XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa
Buenos Aires, Setiembre 1988.

Una implementación funcional a partir de una especificación algebraica

Fernando Carpani - Diana Cukierman

Instituto de Computación
Universidad de la República
Julio Herrera y Reissig 565
Montevideo. URUGUAY

Resumen.

Se describe una implementación realizada en un subconjunto puramente funcional de lenguaje Scheme a partir de una especificación algebraica.

De la experiencia en este proyecto de tamaño mediano (el prototipo de un intérprete de Prolog "puro"), son revisados los problemas hallados en la transformación "especificación formal - implementación".

Introducción.

Una etapa del proceso de construcción de programas es la elaboración de una especificación formal del problema. Esta cumple al menos dos funciones. La primera y principal, ser sometida a tratamiento matemático. La segunda, guiar el proceso de diseño e implementación.

Una forma difundida de implementar esta segunda función, es utilizar la especificación como medio de comunicación no ambiguo entre quien define el problema y quien diseña e implementa los programas.

El objeto de este trabajo es explorar esta visión a través del estudio de un caso: la construcción del prototipo de un intérprete de Prolog "puro" a partir de una especificación algebraica, en una porción puramente funcional de Scheme.

El trabajo se desarrolló bajo la dirección de Juan Echagüe (Instituto de Computación, Escuela Superior Latinoamericana de Informática).

En lo que sigue, se revisan las herramientas de trabajo y la Especificación Algebraica de partida, para luego pasar a puntos relevantes aparecidos en la tarea de implementación. Finalmente se discuten algunas conclusiones.

Especificación Algebraica y Scheme.

En principio, debido a lo diverso de sus objetivos, parece carecer de sentido comparar un lenguaje de especificación con uno de programación .

Sin embargo, en el proceso de desarrollo existe un momento en que deben elegirse ambas herramientas. Y muchas veces estas dos elecciones no son independientes: pesa en ellas (no exclusivamente), una noción informal de "distancia" entre los dos formalismos, que se relaciona con el costo de implementar en uno lo especificado en el otro.

La elección de Scheme [Rees 86] dada la Especificación Algebraica [Guttag 78] [Tasistro 86] fundó en la expectativa de reducir esa distancia, ya que ambos formalismos se apoyan en la noción de función. Además, la experiencia de [Zanni 87] apunta en esa dirección.

La *especificación algebraica* (EA) de *tipos abstractos de datos* (TADs) es un lenguaje de especificación formal especialmente adaptado a las técnicas de diseño basadas en la definición de clases de objetos cuyas propiedades son descriptas por su comportamiento frente a la aplicación de ciertas funciones. Provee mecanismos de representación abstracta que favorecen la separación entre la etapa de diseño y la de implementación.

Scheme es un dialecto de Lisp, que posee una semántica clara y uniforme. Posee elementos imperativos, que no fueron utilizados en este trabajo.

Entre las facilidades de la EA que aparecen en primera instancia no triviales de implementar en Scheme estan:

- semántica estricta de las funciones
- selección de axiomas basado en "filtrado"
- tipos estrictos.

En la implementación debe enfrentarse también el problema de los objetos "sin sentido" [Cornes 88] que aparecen en la EA.

Por su parte las, *funciones de orden superior* -aquellas que tienen funciones en su dominio o recorrido [Henderson 80]) -de Scheme no tiene correspondiente en EA.

Estas funciones son herramientas importantes de modularización [Vidart 88] propias de la programación funcional (sin equivalentes en programación imperativa [Echagüe 88]).

Un ejemplo clásico de función de orden superior es *composición*. Tiene como recorrido las funciones de una variable y como dominio las parejas de estas funciones. Como se espera, para una pareja de funciones f y g devuelve su composición, $f \circ g$. En Scheme se expresa

```
(define composición
     (lambda (f g)
          (lambda (x) (f (g x)))))
```

y definir, a partir de *sqr*, *padre* y *progenitor*

```
(define cuarta_potencia (composición sqr sqr))
(define 2+ (composición 1+ 1+))
(define abuelo (composición padre progenitor))
(define abuelo_paterno (composición padre padre))
```

La Especificación Algebraica de partida.

La EA de partida [Cornes 88] es un documento de 12 hojas, donde se definen 12 tipos. La técnica usada es la propuesta en [Guttag 78]. El estilo es extremadamente detallado, y para obtener completitud, especifica las operaciones selectoras a partir de las constructoras.

En [Cornes 88] se discuten extensamente los problemas de los objetos sin sentido, que aparecen en esta especificación, asi como interacciones con esta implementación en sus primeras etapas.

La tarea de implementación.

A continuación, se describen aspectos relevantes de los problemas encontrados durante la tarea de implementación.

El manejo de errores.

El objeto *indefinido* es usado en la especificación, para trabajar con funciones parciales. El simbolo *Error* en la implementación cumple el rol equivalente.

En la especificación, todas las funciones son estrictas respecto a *Error*. La implementación de esta propiedad en Scheme resulto sencilla a costa de muchas lineas de codigo para detectar la presencia de *Error* o de objetos sin sentido entre los argumentos de llamada.

Por su parte, podemos observar que una función (tanto en la especificación como en la implementación) debe devolver *Error* por una de dos situaciones:

- Alguno de los valores de su dominio es *Error* o un objeto sin sentido.
- Teniendo sentido todos los valores del dominio, la función no esta definida sobre ellos (por ejemplo, unificador más general de dos términos no unificables).

Sin embargo, será suficiente considerar solamente el segundo caso en una implementación en la que no se permita la construcción de objetos sin sentido.

Las funciones de alto orden.

Los tipos abstractos estan soportados en Scheme por enteros, símbolos y el constructor de pares ordenados (con él estan construídos productos cartesianos, listas y árboles).

Es natural definir por inducción las operaciones sobre estas estructuras, y se utiliza para ello funciones de alto orden.

Por ejemplo, las operaciones sobre listas fueron mayoritariamente definidas utilizando la siguiente función:

```
(define inducción
  (lambda ( @cabeza @resto
            constructor
            @caso_base base transformación )
    (lambda ( o )
      (if (@caso_base o)
          (base o)
          (constructor
           (transformación (@cabeza o))
           ((inducción @cabeza @resto
                       constructor
                       @caso_base base transformación)
            (@resto o)))))))
```

Cuando utilizamos *inducción* para computar una cierta función *Alfa*

- *@cabeza* y *@resto* son selectores sobre el dominio de *Alfa*.
- *constructor* es el constructor del tipo del recorrido.
- *@caso_base* detecta los casos base de la inducción, *base* devuelve la imagen de estos, y *transformación* la de los componentes.

Siendo *T1*, *T2* y *T3* tipos (no necesariamente distingos), para definir una función

$$Alfa : lista(T1) \longrightarrow T2$$

a partir de *inducción* necesitamos

$$@cabeza : lista(T1) \longrightarrow T1$$
$$@resto : lista(T1) \longrightarrow lista(T1)$$
$$constructor : T3 \times T2 \longrightarrow T2$$
$$@caso\_base : lista(T1) \longrightarrow boolean$$
$$base : lista(T1) \longrightarrow T3$$
$$transformación : T1 \longrightarrow T3$$

Podemos decir que *@cabeza* y *@resto* aportan información sobre el tipo del dominio, y *constructor* sobre el tipo de recorrido.

Mediante *inducción* es posible definir operaciones sobre tipos distintos, por ejemplo:

$$Var\_en\_operandos? : Variable \times Operandos \longrightarrow Boolean$$

```
(define Var_en_operandos?
  (lambda ( v op )
    ((Inducción Primer_término Resto_operandos
                Funcion_or
                Operandos_vacio?
                  (lambda (x) #!false)
                  ((curry Var_en_término?) v)) op)))
```

Unir_objetivos : objetivo x objetivo ⟶ objetivo

```
(define Unir_objetivos
  (lambda (ob1 ob2)
    ((Inducción
       Primer_Atomo Resto_objetivo
       cons_objetivo
       objetivo_vacio?
         (lambda(x) ob2)
         (lambda(x) x)) ob1)))
```

El filtrado y los objetos sin sentido.

A continuación, se discuten problemas provenientes de la implementación del filtrado y el manejo de los objetos sin sentido a partir del caso particular del tipo *Término*.

Informalmente, *términos* son aquellos objetos sobre los que aplico *Símbolos de predicado* para obtener *Atomos* [Lloyd 84]. La especificación utilizada del TAD *Término* a partir de los TADs *Variable* y *Functor* es la siguiente:

/ Constructoras de Término */*

Aparición_variable : Variable ⟶ Término
Aparición_functor : Functor ⟶ Término
Cons_término : Término X Término ⟶ Término

Donde la intención de *Cons_término* es construir objetos que representan la aplicación de un functor a una serie de términos (la selección del conjunto de constructores de este TAD se discute extensamente en [Cornes 88]).

Lamentablemente esta definición permite la creación de objetos sin sentido: la aplicación de variables sobre otras. Por ejemplo:

Cons_término(Aparición_variable(a),Aparición_variable(b))

Estas construcciones no representan objeto alguno del problema original (los términos en el lenguaje Prolog).

En EA parece una buena forma de tratarlos el disponer una función que permita distinguir los objetos sin sentido de los demás, y que las restantes funciones, aplicadas sobre objetos sin sentido, devuelvan un valor distinguido *Error*.

El selector sobre *Término*, que permite distinguir los objetos sin sentido es *Término_bien_formado?*, cuya especificación es:

Término_bien_formado? : Término ⟶ Boolean

```
Término_bien_formado?(Aparición_variable(x)) = True
Término_bien_formado?(Aparición_functor(f)) = True
Término_bien_formado?(Cons_término(t1,t2)) =
        Término_bien_formado?(t1)
      and Término_bien_formado?(t2)
      and Not(Es_variable(t1))
```

Esta función se utiliza en la definición de las demás para garantizar la evaluación al valor distinguido.

Tomemos por ejemplo, el siguiente selector

Es_variable? : Término → boolean

Al definir sus axiomas, se utiliza *Término_bien_formado?* para que aplicado a objetos sin sentido, devuelva *Error*.

```
Es_variable?(Aparición_variable(x)) = True
Es_variable?(Aparición_functor(f)) = False
Es_variable?(Cons_término(t1,t2)) =
          if Término_bien_formado?(Cons_término(t1,t2))
             then False
             else Error
```

La intención es utilizar *Es_variable?* para determinar si un término fue construído mediante el constructor *Aparición_variable* y utilizar esto para implementar el filtrado.

Tenemos hasta aquí la especificación de dos funciones sobre *Término*, intuitivamente correctas, y decisiones de diseño:

- todos los selectores, aplicados sobre objetos sin sentido, devuelven el valor distinguido *Error*, con la única excepción de una función que sirve para distinguirlos de los demás objetos.
- el filtrado se implementará a partir de selectores que permiten detectar cual fue el "último" constructor utilizado para construir el objeto.
- la representación interna de los objetos será oculta para la mayoría de los selectores.

Una implementación directa de estas funciones produce, sorpresivamente, un resultado erróneo.

Dada la siguiente implementación de los constructores,

```
(define Aparición_variable (lambda(x) (cons 'variable x))
(define Aparición_functor (lambda(x) (cons 'functor x))
(define Cons_término cons)
```

es errónea una implementación inocente de los selectores de la forma siguiente (que parece natural):

```
(define Término_bien_formado?
  (lambda (t)
    (or (Es_constante? t)
        (Es_variable? t)
        (and (Término_bien_formado? (Primer_término t))
             (Término_bien_formado? (Segundo_término t))
             (not (Es_variable? (Primer_término t)))))))

(define Es_variable?
  (lambda (t)
    (if (Término_bien_formado? t)
        (igual_atomo (car t) 'variable)
        Error)))
```

Hay evaluaciones infinitas, inclusive ante términos bien formados, como

```
Es_variable?
  (Cons_término (Aparición_functor(f),Aparición_variable(v)))
```

Cuál es el problema? Las tres decisiones de diseño no pueden satisfacerse a la vez. Basta con levantar una de ellas para encontrar una solución.

Para salvar este problema, se optó por modificar la definición de *Es_variable?* (y algunos otros selectores) para que no devuelva *Error* cuando es aplicado a objetos sin sentido. Así llegó a modificarse la especificación a partir de problemas (no triviales) de implementación.

La especificación y la implementación en su versión final es:

```
Es_variable? : Término ⟶ Boolean

Es_variable? (Aparición_variable (x)) = True
Es_variable? (Aparición_functor (x)) = False
Es_variable? (C_término (t1,t2)) = False

Término_bien_formado? : Término ⟶ boolean

Término_bien_formado? (Aparición_variable (x)) = True
Término_bien_formado? (Aparición_functor (x)) = True
Término_bien_formado? (C_término (t1,t2)) =
      Término_bien_formado? (t1)
      and Término_bien_formado? (t2)
      and (not Es_variable? (t1))
```

y en Scheme

```
(define Es_variable?
  (lambda (te)
    (and (atom? te) (Simbolo_variable? te))))

(define Término_bien_formado?
  (lambda (te)
    (if (or (Es_variable? te)
            (Es_constante? te))
        True
        (and (and (Término_bien_formado? (Primer_término te))
                  (Término_bien_formado? (Segundo_término te)))
             (not (Es_variable? (Primer_término te)))))))
```

La modularización y la etapa de prueba.

Ni la EA ni el Scheme contienen la idea de módulo, en el sentido de proveer mecanismos adecuados para ocultar información.

La especificación entonces no esta dada por móodulos, sino por un conjunto de TADs, y también algunas operaciones que trabajan sobre estos, sin que quede claro que pertenezca a alguno de ellos en particular.

La implementación se construyó muy rapidamente, transcribiendo cada una de las funciones especificadas.

En este punto del trabajo, luego de una primera escritura del código Scheme aparece la necesidad de clasificar las funciones en grupos, que llamamos módulos, ordenados de alguna forma, con el fin de facilitar:

- La comprensión del código escrito.
- La prueba de las funciones, que podria realizarse por partes y en una secuencia razonable.

Esta clasificación realizó por sucesivos refinamientos.

En una primera etapa, se agrupan las funciones de acuerdo a criterios que contemplan solo los tipos de su dominio y codominio.

En una segunda se realiza una partición de estos módulos, separando donde es posible módulos que implementan clases de objetos y otros que implementan relaciones entre ellas. Aquellos se pueden asociar a los TADs de la especificación, y estos a agrupaciones de las funciones que relacionan TADs.

En esta etapa, se consideró no solamente los tipos de dominio y codominio, sino tambien los de las operaciones que se utilizaron para implementar cada función, aunque no aparezcan alli.

En la tercera y última etapa, se ordenan las funciones de cada módulo en "niveles", usando un criterio análogo al de la etapa anterior.

Como resultado se tiene una jerarquía de módulos y dentro de cada uno de éstos, una de funciones.

La prueba se realizó en forma ascendente. Para la construcción de los juegos de datos se consideraron dos criterios. El primero, revisar todos los casos límites y algunos casos usuales. Segundo, definirlos en forma ascendente, a fin de utilizar en cada nivel los datos que sirvieron para los inferiores.

Conclusiones

Se completó, dentro de márgenes de tiempo razonables, una implementación funcional a partir de la especificación algebraica del prototipo de un interprete de la porción "pura" de Prolog.

El único recurso de programación utilizado que puede considerarse fuera de la ortodoxia funcional es el *let*. Este se incluye solamente en lugares en que no en imprescindible, pero si ventajoso por razones de eficiencia. Puede eliminarse trivialmente sin poner en riesgo la corrección de la implementación.

La construcción de módulos por refinamiento para la prueba resulto ser adecuada. Se percibió la ausencia de herramientas conceptuales y de "software" para la construcción de un programa modular cuando se parte de una especificación de TADs.

Este trabajo alienta la conjetura de la corta distancia entre los lenguajes elegidos. Los problemas mayores surgen con el mecanismo de filtrado y el tratamiento de errores.

Las funciones de alto orden disponibles en Scheme son un mecanismo de alto nivel de abstracción con un gran impacto sobre la pragmática. Eso lleva a que el código escrito tenga, aproximadamente, el mismo tamaño que la especificación.

Esto no debe resultar extraño, ya que ambos documentos sirven a objetivos distintos, uno de ellos para razonar y otro para ser ejecutado.

Con dos salvedades, este trabajo alienta también a considerar positivo utilizar el criterio de minimizar esa distancia para la construcción de prototipos. La primera de la salvedades es la (in)eficiencia del programa obtenido.

La segunda es mas sustancial. Respecto a la posibilidad de razonar sobre la especificación, seria interesante verificar si es mas sencillo hacerlo dentro de EA o de la porción de Scheme que se utilizó, que posee una semántica clara.

Agradecimientos.

Deseamos agradecer los integrantes del Departamento de Programación por el apoyo prestado, especialmente a su director J. Cabezas, a C. Cornes, y a J. Echagüe por su dirección.

Tambien a J. Abrial las discusiones sostenidas en su estadia en ESLAI.

Bibliografía.

[Cornes 88] Cornes, C., Echagüe, J., Una especificación algebraica de Prolog. VIII Conferencia de la Sociedad Chilena de Computación, Santiago, 1988

[Echagüe 88] Echagüe, J., Algunas limitaciones de la "function" Pascal. Revista Escuela de Informatica, nro 3. Montevideo, 1988.

[Guttag 78] Guttag, J., Horning, J., The algebraic specification of abstract data types. Acta Informatica 10, 1978.

[Henderson 80] Henderson , P., Functional programing : Aplication and implementation. Prentice-Hall International, 1980.

[Kowalski 74] Kowalski R. A., Predicate Logic as a Programming Language, IFIP 1974.

[Lloyd 84] Lloyd, J. W., Foundations of Logic Programming, Springer-Verlag, Berlin. 1984.

[Rees 86] Rees et al, Revised^3 Report on the Algorithmic Language Scheme. ACM SIGPLAN Notes 21, 12, 1986.

[Tasistro 86] Tasistro, A., Viola, A.,Especificaciones algebraicas de tipos abstractos de datos para un curso medio de programación. XII Conferencia Latinoamericana de Informática, 1986.

[Vidart 88] Vidart, J., Tasistro, A.,Programación funcional y lógica. III Escuela Brasilero Argentina de Informática, 1988.

[Zannni 87] Zanni, C., Pedregal, C., Kesner, D., Cajias, I.. Un Editor Gráfico de Grafos. Sin publicar. 1987.