DISEÑO E IMPLEMENTACION DE TIPOS ABSTRACTOS DE DATOS

Una metodología basada en funciones recursivas

Rodrigo Cardoso R.
Universidad de los Andes
Bogotá - *Colombia*

Resumen: Se presenta una metodología intuitiva para el diseño y la implementación de tipos abstractos de datos (TADs), con ayuda de funciones recursivas. Un TAD se entiende como un conjunto de nombres o formas normales sobre los cuales se definen operaciones que reflejan la estructura de un universo intuitivamente bien conocido que se quiere modelar.

Cada TAD definido es por construcción suficientemente completo.

Se define una relación de equivalencia sobre el conjunto de formas normales que captura la semántica del TAD con base en operaciones que observan propiedades externas de los objetos que se modelan. Se definen conjuntos de estas operaciones que caracterizan la relación de equivalencia mencionada.

Se estudia la implementación de TADs con lenguajes de programación de tipo imperativo.

La teoría se ilustra con el ejemplo "clásico" del TAD Stack[X], que modela pilas de objetos de un tipo primitivo X. Al final se incluye un ejemplo más complejo, el TAD Cubo, que modela el funcionamiento del cubo de Rubik, para mostrar la práctica de la metodología en un caso no trivial.

Palabras Claves: tipo abstracto de datos (TAD), funciones recursivas, formas normales, reducción de términos, completitud suficiente, equivalencia semántica, implementación de TADs.

0. Motivación

Los tipos abstractos de datos (TADs) pueden ser una herramienta útil para la definición y manipulación de datos sobre los que operen programas de computador. Sin embargo, la teoría de TADs se ha desarrollado algo lejos de la práctica, de modo que se ha considerado interesante el estudio de problemas que nunca deberían aflorar en la realidad (v.gr. completitud suficiente) si el diseño se llevara a cabo siguiendo metodologías que justamente evitaran "por construcción" las grandes dificultades que se presentan en una teoría general de TADs.

En la práctica la preocupación debe ser proporcionar al observador (diseñador, programador) una metodología de trabajo que le permita representar simbólicamente un universo O (un conjunto de explícitos) que desea estudiar y operar, de modo que la representación refleje todas las características que el observador juzgue relevantes(1). Si para alcanzar este objetivo se usan TADs, es interesante entonces disponer de una metodología que guíe el diseño de un TAD T del cual O sea un modelo.

Lo que el observador considere importante de O debe ser capturado en T, y aunque es posible que algo más de lo "interesante" sea abstraído, el único modelo de T que interesa al observador es O. El TAD T no es otra cosa que un lenguaje formal para hablar de O, y cuando las cosas van bien, este lenguaje, que es una herramienta sintáctica, encierra todo lo que el observador sabe sobre O, es decir, su semántica.

Como el universo O debe ser "aprehensible", es natural pedir del observador que pueda describirlo de alguna manera efectiva. Por ejemplo, debería poder enumerarlo, o sea concebir una forma recursiva (intuitiva) para nombrar los elementos de O, que también puede pensarse como conocer una forma de construír todos los elementos de O. Esta hipótesis de constructividad intuitiva de O permite comenzar el diseño del TAD T definiendo (recursivamente) el conjunto subyacente o tipo de interés Y como un conjunto de nombres estándar o formas normales para los elementos de O, de modo que cada objeto tenga al menos un nombre.

Eventualmente el observador puede considerar transformaciones efectivas sobre objetos de O que no creen nuevos objetos. En el TAD T se tendrá como contraparte la existencia de funciones recursivas definidas sobre el conjunto de nombres Y. La misma idea se sigue para modelar en T el hecho de que el observador pueda analizar una característica de los objetos. La definición de estas funciones recursivas se hace en T por medio de axiomas que describen reglas de reducción; cuando se consiguen buenas definiciones la axiomatización es por construcción suficientemente

---

(1) Los símbolos en los que en últimas quiere representarse O son los objetos que provee y manipula un lenguaje de programación en el que se quiere simular la realidad.

completa. Es decir, cada vez que se quiere analizar alguna característica de un objeto, los axiomas bastan para calcular la función recursiva correspondiente a la característica, i.e. para determinar un resultado en el conjunto de posibles valores de la función.

La discusión anterior apoya un supuesto metodológico importante en lo que aquí se desarrollará. Se supone del observador una idea clara del universo O que le permita definir operaciones en el TAD T clasificadas así:

<u>iniciales</u> : nombres para objetos "simples" de O (no construíbles a partir de otros objetos de O ya conocidos).

<u>constructoras</u> : nombres de funciones constructoras de objetos de O, que reciben argumentos que son objetos de O ya entendidos.

<u>simplificadoras</u>: transformaciones en Y que reflejan transformaciones correspondientes en O.

<u>analizadoras</u> : funciones sobre Y que permiten observar características de los objetos de O.

Se llama <u>selectora</u> a una operación simplificadora o analizadora.

Los identificadores de estas funciones y sus funcionalidades conforman la <u>signatura</u> del TAD T. Asociada a esta hay un lenguaje formal cuyos términos pueden clasificarse según el símbolo de función más externo que lo constituye.

El conjunto de términos se notará F_*. Se llamará Y_* al conjunto de los términos cuyo símbolo de función más externo corresponde al tipo de interés. Y es subconjunto de Y_*, y resulta natural definir (recursivamente, apoyándose en las definiciones recursivas sobre Y de las operaciones simplificadoras) una función normalizadora que asigne a cada elemento de Y_* una forma normal. Una reducción similar puede hacerse con términos de tipos diferentes al de interés.

Después será sencillo extender recursivamente las definiciones de las diferentes operaciones a todo F_*, simplemente cambiando cada argumento en F_* por su forma normal y aplicando las definiciones ya conocidas.

La hipótesis de trabajo principal es que la estructura del conjunto de nombres Y refleja la estructura de construcción de O. Algo más es deseable: si el observador puede distinguir dos objetos o1 y o2 debería poder distinguir sus nombres y1, y2. La única manera de distinguir dos objetos es observando que tienen diferente alguna característica, y ésto se refleja en el TAD Y cuando para alguna función selectora el valor cambia si se sustituye y1 por y2 (y los demás argumentos no se modifican). La igualdad puede definirse entonces como ausencia de diferencias, y así se

justifica definir en Y una relación de equivalencia (semántica) tal que $y1 \equiv y2$ si y1 y y2 no son distinguibles mediante selectoras en el sentido anotado. Esta relación es naturalmente extendible al conjunto de términos del tipo de interés Y_*.

La situación es verdaderamente satisfactoria cuando $.\equiv.$ captura en Y el conocimiento de O que tiene el observador. Algebraicamente: se quiere que $Y/\equiv$ y O sean "isomorfos", aunque tal noción debe dejarse también a la intuición, teniendo en cuenta el conocimiento apenas informal de O. Más precisamente, la definición del TAD T es satisfactoria cuando O es un álgebra (un modelo) inicial del TAD; cuando éste es el caso, dos elementos de Y son considerados diferentes si no se puede mostrar que son iguales (según $.\equiv.$).

Como notación, llámese T_{TDI} un TAD con tipo de interés TDI. Una <u>implementación</u> del TAD T_Y en otro TAD T_W define una subestructura del TAD T_W que es un álgebra inicial del TAD T_Y. Cuando la definición del TAD T_Y es satisfactoria en el sentido del párrafo anterior, la implementación define entonces una representación igualmente satisfactoria de O. Tal representación en el TAD T_W puede a su vez implementarse en otro TAD T_Z, y continuar esta cadena de representaciones hasta que se trate con objetos explícitos. En este artículo se considerarán explícitos los objetos primitivos que ofrezca el lenguaje de programación en el que se quiera representar el universo O. Este último paso en la cadena de representaciones no es esencialmente diferente de los anteriores si se exige un conocimiento formal de la clase de objetos que maneja el lenguaje de programación.

1. <u>Estructuras computacionales. Tipos abstractos de datos</u>

$E=\langle X,F\rangle$ es una <u>estructura computacional</u> (EC) si X es un conjunto de objetos explícitos y F un conjunto de operaciones sobre estos objetos. Típicamente E corresponde a los objetos provistos por un lenguaje de programación (v.gr. símbolos que representan números enteros, valores de verdad, etc.) y a las transformaciones sobre estos objetos predefinidas en el lenguaje (v.gr. suma de enteros, operaciones booleanas, etc.). Sobre X se supone definido un orden bien fundado $.<.$ (no hay cadenas infinitas descendentes).

$T=\langle X,F\rangle$ es un <u>tipo abstracto de datos</u> (TAD) cuando X es un conjunto de símbolos y F un conjunto de operaciones sobre estos símbolos y tal vez otros TADs. El conjunto X se llama el <u>tipo de interés</u> y se nota tdi(T)= X. Sobre X hay un orden bien fundado conocido $.<.$.

Un TAD es una EC o se construye a partir de TADs conocidos. A continuación se da una metodología para llevar a cabo una tal construcción de modo que el resultado sea una definición que tenga algunas propiedades muy deseables en la práctica (v.gr. completitud suficiente).

2. Metodología para construcción de TADs

La construccción de un TAD nuevo está motivada por el deseo de modelar un universo O entendido al menos intuitivamente. El conocimiento de O que tiene el observador puede entonces guiar la definición de T. Cada argumentación que se apoye en esta hipótesis de trabajo se señalará explícitamente en la siguiente descripción de la metodología de construcción de tipos como (HT)

2.1 Construcción de formas normales

(HT1) O es intuitivamente construíble. La construcción debe poderse hacer en forma recursiva, i.e. a partir de objetos simples o atómicos se construyen objetos más complejos. Los objetos simples y lo que se agrega a éstos para construír otros más complejos pueden describirse con símbolos tomados de TADs conocidos[2].

Sean T1,...,Tn TADs conocidos, con tdi(Tk)=Xk, k=1,...,n.
$N := \{1,\dots,n\}$, $N_0 := \{0,1,\dots,n\}$.

Por (HT1) se pueden definir dos conjuntos finitos de símbolos:

I : Conjunto de símbolos de operaciones iniciales. Con éstos se quieren nombrar los objetos simples de O. En general, los nombres pueden depender de los Xk's.

C : Conjunto de símbolos de operaciones constructoras. Con éstos se quieren construír los nombres de objetos complejos de O, a partir de nombres conocidos y algunos de los Xk's.

La aridad de los símbolos $f \in I \cup C$ se describe formalmente con dos funciones

$$m_X : I \cup C \dashrightarrow N^*,$$
$$m : I \cup C \dashrightarrow \underline{nat},$$

que indicarán respectivamente cuáles de los tipos primitivos Xk's y con qué multiplicidad el tipo de interés Y (que está por definirse!) participan en el dominio de definición del símbolo f.

Nótese que $m_X(f)$ es una sucesión finita de números j1,...,jk, $k \geq 0$. En este caso se notará $dom_X(f) := Xj1 \times \dots \times Xjk$.

Cuando k=0 se nota $m_X(f) :=$ nil, $dom_X(f) := \{\#\}$.

Se quiere además que $m(i)=0$ para $i \in I$, y $m(c)>0$ para $c \in C$.

Ahora se definen los conjuntos de nombres:

$$Y_0 := \{ i(x) \mid i \in I , x \in dom_X(i) \}$$

---

[2] Esta hipótesis se apoya en la Tesis de Church.

$Y_{j+1} := Y_j \cup \{c(x,\wp) \mid c \in C,\ x \in dom_X(c),\ \wp \in Y_j^{m(c)}\},\quad j \geq 0.$

$Y := \bigcup_{j \geq 0} Y_j$: el conjunto de nombres estándar o <u>formas normales</u>.

Cuando para $i \in I$ se tiene que $dom_X(i)=\{\#\}$, se abrevia el nombre 'i(#)' mediante 'i'. Más generalmente, siempre que en una lista de parámetros simbólicos de un nombre deba aparecer '#', el símbolo se omite. El significado de esta convención es justamente denotar constantes con respecto a los Xk's, que se dan con frecuencia en especial para las operaciones iniciales.

Y se ordena según la complejidad de la construcción de sus elementos, y dentro de un mismo nivel de complejidad, lexicográficamente. El orden resultante es bien fundado.

2.1.1 Ejemplo: Modelaje de stacks o pilas

Como ejemplo clásico en cualquier presentación de TADs se incluye el estudio del TAD Stack[X], que abstrae pilas de objetos sacados de un tipo primitivo X (conocido, recursivamente enumerable, bien fundado). Las pilas son estructuras de almacenamiento "LIFO" ("last-in-first-out"), donde el último objeto guardado es el primero que puede retirarse.

Por simplicidad se usará X para denotar el tipo primitivo y su tipo de interés. En este caso: X1=X, N={1}, $N_0=\{0,1\}$.

El stack más simple que se puede concebir es aquel que no tiene elementos. Es natural incluír un símbolo de operación que lo denote:

empty : símbolo de operacíon inicial. Denotará un stack vació de elementos de X.

A partir de un stack conocido se puede construír otro más complejo, agregando al primero un elemento más. La posibilidad de efectuar esta construcción justifica la inclusión de un símbolo de operación constructora:

push : símbolo de operación constructora. Operará sobre un stack s, y un elemento x.

Resumiendo: I = {empty}, C = {push}.

$m_X(\text{empty})=$<u>nil</u>, $m_X(\text{push})=\langle 1\rangle$,

m(empty)=0, m(push)=1,

Así: $dom_X(\text{empty})=\{\#\}$, $dom_X(\text{push})=X$.

$\text{Stack}_0 = \{\text{empty}\}$

$\text{Stack}_{j+1} = \text{Stack}_j \cup \{\text{push}(s,x) \mid s \in \text{Stack}_j,\ x \in X\}$

$Stack = \bigcup_{j \geq 0} Stack_j$: es el conjunto de formas normales para nombrar stacks.

El conjunto Stack se provee de un orden bien fundado, de acuerdo a la complejidad de la construcción y al orden del conjunto X.

2.2 Operaciones selectoras

(HT2) En O puede desearse transformar objetos en objetos. En T ésto se refleja en transformar los nombres correspondientes.

El poder analizar características de objetos de O se traduce en operaciones que analizan nombres y dan como resultado elementos de los Xk's. Esto supone que las partes de los objetos de O pueden nombrarse con símbolos de los Xk's.

Por (HT2) se definen conjuntos finitos de símbolos:

S : conjunto de símbolos de operaciones <u>simplificadoras</u>. Con éstos se quieren denotar las operaciones que transforman objetos de O en objetos de O sin crear nuevos objetos.

A : conjunto de símbolos de operaciones <u>analizadoras</u>. Con éstos se quieren denotar las operaciones que analizan características de los objetos de O.

$S \cup A$ es el conjunto de operaciones <u>selectoras</u>.

$F := I \cup C \cup S \cup A$ es el conjunto de símbolos de operaciones o funciones de T.

La aridad de los símbolos $f \in S \cup A$ se describe extendiendo las funciones m_X y m:

$$m_X: F \longrightarrow N^*,$$
$$m: F \longrightarrow \underline{nat}.$$

Para $f \in S \cup A$ se quiere que $m(f)>0$.

El rango de cada $f \in F$ puede definirse mediante una función

$$r: F \longrightarrow N_0,$$

tal que $r(f)=0$, si $f \in I \cup C \cup S$
$r(f)>0$, si $f \in A$.

La notación $dom_X(f)$ se extiende para $f \in S \cup A$, y se define
$ran(f) := Y$, si $r(f)=0$
Xk, si $r(f)=k, k>0$.

Asociada a cada $f \in S \cup A$ se define una función recursiva

$$f^*: dom_X(f) \times Y^{m(f)} \longrightarrow ran(f)$$

mediante axiomas que se apoyan en el orden bien fundado de Y. La definición de cada f* exige el conocimiento de la transformación correspondiente en los objetos de O (HT2).

2.2.1 Ejemplo: Modelaje de stacks o pilas (continuación)

En la manipulación de stacks se utilizan generalmente dos funciones selectoras, una simplificadora y una analizadora:

pop(s): denota el stack resultante de retirar del stack s el último elemento ingresado. Su acción sobre empty, el stack vacío, no se define.

top(s): denota el último elemento ingresado al stack s. También top(empty) se considera indefinido.

Entonces se definen los conjuntos: S = {pop}, A = {top}.

Y además:

m_X(pop)=<u>nil</u>, m_X(top)=<u>nil</u>, m(pop)=1, m(top)=1,

r(pop)=0, r(top)=1.

Entonces se necesita definir dos funciones recursivas

pop*: Stack ---> Stack
top*: Stack ---> X ,

cuya acción debe explicarse efectivamente, mediante axiomas que describan cómo operan sobre las formas normales de Stack.

Los siguientes axiomas definen recursivamente pop* y top*:

(S1) pop(empty) = $\perp$
(S2) pop(push(s,x)) = s
(S3) top(empty) = $\perp$
(S4) top(push(s,x)) = x .

$\perp$ es una convención para denotar "indefinido" o "error". Ambos casos se consideran "anormales", y el uso de $\perp$ como argumento de cualquier operación da como resultado $\perp$.

2.3. Términos

En el presente contexto la cuádrupla (F,m_X,m,r) conforma la llamada <u>signatura</u> del tipo T. La signatura define naturalmente un conjunto F_* de <u>terminos</u> que constituyen un lenguaje formal para hablar del universo O.

El conjunto de términos se define inductivamente así:
(Sea Ak:= {a $\in$ A | r(a)=k })

$Xk_{[0]}$:= Xk, k=1,...,n

$Y_{[0]}$:= Y_0.

$Xk_{[j+1]}$:= $Xk_{[j]}$ U {a(x,$\mathfrak{y}$) | a $\in$ Ak, m_X(a)=q1...qp,

x $\in$ $Xq1_{[j]}$x...x$Xqp_{[j]}$, $\mathfrak{y}$ $\in$ $Y^{m(a)}_{[j]}$ }

para k=1,...,n.

$$Y_{[j+1]} := Y_{[j]} \cup \{f(x,\mathfrak{y}) \mid f \in I \cup C \cup S,\ m_X(s)=q1\ldots qp,\ x \in Xq1_{[j]}x\ldots xXqp_{[j]},\ \mathfrak{y} \in Y^{m(s)}_{[j]}\}.$$

para $j \geq 0$.

Ahora pueden definirse:

$Xk_* := \bigcup_{j \geq 0} Xk_{[j]}$: el conjunto de los <u>Xk-términos</u>,

$Y_* := \bigcup_{j \geq 0} Y_{[j]}$: el conjunto de los <u>Y-términos</u>,

$F_* := \bigcup_{1 \leq k \leq n} Xk_* \cup Y_*$: el conjunto de los <u>terminos</u>.

2.3.1 Modelaje de stacks o pilas (continuación)

Los términos no son otra cosa que las fórmulas sintácticamente bien formadas de acuerdo a las funcionalidades de las operaciones. Con lo hasta aquí definido (formas normales, símbolos de operaciones,...), los X-términos son las expresiones bien formadas que tienen top como símbolo más externo, y los Stack-términos son las que tienen empty, push o pop como símbolo más externo.

Cada término debería nombrar un objeto; la completitud suficiente se refiere justamente al hecho de saber a qué objeto se refiere cada término que no es del tipo de interés. Por ejemplo, debería ser claro que el término

top(pop(push(push(empty,a),b)

se refiere al objeto primitivo a.

2.4 Reducción de términos

Se define una <u>reducción</u>

$$.' : F_* \longrightarrow \bigcup_{1 \leq k \leq n} X_k \cup Y$$

de la siguiente manera: (Notación: $(z_1,\ldots,z_u)' = (z_1',\ldots,z_u')$)

Si $t \in Xk_{[0]}$: $t' := t$, $k=1,\ldots,n$.

Si $t \in Y_{[0]}$: $t' := t$.

Si $t \in Xk_{[j+1]} \setminus Xk_{[j]}$, $t=a(x,\mathfrak{y})$: $t' := a^*(x',\mathfrak{y}')$, $k=1,\ldots,n$.

Si $t \in Y_{[j+1]} \setminus Y_{[j]}$, $t=s(x,\mathfrak{y})$: $t' := s^*(x',\mathfrak{y}')$

para $j \geq 0$.

Así, cada Xk-término reduce a un elemento de Xk, y cada Y-término a una forma normal en Y.

2.5 Definición del nuevo TAD T

A cada símbolo de operación $f \in F$ se puede asociar una funcíon recursiva también denotada f:

$$f: dom_X(f) \times Y^{m(f)} \longrightarrow ran(f)$$

así:

Para $i \in I$: $i: dom_X(i) \longrightarrow Y$
$x \longmapsto i(x)$

Para $c \in C$: $c: dom_X(c) \times Y^{m(c)} \longrightarrow Y$
$(x,\mathfrak{y}) \longmapsto c(x,\mathfrak{y})$

Para $f \in S \cup A$: $f := f^*$.

El conjunto de funciones asociadas se nombra también F.

Nótese que Y es un conjunto de cadenas de símbolos de la forma $i(x)$ o bien $c(x,\mathfrak{y})$. Las imágenes por las funciones $i \in I$ y $c \in C$ son entonces cadenas de símbolos. Para las $f \in S \cup A$ las imágenes son elementos de los Xk's (que a su vez pueden ser cadenas de símbolos, pero ésto es externo a la definición del nuevo TAD).

Ahora TAD T := <Y,F> es el nuevo TAD. Por construcción (HT1,HT2) O es un modelo intuitivo de T. Además T es <u>suficientemente completo</u>, porque en virtud de 2.4 todo término de la forma $a(x,\mathfrak{y})$ reduce recursivamente a un objeto primitivo.

2.5.1 Modelaje de stacks o pilas (continuación)

La siguiente notación resume la definición del TAD Stack[X]:

```
Tipo Stack[X]
Operaciones
*empty:              ---> Stack
*push  : Stack x X  ---> Stack
 pop   : Stack      ---> Stack
 top   : Stack      ---> X
Axiomas
(S1) pop(empty) = ⊥
(S2) pop(push(s,x)) = s
(S3) top(empty) = ⊥
(S4) top(push(s,x)) = x
finTipo
```

Las operaciones iniciales y constructoras son marcadas con '*'. Esto permite conocer tácitamente las formas normales de los stacks. Es decir se puede enunciar y mostrar fácilmente el siguiente resultado:

<u>LFN(Stack)</u> (Lema de forma normal para Stack)
Todo $s \in$ Stack tiene exactamente una de las siguientes formas:
 i) empty
ii) push(s1,x), con $s1 \in$ Stack, $x \in X$.

Solo hay axiomas para las operaciones selectoras. Debe haber un axioma por cada posible forma normal del tipo de interés, con lo cual se garantiza la buena definición de cada selectora. Las funciones son "totales", en el sentido de que se explican para todas las posibles instancias del dominio; cuando en algún argumento deben quedar indefinidas, ésto se hace explícito (S1,S3).

3. Equivalencia

Considérese el TAD T = <Y,F> de 2. La hipótesis de trabajo (HT1) no exige que cada objeto de O tenga un nombre único, y en la práctica es usual que un mismo objeto tenga varios nombres (v.gr. la notación {1,2,3} denota el mismo conjunto que {2,1,3}). Enseguida se quiere establecer cuándo dos nombres se refieren a un mismo objeto.

La siguiente notación es útil:
Para $\hat{y} = (y_1,\ldots,y_{m-1}) \in Y^{m-1}$, $y_0 \in Y$, $1 \leq j < m$, se define:

$$[\hat{y},y_0]_j := (y_1,\ldots,y_{j-1},y_0,y_{j+1},\ldots,y_{m-1}) \in Y^m.$$

3.1 Definición

Sean y1, y2 $\in$ Y.

i) Sea $f \in S \cup A$.

$y1 =_f y2$ ssi y1 es f-equivalente a y2

ssi Para todo $x \in dom_X(f)$, $\hat{y} \in Y^{m(f)-1}$, $1 \leq j < m(f)$:

$$f(x,[\hat{y},y1]_j) = f(x,[\hat{y},y2]_j).$$

ii) Sea $K \subseteq S \cup A$.

$y1 =_K y2$ ssi y1 es K-equivalente a y2

ssi Para toda $f \in K$: $y1 =_f y2$.

iii) $y1 \equiv y2$ ssi y1 es (semánticamente) equivalente a y2

ssi $y1 =_{S \cup A} y2$.

iv) $K \subseteq S \cup A$ es un caracterizador (de Y), si para todo par y1,y2 $\in$ Y:

$$y1 =_K y2 \Rightarrow y1 \equiv y2.$$

∎

Si y1, y2 nombran objetos o1,o2 respectivamente, $y1 =_f y2$ si la selectora f no los distingue. Por (HT2) ésto significa que el observador tampoco distingue la característica correspondiente entre o1 y o2. Si además $y1 \equiv y2$, con (HT2) se deduce que o1 y o2 son indistinguibles para el observador.

De ésta manera $Y/\equiv$ es intuitivamente isomorfo a O. Cuando en Y

$\equiv$ se considera como igualdad algebraica, O es un modelo inicial de T puesto que dos objetos son diferentes si no se puede probar que son iguales. La existencia de un caracterizador no trivial reduce el número de condiciones que deben verificarse para mostrar equivalencia.

3.1.1 Modelaje de stacks o pilas (continuación)

Para el TAD Stack[X] hay una operación simplificadora (pop), y una analizadora (top). Con estas dos funciones se define la equivalencia de dos stacks. Entonces:

s1 $\equiv$ s2 ssi pop(s1) $\equiv$ s2 and top(s1) = top(s2)
ssi s1 y s2 tienen el mismo elemento en el tope, y al retirarlo de ambos quedan stacks equivalentes.

El que esta sea una definición razonable hace creer que el TAD Stack[X] está bien definido, y que Stack/$\equiv$ es intuitivamente isomorfo al modelo que el diseñador pretende abstraer.

No hay caracterizadores no triviales en este caso. Dos stacks pueden tener iguales sus topes y diferir en el resto o viceversa. Sin embargo, ésto justifica la escogencia que se hizo de las funciones selectoras: cualquier otra definición de una selectora que no distinguiera alguna característica no considerada sería redundante para la determinación de la equivalencia de dos stacks.

4. Implementaciones

Sean $T_Y = \langle Y, F_Y \rangle$, $T_W = \langle W, F_W \rangle$ dos TADs construídos a partir de otros TADs conocidos T1,...,Tn, con $tdi(Tk)=X_k$, k=1,...n.

4.1 Definición

Una implementación Imp = (ß,P) de T_Y en T_W consta de :

i) Una función recursiva parcial sobreyectiva

$$ß: W \longrightarrow Y$$

llamada la representación (de W como Y).
Para $\hat{w}=(w1,...,wp)$, $ß(\hat{w}) := (ß(w1),...,ß(wp))$.

ii) Para cada $f \in F_Y$ una función recursiva

$$f_W: dom_X(f) \times W^{m(f)} \longrightarrow Z,$$

donde $Z = ran(f)$, si $ran(f) \neq Y$
$= W$, si $ran(f)=Y$,

tal que: $f(x,ß(\hat{w})) = f_W(x,\hat{w})$, si $ran(f) \neq Y$
$= ß(f_W(x,\hat{w}))$, si $ran(f)=Y$.

$P := \{f_W \mid f \in F_Y\}$

Cuando T_Y es una estructura computacional hay un lenguaje de programación que la manipula. Las funciones f_W son calculadas por programas en este lenguaje.

Si el lenguaje de programación es de tipo funcional (v.gr. LISP) f_W es una función cuya definición está naturalmente guiada por la de f. Una tal implementación es útil como prototipo del TAD T_Y y para verificar experimentalmente la buena definición de T_Y.

Sin embargo, cuando se desea una implementación eficiente es más aconsejable un lenguaje de tipo imperativo (v.gr. PASCAL). Cada función es calculada con un procedimiento (3) pf_W cuya especificación está determinada por la definición de f en T_Y.

Más precisamente, si F_Y = I U C U S U A como en 2.:

Para i ∈ I, i_W se calcula con

```
proc piW (x: domX(i); var pw: W);
     {Pre : x=x0 }
     {Post: ß(pw) ≡ i(x0) }.
```

Para c ∈ C, c_W se calcula con

```
proc pcW (x: domX(c); var ŵ: W^(m(c)-1), pw: W);
     {Pre : x=x0, ŵ=ŵ0, pw=pw0 }
     {Post: ß(pw) ≡ c(x0,ß(ŵ0),ß(pw0)) }.
```

(Nótese el cálculo de c sobre uno de los parámetros.)

Para s ∈ S, s_W se calcula con

```
proc psW (x: domX(s); var ŵ: W^(m(s)-1), pw: W);
     {Pre : x=x0, ŵ=ŵ0, pw=pw0 }
     {Post: ß(pw) ≡ s(x0,ß(ŵ0),ß(pw0)) }.
```

Para a ∈ A, a_W se calcula con

```
proc paW (x: domX(s); var ŵ: W^m(a), px: ran(a));
     {Pre : x=x0, ŵ=ŵ0 }
     {Post: px ≡ a(x0,ß(ŵ0)) }.
```

---

(3) Eventualmente podrían usarse funciones, pero se adopta esta convención para facilitar la comparación con lenguajes como PASCAL, donde el resultado de una función solo puede ser de un tipo predefinido simple.
También se usarán las convenciones de PASCAL para notar el paso de parámetros.

4.1.1 Modelaje de stacks o pilas (continuación)

Para simplificar, supóngase X ≡ integer, el tipo primitivo de enteros PASCAL, y que se quiere implementar Stack[integer] en PASCAL.

Se necesita una función de representación, parcial, sobreyectiva:
ß: Estructuras de datos en PASCAL ----> Stack[integer],
y para cada operación f en Stack una realización PASCAL de la misma.

La definición de ß se deja usualmente a la experiencia de quien hace la implementación. En el caso de los stacks que se están modelando se puede observar, por ejemplo, que son estructuras no acotadas, y que su imagen PASCAL debería poder cambiar dinámicamente. Entonces es fácil pensar en listas simplemente encadenadas, donde cada elemento señala al anteriormente ingresado; el stack empty se representa con una lista vacía.

Una definición menos intuitiva de ß puede establecerse apoyándose en el LFN(Stack). Es decir, se busca definir, recursivamente, estructuras PASCAL para cada forma normal posible de Stack.

Cuando la experiencia es quien más ayuda a la definición de ß, las funciones selectoras son relativamente sencillas de programar, porque el observador visualiza en la estructura de datos que define los aspectos que le sirven para analizar el objeto. Por el contrario, cuando es el LFN la guía principal para el diseño de ß se fija además, implícitamente, la manera de realizar las funciones iniciales y constructoras.

Se puede ser tan formal como se quiera en la comprensión de ß; mientras más se conozca formalmente (menos intuitivamente), resultará más sencilla la realización de las operaciones del tipo. Sin embargo, es bueno encontrar un nivel de comprensión que permita justificar que la implementación es correcta sin caer en detalles técnicos exagerados.

Las operaciones del TAD Stack se realizan mediante procedimientos PASCAL tales que (pstack es el "type" PASCAL correspondiente a 'stack'):

```
procedure pempty (var p: pstack);
    {Pre : T }
    {Post: ß(p) ≡ empty }

procedure ppush (var p: pstack; x: integer);
    {Pre : p = p0 and x = x0 }
    {Post: ß(p) ≡ push(ß(p0),x0) }

procedure ppop (var p: pstack);
    {Pre : p = p0 }
    {Post: ß(p) ≡ pop(ß(p0)) }
```

```
procedure ptop (var p: pstack; x: integer);
    {Pre : p = p0 }
    {Post: p = p0 and x = top(ß(p0)) }
```

4.2 La programación de una implementación en un lenguaje imperativo

Las especificaciones de procedimientos mencionadas en 4.1 son suficientes para programar una implementación del TAD T_Y en una EC $E_W = \langle W, F_W \rangle$. Es decir, si se tiene una metodología de desarrollo de programas a partir de especificaciones (cf. [Gri81], [Boh86]), lo que sigue entonces es un problema de programación.

Enfrentando un poco este problema se descubre que la principal dificultad es expresar las postcondiciones de una manera "cercana" al lenguaje de programación. O sea, puede ser necesario refrasear las postcondiciones de manera que se vea más claramente lo que se desea, para entonces aplicar las técnicas de desarrollo de programas que se estimen convenientes.

Típicamente cada postcondición involucra al menos una referencia a la función de representación ß. Esto sugiere la conveniencia de establecer con claridad esta función antes de atacar el problema de programar los procedimientos. Esta es la manera "clásica" de construír implementaciones: primero se diseñan las estructuras de datos que representarán los objetos, y después se programan las operaciones que manipulan estas estructuras.

Ahora bien, como ß es una función recursiva parcial sobreyectiva, lo usual es que esté definida apoyándose en el orden bien fundado de Y. Esto es, es de esperarse que se pueda decir, primero, cuál $w \in W$ es preimagen por ß de cada $i \in I$, y con esta información definir inductivamente la preimagen de cada y ($=c(x,\hat{y})$) en Y. De este modo, los procedimientos pi_W, $i \in I$, y pc_W, $c \in C$, resultan particularmente fáciles de implementar.

Las postcondiciones de los procedimientos pf_W, $f \in S \cup A$, se simplifican si se usan formas reducidas para x_0, $\hat{w}_0$ y pw_0, y se recuerda que en estos casos la función f coincide con la función f^*. Como también se está suponiendo una definición recursiva de estas funciones (apoyada en el orden bien fundado de Y), es entonces posible "acercar" la postcondición al lenguaje de programación.

El formalismo deja entrever otro orden para adelantar la implementación, menos "natural", pero igualmente válido. Supóngase que se tiene una idea de cómo son las especificaciones de las operaciones selectoras, aunque no se tenga claro cómo debería ser la implementación de las funciones iniciales y constructoras. Entonces se dispondría de un método para calcular las funciones $f \in S \cup A$, y por ende para decidir los predicados que involucren equivalencias semánticas. En particular, cada ß(pw) en las postcondiciones de los pi_W's y los pc_W's se puede ver como una incógnita, que una vez conocida establece una definición para ß, y da indicios de cómo programar estos procedimientos.

Esta última manera de llevar a cabo la implementación es especialmente viable cuando no hay operaciones simplificadoras. Cuando éste no es el caso, las postcondiciones de las operaciones simplificadoras incluyen equivalencias que a su vez dependen del conocimiento de las mismas operaciones simplificadoras. La aparente circularidad es inexistente si no hay operaciones simplificadoras.

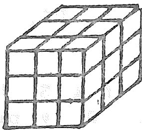
En cualquier caso el diseño se facilita si se dispone de un conjunto caracterizador no trivial que a su vez simplifique el manejo de la relación de equivalencia .≡.

6. Un ejemplo más complejo: El TAD Cubo

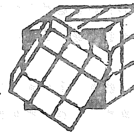
A continuación se incluye un ejemplo "no convencional" como ilustración de la metodología propuesta. Como se podrá observar, en la práctica no es necesario llevar a extremos el formalismo (v. gr. distinguir entre símbolos de operación y operaciones, cuidar la definibilidad de dominios y rangos, etc.), sin descuidar el rigor en el diseño y la implementación.

6.0 Conocimiento intuitivo

El cubo de Rubik es un rompecabezas tridimensional inventado por el profesor húngaro Ernõ Rubik. Se trata de un cubo constituído a su vez (aparentemente) por 27 elementos cúbicos que rotan en 3 direcciones por planos de 9 elementos. Algo como:



Posición normal



Rotación de la cara frontal

Las pequeñas caras de cada uno de los elementos cúbicos se llamarán "carillas", y están coloreadas de modo que hay una configuración del cubo que tiene cada cara (de 9 carillas) de un mismo color, y todas las caras tienen un color diferente.

El acertijo de Rubik consiste en, dada una configuración arbitraria del cubo, encontrar las rotaciones necesarias para llegar a una configuración en la que las 9 carillas de cada cara son de un mismo color.

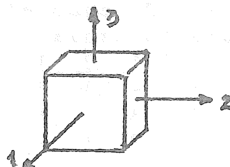
El TAD Cubo no pretende resolver el acertijo, sino modelar el funcionamiento del cubo de Rubik, i.e. las rotaciones y cambios de colores en las carillas. Sin embargo, el disponer de un lenguaje para hablar de estas características permite plantear for-

malmente el acertijo y buscar soluciones del mismo de una manera sistemática.

La metodología supone una idea intuitiva del objeto, que captura los aspectos y sus relaciones. Es quizás imposible separar estas ideas de lo que en el futuro será una representación del objeto, ya que al entenderlo debe existir alguna representación mental del mismo en la cabeza del observador.

En este caso es natural pensar el cubo como una caja de 3x3x3 enmarcada en un sistema de coordenadas cartesianas. Los elementos cúbicos se pueden identificar con su posición en estas coordenadas, y las carillas de cada elemento de acuerdo al eje en dirección al cual se mira el elemento.

Más concretamente, se toman como referencia tres ejes ortogonales fijos, numerados 1, 2, 3. Un objeto q ∈ Cubo se observa siempre en el origen de ese sistema de coordenadas, así:



Cada elemento cúbico tiene una posición

p = (p[1],p[2],p[3]),

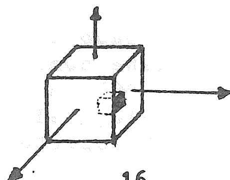
donde p[i] es un número en {-1,0,1}, i=1,2,3.

Cada elemento cúbico está compuesto de seis carillas. Para la descripción del cubo de Rubik se puede notar que un elemento cúbico tiene coloreadas a lo más tres de estas carillas, y más aún, a lo más una carilla coloreada en la dirección de cada eje del sistema de coordenadas. Entonces se puede simplificar un poco la descripción, haciendo que un elemento cúbico esté descrito por tres carillas, una para cada eje coordenado, algunas de las cuales están coloreadas y otras pueden tener un color neutro o indefinido.

Se puede decir, acercándose un poco más al cubo de Rubik, que un objeto q ∈ Cubo se compone de carillas identificables por una posición p -definida como arriba- y un eje que denote la dirección en que debe mirarse el elemento cúbico descrito por p.

De esta manera, una carilla es un par (p,e), donde p es una posición y e es un eje coordenado.

Por ejemplo, en el siguiente dibujo, la carilla sombreada se identifica con el par ((1,1,0),2):



Las siguientes notaciones son importantes:

Eje = {1,2,3} : el conjunto de ejes coordenados.

Dsp = {-1,0,1} : el conjunto de desplazamientos posibles con respecto a cada e ∈ Eje.

Posicion = Dsp^{Eje}

= {(p[1],p[2],p[3]) | p[i] ∈ Dsp, i=1,2,3 }

: el conjunto de posiciones de los elementos cúbicos de un cubo abstracto.

Pinta = {a,v,n,r,b,m}

: el conjunto de colores:

a: azul v: verde n: naranja
r: rojo b: blanco m: amarillo.

Además, para facilitar la descripción del tipo abstracto Cubo se utilizarán las siguientes funciones:

comp: Eje x Eje ---> Eje

donde comp(e1,e2) es el eje diferente a e1 y e2 (si e1=e2, comp(e1,e2) no se define!).

i.e. comp tiene la siguiente tabla:

	1	2	3
1	-	3	2
2	3	-	1
3	2	1	-

r: Posicion x Eje ---> Posicion

donde r(p,e) es una rotación de la posición p, con eje e, 90° en el sentido de las manecillas del reloj.

```
i.e.  r(p,e) = if e=1 --> ( p[1], p[3],-p[2])
               ▯ e=2 --> (-p[3], p[2], p[1])
               ▯ e=3 --> (-p[2], p[1], p[3])
               fi.
```

Obsérvese que Eje , Posicion, Pinta, comp, r son nociones correspondientes a tipos primitivos que en el momento no interesa estructurar. Basta considerarlos conocidos e implementables.

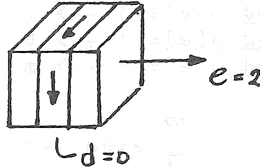
6.1 Construcción de formas normales

Cualquier cubo q ∈ Cubo se construirá a partir de un cubo inicial estándar, mediante rotaciones. Las cosas se simplifican si el cubo inicial estándar tiene los colores dispuestos según "la solución" del acertijo. Esto da lugar a una operación inicial, llamada inic, y a una constructora , llamada rote, entendidas de la siguiente manera:

6.1.1 inic : denota el cubo inicial estándar, que tiene un mismo color en todas las carillas de cada cara.

6.1.2 rote : es una función que, dado un cubo q, un eje e, y un desplazamiento d, tiene como valor el cubo resultado de rotar con eje e, 90° en el sentido contrario de las manecillas del reloj, el plano en q que tiene un desplazamiento d con respecto al eje e.

Por ejemplo, rote(q,2,0) denota un cubo como q en el que el plano central con respecto al eje 2 se ha rotado 90° en el sentido contrario de las manecillas del reloj:



6.1.3 Como siempre, después de conocer las operaciones iniciales y constructoras es posible enunciar y demostrar el lema de forma normal del tipo de interés:

LFN(Cubo) (Lema de forma normal de Cubo)
Todo q ∈ Cubo es de una de las siguientes formas:
i) inic
ii) rote(q1,e,d), con q1 ∈ Cubo, e ∈ Eje, d ∈ Dsp.

6.2 Operaciones selectoras

Hay una función color, para denotar el color de cada carilla de un cubo.

Adelantándose un poco a lo que será la relación de equivalencia semántica derivada de las operaciones selectoras, es natural esperar que dos cubos sean equivalentes cuando sus carillas correspondientes tengan el mismo color. Así se justifica el hecho de que color sea la única función selectora en el TAD.

Es posible que una implementación requiera (para su eficiencia) un lenguaje más rico, v.gr. considerar rotaciones en el sentido de las manecillas del reloj. Estas pueden introducirse fácilmente una vez implementadas las operaciones hasta aquí mencionadas.

6.3 La definición del TAD Cubo

La siguiente definición resume el diseño del TAD Cubo:

Tipo: Cubo

Operaciones:

```
*inic :                          ---> Cubo
*rote : Cubo x    Eje     x Dsp ---> Cubo
 color: Cubo x Posicion x Eje   ---> Pinta
```

```
Axiomas:
(C1) color(inic,p,e)
          = if e=1 and p[e]= 1 --> a
            ▯ e=1 and p[e]=-1 --> v
            ▯ e=2 and p[e]= 1 --> n
            ▯ e=2 and p[e]=-1 --> r
            ▯ e=3 and p[e]= 1 --> b
            ▯ e=3 and p[e]=-1 --> m
            fi
(C2) color(rote(q,e,c),p,em)
      = if p[e] = c --> if e=em --> color(q,r(p,e),em)
                        ▯ e≠em --> color(q,r(p,e),comp(e,em))
                        fi
        ▯ p[e] ≠ c --> color(q,p,em)
        fi
finTipo
```

Los axiomas son más complicados que los correspondientes al TAD Stack[X]. En el lado izquierdo se permiten expresiones *if...fi* y recursión.

6.4 Equivalencia

q1 ≡ q2 ssi Para toda carilla (p,e):
color(q1,p,e) = color(q2,p,e).

6.5 Implementación

Se esboza aquí una implementación PASCAL del TAD Cubo.

El primer paso es establecer la representación para los conjuntos y operaciones considerados primitivos al entender intuitivamente los cubos de Rubik.

Por ejemplo:

```
type Eje = 1..3;
     Dsp = -1..1;
     Posicion = array [Eje] of Dsp;
     Pinta = (a,v,n,r,b,m,indef)
      /* 6 colores definidos: azul,verde,...,amarillo.
         indef: color indefinido, para carillas interiores */
```

e implementar comp y r de acuerdo a lo definido en 2.0. Para r se necesita un procedimiento, puesto que el resultado no es un "type" simple, algo como

```
procedure pr (var pos_inicial,e,pos_final);
                   . . .
```

El segundo paso es definir la función de representación ß. Ya con las definiciones anteriores es natural escoger como representación PASCAL para un cubo una matriz de 3x3x3 de elementos cúbicos; cada elemento cúbico se representa a su vez con un vector de

3 puestos, cada uno de los cuales representa el color de la carilla visible del elememto en la dirección de un eje, si ésta lo es. Si no hay carilla visible se usa el color 'indef'.

Más exactamente, se incluye una definición
<u>type</u> pcubo = <u>array</u> [Dsp,Dsp,Dsp] <u>of</u> <u>array</u>[Eje] <u>of</u> Pinta

(sería deseable poder decir en PASCAL
<u>array</u> [Posicion] <u>of</u> <u>array</u>[Eje] <u>of</u> Pinta,
pero ésto no es sintácticamente posible).

Aquí se da una situación interesante: la operación selectora 'color' es más fácil de implementar que las constructoras 'inic' y 'rote', porque color(ß(pq),p,e) coincide con pq[d1,d2,d3][e], cuando p es la posición descrita por el vector de desplazamientos [d1,d2,d3]. O sea, el cálculo del color de una carilla es directamente consultable en la estructura que representa el cubo. De hecho, es inclusive ineficiente escribir código para esta función.

La operación 'inic'no es otra cosa que una inicialización de la estructura que representa el cubo que pasa como parámetro. Concretando:

```
procedure pinic (var pq: pcubo)
    {Pre : T }
    {Post: ß(pq) ≡ inic }
```

Ahora, la postcondición significa que al terminar pinic, la variable pq, transformada por la función de representación ß, es equivalente a inic. Es decir:

"Para toda posición p, y para todo eje e:
color(ß(pq),p,e) = color(inic,p,e). "

O sea:

"Para toda posición [d1,d2,d3], y para todo eje e:
pq[d1,d2,d3][e] = color(inic,[d1,d2,d3],e). "

Ahora bien, el axioma (C1) de la definición del TAD Cubo se refiere justamente al color del cubo inic. Así, para cumplir con la postcondición de pinic es suficiente observar este axioma y efectuar en la estructura de datos lo anotado allí.

Una discusión análoga se puede adelantar para implementar el procedimiento correspondiente a la operación constructora 'rote'. Se necesita algo del estilo

```
procedure prote (var pq: pcubo; e: Eje; d: Dsp);
    {Pre : pq = pq0 and e = e0 and d = d0 }
    {Post: ß(pq) ≡ rote(ß(pq0),e0,d0) }.
```

Y la postcondición se refrasea así

"Para toda posición [d1,d2,d3], y para todo eje e:
pq[d1,d2,d3][e] = color(rote(ß(pq0),e0,d0),[d1,d2,d3],e)."

El axioma (C2) de la definición del TAD Cubo sirve para desarrollar el código PASCAL para prote.

7. Conclusiones

Las ideas expuestas configuran la base teórica para una metodología de diseño e implementación de TADs.

El posible éxito de la aplicación de la metodología depende fundamentalmente del buen conocimiento intuitivo del universo que se pretende modelar. Una tal exigencia no parece muy restrictiva en la práctica; por el contrario, se puede afirmar que si no se puede hablar con rigor de un universo, mal puede pretenderse intentar un modelo formal que lo describa.

Los TADs diseñados son por construcción suficientemente completos, de modo que la obtención de esta importante propiedad deja de ser un problema. La dificultad se translada al hecho de poder dar axiomas que definan funciones recursivas sobre dominios dotados de órdenes bien fundados, y este problema es más fácil de atacar, sobre todo si, como se supone, se cuenta con una buena visión intuitiva de los objetos que se quieren modelar.

Es importante recalcar que el conjunto subyacente de un TAD así diseñado está constituído por cadenas de símbolos que vienen a ser nombres estándar o formas normales para los objetos del universo modelado. Aunque parezca artificial (y aún lejano de la práctica), esta manera de considerar los TADs es precisamente un indicio para establecer qué tan bueno es el conocimiento intuitivo que del universo a modelar tiene el diseñador: los elementos de un conjunto "bien conocido" deben poderse bautizar de alguna manera sistemática (un matemático diría "enumerar recursivamente").

La implementación de un TAD en una EC imperativa merece, por supuesto, un estudio mucho más profundo que el aquí anotado. Puede resultar especialmente interesante el caso en que las implementaciones de las operaciones iniciales y constructoras se apoyan en las de las funciones selectoras. Un mejor conocimiento de la relación de equivalencia semántica (por ejemplo buscando un conjunto caracterizador pequeño, y si se puede minimal) facilita en cualquier caso la verificación de la corrección de una implementación del TAD.

****

BIBLIOGRAFIA

[Bau82] Bauer F.L., Wössner H., Algorithmic Language and Program Development, Springer Verlag, 1982.

[Boh86] Bohórquez, J., "Cálculo de programas: una metodología precisa para el diseño de programas de computador", Memo de Investigación No.6, Universidad de los Andes - CIFI, Bogotá, 1986.

[Car86] Cardoso, R., "Diseño e implementación de tipos abstractos de datos", Memo de Investigación No. 9, Universidad de los Andes-CIFI, Bogotá, 1986.

[Car86] Cardoso, R., "Prácticas en tipos abstractos de datos", Memo de investigación (por aparecer), Universidad de los Andes-CIFI, Bogotá, 1986.

[Gri81] Gries, D., The Science of Programming, Springer Verlag, 1981.

[Gut77] Guttag, J.V., Horowitz E., Musser D.R., "The Design of Data Type Specifications", en Current Trends in Programming Methodology, R.T. Yeh (edt.), Prentice-Hall, 1978.

[Kuc85] Kucherov, G.A., "An Algorithm to recognize Sufficient Completeness of Algebraic Specifications on an Abstract Data Type", traducido de Programmirovanie, No. 4, pp.3-12, Julio-Agosto 1984.

[Pes83] Pessoa, E.P., Veloso, P.A., Introdução à especificação e implementação de tipos abstratos de dados, Monografias em Ciencia da Computação, No. 20, Pontifícia Universidade Católica do Rio de Janeiro, 1983.

****