Algoritmos y Estructuras de Datos II
Diseño jerárquico de TADs

Esteban Feuerstein y Emmanuel Iarussi
(Fuertemente basado en clases de Carlos López Pombo y Fernando Schapachnik)

Departamento de Computación,
Facultad de Ciencias Exactas y Naturales,
Universidad de Buenos Aires

¿Qué significa diseñar?

- Pasar de la descripción del **qué** del problema al **cómo.**

- Considerar aspectos no funcionales, por ejemplo, eficiencia en tiempo y espacio de acuerdo al contexto de uso, así como buenas prácticas como el encapsulamiento y ocultamiento de información.

- Permitir un cambio de paradigma (del utilizado para especificar al utilizado para programar) que resulte “ordenado”, metódico, “más o menos” formal.

Empecemos con un ejemplo

- Consideremos el TAD Conjunto.
- Veamos dos implementaciones posibles:
  - Un arreglo redimensionable.
    - Inserción (sin repetidos): O(n)
    - Búsqueda: O(log(n))
  - Una secuencia.
    - Inserción (sin repetidos): O(1)
    - Búsqueda: O(n)
- ¿Cuál me conviene?
- Depende de qué necesite...

Pero…

Está claro es que no podemos pasar de la especificación al código directamente, necesitamos una etapa intermedia:

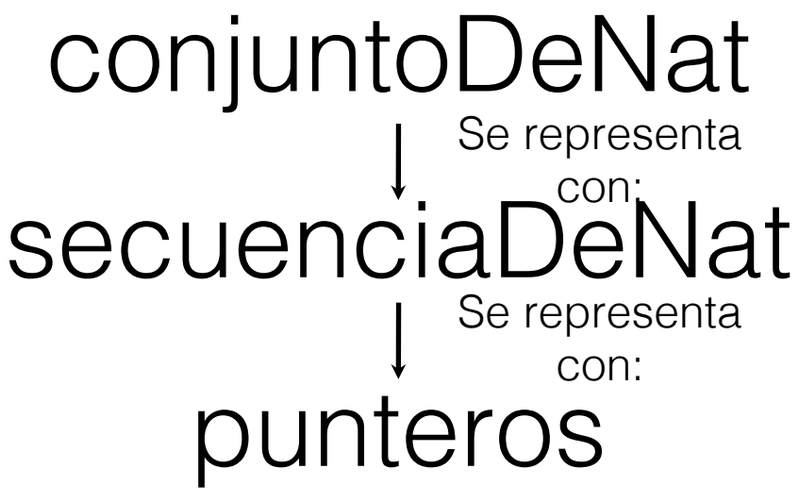
La etapa de diseño

¿Qué significa diseñar?

- 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...
  - Proveer una *representación* para los valores.
  - *Definir* las *funciones* del tipo.
  - Demostrar que eso es *correcto*.

¿Qué significa jerárquico?

Que pensaremos la resolución del *cómo* a partir de representaciones de un tipo sobre otros separando responsabilidades en la construcción de la solución:

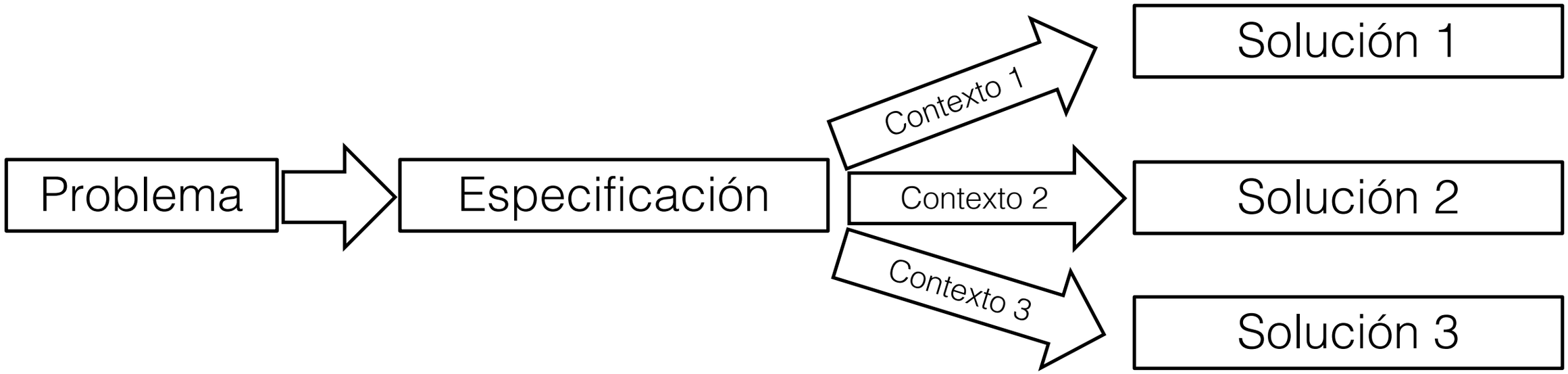


Intuitivamente

¿Qué se resuelve en cada nivel?

Contexto de uso

- ¿Cómo discriminamos entre dos soluciones?
- De acuerdo al contexto de uso, y los requerimientos de eficiencia



Metodología de diseño

Desde un punto de vista abstracto 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”)

Metodología de diseño

- 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”,

- 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.

Metodología de diseño

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.
- **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.

Transparencia referencial, aliasing, etc.

- Una función es *referencialmente transparente* si su resultado sólo depende de sus parámetros explícitos.
- Por ejemplo,
  - 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.

Transparencia referencial, aliasing, etc.

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

Transparencia referencial, aliasing, etc.

- Etc.: 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!

Ejemplo, con Conj(NAT)

TAD CONJ(NAT)

usa BOOL, NAT

géneros conj(nat)

exporta conj(nat), $\bullet \in \bullet$, $\emptyset$, Ag, $\emptyset?$, mínimo

observadores básicos

$\bullet \in \bullet$: nat × conj(nat) ⟶ bool

generadores

$\emptyset$: ⟶ conj(nat)

Ag : nat × conj(nat) ⟶ conj(nat)

otras operaciones

$\bullet - \{\bullet\}$: conj(nat) × nat ⟶ conj(nat)

$\emptyset?$: conj(nat) ⟶ bool

mínimo : conj(nat) c ⟶ nat $\{\neg\emptyset?(c)\}$

axiomas los tradicionales

Fin TAD

Ejemplo de Contexto

Contexto de uso:

- la obtención del mínimo debe tener complejidad *O(1)*,
- quitar un elemento debe tener complejidad *O(n)*, siendo n la cantidad de elementos agregados
- agregar un elemento debe tener complejidad *O(1).*

Ejemplo - Interfaz

Servicios exportados:

pertenece: no produce aliasing ni efectos colaterales sobre los argumentos, posee orden de complejidad temporal *O(n)* con *n* la cantidad de elementos agregados al conjunto,
vacio: no produce aliasing ni efectos colaterales, posee orden de complejidad temporal *O(1)*
agregar: no produce aliasing, modifica colateralmente el conjunto argumento, posee orden de complejidad temporal *O(1)*,
quitar: no produce aliasing sobre los argumentos, modifica colateralmente el conjunto argumento, posee orden de complejidad temporal *O(n)* con *n* la cantidad de elementos agregados al conjunto,
vacio?: no produce aliasing ni efectos colaterales, posee orden de complejidad temporal *O(1)*
minimo: no produce aliasing ni efectos colaterales, posee orden de complejidad temporal *O(1).*

Metodología de diseño

Relación entre los valores imperativos y los valores lógicos

Quiséramos poder decir lo siguiente...

Interfaz:

pertenece (**in** conjuntoDeNat c, **in nat** n) -> res: **bool**

$$\{true\}$$
$$\{res =_{obs} n \in C\}$$

¿Qué problema observan?

Metodología de diseño

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?

Metodología de diseño

Relación entre los valores imperativos y los valores lógicos

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,

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

Metodología de diseño

Relación entre los valores imperativos y los valores lógicos

Es decir que escribiremos...

Interfaz:

pertenece (**in** c: conjuntoDeNat, **in** n: **nat**) -> res: **bool**

$\{true\}$

$\{r\hat{e}s =_{obs} \hat{n} \in \hat{C}\}$

Al operador $\hat{\bullet}$ lo llamaremos cariñosamente “sombrerito”

Metodología de diseño

Relación entre los valores imperativos y los valores lógicos

Ejemplo

Interfaz:

pertenece (**in** c: conjuntoDeNat, **in** n: **nat**) -> res: **bool**

{**true**}

$\{\hat{res} =_{obs} \hat{n} \in \hat{C}\}$

vacio () -> res: conjuntoDeNat

{**true**}

{$\widehat{res} =_{obs} \emptyset$}

agregar (**in/out** c: conjuntoDeNat, **in** n: **nat**)

{$\widehat{c} =_{obs} c_0$ }

{$\widehat{c} =_{obs} Ag(c_0, \widehat{n})$ }

quitar (**in/out** c: conjuntoDeNat c, **in** n: **nat**)

{$\widehat{c} =_{obs} c_0$ }

{$\widehat{c} =_{obs} c_0 - \{\widehat{n}\}$ }

Ejemplo

Relación entre los valores imperativos y los valores lógicos

Interfaz:

vacio? (**in** c: conjuntoDeNat) -> res: **bool**
{**true**}
{ $\widehat{res}$ $=_{obs}$ *b?* ($\widehat{c}$)}
minimo(**in** c: conjuntoDeNat) -> res: **nat**
{*not vacio?* ($\widehat{c}$)}
{ $\widehat{res}$ $=_{obs}$ *minimo* ($\widehat{c}$)}

Ah, y como todos sabemos, el sombrero se está usando poco…

Recapitulando

- Vimos por qué es necesaria la etapa de diseño, qué cambios introduce, y cuál es el lenguaje que usamos.

- Es decir: La diferencia de mundos, cómo los requerimientos de eficiencia deciden la implementación, la idea de diseño top-down, el cambio de paradigma, aliasing y “sombrerito”.

- Veremos ahora: cómo escribir un módulo, qué cosas debemos considerar, cómo verificar su relación con el TAD.

Ocultando información

- ¿Por qué tanto énfasis en la interfaz?
- ¿No es más fácil dar el código y listo?
- Primera piedra: *Information hiding*, David Parnas,“On the Criteria to Be Used in Decomposing Systems Into Modules” (Communications of the ACM, Diciembre de 1972).

Ocultando información

- Abstracción:“Abstraction is a process whereby we identify the important aspects of a phenomenon and ignore its details.” [Ghezzi et al, 1991]

- Information hiding: “The [...] decomposition was made using ’information hiding’ [...] as a criterion. [...] Every module [...] is characterized by its knowledge of a design decision which it hides from all others. Its interface or definition was chosen to reveal as little aspossible about its inner workings.” [Parnas, 1972b]

- “[...] the purpose of hiding is to make inaccessible certain details that should not affect other parts of a system.” [Ross et al, 1975]

- Encapsulamiento: “[...] A consumer has full visibility to the procedures offered by an object, and no visibility to its data. From a consumer’s point of view, an object is a seamless capsule that offers a number of services, with no visibility as to how these services are implemented [...] The technical term for this is encapsulation.” [Cox, 1986]

Ocultando información

- Ventajas del ocultamiento, la abstracción y el encapsulamiento:
  - La implementación se puede cambiar y mejorar sin afectar su uso.
  - Ayuda a modularizar.
  - Facilita la comprensión.
  - Favorece el reuso.
  - Los módulos son más fáciles de entender.
  - Y de programar.
  - El sistema es más resistente a los cambios.
- Aprender a elegir buenas descomposiciones no es fácil. Ese aprendizaje comienza ahora y continúa en Ingeniería del Software I/II.

La Representación

La definición de la **representación** de un módulo de diseño implica tomar cuenta todo aspecto referido a cómo se satisfacen los requerimientos declarados en la interfaz.

•**Estructura**: describe la estructura interna sobre la cual las operaciones aplican.

•**Relación entre la representación y la abstracción**: por un lado expone toda restricción sobre la estructura de representación a fin de que efectivamente pueda ser considerada una implementación de un valor del tipo al que implementa; y por otro vincula los valores con su contraparte abstracta, es decir, con algún término de la especificación a quien este represente.

•**Algoritmos**: la programación en pseudo-código de las operaciones, tanto las exportadas como las auxiliares, y incluyendo para todos ellos el cálculo detallado que justifica su complejidad

•**Servicios usados**: declara toda demanda de complejidad, aliasing o efecto colateral que los servicios usados de otros tipos en la programación de los algoritmos deban satisfacer.

La Representación

Estructura de representación

La *estructura de representación* describe los valores sobre los cuales se representará el género que se está implementando.

Esta tiene que tener en cuenta la posibilidad, no sólo de ser capaz de representar *todos* los términos del género de la especificación, sino también que las operaciones sean implementables de acuerdo a las exigencias del contexto de uso.

Ejemplo: conjunto semi rápido

- Problema: implementar un conjunto de naturales, con el siguiente contexto: los números del *1* al *100* deben manejarse en *O(1)* porque se usan mucho. El resto, en *O(n)*. Rápidamente debo conocer la cardinalidad.

- Propuesta:
  - Un arreglo de 100 posiciones booleanas.
  - Una secuencia.
  - Un nat para la cardinalidad.

- Cómo lo escribimos:
  - Conj_semi_rápido_nat **se representa con**
    tupla <*rápido*: arreglo [1..100] de bool × resto: secu(nat) × cant: nat>

- Alternativamente:
  - Conj_semi_rápido_nat **src** estr
    donde estr **es**
    tupla <rápido: arreglo [1..100] de bool × resto: secu(nat) × cant: nat>

El invariante de representación

- ¿Cualquier instancia es válida?

  ¿<[0...0], <>, 8254> es un Conj_semi_rápido_nat válido?

  ¿<[0...0], <37,107, 28>, 3> es un Conj_semi_rápido_nat válido?

- ¿Para qué nos serviría poder separar con facilidad instancias válidas e inválidas?
  - Como una forma de documentar la estructura.
  - Como condición necesaria para establecer una relación con la abstracción (ver más adelante).
  - Para agregar a las postcondiciones, como una forma de garantizar que nuestros algoritmos no rompen la estructura.
  - Para agregar a las precondiciones, como una “garantía” con la que cuentan nuestros algoritmos.
  - Como una guía a la hora de escribir los algoritmos.
  - Si pudiésemos programar el chequeo, como una forma de detectar instancias corruptas.

El invariante de representación

- Es una función booleana con dominio en el género de representación que da true cuando recibe una instancia válida.

- ¿Podría el dominio ser el género representado? ¿Por qué?

- En realidad, si nos ponemos finos, el dominio es la versión abstracta del género de representación.

- Si representamos T_1 sobre T_2

$$\text{Rep} : \hat{T}_2 \rightarrow bool$$

$(\forall t : \hat{T}_2)\,\text{Rep}(t) \equiv \ldots$ condiciones que garanticen que t representa una instancia válida de T_1...

Volviendo al ejemplo…

- ¿Qué debería decir el invariante?
  1. Que *resto* sólo tiene números mayores a 100, si tiene alguno.
  2. Que *resto* no tiene números repetidos.
  3. Que *cant* tiene la longitud de *resto* más la cantidad de celdas de *rápido* que están en *true*.

$$\text{Rep} : \hat{estr} \rightarrow bool$$

$$(\forall e : \hat{estr})\,\text{Rep}(e) \equiv$$

$$(1)(\forall n : nat)(\text{esta?}(n, e.resto) \Rightarrow n > 100) \wedge$$

$$(2)(\forall n : nat)(\text{cant\_apariciones}(n, e.resto) \leq 1) \wedge$$

$$(3)e.cant = \text{long}(e.resto) + \text{contar\_trues}(e.rápido)$$

La Función de Abstracción

- La **función de abstracción** es una herramienta que permite vincular una estructura con **algún** valor abstracto al que representa. ¿Cómo les parece que esto puede hacerse? Es decir, ¿qué debemos usar para caracterizar algún término abstracto que representa a una estructura particular?

- $\mathrm{Abs} : \hat{\mathrm{T}}_2\ \mathrm{e} \rightarrow \hat{\mathrm{T}}_1\ (\mathrm{Rep}(\mathrm{e}))$

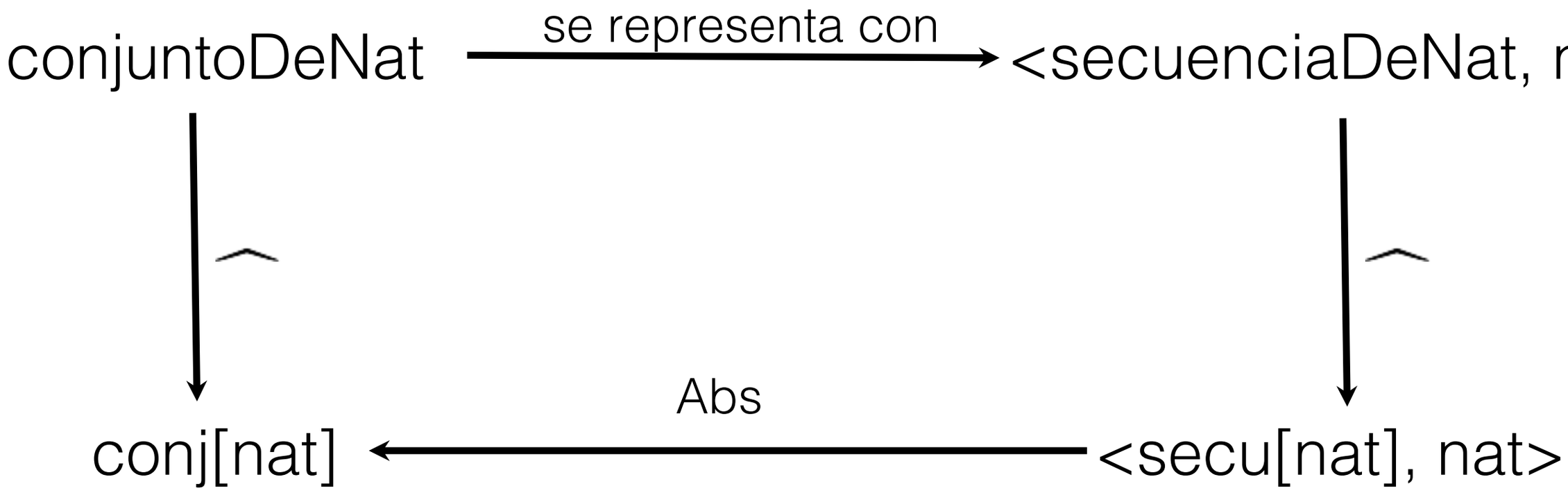
- La manera en la que se caracteriza un término es o bien a través de los generadores o de los observadores. Normalmente el uso de observadores resulta más sencillo

La Función de Abstracción

Ejemplo, Conjunto con <Secuencia,nat>

La Función de Abstracción

Ejemplo, Conjunto con <Secuencia,nat>



La Función de Abstracción

Ejemplo, Conjunto con <Secuencia,nat>

$$Abs :< s, n >: < \widehat{\texttt{secuenciaDeNat}}, \texttt{nat} > \to \texttt{conj}[\texttt{nat}] \qquad Rep(< s, n >)$$

$$(\forall < s, n >:< \texttt{secu}[\texttt{nat}], \texttt{nat} >)(Abs(< s, n >) =_{obs} c \mid$$
$$(\forall n' : \texttt{nat})(esta?(s, n') \iff n' \in c))$$

$$esta? : \texttt{secu}[\texttt{nat}] \times \texttt{nat} \to \texttt{bool}$$
$$esta?([\,], n) \equiv false$$
$$esta?(a \bullet s, n) \equiv (a = n) \vee esta?(s, n)$$

La Función de Abstracción

- Volvamos al ejemplo de conj_semi_rápido
- Conj_semi_rápido_nat src estr
  donde estr es
  tupla <*rápido*: arreglo [1..100] de *bool* × *resto*: *secu(nat)* × *cant: nat*>

$$\text{Abs} : \hat{estr} \quad e \rightarrow conj\_semi\_rapido\_nat \quad (\text{Rep}(e))$$

$$\text{Abs}(e) =_{obs} c \;/$$

$$(\forall n : \text{nat})(n \in C \Leftrightarrow ((n \leq 100 \wedge e.rápido[n]) \vee$$

$$(n > 100 \wedge \text{está?}(n, e.resto)))$$

La Función de Abstracción

Propiedades

- ¿Es total o parcial? Una vez restringida a (Rep(e)), deber ser total.
- No tiene por qué ser inyectiva: Dos estructuras diferentes pueden representar al mismo término de un TAD.
- Debe ser suryectiva sobre las clases de equivalencia determinadas por la igualdad observacional, restringidas a lo especificado por el contexto de uso.
- No tiene por qué ser suryectiva sobre todos los términos del género representado.

Los algoritmos

- Recordemos, interfaz de Ag()

$$\text{Ag}(\text{inout}\, C : conj\_semi\_rápido\_nat, \text{in}\, e : \text{nat})$$
$$\{\hat{C} =_{obs} C_0 \wedge \hat{e} \notin C\}$$
$$\{\hat{C} =_{obs} \text{Agregar}(C_0, \hat{e})\}$$

Los algoritmos

- Implementación

iAg(inout C : *estr*, in e : nat)

$\{\hat{C} =_{obs} C_0 \wedge \hat{e} \notin C \wedge \text{Rep}(C)\}$

$C.cant++$

if $(e \leq 100)$ then $c.rápido[e] := true$ else $ag\_en\_secu(C, e)$ fi

$\{\hat{C} =_{obs} \text{Agregar}(C_0, \hat{e}) \wedge \text{Rep}(C)\}$

ag_en_secu(inout E: estr, in e: nat)

$\{\hat{E} =_{obs} E_0 \wedge e > 100\}$

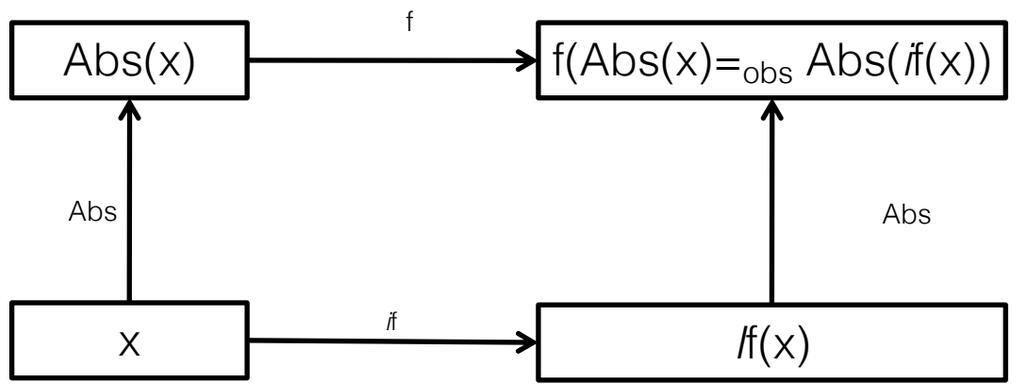
// Notar: acá no vale Rep(E)

InsertarAlFinal(E.resto, e)

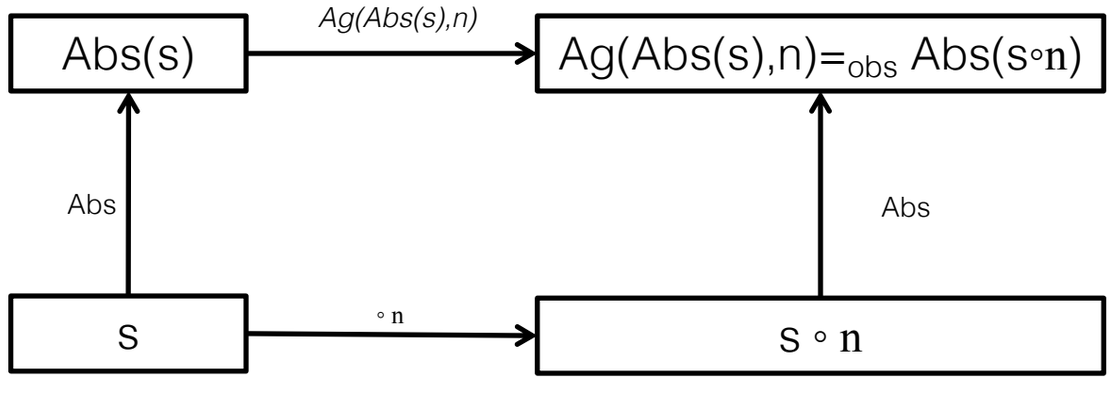
$\{\hat{E}.resto =_{obs} E_0.resto \circ \hat{e}\}$

Probando Corrección

- Para toda operación f que implementa una operación del TAD y toda x instancia del género de representación, debemos ver que el siguiente diagrama conmuta



- Ejemplo para Conjunto sobre secuencia



Probando Corrección

•Abs debe ser un homomorfismo respecto de la signatura del TAD, o sea, para toda operación ●,

Abs(i●(p1, . . . , pn)) =obs ●(Abs(p1),. . . , Abs(pn))

•Abs y Rep se las debemos a C.A.R Hoare (Premio Turing 1980) [Hoare, 1972].

Bibliografía

- “Abstraction, Encapsulation, and Information Hiding”. By Edward V. Berard. The Object Agency. http://www.tonymarston.net/php-mysql/abstraction.txt
- [Parnas, 1972b] D.L. Parnas,“On the Criteria To Be Used in Decomposing Systems Into Modules,”Communications of the ACM, Vol. 5, No. 12, December 1972, pp. 1053-1058.
- [Ghezzi et al, 1991] C. Ghezzi, M. Jazayeri, and D. Mandrioli, Fundamentals of Software Engineering, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
- [Ross et al, 1975] D.T. Ross, J.B. Goodenough, and C.A. Irvine, “Software Engineering: Process, Principles, and Goals,” IEEE Computer, Vol. 8, No. 5, May 1975, pp. 17 - 27.
- [Cox, 1986] B.J. Cox, Object Oriented Programming: An Evolutionary Approach, Addison-Wesley, Reading, Massachusetts, 1986.
- [Hoare, 1972] C.A.R. Hoare. ”Proof of correctness of Data Representation”. Acta Informatica 1(1), 1972.