



32
20/

Universidad Nacional Autónoma de México

FACULTAD DE CIENCIAS

Fundamentos de las Especificaciones Algebraicas

T E S I S
QUE PARA OBTENER EL TITULO DE
M A T E M A T I C O
P R E S E N T A
ALEJANDRO ISIDORO SALDIVAR UGALDE



MEXICO. D. F.

1991

FALLA DE ORIGEN



Universidad Nacional
Autónoma de México



UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

DERECHOS RESERVADOS © PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis está protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

INDICE.

INTRODUCCION

1.- MOTIVACION PARA EL DESARROLLO DE LAS ESPECIFICACIONES FORMALES.

- 1.1 Crisis del Software
- 1.2 El papel de las especificaciones en el proceso de programación de computadoras
 - 1.2.1 Facetas en el desarrollo de programas
 - 1.2.2 Definición de especificación
 - 1.2.3 Principios de una especificación
- 1.3 Especificaciones algebraicas

2.- CONCEPTOS BASICOS

- 2.1 Sistemas algebraicos
 - 2.1.1 Ejemplos de sistemas algebraicos en matemáticas
 - 2.1.2 Definición de sistema algebraico
 - 2.1.3 Algebras heterogéneas
- 2.2 Signaturas
- 2.3 Algebras de una signatura dada
- 2.4 Variables y términos en una signatura
 - 2.4.1 Definición de variable en una signatura
 - 2.4.2 Definición de términos
 - 2.4.3 Evaluación de términos
 - 2.4.3.1 Definición de evaluación de términos
 - 2.4.3.1 Definición de asignación extendida
- 2.5 Inducción estructural
- 2.6 Especificación y álgebra de una especificación
- 2.7 Sintaxis de las especificaciones algebraicas
- 2.8 Especificación algebraica de un sistema computacional

3.- TIPOS DE DATOS ABSTRACTOS

- 3.1 Tipos de datos
- 3.2 Estructuras de datos
- 3.3 Tipos de datos abstractos
 - 3.3.1 Definición de tipos de datos abstractos
 - 3.3.2 Ejemplo de un tipo de dato abstracto TDA(Pilas)
 - 3.3.3 Especificación del tipo de datos Pilas
 - 3.3.4 Algebra para la especificación de Pilas
- 3.4 Congruencia de términos elementales
- 3.5 Algebra de términos cocientes (T_{ESPEC})
- 3.6 Homomorfismos e isomorfismos
- 3.7 Propiedades del álgebra de términos cocientes T_{ESPEC}

4.- SEMANTICA DE LAS ESPECIFICACIONES

- 4.1 Semántica de una especificación
- 4.2 Propiedades de los homomorfismos
 - 4.2.1 Diagramas conmutativos
 - 4.2.2 Principales propiedades de los homomorfismos
- 4.3 Algebra de términos
- 4.4 Concepto de álgebra inicial y álgebra libre
- 4.5 Inicialidad del álgebra de términos cocientes
- 4.6 Definición de generadora y típica
- 4.7 Comentarios sobre inicialidad

5.- ESPECIFICACIONES PARAMETRIZADAS

- 5.1 Especificaciones parametrizadas
 - 5.1.1 Importancia de las especificaciones parametrizadas
 - 5.1.2 Especificaciones parametrizadas
 - 5.1.3 Ejemplos de especificaciones parametrizadas
- 5.2 Categorías y funtores
 - 5.2.1 Definición de categoría
 - 5.2.2 Definición de functor

- 5.2.3 Propiedades de los funtores
- 5.2.4 Ejemplos de funtores
- 5.3 Construcciones libres y funtores libres
 - 5.3.1 Ejemplos de objetos libres
 - 5.3.2 Definición de construcción libre
 - 5.3.3 Definición de transformación e isomorfismo natural
 - 5.3.4 Definición y propiedades de los funtores libres
- 5.4 Definición de A -álgebra de términos cocientes
- 5.5 Semántica de las especificaciones parametrizadas

6.- INTRODUCCION AL LENGUAJE ACT-ONE

- 6.1 Generalidades
- 6.2 Descripción del lenguaje
- 6.3 Especificaciones básicas
- 6.4 Ejemplos de especificaciones en ACT-ONE

CONCLUSIONES

APENDICE

BIBLIOGRAFIA

INTRODUCCION

En la breve historia de las computadoras de electrónica digital, los años 50 y los 60 fueron décadas de *hardware*. Los años 70 fueron un período de transición y un tiempo de reconocimiento de *software*. La década de los 80 y lo que llevamos de los 90 se puede decir que fue y nos encontramos en la de *software*. De hecho, los avances de la informática pueden llegar a estar limitados por la incapacidad de producir *software* de calidad que pueda hacerse con la enorme capacidad de los procesadores de estos últimos años.

Durante las tres primeras décadas de la computación, el principal desafío era desarrollar el *hardware* de las computadoras de forma que se redujera el costo de procesamiento y almacenamiento de datos. A lo largo de la década de los 80, los avances en microelectrónica han dado como resultado una mayor potencia de cálculo a la vez que una reducción del costo.

Hoy el principal problema es diferente. El principal desafío es reducir el costo y mejorar la calidad de las soluciones basadas en computadoras.

La potencia de las grandes computadoras de ayer está hoy disponible en un simple circuito integrado. Las imponentes capacidades de procesamiento y almacenamiento del *hardware* moderno representan un gran potencial de cálculo. El *software* es el mecanismo que nos facilita utilizar y explotar este potencial.

Desde hace algunos años por primera vez en la historia de la informática, el *software* ha sobrepasado al *hardware* como elemento clave del éxito de muchos productos y sistemas.

En esencia, el *software* es normalmente el factor que marca la diferencia. La suficiencia y oportunidad de la información dada por el *software*, diferencia a una compañía de sus

lógica, del álgebra y de otras disciplinas de las matemáticas.

La algebraica es una de las principales especificaciones formales más populares.

La finalidad de esta tesis es dar los fundamentos matemáticos en los cuales se basan las especificaciones algebraicas.

Este trabajo está compuesto por las siguientes secciones o capítulos:

-INTRODUCCION.

-MOTIVACION PARA EL DESARROLLO DE LAS ESPECIFICACIONES FORMALES.

En este punto se explicará por que son necesarias las especificaciones formales, en particular las algebraicas, en el desarrollo de programas.

-CONCEPTOS BASICOS.

Aquí se definen los conceptos fundamentales para el desarrollo de las especificaciones algebraicas.

-TIPOS DE DATOS ABSTRACTOS.

En esta parte se define el concepto de TIPO DE DATOS ABSTRACTOS y su importancia en las especificaciones formales.

-SEMANTICA DE LAS ESPECIFICACIONES.

En esta parte se define la semántica de una especificación algebraica como el álgebra inicial de la clase de todas las álgebras que cumplen con la especificación. Y se hará un estudio formal del porque de la definición.

-ESPECIFICACIONES PARAMETRIZADAS.

En esta sección se extenderá el concepto de especificaciones al de especificaciones parametrizables. y se introducirán conceptos básicos de la TEORIA DE CATEGORIAS y su vinculación con las especificaciones parametrizables.

-LENGUAJE DE ESPECIFICACION ACT-ONE.

En esta parte se dará una introducción al lenguaje de especificación ACT-ONE. Así como ejemplos de su uso.

-APENDICE A.

-CONCLUSIONES.

-BIBLIOGRAFIA.

CAPITULO I.

MOTIVACION PARA EL DESARROLLO DE LAS ESPECIFICACIONES FORMALES

Desde hace algunos años se ha visto que las computadoras son una herramienta esencial para el desarrollo de las distintas actividades de los hombres, desde las fábricas automatizadas de ensamblaje de automóviles hasta nuestra propia casa.

Actualmente estamos viendo como estas computadoras están invadiendo muy rápidamente todos los campos de trabajo y diversión del hombre, las podemos encontrar en las fábricas, controlando algún proceso de extrema exactitud o llevando la contabilidad de la misma; en la educación; en los videojuegos y en una infinidad de actividades.

Debido a este creciente uso de la computación han surgido una gran cantidad de hombres dedicados al desarrollo de los programas de aplicación. Estos programas se van volviendo cada vez más complejos a fin de que cualquier usuario pueda sacar provecho de ellos y pueda confiar en su calidad para ayudar a resolver sus problemas.

Sin embargo, se ha visto que en muchos de los casos, estos programas no son del todo confiables como podríamos esperar y sus causas son diversas.

Por un lado usuario y experto en computación tienen "formas de hablar" diferentes y a veces suceden conflictos de comunicación, malos entendidos o ambigüedades en el planteamiento de lo que se requiere que haga el programa. Además, los sistemas complejos de programación no siempre son desarrollados por una sola persona, lo que ocasiona también problemas de comunicación entre las personas del equipo.

Otro factor más, es que al tener un programa ya funcionando "bien", no hay forma de probar que cumpla con lo que exactamente se requería de él, y es difícil verificar que dé resultados correctos, para todos los casos posibles.

prueba sistemática y técnicamente completa del software.

Debido a estas razones, muchos científicos del mundo, se han dedicado a encontrar métodos de programación que ayuden a probar que las implementaciones sean las correctas de acuerdo a los modelos de la solución del problema.

1.2 EL PAPEL DE LAS ESPECIFICACIONES EN EL PROCESO DE PROGRAMACION DE COMPUTADORAS.

1.2.1 FACES EN EL DESARROLLO DE PROGRAMAS.

Ahora bien en el proceso de desarrollo de programas existen cuatro fases principales, las cuales son:

- ANALISIS DE REQUERIMIENTOS.
- DISEÑO DEL PROGRAMA.
- IMPLEMENTACION.
- MANTENIMIENTO.

En general el usuario al plantear su problema al analista lo hace de manera vaga, ambigua, incompleta, sin claridad, etc., esto debido a que la forma en comunicarse es en lenguaje natural, y el lenguaje natural es ambiguo.

Así, el primer problema del analista es desarrollar un modelo formal del problema con base en cual discutirá con el usuario hasta que satisfaga sus necesidades. Esto implica la necesidad de formalidad matemática en el lenguaje con que se presentará la especificación de los requerimientos, que deberá ser suficientemente formal pero también entendible por el usuario a fin de aceptarse o modificarse.

De esta forma dada la especificación formal del problema, las siguientes fases del desarrollo estarán libres de las incertidumbres asociadas a la especificación del problema.

1.2.2 DEFINICION DE ESPECIFICACION.

¿ Qué es la especificación ?

La especificación de requerimientos es el primer documento formal necesario en el desarrollo de programas. Describe las funciones que se requiere que el programa o software efectúe para el usuario. Su principal preocupación es contestar el Qué hará un programa y no Cómo y a partir de éste documento, el diseñador prepara la solución al problema.

Podemos decir que la especificación se requiere para:

- Saber exactamente qué va a hacer un programa.
- Entender como se unirán los programas de un sistema complejo.
- Permitir ver que tan correcto es un programa, ya sea al probarlo, o por medio de métodos de verificación formal.

No hay duda de que la forma de especificar tiene mucho que ver con la calidad de la solución. Los programadores que se han esforzado en trabajar con especificaciones incompletas, inconsistentes o mal establecidas han experimentado la frustración y confusión que invariablemente se produce. Las consecuencias se padecen en la calidad, oportunidad y completitud del programa resultante.

La especificación, independientemente del modo en que se realice, puede ser vista como un proceso de *representación*.

Ahora bien, ¿ Cuáles podrían ser los principios para una buena especificación ?

1.2.3 PRINCIPIOS DE UNA ESPECIFICACION.

Baltzer y Goldman [79] proponen varios principios para una

buena especificación, en seguida enunciaremos algunos:

-SEPARAR FUNCIONALIDAD DE IMPLEMENTACION.

Por definición, una especificación es una descripción de lo que se desea, en vez de cómo se realiza. Las especificaciones pueden adoptar formas muy diferentes. Una de estas formas es la de funciones matemáticas; dado algún conjunto de entrada, producir un conjunto particular de salida. en tales especificaciones, el resultado a ser obtenido ha sido expresado enteramente en una forma sobre el *Qué* (en vez de *Cómo*). En parte, esto debido a que el resultado es una función matemática de entrada (la operación tiene puntos de comienzo y parada bien definidos) y no está afectado por el entorno que le rodea.

UNA ESPECIFICACION DEBE ABARCAR EL SISTEMA DEL CUAL EL SOFTWARE ES UN COMPONENTE.

Un sistema está compuesto de componentes que interactúan. Sólo dentro del contexto del sistema completo y de la interacción entre sus partes puede ser definido el comportamiento de una componente específica. En general, un sistema puede ser modelado como una colección de objetos pasivos y activos. Estos objetos están interrelacionados y dichas relaciones entre los objetos cambian con el tiempo.

Estas relaciones dinámicas suministran los estímulos a los cuales responden los objetos activos, llamados agentes. Las respuestas pueden causar posteriormente cambios y, por tanto, estímulos adicionales a los cuales los agentes deben responder.

UNA ESPECIFICACION DEBE SER UN MODELO COGNITIVO.

La especificación de un sistema debe ser un modelo

de análisis empleadas para ayudar a los especificadores y para probar especificaciones, deben ser capaces de tratar con la incompletitud.

Y debido a la incompletitud se necesita que nuestros métodos de especificación tengan contemplado mecanismos que puedan aumentar nuevos módulos a la especificación.

1.3 ESPECIFICACIONES ALGEBRAICAS.

Las especificaciones algebraicas fueron desarrolladas en la década de los 70 y fueron foco de estudio para varios investigadores. Conceptualmente este tipo de especificaciones están basadas en ideas del álgebra clásica y universal.

Las especificaciones algebraicas definen a través de ecuaciones e igualdades algebraicas, relaciones entre funciones, las cuales establecen propiedades y características de un tipo de dato abstracto.

Este tipo de especificaciones se desarrollaron con las bases que se tenían de los conceptos de Tipos de Datos Abstractos.


Los cuales fueron desarrollados a partir de un artículo de Liskov, B. y Zilles, S. escrito en el año de 1974. El artículo consistía en explicar la programación de Tipos de Datos Abstractos y se basa en la idea principal de que un Tipo de Dato Abstracto puede ser definido a partir de la caracterización de las operaciones para ese tipo.

Y a partir de ese momento muchos investigadores se dieron a la tarea de desarrollar esta novedosa técnica para el desarrollo de sistemas computacionales. Actualmente ya han sido muy estudiados varios tipos de datos abstractos por medio de esta técnica de especificación. Entre estas estructuras podemos encontrar a las pilas, colas, listas y tablas Hash.


Ahora bien, una de las principales preocupaciones, de las especificaciones algebraicas, es hacer especificaciones que sean independientes de una representación en particular.

Las especificaciones algebraicas, para mostrar esta independencia de representación, hace uso de conceptos del álgebra abstracta como son los homomorfismos y los isomorfismos. A la vez estos isomorfismos y homomorfismos son una herramienta que se utiliza para dar una formulación concisa a conceptos, propiedades y demostraciones.

En el presente trabajo, se mostrarán los fundamentos matemáticos en los cuales se basa este tipo de especificación.



CONCEPTOS BASICOS



elementos y una operación binaria llamado producto \cdot y que además satisface el siguiente axioma:

$$\cdot) X \cdot (Y \cdot Z) = (X \cdot Y) \cdot Z$$

este axioma indica que la operación \cdot es asociativa.

Un sistema más es el *monoid*, el cual consiste de (M, \cdot, e) , donde M es un conjunto, \cdot es una operación binaria entre los elementos de M , llamado producto y un elemento distinguido e de M llamado el elemento identidad, además este sistema cumple con algunas reglas que enunciamos a continuación

$$\cdot) X \cdot (Y \cdot Z) = (X \cdot Y) \cdot Z \text{ con } X, Y, Z \in M \text{ (ley asociativa)}$$

$$\cdot) e \cdot X = X \cdot e = X$$

ejemplos de tal sistema son $(\mathbb{N}, +, 0)$, los naturales con la operación binaria $+$, la suma, y con el elemento distinguido 0 , cero; y $(\mathbb{Z}, +, 0)$, los enteros con la suma y el cero.

Un último ejemplo es el de *grupo* el cual consiste de $(G, \cdot, {}^{-1}, e)$ o sea un conjunto G , una operación binaria llamada producto, una operación unaria ${}^{-1}$ llamada el inverso multiplicativo y un elemento distinguido e llamado neutro multiplicativo. Este sistema además cumple con las siguientes leyes:

$$\cdot) X \cdot (Y \cdot Z) = (X \cdot Y) \cdot Z \quad (\text{ ley asociativa })$$

$$\cdot) e \cdot X = X \cdot e = X$$

$$\cdot) X \cdot X^{-1} = X^{-1} \cdot X = e$$

como sabemos existen muchos ejemplos de este tipo de sistema algebraico.

distintos tipos, así pues demos otra definición para estas nuevas formas de sistemas algebraicos.

2.1.3 ALGEBRAS HETEROGENEAS.

Definición. Un Algebra Heterogénea es un par (Φ, Ω) en donde Φ consiste de una familia de tipos diferentes A_i ($i \in I$) junto con una colección Ω de operaciones definidas en estos tipos de la siguiente forma, asociada con cada operación $\omega \in \Omega$ de grado n se puede escribir como

$$\omega : A_{i_1} \times A_{i_2} \times \dots \times A_{i_n} \longrightarrow A_i$$

donde $i_1, i_2 \in I$ y $i_j \in \{1, \dots, n\}$, e I una familia de índices.

Un ejemplo de este tipo de sistema algebraico es el concepto de autómeta.

Definición. Un Autómeta es un álgebra heterogénea que consiste de una quintupla (S, A, δ, q_0, Z) en donde S , A y Z son los tipos del sistema y significan lo siguiente:

S es el conjunto de estados.

A es el alfabeto de entrada.

$Z \subset S$ es el conjunto de estados finales.

y los símbolos de operación

$q_0 : \{*\} \longrightarrow S$ que llamaremos estado inicial.

$\delta : S \times A \longrightarrow S$ es la función de siguiente estado.
 $\delta(s, x)$ es el siguiente estado que resulta cuando la máquina está en el estado s y recibe de entrada el símbolo x . Se dice que δ es la operación heterogénea.

Observemos que a $q_0(*)$ lo identificamos con la constante q_0 , donde $\{*\}$ es un conjunto, cualquiera, con un sólo elemento; de

Otro ejemplo de estos sistemas son los tipos de datos abstractos que estudiaremos en el capítulo siguiente.

2.2 SIGNATURAS

Después de haber visto algunos ejemplos de estructuras algebraicas y de establecer la definición de lo que se entiende por un Sistema Algebraico desde el punto de vista del álgebra clásica, pasaremos ahora a definir este mismo concepto pero que será más acorde para nuestros fines.

Para definir signatura es necesario ver antes algunos conceptos.

Sea S el conjunto $\{s_1, s_2, \dots, s_n\}$ con $n \in \mathbb{I}$, en donde \mathbb{I} es una familia de índices, entonces se construyen dos conjuntos:

S^+ que será el conjunto de todas las posibles cadenas, no vacías, de los símbolos que representan a los elementos del conjunto S .

Y

S^* que será el conjunto de todas las posibles cadenas, incluyendo la cadena vacía, de los símbolos que representan a los elementos del conjunto S .

Teniendo lo anterior en cuenta, pasamos a definir lo que es una signatura.

Definición Una SIGNATURA $\Sigma = (S, F)$ consiste de un conjunto $S = \{s_1, s_2, \dots, s_n\}$ de tipos, donde cada s_i es un tipo. Con \mathbb{I} una familia de índices.

Y de una familia de conjuntos:

F el conjunto de los símbolos de operación entre los elementos de S .

ii) TRID

$$\text{TRID}_{\text{ent}} = \langle -1, 0, 1 \rangle$$

$$\text{cero}_{\text{TRID}} : \quad \longrightarrow \quad \text{TRID}_{\text{ent}} \quad \text{y} \quad \text{cero}_{\text{TRID}} = 0$$

$$\text{suc}_{\text{TRID}} : \text{TRID}_{\text{ent}} \quad \longrightarrow \quad \text{TRID}_{\text{ent}} \quad \text{y}$$

$$\text{suc}_{\text{TRID}} = \langle \langle -1, 0 \rangle, \langle 0, 1 \rangle, \langle 1, -1 \rangle \rangle$$

$$\text{pred}_{\text{TRID}} : \text{TRID}_{\text{ent}} \quad \longrightarrow \quad \text{TRID}_{\text{ent}} \quad \text{y}$$

$$\text{pred}_{\text{TRID}} = \langle \langle -1, 1 \rangle, \langle 0, -1 \rangle, \langle 1, 0 \rangle \rangle$$

Es claro que para nuestra experiencia el primer ejemplo i) es el representante exacto de la signatura Σ , donde S representa a los enteros. Sin embargo, el ejemplo ii) también tiene la misma estructura que la signatura.

Podemos decir que una signatura representa la abstracción de la estructura de un sistema algebraico.

Los ejemplos i) y ii), de nuestra signatura Σ , no son más que una interpretación de la estructura, en ellos hablamos de tipos y de funciones bien definidas y no de símbolos, a los ejemplos que cumplan con una cierta signatura se les llamará álgebras.

2.3 ALGEBRAS DE UNA SIGNATURA DADA.

Definición. Dada una signatura S-tipificada $\Sigma = (S, F)$, diremos que un ALGEBRA A es una Σ -álgebra (sigma álgebra) si sucede lo siguiente

$$A = (S_A, F_A) \quad \text{donde}$$

- i) S_A es una familia S-indicada de conjuntos. $S_A = \{A_s\} \text{ se } S$ a cada conjunto A_s se le denominará el portador del tipo s.

A las variables de una signatura las interpretaremos, al igual que las constantes, como funciones 0-arias y cuyo tipo es el correspondiente al de la variable.

2.4.2 DEFINICION DE TERMINOS.

Los términos en nuestros sistemas algebraicos son los nombres y pronombres; son las expresiones que se pueden interpretar que nombran un objeto. Los términos se pueden definir como aquellas expresiones que se pueden construir a partir de los símbolos de constante y de las variables mediante los símbolos de función. la definición formal de término se da a continuación.

Supongamos que $\Sigma = (S, F)$ es una signatura.

Definición. Los conjuntos $T_{F,S}(X)$ de términos del tipo $s \in S$ se definen recursivamente como:

$$1. X_s \cup C_s \subseteq T_{F,S}(X) \quad \text{con } X_s = \text{Conjunto de variables del tipo } s \\ \text{y } C_s = \text{Conjunto de constantes del tipo } s$$

$$2. F(t_1, \dots, t_n) \in T_{F,S}(X) \\ \text{donde } F \in F_{w,s}, \quad w = s_1 s_2 \dots s_n \\ \text{y con } t_i \in T_{F,S}(X) \quad \forall i \in \{1, \dots, n\}$$

3. No existen más términos que los definidos por 1 y 2.

Al conjunto definido por el punto 1 lo llamaremos conjunto de *Términos básicos*. Y los términos definidos por el punto 2 serán los *Términos compuestos*.

A los términos sin variables los representaremos por $T_{F,S}$ y los llamaremos *Términos elementales* del tipo S .

Así es que si $\Sigma = (S, F)$ es una signatura s -tipificada entonces el conjunto de términos de Σ es la unión de los términos con variable de todos los tipos con los términos sin variable de todos los tipos.

$$T_F(X) = \bigcup_{s \in S} T_{F,S}(X) \cup \bigcup_{s \in S} T_{F,S}$$

Ejemplos:

Sea la signatura Σ definida anteriormente, $X_{\text{ent}} = \{x\}$ y $s = \text{ent}$ entonces los términos de Σ son :

$$T_{F,S} = \{ \text{cero}, \text{suc}(\text{cero}), \dots, \text{suc}^n(\text{cero}), \text{pred}(\text{cero}), \dots, \text{pred}^n(\text{cero}), \\ \text{pred}(\text{suc}(\text{cero})), \text{pred}(\text{suc}^2(\text{cero})), \dots, \text{pred}(\text{suc}^n(\text{cero})), \\ \text{suc}(\text{pred}(\text{cero})), \text{suc}(\text{pred}(\text{suc}(\text{cero}))), \dots \}$$

$$T_{F,S}(X) = T_{F,S} \cup \{ \text{suc}(x), \dots, \text{suc}^n(x), \text{pred}(x), \dots, \text{pred}^n(x), \dots \}$$

donde si $f \in F_{F,S}$ $f^n(X)$ es la aplicación de f a X , n -veces.

$$Y = \bigcup_F T_F(X) = \bigcup_{F,S} T_{F,S} \cup \bigcup_{F,S} T_{F,S}(X)$$

2.4.3 EVALUACION DE TERMINOS.

Ahora nuestro siguiente paso es definir la evaluación de términos con y sin variables en un álgebra A.

2.4.3.1 DEFINICION DE EVALUACION DE TERMINOS.

Definición. Sea T_F el conjunto de términos, sin variables de una signatura $\Sigma = (S, F)$ y A una Σ -álgebra. La EVALUACION $eval: T_F \rightarrow A$ se define recursivamente como sigue:

$$\begin{aligned} eval(K) &= K_A \quad \forall K \in \text{Constantes de } \Sigma \\ eval(F(t_1, \dots, t_n)) &= F_A(eval(t_1), \dots, eval(t_n)) \\ &\quad \forall F(t_1, \dots, t_n) \in T_F \end{aligned}$$

Observemos que la evaluación, para el caso de las constantes, no es más que la interpretación que tendremos de ellas en el álgebra.

2.4.3.2 DEFINICION DE ASIGNACION EXTENDIDA.

Dado un conjunto de variables X, de la signatura $\Sigma = (S, F)$, y una asignación $asig: X \rightarrow A$, donde a cada elemento de X le damos un cierto valor de A, o sea, $asig(x) = x_s$ con $x \in X_s$, $s \in S$ y $x_s \in A_s$. Definimos la asignación extendida como sigue:

$$\begin{aligned} asig: T_F(X) &\rightarrow A \quad \text{tal que} \\ asig(x) &= asig(x) \quad \forall x \in X \\ asig(K) &= K_A \quad \forall K \in \text{Constantes} \\ asig(F(t_1, \dots, t_n)) &= F_A(asig(t_1), \dots, asig(t_n)) \\ &\quad \forall F(t_1, \dots, t_n) \in T_F(X) \end{aligned}$$

Ejemplos:

De nuestro ejemplo tenemos lo siguiente:

$$\begin{aligned} i.1) \text{eval}_{\mathbb{Z}}(\text{suc}(\text{suc}(\text{cero}))) &= \text{suc}_{\mathbb{Z}}(\text{eval}(\text{suc}(\text{cero}))) \\ &= \text{suc}_{\mathbb{Z}}(\text{suc}_{\mathbb{Z}}(\text{eval}(\text{cero}))) \\ &= \text{suc}_{\mathbb{Z}}(\text{suc}_{\mathbb{Z}}(0)) \\ &= \text{suc}_{\mathbb{Z}}(1) \\ &= 2 \end{aligned}$$

$$\begin{aligned} i.2) \text{eval}_{\text{TRID}}(\text{pred}(\text{pred}(\text{cero}))) &= \text{pred}_{\text{TRID}}(\text{eval}(\text{pred}(\text{cero}))) \\ &= \text{pred}_{\text{TRID}}(\text{pred}_{\text{TRID}}(\text{eval}(\text{cero}))) \\ &= \text{pred}_{\text{TRID}}(\text{pred}_{\text{TRID}}(0)) \\ &= \text{pred}_{\text{TRID}}(-1) \\ &= 1 \end{aligned}$$

Sea $X = \{x\}$ y la asignación $X \longrightarrow \mathbb{Z}$ dada por $\text{asig}(x) = 10$

$$\begin{aligned} \text{entonces } \text{asig}(\text{suc}(\text{suc}(x))) &= \text{suc}_{\mathbb{Z}}(\text{asig}(\text{suc}(x))) \\ &= \text{suc}_{\mathbb{Z}}(\text{suc}_{\mathbb{Z}}(\text{asig}(x))) \\ &= \text{suc}_{\mathbb{Z}}(\text{suc}_{\mathbb{Z}}(10)) \\ &= \text{suc}_{\mathbb{Z}}(11) \\ &= 12 \end{aligned}$$

En el capítulo correspondiente a semántica inicial de las especificaciones se presentarán las principales propiedades de las funciones de asignación, evaluación y asignación extendida, así como la importancia que estas tienen sobre el concepto de independencia de la representación de datos.

2.9 INDUCCION ESTRUCTURAL.

Para demostrar que el conjunto de los términos de una signatura Σ , cumplen con una cierta propiedad, utilizaremos un método inductivo sobre el largo del término, donde el largo de un término se define con respecto al número de veces que se aplica un símbolo de función a una constante o a una variable.

Escribiremos $I=D$ donde cada uno de los términos I y D tienen variables en X y estas variables están cuantificadas universalmente.

ω) Una ecuación $e=(X,I,D)$ se dice que es **VALIDA** en una Σ -álgebra A si para toda asignación $\alpha: X \longrightarrow A$ se tiene que

$$\underline{\alpha \text{ sig}}(I) = \underline{\alpha \text{ sig}}(D).$$

Si e es válida en A diremos que A **SATISFACE** e .

$\omega\omega$) Una **ESPECIFICACION** $\text{ESPEC} = (S,F,E)$ consiste de una **signatura** $\Sigma = (S,F)$ y un conjunto E de ecuaciones en la **signatura** Σ .

$\omega\omega\omega$) Una **ALGEBRA** A de una especificación ESPEC , **ESPEC-álgebra**, es un álgebra A de la **signatura** Σ que **satisface** todas las ecuaciones E .

De nuestra definición anterior se puede ver que una especificación estará constituida por:

1. Algunos conjuntos de objetos, que representan los tipos.
2. Un conjunto de descripciones sintácticas de las funciones u operaciones que podemos realizar con los objetos anteriores.
3. Una descripción semántica, o sea, un conjunto suficientemente completo, de las relaciones que especifican cómo las funciones interactúan con cada una de las demás funciones del sistema.

Observemos que la definición de especificación, es idéntica al de sistema algebraico.

2.7 SINTAXIS DE LAS ESPECIFICACIONES ALGEBRAICAS.

Con base en el párrafo anterior veamos una forma de escribir una especificación algebraica para un sistema.

nom-sist. =

tipos:

< tipo₁ >
< tipo₂ >
:
< tipo_n >

oper:

$$\begin{array}{ccccccc} \text{op}_1: <\text{tipo}_{i_1}> \times <\text{tipo}_{i_1+1}> \times \dots \times <\text{tipo}_{i_n}> & \longrightarrow & <\text{tipo}_{i_h}> \\ & & & & & & \vdots \\ & & & & & & \vdots \\ \text{op}_m: <\text{tipo}_{m_1}> \times <\text{tipo}_{m_1+1}> \times \dots \times <\text{tipo}_{m_k}> & \longrightarrow & <\text{tipo}_{i_l}> \end{array}$$

var:

var_{i1} ... var_{im} ∈ <tipo₁>
:
:
:
var_{n1} ... var_{nm} ∈ <tipo_n>

ecuac:

e₁ = e'₁
e₂ = e'₂
.
.
.
e_p = e'_p

Fin nom-sist.

De donde

nom sist:

Es el nombre del sistema que estamos definiendo.

tipos:

Es el conjunto de los tipos utilizados en nuestro sistema.

oper:

Las operaciones definidas en nuestro sistema junto con su dominio y su contradominio.

var:

Son las variables, con su respectivo tipo que serán utilizadas en las ecuaciones

ecuac:

Nuestros axiomas o ecuaciones que cumple el sistema y

Fin nom-sist.

Fin del sistema.

Observemos que la signatura de la especificación es la parte correspondiente a la declaración de tipos y oper. La parte de var significa las variables, cuantificadas universalmente, con su respectivo tipo, que aparecen en las ecuaciones. Si existen varias variables del mismo tipo se pueden escribir separadas por comas. Adicionalmente en nuestras especificaciones, especialmente en los sistemas computacionales, utilizaremos una función condicional la cual consiste de SI a ENTONCES b SINO c, equivale a la siguiente función:

$$f(x) = \begin{cases} b & \text{si } a \text{ es cierta} \\ c & \text{si } a \text{ es falsa} \end{cases}$$

En seguida se dan algunos ejemplos de especificaciones :

naturales =

tipos: nat

oper: 0; \longrightarrow nat.

 suc; nat \longrightarrow nat

$$+; S \times S \longrightarrow S$$

$$-; S \times S \longrightarrow S$$

var:

$$a, a_1, a_2, a_3 \in S$$

ecuac:

$$(a_1 + a_2) + a_3 = a_1 + (a_2 + a_3)$$

$$a_1 + a_2 = a_2 + a_1$$

$$a + e = a$$

$$a + (-a) = e$$

$$(a_1 + a_2) * a_3 = (a_1 * a_3) + (a_2 * a_3)$$

$$a_1 * (a_2 + a_3) = (a_1 * a_2) + (a_1 * a_3)$$

Fin anillo.

Del álgebra moderna sabemos que cuando definimos una nueva estructura tratamos de dar una axiomatización lo más completa posible, pero el trabajo de un matemático no se acaba aquí sino que ahora tiene que encontrar más propiedades o teoremas, esto con el propósito de enriquecer a la estructura. Para la demostración de los teoremas, el algebraísta utiliza un cálculo ecuacional o una reescritura de términos. La cual consiste en ir sustituyendo términos por otros términos que sean iguales. Este método es el que utilizaremos para demostrar o encontrar propiedades de las especificaciones algebraicas.

2.0 ESPECIFICACION ALGEBRAICA DE UN SISTEMA COMPUTACIONAL.

Para finalizar este capítulo veremos un ejemplo de una especificación algebraica de un sistema computacional.

El ejemplo consiste en dar la especificación de un manejador de bases de datos.

Definición. Desde el punto de vista del usuario de computadoras, UN SISTEMA MANEJADOR DE BASES DE DATOS provee fácil acceso interactivo a un banco de información. Además, ésta información es mantenida por medio del sistema.

Vamos a suponer que nuestro banco de datos está compuesto por registros que contienen datos personales de gentes. Las operaciones que podemos realizar con nuestro banco de información son: *agregar, borrar, consultar, etc.*

La especificación informal de nuestro sistema es la siguiente:

crea:

crea una nueva base de datos

agrega(bd,nombre,datos):

retorna la base de datos <bd> agregándole el nuevo registro compuesto por el nombre de la persona <nombre> y sus datos <datos>.

borra(bd,nombre):

borra de la base de datos <bd> el registro de la persona cuyo nombre es <nombre>.

vacía(bd):

esta operación regresa cierto si la base de datos <bd> se encuentra vacía de lo contrario regresa falso.

pertenece(bd,nombre):

regresa cierto si el registro de la persona <nombre> se encuentra en la base de datos <bd>, de lo contrario falso.

consulta(bd,nombre):

esta operación retorna el registro de la persona cuyo nombre es <nombre>.

A continuación se da la especificación algebraica de nuestro sistema.

SisManBD =

tipos:

BaseDatos

Nombres

ReqPer

Boolean

oper:

CIERTO : \longrightarrow Boolean

FALSO : \longrightarrow Boolean

crea : \longrightarrow BaseDatos

inserta: BaseDatos Nombres ReqPer \longrightarrow BaseDatos

agrega : BaseDatos Nombres ReqPer \longrightarrow BaseDatos

U (error)

pertenece: BaseDatos Nombres \longrightarrow Boolean

consulta: BaseDatos Nombres \longrightarrow ReqPer

U (error)

borra : BaseDatos Nombres \longrightarrow BaseDatos

U (error)

vacía : BaseDatos \longrightarrow Boolean

variables:

bd \in BaseDatos

nom1 \in Nombres

nom2 \in Nombres

dat \in ReqPer

ecuac:

- 1 vacía(crea) = CIERTO
- 2 vacía(inserta(bd,nom1,dat)) = FALSO
- 3 pertenece(crea,nom1) = FALSO
- 4 pertenece(inserta(bd,nom2,dat),nom1) =
SI nom1 = nom2
ENTONCES CIERTO
SINO pertenece(bd,nom1)

```

5      borra(crea,nom1) = ERROR
6      borra(inserta(bd,nom2,dat),nom1) =
          SI nom1 = nom2
          ENTONCES bd
          SINO inserta(borra(bd,nom1),nom2,dat)
7      consulta(crea,nom1) = ERROR
8      consulta(inserta(bd,nom2,dat),nom1) =
          SI nom1 = nom2
          ENTONCES dat
          SINO consulta(bd,nom1)

9      agrega(bd,nom1,dat) =
          SI pertenece(bd,nom1)
          ENTONCES ERROR
          SINO inserta(bd,nom1,dat)

```

Fin SisManBD.

En la especificación formal del manejador se definió una función, inserta, la cual no se encuentra en la especificación informal. Inserta es idéntica a la función de agrega sólo que no verifica si se encuentra el registro en la base de datos.

Observemos que existe una constante llamada ERROR, esta constante nos indica que la operación que estamos efectuando es incorrecta.

Por convención, cualquier función que tenga como argumento un valor ERROR regresará como resultado ERROR. Tomando en cuenta que ERROR representa a una clase de errores de distintos tipos.

El uso de los axiomas o ecuaciones en nuestras especificaciones algebraicas determinará el resultado de aplicar una operación a un término en particular. En base a esto, nosotros podemos inferir el comportamiento de nuestras operaciones al aplicarlas. Para inferir nuevos resultados de nuestras ecuaciones, utilizaremos un cálculo ecuacional, esto lo veremos en seguida utilizando nuestro ejemplo del sistema manejador de base de datos.

crea, Representa la base de datos vacía y además estamos creando un objeto llamado base de datos.

Podemos insertar en la base de datos a Juan Lopez y a sus

datos personales *dat* de la siguiente forma
`agrega(crea, Juan Lopez, dat)`

`= inserta(crea, Juan Lopez, dat)` -por la ecuación 9

e.i. la base de datos con un registro.

Podemos agregar ahora a Teresa M. y a sus datos *datTer*

`agrega(inserta(crea, Juan Lopez, dat), Teresa M., datTer)`

`=inserta(inserta(crea, Juan Lopez, dat), Teresa M., datTer)`

- ésto utilizando la ecuación 9.

Podemos preguntar por los datos de Julia Zeta.

`consulta(inserta(inserta(crea, Juan Lopez, dat),`

`Teresa M., datTer), Julia Zeta).`

`=consulta(inserta(crea, Juan Lopez, dat), Julia Zeta)` por 8.

`=consulta(crea, Julia Zeta)` por la ecuación 8.

`=ERROR` por la ecuación 7.

Y podríamos seguir viendo el comportamiento de las distintas operaciones de nuestra especificación anterior de la misma forma, utilizando nuestras ecuaciones como axiomas.

A continuación se presenta la especificación formal del manejador de bases de datos.

Caja 1. Especificación formal
del manejador de bases de datos.

SISTEMA MANEJADOR DE BASES DE DATOS

Sea la signatura S-tipificada $\Sigma = (S,F)$ donde

$S = (\text{nombres}, \text{regper}, \text{basedatos}, \text{boolean})$

y F esta compuesto de las siguientes funciones

F $\lambda, \text{boolean} = (\text{cierto}, \text{falso})$

F $\lambda, \text{basedatos} = (\text{crea})$

F $\text{basedatos} \text{ nombres} \text{ regper}, \text{basedatos} = (\text{agrega}, \text{inserta})$

F $\text{basedatos} \text{ nombres}, \text{regper} \cup (\nu) = (\text{consulta})$

F $\text{basedatos} \text{ nombres}, \text{boolean} = (\text{pertenece})$

F $\text{basedatos}, \text{boolean} = (\text{vacio})$

Definamos los siguientes conjuntos de variables:

X $\text{nombres} = (\text{nom1}, \text{nom2})$ X $\text{basedatos} = (\text{bd})$

X $\text{regper} = (\text{dat})$

Sea el conjunto de variables de Σ igual a X donde

$$X = X_{\text{nombres}} \cup X_{\text{basedatos}} \cup X_{\text{regper}}$$

Sea pues la especificación $\Sigma' = (\Sigma, E)$ donde Σ es la signatura anterior y con el conjunto de variables X.

E igual al siguiente conjunto de ecuaciones:

$\text{vacio}(\text{crea}) = \text{cierto}$
 $\text{vacio}(\text{inserta}(\text{bd}, \text{nom1}, \text{dat})) = \text{falso}$
 $\text{pertenece}(\text{crea}, \text{nom1}) = \text{falso}$


```

pertenece(inserta(bd,nom2,dat),nom1) =
    Si nom1 = nom2 Entonces cierto
    Sino pertenece(bd,nom1)
borra(crea,nom1) = e
borra(inserta(bd,nom2,dat),nom1) =
    Si nom1 = nom2 Entonces bd
    Sino inserta(borra(bd,nom1),nom2,dat)
consulta(crea,nom1) = e
consulta(inserta(bd,nom2,dat),nom1) =
    Si nom1 = nom2 Entonces dat
    Sino consulta(bd,nom1)
agrega(bd,nom1,dat) =
    Si pertenece(bd,nom1) Entonces e
    Sino inserta(bd,nom1,dat)

```


En nuestra especificación existe un cierto símbolo e el cual representa al error, esta constante es hasta cierto punto sin tipo o de todos los tipos y significa que cuando en alguna operación nos regrese error entonces estamos realizando un trabajo inválido.

Continuación... caja 1.


Ahora bien ¿Qué sería un álgebra de la especificación anterior?

Un álgebra de la especificación anterior puede ser la implementación, en algún lenguaje, de nuestra especificación.

Caja 2. Σ -álgebra.



TIPOS DE DATOS ABSTRACTOS.



CAPITULO III.

TIPOS DE DATOS ABSTRACTOS.

En el capítulo anterior se definieron los conceptos básicos para el desarrollo de las especificaciones algebraicas y se presentaron algunos ejemplos tanto de tipo numérico como de sistemas computacionales.

En el presente capítulo veremos los tipos de datos abstractos (TDA), los cuales son el antecedente de las especificaciones algebraicas. Los tipos de datos abstractos fueron desarrollados en la década de los 70s y actualmente existen una gran variedad de artículos que tratan este tema. Entre los principales artículos encontramos los escritos por Liskov, Zilles, Guttag, Horning y Hoare.

El principal objetivo de los tipos de datos abstractos es, construir tipos de datos que sean independientes de la representación.

Comenzaremos este capítulo definiendo lo que entenderemos por tipo de dato, estructura de datos y tipo de dato abstracto, enseguida construiremos un álgebra distinguida para una especificación dada, la cual tendrá por nombre álgebra de términos cocientes y veremos por que esta álgebra es tan importante.

3.1 TIPOS DE DATOS.

Generalmente al resolver un problema creamos algún tipo de representación del mismo y le asociamos información. Si la solución del problema llega a ser programada en una computadora, tendremos que diseñar representaciones de los objetos y de sus operaciones, para la máquina. Esta representación, tendrá que ser lo más anegada a como nosotros concebimos el problema y su solución. Esto nos lleva a la siguiente definición:

Definición: Un Tipo de Dato es una colección de elementos y las operaciones que los transforman.

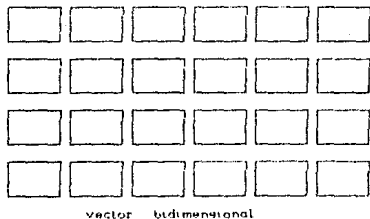
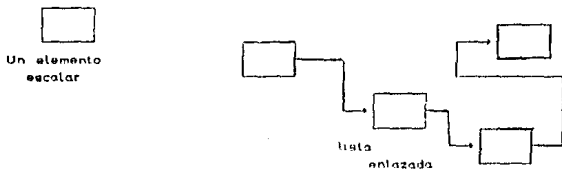
Ejemplos típicos son los enteros con sus operaciones

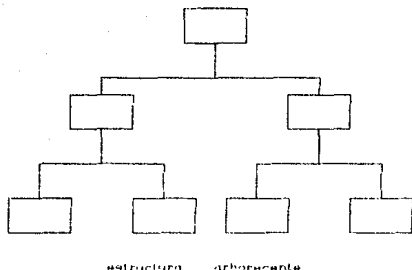
estructuras, de acuerdo a sus necesidades y a como él concibe los objetos con los cuales está íntimamente relacionada su representación de la solución del problema.

Así pues, la organización y complejidad de una estructura de datos sólo está limitada por la ingeniosidad del diseñador.

Sin embargo, existe un número limitado de estructuras clásicas, que forman los bloques con los que se construyen estructuras más sofisticadas.

Estas estructuras clásicas se muestran en la siguiente figura.





3.3 TIPOS DE DATOS ABSTRACTOS.

En los últimos años los tipos de datos abstractos, han cobrado gran importancia, debido tal vez al origen de una gran cantidad de lenguajes de computación y a la necesidad de contar con programas confiables.

Si contamos con cierta variedad de lenguajes, nos gustaría que a partir de una misma especificación sobre un problema, se pueda programar en cualquier lenguaje disponible, sin necesidad de cambiar nuestra especificación. Además, es muy importante poder demostrar que nuestra implementación corresponde a la especificación, esto por consiguiente nos da programas confiables.

De aquí que sean tan importantes los tipos de datos abstractos y las especificaciones de los mismos.

3.3.1 DEFINICION DE TIPOS DE DATOS ABSTRACTOS.

Para el diseño de algoritmos y sistemas computacionales, uno no se interesa en la representación concreta de un tipo de dato sino mas bien en un nivel más abstracto, en el cual vamos a distinguir claramente como interactúan nuestros objetos con otros objetos y consigo mismos.

Definición. Un Tipo de Dato Abstracto (TDA) es una clase de tipo de dato el cual es independiente de la representación.

Actualmente existe una gran variedad de especificaciones de varios tipos de datos abstractos, en este capítulo desarrollaremos el TDA Pilas y sobre éste ejemplo se desarrollará la teoría.

3.3.2 EJEMPLO DE UN TIPO DE DATO ABSTRACTO. TDA(PILAS).

Se construirá la especificación de Pilas utilizando los conocimientos intuitivos sobre esta estructura de datos tratando de captar sus propiedades en forma abstracta, y sin importar una forma particular de representación. En pocas palabras, lo que se desea hacer es expresar QUE es una pila y no dar un ejemplo de una representación de pila, en particular.

Concepto.

En ciencias de la computación existe una estructura muy conocida, cuyo nombre es pila. Las pilas se utilizan para guardar objetos o elementos, como por ejemplo; números, caracteres, registros, ventanas, archivos e incluso pilas, pero, se verá mas tarde, que el tipo de éstos objetos no influye de ninguna manera sobre el comportamiento de las pilas. Por esta razón vamos a llamar **Elemento** el tipo de los objetos guardados en la pila y se considerará como un parámetro de la especificación.

La principal característica de una pila es que el acceso de los elementos a la pila es de tal forma que el último en entrar es el primero en salir, las pilas se utilizan en gran cantidad de programas (siendo su principal aportación en programación de sistemas).

En la literatura a la pila se le considera como un objeto, el cual se modificará por las operaciones de meter y sacar elementos de ella, pero, para nuestro caso veremos a la pila como un valor el cual será modificado o transformado por una serie de funciones.

Así pues un primer valor que podrá tomar una pila es aquel en donde la pila no contiene elemento alguno y que la

consideraremos como una operación constante

crea_pila; \longrightarrow Pila

la cual generará una pila vacía.

Dada una pila p se puede transformar en una pila p' metiendo un elemento a p . Así que a la acción de meter un elemento a la pila se le puede considerar como una función que dado un elemento e y una pila p devuelve como valor una pila nueva. La especificación de esta operación queda como sigue;

mete; Elemento Pila \longrightarrow Pila

Con estas dos operaciones se puede construir cualquier valor posible para una pila. Es por esta razón que a las funciones de `crea_pila` y `mete` se les llaman *constructoras* o *generadoras* del tipo de dato que estamos construyendo, que en este caso es Pilas.

Pero, para los problemas en los que se utilizan las pilas, se necesitan de otras funciones para modificar una pila; entre estas operaciones encontramos una que nos regresa cierto si la pila esta vacía y falso en otro caso. Esta función tendrá como argumento a una pila y regresará un valor de cierto o falso, así que su especificación queda como sigue;

es_vacia; Pila \longrightarrow (cierto,falso)

En ciencias de la computación al tipo de dato cuyos posibles valores son: cierto o falso, tiene por nombre Booleano, así que reescribiendo la especificación, anterior, queda como:

es_vacia; Pila \longrightarrow Booleano.

y las propiedades que debe cumplir esta función son las siguientes

`es_vacia(crea_pila)` = cierto

`es_vacia(mete(e,p))` = falso

en donde e es del tipo elemento y p es una pila.

Para la especificación de las siguientes operaciones se asume que $e \in \text{Elemento}$ y $p \in \text{Pilas}$.

Como siguiente operación tenemos aquella que nos permite obtener el último elemento metido a la pila, es claro que ésta transformación tendrá como argumento una pila y regresará un valor de tipo elemento, esta operación no altera a la pila, su especificación y sus propiedades a continuación se presentan:

```
tope: Pila  $\longrightarrow$  Elemento
tope(crea_pila) = error
tope(mete(e,p)) = e
```

Por último se necesita de una operación que saque el elemento que se encuentra en el tope de la pila. Esta operación tiene como parámetro un elemento del tipo pila y nos regresa a la pila ya modificada, la especificación de esta operación y sus propiedades quedan como sigue:

```
saca: Pila  $\longrightarrow$  Pila
saca(crea_pila) = error
saca(mete(e,p)) = p
```

Observemos que

```
tope(crea_pila) = error
saca(crea_pila) = error
```

esto lo que nos indica es que las aplicaciones anteriores son indebidas, la existencia del valor `error` la supondremos, así como también vamos a suponer que `error` pertenece a todos los tipos incluyendo al tipo Pila. La existencia del valor `error` nos lleva a la siguiente convención: si alguno de los argumentos de una función es `error` entonces la operación devuelve el valor de `error`.

Las operaciones con esta propiedad se llaman estrictas, y en el caso de nuestra especificación vamos a suponer que las

3.3.4 ALGEBRA PARA LA ESPECIFICACION DE PILAS.

Un ejemplo de álgebra para la especificación de Pilas es la siguiente:

$Pil = (A, A', B, a_1, \dots, a_n, \text{\$error}, \text{\$vacia}, \text{\$mete}, \text{\$svacia}, \text{\$tome}, \text{\$faca})$
 de donde A es el conjunto cuyos elementos son a_1, \dots, a_n ; A' es el conjunto de todas las posibles cadenas, incluyendo la cadena vacía de los elementos de A , y B es el conjunto de los valores de verdad, verdadero y falso. Y las operaciones se definen como sigue:

$\text{\$vacia} : \longrightarrow A'$
 $\text{\$vacia}$ es la cadena vacía en A'

$\text{\$mete} : A, A' \longrightarrow A'$
 $\text{\$mete}(a, c) = ac$

$\text{\$svacia} : A' \longrightarrow \text{Boolean}$
 $\text{\$svacia}(c) = \text{Si } c = \text{\$vacia} \text{ Entonces verdadero}$
 Sino falso

$\text{\$faca} : A' \longrightarrow A'$
 $\text{\$faca}(c) = \text{Si } \text{\$svacia}(c) \text{ Entonces } \text{\$error}$
 $\text{Sino Si } c = ac' \text{ Entonces } c'$

$\text{\$tome} : A' \longrightarrow A$
 $\text{\$tome}(c) = \text{Si } \text{\$svacia}(c) \text{ Entonces } \text{\$error}$
 $\text{Sino Si } c = ac' \text{ Entonces } a$

Hasta este momento vemos que la especificación de Pilas y el álgebra Pil , representan el mismo tipo de datos abstractos TDA(PILAS), así como de todas las álgebras "isomorfas" a PILAS, incluyendo el álgebra de términos cocientes el cual definiremos más adelante, primero vamos a definir una relación de congruencia entre los términos elementales de una especificación, y mas adelante veremos como podemos demostrar que Pilas y Pil representan el mismo objeto.

5. Para cada derivación $t_1 \neq t_2$ via E, entre $t_1, t_2 \in T_F$ implica $t_1 \equiv t_2$
6. Si existe una ESPEC-álgebra B con $eval_B(t_1) \neq eval_B(t_2)$ para $t_1, t_2 \in T_F$ entonces $t_1 \neq t_2$.

Demostración:

Sea la especificación ESPEC=(S,F,E), A una ESPEC-álgebra y la evaluación $eval_A: T_F \longrightarrow A$ entonces

1. Sea $t_1 \in T_F$ entonces $eval_A(t_1) = eval_A(t_1)$ ya que $t_1 = t_1$.

$$\therefore t_1 \equiv t_1$$

2. Sean t_1 y $t_2 \in T_F$ y supongamos que $t_1 \equiv t_2$ entonces por la definición de \equiv tenemos que

$eval_A(t_1) = eval_A(t_2)$ conmutando la igualdad tenemos

$$eval_A(t_2) = eval_A(t_1) \quad \therefore t_2 \equiv t_1$$

3. Sean $t_1, t_2, t_3 \in T_F$ y tal que se cumple lo siguiente

$t_1 \equiv t_2$ y $t_2 \equiv t_3$ entonces tenemos las siguientes igualdades

$$eval_A(t_1) = eval_A(t_2) \quad \text{y} \quad eval_A(t_2) = eval_A(t_3) \quad \text{entonces}$$

$eval_A(t_1) = eval_A(t_3)$ por las propiedades de la igualdad y

$$\therefore t_1 \equiv t_3$$

Hasta estas tres propiedades se ha demostrado que \equiv es una relación de equivalencia.

4. Sean $t_i, t'_i \in S_i$ y a la vez términos elementales $\forall i=1, \dots, n$ tal que se cumplen las siguientes equivalencias $t_i \equiv t'_i$ $\forall i=1, \dots, n$ y sea $G \in F$, con la aridad de G igual a $n \geq 1$.

PD.

$$G(t_1, \dots, t_n) \equiv G(t'_1, \dots, t'_n) \quad \text{o que}$$

$$eval_A(G(t_1, \dots, t_n)) = eval_A(G(t'_1, \dots, t'_n)).$$

Dem.

$$eval_A(G(t_1, \dots, t_n)) = G_A(eval_A(t_1), \dots, eval_A(t_n))$$

$$= G_A (eval_A(t'_1), \dots, eval_A(t'_n))$$

$$= eval_A(G(t'_1, \dots, t'_n)).$$

$$\therefore G(t_1, \dots, t_n) = G(t'_1, \dots, t'_n).$$

Las propiedad 5 no se demostrará, pues para realizar esta demostración se necesita haber definido algún cálculo ecuacional o algunas reglas para derivar términos a partir de otros términos. Y en realidad no definimos nada de esto.

Y la propiedad 6 es la contrapositiva de la definición.

3.5 ALGEBRA DE TERMINOS COCIENTES(T_{ESPEC}).

Definición. Dada la especificación ESPEC(S,F,E) el álgebra de términos cocientes,

$T_{ESPEC} = ((Q_s)_{s \in S}, (N_o)_{o \in F})$ está definida como sigue:

1. Para cada $s \in S$ se tiene un conjunto base $Q_s = (\{t\} / t \in T_{F,s})$ donde la clase de equivalencia $\{t\}$ esta definida por $\{t\} = \{t' / t' \equiv t\}$, donde \equiv es la congruencia entre términos elementales.
2. Para cada símbolo de constante $K; \longrightarrow s$ en F la constante K_o es la clase de equivalencia generada por K .

$$K_o = [K]$$

3. Para cada símbolo de operación $G; s_1 \dots s_n \longrightarrow s$ con $G \in F$, tenemos que

$G_o : Q_{s_1} \times \dots \times Q_{s_n} \longrightarrow Q_s$ esta definido por

$$G_o([t_1], \dots, [t_n]) = [G(t_1, \dots, t_n)]$$

para todos los términos del tipo s_i con $i=1, \dots, n$.

Observación. La operación G_{α} está bien definida, para cada clase de equivalencia $[t]$, esto es consecuencia directa de la propiedad número cuatro, de la relación de congruencia \equiv .

Ejemplo. Continuando con nuestro ejemplo del TDA Pilas, construimos la siguiente álgebra cociente.

Comenzaremos construyendo los siguientes conjuntos de términos para la signatura (S,F) de la especificación Pilas.

$$T_{\text{Elemento}} = T_{F,\text{Elemento}}, T_{\text{Pilas}} = T_{F,\text{Pilas}} \quad \vee \quad T_{\text{Boolean}} = T_{F,\text{Boolean}}$$

Recordemos que $T_{f,S}$ es igual al conjunto de términos del tipo s , sin variables, así que estos conjuntos son los siguientes:

T_{Elemento} se construye de la siguiente manera:

- $a_1, \dots, a_n \in T_{\text{Elemento}}$
- $\vee \forall tp \in T_{\text{Pilas}}$ entonces
- $\text{tope}(tp) \in T_{\text{Elemento}}$

El conjunto T_{Boolean} está definido como sigue:

- verdadero, falso $\in T_{\text{Boolean}}$
- $\forall tp \in T_{\text{Pilas}}$ se tiene que
- $\text{es_vacía}(tp) \in T_{\text{Boolean}}$

Y por último el conjunto T_{Pilas} está definido por

- $\text{crea_pila} \in T_{\text{Pilas}}$
- $\forall te \in T_{\text{Elemento}} \quad \vee \quad \forall tp \in T_{\text{Pilas}}$
- $\text{mete}(te, tp), \text{saca}(te, tp) \in T_{\text{Pilas}}$

3.6 HOMOMORFISMOS E ISOMORFISMOS.

Cuando, en matemáticas, estudiamos las estructuras algebraicas tales como grupos, anillos y otras estructuras, nos encontramos con los conceptos de homomorfismo e isomorfismo.

Los homomorfismos son transformaciones que nos relacionan estructuras, y son además, una herramienta poderosa para el estudio formal de conceptos utilizados en las especificaciones algebraicas. Más adelante hablaremos de ellos.

Los isomorfismos son homomorfismos que cumplen ciertas propiedades adicionales.

Cuando en álgebra decíamos que dos grupos fuesen isomorfos es que hasta cierto punto son iguales. Lo único en que difieren es que los elementos se denominan en forma distinta.

El isomorfismo es el que da la clase de esta diferencia de denominación, y con él, conociendo un determinado cálculo en un grupo, podemos efectuar el cálculo análogo en el otro. Así pues, podemos pensar en el isomorfismo como un diccionario que nos permite traducir una frase de un idioma a otro idioma.

En forma análoga, el isomorfismo en las álgebras de una especificación, nos indicará que las estructuras de ambas álgebras son idénticas salvo por los nombres de los símbolos.

Pero ¿Cuál es la importancia de los homomorfismos e isomorfismos entre la especificación de un sistema de cómputo y sus posibles implementaciones. ? Los homomorfismos e isomorfismos entre álgebras de una especificación son la formulación algebraica para el concepto de "independencia de la representación", que es el principal objetivo de los tipos de datos abstractos.

Sean A y B dos álgebras de la misma signatura $SIG=(S,F)$ o de la especificación $ESPEC = (S,F,E)$. Se define lo siguiente:

Definición.

1. Un **HOMOMORFISMO** $f:A \longrightarrow B$ o SIG -o $ESPEC$ -homomorfismo, es una familia de funciones

$$f_s : A_s \longrightarrow B_s \quad \forall s \in S \quad \text{tal que se cumple lo siguiente:}$$

a) para todo símbolo constante $k_s \longrightarrow s$ en F y $s \in S$

$$f_{g_i}^s(k_A) = k_B$$

b) y para cada operación $G: s_1 \dots s_n \longrightarrow s$ en F y $\forall a_i \in A_{g_i}$ para $i=1, \dots, n$

tenemos que

$$f_{g_i}^s(G(a_1, \dots, a_n)) = G_{g_i}^s(f_{g_1}^{s_1}(a_1), \dots, f_{g_n}^{s_n}(a_n))$$

2. Un **Isomorfismo** es un homomorfismo $f: A \longrightarrow B$, tal que toda función $f_{g_i}^s: A_{g_i} \longrightarrow B_{g_i} \forall s \in S$ es biyectiva. Si existe tal función diremos que A es isomorfo a B , y lo denotaremos por $A \cong B$.

Recordemos que una función $f: A \longrightarrow B$ es:

- i) Inyectiva cuando $\forall a, b \in A$ se tiene que $a \neq b \rightarrow f(a) \neq f(b)$.
- ii) Sobre si $\forall b \in B$, $\exists a \in A$ tal que se cumple $f(a)=b$.
- iii) Biyectiva si f es inyectiva y sobre.

Ejemplos.

En el capítulo anterior se definió la **signatura SIG**-para los enteros, a continuación se escribe la especificación algebraica para esa signatura.

ent =

tipos:

ent

oper:

cero : \longrightarrow ent

suc : ent \longrightarrow ent

pred : ent \longrightarrow ent

var

z : ent

ecuac:

suc(pred(z)) = z

pred(suc(z)) = z

Fin ent.

También en ése mismo capítulo se presentaron dos álgebras para la signatura, ahora bien las dos álgebras a la vez son ent-álgebras, es fácil demostrar que \mathbb{Z} y TRID cumplen con las ecuaciones de ent. Pero \mathbb{Z} y TRID son isomorfos, o al menos existe un homomorfismo de TRID a \mathbb{Z} , ¿?

La respuesta a la pregunta anterior es que, no existe un homomorfismo de TRID a \mathbb{Z} y por consiguiente no son isomorfos.

La demostración de que no existe un homomorfismo de TRID a \mathbb{Z} es por contradicción.

Dem. Supongamos que existe $f: \text{TRID} \rightarrow \mathbb{Z}$, notese que como existe un sólo tipo en nuestra especificación entonces tenemos una sólo función, tal que f es un homomorfismo y por consiguiente debe cumplir que $f: \text{ent}_{\text{TRID}} \rightarrow \text{ent}_{\mathbb{Z}}$ y

$$i) f(\text{cero}_{\text{TRID}}) = \text{cero}_{\mathbb{Z}} \quad \text{o} \quad f(0) = 0$$

$$ii) f(\text{suc}_{\text{TRID}}(t)) = \text{suc}_{\mathbb{Z}}(f(t)) \quad \forall t \in \text{TRID} \quad \text{y}$$

$$f(\text{pred}_{\text{TRID}}(t)) = \text{pred}_{\mathbb{Z}}(f(t)) \quad \forall t \in \text{TRID}.$$

observemos que podemos definir a $f(0)=0$ y no existe problema, pero ¿qué sucede con las otras dos propiedades?, pues tenemos lo siguiente:

$$f(\text{suc}_{\text{TRID}}(-1)) = f(0) \quad \text{pero } f(0)=0 \quad \text{por como se definió, } \rightarrow$$

$$\text{suc}_{\mathbb{Z}}(f(-1)) = 0 \quad \text{pues } f \text{ es homomorfismo} \quad \rightarrow$$

$$f(-1) = -1 \quad \text{por como se definió } \mathbb{Z} \quad \therefore f(-1) = -1.$$

También tenemos lo siguiente

$$f(\text{pred}_{\text{TRID}}(1)) = f(0) \quad \text{y } f(0) = 0 \quad \text{y}$$

$$\text{pred}_{\mathbb{Z}}(f(1)) = 0 \quad \text{ya que } f \text{ es un homomorfismo y por lo anterior}$$

$$\therefore f(1) = 1 \quad \text{pero observemos lo siguiente}$$

$$f(\text{suc}_{\text{TRID}}(1)) = f(-1) = -1 \quad \text{y esto debe ser igual a } \text{suc}_{\mathbb{Z}}(f(1))$$

pero

$\text{suc}_{\mathbb{Z}}(f(1)) = \text{suc}_{\mathbb{Z}}(1) = 2$ y $2 \neq 1 \quad \forall$. por lo tanto f no es un homomorfismo de TRID a \mathbb{Z} .

Por lo tanto no existe un homomorfismo de TRID a \mathbb{Z} .

Construyamos ahora el álgebra de términos cocientes T_{ent} para la especificación de ent:

$$T_{\text{ent}} = (Q_{\text{ent}}, 0_{\alpha}, \text{suc}_{\alpha}, \text{pred}_{\alpha}) \text{ de donde}$$

$$Q_{\text{ent}} = (\{\text{suc}^n(0)\} / n \geq 1) \cup \{0\} \cup (\{\text{pred}^n(0)\} / n \geq 1)$$

$$0_{\alpha} = \{0\}$$

$$\text{suc}_{\alpha}(\{\text{suc}^n(0)\}) = \{\text{suc}^{n+1}(0)\} \quad \forall n \geq 0$$

$$\text{suc}_{\alpha}(\{\text{pred}^n(0)\}) = \{\text{pred}^{n-1}(0)\} \quad \forall n \geq 0$$

$$\text{pred}_{\alpha}(\{\text{suc}^n(0)\}) = \{\text{suc}^{n-1}(0)\} \quad \forall n \geq 0$$

$$\text{pred}_{\alpha}(\{\text{pred}^n(0)\}) = \{\text{pred}^{n+1}(0)\} \quad \forall n \geq 0$$

ahora bien $f: T_{\text{ent}} \longrightarrow \mathbb{Z}$ definido de la siguiente forma es un isomorfismo:

$$f(\{\text{suc}^n(0)\}) = n \quad n \in \mathbb{N}$$

$$f(\{\text{pred}^n(0)\}) = -n \quad n \in \mathbb{N}$$

Recordemos que como T tiene un sólo tipo entonces f consiste de una sólo función, y la denotaremos como f , entonces en realidad

$$f: Q_{\text{ent}} \longrightarrow \mathbb{Z}_{\text{ent}}$$

Demostración.

PD. i) f es un homomorfismo.

ii) f es biyectiva.

Dem.

i) Sea $f: T_{\text{ent}} \longrightarrow \mathbb{Z}$ donde T_{ent} es el álgebra de términos cocientes.

$$i) f(0_{\alpha}) = f(\{0\}) \quad \text{por definición de } 0_{\alpha}, \quad \text{pero}$$

$$f(\{0\}) = f(\{\text{suc}^0(0)\}) \quad \text{pero}$$

$f(\text{suc}^0(0)) = 0$ por la definición de f ,
 y como $0_{\mathbb{Q}}$ es la única constante se cumple la primera
 parte de la definición de homomorfismo.

(i)

$$a) f(\text{suc}_{\mathbb{Q}}(\text{suc}^n(0))) = f(\text{suc}^{n+1}(0)) \text{ por def. de } \text{suc}_{\mathbb{Q}}.$$

$$y \quad f(\text{suc}^{n+1}(0)) = n+1 \quad \forall n \in \mathbb{N}. \quad Y$$

$$\text{suc}_{\mathbb{Z}}(f(\text{suc}^n(0))) = \text{suc}_{\mathbb{Z}}(n) = n+1.$$

$$\therefore f(\text{suc}_{\mathbb{Q}}(\text{suc}^n(0))) = \text{suc}_{\mathbb{Z}}(f(\text{suc}^n(0))), \quad \forall n \in \mathbb{N}.$$

$$b) f(\text{suc}_{\mathbb{Q}}(\text{pred}^n(0))) = f(\text{pred}^{n-1}(0)) \text{ por def. } \text{suc}_{\mathbb{Q}} \text{ y}$$

$$f(\text{pred}^{n-1}(0)) = 1-n \quad \text{por def. de } f. \quad Y$$

$$\text{suc}_{\mathbb{Z}}(f(\text{pred}^n(0))) = \text{suc}_{\mathbb{Z}}(-n) \quad y$$

$$\text{suc}_{\mathbb{Z}}(-n) = 1-n \quad \text{por def. } \text{suc}_{\mathbb{Z}} \quad \forall n \geq 0.$$

$$\therefore f(\text{suc}_{\mathbb{Q}}(\text{pred}^n(0))) = \text{suc}_{\mathbb{Z}}(f(\text{pred}^n(0)))$$

$$\therefore f(\text{suc}_{\mathbb{Q}}(X)) = \text{suc}_{\mathbb{Z}}(f(X)), \quad \forall X \in \mathbb{Q}_{\text{ent}}.$$

En forma análoga se puede probar que

$$f(\text{pred}_{\mathbb{Q}}(X)) = \text{pred}_{\mathbb{Z}}(f(X)) \quad \forall X \in \mathbb{Q}_{\text{ent}}.$$

y por lo tanto f es un homomorfismo.

(ii) f es biyectiva.

Por la forma en que se define f se ve que f es inyectiva y
 sobreyectiva, y por lo tanto biyectiva.

Y por i) y ii) f es un isomorfismo. Por lo que T_{ent} y \mathbb{Z}
 son dos representaciones válidas de los enteros.

3.7 PROPIEDADES DEL ALGEBRA DE TERMINOS COCIENTES T_{ESPEC} .

Teorema. El álgebra de términos cocientes T_{ESPEC} de una
 especificación $\text{ESPEC}=(S,F,E)$, cumple con las siguientes propiedades

1. La función de evaluación $\text{eval}: T_F \longrightarrow T_{\text{ESPEC}}$ es igual a la función natural $\text{nat}: T_F \longrightarrow T_{\text{ESPEC}}$ donde $\text{nat}(t)=[t]$ y además es sobreyectiva.

2. $\forall \xi \in E$ y $\xi=(t_1, t_2)$ con $t_1, t_2 \in T_F$ es válida en $T_{\text{ESPEC}} \iff \xi$ es válida en todas las ESPEC-álgebras A.

3. T_{ESPEC} es una ESPEC-álgebra.

Demostración.

Sea la especificación $\text{ESPEC}=(S, F, E)$ y T_{ESPEC} álgebra de términos cocientes de ESPEC entonces:

1. Pd. $\text{eval}(t)=\text{nat}(t) \quad \forall t \in T_F$ puesto que estamos demostrando una propiedad de los términos elementales, utilizaremos inducción estructural, la cual fue introducida en el capítulo anterior.

i) Pd. Para todo símbolo de constante K se cumple la igualdad.

Así que sea $N \in T_F$, con N una constante entonces $\text{eval}(N)=N_{\alpha}$ esto por la definición de eval.
pero

$\text{nat}(N)=[N]$ y $N_{\alpha}=[N]$ esto por la definición de N_{α} .
 $\therefore \text{eval}(N)=\text{nat}(N)$ para toda N constante en T_F .

ii) Sea $G \in F_F$, con $G: s_1, \dots, s_n \longrightarrow s \quad \forall s_i, s \in S_F$, donde F_F y S_F son el conjunto de funciones y de los tipos, respectivamente, de la álgebra T_F . Y

supongamos que $\forall t_i \in s_i$ se cumple que $\text{eval}(t_i)=\text{nat}(t_i) \iff$

Pd. $\text{eval}(G(t_1, \dots, t_n))=\text{nat}(G(t_1, \dots, t_n))$.

tenemos lo siguiente:

$\text{eval}(G(t_1, \dots, t_n)) = G_{\alpha}(\text{eval}(t_1), \dots, \text{eval}(t_n))$ por def. de la función eval, pero como $\text{eval}(t_i)=\text{nat}(t_i) \quad \forall i=1, \dots, n \Rightarrow$
 $G_{\alpha}(\text{eval}(t_1), \dots, \text{eval}(t_n)) = G_{\alpha}(\text{nat}(t_1), \dots, \text{nat}(t_n))$ y

$G_{\alpha}(\text{nat}(t_1), \dots, \text{nat}(t_n)) = G_{\alpha}([t_1], \dots, [t_n])$ por def. de nat , pero

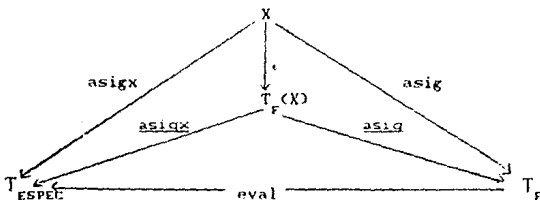
$G_{\alpha}([t_1], \dots, [t_n]) = [G(t_1, \dots, t_n)]$ por def. de G_{α} , y

$[G(t_1, \dots, t_n)] = \text{nat}(G(t_1, \dots, t_n))$ por def. de nat.

$\underline{asig}_X(I) = \underline{asig}_X(D)$ donde $\underline{asig}_X : T_F(X) \longrightarrow T_{ESPEC}$
 es la asignación extendida de $asig_X$.

Demostración. Sea la ecuación (I,D), con $I, D \in T_F(X)$ y
 sea la asignación $asig_X : X \longrightarrow T_{ESPEC}$ entonces
 Si I y $D \in T_F$ entonces por 2 se sigue la igualdad.

Si no es cierto que I y $D \in T_F$ tenemos lo siguiente, veamos
 el siguiente diagrama conmutativo, el cual nos puede dar una
 idea de la demostración:



$asig_X : X \longrightarrow T_{ESPEC}$ es una función de asignación y $eval$
 es la función de evaluación $eval : T_F \longrightarrow T_{ESPEC}$ del segundo
 inciso de este Teorema, tenemos que $eval$ es una función
 sobreyectiva, así que vamos a definir una nueva función de
 asignación de X a T_F , y está dada como sigue:

$asig : X \longrightarrow T_F$, tal que $\forall x \in X$
 $asig(x) = t'$, donde t' es el elemento de T_F , tal que
 $asig_X(x) = eval(t')$, t' está bien definido por que $eval$ es
 sobre. Así que tendremos la siguiente igualdad

$$asig_X = eval \circ asig.$$

supongamos que $t_1 = \underline{asig}_X(I)$ y $t_2 = \underline{asig}_X(D)$; donde $t_1, t_2 \in T_F$.

Tomemos una ESPEC-álgebra A , y sea $eval_A$ la función de
 evaluación de T_F a A .



SEMANTICA
DE LAS
ESPECIFICACIONES.



SEMANTICA DE LAS ESPECIFICACIONES.

En este capítulo comenzamos definiendo la semántica de una especificación como la clase de todas las álgebras isomorfas al álgebra de términos cocientes de la especificación.

También, en el presente capítulo estudiaremos más a fondo las propiedades de los homomorfismos, esto es debido a que por medio de homomorfismos se puede realizar un estudio más formal sobre las evaluaciones y las sustituciones de términos abstractos por términos específicos. Además se enunciarán y se demostrarán más propiedades del álgebra de términos cocientes y del porque esta álgebra representa la semántica de una especificación.

4.1 SEMANTICA DE UNA ESPECIFICACION.

Definición. La semántica de una especificación ESPEC es la clase

$$TDA(ESPEC) = \{ A / A \cong T_{ESPEC} \},$$

de todas las álgebras

isomorfas al álgebra de términos cocientes de la especificación.

Definición. Dada una SIG-álgebra A, la especificación ESPEC es llamada Correcta, si $A \cong T_{ESPEC}$.

Definición. La Semántica Clásica o débil, de una especificación ESPEC es la clase

$$Alg(ESPEC) = \{ A / A \text{ es una ESPEC-álgebra} \}$$

A continuación estudiaremos a los homomorfismos.

4.2 PROPIEDADES DE LOS HOMOMORFISMOS.

En la sección 3.6 se vio la definición de homomorfismo e isomorfismo, ahora veremos algunas propiedades importantes que cumplen.

4.2.2 PRINCIPALES PROPIEDADES DE LOS HOMOMORFISMOS.

En esta sección se presentarán algunas propiedades importantes de los homomorfismos, debido a que los homomorfismos los utilizaremos como una herramienta descriptiva para dar una formulación concisa de conceptos, propiedades y demostraciones.

Teorema. (Propiedades de los homomorfismos.) Sea $SIG=(S,F)$ una signatura, y sean A,B,C y D SIG -álgebras. Entonces se cumplen las siguientes propiedades:

1. La composición de SIG -homomorfismos

$$f:A \longrightarrow B \quad \text{y} \quad g:B \longrightarrow C \quad \text{denotado por} \\ gf:A \longrightarrow C$$

y definida por $g \circ f(x) = g(f(x))$ para $x \in A$, se S , es

también un homomorfismo y además esta composición es asociativa.

2. Si $f:A \longrightarrow B$ es un SIG -homomorfismo, entonces cualquier ecuación e , sin variables, que sea válida en A entonces también es válida en B .

3. Si $f:A \longrightarrow B$ es un SIG -homomorfismo y si además es sobreectivo entonces cualquier ecuación e que es válida en A también lo será en B .

Demostración.

1. Para todo símbolo de constante $K \in F$ y $s \in S$ tenemos que

$$\begin{aligned} g \circ f(K_A) &= g(f(K_A)) && \text{por def. de } gf \\ &= g(K_B) && \text{por prop. de los homomorfismos.} \\ &= K_C && \text{por prop. de los homomorfismos.} \end{aligned}$$

Para toda $G \in F$, $G; s_1, \dots, s_n \longrightarrow s$, $s \in S$ y sea $a_i \in A$, $\forall i=1..n$ tenemos que

$$\begin{aligned}
g_{\theta} f_{\theta} (G_A(a_1, \dots, a_n)) &= g_{\theta} (f_{\theta} (G_A(a_1, \dots, a_n))) \quad \text{por def. de gf} \\
&= g_{\theta} (G_B(f_{\theta_1}(a_1), \dots, f_{\theta_n}(a_n))) \\
&\quad \text{por def. de los hom.} \\
&= G_C(g_{\theta_1}(f_{\theta_1}(a_1)), \dots, g_{\theta_n}(f_{\theta_n}(a_n))) \\
&= G_C(g_{\theta_1} f_{\theta_1}(a_1), \dots, g_{\theta_n} f_{\theta_n}(a_n))
\end{aligned}$$

Por lo que gf es un homomorfismo. La asociatividad se sigue inmediatamente de la asociatividad de las funciones $g_{\theta_i}, f_{\theta_i}$ y h_{θ_i} .

Para la demostración de las otras dos propiedades se necesita de un resultado que no se ha demostrado, así es que se dejan para más tarde.

4.3 ALGEBRA DE TERMINOS.

En el capítulo 2 se definió el concepto de término, con variables y sin variables, para una signatura dada, así como también se definió lo que entenderíamos por la evaluación de términos, en esta sección se demostrarán algunas de las propiedades más importantes sobre evaluación de términos y asignación de variables. Comenzamos viendo lo que es un álgebra de términos, para enseguida dar paso a las principales propiedades de esta álgebra distinguida.

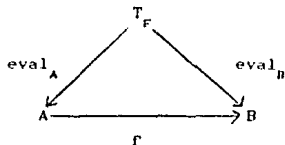
Anteriormente definimos a los términos de una signatura como sigue:

Si $SIG=(S,F)$ es una signatura y X el conjunto de variables entonces $T_F(X) = \bigcup_{s \in S} T_{F,s}(X)$ es el conjunto de los términos de SIG .

Ahora definiremos el álgebra de términos, esta álgebra es muy importante para nosotros ya que es igual a $T_F(X)$ y a partir de esta álgebra se definen los conceptos de evaluación, asignación y asignación extendida, los cuales nos permiten interpretar objetos de una estructura abstracta a objetos específicos o tal vez abstractos.

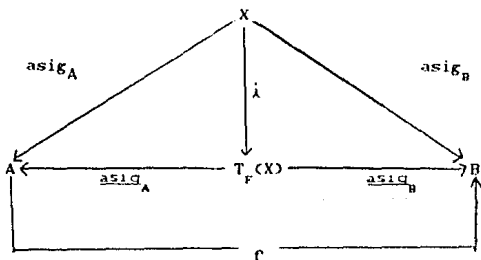
2) Existe solamente un homomorfismo $\text{eval}_A: T_F \longrightarrow A$, que es igual a la función de evaluación definida antes.

3) Si $f: A \longrightarrow B$ es un homomorfismo, entonces $\text{eval}_B = f \circ \text{eval}_A$ o equivalentemente, el siguiente diagrama es conmutativo.



4) Si $f: A \longrightarrow B$ es un homomorfismo, y $\text{asig}_A: X \longrightarrow A$ y $\text{asig}_B: X \longrightarrow B$ son asignaciones, entonces

SI $\text{asig}_B = f \circ \text{asig}_A$ entonces $\underline{\text{asig}}_B = f \circ \underline{\text{asig}}_A$



Demostración.

1). Sea $\text{asig}: X \longrightarrow A$ una función de asignación sobre las variables de la signatura, entonces definimos a la asignación extendida $\underline{\text{asig}}: T_F(X) \longrightarrow A$, de asig como sigue:

$$\text{1) } \underline{\text{asig}}(x) = \text{asig}(x) \quad \forall x \in X.$$

i) $\underline{asig}(k) = k_A$ para todo k constante.

ii) $\underline{asig}(G(t_1, \dots, t_n)) = G_A(\underline{asig}(t_1), \dots, \underline{asig}(t_n))$ para

todo término de la forma $G(t_1, \dots, t_n) \in T_F(X)$.

Observemos que, por la forma en que esta definida \underline{asig} es un homomorfismo, así pues existe el homomorfismo, y por el punto i) es claro que

$\forall x \in X, asig(x) = \underline{asig}(x) = \underline{asig}(i(x))$, donde $i: X \rightarrow X$ y $i(x) = x$
por lo tanto

$$asig = \underline{asig} \circ i.$$

Basta con demostrar que \underline{asig} es único.

Supongamos que existe un homomorfismo λ de $T_F(X)$ a A , tal que se cumple lo siguiente

$$asig = \lambda \circ i \text{ entonces probaremos que } \lambda = \underline{asig}.$$

La demostración se hará por inducción sobre los términos

i) $\underline{asig}(K) = K_A = \lambda(K)$ para todo símbolo de constante K , esto por definición de los homomorfismos.

ii) $\underline{asig}(x) = asig(x)$ como suponemos que λ es otra asignación extendida entonces se cumple que

$$asig = \lambda \circ i \text{ así pues } \underline{asig}(x) = \lambda(x) \quad \forall x \in X.$$

iii) Supongamos que $G(t_1, \dots, t_n) \in T_F(X)$ y que para todo $t_i \in T_F(X), i=1 \dots n$ se cumple que $\underline{asig}(t_i) = \lambda(t_i)$ entonces

$$\begin{aligned} \underline{asig}(G(t_1, \dots, t_n)) &= G_A(\underline{asig}(t_1), \dots, \underline{asig}(t_n)) \\ &= G_A(\lambda(t_1), \dots, \lambda(t_n)) \quad \text{por hipótesis} \\ &= \lambda(G(t_1, \dots, t_n)) \quad \text{pues } \lambda \text{ es un} \\ &\quad \text{homomorfismo.} \end{aligned}$$

$\therefore \underline{asig} = \lambda$ por lo tanto \underline{asig} es único.

2). Recordemos que T_F son los términos sin variables, así pues sea $X=0$ y sea la signación vacía $asig: X \rightarrow A$, la cual manda al vacío en el vacío, entonces por uno existe un solo homomorfismo \underline{asig} la cual extiende a $asig$ y tenemos lo siguiente

$\text{asig}: T_F(X) \longrightarrow A$ tal que cumple con las propiedades i), ii) y iii) de la parte uno, como $X \neq \emptyset$ entonces i) no cuenta y como para cada $t_i \in T_F(X) = T_F$, para todo $i=1 \dots n$, en el punto iii) tenemos la definición de eval_A definida en el capítulo 3.

3) Sea $f: A \longrightarrow B$ un homomorfismo, entonces por el punto dos tenemos que existe un homomorfismo único, eval_A , de T_F en A y también existe un único homomorfismo, eval_B , de T_F a B .

Sea la composición de homomorfismos $f \circ \text{eval}_A$, el cual por las propiedades de homomorfismos tenemos que $(f \circ \text{eval}_A)$ es a la vez un homomorfismo de T_F en B , pero como eval_B es único, entonces no queda otra opción más que $\text{eval}_B = f \circ \text{eval}_A$.

$$\therefore \text{eval}_B = f \circ \text{eval}_A$$

4) Sean $f: A \longrightarrow B$ un homomorfismo, asig_A y asig_B asignaciones para X en A y B respectivamente, y supongamos además que se cumple

$\text{asig}_B = f \circ \text{asig}_A$ tenemos las siguientes igualdades
 $\text{asig}_A = \text{asig}_A \circ i$ y $\text{asig}_B = \text{asig}_B \circ i$ entonces
 $\text{asig}_B = (f \circ \text{asig}_A) \circ i$ pero de la parte uno sabemos que el único homomorfismo con esta propiedad es asig_B .

$$\therefore \text{asig}_B = f \circ \text{asig}_A$$

Ya demostrado este teorema, pasamos a demostrar los incisos 2) y 3) del teorema correspondiente a las propiedades de los homomorfismos.

2) Esta parte pide demostrar que si $f: A \longrightarrow B$ es un homomorfismo, entonces cualquier ecuación, $e \in T_F$, que es válida en A también lo es en B .

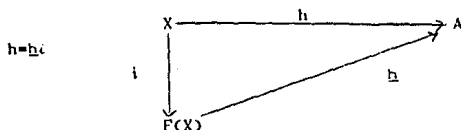
Demostración. Sea $e = (I, D) \in T_F$, una ecuación elemental, para demostrar que es válida debemos de probar que para toda asignación se cumple que la asignación extendida de I es igual a la asignación extendida de D . Como en este caso tenemos que $e \in T_F$.

4.4 CONCEPTO DE ALGEBRA INICIAL Y ALGEBRA LIBRE.

Definición. Sea \mathcal{C} una clase de SIG-álgebras, una SIG-álgebra \mathfrak{A} se dice que es **INICIAL** en \mathcal{C} si $\mathfrak{A} \in \mathcal{C}$ y para cada SIG-álgebra $A \in \mathcal{C}$ existe uno y solamente un SIG-homomorfismo $f: \mathfrak{A} \longrightarrow A$, a f lo llamaremos el **HOMOMORFISMO INICIAL** a A .

Otro concepto también muy importante es el del álgebra libre.

Def. Sea \mathcal{C} una clase de SIG-álgebras, X_s un conjunto de variables de la clase s , para $s \in S$, y $X = \bigcup_{s \in S} X_s$. Una SIG-álgebra $F(X)$ se dice que es **LIBRE SOBRE X** en \mathcal{C} si $F(X) \in \mathcal{C}$ y $X \subset F(X)$, sea $i: X \longrightarrow F(X)$ la función de inclusión, y para toda asignación $h: X \longrightarrow A$, con $A \in \mathcal{C}$, entonces existe solamente un homomorfismo \underline{h} tal que el siguiente diagrama conmuta, ie. $h = \underline{h} \circ i$



¿Qué trae de fondo el concepto de álgebra inicial y libre?

Para responder esta pregunta recordemos que, el principal objetivo del presente trabajo es presentar toda la herramienta matemática bajo la cual se fundamentan las especificaciones algebraicas.

Una de las principales tareas a resolver por las especificaciones algebraicas es poder hacer especificaciones que sean independientes de una representación en particular.

En sí, la idea primordial es que dada una caracterización de una álgebra, poder dar una interpretación de los operadores para que tengamos un único valor para cada expresión.

por lo tanto $h(1) = f(1)$. en forma análoga se hace para el paso inductivo.

Supongamos que se vale para $n-1$, demostraremos que es cierto para n .

$$\begin{aligned} h(n) &= h(\text{suc}(n-1)) = \text{suc}_A(h(n-1)) \\ &= \text{suc}_A(f(n-1)) \quad \text{hip. inductiva} \\ &= f(\text{suc}(n-1)) \\ &= f(n). \end{aligned}$$

Por lo tanto $h(n)=f(n) \quad \forall n \in \mathbb{N}$.

En forma análoga se puede demostrar para cuando $n < 0$. Cambiando suc por pred . en la demostración anterior, tendríamos la demostración.

Por lo que h es el único homomorfismo.

Como h es el único homomorfismo de \mathbb{Z} en A , esto indica que, \mathbb{Z} es inicial en $\text{Alg}(\text{grnt})$.

3.- Otro ejemplo más es que el álgebra de términos cocientes de una especificación es inicial y esto se demostrará en la siguiente sección.

4.5 INICIALIDAD DEL ALGEBRA DE TERMINOS COCIENTES.

El siguiente teorema declara que el álgebra T_{ESPEC} es inicial en $\text{Alg}(\text{ESPEC})$, y además nos muestra el homomorfismo inicial.

Teorema. Sea la especificación $\text{ESPEC}=(S,F,E)$, y sea T_{ESPEC} el álgebra de términos cocientes, definida en el capítulo anterior, entonces la siguiente proposición es verdadera:

T_{ESPEC} es inicial en $\text{Alg}(\text{ESPEC})$ y además el homomorfismo inicial $f: T_{\text{ESPEC}} \longrightarrow A$, para $A \in \text{Alg}(\text{ESPEC})$ está dado por

$$f(t) = \text{eval}_A(t)$$

Demostración. En un teorema anterior se probó que T_{ESPEC} es un ESPEC-álgebra. Así es que sea la función

$$f([t]) = \text{eval}_A(t)$$

vamos a demostrar que es una función bien definida, que es un homomorfismo y que este homomorfismo es único.

En la demostración suponemos que A es una ESPEC-álgebra.
 \circ Pd. f esta bien definida.

Sean $t, t' \in [t]$, entonces por definición de las clases de equivalencia tenemos que $\text{eval}_A(t) = \text{eval}_A(t')$, por lo que si $t, t' \in [t]$, entonces

$$f([t]) = \text{eval}_A(t) = \text{eval}_A(t') = f([t'])$$

por lo tanto

$$f([t]) = f([t'])$$

por lo tanto f está bien definida.

ω) f es un homomorfismo.

$$f: T_{\text{ESPEC}} \longrightarrow A$$

a) Sea N un símbolo de constante entonces

$$f([N]) = f([N]) = \text{eval}_A(N) = N_A$$

b) Sea $G; s_1 \dots s_n \longrightarrow s$ un símbolo de operación y t_i un término del tipo s_i , para $i=1, \dots, n$. Entonces

$$\begin{aligned} f([G([t_1], \dots, [t_n])]) &= f([G(t_1, \dots, t_n)]) \\ &= \text{eval}_A(G(t_1, \dots, t_n)) \\ &= G_A(\text{eval}_A(t_1), \dots, \text{eval}_A(t_n)) \\ &= G_A(f([t_1]), \dots, f([t_n])). \end{aligned}$$

Por lo tanto f es un homomorfismo.

$\omega\omega$) Basta probar que es único.

Para demostrar la unicidad utilizaremos inducción estructural.

Así que supongamos que existe otro homomorfismo

$$g: T_{\text{ESPEC}} \longrightarrow A.$$

que es TÍPICA si se cumple la siguiente condición:

Cualquier ecuación elemental e sobre SIG es válida en A si y solamente si es válida en toda $B \in \mathcal{C}$.

El siguiente teorema muestra la relación existente entre típica, generadora e inicial.

Teorema. Sea \mathcal{C} una clase de SIG-álgebras y $A \in \mathcal{C}$, entonces:

Ø A es inicial en \mathcal{C} implica que A es típica en \mathcal{C} .

ü Si A es generadora y típica en \mathcal{C} entonces A es inicial.

Pero a la inversa no se cumple.

Por el momento es todo lo que vamos a decir sobre estos términos.

4.7 COMENTARIOS SOBRE INICIALIDAD.

El término de inicialidad es muy estudiado por un grupo de gente dedicada al estudio de las especificaciones formales, el nombre del grupo es ADJ, y el concepto lo utilizan para hacer una teoría de especificaciones de tipos de datos. La importancia de este concepto radica en las siguientes razones.


Primero, el álgebra inicial como semántica de una especificación está caracterizada únicamente, salvo isomorfismo, por la simple propiedad de que existe un único homomorfismo para cada álgebra de la especificación, y además, este homomorfismo, se conoce explícitamente en todos los casos.

Segundo, el álgebra inicial como semántica de una especificación refleja los siguientes hechos:


Una especificación es una declaración concerniente a la existencia de datos, operaciones y propiedades.

Además es generadora, lo cual indica que no tenemos más elementos que los resultantes de aplicar las operaciones a los términos elementales, o que todos los datos existentes, provienen de un término.

Tercero, Por otra parte el álgebra de términos cocientes T_{SPEC} es un álgebra que permite una buena idea del significado de la especificación y, por su construcción, provee herramientas para discutir la semántica de una especificación.



ESPECIFICACIONES PARAMETRIZADAS.



CAPITULO 5. ESPECIFICACIONES PARAMETRIZADAS.

En ciencias de la computación nos encontramos con varios ejemplos de tipos de datos abstractos, por ejemplo las pilas, las colas, árboles, listas, etc., los cuales se aplican en una gran variedad de problemas.

Y dependiendo del problema en particular podemos tener, por ejemplo, pilas sobre cadenas de caracteres o tal vez pilas sobre los enteros o, lo más posible, pilas donde los elementos sean registros con varios campos de distintos tipos.

Es claro, que podemos hacer una especificación algebraica para las pilas sobre cadenas de caracteres o la especificación para las pilas sobre registros, pero, sería más óptimo tener una especificación para pilas, la cuál pueda estar parametrizada, tal que para todas las variantes de pilas pueda ser obtenida por actualización del parámetro. A este tipo de especificación la llamaremos *Especificación Parametrizada*.

Así pues, el principal objetivo del presente capítulo, es estudiar las especificaciones parametrizadas.

Comenzaremos por decir que entendemos por especificaciones parametrizadas y continuaremos con una pequeña introducción a la teoría de categorías, esto con el fin de dar una definición formal de la semántica de las especificaciones parametrizadas.

5.1 ESPECIFICACIONES PARAMETRIZADAS.

5.1.1 IMPORTANCIA DE LAS ESPECIFICACIONES PARAMETRIZADAS.

a Cuál es la importancia de las especificaciones parametrizadas ?.

Detrás del concepto de especificación parametrizada nos encontramos con dos conceptos que son de suma importancia para el desarrollo de sistemas computacionales, estos conceptos son el de *Modularidad* y *Reusabilidad*.

Modularidad: significa dividir una especificación muy grande en pequeñas partes, donde cada una de ellas este bien definida en

si misma, y además la estructura y la semántica de cada una de estas partes, contienen una subparte que está abierta.

Esta subparte es la especificación del parámetro formal el cual puede ser considerado como "una interfaz importante" de las especificaciones parametrizadas.

Reusabilidad: significa poder usar un módulo, ya definido, en la especificación de un sistema. Este módulo puede ser utilizado tantas veces como sea necesario, sin necesidad de volver a especificarlo.

Más tarde regresaremos a estas dos definiciones, que para el desarrollo de programas son muy importantes.

5.1.2 ESPECIFICACIONES PARAMETRIZADAS.

Las especificaciones parametrizadas las definimos como un par $ESPEC\bar{P}=(PAR,ESPEC)$, de especificaciones, donde, la especificación del parámetro formal PAR es una subespecificación de la especificación destino $ESPEC$.

Definición: Una **ESPECIFICACION PARAMETRIZADA** $ESPEC\bar{P}(PAR,ESPEC)$ consiste de un par de especificaciones
 $PAR = (S',F',E')$ llamada especificación del parámetro formal,
 y
 $ESPEC=PAR + (S,F,E)$ llamada especificación destino.

Notación, comentarios y más definiciones.

1.- Dada una especificación $ESPEC\bar{O}=(S_0,F_0,E_0)$ decimos que es una **Subespecificación** de $ESPEC=(S,F,E)$ si sucede que
 $S_0 \subseteq S$, $F_0 \subseteq F$ Y $E_0 \subseteq E$
 también nos podemos referir a (S_0,F_0) como una *subsignatura* de (S,F) .

2.- Supongamos que $SIG_0=(S_0,F_0)$ es una *subsignatura* de $SIG=(S,F)$ y sea

$A_1 = ((A_{\alpha})_{\alpha \in S}, (F_{\alpha})_{\alpha \in S})$ un álgebra de la *signatura* SIG .

Entonces diremos que el álgebra

$A_0 = ((A_{\alpha})_{\alpha \in S_0}, (F_{\alpha})_{\alpha \in S_0})$ es un *sub-álgebra* de A_1

3. Por definición, de la especificación parametrizada, ESPEC=(PAR,ESPEC). PAR es una subespecificación de ESPEC.

4. Para indicar que una especificación ESPEC es parametrizada, con el parámetro PAR, la escribiremos de la siguiente manera ESPEC(PAR). Donde de alguna manera estamos abusando de la notación, ya que ESPEC es la especificación destino y no exactamente el nombre de la especificación parametrizada.

Un ejemplo, es el tipo de dato abstracto Pilas, el cual fue definido en el capítulo correspondiente, a los tipos de datos abstractos. Aquí el parámetro formal es Elementos. Así que la notación para la especificación parametrizada Pilas, queda como sigue: Pilas(Elementos).

9.1.3 EJEMPLOS DE ESPECIFICACIONES PARAMETRIZADAS.

1.- Comenzamos definiendo la especificación para el parámetro formal y es la siguiente:

```
Elementos =  
    tipos = Elementos  
Fin Elementos.
```

La clase de todos los Elementos-álgebras consiste de todos los conjuntos. El álgebra inicial T_{Elementos} es el conjunto vacío, puesto que no tenemos símbolos constantes.

La especificación para pilas, con parámetro formal Elementos es idéntica a la especificada en el capítulo correspondiente a los tipos de datos abstractos.

```
Pilas(Elementos) = Elementos +  
tipos = Pila  
        Booleano  
oper =  
        cierto,falso;           -----> Booleano  
        crea_pila :             -----> Pila  
        mete      :Elemento, Pila -----> Pila  
        saca      :Pila          -----> Pila
```

```

mete      ;Elemento,Cola  —————> Cola
resto     ;Cola           —————> Cola
primero   ;Cola           —————> Elemento
es_vacia  ;Cola           —————> Booleano

```

var

```

e  ; Elemento
c  ; Cola

```

ecuac =

```

resto(creaCola) = error
resto(mete(e,c)) = Si es_vacia(c)
                    Entonces creaCola
                    Sino mete(e,resto(c))
primero(creaCola) = error
primero(mete(e,c)) = Si es_vacia(c)
                    Entonces e
                    Sino primero(c)
es_vacia(creaCola) = cierto
es_vacia(mete(e,c)) = falso

```

Fin Colas.

3. Un ejemplo más, es el correspondiente a la especificación de cadenas con parámetro formal especificado por Elementos.

Las operaciones sobre cadenas serán las siguientes:

Una operación que crea una cadena vacía.

Una operación que convierta un elemento en una cadena, compuesta por ese elemento.

Una operación que una cadenas.

Un par de operaciones que conviertan un elemento en una cadena y luego la pegue a otra cadena una por la parte izquierda y otra por la derecha.

Así que la especificación queda como sigue:

Cadenas(Elementos) = Elementos +

tipos =

Cadena

oper =

```
cad_vacia : _____ → Cadena
elem_a_cad: Elemento  _____ → Cadena
concatena: Cadena,Cadena _____ → Cadena
elem_cad_izq: Elemento,Cadena _____ → Cadena
elem_cad_der: Cadena,Elemento _____ → Cadena
```

var

```
e: Elemento
cad,cad1,cad2 : Cadena
```

ecuac =

```
concatena(cad,cad_vacia) = cad
concatena(cad_vacia,cad) = cad
concatena(cad,concatena(cad1,cad2)) =
    concatena(concatena(cad,cad1),cad2)
elem_cad_izq(e,cad) =
    concatena(elem_a_cad(e),cad)
elem_cad_der(cad,e) =
    concatena(cad,elem_a_cad(e))
```

Fin Cadenas.

En la siguiente sección estudiaremos un poco de teoría de categorías, la cual es una herramienta indispensable, para dar la semántica de las especificaciones parametrizadas.

5.2 CATEGORIAS Y FUNTORES.

En esta sección se darán los conceptos básicos de la teoría de categorías, como son la definición de Categoría y la de Funtores, además se proporcionan ejemplos. En la siguiente sección se enunciarán algunas de las propiedades de la teoría de categorías, que serán de gran utilidad, para definir la semántica de las especificaciones parametrizadas. Algunas de estas propiedades no se demostrarán, puesto que el objetivo de este trabajo, no es hacer un tratado formal de teoría de categorías, sino de la aplicación de los conceptos y algunas propiedades de esta teoría, en la definición formal de la semántica de las especificaciones algebraicas.

5.2.1 DEFINICION DE CATEGORIA.

Definición: Una CATEGORIA \mathcal{C} consiste de lo siguiente:

- i) Una clase de objetos $\text{Obj}_{\mathcal{C}}$. (Posiblemente conjuntos)
- ii) Dado cualquier par de objetos en \mathcal{C} , $A, B \in \text{Obj}_{\mathcal{C}}$, tenemos una colección de mapeos $f: A \rightarrow B$, llamados **MORFISMOS** de A a B, a esta colección la denotaremos por $\text{Mor}_{\mathcal{C}}(A, B)$.
- iii) Dados $f \in \text{Mor}_{\mathcal{C}}(A, B)$ y $g \in \text{Mor}_{\mathcal{C}}(B, C)$ tenemos una **COMPOSICION** de morfismos:
$$\cdot : \text{Mor}_{\mathcal{C}}(A, B) \times \text{Mor}_{\mathcal{C}}(B, C) \rightarrow \text{Mor}_{\mathcal{C}}(A, C).$$
además esta composición cumple con ser asociativa.
- iv) Para cada objeto A existe un morfismo $\text{id}_A: A \rightarrow A$, llamado **IDENTIDAD** de A. Tal que para todo $f \in \text{Mor}_{\mathcal{C}}(A, B)$ y $g \in \text{Mor}_{\mathcal{C}}(C, A)$, con $B, C \in \text{Obj}_{\mathcal{C}}$ tenemos que se cumple lo siguiente:

$$f \cdot \text{id}_A = f \quad \text{y} \quad \text{id}_A \cdot g = g$$

Antes de dar algunos ejemplos de categorías, primero vamos a definir lo que es un objeto inicial en las categorías y lo que es un isomorfismo.

Definición: Un morfismo $f: A \rightarrow B$ en una categoría \mathcal{C} es un **ISOMORFISMO** si existe un morfismo $g: B \rightarrow A$ en \mathcal{C} , tal que

$$g \cdot f = \text{id}_A \quad \vee \quad f \cdot g = \text{id}_B,$$

en este caso diremos que A y B son isomorfos.

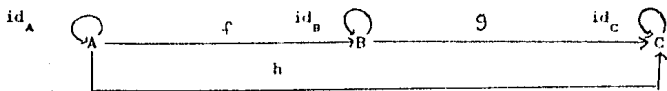
Definición: Se dice que un objeto I, en una categoría \mathcal{C} , es un **OBJETO INICIAL** en \mathcal{C} , si para cualquier otro objeto A de \mathcal{C} existe exactamente un morfismo $f: I \rightarrow A$.

Ejemplos de categorías.

1. Comenzamos con una categoría que hemos estudiado mucho, en nuestros cursos de cálculo, la cual consiste de la categoría de los conjuntos, a la cual denotaremos como \mathfrak{P} , y que consiste de todos los conjuntos como objetos y todas las funciones $f: A \rightarrow B$ como morfismos. La operación de composición es la usual entre funciones y la función $\text{id}_A: A \rightarrow A$ es la función identidad, claramente la composición y la identidad satisfacen los axiomas de asociatividad y los de la identidad. Los isomorfismos en \mathfrak{P} son las funciones biyectivas. El objeto inicial en \mathfrak{P} es el conjunto \emptyset puesto que la función vacía, $\emptyset: \emptyset \rightarrow A$, es la única función de \emptyset al conjunto A.

2. Un ejemplo más, es la categoría $\text{Cat}(\text{ESPEC})$ de las ESPEC-álgebras, que consiste de todas las ESPEC-álgebras como objetos y todos los ESPEC-homomorfismos $f: A \rightarrow B$ como morfismos de A a B. La composición es la composición de ESPEC-homomorfismos e $\text{id}_A: A \rightarrow A$ es el ESPEC-homomorfismo identidad, los cuales satisfacen los axiomas de la identidad y de composición. Los isomorfismos en $\text{Cat}(\text{ESPEC})$ son los ESPEC-homomorfismos biyectivos y el álgebra de términos cocientes T_{ESPEC} es un objeto inicial de $\text{Cat}(\text{ESPEC})$ esto es consecuencia de un teorema demostrado en el capítulo anterior.

3. La siguiente gráfica con nodos como objetos y aristas como morfismos define una categoría



5.2.3 PROPIEDADES DE LOS FUNTORES.

1) Cada funtor $F: \mathfrak{G}_1 \longrightarrow \mathfrak{G}_2$ preserva el isomorfismo, i.e., para cada isomorfismo $f: A \longrightarrow B$ también $F(f): F(A) \longrightarrow F(B)$ es un isomorfismo.

2) La composición $G \circ F: \mathfrak{G}_1 \longrightarrow \mathfrak{G}_3$ de funtores F y G es también un funtor.

Demostración.

1) Sea $f: A \longrightarrow B$ un isomorfismo en \mathfrak{G}_1 , entonces tenemos una $g: B \longrightarrow A$ tal que se cumple que $g \circ f = id_A$ y $f \circ g = id_B$, entonces por definición de funtores tenemos que:

$$F(g) \circ F(f) = F(g \circ f) = F(id_A) = id_{F(A)}$$

$$F(f) \circ F(g) = F(f \circ g) = F(id_B) = id_{F(B)}$$

lo cual prueba que $F(f)$ es un isomorfismo en \mathfrak{G}_2 .

2). Esta propiedad es trivial, por la definición de funtor.

5.2.4 EJEMPLOS DE FUNTORES.

1. Comenzamos con el funtor identidad $ID_{\mathfrak{G}}: \mathfrak{G} \longrightarrow \mathfrak{G}$ y está definido por $ID_{\mathfrak{G}}(A) = A$ y $ID_{\mathfrak{G}}(f) = f$, con $A \in \mathfrak{G}$ y f un morfismo de \mathfrak{G} .

2. Definición. Un MONOIDE es un conjunto C equipado con una función $\cdot: C \times C \longrightarrow C$ (la multiplicación del monoide) y un elemento distinguido e (el monoide identidad) sujeto a las siguientes dos leyes

$$X \cdot (Y \cdot Z) = (X \cdot Y) \cdot Z \quad \forall X, Y, Z \in C$$

$$X \cdot e = e \cdot X = X \quad \forall X \in C.$$

Y sean los mapeos $f: A \longrightarrow B$ con A y B monoides, que cumplen la propiedad $f(x \cdot x') = f(x) \cdot f(x')$, a los cuales les llamaremos homomorfismos entre monoides.

Ahora bien sea la categoría Mon que consiste de monoides como objetos y homomorfismos, entre monoides, como morfismos. Y sea la categoría \mathfrak{F} de conjuntos como objetos y funciones como morfismos. Entonces sea el funtor $U: \text{Mon} \longrightarrow \mathfrak{F}$ definido como sigue:

$$U: \text{Mon} \longrightarrow \mathfrak{F}$$

El cual envía al conjunto de elementos del monoide al mismo conjunto, en la categoría de los conjuntos y a cada morfismo entre monoides lo manda al mismo morfismo pero ahora interpretado como función en la categoría de los conjuntos. Observemos que este funtor se olvida de que los morfismos en los monoides cumplen ciertas propiedades, así como del elemento distinguido e .

3. De una forma similar podemos construir un funtor de la categoría de los grupos (la cual tiene como objetos a los grupos y como morfismos a los homomorfismos entre grupos) a la categoría de los conjuntos, de tal manera que al conjunto base de un grupo lo mande a él mismo, en los conjuntos, y a los homomorfismo entre categorías lo mande al mismo, sólo que ahora interpretado como una función entre conjuntos.

4. Sea la especificación parametrizada de Pilas(Elementos), con parámetro formal definido por la especificación dada en el primer ejemplo de éste capítulo. Entonces podemos construir un funtor que vaya de la categoría de los Pilas(Elementos), $\text{Cat}(\text{Pilas}(\text{Elementos}))$, a la categoría de Elementos, $\text{Cat}(\text{Elementos})$, definido de la siguiente forma

$$U: \text{Cat}(\text{Pilas}(\text{Elementos})) \longrightarrow \text{Cat}(\text{Elementos})$$

tal que para cada Pilas(Elementos)-álgebra,

$$P = (P_{\text{Elementos}}, P_{\text{Pilas}}, P_{\text{Boolean}}, \text{cierto}_p, \text{falso}_p,$$

$$\text{crea_pila}_p, \text{mete}_p, \text{saca}_p, \text{lope}_p, \text{es_vacía}_p)$$

la definimos como:

$$U(P) = P_{\text{Elementos}}$$

a cada Pilas(Elementos)-homomorfismo $f: P \longrightarrow P'$

$$U(f) = f_{\text{Elementos}}$$

o sea, f bajo U va a dar al homomorfismo $f_{\text{Elementos}}$ definido para el tipo Elementos. Recordemos que un homomorfismo f entre álgebras, está definido por una clase de homomorfismos.

ejemplo 4 el funtor se olvida de los tipos Boolean y Pilas, así como de las operaciones definidas para estos tipos.

A estos tipos de funtores los llamaremos olvidadizos.

A primera vista, el concepto de funtor olvidadizo nos parece trivial y completamente inútil. Sorpresivamente, vamos a ver que esta impresión es completamente falsa.

Por ejemplo, en la siguiente sección, exploraremos la posibilidad de asociar a cada funtor G otro funtor F , llamado adjunto izquierdo de G . Y veremos entonces que el adjunto izquierdo al funtor olvidadizo, $\mathbb{H}:\text{Cat}(\text{Grupos}) \longrightarrow \text{Cat}(\text{Conjuntos})$ nos enviará al grupo libre generado por un conjunto. Y este adjunto izquierdo de un funtor olvidadizo representará la semántica de una especificación parametrizada.

5.3 CONSTRUCCIONES LIBRES Y FUNTORES LIBRES.

En esta sección vamos a introducir los conceptos de construcciones libres y a partir de esta construcción tendremos un functor el cual llamaremos functor libre y este último representará la semántica de nuestras especificaciones parametrizadas.

Comenzaremos esta sección viendo algunos ejemplos, para darnos una idea intuitiva de las definiciones que más tarde formalizaremos.

5.3.1 EJEMPLOS DE OBJETOS LIBRES.

1. Comenzaremos construyendo un monoide libre. Sea X un conjunto, al cual llamaremos conjunto de generadores, y sea X^* el conjunto de todas las cadenas finitas de los elementos de X , incluyendo la cadena vacía λ , definimos una operación en X^* , llamada concatenación, que consiste en pegar dos cadenas de X^* una seguridad de la otra

$$(x_1 \dots x_n) \cdot (x'_1 \dots x'_m) = (x_1 \dots x_n x'_1 \dots x'_m)$$

claramente, esta operación es asociativa.

Además λ es la identidad, pues se cumple

$$\lambda \cdot w = w = w \cdot \lambda \quad \forall w \in X^*$$

A la terna (X^*, \cdot, λ) , la llamaremos monoide libre generado por X . Además este monoide libre cumple con la siguiente propiedad:

Dado un monoide (S, \cdot, e) y algún mapeo $f: X \rightarrow S$, visto como función entre los conjuntos X y S , entonces existe un único homomorfismo $\psi: (X^*, \cdot, \lambda) \rightarrow (S, \cdot, e)$ el cual extiende a f , i.e., que

$$\psi(x) = f(x) \quad \forall x \in X$$

$$\psi(x \cdot x') = \psi(x) \cdot \psi(x') \quad \forall x, x' \in X^*$$

$$\psi(\lambda) = e$$

es claro que esta ψ es única, y para que cumpla lo anterior basta que nosotros definamos a ψ como sigue:

$$\psi(\lambda) = e$$

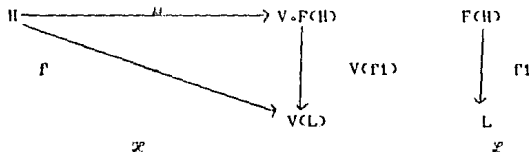
$$\psi(x_1 \dots x_n) = f(x_1) \cdot \dots \cdot f(x_n)$$

Y si además utilizamos un mapeo $\mu: X \rightarrow X^*$, que denota la inclusión de "generadores", es decir $\mu(x) = x$. Podemos representar lo anterior en una forma categorica. La cual veremos en seguida:

5.3.2 DEFINICION DE CONSTRUCCION LIBRE.

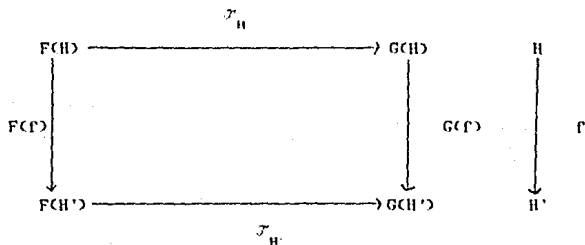
Definición. Dadas las categorías \mathcal{X} y \mathcal{Y} , un funtor $V: \mathcal{X} \rightarrow \mathcal{Y}$ y un objeto $H \in \mathcal{X}$, llamaremos a un objeto $F(H)$ en \mathcal{Y} la **construcción libre sobre H con respecto a V** , si existe un morfismo $\mu(H): H \rightarrow V \cdot F(H)$ en \mathcal{X} , llamado **morfismo universal** el cual satisface la siguiente propiedad:

Para cada objeto $L \in \mathcal{Y}$ y cada morfismo $f: H \rightarrow V(L)$ existe un único morfismo $f_1: F(H) \rightarrow L$ tal que $V(f_1) \cdot \mu(H) = f$, esto indica que el siguiente diagrama es conmutativo:



5.3.3 DEFINICION DE TRANSFORMACION E ISOMORFISMO NATURAL.

Definición. Sean $F, G: \mathcal{X} \rightarrow \mathcal{Y}$ funtores, una **transformación natural** $\mathcal{T}: F \rightarrow G$ es una regla que asigna a toda $H \in \mathcal{X}$ un morfismo $\mathcal{T}_H: F(H) \rightarrow G(H)$ en \mathcal{Y} tal que para toda $f: H \rightarrow H'$, con H y $H' \in \mathcal{X}$, el siguiente diagrama conmuta.



Definición. Una transformación natural $\mathcal{T}: F \rightarrow G$ es un **ISOMORFISMO NATURAL**, si \mathcal{T}_X es un isomorfismo en \mathcal{Y} , para todo objeto $X \in \mathcal{X}$.

5.3.4 DEFINICION Y PROPIEDADES DE LOS FUNTORES LIBRES.

1. Si una construcción libre $F(H)$ sobre H con respecto de V existe para alguna V y todo objeto H en \mathcal{X} entonces esta construcción libre puede ser extendida a un morfismo en \mathcal{X} tal que para cada morfismo $h: H \longrightarrow G$ en \mathcal{X} , $F(h)$ está únicamente determinado por la conmutatividad del siguiente diagrama.

$$\begin{array}{ccc}
 H & \xrightarrow{h} & G \\
 \mu(H) \downarrow & & \downarrow \mu(G) \\
 VF(H) & \xrightarrow{VF(h)} & VF(G)
 \end{array}$$

de esta forma obtenemos un functor $F: \mathcal{X} \longrightarrow \mathcal{X}$, el cual llamaremos **FUNTOR LIBRE** con respecto de V , y una transformación natural $\mu: ID_{\mathcal{X}} \longrightarrow V \circ F$, a la cual la llamaremos **TRANSFORMACION UNIVERSAL**.

2. Un functor libre F es único, salvo isomorfismo natural.
3. Los funtores libres preservan las álgebras iniciales, i.e., si I es un objeto inicial en \mathcal{X} entonces $F(I)$ es un objeto inicial en \mathcal{X} .

A continuación veamos un ejemplo de functor libre.

Ejemplo.

Como un ejemplo de construcción libre encontramos el dado en la sección 5.3.1 en el cual construimos el monoide libre a partir de un conjunto X , el conjunto generador del monoide. El functor libre en este caso es el siguiente:

$F: \text{Cat}(\text{Conjuntos}) \longrightarrow \text{Cat}(\text{Mon})$ el cual envía a cada conjunto A al monoide libre (A^*, \cdot, λ) .

operación $F(A)$ son dados por $F(A) = F + \text{Const}(A)$
 y la eval_A: $T_{F(A)} \rightarrow A$ es la evaluación de los términos
 elementales de $F(A)$ en A , donde A es considerada una
 $F(A)$ -álgebra.

3. Así pues, construimos una nueva especificación
 $\text{ESPEC}(A) = \text{ESPEC} + (0, \text{Const}(A), \text{Ecuac}(A))$

4. La ESPEC -álgebra

$T_{\text{ESPEC}}(A) = V_A(T_{\text{ESPEC}(A)})$ se le llama A -ALGEBRA DE
 TÉRMINOS COCIENTES, donde $T_{\text{ESPEC}(A)}$ es el álgebra de
 términos cocientes para $\text{ESPEC}(A)$ y V_A es el funtor
 olvidadizo de las $\text{ESPEC}(A)$ -álgebras a las ESPEC -álgebras.

En seguida veremos una propiedad importante de esta A -álgebra
 de términos cocientes.

Teorema. Dada una especificación parametrizada
 $\text{ESPEC} = (\text{PAR}, \text{ESPEC})$, con funtor olvidadizo

$$V: \text{Cat}(\text{ESPEC}) \rightarrow \text{Cat}(\text{PAR}) \text{ entonces}$$

para cada PAR -álgebra A el A -álgebra de términos cocientes

$$F(A) = T_{\text{ESPEC}}(A) \text{ es}$$

una construcción libre sobre A con respecto a V . Y ésta
 construcción libre se extiende a un funtor libre F

$$F: \text{Cat}(\text{PAR}) \rightarrow \text{Cat}(\text{ESPEC})$$

Veamos un ejemplo de una A -álgebra de términos cocientes.

Ejemplo.

Consideremos la especificación parametrizada
Colas(Elementos), la que definimos en la sección 5.1.3 en el
 ejemplo 2; y sea Alfabeto un Elementos-álgebra con
 $\text{Alfabeto} = \{a, b, \dots, z\}$ i.e., el alfabeto para las letras minúsculas,
 entonces para construir el Alfabeto-álgebra de términos cocientes
 tenemos los siguientes conjuntos:

especificaciones. Y a partir de ésta álgebra de términos cocientes podemos definir un funtor que es libre. Así que pasemos a ver lo que entenderemos por la semántica de una especificación parametrizada.

Definición. La SEMANTICA de una especificación parametrizada ESPECP=(PAR,ESPEC) con funtor olvidadizo

$$V: \text{Cat}(\text{ESPEC}) \longrightarrow \text{Cat}(\text{PAR}) \text{ es la clase}$$

$$\text{TDAP}(\text{ESPECP}) = \{ F: \text{Cat}(\text{PAR}) \longrightarrow \text{Cat}(\text{ESPEC}) \mid F \text{ es un funtor libre con respecto de } V \}$$

de todos los funtores libres con respecto de V , los cuales son llamados TIPOS DE DATOS ABSTRACTOS PARAMETRIZADOS definidos por ESPECP. Por la propiedad de unicidad, salvo isomorfismo natural, podemos decir que la semántica de ESPEC es cualquier funtor libre $F \in \text{TDAP}(\text{ESPECP})$.

EJEMPLOS:

1. Tomemos la especificación parametrizada Cadenas(Elementos) definida en la sección 5.1.3, ejemplo 3, entonces podemos construir un funtor

$$\text{STR} : \text{Cat}(\text{Elementos}) \longrightarrow \text{Cat}(\text{Cadenas}(\text{Elementos}))$$

el cual en seguida se da:

Para cada Elementos-álgebra A , la Cadenas(Elementos)-álgebra $\text{STR}(A)$ va a estar definida por

$$\text{STR}(A) = (A.A^*, \text{vacía}, \text{elemcad}, \text{concatena}, \text{añadeizq}, \text{añadeder})$$

donde A^* es el monoide libre sobre A y las operaciones son las siguientes:

"vacía" crea una nueva cadena vacía;

"elemcad" es la operación que convierte un elemento de A en una cadena que consiste de ese elemento.

"concatena" es la operación de concatenación entre cadenas.

"añadeizq" es la operación de añadir un elemento a una cadena

por la izquierda.

"añadeder" es análoga a la anterior sólo que el elemento se anexa por la derecha.

Estas operaciones las podemos definir formalmente como sigue:

Sean $a \in A$ y $u, v \in A^*$

vacía:	$\longrightarrow A^*$	
elemcad:	$A \longrightarrow A^*$: elemcad(a)=a
concatena:	$A^* A^* \longrightarrow A^*$: concatena(u,v)=uv
añadeizq:	$A A^* \longrightarrow A^*$: añadeizq(a,u)=au
añadeder:	$A^* A \longrightarrow A^*$: añadeder(u,a)=ua

Ahora bien ¿Cómo vamos a definir $STR(f)$, con f un morfismo en la $Cat(\underline{Elementos})$?

Para cada Elementos-morfismos $f: A \longrightarrow B$, tenemos que

$STR(f)=f$; con $f: STR(A) \longrightarrow STR(B)$ donde

$f: A^* \longrightarrow B^*$, por definición de STR , es la extensión de f definida por

$$\begin{aligned} f(a_1 \dots a_n) &= f(a_1) \dots f(a_n) & \forall a_1, \dots, a_n \in A^* \text{ y} \\ f(\text{vacía}) &= \text{vacía}_B \end{aligned}$$

El objetivo ahora es probar que el funtor STR es la semántica de la especificación parametrizada $Cadenas(\underline{Elementos})$. Así pues lo que debemos de ver es que STR es un funtor libre con respecto del funtor olvidadizo $V: Cat(Cadenas(\underline{Elementos})) \longrightarrow Cat(\underline{Elementos})$

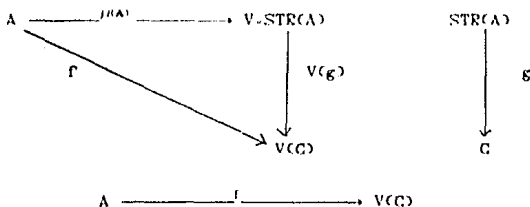
Para demostrar que STR es un funtor libre, lo que debemos de ver es que $STR(A)$, para todo Elementos-álgebra A es una construcción libre, con respecto de V , y probando esto por la propiedad 1 de la sección 5.3.4 tenemos que STR es un funtor libre.

Así pues demostremos que $STR(A)$ es una construcción libre, para todo objeto A de la categoría de Elementos.

Demostración.

Sea A un conjunto, tomemos el morfismo universal $\mu(A)$ que es igual a la identidad para A , $\text{id}_A: A \longrightarrow V\text{-STR}(A)$, con V el functor olvidadizo correspondiente, entonces para cada Cadenas(Elementos)-álgebra C y cada función $f: A \longrightarrow V(C)$

P.d. existe un único Cadenas(Elementos)-homomorfismo $g: \text{STR}(A) \longrightarrow C$ tal que el siguiente diagrama es conmutativo



Para tener esta conmutatividad definamos a g como sigue

$$g(a) = f(a) \quad \forall a \in A$$

Y para que g sea un homomorfismo, continuamos su definición como sigue:

$$g(a) = f(a) \quad \forall a \in A$$

$$g(\text{vacío}) = \text{cad_vacío}_C$$

Supongamos que $u \in A^*$ tal que $u = a_1 \dots a_n$ entonces


$$g(u) = g(a_1 \dots a_n)$$

$$g(a_1 \dots a_n) = \text{elem_cad}_C(f(a_1)) \cdot_C \dots \cdot_C \text{elem_cad}_C(f(a_n))$$


donde \cdot_C es la operación de concatenación en C .

Demostremos ahora que g es un Cadenas(Elementos)-homomorfismo

Es claro, que por definición de g , ésta cumple con la propiedad de ser homomorfismo para la constante "vacío", y para las operaciones "elem_cad" y "concatenación", así pues basta probar para las operaciones "añadeizq" y "añadeder". Pero sea $a \in A$ y $a_1 \dots a_n \in A^*$ entonces



INTRODUCCION.
AL
LENGUAJE ACT - ONE.



-Y permite el uso de bibliotecas.

6.2 DESCRIPCION DEL LENGUAJE.

ACT-ONE está basado, principalmente, en 5 conceptos fundamentales para el desarrollo de las especificaciones, y a través de éstos, podemos facilmente realizar especificaciones algebraicas. Estos conceptos son;

Especificaciones básicas: Donde podemos escribir especificaciones, especificaciones parametrizadas y especificaciones de parámetros formales.

Combinación: Lo que nos indica este concepto, es que podemos tomar varias especificaciones, ya hechas y juntarlas, para crear una nueva especificación.

Renombrado: En ocasiones vamos a desear especificaciones que posiblemente ya hayan sido definidas, con anterioridad, sólo que ahora nos convenga que tengan otro nombre los conjuntos de los tipos o los nombres de los símbolos de operación, y por medio de éste concepto vamos a poder dar nuevos nombres a los objetos de una especificación.

De alguna manera, con el renombramiento vamos a poder tener especificaciones reutilizables.

Actualización: Con este concepto va ha ser posible actualizar el parámetro formal en una especificación parametrizable.

Modularización: Por lo general cuando creamos un sistema de cómputo, no lo implementamos de una sola parte, si no que más bien, dividimos el sistema en varias partes funcionales y por último creamos un sólo módulo que nos mande llamar a los demás.

En este último capítulo, veremos la primera parte, la correspondiente a las especificaciones básicas. Si se desea ampliar un poco más los conocimientos en este lenguaje se recomienda el libro de H. Ehrig y B. Mahr [85].

6.3 ESPECIFICACIONES BASICAS.

Para especificar un sistema, en ACT-ONE, se comienza por definir el nombre del sistema precedido de la palabra def y después del nombre sigue la palabra reservada is, que nos indica que aquí comienza la definición del sistema, a continuación viene el cuerpo de la especificación y se finaliza con las palabras reservadas end of def.

Una especificación básica en ACT-ONE está dividada en 4 secciones principales:

- 1) Un encabezado, en donde se indica el nombre del sistema;
- 2) Una sección de declaración de tipos. (sorts)
- 3) Otra para la declaración de funciones. (opns)
- 4) Y una sección más para la definición de las ecuaciones y de las variables utilizadas.

Un esquema para las especificaciones básicas es el siguiente:

```

def NombreEspec is
    sorts listatipos
    opns listafuncs
    eqns listaecuas
end of def.

```

Como un ejemplo, veamos la especificación de grupo.

```

def Grupo is
    sorts Conjunto
    opns
        e :                → Conjunto
        * : Conjunto Conjunto → Conjunto

```

Aquí estamos dando, explícitamente la especificación del parámetro formal, pero no es la única forma sino que podemos tener una especificación para un parámetro formal utilizando el siguiente esquema:

```
def NombreEspecFormal is  
  formal sorts listatipos  
  formal opns listafuncs  
  formal eqns listaecuas  
end of def.
```

y hacer la combinación de la especificación del parámetro formal y la especificación parametrizada.

Observemos que una signatura $\Sigma = (S, F)$ estará representada por las secciones correspondientes a sorts y opns de donde las listas de tipos y funciones son los elementos de los conjuntos S y F.

A continuación vamos a dar algunos ejemplos de especificaciones en ACT-ONE, utilizando los esquemas anteriores.

6.4 EJEMPLOS DE ESPECIFICACIONES EN ACT-ONE.

1. Comenzamos haciendo la especificación del parámetro formal Elementos, que hemos utilizado en las especificaciones parametrizadas.

```
def Elementos is  
  formal sorts Elemento  
end of def.
```

de donde la semántica para esta especificación es la clase de todos los conjuntos, lo que llamamos semántica débil.

2. Ahora pasemos a definir la especificación parametrizada de un árbol binario.

A los árboles vamos a definirlos como un tipo de dato abstracto, que sirven para guardar y organizar datos. Por lo general un árbol es utilizado para la búsqueda y ordenamiento de datos. Los datos que se guardan en los árboles pueden ser de diferentes tipos, un árbol para cada tipo de elementos, por lo que

```

        Nodo(NodoNulo) = error
        Nodo(CreaNodo(ai,e,ad)) = e
    of sort Boolean
    for all ai,ad in Arbol;
        e in Elemento
        EsNulo(NodoNulo) = cierto
        EsNulo(CreaNodo(ai,e,ad)) = falso
end of def

```

De donde la semántica de la especificación **ArbolesBin(Elementos)** estará dada por el funtor libre, que manda a cada **Elementos**-álgebra **A** a

$(A^*, A, \text{Boolean}, \text{nodonulo}, \text{creanodo}, \text{hojaizq}, \text{hojader}, \text{nodo}, \text{esnulo})$

3. Para finalizar este capítulo, veremos un ejemplo más, el cual consiste en hacer la especificación de un analizador léxico.

En la construcción de un compilador, para un lenguaje de alto nivel, suele describirse en función de una *gramática*, que especifica la forma, o sintaxis, de las proposiciones legales del lenguaje. Por ejemplo la gramática puede definir una proposición de asignación como un nombre de variable seguido de un operador de asignación (=) y de una expresión.

Así, el problema de la compilación consiste en comparar las proposiciones escritas por el programador con las estructuras definidas por la gramática, y en generar el código objeto apropiado para cada proposición.

Es conveniente considerar a una proposición de programa fuente como una secuencia de *componentes léxicos (tokens)*.

Los componentes léxicos pueden suponerse como los fundamentos del lenguaje.

Por ejemplo, un componente léxico podría ser una palabra reservada, un nombre de variable, un entero, etc.

La tarea de examinar la proposición fuente con el fin de reconocer y clasificar los distintos componentes léxicos se conoce

CONSTRUCCION DE LA ESPECIFICACION FORMAL DE UN ANALIZADOR LEXICO

Para la creación de la especificación del analizador léxico, utilizaremos el concepto de modularidad, por lo que vamos a dividir la especificación del sistema, en varias especificaciones.

La forma en como vamos a dividir el analizador, es en crear varios tipos de datos abstractos, los cuales consisten del tipo de datos "cadenas", el cual utilizaremos para construir la cadena de los códigos, correspondientes a los componentes léxicos; el de "tablas de símbolos", un módulo para la especificación del parámetro formal llamado texto, que representará a las cadenas de caracteres de que está compuesto el archivo a analizar; otro módulo, para la especificación de los archivos, y por último un módulo más, el correspondiente al cuerpo del analizador.

Comenzaremos con la especificación del parámetro texto, esta especificación contendrá una única función, llamada iguales, la cual nos verificará si son idénticos dos cadenas o no.

```
def Textos is
  formal sort:
    Texto
    Boolean
  formal opns:
    verdadero,falso; --- Boolean
    iguales; Texto Texto --- Boolean
  formal eqns:
    of sort Boolean
    for all cad in Texto
      iguales(cad,cad)=verdadero.
end def
```

Como se había dicho antes, la salida del analizador consiste de una secuencia de los códigos, correspondientes a cada uno de los componentes léxicos del programa fuente, así pues para la construcción de esta lista utilizaremos el tipo de dato abstracto cadenas.

La especificación para el tipo cadenas es análoga a la que se vio en el capítulo anterior, y escrita en lenguaje ACT-ONE queda como sigue:

```

def Cadenas(Elementos) is
    Elementos
    formal sort
        Elemento
    sorts
        Cadena
    ops
        cadvacía: → Cadena
        elemcad: Elemento → Cadena
        concatena: Cadena Cadena → Cadena
        añadeizq: Elemento Cadena → Cadena
        añadeder: Cadena Elemento → Cadena
    eqns
        of sort Cadena
        for all cad,cad1,cad2 in Cadena
            e:Elemento
                concatena(cadvacía,cad)=cad
                concatena(cad,cadvacía)=cad
                concatena(cad,concatena(cad1,cad2))=
                    concatena(concatena(cad,cad1),cad2)
                añadeizq(e,cad)=concatena(elemcad(e),cad)
                añadeder(cad,e)=concatena(cad,elemcad(e))
    end of def

```

Para reconocer el código de un identificador o de una constante numérica, utilizaremos, las tablas de símbolos, de donde si el componente léxico que estamos viendo va se encuentra en la tabla, mandaremos escribir su código y en caso de no encontrarse se añade a la tabla de símbolos con su respectivo código y se retorna el código.

Una tabla de símbolos estará compuesta por dos campos, uno que indicará el nombre del componente léxico y el otro que representará su código.


```

entabla(cad,inserta(cad1,cod,tab)) =
    si_ent_sino_bol(iguales(cad,cad1),
        verdadero,entabla(cad,tab))
si_ent_sino_bol(verdadero,b1,b2) = b1
si_ent_sino_bol(falso,b1,b2) = b2

```

of sort Código

```

for all cad,cad1 in Texto;
    tab in Tabla;
    c1,c2,cod in Código;

```

```

códigotok(cad,creatab) = error
códigotok(cad,inserta(cad1,cod,tab))=
    si_ent_sino_cod(iguales(cad,cad1),
        cod,códigotok(cad,tab) )
si_ent_sino_cod(verdadero,c1,c2) = c1
si_ent_sino_cod(falso,c1,c2) = c2

```

end of def

Observemos las dos funciones, que no hablamos de ellas en la especificación informal, éstas son

`si_ent_sino_bol` y `si_ent_sino_cod`; que se pueden interpretar como proposiciones condicionales, donde la primera retorna un booleano y la segunda es del tipo código, es la razón de las terminaciones `bol` y `cod` en el nombre de las funciones.

De alguna forma ésta es una de las principales desventajas del lenguaje ACT-ONE, pues esta proposición no se encuentra dentro de la gramática del lenguaje, y es por eso que el diseñador de la especificación la escribe explícitamente.

Recordemos que el objetivo del *análisis* es analizar el archivo fuente que contiene al programa, así pues necesitamos de un módulo que nos maneje operaciones con archivos. Así que definiremos un tipo de dato abstracto llamado `archivo` el cual estará parametrizado por el tipo `texto`.

Las operaciones que tenemos para transformar un archivo son las siguientes:

archvacio: nos creará un nuevo archivo.
escribe: con esta función escribiremos en el archivo una cierta cadena de caracteres. Un *token*.
lee: con esta función vamos a poder leer el siguiente elemento del archivo en el cual estamos posicionados.
esvacio: esta función nos regresará cierto si el archivo no contiene texto, o falso de lo contrario.
resto: que tiene como fin borrar el componente léxico en el cual nos encontramos.

La especificación formal en lenguaje ACT-ONE, queda como sigue:

```

def Archivos(Textos) is
    Textos
    sorts
        Archivo
    opns
        archvacio;  —→ Archivo
        escribe; Texto Archivo —→ Archivo
        lee; Archivo —→ Texto
        resto; Archivo —→ Archivo
        esvacio; Archivo —→ Boolean
        si_ent_sino_arc; Boolean Archivo Archivo
                                —→ Archivo
        si_ent_sino_lex; Boolean Texto Texto
                                —→ Texto
    eqns
        of sort Archivo
        for all arc in Archivo;
            cad in Texto

            resto(archvacio)=error
            resto(escribe(cad,arc))=
                si_ent_sino_arc(esvacio(arc),
                    archvacio,arc)

```

```
of sort Texto
for all arc in Archivo:
    cad in Texto
```

```
lee(archvacio)=FDA
lee(escrbe(cad,arc))=
    si_ent_sino_tex(esvacio(arc),cad,
                    lee(arc))
```

```
of sort Boolean
for all arc in Archivo
    cad in Cadena
```

```
esvacio(archvacio)=verdadero
esvacio(escrbe(cad,arc)) = falso
```

```
end of def
```

La constante utilizada FDA corresponde al fin de archivo, esta constante no es necesariamente de tipo Texto, sino que mas bien es una marca lógica, que se deja en un dispositivo externo, como podría ser un disco, o cinta magnetica, y esta marca la coloca el sistema operativo de la computadora que estemos utilizando para la implementación del sistema. Así que supondremos que FDA es una constante sin tipo.

Para finalizar la especificación del analizador léxico, se escribirá primero una especificación informal sobre las funciones que utilizaremos y su respectiva tarea.

- analiza:** esta función recibe un archivo fuente, y retorna una lista de los códigos, correspondientes a los componentes léxicos del programa.
- retornacod:** esta función retornará el código correspondiente a un *token* especificado.
- essimbesp:** regresará cierto si un *token* especificado es un símbolo especial del lenguaje.

espales: regresará cierto si un *token* especificado es una palabra reservada del lenguaje.

esnum: regresará cierto si un *token* especificado es un número.

esident: regresará cierto si un *token* especificado es un identificador válido del lenguaje.

def Analizador is

Archivos(Textos) and Tablas(Textos) and
Cadenas(Codigos)

opns

analiza; Archivo → Cadena
retornacod; Texto → Código
essimbep; Texto → Boolean
espales; Texto → Boolean
esnum; Texto → Boolean
esident; Texto → Boolean
si_ent_sino_cod ; Boolean Código Código → Código
si_ent_ ; Boolean Código → Código

eqns

of sort Cadena
for all arch in Archivo;
cad in Texto

analiza(archvacfo)=FDA
analiza(arch) =
concatena(retornacod(lee(arch)),
analiza(resto(arch)))
of sort Código
for all arch in Archivo;
cad in Texto;
c1,c2,cod in Código;
labimbep,labpalres,labident,
labconst in Tabla

```

retornacod(cad)=
    si_ent_cod(  essimbesp(cad),
                códigotok(cad,tabsimbesp) )
retornacod(cad)=
    si_ent_cod(  espalres(cad),
                códigotok(cad,tabpalres) )
retornacod(cad)=
    si_ent_cod(  esnumero(cad),
                si_ent_sino_cod(entabla(cad,tabconst),
                códigotok(cad,tabconst),
                códigotok(cad,inserta(cad,cod,tabconst)))
retornacod(cad)=
    si_ent_cod(  esident(cad),
                si_ent_sino_cod(entabla(cad,tabident),
                códigotok(cad,tabident),
                códigotok(cad,inserta(cad,cod,tabident)))

si_ent_cod(verdadero,cod)=cod
si_ent_cod(falso,cod)=vacío

```

end def

En esta última especificación utilizamos una palabra reservada que no explicamos su función, esta es and y lo que nos indica es que vamos a combinar las especificaciones para crear una nueva; por medio de la combinación podemos utilizar los tipos y funciones definidas en las otras especificaciones, para definir las propiedades en la especificación que estamos construyendo.

En varios de los módulos especificados aquí, nos encontramos que algunas funciones regresan error, este valor no necesariamente tiene que tener un tipo bien definido, sino que mas bien no tiene tipo, y lo que representa es que la evaluación que estamos realizando es invalida. De ahí que no se defina como una constante con su respectivo dominio.



CONCLUSIONES.



CONCLUSIONES.

En la introducción de la tesis, se mencionó que el objetivo del presente trabajo es realizar una visión de los fundamentos matemáticos en los cuales se basan las especificaciones algebraicas.

En el primer capítulo vimos la necesidad de tener métodos, de tal forma que sean suficientemente confiables en la creación de sistemas de cómputo, utilizando especificaciones formales.

En los siguientes cuatro capítulos se presentaron los fundamentos bajo los cuales, podemos decir que las especificaciones algebraicas son modelos para los sistemas de tipo algebraico.


Vimos de que forma, conceptos tan importantes como objetos iniciales o los homomorfismos entre álgebras, las podemos utilizar para explicar formalmente ideas sobre la independencia de la representación y validación de un sistema.

Por otra parte, el uso de las especificaciones algebraicas provee las definiciones abstractas de las operaciones usadas para establecer la comunicación entre los distintos módulos de un sistema de cómputo.


Actualmente, gente dedicada al estudio de las especificaciones formales, esperan que las especificaciones de tipos de datos abstractos tengan un impacto substancial, para la demostración de las propiedades sobre los programas y su verificación.

Después de la experiencia propia, en las especificación de tipos de datos abstractos y de algunos sistemas computacionales, puedo decir, que las especificaciones formales no pueden reemplazar a las informales, sino más bien pueden ser complementarias, mientras que las informales son fáciles de leer y de entender, las formales tienden a ser claras, precisas y no ambiguas.

Por otra parte, algunas de las principales desventajas de las especificaciones algebraicas, es que no ofrecen características sobre los requerimientos de la memoria que se solicita para un sistema de cómputo, o los requerimientos de entrada y salida de datos, ni sobre otras propiedades que son importantes, como por ejemplo, el problema que se encontró a la hora de especificar el módulo de archivos, con la constante de fin de archivo, que no tiene un tipo bien definido.



APENDICE.




A P E N D I C E A

SINTAXIS DEL LENGUAJE ACT-ONE


`<act text> ::= act text <def>* [uses from library listnoml end text
<def> ::= def nombre is pexpr end of def
<pspec> ::= [formal sorts listatipos]
 [formal opns listaoper]
 [formal eqns listaecuac]
 [sorts listatipil]
 [opns listaoperil]
 [eqns listaecuacil]
<pexpr> ::= [<nombreexpr>]<pspec>
<pexpr> ::= nombre renamed by <repl>
<nombreexpr> ::= [<nombreexpr> and] nombre
<repl> ::= [sortnames sortpairlist]
 [opnames oopairlist]`

Donde los símbolos no terminales tienen el siguiente significado:

- <act text> : Texto de definiciones y nombre para la librería.
- <pspec> : Presentación explícita de una especificación o parte de ésta.
- <pexpr> : Expresión para una especificación parametrizada.
- <nombreexpr> : Nombre de expresión para una combinación de especificaciones parametrizadas.
- <repl> : Lista de reemplazamiento para nombres de tipos y operaciones.



BIBLIOGRAFIA.



Roger S. Pressman.

Ingeniería del Software un enfoque práctico.

2a. edición, Ed. McGrawHill.

Hanna Oktaba, Laura Espitia

Guadalupe Ibaranguoitia, Carlos Velarde.

Especificación Formal en el Diseño de Programas.

Serie Azul; Monografías No. 107

IIMAS, 1988.

Balzer R. y N. Goodman.

Principles of good software specification.

Proc. on specifications of reliable software, IEEE.

1979.

Narain Gehany.

Specifications: Formal and Informal: A case Study.

Software-Practice and Experience Vol. 12, 1982.

John D. Lipson.

Elements of Algebra and Algebra Computing.

Edit. Addison-Wesley.

David E. Rydeheard y Rod M. Burstall.

Computational Category Theory.

Ed. Prentice Hall., 1988.