Tipos abstractos de datos

“Conjunto de instancias/valores que tienen operaciones”.

La palabra tipo no es inocente. Identifica la colección de “cosas” que comparten el comportamiento descripto.

¿Qué es un TAD?

- Desde el punto de vista formal: es una herramienta matemática. Eso es bueno porque nos da la rigurosidad y formalidad que necesitamos para entender claramente qué es lo que hay que hacer, es decir, para describir el problema que se pretende resolver.
- Desde el punto de vista práctico: es una herramienta poderosa y flexible que como veremos más adelante nos va a permitir describir una enorme cantidad de problemas.
- Desde el punto de vista histórico: es uno de los primeros intentos por abordar la problemática de describir formalmente el comportamiento de un artefacto de software.

Partes de un TAD

- Genero: Nombres de los tipos que un TAD define. Tipos que estamos especificando.
- Exporta: Detalla que operaciones y géneros se dejan a disposición de los usuarios del tipo (para usar en otros TADs). Por defecto: todo.
- Generadores: Signatura de las funciones que se usan para definir las instancias. Nos permiten construir todos los terminos de interes del tipo.

  Una instancia es un termino cuyos unicos simbolos de funcion son los generadores.
- Observadores: Son aquellas operaciones que nos permiten, utilizadas en conjunto, distinguir si dos instancias del tipo son iguales o no. (funciones que se usan para definir la igualdad observacional).
- Igualdad observacional: Identifican cuándo dos instancias de un tipo son iguales (o diferentes), y por ende exhiben todo el comportamiento relevante de estos objetos del mundo real.

  Sentencia de LPO que representa una relacion de equivalencia definida sobre el conjunto de terminos.
- Otras operaciones: Signatura de funciones que no son generadores ni observadores).
- Axiomas: Son las reglas que definen matemáticamente el comportamiento de las operaciones identificadas. Sentencias que definen el significado de los simbolos de funcion.

  Requisitos para que la axiomatizacion sea razonable:

  - El conjunto de axiomas es completo para el conjunto de términos sobre los que aplica. Es suficiente con proveer un axioma por cada generador.
  - No hay dos axiomas que apliquen y den resultados diferentes. Si para cada término construible solo un axioma de la definición aplica, entonces no es posible obtener valores diferentes.

- Las recursiones terminan. Si se tienen bien definidos los casos bases y en todo axioma recursivo el término de la derecha es de complejidad menor que el de la izquierda, entonces no puede existir una recursión infinita.

- Usa: TADs que utiliza.
- Parametros formales: Requerimientos sobre los parametros de tipo (genero, existencia de simbolos de funcion, etc)

Nota final: la funcion permutacion que recibe un conjunto y devuelve secuencia es correcta

Como debe ser la axiomatizacion

- $f(t_1^1, ..., t_k^1) \equiv R$
- Ax1: $f(t_1^1, ..., t_i^1, ..., t_k^1) \equiv \ldots$

  Ax2: $f(t_1^2, ..., t_i^2, ..., t_k^2) \equiv \ldots$

  …

  Axh: $f(t_1^h, ..., t_i^h, ..., t_k^h) \equiv \ldots$

  -No ∀, ∃.
  -Variables ≠ y ≤1 generador.
  -Si en una posicion de la “matriz” hay un generador, en toda la columna hay generadores y deben ser todos distintos.

  t_i^j : Variable o generador del TAD X

Como especificar un TAD?

1. Leer bien el enunciado e identificar qué cosas son importantes y qué cosas no lo son.
2. Definir los observadores y la igualdad observacional.
3. Definir los generadores.
4. Definir las otras operaciones.
5. Definir las restricciones donde corresponda
6. Incluir otras operaciones auxiliares, de haberlas.
7. Axiomatizar todo.

Recursion

Una funcion recursiva es una funcion que se llama a si misma y termina en (y tiene al menos un) caso base y cada llamado recursivo nos acerca un poco a este.

Funciona (por induccion) si:

- Existen uno o mas casos base
- Cada llamado recursivo simplifica el problema (y alcanza un caso base)
- El llamado recursivo funciona bien

Minimalidad

- El conjunto de observadores y debe ser minimal.

- Es deseable que el conjunto de generadores sea minimal.

De lo contrario, la especificación se torna más difícil, menos clara y más propensa a errores.

Comportamiento automatico

- ¿Qué impacto tiene el comportamiento automático en la elección de los observadores?
    - Ninguno: Para distinguir lo que diferenciaba a las instancias no importa si el comportamiento es automático o no.
- ¿Qué impacto tiene el comportamiento automático en los generadores?
- Algo: Puede ser que no necesitemos algun generador que haga una accion (ya que eso es parte del comportamiento automático).
- ¿Qué impacto tiene el comportamiento automático en los axiomas?
- Mucho: El comportamiento queda plenamente descripto en los axiomas.

Cuando alguna parte del comportamiento del TAD debe ser automática:

- No deberíamos especificar una acción “manual” para este comportamiento (a menos que también sea el caso).
- No deberíamos permitir que existan instancias del TAD en las que el comportamiento deber aí haberse aplicado y no lo hizo.
- No deberíamos restringir acciones que requieran que suceda el comportamiento, ya que éste es automático.

TADs en dos niveles:

- Identificar la parte estática de la parte dinámica del problema
- Analizar si conviene partir en 2 niveles para sumar claridad a la especificación
- Las partes tienen que tener comportamiento marcadamente distinto y separable

Cuándo conviene usar los TAD básicos y cuándo TADs más específicos?

- Analizar las “partes” o “etapas” de un problema y como impactan en un TAD.
- Verificar si hay partes inmutables o siempre están cambiando.
- Identificar en el problema “entidades” con comportamiento/responsabilidades/propiedades asociadas y como se relacionan con otras “entidades”.

Eficiencia de algoritmos

- Para que? Para comparar algoritmos que resuelven el mismo problema.
- Por que? Para optimizar recursos/minimizar su consumo.

Costo: El costo de aplicar un algoritmo a una instancia depende de el # de instrucciones para esa instancia.

Tamaño (de una instancia): #bits

Complejidad: mide el costo en funcion del tamaño de sus instancias. Vamos a estudiar la **eficiencia asintotica,** es decir, como el costo aumenta con el crecimiento del input, cuando este tiende a infinito (en el limite), ignorando factores multiplicativos. Usualmente el algoritmo que es asintoticamente mas eficiente, va a ser la mejor opcion excepto en los inputs muy pequeños.

Tipos de analisis:

- Peor caso: $T(n)=max\{c(i)|i\ mide\ n\}$ (n = tamaño de las sentencias, c(i) = costo de la instancia I.
- Mejor caso: $T(n)=min\{c(i)|i\ mide\ n\}$
- Tiempo esperado: no lo vamos a usar.

La complejidad se puede medir con diferentes escalas y contando operaciones no elementales →

Tiempo en “operaciones” (elementales) en funcion de “tamaño de entrada” (bits).

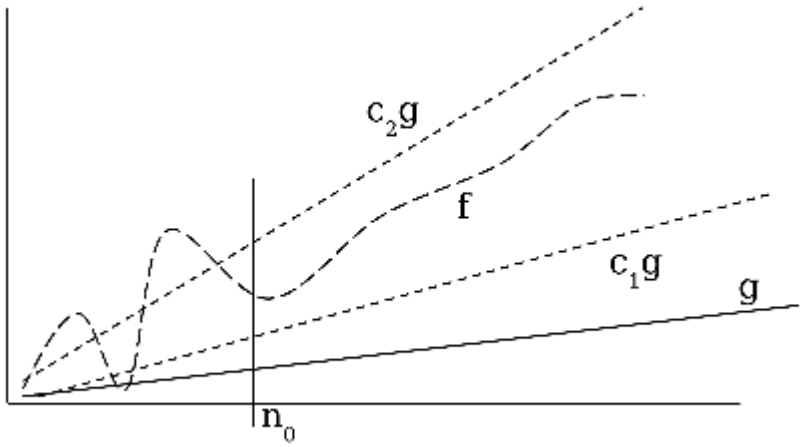
Orden de crecimiento: Ignoramos factores multiplicativos.

- $\Theta(g(n))=\{f(n):\exists\ ctes\ positivas\ c_1, c_2, y\ n_0\ tal\ que\ 0\le c_1 g(n)\le f(n)\le c_2 g(n)\ \forall\ n\ge n_0\}$

  En otras palabras, $f(n)=\Theta(g(n))$ si existen c_1 y c_2 ctes positivas tal que se pueda “ensandwichear” entre $c_1g(n)$ y $c_2g(n)$ para un n suficientemente grande.

  $f(n)=\Theta(g)\Rightarrow f=O(g)\ y\ f=\Omega(g)$

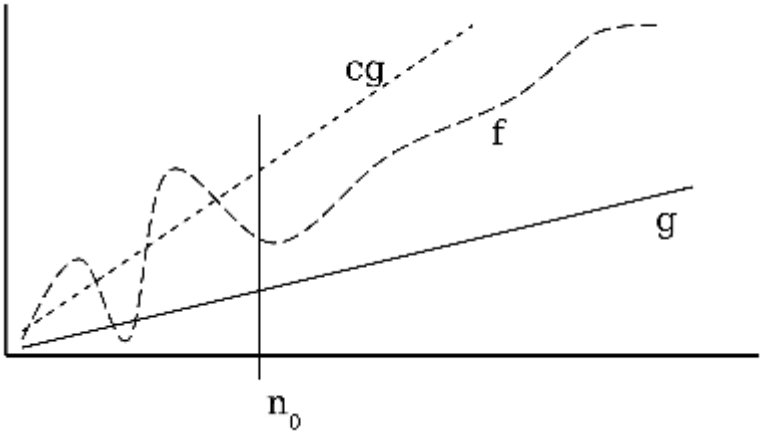
  $c_0 g(n)\le f(n)\le c_1 g(n)$



- $O(g(n))=\{f(n):\exists\ ctes\ positivas\ c\ y\ n_0\ tal\ que\ 0\le f(n)\le c\ g(n)\ \forall\ n\ge n_0\}$

  Usamos la notacion O para darle un limite superior asintotico a una funcion. (Acotado superiormente por f)

  $f(n)\le cg(n)$

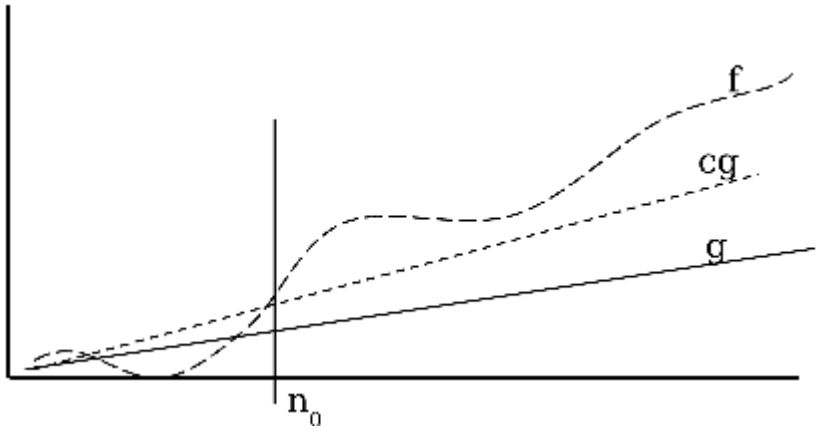


Not: $f(n)=O(g(n))! \Rightarrow g(n)=O(f(n))$

- $\Omega(g(n))=\{f(n): \exists \text{ctes positivas } c \text{ y } n_0 \text{ tal que } 0 \le c\,g(n) \le f(n) \; \forall\, n \ge n_0\}$

  Usamos la notacion Ω para darle un limite inferior asintotico a una funcion. (Acotado inferiormente por f)

  $cg(n) \le f(n)$



Not: $f(n)=\Theta(g(n))$ implica $f(n)=\Omega(g(n))$.

Mini teorema: $f(n)=\Theta(g(n)) \Leftrightarrow f(n)=O(g(n)) \text{ y } f(n)=\Omega(g(n))$

Propiedades:

- O(f):
  - $m \log n = O(mn)$
  - $f=O(n^k)$?
    - $\forall n \in \mathbb{N}: \text{si } f=O(n^k) \Rightarrow n^k=O(n^{k+1})$
    - $\forall k \in \mathbb{N}: n^k=O(2^n)$
    - Si f es un polinomio, $f=O(n^k)$ para algun $k=O(1)$

        - $n^k \neq O(k^n)$
        - $n \to n^k = O(n \to k^n)$
    - Sea $f(n) = 2^n (como\ numero, no\ como\ funcion) \Rightarrow \forall\, n: f(n) = O(1)$
    - A^n, B^m donde n=tam(A) y m=tam(B)

      $T_{peor}(n+m) = O(n * m)$
- <u>Ω(f):</u> igual a O(f) pero al revez.
- <u>Θ(f):</u> $O(f) \cap \Omega(f)$
    - $f = O(g)\ y\ g = O(f) \to f = \Omega(g) \to f = \Theta(g)$
    - $O(1) \subset O(\log n) \subset O(n) \subset O(n^k)[k<1] \subset O(n) \subset O(n^k)[k>1] \subset O(k^n) \subset O(n!) \subseteq O(n^n)$

<u>Operaciones sobre ordenes</u>

1. Agrego las constantes
2. Resuelvo
3. Vuelvo a incluir los ordenes

$f = O(n^2) + O(n) \to f \leq c_1 n^2 + c_2 n \leq (c_1 + c_2) n^2 = O(n^2)$

$g = O(1) + O(1) \to c_1 * 1 + c_2 * 1 = O(1)$

<u>Algebra de ordenes</u>

- $O(f) + O(g) = O(f+g) = O(max\{f, g\})$
- $O(f) * O(g) = O(f * g)$
- $O(f) \circ O(g) = O(f \circ g)$ (en general)
- $\sum_{i=1}^{n} O(f) = O(\sum_{i=1}^{n} f) = O(nf)$ y
    - Si n es constante: $\sum_{i=1}^{n} O(f) = O(f)$
- $\prod_{i=1}^{n} O(f) = O(\prod_{i=1}^{n} f) = O(f^n)$
- En general $f(n) = g(n) \circ O(h(n))$ significa que
    - $f(n) = g(n) \circ h'(n)$ para algun $h' = O(h)$ es decir $f(n) \leq g(n) \circ (c * h(n))\ \forall\, n \geq n_0$

Diseño jerarquico de TADs

<u>Diseñar:</u> Pasar de la descripcion del que del problema al como.

- A nivel conceptual:

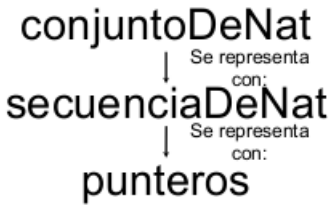
- Preocuparnos no ya del qué sino del cómo.
- Cambiar de paradigma (del funcional de la especificación, al imperativo del programa).
- Resolver los problemas que surgen como consecuencia de eso.

- En un plano un poco más concreto… un modulo es:
  - Proveer una representación para los valores.
  - Definir las funciones del tipo.
  - Demostrar que eso es correcto.

  O

  - Proveer una representacion del tipo para las instancias.
  - Definir la interfaz e implementar.
  - Demostrar correccion.

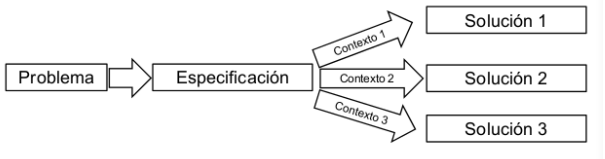
Diseño Jerarquico: Un tipo se representa usando tipos ya existentes (salvo primitivos)



Primitivos: nat, puntero(α), bool, arreglo_dimensionable(α), tupla.

Que se resuelve en cada nivel?

Contexto de uso: ¿Cómo discriminamos entre dos soluciones? De acuerdo al contexto de uso, y los requerimientos de eficiencia. (solucion=modulo)



Metodologia de diseño: Diseñar implica las siguientes tareas:

- Elección del tipo a diseñar
- Introducción de los elementos no funcionales
- Vinculación entre la representación y su abstracción
- Iteración sobre los tipos restantes (filosofía “top-down”)

Partes del diseño/modulo:

- Los **aspectos de la interfaz** de un tipo describen todo elemento relacionado con los aspectos de uso de dicho tipo, es decir, toda cuestión referida a lo que resulte visible desde “afuera”.

Es la seccion accesible para los usuarios del modulo (que pueden ser otros modulos). Se detallan los servicios exportados. Cada operacion del modulo viene acompañado de:

- Signatura.
- Precondición.
- Postcondición.
- Complejidad Temporal.
- Descripción (Importante!).
- Aliasing .

Interfaz: define en el paradigma imperativo las operaciones exportadas junto con su precondición y postcondición. Esto establecerá la relación entre la implementación de las operaciones y su especificación.

- Las **pautas de implementación** serán todo aspecto que refiera a cuestiones vinculadas a los medios a través de los cuales el tipo garantiza esos aspectos de uso.

  No es accesible a los usuarios del modulo. Aqui se detalla la eleccion de estructura de representacion y los algoritmos. Se justifica la eleccion de las estructuras asi como la complejidad de los algoritmos.

La definición de la interfaz de un módulo de diseño implica tomar cuenta de varias cosas y esencialmente debe explicarle al eventual usuario todos los aspectos relativos a los servicios que exporta:

- **Servicios exportados:** describe para cada operación su complejidad, aspectos de aliasing, efectos colaterales sobre los argumentos, etc.

Transparencia referencial: Una función es referencialmente transparente si su resultado sólo depende de sus parámetros explícitos.

Ej:

- si f(x) := {return x + 1}, f (4) + f (3) es r.t, pero
- si f (x) := {y = G*(x + 1);G + +; return y}, no lo es.

Aliasing: significa la posibilidad de tener más de un nombre para la misma cosa. En concreto, dos punteros o referencias hacia el mismo objeto.

Ej: una operación que dado un árbol binario (no vacío) nos devolviera dos árboles y un elemento (subárbol izquierdo , derecho y raíz).

Podar(in A: ab(elem), out I: ab(elem), out r: elem, out D: ab(elem))

- Implementación 1: armar copias de los subárboles izquierdo y derecho de A, devolverlas como I y D.
- Implementación 2 (más rápida): devolver en I y D referencias a los subárboles de A.

En este último caso estaríamos provocando aliasing entre los árboles, causando que cualquier modificación que se realice sobre I o D, luego de llamar a la operación Podar, repercuta en A.

Manejo de errores: Ejemplo: Encolar(inout C: cola, in e: elem) vs. Encolar(inout C: cola, in e: elem, out s: status)

Debido a que el paradigma funcional tiene transparencia referencial, no teníamos este “problema”. ¿Es malo? No, pero debe ser documentado porque ¡es tan público como la complejidad!

Relación entre los valores imperativos y los valores lógicos

Si el lenguaje de implementación es diferente del lenguaje de especificación,

- qué diferencia existe entre los valores que pueden tomar las variables imperativas respecto de los términos lógicos?
- cómo podemos vincularlos?

Para resolver este problema introduciremos una función que para cada "valor imperativo" nos retornará un término lógico al cual representa, de forma que éste pueda participar de los predicados lógicos que definen el comportamiento formal de las operaciones...

... es decir que introduciremos una notación definida como una función que va del género imperativo G I al género G T correspondiente a su especificación o al genero representado,

$$\hat{o} : G_I \rightarrow G_T$$

Al diseñar…

- El limite de diseño son los iteradores, y no hace falta alclararlos.
  - Ejemplos de iteradores: dameAlguno, sinAlguno.
  - No puedo especificar dameUno y sinUno.
- No me tengo que olvidar el iterador al final! Si no, no funciona.

Representacion

- Estructura (struct)
- Relacion entre representacion y abstraccion
- Implementacion de algoritmos
- Damos cuenta de que servicios usamos

Estructura de representacion

- Tipo con el que represento el tipo que esta implementando
- Necesito poder representar todas las instancias cumpliendo con el contexto de uso.

Ej: vector<α> (genero) se representa con iVector(α), donde iVector(α) es:

tupla<elems:arreglo_dimensionable(α), ultimo: nat>

Invariante de representacion

Nos dice que valores pueden tomar las variables dentro de nuestra estructura para que esta sea válida

Tiene que valer siempre en el momento inicial y final de cada función descrita en la interfaz.

Si T_R se representa con T_I → $Rep : \widehat{T_I} \rightarrow bool$ o $Rep : \widehat{estr} \rightarrow bool$

$(\forall t : T_I)\, Rep(t) \equiv ...condiciones\ que\ tiene\ que\ satisfacer\ t\ para\ representar\ un\ \widehat{T_R}$

<u>Consejos para escribir el invariante de representacion</u>

- Escribirlo primero en castellano y luego pasarlo a lógica (y relacionar ambas partes con números). De hecho, esto se pide explícitamente en algunos ejercicios.
- Tratar de resolver un predicados lógico de entrada con un ⇔ (“si y sólo si”) puede trabarnos y hacernos incurrir en errores. Usar el ⇔ con cuidado y verificar que las dos implicaciones ⇒ y ⇐ sean equivalentes al ⇔.
- Tener en cuenta (tipo checklist) que el invariante debe abarcar estos aspectos (notar que algunos de ellos se solapan):
  - Coherencia en la información redundante: Hay que chequear que distintos campos que proveen la misma información no se contradigan entre sí.
  - Restricciones del TAD: Hay que chequear que se vean reflejadas en la estructura de representación. Por ejemplo, en conjunto en rango el TAD indica que no se puede crear un conjunto con el límite inferior del rango mayor al límite superior. Entonces en el invariante hay que pedir que e.lower ≤ e.upper.
  - Decisiones de diseño: Hay que chequear las restricciones a la estructura de representación que no provengan de un chequeo de coherencia o de restricciones especificadas en el TAD. Por ejemplo, si decidimos implementar conjunto con una secuencia sin repetidos, entonces en el invariante debemos chequear que la secuencia, efectivamente, no tenga repetidos. La necesidad de hacer este chequeo no puede deducirse del TAD, y tampoco tiene que ver con un chequeo de coherencia de información redundante.

<u>Ej:</u>

- <u>vector<α>:</u> $Rep : \widehat{iVector(\alpha)} \rightarrow bool$

  $Rep(t) \equiv true \Leftrightarrow t.ultimo \leq tam(t.elems)\ y_L (\forall i : Nat)(0 \leq i < t.ultimo \Rightarrow_L def(elem, i))$

- <u>Conj. Lineal:</u> $Rep : \widehat{vector(\alpha)} \rightarrow bool$

  $Rep(s) \equiv sinRepetidos(s)$

- <u>Conjunto en rango:</u> se representa con estr = <low: nat, upper: nat, elems: secu(nat), min: nat>
  - Restricciones de los generadores:

En castellano	En logica

La cota inferior es menor o igual que la cota superior	$e.low \leq e.upper$
Todos los elementos del conjunto son mayores que la cota inferior y menores que la cota superior	$(\forall n:Nat)\, esta?(n, e.elems) \Rightarrow e.low \leq n \leq e.upper$

- Decisiones de disenio:

En castellano	En logica
La secuencia no tiene elem. repetidos	$(\forall n:Nat)\, esta?(n, e.elems) \Rightarrow_L cantApariciones(n, e.elems) = 1$

- Coherencia en la informacion redundante

En castellano	En logica
Si hay elementos en la secuencia, el minimo es uno de ellos	$\neg \emptyset?(e.elems) \Rightarrow e.min \in e.elems$
El minimo es efectivamente el minimo del conjunto	$(\forall n:Nat)\, esta?(n, e.elems) \Rightarrow e.min \leq n$

Funcion de abstraccion

Nos dice con qué instancia del TAD que estoy diseñando se vincula mi instancia de estructura de representación. Es decir, vincula la imagen abstracta de una estructura con el valor abstracto al que representa.

$Abs: \widehat{T_I} \rightarrow \widehat{T_R}$ o $Abs: \widehat{estr} \rightarrow \widehat{tipoAbstractoRepresentado}$

$Abs(t) \equiv e | ...describir\ e\ con\ observadores$

Ej:

- Vector<α>: $Abs: iVector(\alpha) \rightarrow secu(\alpha)$

  $Abs(t) \equiv arrayToSecu(t.elems, 0, ultimo)$

- conjLineal(α): $Abs: secu(\alpha) \rightarrow conj(\alpha)$

  $Abs(t) \equiv c | (\forall e:\alpha)(pertenece(e,c) \Leftrightarrow esta?(e,t))$

- Conjunto en rango: $Abs: \widehat{estr}\ e \rightarrow conjRan$

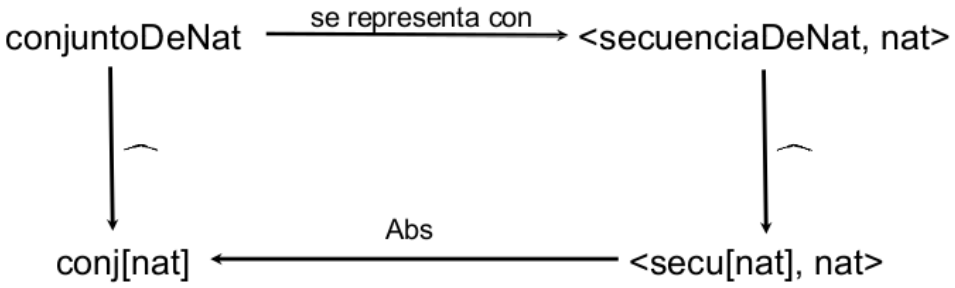
  $Abs(e) = c | ((low(c) = e.lower) \wedge (up(c) = e.upper) \wedge (\forall n:Nat)(n \in c \Rightarrow esta?(n, e.elems)))$

Propiedades

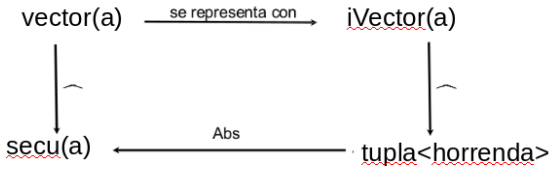
- Total restringido a rep
- No es inyectiva (muchas representaciones del mismo representado)
- Es sobreyectiva sobre el tipo representado modulo $=_{obs}$
- No es sobreyectuva sobre los terminos que describen las instancias

Ejemplo:

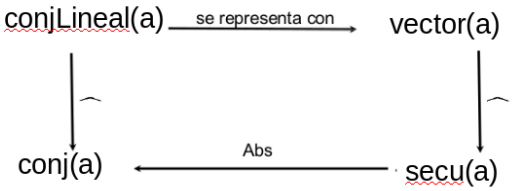
- conjunto con <Secuencia, nat>



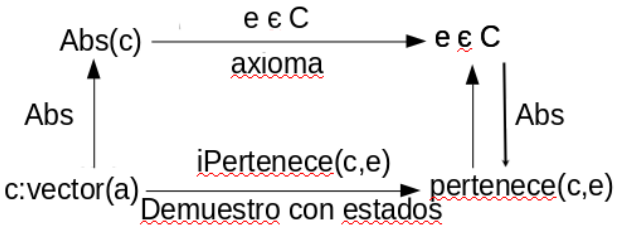
- Vector(α)



- conjLineal(α)



Interfaz	**Algoritmos**
pertenece(in c:conjLineal(α), in e:α) →res:bool P≡{true} Q≡{res=$_{obs}$e ∈ C}	ipertenece(in c:vector(α), in e:α) →res:bool P≡{Rep(c)} return esta?(c,e); Q≡{res=$_{obs}$esta?(c,e)}



Sombrerito(^)

La función ^ toma una estructura del mundo del diseño y nos devuelve automágicamente su correspondiente instancia abstracta del mundo de los TADs.

- Para tipo primitivo: Para cada tipo primitivo T defino $^{\wedge}_{T}$ indicando que representa cada secuencia de bits

  $^{\wedge}_{int64}$: int64 i → nat {iRep(i)}

  $^{\wedge}_{nat}$: nat → nat

- Para tipo no primitivo: se representa con E

  $^{\wedge}_{T}$: T t → $\hat{T}$ {rep($\hat{t}^{E}$)}

  $^{\wedge}_{T}$(t) ≡ abs($\hat{t}^{E}$)