de Java [2] simplificándola para el ejemplo.

La especificación parte de un árbol que representa la sintaxis abstracta del programa. La semántica se considera como un recorrido sobre dicho árbol. En cada nodo, se ejecuta la tarea especificada y luego, el flujo de control prosigue con la siguiente tarea. Se utilizan una serie de funciones dinámicas para representar el estado de la computación. Una función dinámica es una función que puede modificarse durante la ejecución.

Para la especificación del lenguaje de expresiones aritméticas se utiliza una función dinámica

$val : Expr \rightarrow Int$

que almacena el valor intermedio a devolver. Las reglas de transición serán las siguientes

if *task* **is** *num* **then**
  $val(task) := num$
  proceed

if *task* **is** $e_1 + e_2$ **then**
  $val(task) := val(e_1) + val(e_2)$
  proceed

Para la inclusión de variables, se añade la función dinámica

$loc : Var \rightarrow Value$

que devuelve el valor actual de una variable. La regla de transición será:

if *task* **is** *var* **then**
  $val(task) := loc(var)$
  proceed

La regla de transición para órdenes imperativas será

if *task* **is** *skip* **then** *proceed*

if *task* **is** $x := e$ **then**
  $loc(x) := val(e)$
  proceed

En el caso de la estructura secuencial $c_1; c_2$ la regla de transición simplemente continua con c_1 tras haber modificado la función de ejecución para que, al finalizar c_1 continúe la ejecución con c_2.

8 Semántica de Acción

La *semántica de acción* fue propuesta por Mosses [19] con el fin de crear especificaciones semánticas más legibles, modulares y utilizables. La semántica de un lenguaje se especifica mediante *acciones*, que expresan computaciones. Sintácticamente, la notación de acciones tiene bastante libertad simulando un lenguaje natural restringido. Internamente, dicha notación se transforma en semántica operacional estructurada.

Existen acciones primitivas y combinadores de acciones. Un combinador de acciones toma una o más acciones y genera una acción más compleja. La modelización de los diferentes tipos de acciones se realiza mediante *facetas*. Existen varios tipos de facetas como la faceta básica, funcional, declarativa, imperativa y comunicativa, y cada faceta captura una determinada noción computacional.

A continuación se incluyen las semánticas utilizadas para el lenguaje aritmético

- evaluate _ :: Expr → action [giving Integer]

```
(1) evaluate ⟦ n:Numeral ⟧ =
        give n
(2) evaluate ⟦ E1:Expr "+" E2:Expr ⟧ =
        |evaluate E1
        |and
        |evaluate E2
        then give sum(given Integer#1,given Integer#2)
```

Al especificar $\mathcal{L}_2$ se añaden identificadores cuyo valor depende del contexto.

```
(1) evaluate ⟦ x : Ident ⟧ =
        |give value stored in cell bound to x
```

El lenguaje $\mathcal{L}_3$ podría especificarse de la siguiente forma

```
• execute _ :: Comm → Action [completing | storing]
(1) execute ⟦ "skip" ⟧ = complete
(2) execute ⟦ x:Ident ":=" E:Expr ⟧ = ||give the cell bound to x
                                      ||and
                                      ||evaluate E
                                      then
                                      |store the given Value#2 in the given cell#1
(3) execute ⟦ C1:Comm ";" C2:Comm ⟧ = execute C1 and then execute C2
```

Aunque pueda parecer que las descripciones en semántica de acción son realizadas en lenguaje natural. En realidad, se utiliza un lenguaje formal con una sintaxis bastante flexible que permite utilizar identificadores formados por varias palabras pero que son reconocidos por el analizador sintáctico sin que se produzcan ambigüedades.

9 Semántica Monádica Modular

La semántica monádica modular [17,16] surge a partir de la utilización de mónadas y transformadores de mónadas para describir una semántica denotacional modular.

Intuitivamente, una mónada separa una computación del valor devuelto por dicha computación. Con dicha separación se favorece la modularidad semántica, pudiendo especificarse características computacionales de los lenguajes de forma separada. Aunque el concepto de mónada deriva de Teoría de Categorías, en semántica de lenguajes, una mónada M puede considerarse como un tipo abstracto con dos operaciones

$$\begin{aligned} &\mathit{return} &&: \alpha \to \mathsf{M}\ \alpha \\ &(\gg\!=) &&: \mathsf{M}\ \alpha \to (\alpha \to \mathsf{M}\ \beta) \to \mathsf{M}\ \beta \end{aligned}$$

Existen diversos tipos de mónadas que añaden nuevas operaciones. Por ejemplo, la mónada que admite acceso a un entorno *Env* incorpora las operaciones

$$\begin{aligned} &\mathit{rdEnv} &&: \mathsf{M}\ \mathit{Env} \\ &\mathit{inEnv} &&: \mathit{Env} \to \mathsf{M}\ \alpha \to \mathsf{M}\ \alpha \end{aligned}$$

y la mónada que admite la modificación de un estado *State* incluye las operaciones

$$\begin{aligned} &\mathit{fetch} &&: \mathsf{M}\ \mathit{State} \\ &\mathit{set} &&: \mathit{State} \to \mathsf{M}\ \mathit{State} \end{aligned}$$

La interacción entre las diferentes operaciones se especifica mediante axiomas observacionales que delimitan su significado y permiten demostrar propiedades. Un problema de esta técnica es que, en

general, no es posible componer dos mónadas para obtener una nueva mónada. Una solución será la utilización de transformadores de mónadas que toman una mónada My la convierten en una nueva mónada M' añadiendo nuevas operaciones.

En la especificación del lenguaje $\mathcal{L}_1$ la función de evaluación tendrá el tipo $\mathcal{E} : Expr \to \mathsf{M}\ Int$, es decir, toma una expresión y devuelve una computación M que al ejecutarse devuelve un entero.

Las definiciones semánticas serán:

$$\begin{array}{ll}
\mathcal{E}[\![Const\ n]\!] & = return\ n \\
\mathcal{E}[\![e_1 + e_2]\!] & = \mathcal{E}[\![e_1]\!] \gg\!= \lambda v_1 \to \\
 & \quad \mathcal{E}[\![e_2]\!] \gg\!= \lambda v_2 \to \\
 & \quad return\ v_1 + v_2
\end{array}$$

En $\mathcal{L}_2$ se incluyen variables cuyo valor debe obtenerse de un entorno. Será necesario modificar la estructura computacional mediante un transformador de mónadas.

$$\mathcal{E}[\![Var\ x]\!] = rdEnv \gg\!= \lambda\varsigma \to return(lkp\ \varsigma\ x)$$

En el lenguaje $\mathcal{L}_3$ aparecen las órdenes imperativas. Para su modelización semántica se debe modificar de nuevo la estructura computacional, incorporando un transformador de mónadas que añada un estado actualizable. El tipo será $\mathcal{C} : Comm \to \mathsf{M}\ ()$, es decir, realiza computaciones, pero no devuelve ningún valor. Las definiciones semánticas serían:

$$\begin{array}{ll}
\mathcal{C}[\![skip]\!] & = return\ () \\
\mathcal{C}[\![c_1; c_2]\!] & = \mathcal{C}[\![c_1]\!] \gg\!= \lambda_ \to \mathcal{C}[\![c_2]\!] \\
\mathcal{C}[\![x\ :=\ e]\!] & = \mathcal{E}[\![e]\!] \gg\!= \lambda v \to \\
 & \quad fetch \gg\!= \lambda\varsigma \to \\
 & \quad set\ (upd\ \varsigma\ x\ v)
\end{array}$$

10 Semántica Monádica Reutilizable

La semántica monádica reutilizable añade conceptos de programación genérica a la semántica monádica modular facilitando la reusabilidad de las descripciones de este formalismo. Ha sido utilizada para la descripción de lenguajes imperativos [14], funcionales [12], orientados a objetos [15] y de programación lógica [13].

La descripción utiliza functores no recursivos que capturan la estructura sintáctica y álgebras que toman como conjunto soporte la estructura computacional. Posteriormente, el intérprete se obtiene de forma automática como un catamorfismos sobre la suma de las álgebras definidas.

Las definiciones semánticas serán

$$\begin{array}{lll}
\varphi_{\mathsf{E}} & : & \mathsf{E}(\mathsf{M}\,\mathbb{N}) \to \mathsf{M}\,\mathbb{N} \\
\varphi_{\mathsf{E}}\ (Num\ n) & = & return\ n \\
\varphi_{\mathsf{E}}\ (e_1\ +\ e_2) & = & e_1 \gg\!= \lambda v_1 \to \\
 & & \quad e_2 \gg\!= \lambda v_2 \to \\
 & & \quad return\ v_1 + v_2
\end{array}$$

El valor de una variable debe buscarse en el entorno. La modelización de este concepto semántico se realiza transformando la mónada anterior en una mónada lectora del entorno.

$$\begin{array}{lll}
\varphi_{\mathsf{V}} & : & \mathsf{V}(\mathsf{M}\,\mathbb{N}) \to \mathsf{M}\,\mathbb{N} \\
\varphi_{\mathsf{V}}\ (Var\ x) & = & rdEnv \gg\!= \lambda\varsigma \to return(lkp\ \rho\ x)
\end{array}$$

La incorporación de comandos supone utilizar dos categorías sintácticas, una de expresiones y otra de órdenes aritméticas. Las expresiones pueden reutilizarse convirtiendo el functor en un bifunctor.

A nivel semántico, la posibilidad de utilizar un estado modificable durante la ejecución requiere únicamente transformar la mónada que define la estructura computacional.

$$
\begin{array}{ll}
\psi_{\mathbb{C}} & : \ \mathbb{C}\,(\mathsf{M}\,\mathbb{N})\,(\mathsf{M}\,())\ \rightarrow\ (\mathsf{M}\,()) \\
\psi_{\mathbb{C}}\,(\mathit{Skip}) & = \mathit{return}\ () \\
\psi_{\mathbb{C}}\,(c_1\ ;\ c_2) & = c_1 \gg\!= \lambda\_ \rightarrow\ c_2 \\
\psi_{\mathbb{C}}\,(x\ :=\ e) & = e \gg\!= \lambda v \rightarrow \\
 & \quad \mathit{fetch} \gg\!= \lambda\varsigma \rightarrow \\
 & \quad \mathit{set}\,(\mathit{upd}\ \varsigma\ x\ v)
\end{array}
$$

11 Conclusiones

En la tabla 1 se presenta una valoración comparada de las diferentes técnicas de especificación semántica. La comparación entre estas características es cualitativa, no cuantitativa, intentando huir de valoraciones numéricas que podrían dar lugar a juicios equívocos. Se utiliza el siguiente esquema: si el formalismo X admite la característica Y, entonces en la celda (X, Y) aparece el símbolo $\uparrow$. Si admite la característica de forma parcial, se utiliza el símbolo $\approx$. Finalmente, si no admite la característica, no se pone nada.

	NAM	MOD	REU	DEM	PRO	LEG	FLE	EXP
Leng. Natural		$\uparrow$	$\uparrow$			$\approx$	$\uparrow$	$\uparrow$
Sem. Operacional Est.	$\uparrow$			$\approx$	$\approx$	$\approx$	$\uparrow$	$\uparrow$
Sem. Natural	$\uparrow$			$\approx$	$\uparrow$	$\approx$	$\uparrow$	$\uparrow$
Sem. Denotacional	$\uparrow$			$\uparrow$	$\approx$		$\uparrow$	$\approx$
Sem. Axiomática	$\approx$			$\uparrow$		$\approx$		$\uparrow$
Sem. Algebraica	$\uparrow$		$\approx$	$\uparrow$	$\uparrow$	$\approx$	$\approx$	$\approx$
Máq. Estado Abstracto	$\uparrow$	$\uparrow$	$\approx$	$\approx$	$\uparrow$		$\approx$	$\uparrow$
Sem. de Acción	$\uparrow$	$\uparrow$	$\approx$	$\approx$	$\uparrow$	$\uparrow$	$\approx$	$\approx$
Sem. Monádica Modular	$\uparrow$	$\uparrow$		$\uparrow$	$\approx$		$\uparrow$	
Sem. Monádica Reutilizable	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$		$\uparrow$	

Tabla 1. Tabla comparativa entre técnicas de especificación semántica

- No Ambigüedad (NAM). Evidentemente, el lenguaje natural es ambigüo y poco riguroso. En ese sentido, todos los formalismos estudiados resuelven el problema de la ambigüedad mediante especificaciones de raíz matemática. En el caso de la semántica axiomática pueden surgir problemas de ambigüedad debido a la no especificación de ciertas características.
- Modularidad semántica (MOD). La modularidad semántica es alcanzada en el lenguaje natural debido a la falta de rigurosidad, que permite realizar descripciones ligeramente vagas, que se adaptan a la introducción de nuevas características semánticas independientes. Los formalismos tradicionales no resuelven este problema y quizás sea ésa una de las razones de su escasa utilización práctica [23]. Las máquinas de estado abstracto resuelven el problema mediante la especialización de la abstracción del estado, la semántica de acción lo resuelve mediante la especialización de facetas y las semánticas monádica modular y reutilizable lo resuelven mediante transformadores de mónadas.
- Componentes semánticos reutilizables (REU). El lenguaje natural, por la misma razón anterior, admite la reutilización de las especificaciones, las cuales siguen la modelización conceptual de las características del lenguaje. Los formalismos tradicionales no atacan este problema y por tanto, tampoco lo resuelven. La semántica algebraica contiene potentes mecanismos de reutilización, aunque las especificaciones realizadas en este formalismo son monolíticas [5]. En la semántica de acción se han planteado soluciones al problema, aunque no han sido implementadas [4] y parece que su implementación traería

problemas de inconsistencias. La semántica monádica reutilizable resuelve el problema admitiendo la definición y reutilización de componentes semánticos que pueden incorporarse a un lenguaje de forma independiente. El sistema no produce inconsistencias al separar en categorías sintácticas diferentes las entidades que se van incorporando.

- Demostración de propiedades (DEM). El lenguaje natural no admite la demostración de propiedades de los lenguajes especificados. Las otras técnicas admiten la demostración de propiedades con mayor medida cuanto menos operacionales sean.
  La semántica axiomática y algebraica facilitan el razonamiento formal al utilizar axiomas independientes de una implementación particular. De la misma forma, la semántica denotacional describe el lenguaje mediante conceptos matemáticos abstractos con los que se pueden realizar demostraciones.
  La demostración de propiedades en semántica de acción es difícil ya que la notación de acciones se basa en semántica operacional. En [22] se plantea la relación entre la semántica de acción y la semántica monádica modular. Para ello define la notación de acciones mediante transformadores de mónadas en lugar de utilizar semántica operacional estructurada.
  Las semánticas monádica modular y reutilizable tienen su base en la combinación entre semántica denotacional y semántica algebraica, ya que los conceptos del lenguaje se definen a partir de mónadas, los cuales tienen un comportamiento observacional definido a partir de axiomas.
- Prototipos (PRO). Los sistemas operacionales facilitan la creación de prototipos. De hecho, todos los sistemas que tienen una base operacional (semántica operacional estructurada, semántica natural, máquinas de estado abstracto y semántica de acción) permiten el desarrollo de prototipos. La semántica axiomática tradicional no permite esta posibilidad, ya que la definición de los axiomas se realiza precisamente teniendo en cuenta la independencia de la implementación. A diferencia de ella, la semántica algebraica, al basarse el lógica ecuacional, permite desarrollar prototipos mediante técnicas de reescritura. Por la misma razón, las semánticas monádica modular y reutilizable permiten obtener prototipos de forma directa.
- Legibilidad (LEG). El único sistema que ha prestado gran importancia a la legibilidad evitando además los problemas de ambigüedad del lenguaje natural, es la semántica de acción. Podría ser interesante incorporar las facilidades sintácticas de la semántica de acción en la semántica monádica reutilizable mediante la definición de un metalenguaje de dominio específico.
- Flexibilidad (FLE). Las técnicas operacionales y denotacionales se adaptan a la definición de lenguajes de diferentes paradigmas. Sin embargo, es más difícil desarrollar una semántica axiomática para nuevos lenguajes y paradigmas, lo cual requiere cierta experiencia.
  Como ya se ha indicado, la semántica monádica reutilizable ha sido utilizada para la descripción de lenguajes en los paradigmas imperativo, lógico, funcional y orientado a objetos.
- Experiencia (EXP). Las técnicas tradicionales, especialmente, las técnicas de base operacional, han sido aplicadas a la descripción de lenguajes *reales*. En teoría, la semántica monádica reutilizable podría utilizarse para la especificación de lenguajes de programación prácticos.

Referencias

1. R. Backhouse. *Program Construction and Verification.* Prentice Hall International, Englewood Cliffs, NJ, 1986.
2. E. Börger and W. Schulte. Programmer friendly modular definition of the semantics of java. In J. Alver-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998.
3. Computer and Business Equipment Manufacturers Association, editors. *Programming Language FORTRAN.* American National Standard Institute, Inc., 1978. X3.9-1978, Revision of ANSI X3.9-1966.
4. K. Doh and P. Mosses. Composing programming languages by combining action-semantics modules. In M. van den Brand, M. Mernik, and D. Parigot, editors, *First workshop on Language, Descriptions, Tools and Applications*, Genova, Italy, April 2001.
5. Joseph A. Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs.* Foundations in Computer Science. The MIT Press, 1996.
6. D. Gries. *The Science of Programming.* Springer-Verlag, New York, 1981.

7. Y. Gurevich. Evolving algebra 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
8. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.
9. Bill Joy, Guy Steele, Jame Gosling, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 2 edition, 2000.
10. B. L. Kurtz K. Slonneger. *Formal Syntax and Semantics of Programming Languages*. Addison-Wesley, 1995.
11. G. Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, volume 247, pages 22–39. Springer-Verlag, 1987.
12. J. E. Labra, J. M. Cueva, M. C. Luengo, and A. Cernuda. LPS: A language prototyping system using modular monadic semantics. In M. van den Brand and D. Parigot, editors, *First Workshop on Language Descriptions, Tools and Applications*, volume 44, Genova, Italy, April 2001. Electronic Notes in Theoretical Computer Science – Elsevier.
13. J. E. Labra, J. M. Cueva, M. C. Luengo, and A. Cernuda. Specification of logic programming languages from reusable semantic building blocks. In *International Workshop on Functional and (Constraint) Logic Programming*, Kiel, Germany, September 2001. University of Kiel.
14. J. E. Labra, J. M. Cueva Lovelle, M. C. Luengo Díez, and B. M. González. A language prototyping tool based on semantic building blocks. In *Eight International Conference on Computer Aided Systems Theory and Technology (EUROCAST'01)*, Lecture Notes in Computer Science, Las Palmas de Gran Canaria – Spain, February 2001. Springer Verlag.
15. Jose E. Labra. *Modular Development of Language Processors from Reusable Semantic Specifications*. PhD thesis, Dept. of Computer Science, University of Oviedo, 2001. In spanish.
16. Sheng Liang and Paul Hudak. Modular denotational semantics for compiler construction. In *Programming Languages and Systems – ESOP'96, Proc. 6th European Symposium on Programming, Linköping*, volume 1058 of *Lecture Notes in Computer Science*, pages 219–234. Springer-Verlag, 1996.
17. Sheng Liang, Paul Hudak, and Mark P. Jones. Monad transformers and modular interpreters. In *22nd ACM Symposium on Principles of Programming Languages, San Francisco, CA*. ACM, January 1995.
18. P. Mosses. Foundations of modular SOS. Technical Report RS-99-54, BRICS, Dept. of Computer Science, University of Aarhus, 1999.
19. Peter D. Mosses. *Action Semantics*. Cambridge University Press, 1992.
20. Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.
21. C. Strachey. Fundamental Concepts in Programming Languages. *Higher Order and Symbolic COmputation*, 13, April 2000. Reprinted from Lecture notes, International Summer School in Computer Programming at Copenhagen, 1967.
22. Keith Wansbrough. A modular monadic action semantics. Master's thesis, Department of Computer Science, University of Auckland, February 1997.
23. David A. Watt. Why don't programming language designers use formal methods? In *Seminario Integrado de Software e Hardware - SEMISH'96*, pages 1–6, Recife, Brazil, 1996. University of Pernambuco.
24. Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. Foundation of Computing Series. MIT Press, Cambridge, MA, 1993.