ESPECIFICACIONES ALGEBRAICAS DE TIPOS ABSTRACTOS DE DATOS PARA UN CURSO MEDIO DE PROGRAMACION

Alvaro Tasistro
Alfredo Viola
Facultad de Ingeniería
Montevideo - *Uruguay*

1.-Antecedentes.-

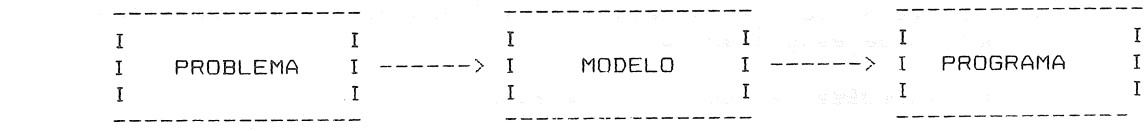
A partir del segundo semestre de 1985 comienza a aplicarse un Plan de Ajustes de Contenidos a las materias de la carrera de Ingenieria en Sistemas de Computacion, ideado para contrarrestar parcialmente el estancamiento de mas de diez anos en el desarrollo universitario de esta area de conocimiento provocado por la politica aplicada durante el regimen de facto.

Este Plan de Ajustes guio la elaboracion de nuevos cursos de Programacion basandose en un enfoque ingenieril de la disciplina que se describe a continuacion.-

1.1.- Orientacion General de los cursos de Programacion

Se visualiza la programacion como un proceso de transformacion de enunciados informales de problemas en programas aptos para ser ejecutados en computadores.

Una simplificacion esquematica podria ser la siguiente (AHU83):



En el proceso de transformacion de problemas enunciados originalmente de manera natural-informal a programas para computador se considera una etapa de modelacion o especificacion formal, con las siguientes caracteristicas:

- Se genera un enunciado del problema en terminos de propiedades consideradas relevantes de los objetos presentes en la proposicion original y no en terminos de los objetos relativos al computador o lenguaje de programacion en que se resolvera el problema. En este sentido, la especificacion formal es de "alto **nivel de abstraccion**".

- Es un enunciado del problema en un lenguaje formal, caracteristica que le es comun con el lenguaje de programacion final y que lo distingue del lenguaje natural ambiguo. Esto, ademas de proporcionar rigor al enunciado posibilita la verificacion formal de la adecuacion de la implementacion final.

Idealmente podria visualizarse la programacion como la modelizacion (simplificacion - formalizacion) de una situacion propuesta (problema), seguida de su transformacion en un enunciado equivalente logicamente, escrito en un cierto lenguaje de programacion.

Segun esta vision (extrema) el esfuerzo central estaria

concentrado en la especificacion del problema en tanto la codificacion de programas tenderia a convertirse en un proceso mecanico.

Aunque existen lineas de investigacion que se desarrollan en ese sentido (iniciadas por M&W75, B&D77), este extremo no se ha alcanzado.

La tendencia, sin embargo, es evidente.

Hay una evolucion desde la etapa en que el esfuerzo central de la programacion residia en la codificacion de algoritmos y estructuras de datos "ad-hoc", manejando elementos de bajo nivel (la obra de Knuth (Knu73) puede ser considerada la culminacion de aquel estado del arte) hacia una nueva forma de trabajo donde los elementos manejados son de un nivel de abstraccion comparable al de los enunciados naturales y donde la generacion de algoritmos y estructuras de datos puede, si no mecanizarse, al menos convertirse en la adaptacion de soluciones ya bien estudiadas y clasificadas (ejemplo claro de lo cual es el texto de Aho et al (AHU83)).

Esta tendencia tiene tambien efecto sobre los lenguajes de programacion. Estos recorren un camino que va en direccion de los problemas, incorporando elementos de nivel de abstraccion creciente e intentando desprenderse del caracter procedural de los lenguajes originales.

La constatacion de esta evolucion y la necesidad de adaptarse a ella inspiraron la elaboracion de los nuevos cursos de Programacion de esta carrera. En particular, en el paragrafo que sigue se comentara la situacion del curso "Programacion III" en el cual se incluyeron los temas relativos a Tipos Abstractos de Datos que son motivo de este trabajo.

1.2.- Antecedentes del curso "Programacion III".-

Los estudiantes de este habian tomado previamente un curso de Programacion inicial donde se desarrollaba la perspectiva senalada en el paragrafo anterior, basicamente en dos lineas:

- Estudio de los elementos de los lenguajes de programacion algoritmicos de alto nivel (Pascal fue el mas referenciado, aunque ningun curso de la carrera se basa en la ensenanza de un lenguaje especifico).

- Elementos de estilo y tecnicas de diseno de algoritmos.

Dentro de esta segunda parte se hacia enfasis en el diseno de modelos de datos e inclusive se manejaban ideas de tipos abstractos de datos. Sin embargo, por tratarse de un curso de iniciacion, ningun lenguaje de especificacion era definido.

Las especificaciones se hacian en una variedad de ellos, siendo los mas utilizados algunas derivaciones de la logica de predicados y extensiones "ad-hoc" de Pascal.

El curso "Programacion III", considerado de nivel intermedio, debia ocuparse del estudio de estructuras de datos y algoritmos, con especial atencion a los aspectos de complejidad.

De acuerdo a la orientacion general ya senalada, se opto por seguir el texto de Aho et al (AHU83) que le es razonablemente fiel, complementado por la presentacion de un lenguaje formal de especificacion para tipos abstractos de datos, a saber: Especificaciones Algebraicas de Tipos Abstractos de Datos (en adelante, EATAD).

2.- Presentacion de EATAD.-

2.1.- Forma de elaboracion.-

El diseno y preparacion de los cursos del primer semestre lectivo de 1986 se realizaron conjuntamente por los docentes encargados y grupos de estudiantes que colaboraron honorariamente. Esto permitio la realizacion de seminarios donde los temas del curso eran parcialmente expuestos, con el objetivo de identificar los conceptos de mas dificil comprension.

La mayor dificultad en relacion a las EATAD residio en la ausencia de textos donde el tema se dearrollara didacticamente. Los materiales disponibles eran articulos donde las investigaciones originales eran publicadas.

Esto condujo a la decision de elaborar el material que se incluye en la seccion 4 de este trabajo. Las discusiones en los seminarios permitieron definir la estructura de esta presentacion, la cual comentamos brevemente en el siguiente paragrafo.

2.2.- Generalidades sobre la presentacion.-

La presentacion se estructura en tres partes:

i) Una introduccion a los objetivos que la tecnica persigue, esencialmente: la independencia de la implementacion, que permite mayor generalidad en la especificacion y mayor amplitud en la eleccion posterior de implementaciones. Aqui, se trata de justificar como las caracteristicas de las EATAD (basicamente, definicion de objetos por su comportamiento frente a determinadas operaciones) conduce a la independencia de representacion pretendida.

ii) Presentacion formal del lenguaje de EATAD.
Se da aqui el marco matematico (algebras heterogeneas y de tipo) (G&H78) y luego se concentra el desarrollo en la definicion por clausura del espacio del tipo, punto que merece especial atencion, y en el estudio de las operaciones selectoras, como portadoras de la semantica del tipo definido.

iii) Una guia metodologica para el diseno de EATAD.
Es una simplificacion del algoritmo de (G&H78) que garantiza la completitud-suficiente de la especificacion. No se incluye el desarrollo exhaustivo del mismo ni las pruebas relacionadas por exceder los prerrequisitos teoricos del curso.

3.- Conclusiones.-

Las conclusiones sobre el aprovechamiento del tema por parte de los estudiantes no pueden ser completas debido a que, a la fecha, la unica herramienta de evaluacion masiva disponible (examen final) no ha sido llevada a la practica.

Sin embargo, algunas impresiones parciales recogidas durante el curso pueden ser resenadas:

- Los estudiantes parecieron atraidos por el tema. Las ideas de diseno desarrolladas en 1.- son, en general, comprendidas y puestas en practica.

- Se detecto la necesidad de un mayor desarrollo de la teoria relativa a EATAD para cubrir problemas complejos donde aparezcan funciones con efectos secundarios. Estos parecen ser de dificil resolucion, sobre todo para personal habituado a lenguajes algoritmicos. Esta clase de funciones debe ser especificada ya sea por la definicion de TAD estructurados o por una descomposicion adecuada en funciones elementales. Cualquiera de las alternativas no es, generalmente, trivial.

- El resto del curso cubrio el estudio de implementaciones (estructuras de datos y algoritmos). Alli se detectaron problemas para derivar implementaciones a partir de las EATAD. En especial se noto la tendencia a considerar la EATAD como un algoritmo pasible de ser directamente implementado, lo cual a menudo conduce a soluciones totalmente ineficientes.

Bibliografia.-

AHU83 : Aho A., Hopcroft J., Ullman J.
Data Structures and Algorithms
(Addison Wesley, 1983).

B&D77 : Burstall R., Darlington J.
A Transformation System For Developing Recursive Programs
(JACM 24(1), 1977).

G&H78 : Guttag J., Horning J.
The Algebraic Specification Of Abstract Data Types
(Acta Informatica 10, 1978).

Knu73 : Knuth D.
The Art Of Computer Programming (vol. 1, 3)
(Addison Wesley, 1973).

M&W75 : Manna Z., Waldinger R.
Knowledge and Reasoning in Program Synthesis
(AIJ 6,2, 1975).

ANEXO

ESPECIFICACION ALGEBRAICA DE TIPOS ABSTRACTOS DE DATOS

Autores: Alvaro TASISTRO, Alfredo VIOLA.

0- RESUMEN Y OBJETIVOS

Este trabajo tiene como objetivo servir como texto de apoyo al curso teorico de Programacion III para el desarrollo del tema "Especificacion algebraica de Tipos Abstractos de Datos". Para la elaboracion del mismo se consultaron diversos trabajos originales, asi como aportes metodologicos extraidos de los seminarios previos al curso.

Reconocemos la colaboracion estudiantil en la edicion del trabajo, asi como en las discusiones metodologicas para la mejor presentacion del tema.

Se presenta la tecnica de especificacion algebraica de tipos abstractos de datos (en adelante TAD) a traves de un analisis en que progresivamente se profundizan los conceptos propios de esta tecnica (secciones 1 y 2). En particular se da en 1 una caracterizacion informal de los objetivos perseguidos con la especificacion algebraica de TAD y en 2 su presentacion formal. Se culmina en 3 con pautas generales que guian el disenio de este tipo de especificacion.

La presentacion hace uso de conceptos estudiados en matematicas, la mayoria de los cuales pretenden ser explicados en el propio desarrollo. Sin embargo, es obviamente necesaria la familiaridad del lector con conceptos elementales de teoria de conjuntos, y la definicion axiomatica de los naturales y recomendable el poseer nociones de la teoria de estructuras algebraicas.

Las definiciones y citas fueron extraidas del trabajo original de (G&H78),salvo en los casos en que la fuente se indique expresamente.

1- QUE ES UNA ESPECIFICACION ALGEBRAICA DE TAD? (I: CONCEPTO)

1.0- RESUMEN:

Aqui damos una aproximacion intuitiva al concepto de especificacion algebraica de TAD por la via de presentar los objetivos que esta persigue y el modo en que propone alcanzarlos (razonablemente justificados).

Sobre el final, presentamos un primer caso concreto de especificacion algebraica: el STACK (de venerable tradicion como ejemplo de TAD) y comentamos las ventajas que pueden obtenerse al aplicar este tipo de tecnica.

1.1- DEFINICIONES DE OBJETOS Y NIVELES DE ABSTRACCION.

Podemos manejar un sinonimo mas conocido para especificacion: definicion.

Esperamos que la mera mencion de esta palabra sea, a los efectos de comprender el concepto, suficiente. No nos introduciremos en la riesgosa tarea de definir "definicion", pero si profundizaremos en la idea a traves de un ejemplo:

Supongamos que sea necesario definir (especificar) un objeto sumamente conocido y comun, por ejemplo: un vehiculo automotor. Lo mas normal en estas circunstancias (o al menos lo que ha sucedido a quienes suscriben) es formarse una imagen visual particular (o eventualmente, mas de una, en sucesion) de este objeto que se desea caracterizar. De hecho, la presencia de esta imagen es el antecedente inmediato de la definicion, lo cual puede condicionar su generalidad.

Nuestra intencion es obtener las definiciones (especificaciones) lo mas abstractas (menos dependientes de una relizacion particular) y por lo tanto lo mas generales que sea posible. Si bien podemos definir el objeto vehiculo automotor por la via de recorrer imagenes (en general visuales) de realizaciones particulares, preferiremos otra aproximacion: evitar el riesgo de comprometernos con casos particulares (coches, motos, camiones, omnibus, etc) y extraer de todos estos los COMPORTAMIENTOS COMUNES que nos importen. En un sentido, elegiremos una abstraccion funcional, en cuanto NO MANEJAREMOS IMAGENES DE VEHICULOS SINO SUS COMPORTAMIENTOS FRENTE A DETERMINADAS OPERACIONES. Habra operaciones que permitan verificar propiedades del objeto (en nuestro caso: ?esta encendido?, ?esta en movimiento?, ?que trayectoria sigue?) y otras que alteren estas propiedades (encender, arrancar, frenar, apagar).

En nuestro ejemplo, trataríamos, por esta via, de definir vehiculo automotor sin hacer referencia a formas o imagenes, sino apenas a comportamientos.

Esto independiza de las realizaciones particulares estableciendo apenas aquellas operaciones que pueden ser aplicadas al objeto en cuestion.

1.2- EL STACK

Vayamos a un ejemplo mas familiar en el mundo de la computacion: el STACK. Trataremos de evitar las imagenes de representaciones particulares (array, listas con punteros, aun mas: la idea de secuencia) prefiriendo, por el contrario, la via de listar las operaciones que realizaremos sobre el tipo y los efectos que estas provocan.

En un sentido ,algo similar ocurriria si estuvieramos hablando en terminos de un lenguaje de programacion en que STACK estuviera predefinido: no nos dariamos cuenta de su forma,sino de las operaciones que podriamos aplicarle y de sus posibles resultados.

Una especificacion algebraica para el tipo STACK (de naturales) seria como la siguiente:

```
        | New:---->STACK
        | Push:---->STACK X N ---->STACK
    1 (
        | Pop:STACK---->STACK
        | Top:STACK---->N


        | Pop(New)=indefinido
        | Pop(Push(s,i))=s
    2 (
        | Top(New)=indefinido
        | Top(Push(s,i))=i
```

No queremos profundizar aqui en el estudio de esta especificacion algebraica. Basta observar que se definen en 1 los dominios y recorridos de 4 funciones: New, Push, Pop y Top.

Son las operaciones aplicables al tipo Stack. Los efectos de estas se vinculan entre si en 2.A la parte 1 la llamaremos especificacion sintactica y a la 2, los axiomas (o especificacion semantica) del tipo.

Esta declaracion establece el comportamiento esperado del tipo Stack. En un lenguaje dado (ejemplo:Pascal) mas de una realizacion concreta seria posible y puede elegirse en un proceso posterior.Al separar especificacion de representacion (implementacion) logramos:

A) dividir un gran problema en otros menores. Ejemplo: programar funciones de Stack se separa en definirlas formalmente y elegir implementaciones.

B) al ser formales la especificacion y la version final de la implementacion es posible

verificar que la segunda sea rigurosamente correcta.

C) El problema de elegir una implementacion puede atacarse en profundidad con el objetivo de administrar lo mas eficientemente posible los recursos disponibles .

El punto central esta en que por esta via se logre trasladar el esfuerzo de diseno a la etapa de generacion de la especificacion formal, desplazandola desde la etapa de codificacion.

El trabajo de programacion era, originalmente, un esfuerzo que requeria, a la vez, la comprension del problema estudiado e inventiva para la creacion de algoritmos y su codificacion para una maquina particular.

La idea es separar el enunciado preciso del problema (su especificacion) de su resolucion para un ambiente de programacion particular (implementacion). En la primera etapa podrian, idealmente, ignorarse las caracteristicas del computador o lenguaje de programacion final. En la segunda, no deberian existir problemas de comprension del problema originalmente enunciado.

El costo total del diseno disminuiria desde el punto en que el lenguaje de especificacion es de alto nivel de abstraccion lo cual abarata la tarea de modelacion y que, una vez ella este concluida, su traduccion al lenguaje de programacion final no requiere de esfuerzos adicionales de comprension y de inventiva.

La ventaja de esta aproximacion es especialmente notable cuando nos enfrentamos a problemas de gran porte. En estos casos, mezclar discusiones sobre implementacion (como hacer las cosas?) con discusiones sobre especificacion (que hay que hacer?), suele conducir a productos que, con enorme eficiencia en terminos de tiempo de maquina, resuelven problemas que no tienen nada que ver con el propuesto y son dificiles de corregir.

2.- QUE ES UNA ESPECIFICACION ALGEBRAICA DE TAD? (II: Forma)

2.0.- RESUMEN:

Aqui presentamos una base formal para desarrollar los antecedentes intuitivos de 1. En particular se aplica la rama del Algebra que estudia las estructuras algebraicas. En la literatura estas son llamadas "algebras", lo cual no implica que se este hablando de la estructura particular llamada algebra.

Con este marco, se estudian propiedades de las operaciones definidas para los tipos en cuestion y se dan orientaciones generales para el diseno de especificaciones.

2.1- TIPOS COMO ALGEBRAS:

El ya conocido concepto de TIPO DE DATOS (un dominio de valores y un conjunto de operaciones aplicadas sobre el dominio) es enunciable formalmente con herramientas matematicas analogas: las ESTRUCTURAS ALGEBRAICAS o ALGEBRAS.

Conviene prestar atencion a la terminologia,que usualmente esta cargada de ambiguedades:

Un ALGEBRA HOMOGENEA es una pareja (C,F), donde C (dominio) es un conjunto no vacio (de valores) y F un conjunto finito de operaciones tales que cada operacion Fj es una funcion $Fj: C^n \longrightarrow C$

Observar que todas las operaciones se aplican a n-uplas tomadas de un unico conjunto (el C). Esta definicion no incluye estructuras como la del STACK, en la cual existen operaciones con dominios y recorridos en mas de un conjunto. De aqui surge la definicion (mas adecuada a nuestras intenciones):

Un ALGEBRA HETEROGENEA es una pareja (V,F), donde V es un conjunto de conjuntos no vacios Vi y F un conjunto de funciones Fj, $1 \leq j \leq m$ tales que $Fj: Vi1 \times Vi2 \times \ldots\ldots\ldots \times Vin \longrightarrow Vk$, donde Vih pertenece a V, h= 1,2,.....,n y Vk pertenece a V

Analicemos la nueva definicion. Recordando que queremos definir operaciones que involucren mas de un conjunto (ejemplo: stacks y naturales) la forma de estas operaciones debe cambiar, dejando de ser:

$$Fj: C \xrightarrow{n} C$$

para permitir componer en el dominio a mas de un conjunto.

Asi, al contrario de un algebra homogenea en que las operaciones involucran a un unico conjunto C, en las heterogeneas se definen sobre un conjunto de conjuntos (V). Cualquiera de estos puede tomarse tambien como recorrido de las operaciones definidas .

Con esto, el planteo gana generalidad y es posible representar funciones como PUSH : STACK x NATURAL ----> STACK. Sin embargo, rapidamente observamos que la generalidad que hemos ganado puede resultar excesiva.

Recuerdese que estamos desarrollando una herramienta para describir formalmente el comportamiento de una determinada clase de objetos (los objetos del tipo abstracto en cuestion). El poder de las algebras heterogeneas permite incluir operaciones que describen comportamientos de mas de una clase de objetos, relacionandolos mutuamente.

En nuestro ejemplo, el conjunto V tiene como elementos al conjunto de los Stacks y al de los Naturales. Aplicando la definicion de algebra heterogenea, podria existir en F una operacion como:

Sum : N X N --> N (suma de naturales).

Es obvio que esto es redundante si nuestra intencion es definir el TAD STACK. La suma de naturales no ayuda a describir propiedades de los STACKs, sino de los propios naturales.

Las algebras heterogeneas son un formalismo apropiado para definir a la vez a los STACKs y a los naturales. Si, por el contrario, se pretende hacer corresponder una de estas estructuras a la definicion de un unico tipo, se tienen dos consecuencias:

- Aquellos tipos distintos del que se pretende definir y que intervengan en esa especificacion deben ser considerados predefinidos, o definidos por separado.

  Ejemplo: Se pretende definir STACK de naturales.
  El tipo Naturales se considera definido a los efectos de la especificacion de STACK.

- Conviene restringir la definicion de algebra heterogenea para adaptarla a esta situacion, definiendo un nuevo tipo de estructura: el algebra de tipo.

| Un ALGEBRA DE TIPO es un par (V,F) donde V es un conjunto de con- |
|juntos Vi, 1<=i<=m, no vacios, uno de los cuales es distinguido y se|
|denomina TOI (tipo de interes o type of interest en ingles), y F es |
|un conjunto finito de operaciones Fj tales que: |
| Fj:Vi1xVi2x....xVin----->Vk, |
|donde Vih pertenece a V, con1<=h<=n, Vk pertenece a V |
|y al menos uno de los conjuntos Vi1,Vi2,..Vin,Vk es el TOI |

Hay que apreciar el nuevo concepto introducido: estamos modificando apenas una condicion de las operaciones del algebra exigiendo que el TOI (el conjunto de los objetos cuyo comportamiento queremos definir) participe (como parte del dominio o como recorrido) en todas las operaciones.
La ventaja de esta definicion frente a la definicion concurrente de mas de un tipo, implicada por las algebras heterogeneas, es la de la nitidez ganada. Efectivamente, construir objetos mas complejos a partir de otro cuya estructura es, o bien conocida, o bien se define separadamente, es una practica tipica en la programacion y que coherentemente, aplicamos aqui. Frente a ella, la tecnica de mezclar propiedades y comportamiento de indole diversa resulta superada por engorrosa.

2.2- SOBRE LOS DOMINIOS DE LOS TIPOS.

2.2.1- COMO DEFINIRLOS?

Hemos dado un formalismo para especificar TAD: los algebras de tipo.
Nuestro objetivo debe ser ahora dar metodologias o al menos orientaciones para construir este tipo de especificaciones. Con esta perspectiva estudiaremos ejemplos con el fin de inducir propiedades generales interesantes. Retomemos el caso STACK de NATURALES (dado en 1):
Deberiamos poder reconocer en esta especificacion, los elementos del algebra de tipo correspondiente. En particular concentremonos en el conjunto V, cuyos elementos son los dominios de los tipos involucrados en las operaciones definidas.
Podemos establecer inmediatamente que V = (STACK, N), es decir el conjunto de dominios incluye el conjunto de los STACKS de NATURALES y el de los NATURALES, donde el TOI es obviamente STACK.
No obstante, haber adjudicado un nombre a cada conjunto interviniente no alcanza para definirlos. Si bien podemos parecer satisfechos con la simple mencion del nombre NATURALES o de su simbolo mas comun "N" , esto no constituye una definicion.

En este caso particular esos nombres evocan un conjunto ya conocido y sobre cuya definicion no vale la penainsistir.

No nos preocupamos entonces demasiado por los naturales, sabiendo que existe una forma de definirlos rigurosamente, que, sin embargo, no es sustancial en este momento. El caso no es igual para los *STACKS*.

El nombre *STACK* no evoca ninguna definicion anterior, ya que en este momento se supone estamos especificando el TAD *STACK* por primera vez.

No hay siquiera un conjunto que puede ser tomado como base, ni simbologia apropiada para los *STACKS*. Por tanto tenemos que dar una DEFINICION RIGUROSA de este conjunto.

Ahora bien, en este punto en que comprendemos la necesidad de definir expresamente el dominio del tipo, deseamos recordar nuestras intenciones en relacion a la forma de la especificacion. Por lo expuesto en 1, queremos conseguir que nuestra especificacion sea independiente de la representacion de los objetos definidos. En particular tratamos de definir estos objetos no a traves de caracterizar su forma sino su comportamiento frente a determinadas operaciones. Esta idea, aplicada al caso que estamos estudiando, conduce a una tecnica de definir conjuntos que, en abstracto, podria caracterizarse asi:

"El conjunto X esta formado por todos aquellos valores que sean resultados del conjunto de operaciones Cx".

La idea sera definir ciertas operaciones y establecer que los resultados obtenidos por todas las aplicaciones posibles de estas operaciones son los elementos del conjunto a definir .

Esta idea esta en la base de la definicion axiomatica de los naturales. En ella se dice :

1) El cero es natural
2) Hay una funcion, (succ) que aplicada a un natural devuelve un natural.
3) Los naturales son solo los objetos descritos en 1) y 2).

Hay que puntualizar que esto es apenas una parte de la definicion de los naturales, la cual completaremos luego. Sin embargo, ya obtenemos de aqui conclusiones valiosas .

Podemos definir un conjunto C a partir de un conjunto G y un conjunto de operaciones F como sigue:

1) x pertenece a G -----> x pertenece a C
2) Fj pertenece a F y x1, x2, .., xn pertenece a C ---->
   -----> Fj (x1,....xn) pertenece a C
3) Esos son los unicos miembros de C

C se llama la "clausura de F sobre G" (Cl(G,F)), G se llama "el generador de C". En el caso de los naturales, podemos tomar como generador al conjunto que contiene al cero y F contiene una sola operacion: SUCC.

Entonces, siguiendo el esquema anterior, definimos los naturales: N = Cl((0), (SUCC) de donde :

por 1) 0 pertenece a G -----> 0 pertenece a N
por 2) 0 pertenece a N ----->succ(0) pertenece a N , etc

Con este esquema de definicion por clausura nos acercamos a la idea de caracterizar todo el dominio a traves de operaciones. En el caso de naturales, decimos que obtenemos los elementos del dominio por sucesiva aplicaciones de la operacion SUCC, salvo uno de ellos, que suponemos existente y que es el cero.

Sin embargo, un recurso mas puede ser usado para que tambien el cero sea resultado de una operacion, la cual debera comportarse como constante y, por lo tanto, no necesitara argumentos. De este modo definimos la operacion CERO.

Hecho esto, no necesitamos que G contenga ningun valor. De acuerdo al esquema de definicion por clausura, los naturales se obtendran componiendo de todas las formas posibles a las dos funciones: CERO Y SUCC.

En otras palabras: N= Cl (( ), (CERO, SUCC)) y el conjunto generador pasa a ser vacio pues el antiguo elemento primitivo (cero) puede ahora ser obtenido aplicando una operacion (CERO), no necesitandose suponer su existencia.

Como se traduce esto en una especificacion algebraica?. En el caso de los naturales escribiriamos:

CERO: -----> N
SUCC: N ----> N

Estas dos clausulas contienen todas las ideas manejadas hasta aqui. Estan estableciendo:

Los elementos de N se obtienen por la clausura de CERO y SUCC sobre el conjunto vacio. En otras palabras: CERO produce un natural (vease que es funcion sin argumentos, o sea, constante).

SUCC(CERO) es un natural, pues es la aplicacion de SUCC a un natural. Ahora: SUCC(SUCC(CERO)) lo sera tambien, etc.

Es claro que esto no completa la definicion de los naturales. Otros axiomas estan todavia omitidos. Mas adelante se vera como enunciarlos.

En el caso de STACKS de NATURALES, existen tres operaciones que cumplen el papel de CERO y SUCC:

New: ----> STACK
Push: STACK x N ----> STACK
Pop: STACK -----> STACK

Notar que en esta clausura intervienen dos conjuntos (STACKS y N) de lo cual proviene la necesidad de la siguiente definicion :

Dada una familia de generadores Gi y una familia F de funciones, se define una familia de conjuntos Ci como:

1) x pertenece a Gi --) x pertenece a Ci

2) Fj : Vi1 X Vi2 X ... X Vin --) Vk pertenece a F y
(x1, x2, ... ,xn) pertenece a Ci1 X Ci2 X ... X Cin
entonces Fj (x1, x2, ... ,xn) pertenece a Ck

3) Estos son los unicos miembros de los Ci

En este caso decimos que el conjunto V = Cl ((Gi, i=1,n) , F)

En el caso del TAD STACK (de naturales), V tiene como miembros al conjunto de los STACKs y a N. La definicion anterior podria aplicarse de dos formas:

i) Considerando G = ((), ()) y F contendria todas las funciones necesarias para generar los stacks (New, Push, Pop) y a los naturales (CERO, SUCC). En este caso estariamos definiendo en una misma algebra ambos tipos.

ii) Siguiendo el criterio discutido anteriormente, puede considerarse predefinido al conjunto N, y definir el algebra correspondiente a los STACKs con:

G = ((), N), F = (New, Push, Pop)

Aplicaremos el segundo criterio, por las razones ya expuestas, el cual podemos enunciar con generalidad de la siguiente forma:

En un algebra de tipo, el TOI se define como Cl (((),I), S), siendo I=V-TOI o sea aquellos dominios del algebra distintos del TOI y siendo S el conjunto de operaciones que en la especificacion sintactica aparecen con recorrido en el TOI es decir, de la forma:

Vi1 x Vi2 x ... x Vin ----> TOI.

Los demas conjuntos del algebra se consideran definidos.

A su vez, el conjunto de operaciones S puede particionarse en dos conjuntos:

S1: el conjunto de las operaciones de la forma:

f : TOI x Vi1 x Vi2 x ... x Vik -----> TOI. (Ej: SUCC, push, pop),que como se ve, requieren como argumento al menos un elemento del TOI.

Para que alguna de estas operaciones pueda ser aplicada, deberan existir funciones que generen elementos del TOI sin usar argumentos que pertenezcan a el. Estas forman:

S2: el conjunto de las operaciones de la forma:

f : Vi1 x Vi2 x ... x Viq ---> TOI, con Vih distinto al TOI,h= 1,..,q

Las operaciones del conjunto S2, en general, se aplican a n-uplas donde ningun componente pertenece al TOI. Un recurso de notacion se emplea para definir constantes del TOI (como el Cero). Estas se denominan funciones nularias, o sea sin argumentos y pertenecen, por lo tanto, al conjunto S2. Son ejemplos las funciones CERO y New.

2.2.2 EL AXIOMA DE INDUCCION

Nos interesa ahora introducir un axioma fundamental, asociado a la definicion de clausura, que es el axioma de induccion(L&Z77).

La definicion de naturales, que parcialmente introdujimos mas atras incluye el siguiente axioma:

Si dada una propiedad P se cumplen 1 y 2 tales que:

1) P vale para el natural cero
2) P vale para n implica que vale para SUCC(n),

entonces P vale para todos los naturales.

Esta idea puede generalizarse:

Si dada una propiedad P se cumplen 1 y 2 tales que:

1) P vale para todos los elementos del TOI generados por operaciones del conjunto *S2* (definido mas arriba)
2) P vale para t1, t2,...., tp, implica que vale para el elemento F(t1,t2, ...,tp,v1, ..., vq) para toda operacion F del conjunto *S1* (definido mas arriba), donde
   t1,...,tp pertenecen al TOI y
   v1,...,vq no pertenecen al TOI

entonces P vale para todos los elementos del TOI.

Este axioma esta implicito en la definicion por clausura de los TOI conduce a un metodo para probar propiedades de estos: la demostracion por induccion. Esto, que se hacia en cursos liceales con los naturales, aparece aqui generalizado.

Asi, para los *STACKS*, si queremos probar alguna propiedad P cualquiera basta:

1) Probar P(New( )) (Unica operacion del tipo *S2*)
2) *Suponer* P valida para un stack generico *S* y demostrar que entonces P vale para Push(s, h), (siendo h un natural cualquiera) y Pop (s).

2.2.3 PROPIEDADES DE LAS OPERACIONES *S*

Por ahora solo nos hemos referido, como puede verse, a una parte de la especificacion sintactica. No hemos hablado ni de operaciones cuyo recorrido sea distinto del TOI ni de la especificacion semantica del tipo.

De las primeras se hablara en la seccion siguiente. En este punto nos concentraremos en las ecuaciones de la especificacion semantica que corresponden a las operaciones del conjunto *S* (es decir, aquellas cuyos miembros son composiciones de funciones que dan como resultado elementos del TOI).

Veamos en particular el axioma:

$$Pop\ (Push(s, n)) = s$$

A partir del axioma se ve que si a un stack *S* se aplica Push con un natural cualquiera y, a continuacion, Pop, volvemos a obtener el mismo stack.

En algun sentido Push y Pop se definen como inversos entre si.

Este axioma induce una propiedad importante: de todas las composiciones posibles de Push, Pop y New hay algunas de ellas que dan el mismo resultado, por ejemplo: Pop(Push(New, n)) = New.

Estos axiomas definen relaciones de equivalencia entre los elementos del TOI. Este conjunto queda, entonces, particionado en clases de equivalencia segun aquella relacion.

Dos expresiones z, w pertenecen a la misma clase si existe una secuencia de igualdades (deducibles de los axiomas) tales que:
z = z1 = z2 = ... = zn = w.

Podemos asociar a cada clase un unico elemento (canonico).
En el ejemplo, se puede demostrar que todos los elementos canonicos pueden generarse por composiciones adecuadas de las funciones New y Push.
Como caso particular: Pop (Push (New, n)) y New pertenecen a una misma clase cuyo elemento canonico es el elemento New.

Este resultado es fundamental pues establece que incluido en *S* hay un subconjunto de operaciones suficiente para generar el TOI (llamados operadores constructores que designaremos con la letra C) y el complemento no genera valores que no puedan ser generados por los anteriores (se llaman extensiones, designados con la letra E).

En el STACK, New y Push seran constructores y Pop una extension. No vamos a incluir aqui la demostracion de que Pop es una extension.
Basta puntualizar que:

Dentro del conjunto *S*, la semantica puede establecer cierta "superposicion" en el efecto de las operaciones y que de acuerdo a esto puede definirse los constructores como el minimo conjunto de operaciones *S* que pueden generar todo el conjunto TOI.

2.3. LAS OPERACIONES CON RECORRIDO FUERA DEL TOI

Hasta ahora nos hemos concentrado en estudiar las operaciones cuyo recorrido es el conjunto TOI que estamos definiendo. Ahora, vamos a centrar nuestro estudio en las operaciones cuyo recorrido no es el conjunto TOI. Nombraremos O al conjunto de operaciones con recorrido fuera del TOI. Se vera como estas operaciones resultan imprescindibles para manipular con utilidad los objetos que se estan definiendo.

Retomemos la especificacion del TAD NATURALES:

```
        | CERO : -----> N
       (|
        | SUCC : N -----> N
```

Se pueden realizar algunas observaciones:

1) Son todas operaciones del conjunto S(segun fue visto).

2) No existe especificacion semantica.

3) La clausura de (CERO, SUCC) sobre el conjunto vacio define el TOI :

$$(CERO, SUCC (CERO), SUCC (SUCC (CERO)),...., SUCC^{n} (CERO),.....)$$

(donde $SUCC^{0}(CERO) = CERO$ y $SUCC^{n}(CERO) = SUCC^{n-1}(CERO)$).

A partir de las observaciones anteriores se pude concluir que:

i) No es posible probar que este conjunto sea el de los naturales definido por los axiomas de Peano.
En particular,por ejemplo,nada asegùra que CERO sea distinto de SUCC(CERO).
Si esto no ocurriera, todos los elementos del TOI serian iguales. La propiedad CERO <> SUCC (CERO) es asegurada por uno de los axiomas de Peano, el cual no fue incluido en la especificacion.

ii) Una operacion que permita determinar la igualdad/desigualdad de dos naturales deberia tener la siguiente sintaxis:
EQUAL : N X N -----> B (Siendo B un conjunto con dos elementos diferentes T y F que representan los dos resultados posibles de la funcion).
Esta funcion tiene recorrido en un conjunto diferente al TOI.
Es, por lo tanto, una operacion tipo O.

Se ve, con este pequenio ejemplo que al no existir operaciones tipo O no es posible asignarle propiedades que caracterice a los elementos del TOI. Vamos a modificar levemente nuestro tipo abstracto.

Vamos a definir:

```
CERO : -----> N
SUCC : N x N -----> N
EQUAL : N x N -----> BOOLEAN
```

1 EQUAL (CERO, CERO) = T
2 EQUAL (CERO, SUCC (n)) = F
3 EQUAL (SUCC (n), CERO) = F
4 EQUAL (SUCC (n1), SUCC (n2)) = EQUAL (n1, n2)

Los axiomas de Peano omitidos en la especificacion anterior aparecen aqui en los axiomas 2 al 4. El axioma de induccion, como ya se ha dicho, esta implicito en la especificacion.

Es importante destacar que ahora con la operacion EQUAL es posible distinguir los elementos del TOI. Se ve que es una operacion que OBSERVA (de alli O, el nombre del conjunto de estas operaciones) el comportamiento de los elementos del TOI cuando les son aplicadas operaciones del conjunto S.

Hasta ahora, dado un elemento del TOI, le aplicabamos una operacion de S y se obtenia un elemento del TOI.

Pero, al realizar esta operacon ?se obtiene un elemento distinto del original?

Solamente podemos verificar que dos elementos son distintos cuando tienen comportamientos diferentes frente a una misma operacion.

Por ejemplo, sean los naturales CERO y SUCC (CERO) (resultado de aplicar la operacion SUCC al natural cero dado). Para verificar que son distintos, usando el axioma 2, se ve que si n = CERO, EQUAL(CERO, SUCC (CERO))= F y por el axioma 1 EQUAL (CERO, CERO) = T.

En otras palabras, si al elemento CERO se le aplica la funcion SUCC, se obtiene un natural que tiene un comportamiento distinto que el CERO respecto de la funcion EQUAL. Entonces se ve que con las operaciones de O, se definen propiedades sobre los elementos del TOI. De aqui surge la conclusion siguiente:

LA SEMANTICA DEL TIPO ABSTRACTO ESTA DADA POR LA ESPECIFICACION SEMANTICA DE LAS OPERACIONES DE O.

Esta idea complementa el hecho que estamos definiendo el conjunto segun sea el comportamiento respecto a determinadas operaciones.

Con este ejemplo se recuerdan dos ideas fundamentales:

1) Se define el tipo abstracto indicando el comportamiento de sus elementos respecto a determinadas operaciones.
2) La semantica del TAD (el significado) esta dada por la especificacion semantica de las operaciones tipo O.

Un ultimo detalle a remarcar. Vimos que para que el tipo abstracto tuviera interes, es necesario que haya operaciones en O, o sea es necesario definir operaciones con recorrido en un conjunto diferente al TOI. Este conjunto debe estar definido, para lo cual creamos en general otro tipo abstracto de datos.

Por lo anterior se ve que, dado un TAD T cualquiera siempre existiran, en su especificacion, operaciones con recorrido en otro TAD T'.

Si tambien hubiera que definir el TAD T' se tendria nuevamente la misma situacion. Para evitar que la definicion de un TAD se transforme en una sucesion infinita de especificaciones, debe existir un TAD primitivo. Dicho TAD es BOOLEAN.

O sea, consideramos que EXISTEN DOS VALORES DIFERENTES (T y F).

3 ALGUNAS IDEAS SOBRE COMO CONSTRUIR ESPECIFICACIONES ALGEBRAICAS

3.0 OBJETIVO

Se definen propiedades que deben cumplir las especificaciones algebraicas. Luego se presenta informalmente un procedimiento para construir especificaciones algebraicas con dichas propiedades.

Es muy importante destacar el alcance de esta descripcion: NO pretende ser un tratado riguroso del tema, pues supera sobremanera los alcances del curso. Solo se pretende insinuar que hay una teoria matematica que respalda algunos resultados presentados que se aplicaran para llegar al objetivo de este estudio: tener una metodologia para disenar especificaciones algebraicas interesantes.

3.1 CONSISTENCIA Y COMPLETITUD

Vamos a definir el tipo "bolsa de enteros" con

NUEVA-BOLSA : ---> Bolsa
AGREGAR : Bolsa x entero ---> Bolsa
QUITAR : Bolsa x entero ---> Bolsa
PERTENECE : Bolsa x entero ---> BOOLEAN

1) PERTENECE (NUEVA-BOLSA,i) = F
2) PERTENECE (AGREGAR(b,i),j) = si i = j entonces T
sino PERTENECE (b,j)
3) QUITAR (NUEVA_BOLSA) = (NUEVA_BOLSA)
4) QUITAR (AGREGAR (b,i),j) = si i = j entonces b
sino AGREGAR (QUITAR (b,j),i)
5) PERTENECE (QUITAR (b,i),j) = si i = j entonces F
sino PERTENECE (b,j)

Consideramos el valor:
QUITAR(AGREGAR(AGREGAR(BOLSA_NUEVA,9),9),9) que pertenece al TOI (aplicando la clausura, ya vista anteriormente).
Este valor tiene la siguiente propiedad:

a) Usando el axioma 4) vemos que es igual a AGREGAR(BOLSA_NUEVA,9) y luego, usando el axioma 2) llego a:
PERTENECE(QUITAR(AGREGAR(AGREGAR(BOLSA_NUEVA,9),9),9),9) = T
b) Usando el axioma 5) llego a:
PERTENECE(QUITAR(AGREGAR(AGREGAR(BOLSA_NUEVA,9),9),9),9) = F

Entonces la funcion PERTENECE, aplicada al mismo elemento da como resultado dos valores diferentes lo cual contradice el hecho de que es una funcion. Llegamos a una contradiccion en los axiomas o, de otra manera, los axiomas son inconsistentes.
De aqui que una propiedad deseable es que la especificacion sea CONSISTENTE. Se puede ver que si eliminamos el axioma 5), la especificacion deja de ser inconsistente. De ahora en adelante trabajaremos solo con los axiomas 1) al 4).

Ahora, la especificacion algebraica debe indicar en los axiomas el comportamiento de las funciones para TODOS los elementos del conjunto. Por ejemplo, si NO usamos el axioma 1) (nuestra especificacion solo contiene los axiomas 2) al 4)) vamos a llegar a que NO esta definida la operacion PERTENECE para el elemento NUEVA_BOLSA, o sea que la especificion no es suficientemente completa.
Una especificacion es SUFICIENTEMENTE COMPLETA si todos los resultados posibles se pueden deducir de los axiomas.
(Esta es una nocion intuitiva, de un concepto que puded definirse con rigor).

Lo importante es recalcar que son conceptos diferentes. Por ejemplo:

1) Si tenemos los axiomas 1) al 4) la especificacion es CONSISTENTE Y SUFICIENTEMENTE COMPLETA.
2) Si consideramos los axiomas 1) al 5) la especificacion NO ES CONSISTENTE pero SI SUFICIENTEMENTE COMPLETA.
3) Si consideramos los axiomas 2) al 4) la especificacion es CONSISTENTE pero NO SUFICIENTEMENTE COMPLETA.
4) Si consideramos los axiomas 2) al 5) la especificacion NO CUMPLE NINGUNA DE LAS DOS CONDICIONES.

3.2 METODO HEURISTICO PARA CONSTRUIR ESPECIFICACIONES

Un problema interesante de resolver es el siguiente: dadas de un TAD cualquiera, las especificaciones sintacticas y semantica, dicha especificacion es consistente?, ?dicha especificacion es suficientemente completa? Podriamos interesarnos en condiciones NECESARIAS y SUFICIENTES (algun conjunto de condiciones tales que: se verifican dichas condiciones (===) la especificacion es consistente (o suficientemente completa)).

Desafortunadamente se puede probar que NO existe tal condicion.
Entonces se veran condiciones mas debiles, en particular, condiciones suficientes para la completitud suficiente de la especificacion.
Se vera un metodo que POR CONSTRUCCION garantiza que la especificacion creada sea suficientemente completa. De otra manera: si construimos la especificacion usando el metodo nos aseguramos que la especificacion cumple esta propiedad (condicion suficiente), pero pueden existir especificaciones suficientemente completas NO construidas a partir de este metodo. (Por eso, no es necesaria).
El metodo presentado es una simplificacion del algoritmo en (G&H78) (donde puede estudiarse con rigor).

Vamos a ver el metodo con un ejemplo, indicando desde ya que va a haber expresiones informales (no rigurosas), dado que un estudio completo escaparia de lejos el alcance del curso. Vamos a intentar construir una especificacion algebraica SUFICIENTEMENTE COMPLETA para el TAD bolsa de enteros:

NUEVA_BOLSA : ---> bolsa
AGREGAR : bolsa x entero ---> bolsa
QUITAR : bolsa x entero----> bolsa
PERTENECE : bolsa x entero ---> boolean

Nuestro objetivo es crear un conjunto de axiomas SUFICIENTEMENTE COMPLETO. Entonces:

1) PARTICIONAR LAS OPERACIONES EN LOS CONJUNTOS C,E y O

Mientras las operaciones de O son faciles de reconocer,no lo es tanto dividir el conjunto S en los subconjuntos C y E. Por ejemplo: podriamos considerar como constructores (C) a (NUEVA-BOLSA,AGREGAR, QUITAR), y no tenemos ninguna operacion tipo E. Pero tambien podriamos definir C = (NUEVA-BOLSA, AGREGAR) E = (QUITAR) o C=(NUEVA-BOLSA, QUITAR), E = (AGREGAR).
En el caso general no es sencillo saber cuales operaciones son Constructores y cuales son Extensiones. En la practica, nosotros tenemos alguna nocion de las propiedades del conjunto (incluso los nombres de las operaciones son mnemotecnicas: AGREGAR, QUITAR nos indican alguna idea), con la cual podemos decir cuales son constructores y cuales no. Este es el caso de los ejemplos que veremos nosotros.
Vamos a considerar entonces que las operaciones se dividen:

```
┌──────────────────────────────┐
| C = (NUEVA-BOLSA, AGREGAR)   |
| E = (QUITAR)                 |
| O = (PERTENECE)              |
└──────────────────────────────┘
```

Es importante destacar que las operaciones nularias pertenecen al conjunto C.

2) CONSTRUIR EL CONJUNTO CTERMS = (c(x1, ...,xn) donde c es una funcion que pertenece a C y para todo i, xi es una variable independiente del tipo apropiado segun la sintaxis de C).

En nuestro caso CTERMS = (NUEVA-BOLSA, AGREGAR (b,i)) donde b es un variable tipo bolsa e i una variable entera.

3) CONSTRUIR EL CONJUNTO OTERMS = (o(x1,...,xn) donde o es una funcion que pertenece a O , si xi no pertenece al TOI es una variable independiente y si xi pertenece al TOI es un elemento de CTERMS).

En nuestro caso tenemos:
OTERM = (PERTENECE? (NUEVA-BOLSA, j),PERTENECE?(AGREGAR(b, i), j))

Vamos a ver mas detenidamente estos conjuntos. CTERMS seria el conjunto de todas las funciones de C aplicadas a cualquier variable.
Intuitivamente se ve que si a las variables (en este caso i, b) le asignamos todos los valores posibles obtenemos todos los elementos del TOI (de alli que a las funciones de C se les llame constructores).
OTERMS es el conjunto de operaciones de O aplicadas a todos los elementos de CTERMS, o sea a TODOS los elementos del conjunto TOI.
En nuestro caso si damos a j cualquier valor entero, como con _BOL-NUEVA_BOLSA y AGREGAR (b, i) se obtienen todos los elementos del TOI, obtenemos todos los casos posibles a los cuales se puede aplicar la funcion PERTENECE. Se garantiza asi la completitud de la misma.

4) CONSTRUIR EL CONJUNTO ETERMS= (e(x1, ..., xn) e es una funcion que pertenece a E , si xi no pertenece al TOI es una variable independiente y si xi pertenece al TOI es un elemento de CTERM).

En nuestro caso :
ETERMS = (QUITAR (NUEVA_BOLSA), QUITAR(AGREGAR(b, i), j)).
Es la misma idea del conjunto OTERMS pero para las funciones de E.

5) CONSTRUIR LOS AXIOMAS, tomando como parte izquierda todos los elementos del conjunto OTERMS, y como partes derechas los resultados de aplicar las funciones correspondientes.

En nuestro caso, vamos a tener dos axiomas :

1) PERTENECE (NUEVA_BOLSA, j) = ...
2) PERTENECE (AGREGAR(b, i), j) = ...

donde cada parte izquierda en un elemento de OTERMS. Aqui le damos significado a las operaciones tipo O(realizamos la semantica de dichas operaciones).

La idea es que las partes derechas sean mas simples que las izquierdas. En nuestro caso :

PERTENECE (NUEVA_BOLSA, j) = F
PERTENECE (AGREGAR (b, i), j) = IF i=j THEN T
ELSE PERTENECE(b,j)

Notar que las partes derechas son en cierto sentido mas sencillas que las izquierdas, en el hecho de que son constantes o la funcion PERTENECE (b, j) no aparece compuesta con la funcion AGREGAR.

Estas ideas pueden estudiarse con rigor en (G&H78).

6) COMPLETAR LA AXIOMATIZACION usando las equivalencias de las funciones de la clase E en la clase C; todas las partes izquierdas son los elementos de ETERMS

En nuestro caso tendremos : QUITAR (NUEVA_BOLSA,j) = ...
QUITAR (AGREGAR (b,i),j) = ...

Entonces completamos las equivalencias

c) QUITAR(NUEVA_BOLSA,j) = NUEVA_BOLSA
d) QUITAR (AGREGAR (b,i),j) = si

i = j entonces b

sino AGREGAR (QUITAR(b,j),i),

La verificacion de la consistencia tiene reglas que derivan de la logica.Un ejemplo de ellos es la regla de Church-Posser pero NO profundizaremos mayormente en el tema. En los casos en que trabajamos nosotros la consistencia es practicamente trivial,dado que si se entendio el problema y las operaciones a realizar,la consistencia va a estar casi siempre garantida.

4. APENDICE.-

Es importante destacar que para realizar las especificaciones solo precisamos 5 primitivas (o sea herramientas que suponemos que existen sin definicion y son utilizadas), a saber:

a)la composicion de funciones (QUITAR(AGREGAR(b,i),j))

b)la relacion de igualdad (=)

c)TRUE |
)---)constantes que asumimos que son DISTINTAS
d)FALSE| (como vimos al final de 2)

e)cantidad ilimitable de variables (b,i,j)

Se puede ver que SI_ENTONCES_SINO puede especificarse:

SI_ENTONCES_SINO : BOOLEAN X T X T ----) T

1) SI_ENTONCES_SINO (TRUE, t1, t2) = t1
2) SI_ENTONCES_SINO (FALSE, t1,t2) = t2

SI_ENTONCES_SINO (b, t1, t2) aparece escrito en notacion infija:
SI b ENTONCES t1 SINO t2.

5. BIBLIOGRAFIA.-

G&H78. Guttag J., Horning J.
The Algebraic Specification of Abstract Data Types.
(Acta Informatica 10, 1978)

L&Z77. Liskov B., Zilles S.
An Introduction to Formal Specifications of Data Abstractions.
(en Current Trends in Programming Methodology, vol 1,
Prentice Hall, 1977)