Ambiente de Especificación Denotacional

Pedro Borges
Roger Soler

Departamento de Computación y
Tecnología de la Información
Universidad Simón Bolívar
Apartado Postal 89000
Caracas, Venezuela

Resumen

En este trabajo presentamos un ambiente basado en el método denotacional para la definición de lenguajes de programación. El ambiente permite manipular o ejecutar especificaciones denotacionales mediante su concretización en LISP. Presentamos una breve descripción de las especificaciones que el ambiente maneja y la descripción funcional del ambiente.

1. INTRODUCCION

El ambiente propuesto tiene como objetivo manipular especificaciones denotacionales, las cuales están basadas en el Método Denotacional desarrollado por D. Scott y C. Strachey ([Stoy77], [Gordon79]), para especificar la semántica de los lenguajes de programación secuenciales.

Este método consiste en asociar a cada constructor sintáctico del lenguaje de programación un valor u objeto matemático. Esta asociación está dirigida por la morfología del lenguaje y los valores u objetos matemáticos pertenecen a lo que se conoce como Teoría de Dominios ([Scott76], [Soler82]). Para lograr la manipulación de estos objetos, el ambiente crea aproximaciones a los mismos mediante estructuras de un lenguaje funcional (LISP).

La especificación denotacional presenta la descripción abstracta del interpretador de un lenguaje. Varios trabajos se han orientado a la generación de compiladores "eficientes" a partir de dichas especificaciones ([Bodwin82], [Raskovsky82], [Sethi83], [Wand82], [Kelsey89]). En nuestro caso, se ha buscado que la implementación sea lo más próxima posible a la descripción abatracta, principalmente manteniendo la estructura funcional del interpretador abstracto. Esto nos permite, en particular, observar el funcionamiento de las partes del interpretador.

A continuación se presenta una breve descripción de las especificaciones denotacionales que manipula el ambiente, luego la descripción funcional del ambiente y algunos comentarios sobre el mismo.

2. ESPECIFICACION DENOTACIONAL

2.1. Dominios sintácticos.

La especificación de los dominios sintácticos consta de dos partes. La primera es una lista de los nombres de los dominios y sus correspondientes elementos genéricos. La segunda es la descripción de la morfología de dichos elementos genéricos, en forma de "operador(operandos)", de modo de eliminar ambigüedades. Por ejemplo, la morfología de una instrucción del tipo "si exp entonces inst_1 sino inst_2" sería de la forma "si(exp,inst_1,inst_2)".

Esta notación para la morfología de los dominios sintácticos se considera equivalente a la descripción de la forma de los árboles abstractos de los elementos de dichos dominios, donde el operador se considera el nodo raíz y los operandos los nodos hijos.

Los dominios de identificadores y cifras están predefinidos con ciertos operadores. En particular, la igualdad sintáctica de identificadores.

2.2. Dominios semánticos.

Esta especificación es una lista de las variables genéricas, los nombres y las estructuras de cada uno de los dominios

semánticos que se definen. Los dominios de los números enteros y los valores de verdad (cierto, falso) se consideran dominios primitivos con nombre y estructura predefinidas, por lo que no es necesario especificarlas. Las operaciones usuales sobre estos dominios también están predefinidas: aritméticas (suma, resta, multiplicación, ...), relacionales (igualdad, menor, mayor, menor o igual, ...) y lógicas.

En general la estructura de un dominio se describe a partir de otros dominios mediante el uso de los constructores producto cartesiano(x), unión disjunta(+), espacio de funciones(->) y secuenciación como generalización del producto cartesiano(*); para los cuales se dispone de un conjunto de operaciones tales como: selección de una componente en un producto cartesiano (↓), proyección en un componente de una unión (|), aplicación de funciones, cabeza y cola de una secuencia, etc.

Típicamente, el significado o interpretación de los programas en el lenguaje que se define es una función que transforma elementos de un dominio de entrada en elementos de un dominio de salida o respuesta. Estos dominios se consideran "elementos distinguidos" o "dominios de interés".

2.3. Funciones de Interpretación.

La especificación de una funcion de interpretación consiste de su funcionalidad y su definición utilizando el lambda cálculo con las operaciones mencionadas. Cabe mencionar que los elementos sintácticos a los cuales se aplican las funciones de interpretación se denotan por sus árboles abstractos tal y como fueron especificados en la sintaxis abstracta.

En la Figura 1. se presenta la especificación denotacional de un lenguaje muy sencillo, sin incluir la función de interpretación para las expresiones booleanas. Las funciones subrayadas son funciones predefinidas en el ambiente.

3. DESCRIPCION FUNCIONAL DEL AMBIENTE

3.1. Dominio de las especificaciones.

Las especificaciones denotacionales pertenecen a un dominio **ED** de la forma:

$$\mathbf{ED} = \mathbf{DSIN} \times \mathbf{DSEM} \times \mathbf{FINT}$$

donde **DSIN** son los conjuntos de dominios sintácticos, **DSEM** son los conjuntos de dominios semánticos y **FINT** los conjuntos de funciones de interpretación que pueden ser descritos por una especificación.

```
DOMINIOS SINTACTICOS
    prog  ∈  PROGRAMAS
    ide   ∈  IDENT
    inst  ∈  INSTRUCCIONES
    exp   ∈  EXPRESIONES
    ebool ∈  EXPBOOLEANAS
    n     ∈  CIFRAS
MORFOLOGIA
    prog ::=  inst
    inst ::=  "si" ( ebool, inst_1, inst_2 )
         ::=  ";" ( inst_1, inst_2 )
         ::= "mientras" ( ebool, inst )
         ::= "escribir"(ide)
         ::= "sea" ( "=" ( ide, exp ) , inst )
    exp  ::=  ide
         ::=  "+" ( exp_1 , exp_2 )
         ::=  n
DOMINIOS SEMANTICOS
           BOOL
           RES = NATURALES *
    s  ∈  S = [IDENT -> V] x NATURALES *
    v  ∈  V = NATURALES
           C = S -> S
FUNCIONES DE INTERPRETACION
    P : PROGRAMAS -> RES
    I : INSTRUCCIONES -> S -> S
    E : EXPRESIONES -> S -> V
    EB: EXPBOOLEANAS -> S -> BOOL
DEFINICIONES
P[[ inst ]] = ( I[[ inst ]] < LAMBDA id.0 , nil > ) ↓ 2
I[[ "si"(ebool,inst_1,inst_2) ]] =
    LAMBDA s. EB[[ebool]]s -> I[[inst_1]]s ,I[[inst_2]]s
I[[";"(inst_1,inst_2)]] =
          LAMBDA s. I[[inst_2]] ( I[[inst_1]] s )
I[["mientras"(ebool,inst)]] =
         LAMBDA s. EB[[ebool]]s ->
              I[["mientras"(ebool,inst)]] (I[[inst]] s),
              s
I[["escribir"(ide)]] = LAMBDA s.
        < s ↓1, postfijar (I[[ide]] s) s ↓ 2 >
I[["sea"("="(ide,exp),inst)]] =
         LAMBDA s. I[[inst]] < s ↓ 1[v/ide], s ↓2 >
                   donde  v = E[[exp]] s
E[[ide]] = LAMBDA s. (s ↓1) ide
E[["+"(exp_1,exp_2)]] = LAMBDA s.
         suma (E[[exp_1]] s) (E[[exp_2]] s )
E[[n]] = LAMBDA s. nat n
```

Fig 1. Ejemplo de Especificación Denotacional

Una especificación particular se denota **ed**, y sus componentes **dsin**, **dsem** y **fint** respectivamente. Los dominios de entrada y respuesta ("elementos distinguidos") de **dsem** se denotan **ENT** y **RES** respectivamente, esto es:

$$ed \in ED; \quad ed = < dsin, dsem, fint >$$
$$dsin \in DSIN$$
$$dsem \in DSEM$$
$$fint \in FINT$$
$$ENT \in dsem$$
$$RES \in dsem$$

3.2. Chequeo estático de tipos.

El ambiente chequea la correctitud (consistencia) de tipos de las definiciones de las funciones de interpretación y las funciones auxiliares en **fint** según la funcionalidad de las mismas, los dominios sintácticos en **dsin** y los dominios semánticos en **dsem**. En caso de detectar algún error, el ambiente emite un mensaje descriptivo del mismo.

Para simplificar supondremos que el resultado de la verificación es "cierto" si no se detectaron errores o "falso" si se detectaron inconsistencias de tipos. Por lo tanto, si VT es la función verificadora de tipos, se tiene:

```
VT : ED -> { "cierto", "falso" }
VT ed =  si ed contiene inconsistencia de tipos
                entonces "cierto"
         si no
                entonces "falso"
```

3.3. Concretización de los dominios semánticos.

Para manipular los dominios semánticos en **dsem**, éstos se concretizan o representan como estructuras en LISP. Si denotamos **ELISP** al dominio de las estructuras en LISP, y **P(ELISP)** al conjunto de las partes del mismo, podemos definir la función concretizadora de dominios semánticos

$$CDSEM : \mathbf{DSEM} \rightarrow P(\mathbf{ELISP})$$

De modo que CDSEM aplicada a un conjunto de dominios semánticos genera el conjunto de las estructuras en LISP que representan a cada uno de los dominios semánticos en el conjunto al cual se aplica. En particular, CDSEM **dsem** es el conjunto de estructuras en LISP que representan los dominios semánticos en **dsem**.

3.4. Concretización de las Funciones de Interpretación.

La concretización de las funciones de interpretación produce un programa (en LISP) que engloba la semántica del lenguaje que se define en la especificación. Dado que en la definición de estas funciones intervienen los dominios semánticos, para su concretización es necesario conocer las concretizaciones de dichos dominios. Por lo tanto, si denotamos **PLISP** al dominio de los programas en LISP, y CFINT a la función concretizadora, tenemos:

CFINT : **FINT** -> P(**ELISP**) -> **PLISP**

es decir que CFINT, dado un conjunto de funciones auxiliares y de interpretación (en **FINT**) y las concretizaciones de los dominios semánticos (en **ELISP**) produce un programa en LISP que concretiza la semántica del lenguaje que se define en la especificación. En particular

CFINT **fint** (CDSEM **dsem**)

es el programa en LISP que representa la semántica del lenguaje que se define en **ed** (**ed** = < **dsin**,**dsem**,**fint** >).

Hacemos notar que LISP tiene la particularidad de que los programas no son maá que estructuras manipulables por el lenguaje ([Allen78]). En este sentido, el dominio **PLISP** es equivalente al dominio **ELISP**, sin embargo, los denotamos en

forma distinta para obtener mayor claridad en la descripción.

3.5. Acción global del Ambiente.

Podemos definir una función de concretización global en base a las funciones descritas anteriormente, como la función CONC, que realiza el chequeo estático de tipos, y de no encontrar errores, produce la concretización de la semántica contenida en la especificación que se maneja. En caso de encontrar inconsistencias de tipos, CONC regresa un valor especial que denotaremos "error". Tenemos entonces:

```
CONC : ED -> PLISP + { "error" }
CONC ed = si VT ed
                  entonces CFINT fint ( CDSEM dsem )
          si no
                  entonces "error"
donde  ed = < dsin,dsem,fint >
```

3.6. Ejecución de la semántica.

El programa (en LISP) generado por CONC para una especificación **ed** constituye un interpretador para el lenguaje descrito en **ed**, que puede ser ejecutado por una máquina LISP, la cual puede ser una máquina real o,

típicamente, simulada por software. En todo caso, la podemos representar por la función EVAL de LISP. De modo que:

EVAL (CFINT **fint** (CDSEM **dsem**))

es un interpretador para el lenguaje descrito en ed.

Este interpretador recibe la concretización (en LISP) de un programa en el lenguaje definido por **ed** y de un elemento del dominio **ENT** (de **ed**) y produce una concretización de un elemento del dominio **RES** (de **ed**). Este elemento es el generado por la aplicación del programa recibido sobre el elemento de **ENT**. Por lo que tenemos:

EVAL (CFINT **fint** (CDSEM **dsem**)) : **ELISP -> ELISP -> ELISP**

donde el primer elemento en **ELISP** es la concretización de un programa en el lenguaje que se define (en **ed**), el segundo es la concretización de un elemento del dominio de entrada **ENT** y el tercero la concretización de un elemento del dominio de salida **RES**. Para generar las concretizaciones que recibe el interpretador como argumentos, se definen nuevas funciones que se describen a continuación.

Dada una especificación **ed** = < **dsin,dsem,fint** >, denotaremos **L(dsin)** al lenguaje descrito por **dsin**, es decir, al conjunto de programas (árboles abstractos) del lenguaje definido por

ed, y **prog** a un programa en particular (es decir, **prog** $\in$ **L(dsin)**). Si CPROG es la función concretizadora de programas en **L(dsin)** a LISP, tenemos:

CPROG : **L(dsin)** -> **ELISP**

CPROG **prog** = estructura en LISP que representa a **prog**

donde **prog** $\in$ **L(dsin)**

En cuanto a la concretización de los elementos del dominio **ENT**, si **ent** es un elemento particular (**ent** $\in$ **ENT**), definimos la función concretizadora CENT como:

CENT : **ENT** -> **ELISP**

CENT **ent** = estructura en LISP que representa a **ent**

Utilizando las funciones descritas anteriormente, la aplicación del interpretador a CPROG **prog** y a CENT **ent**, obtenemos la concretización del elemento del dominio de salida **SAL** (es decir, una estructura en LISP que lo representa) generado por la aplicación de **prog** sobre **ent**, según la semántica descrita en **ed**, por lo que:

EVAL(CFINT **fint** (CDSEM **dsem**))(CPROG **prog**)(CENT **ent**) $\in$ **ELISP**

es la concretización del elemento de **SAL** descrito anteriormente.

Si CONC ed es distinto de "error" (y por lo tanto CONC ed ∈ PLISP), la expresión del párrafo anterior es equivalente a:

EVAL (CONC ed) (CPROG prog) (CENT ent) ∈ ELISP

4. COMENTARIOS

Como se mencionó en la sección 2.1, las especificaciones manejadas por el ambiente no describen la sintaxis de los programas "reales" de los lenguajes que definen, sino la morfología de los árboles abstractos de dichos programas. Es sobre estos árboles que el ambiente trabaja. Pensamos incorporar la capacidad de trabajar directamente sobre los programas, para lo que será necesario desarrollar la función de transformación de los programas en sus árboles (lo que se conoce como "parsing"). Esta transformación, sin embargo, es un problema bastante estudiado y para la cual existen diversas técnicas eficientes ([Aho77]).

En cuanto a las funciones de concretización descritas en la sección 3, hemos desarrollado versiones que trabajan sobre especificaciones sin dominios reflexivos ([Naranjo89]), trabajándose actualmente en la incorporación de dichos dominios.

Otro aspecto que se encuentra en estudio es el establecimiento de un mayor número de dominios predefinidos con sus operaciones respectivas como parte del ambiente.

5. CONCLUSIONES

La descripción funcional del ambiente ha permitido identificar claramente las distintas componentes del proceso de implementación de una especificación denotacional. Pensamos que, aún cuando el ambiente no está totalmente desarrollado, será de utilidad para crear especificaciones denotacionales y obtener prototipos de las mismas.

6. REFERENCIAS

[Aho77] AHO, A.V., Y ULLMAN, J.D. Principles of Compiler Design. Addison-Wesley, Reading, Mass, 1977.

[Allen78] ALLEN, J. Anatomy of LISP. McGraw-Hill, 1978.

[Bodwin82] BODWIN, J., BRADLEY, L., KANDA, K., LITLE, D., Y PLEBAN, U. Experience with an experimental compiler generator based on denotational semantics. En Proc. SIGPLAN '82 Symp. Compiler Construction, SIGPLAN Notices 17(6), pp. 216-229 (Junio 82).

[Gordon79] GORDON, M.J.C. The Denotational Description of Programming Languages. Springer Verlag, New York, 1979.

[Kelsey89] KELSEY, R. Y HUDAK, P. Realistic Compilation by Program Transformation. Proc. ACM Symposium on principles of Programming Languages. Enero 1989, pp. 281-292.

[Naranjo79] NARANJO, F. Verificacion de tipos y generación de código a partir de una especificación denotacional. Proyecto de Grado. Tutor: Roger Soler. Universidad Simón Bolívar. Septiembre 1989.

[Raskovsky 82] RASKOVSKY, M. Denotational semantics as a specification of code generators. En Proc. SIGPLAN '82 Symp. Compiler Construction, SIGPLAN Notices 17(6), pp. 230-244 (Junio 82).

[Scott76] SCOTT, D. Data Types as Lattices. SIAM Journal of Computing 5, pp. 522-587, 1976.

[Sethi83] SETHI, R. Control Flow Aspects of Semantics-Directed Compiling. ACM Transactions on

Programming Languages and Systems 5(4), Octubre 1983, pp. 554-595.

[Soler82] SOLER, R. Fundamentos de un Meta-Lenguaje para describir Semántica. Dpto. de Computación, Universidad Central de Venezuela, 1982.

[Stoy77] STOY, J. Denotational Semantics. MIT Press, Cambridge, Mass. 1977.

[Wand82] WAND, M. Deriving Target Code as a representation of continuation semantics. ACM Transactions on Programming Languages and Systems 4(3), Julio 1982, pp. 496-517.