Apunte de M?dulos B?sicos (v. 0.3α)

Algoritmos y Estructuras de Datos II, DC, UBA.

1er cuatrimestre de 2019

Índice

1. Introducci?n

El presente documento describe varios m?dulos que se pueden utilizar para realizar el TP de dise?o. Adem?s, sirve como ejemplo de qu? se espera del TP de dise?o, y muestra algunas t?cnicas que podr?an ser ?tiles a la hora de desarrollar nuevos m?dulos.

Antes de introducir los m?dulos, se especifican los tipos de iteradores que se van a utilizar. Esta especificaci?n es auxiliar, para simplificar las precondiciones y postcondiciones de los algoritmos que utilizan iteradores. Luego, se presentan todos los m?dulos, con su interfaz, representaci?n y c?lculos de complejidad.

NOTA: Este apunte no est? terminado. Adem?s de ser incompleto (faltan los algoritmos y los c?lculos de complejidad de todos los m?dulos), puede tener (mejor dicho, tiene) errores y podr?a sufrir cambios en cualquier momento.

2. TADs para especificar iteradores

En esta secci?n se describen los TADs que utilizamos en la materia para especificar los iteradores. Los mismos no son m?s que un conjunto de funciones auxiliares que sirven para especificar las precondiciones y postcondiciones de las funciones que involucran iteradores. La forma de especificar estos iteradores es "envolviendo" una estructura que representa el concepto de ordenamiento de los valores contenidos. En este sentido, la especificaci?n de los iteradores con TADs podr?a evitarse, pero lo incluimos para simplificar la especificaci?n de los m?dulos.

2.1. TAD Iterador Unidireccional(α)

El iterador unidireccional permite recorrer los elementos una ?nica vez, avanzando continuamente. Es el tipo de iterador m?s simple que se puede especificar y no permite modificar la estructura iterada. Como la idea es convertir cualquier estructura en una secuencia, es razonable que este iterador tome una secuencia en la parte de especificaci?n. La idea final es que esta secuencia describa el orden en el que se recorrer?n los elementos de la estructura, i.e., esta secuencia es una "permutaci?n" de la estructura iterada.

TAD Iterador Unidireccional(α)

parámetros formales
g?neros α

géneros itUni(α)

igualdad observacional
$(\forall it_1, it_2 : \text{it}(\alpha))\ (it_1 =_{\text{obs}} it_2 \iff (\text{Siguientes}(it_1) =_{\text{obs}} \text{Siguientes}(it_2)))$

observadores básicos
Siguientes : itUni(α) $\longrightarrow$ secu(α)

generadores
CrearItUni : secu(α) $\longrightarrow$ itUni(α)

otras operaciones

HayMas?	: itUni(α)	$\longrightarrow$	bool	
Actual	: itUni(α) it	$\longrightarrow$	α	{HayMas?(it)}
Avanzar	: itUni(α) it	$\longrightarrow$	itUni(α)	{HayMas?(it)}

axiomas

Siguientes(CrearItUni(i))	$\equiv$	i
HayMas?(it)	$\equiv$	$\neg$Vacia?(Siguientes(it))
Actual(it)	$\equiv$	Prim(Siguientes(it))
Avanzar(it)	$\equiv$	CrearItUni(Fin(Siguientes(it)))

Fin TAD

2.2. TAD Iterador Unidireccional Modificable(α)

El iterador unidireccional modificable es una extensi?n del iterador unidireccional que permite realizar algunas operaciones de modificaci?n sobre los elementos de la estructura recorrida. Para poder especificar las modificaciones a la estructura iterada, se guarda la secuencia de los elementos que ya fueron recorridos. Observar que para especificar

los efectos secundarios que tienen estas modificaciones en el tipo iterado, hay que aclarar c?mo es el aliasing entre el iterador y el tipo iterado en el m?dulo correspondiente.

TAD ITERADOR UNIDIRECCIONAL MODIFICABLE(α)

parámetros formales
g?neros α

géneros itMod(α)

igualdad observacional

$$(\forall it_1, it_2 : \text{itMod}(\alpha)) \left(it_1 =_{\text{obs}} it_2 \iff \begin{pmatrix} \text{Anteriores}(it_1) =_{\text{obs}} \text{Anteriores}(it_2) \wedge \\ \text{Siguientes}(it_1) =_{\text{obs}} \text{Siguientes}(it_2) \end{pmatrix} \right)$$

observadores básicos

Anteriores	: itMod(α)	$\longrightarrow$	secu(α)	
Siguientes	: itMod(α)	$\longrightarrow$	secu(α)	

generadores

CrearItMod	: secu(α) $\times$ secu(α)	$\longrightarrow$	itMod(α)	

otras operaciones

SecuSuby	: itMod(α)	$\longrightarrow$	secu(α)	
HayMas?	: itMod(α)	$\longrightarrow$	bool	
Actual	: itMod(α) it	$\longrightarrow$	α	{HayMas?(it)}
Avanzar	: itMod(α) it	$\longrightarrow$	itMod(α)	{HayMas?(it)}
Eliminar	: itMod(α) it	$\longrightarrow$	itMod(α)	{HayMas?(it)}
Agregar	: itMod(α) $\times$ α	$\longrightarrow$	itMod(α)	

axiomas

Anteriores(CrearItMod(i, d))	$\equiv$	i
Siguientes(CrearItMod(i, d))	$\equiv$	d
SecuSuby(it)	$\equiv$	Anteriores(it) & Siguientes(it)
HayMas?(it)	$\equiv$	$\neg$Vacia?(Siguientes(it))
Actual(it)	$\equiv$	Prim(Siguientes(it))
Avanzar(it)	$\equiv$	CrearItMod(Anteriores(it) $\circ$ Actual(it), Fin(Siguientes(it)))
Eliminar(it)	$\equiv$	CrearItMod(Anteriores(it), Fin(Siguientes(it)))
Agregar(it, a)	$\equiv$	CrearItMod(Anteriores(it) $\circ$ a, Siguientes(it))

Fin TAD

2.3. ITERADOR BIDIRECCIONAL(α)

El iterador bidireccional es una generalizaci?n del iterador unidireccional modificable. El mismo permite recorrer los elementos avanzando y retrocediendo. Si bien se podr?a hacer una versi?n de iterador bidireccional no modificable, la especificaci?n de ambas es similar. Cuando se utilice en un m?dulo que no permita algunas modificaciones, simplemente se puede omitir el dise?o de las funciones que realizan estas modificaciones (ver e.g., m?dulo Conjunto Lineal). Por este motivo, optamos s?lo por la versi?n modificable.

Para que el iterador bidireccional sea lo mas sim?trico posible, cambiamos la operaci?n actual por dos: anterior y siguiente. La idea conceptual es pensar que el iterador est? posicionado en el medio de dos posiciones, y puede acceder tanto a la anterior como a la siguiente. Obviamente, la implementaci?n puede diferir de esta visi?n conceptual.

TAD ITERADOR BIDIRECCIONAL(α)

parámetros formales
g?neros α

géneros itBi(α)

igualdad observacional

$$(\forall it_1, it_2 : \text{itBi}(\alpha)) \left(it_1 =_{\text{obs}} it_2 \iff \begin{pmatrix} \text{Anteriores}(it_1) =_{\text{obs}} \text{Anteriores}(it_2) \wedge \\ \text{Siguientes}(it_1) =_{\text{obs}} \text{Siguientes}(it_2) \end{pmatrix} \right)$$

observadores básicos

Anteriores	: itBi(α)	$\longrightarrow$	secu(α)	
Siguientes	: itBi(α)	$\longrightarrow$	secu(α)	

generadores

CrearItBi : secu(α) $\times$ secu(α) $\longrightarrow$ itBi(α)

otras operaciones

SecuSuby	: itBi(α)	$\longrightarrow$ secu(α)	
HayAnterior?	: itBi(α)	$\longrightarrow$ bool	
Anterior	: itBi(α) it	$\longrightarrow$ α	{HayAnterior?(it)}
Retroceder	: itBi(α) it	$\longrightarrow$ itBi(α)	{HayAnterior?(it)}
HaySiguiente?	: itBi(α)	$\longrightarrow$ bool	
Siguiente	: itBi(α) it	$\longrightarrow$ α	{HaySiguiente?(it)}
Avanzar	: itBi(α) it	$\longrightarrow$ itBi(α)	{HaySiguiente?(it)}
EliminarSiguiente	: itBi(α) it	$\longrightarrow$ itBi(α)	{HaySiguiente?(it)}
EliminarAnterior	: itBi(α) it	$\longrightarrow$ itBi(α)	{HayAnterior?(it)}
AgregarComoAnterior	: itBi(α) $\times$ α	$\longrightarrow$ itBi(α)	
AgregarComoSiguiente	: itBi(α) $\times$ α	$\longrightarrow$ itBi(α)	

axiomas

Anteriores(CrearItBi(i, d))	$\equiv$	i
Siguientes(CrearItBi(i, d))	$\equiv$	d
SecuSuby(it)	$\equiv$	Anteriores(i) & Siguientes(d)
HayAnterior?(it)	$\equiv$	$\neg$Vacia?(Anteriores(it))
Anterior(it)	$\equiv$	Ult(Anteriores(it))
Retroceder(it)	$\equiv$	CrearItBi(Com(Anteriores(it)), Anterior(it) $\bullet$ Siguientes(it))
HaySiguiente?(it)	$\equiv$	$\neg$Vacia?(Siguientes(it))
Siguiente(it)	$\equiv$	Prim(Siguientes(it))
Avanzar(it)	$\equiv$	CrearItBi(Anteriores(it) $\circ$ Siguiente(it), Fin(Siguientes(it)))
EliminarSiguiente(it)	$\equiv$	CrearItBi(Anteriores(it), Fin(Siguientes(it)))
EliminarAnterior(it)	$\equiv$	CrearItBi(Com(Anteriores(it)), Siguientes(it))
AgregarComoAnterior(it, a)	$\equiv$	CrearItBi(Anteriores(it) $\circ$ a, Siguientes(it))
AgregarComoSiguiente(it, a)	$\equiv$	CrearItBi(Anteriores(it), a $\bullet$ Siguientes(it))
SecuSuby(it)	$\equiv$	Anteriores(it) & Siguientes(it)

Fin TAD

3. Invariantes de aliasing

Para simplificar la descripci?n del aliasing entre dos variables, vamos a definir un "metapredicado". Este metapredicado, llamado *alias*, lo vamos a utilizar para describir aquellas variables que comparten memoria en la ejecuci?n del programa. Si bien el metapredicado alias no es parte del lenguaje de TADs y no lo describimos en l?gica de primer orden, lo vamos a utilizar en las precondiciones y postcondiciones de las funciones. En esta secci?n vamos a describir su sem?ntica en castellano.

Alias es un metapredicado con un ?nico par?metro ϕ que puede ser una expresi?n booleana del lenguaje de TADs o un predicado en l?gica de primer orden. Este param?tro ϕ involucrar? un conjunto V con dos o m?s variables del programa. El significado es que las variables de V satisfacen ϕ durante la ejecuci?n del resto del programa, siempre y cuando dichas variables no sean asignadas con otro valor. En particular, el invariante puede dejar de satisfacerse cuando una variable de V se indefine. Una variable se indefine, cuando el valor al que hace referencia deja de ser valido. Esto ocurre principalmente cuando se elimina un elemento que est? siendo iterado.

Por ejemplo, supongamos que s y t son dos variables de tipo α. Si escribimos

$$\text{alias}(s = t),$$

lo que significa informalmente es que s y t comparten la misma posici?n de memoria. Un poco m?s rigurosamente, lo que significa es que cualquier modificaci?n que se realice a s afecta a t y viceversa, de forma tal que $s = t$, mientras a s y a t no se les asigne otro valor.

El ejemplo anterior es un poco b?sico. Supongamos ahora que tenemos dos variables s y c de tipos secu(α) y conj(α), respectivamente. Si escribimos

$$\text{alias}(\text{esPermutacion}(s, c)),$$

estamos diciendo que s y c comparten la misma memoria de forma tal que cualquier modificaci?n sobre s afecta a c y viceversa, de forma tal que se satisface esPermutacion(s, c). En particular, si se agrega un elemento a a c, se obtiene que la secuencia s se modifica de forma tal que resulta una permutaci?n de $c \cup \{a\}$. Notemos que, en particular, s

podr?a cambiar a cualquier permutaci?n, salvo que se indique lo contrario. De la misma forma, si se eliminara un elemento a de s, entonces c tambien se ver?a afectado de forma tal que s sea una permutaci?n de c. En particular, c pasar?a a ser $c \setminus \{a\}$.

Debemos observar que este invariante no es magico, sino que es una declaraci?n como cualquier otra, y el programado debe asegurarse que este invariante se cumpla. En particular, en el ejemplo anterior, no deberiamos permitir la inserci?n de elementos repetido en s, ya que dejar?a de ser una posible permutaci?n de un conjunto.

4. Modulo Mapa

Interfaz

se explica con: HABITACION.

generos: `mapa`.

Operaciones basicas de mapa

NUEVOMAPA(**in** n : `nat`) $\rightarrow$ res : `mapa`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} nuevaHab(n)\}$
Complejidad: $\Theta(n^2)$
Descripci?n: genera un mapa de tamano n x n.

OCUPAR(**in** m : `mapa`, **in** c : $tupla(int, int)$) $\rightarrow$ res : `mapa`
Pre $\equiv$ $\{c \in casilleros(m) \wedge_{\text{L}} libre(m, c) \wedge alcanzan(libres(m) - c, libres(m) - c, m)\}$
Post $\equiv$ $\{res =_{\text{obs}} ocupar(c, h)\}$
Complejidad: $\Theta(1)$
Descripci?n: ocupa una posicion del mapa siempre y cuando este no deje de ser conexo.

TAM(**in** m : `mapa`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} tam(m)\}$
Complejidad: $\Theta(1)$
Descripci?n: devuelve el tamano del mapa.

LIBRE(**in** m : `mapa`, **in** c : $tupla(int, int)$) $\rightarrow$ res : `bool`
Pre $\equiv$ $\{c \in casilleros(m)\}$
Post $\equiv$ $\{res =_{\text{obs}} libre(c, m)\}$
Complejidad: $\Theta(1)$
Descripci?n: devuelve si un elemento esta ocupado.

Representaci?n

Representaci?n de la lista

El objetivo de este m?dulo es implementar una lista doblemente enlazada con punteros al principio y al fin. Para simplificar un poco el manejo de la estructura, vamos a reemplazarla por una lista circular, donde el siguiente del ?ltimo apunta al primero y el anterior del primero apunta al ?ltimo. La estructura de representaci?n, su invariante de representaci?n y su funci?n de abstracci?n son las siguientes.

`mapa` **se representa con** `m`

donde `m` es `tupla`(*tamano*: `nat`, *casilleros*: `vec(vec(bool))`)

Rep : lst $\longrightarrow$ bool
Rep(l) $\equiv$ true $\iff$ (l.primero = NULL) = (l.longitud = 0) $\wedge_{\text{L}}$ (l.longitud $\neq$ 0 $\Rightarrow_{\text{L}}$
Nodo(l, l.longitud) = l.primero $\wedge$
($\forall i$: nat)(Nodo(l,i)$\rightarrow$siguiente = Nodo(l,$i+1$)$\rightarrow$anterior) $\wedge$
($\forall i$: nat)($1 \leq i < l$.longitud $\Rightarrow$ Nodo(l,i) $\neq$ l.primero)

Abs : mapa m $\longrightarrow$ hab {Rep(m)}

Abs(m) $\equiv$ $m.tamano =_{obs} tam(h) \wedge_L (\forall t$: tuple(nat,nat))$(0 \leq \Pi_1(t), \Pi_2(t) < m.tamano - 1 \Rightarrow_L$ libre(m, t) $=_{obs}$ $m.casilleros[\Pi_1(t)][\Pi_2(t)])$

Representaci?n del iterador

El iterador es simplemente un puntero al nodo siguiente. Este puntero apunta a NULL en el caso en que se lleg? al final de la lista. Por otra parte, el nodo anterior se obtiene accediendo al nodo siguiente y retrocediendo (salvo que el nodo siguiente sea el primer nodo). Para poder modificar la lista, tambien hay que guardar una referencia a la lista que est? siendo iterada. Adem?s, de esta forma podemos saber si el iterador apunta al primero o no.

`itLista(`α`)` se representa con `iter`

donde `iter` es `tupla(`*siguiente*: `puntero(nodo)`, *lista*: `puntero(lst)`)

Rep : iter $\longrightarrow$ bool
Rep(it) $\equiv$ true $\iff$ Rep($*(it$.lista)) $\wedge_L$ (it.siguiente = NULL $\vee_L$ $(\exists i$: nat)(Nodo($*it$.lista, i) = it.siguiente)

Abs : iter it $\longrightarrow$ itBi(α) {Rep(it)}
Abs(it) $=_{obs}$ b: itBi(α) | Siguientes(b) = Abs(Sig(it.lista, it.siguiente)) $\wedge$
Anteriores(b) = Abs(Ant(it.lista, it.siguiente))

Sig : puntero(lst) l $\times$ puntero(nodo) p $\longrightarrow$ lst {Rep($\langle l, p\rangle$)}
Sig(i, p) $\equiv$ Lst(p, l→longitud − Pos($*l$, p))

Ant : puntero(lst) l $\times$ puntero(nodo) p $\longrightarrow$ lst {Rep($\langle l, p\rangle$)}
Ant(i, p) $\equiv$ Lst(**if** $p = l$→primero **then** NULL **else** l→primero **fi**, Pos($*l$, p))

Nota: cuando p = NULL, Pos devuelve la longitud de la lista, lo cual est? bien, porque significa que el iterador no tiene siguiente.

Pos : lst l $\times$ puntero(nodo) p $\longrightarrow$ puntero(nodo) {Rep($\langle l, p\rangle$)}
Pos(l,p) $\equiv$ **if** l.primero = $p \vee l$.longitud = 0 **then** 0 **else** 1 + Pos(FinLst(l), p) **fi**

Algoritmos

En esta secci?n se hace abuso de notaci?n en los c?lculos de ?lgebra de ?rdenes presentes en la justificaciones de los algoritmos. La operaci?n de suma "+" denota secuencializaci?n de operaciones con determinado orden de complejidad, y el s?mbolo de igualdad "=" denota la pertenencia al orden de complejidad resultante.

Algoritmos del m?dulo

---

iVac?a() $\rightarrow$ res : lst
1: $res \leftarrow \langle NULL, 0\rangle$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

iAgregarAdelante(**in/out** l : `lst`, **in** a : α) $\rightarrow$ res : $iter$
$it \leftarrow CrearIt(l)$ ▷ $\Theta(1)$
$AgregarComoSiguiente(it, a)$ ▷ $\Theta(copy(a))$
$res \leftarrow it$ ▷ $\Theta(1)$

Complejidad: $\Theta(copy(a))$
Justificaci?n: El algoritmo tiene llamadas a funciones con costo $\Theta(1)$ y $\Theta(copy(a))$. Aplicando ?lgebra de ?rdenes: $\Theta(1) + \Theta(1) + \Theta(copy(a)) = \Theta(copy(a))$

---

iAgregarAtras(**in/out** l : `lst`, **in** $a : \alpha$) $\rightarrow$ res : $iter$

1: $it \leftarrow CrearItUlt(l)$ ▷ $\Theta(1)$
2: $AgregarComoSiguiente(it, a)$ ▷ $\Theta(copy(a))$
3: $res \leftarrow it$ ▷ $\Theta(1)$

Complejidad: $\Theta(copy(a))$
Justificaci?n: El algoritmo tiene llamadas a funciones con costo $\Theta(1)$ y $\Theta(copy(a))$. Aplicando ?lgebra de ?rdenes: $\Theta(1)+\Theta(copy(a))+\Theta(1) = \Theta(copy(a))$

---

iEsVac?a?(**in** l : `lst`) $\rightarrow$ res : $bool$

1: $res \leftarrow (l.primero = NULL)$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

iFin(**in/out** l : `lst`)

1: $CrearIt(l).EliminarSiguiente()$ ▷ $\Theta(1) + \Theta(1)$

Complejidad: $\Theta(1)$
Justificaci?n: $\Theta(1) + \Theta(1) = \Theta(1)$

---

iComienzo(**in/out** l : `lst`)

1: $CrearItUlt(l).EliminarAnterior()$ ▷ $\Theta(1) + \Theta(1)$

Complejidad: $\Theta(1)$
Justificaci?n: $\Theta(1) + \Theta(1) = \Theta(1)$

---

iPrimero(**in** l : `lst`) $\rightarrow$ res : α

1: $res \leftarrow CrearIt(l).Siguiente()$ ▷ $\Theta(1) + \Theta(1)$

Complejidad: $\Theta(1)$
Justificaci?n: $\Theta(1) + \Theta(1) = \Theta(1)$

---

i?ltimo(**in** l : `lst`) $\rightarrow$ res : α

1: $res \leftarrow CrearItUlt(l).Anterior()$ ▷ $\Theta(1) + \Theta(1)$

Complejidad: $\Theta(1)$
Justificaci?n: $\Theta(1) + \Theta(1) = \Theta(1)$

---

iLongitud(**in** l : `lst`) $\rightarrow$ res : nat

1: $res \leftarrow l.longitud$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

•[•](in l : lst, in i : nat) → res : α

1: $it \leftarrow CrearIt(l)$ ▷ $\Theta(1)$
2: $indice \leftarrow 0$ ▷ $\Theta(1)$
3: **while** $indice < i$ **do** ▷ $\Theta(i)$
4: $Avanzar(it)$ ▷ $\Theta(1)$
5: $indice \leftarrow indice + 1$ ▷ $\Theta(1)$
6: **end while**
7: $res \leftarrow Siguiente(it)$ ▷ $\Theta(1)$

Complejidad: $\Theta(i)$
Justificaci?n: El algoritmo tiene un ciclo que se va a repetir i veces. En cada ciclo se hacen realizan funciones con costo $\Theta(1)$. Aplicando ?lgebra de ?rdenes sabemos que el ciclo tiene un costo total del orden $\Theta(i)$. El costo total del algoritmo ser? de: $\Theta(1) + \Theta(1) + \Theta(i)*(\Theta(1)+\Theta(1)) + \Theta(1) = \Theta(i)$

---

iCopiar(in l : lst) → res : lst

1: $res \leftarrow Vacia()$ ▷ $\Theta(1)$
2: $it \leftarrow CrearIt(l)$ ▷ $\Theta(1)$
3: **while** $HaySiguiente(it)$ **do** ▷ $\Theta(long(l))$
4: $AgregarAtras(res, Siguiente(it))$ ▷ $\Theta(copy(Siguiente(it)))$
5: $Avanzar(it)$ ▷ $\Theta(1)$
6: **end while**

Complejidad: $\Theta\left(\sum_{i=1}^{long(l)} copy(l[i])\right)$
Justificaci?n: El algoritmo cuenta con un ciclo que se repetir? long(l) veces (recorre la lista entera). Por cada ciclo realiza una copia del elemento, el costo ser? el de copiar el elemento. Por lo tanto, el costo total del ciclo ser? la suma de copiar cada uno de los elementos de la lista. El resto de las llamadas a funciones tiene costo $\Theta(1)$. Por lo tanto el costo total es de: $\Theta(1) + \Theta(1) + \Theta(long(l))$ * $(\Theta(copy(Siguiente(it))) + \Theta(1)\ ) = \Theta\left(\sum_{i=1}^{long(l)} copy(l[i])\right)$

• $=_i$ •(in l_1 : lst, in l_2 : lst) → res : $bool$

1: $it_1 \leftarrow CrearIt(l_1)$ ▷ $\Theta(1)$
2: $it_2 \leftarrow CrearIt(l_2)$ ▷ $\Theta(1)$
3: **while** $HaySiguiente(it_1) \wedge HaySiguiente(it_2) \wedge Siguiente(it_1) = Siguiente(it_2)$ **do** ▷ [∗]
4: $Avanzar(it_1)$ // $\Theta(1)$
5: $Avanzar(it_2)$ // $\Theta(1)$
6: **end while**
7: $res \leftarrow \neg(HaySiguiente(it_1) \vee HaySiguiente(it_2))$ ▷ $\Theta(1) + \Theta(1)$

Complejidad: $\Theta\left(\sum_{i=1}^{\ell} equal(l_1[i], l_2[i])\right)$, donde $\ell = \text{mín}\{\text{long}(l_1), \text{long}(l_2)\}$. [∗]
Justificaci?n: [∗] Ya que continua hasta que alguna de las dos listas se acabe (la de menor longitud) y en cada ciclo compara los elementos de la lista.

Algoritmos del iterador

1: **iCrearIt**(in l : lst) → res : iter
2: $res \leftarrow \langle l.primero, l \rangle$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

1: **iCrearItUlt**(**in** l : `lst`) $\rightarrow$ res : iter
2: $res \leftarrow \langle NULL, l \rangle$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

1: **iHaySiguiente**(**in** it : `iter`) $\rightarrow$ res : $bool$
2: $res \leftarrow it.siguiente \neq NULL$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

1: **iHayAnterior**(**in** it : `iter`) $\rightarrow$ res : $bool$
2: $res \leftarrow it.siguiente \neq (it.lista \rightarrow primero)$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

1: **iSiguiente**(**in** it : `iter`) $\rightarrow$ res : α
2: $res \leftarrow (it.siguiente \rightarrow dato)$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

iAnterior(**in** it : `iter`) $\rightarrow$ res : α
1: $res \leftarrow (SiguienteReal(it) \rightarrow anterior \rightarrow dato)$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

1: **iAvanzar**(**in/out** it : `iter`)
2: $it.siguiente \leftarrow (it.siguiente \rightarrow siguiente)$ ▷ $\Theta(1)$
3: **if** $it.siguiente = it.lista \rightarrow primero$ **then** ▷ $\Theta(1)$
4: $it.siguiente \leftarrow NULL$
5: **end if**

Complejidad: $\Theta(1)$
Justificaci?n: $\Theta(1) + \Theta(1) = \Theta(1)$

1: **iRetroceder**(**in/out** it : `iter`)
2: $it.siguiente \leftarrow (SiguienteReal(it) \rightarrow anterior)$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

1: **iEliminarSiguiente**(**in**/**out** it : `iter`)
2: $puntero(nodo)$ $temp \leftarrow it.siguiente$
3: $(tmp \rightarrow siguiente \rightarrow anterior) \leftarrow (tmp \rightarrow anterior)$ ▷ Reencadenamos los nodos // $\Theta(1)$
4: $(tmp \rightarrow anterior \rightarrow siguiente) \leftarrow (tmp \rightarrow siguiente)$
5: **if** $(tmp \rightarrow siguiente) = (it.lista \rightarrow primero)$ **then** ▷ Si borramos el ?ltimo nodo, ya no hay siguiente // $\Theta(1)$
6: $it.siguiente \leftarrow NULL$
7: **else** ▷ Sino, avanzamos al siguiente // $\Theta(1)$
8: $it.siguiente \leftarrow (tmp \rightarrow siguiente)$
9: **end if**
10: **if** $tmp = (it.lista \rightarrow primero)$ **then** ▷ Si borramos el primer nodo, hay que volver a setear el primero // $\Theta(1)$
11: $(it.lista \rightarrow primero) \leftarrow it.siguiente$
12: **end if**
13: $tmp \leftarrow NULL$ ▷ Se libera la memoria ocupada por el nodo // $\Theta(1)$
14: $(it.lista \rightarrow longitud) \leftarrow (it.lista \rightarrow longitud) - 1$

Complejidad: $\Theta(1)$
Justificaci?n: $\Theta(1) + \Theta(1) + \Theta(1) + \Theta(1) + \Theta(1) = \Theta(1)$

1: **iEliminarAnterior**(**in**/**out** it : `iter`)
2: $Retroceder(it)$ ▷ $\Theta(1)$
3: $EliminarSiguiente(it)$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$
Justificaci?n: $\Theta(1) + \Theta(1) = \Theta(1)$

1: **iAgregarComoSiguiente**(**in**/**out** it : `iter`, **in** $a : \alpha$)
2: $AgregarComoAnterior(it, a)$ ▷ $\Theta(1)$
3: $Retroceder(it)$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$
Justificaci?n: $\Theta(1) + \Theta(1) = \Theta(1)$

1: **iAgregarComoAnterior**(**in**/**out** it : `iter`, **in** $a : \alpha$)
2: $puntero(nodo)$ $sig \leftarrow SiguienteReal(it)$
3: $puntero(nodo)$ $nuevo \leftarrow$ **&** $\langle a, NULL, NULL \rangle$ ▷ Reservamos memoria para el nuevo nodo // $\Theta(1)$
4: **if** $sig = NULL$ **then** ▷ Asignamos los punteros de acuerdo a si el nodo es el primero o no en la lista circular // $\Theta(1)$
5: $(nuevo \rightarrow anterior) \leftarrow nuevo$
6: $(nuevo \rightarrow siguiente) \leftarrow nuevo$
7: **else**
8: $(nuevo \rightarrow anterior) \leftarrow (sig \rightarrow anterior)$
9: $(nuevo \rightarrow siguiente) \leftarrow sig$
10: **end if**
11: $(nuevo \rightarrow anterior \rightarrow siguiente) \leftarrow nuevo$ ▷ Reencadenamos los otros nodos // $\Theta(1)$
12: **if** $it.siguiente = (it.lista \rightarrow primero)$ **then** ▷ Cambiamos el primero en caso de que estemos agregando el primero // $\Theta(1)$
13: $(it.lista \rightarrow primero) \leftarrow nuevo$
14: **end if**
15: $(it.lista \rightarrow longitud) \leftarrow (it.lista \rightarrow longitud) + 1$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$
Justificaci?n: $\Theta(1) + \Theta(1) + \Theta(1) + \Theta(1) = \Theta(1)$

iSiguienteReal(**in** *it* : `iter`) → *res* : *puntero*(*nodo*) ▷ Esta es una operaci?n privada que
if *it.siguiente* = *NULL* **then** ▷ devuelve el siguiente como lista circular // Θ(1)
    res ← (*it.lista* → *siguiente*)
else
    res ← *it.siguiente*
end if

Complejidad: Θ(1)

5. M?dulo Pila(α)

El m?dulo Pila provee una pila en la que s?lo se puede acceder al tope de la misma. Por este motivo, no incluye iteradores.

Para describir la complejidad de las operaciones, vamos a llamar $copy(a)$ al costo de copiar el elemento $a \in \alpha$ (i.e., *copy* es una funci?n de α en $\mathbb{N}$).[1]

Interfaz

par?metros formales
g?neros α
funci?n COPIAR(**in** a: α) $\to$ res : α
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{obs} a$}
Complejidad: $\Theta(copy(a))$
Descripci?n: funci?n de copia de α's

se explica con: PILA(α).

g?neros: `pila`(α).

VAC?A() $\to$ res : `pila`(α)
Pre $\equiv$ {}
Post $\equiv$ {$res =_{obs}$ vac?a}
Complejidad: $\Theta(1)$
Descripci?n: genera una pila vac?a.

APILAR(**in/out** p: `pila`(α), **in** a: α)
Pre $\equiv$ {$p =_{obs} p_0$}
Post $\equiv$ {$p =_{obs}$ apilar(p, a)}
Complejidad: $\Theta(copy(a))$
Descripci?n: apila a en p
Aliasing: el elemento a se apila por copia.

ESVACIA?(**in** p: `pila`(α)) $\to$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{obs}$ vacia?(p)}
Complejidad: $\Theta(1)$
Descripci?n: devuelve `true` si y s?lo si la pila no contiene elementos

TOPE(**in** p: `pila`(α)) $\to$ res : α
Pre $\equiv$ {$\neg$vac?a?(p)}
Post $\equiv$ {alias($res =_{obs}$ tope(p))}
Complejidad: $\Theta(1)$
Descripci?n: devuelve el tope de la pila.
Aliasing: res es modificable si y s?lo si p es modificable.

DESAPILAR(**in/out** p: `pila`(α)) $\to$ res : α
Pre $\equiv$ {$p =_{obs} p_0 \wedge \neg$vac?a?($p$)}
Post $\equiv$ {$p =_{obs}$ desapilar(p_0) $\wedge$ $res =_{obs}$ tope(p)}
Complejidad: $\Theta(1)$
Descripci?n: desapila el tope de p.

TAMA?O(**in** p: `pila`(α)) $\to$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{obs}$ tama?o(p)}
Complejidad: $\Theta(1)$
Descripci?n: devuelve la cantidad de elementos apilados en p.

[1] N?tese que este es un abuso de notaci?n, ya que no estamos describiendo *copy* en funci?n del tama?o de a. A la hora de usarlo, habr? que realizar la traducci?n

COPIAR(**in** p : `pila(`α`)`) $\to$ res : `pila(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} p\}$
Complejidad: $\Theta\left(\sum_{i=1}^{t} copy(p[i])\right) = O\left(t \max_{i=1}^{t} copy(p[i])\right)$, donde t = tama?o(p).
Descripci?n: genera una copia nueva de la pila

$\bullet = \bullet$(**in** p_1 : `pila(`α`)`, **in** p_2 : `pila(`α`)`) $\to$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} p_1 = p_2\}$
Complejidad: $\Theta\left(\sum_{i=1}^{t} equal(p_1[i], p_2[i])\right)$, donde $t = \text{mín}\{\text{tama?o}(p_1), \text{tama?o}(p_2)\}$.
Descripci?n: compara p_1 y p_2 por igualdad, cuando α posee operaci?n de igualdad.
Requiere: $\bullet = \bullet$(**in** $a_1 : \alpha$, **in** $a_2 : \alpha$) $\to$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} (a_1 = a_2)\}$
Complejidad: $\Theta(equal(a_1, a_2))$
Descripci?n: funci?n de igualdad de α's

Especificaci?n de las operaciones auxiliares utilizadas en la interfaz

TAD Pila Extendida(α)

extiende PILA(α)

otras operaciones (no exportadas)

$\bullet[\bullet]$: pila(α) $p \times$ nat i $\longrightarrow$ α $\{i <$ tama?o(p)$\}$

axiomas

$p[i]$ $\equiv$ **if** $i = 0$ **then** tope(p) **else** desapilar(p)$[i-1]$ **fi**

Fin TAD

Representaci?n

El objetivo de este m?dulo es implementar una pila lo m?s eficientemente posible, y eso se puede obtener utilizando una lista enlazada. Claramente, cualquier lista representa una pila, donde el tope se encuentra o en el primer o en el ?ltimo elemento. En este caso, elegimos que el tope se encuentre en el primer elemento.

`pila(`α`)` **se representa con** `lista(`α`)`

Rep : lista(α) $\longrightarrow$ bool
Rep(l) $\equiv$ true

Abs : lista(α) l $\longrightarrow$ pila(α) $\{$Rep(l)$\}$
Abs(l) $\equiv$ **if** vacia?(l) **then** vac?a **else** apilar(prim(l), Abs(fin(l))) **fi**

Algoritmos

6. M?dulo Cola(α)

El m?dulo Cola provee una cola en la que s?lo se puede acceder al proximo de la misma. Por este motivo, no incluye iteradores.

Para describir la complejidad de las operaciones, vamos a llamar $copy(a)$ al costo de copiar el elemento $a \in \alpha$ (i.e., *copy* es una funci?n de α en $\mathbb{N}$).[2]

Interfaz

[2] N?tese que este es un abuso de notaci?n, ya que no estamos describiendo *copy* en funci?n del tama?o de a. A la hora de usarlo, habr? que realizar la traducci?n

par?metros formales
g?neros α
funci?n COPIAR(**in** a : α) $\rightarrow$ res : α
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} a$}
Complejidad: $\Theta(copy(a))$
Descripci?n: funci?n de copia de α's

se explica con: COLA(α).

g?neros: `cola`(α).

VAC?A() $\rightarrow$ res : `cola`(α)
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ vac?a}
Complejidad: $\Theta(1)$
Descripci?n: genera una cola vac?a.

ENCOLAR(**in/out** c: `cola`(α), **in** a : α)
Pre $\equiv$ {$c =_{\text{obs}} c_0$}
Post $\equiv$ {$p =_{\text{obs}}$ encolar(c, a)}
Complejidad: $\Theta(copy(a))$
Descripci?n: encola a a c
Aliasing: el elemento a se encola por copia.

ESVACIA?(**in** c: `cola`(α)) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ vacia?(c)}
Complejidad: $\Theta(1)$
Descripci?n: devuelve `true` si y s?lo si la cola es vac?a.

PROXIMO(**in** c: `cola`(α)) $\rightarrow$ res : α
Pre $\equiv$ {ýac?a?(c)}
Post $\equiv$ {alias($res =_{\text{obs}}$ proximo(c))}
Complejidad: $\Theta(1)$
Descripci?n: devuelve el proximo de la cola.
Aliasing: res es modificable si y s?lo si p es modificable.

DESENCOLAR(**in/out** c: `cola`(α))
Pre $\equiv$ {$c =_{\text{obs}} c_0 \wedge \neg$vac?a?($c$)}
Post $\equiv$ {$c =_{\text{obs}}$ desacolar(c_0)}
Complejidad: $\Theta(1)$
Descripci?n: desencola el proximo de c.

TAMA?O(**in** c: `cola`(α)) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ tama?o(c)}
Complejidad: $\Theta(1)$
Descripci?n: devuelve la cantidad de elementos encolados en c.

COPIAR(**in** c: `cola`(α)) $\rightarrow$ res : `cola`(α)
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} c$}
Complejidad: $\Theta\left(\sum_{i=1}^{t} copy(c[i])\right)$, donde t = tama?o(c)
Descripci?n: genera una copia nueva de la cola

$\bullet = \bullet$(**in** c_1 : `cola`(α), **in** c_2 : `cola`(α)) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} c_1 = c_2$}

Complejidad: $\Theta\left(\sum_{i=1}^{t} equal(c_1[i], c_2[i])\right)$, donde $t = \text{mín}\{\text{tama?o}(c_1), \text{tama?o}(c_2)\}$.
Descripci?n: compara c_1 y c_2 por igualdad, cuando α posee operaci?n de igualdad.
Requiere: $\bullet = \bullet(\textbf{in } a_1 : \alpha, \textbf{ in } a_2 : \alpha) \to res$: `bool`
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} (a_1 = a_2)\}$
Complejidad: $\Theta(equal(a_1, a_2))$
Descripci?n: funci?n de igualdad de α's

Especificaci?n de las operaciones auxiliares utilizadas en la interfaz

TAD Cola Extendida(α)

extiende Cola(α)

otras operaciones (no exportadas)

$\bullet[\bullet]$: cola(α) $p \times$ nat $i \longrightarrow \alpha$ $\{i < \text{tama?o}(p)\}$

axiomas

$c[i] \equiv$ **if** $i = 0$ **then** proximo(c) **else** desencolar(c)$[i-1]$ **fi**

Fin TAD

Representaci?n

El objetivo de este m?dulo es implementar una cola lo m?s eficientemente posible, y eso se puede obtener utilizando una lista enlazada. Claramente, cualquier lista representa una cola, donde el proximo se encuentra o en el primer o en el ?ltimo elemento. En este caso, elegimos que el proximo se encuentre en el primer elemento.

`cola`(α) **se representa con** `lista`(α)

Rep : lista(α) $\longrightarrow$ bool
Rep(l) $\equiv$ true

Abs : lista(α) $l \longrightarrow$ cola(α) $\{\text{Rep}(l)\}$
Abs(l) $\equiv$ **if** vacia?(l) **then** vac?a **else** encolar(ult(l), Abs(com(l))) **fi**

Algoritmos

7. M?dulo Vector(α)

El m?dulo Vector provee una secuencia que permite obtener el i-?simo elemento de forma eficiente. La inserci?n de elementos es eficiente cuando se realiza al final de la misma, si se utiliza un an?lisis amortizado (i.e., n inserciones consecutivas cuestan $O(n)$), aunque puede tener un costo lineal en peor caso. La inserci?n en otras posiciones no es tan eficiente, ya que requiere varias copias de elementos. El borrado de los ?ltimos elementos es eficiente, no as? el borrado de los elementos intermedios.

Una consideraci?n a tener en cuenta, es que el espacio utilizado por la estructura es el m?ximo espacio utilizado en cualquier momento del programa. Es decir, si se realizan n inserciones seguidas de n borrados, el espacio utilizado es $O(n)$ por el espacio de cada α. Si fuera necesario borrar esta memoria, se puede crear una copia del vector con los elementos sobrevivientes, borrando la copia vieja.

En cuanto al recorrido de los elementos, como los mismos se pueden recorrer con un ?ndice, no se proveen iteradores.

Para describir la complejidad de las operaciones, vamos a llamar $copy(a)$ al costo de copiar el elemento $a \in \alpha$ (i.e., *copy* es una funci?n de α en $\mathbb{N}$), y vamos a utilizar

$$f(n) = \begin{cases} n & \text{si } n = 2^k \text{ para alg?n } k \\ 1 & \text{en caso contrario} \end{cases}$$

para describir el costo de inserci?n de un elemento. Vale la pena notar que $\sum_{i=1}^{n} \frac{f(j+i)}{n} \to 1$ cuando $n \to \infty$, para

todo $j \in \mathbb{N}$. En otras palabras, la inserci?n consecutiva de n elementos costar? $O(1)$ copias por elemento, en t?rminos asint?ticos.

Interfaz

par?metros formales
g?neros α
funci?n COPIAR(**in** $a : \alpha$) $\rightarrow$ res : α
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} a$}
Complejidad: $\Theta(copy(a))$
Descripci?n: funci?n de copia de α's.

se explica con: SECU(α).

g?neros: `vector`(α).

VAC?A() $\rightarrow$ res : `vector`(α)
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} <>$}
Complejidad: $\Theta(1)$
Descripci?n: genera un vector vac?o.

AGREGARATRAS(**in/out** v : `vector(`α`)`, **in** $a : \alpha$)
Pre $\equiv$ {$v =_{\text{obs}} v_0$}
Post $\equiv$ {$v =_{\text{obs}} v_0 \circ a$}
Complejidad: $\Theta(f(\text{long}(v)) + copy(a))$
Descripci?n: agrega el elemento a como ?ltimo elemento del vector.
Aliasing: el elemento a se agrega por copia. Cualquier referencia que se tuviera al vector queda invalidada cuando long(v) es potencia de 2.

ESVAC?O?(**in** v : `vector(`α`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ vacia?(v)}
Complejidad: $\Theta(1)$
Descripci?n: devuelve `true` si y s?lo si v esta vac?o

COMIENZO(**in/out** v : `vector(`α`)`)
Pre $\equiv$ {$v =_{\text{obs}} v_0 \wedge \neg$vac?a?($v$)}
Post $\equiv$ {$v =_{\text{obs}}$ com(v_0)}
Complejidad: $\Theta(1)$
Descripci?n: elimina el ?ltimo elemento de v.

TOMARPRIMEROS(**in/out** v : `vector(`α`)`, **in** n : `nat`)
Pre $\equiv$ {$v =_{\text{obs}} v_0$}
Post $\equiv$ {$v =_{\text{obs}}$ Tomar(v_0, n)}
Complejidad: $\Theta(1)$
Descripci?n: elimina los ?ltimos máx{long(v) - n, 0} elementos del vector, i.e., se queda con los primeros n elementos del vector.

TIRARULTIMOS(**in/out** v : `vector(`α`)`, **in** n : `nat`)
Pre $\equiv$ {$v =_{\text{obs}} v_0$}
Post $\equiv$ {$v =_{\text{obs}}$ Tomar(v_0, long(v_0) - n)}
Complejidad: $\Theta(1)$
Descripci?n: elimina los ?ltimos máx{long(v), n} elementos del vector.

ULTIMO(**in** v : `vector(`α`)`) $\rightarrow$ res : α
Pre $\equiv$ {$\neg$vac?a?(v)}
Post $\equiv$ {alias($res =_{\text{obs}}$ ult(v))}
Complejidad: $\Theta(1)$
Descripci?n: devuelve el ?ltimo elemento del vector.

Aliasing: res es modificable si y s?lo si v es modificable.

LONGITUD(**in** l : `vector(`α`)`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} \text{long}(v)$}
Complejidad: $\Theta(1)$
Descripci?n: devuelve la cantidad de elementos que contiene el vector.

$\bullet[\bullet]$(**in** v : `vector(`α`)`, **in** i : `nat`) $\rightarrow$ res : α
Pre $\equiv$ {$i < \text{long}(v)$}
Post $\equiv$ {alias($res =_{\text{obs}} \text{iesimo}(v, i)$)}
Complejidad: $\Theta(1)$
Descripci?n: devuelve el elemento que se encuentra en la i-?sima posici?n del vector en base 0. Es decir, $v[i]$ devuelve el elemento que se encuentra en la posici?n $i + 1$.
Aliasing: res es modificable si y s?lo si v es modificable.

AGREGAR(**in/out** v : `vector(`α`)`, **in** i : `nat`, **in** a : α)
Pre $\equiv$ {$v =_{\text{obs}} v_0 \wedge i \leq \text{long}(v)$}
Post $\equiv$ {$v =_{\text{obs}} \text{Agregar}(v, i, a)$}
Complejidad: $\Theta(f(\text{long}(v)) + \text{long}(v) - i + \mathit{copy}(a))$
Descripci?n: agrega el elemento a a v, de forma tal que ocupe la posici?n i.
Aliasing: el elemento a se agrega por copia. Cualquier referencia que se tuviera al vector queda invalidada cuando $\text{long}(v)$ es potencia de 2.

ELIMINAR(**in/out** v : `vector(`α`)`, **in** i : `nat`)
Pre $\equiv$ {$v =_{\text{obs}} v_0 \wedge i < \text{long}(v)$}
Post $\equiv$ {$v =_{\text{obs}} \text{Eliminar}(v, i)$}
Complejidad: $\Theta(\text{long}(v) - i)$
Descripci?n: elimina el elemento que ocupa la posici?n i de v.

COPIAR(**in** v : `vector(`α`)`) $\rightarrow$ res : `vector(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} v$}
Complejidad: $\Theta\left(\sum_{i=1}^{\ell} \mathit{copy}(v[i])\right)$, donde $\ell = \text{long}(v)$.
Descripci?n: genera una copia nueva del vector.

$\bullet = \bullet$(**in** v_1 : `vector(`α`)`, **in** v_2 : `vector(`α`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} v_1 = v_2$}
Complejidad: $\Theta\left(\sum_{i=1}^{\ell} \mathit{equal}(v_1[i], v_2[i])\right)$, donde $\ell = \text{mín}\{\text{long}(v_1), \text{long}(v_2)\}$.
Descripci?n: compara v_1 y v_2 por igualdad, cuando α posee operaci?n de igualdad.
Requiere: $\bullet = \bullet$(**in** a_1 : α, **in** a_2 : α) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} (a_1 = a_2)$}
Complejidad: $\Theta(\mathit{equal}(a_1, a_2))$
Descripci?n: funci?n de igualdad de α's

Especificaci?n de las operaciones auxiliares utilizadas en la interfaz

TAD Secuencia Extendida(α)

extiende SECUENCIA(α)

otras operaciones (exportadas)

Agregar	:	secu(α) s × nat i × α a	$\longrightarrow$	secu(α)	$\{i \leq \text{long}(s)\}$
Eliminar	:	secu(α) s × nat i	$\longrightarrow$	secu(α)	$\{i < \text{long}(s)\}$
Tomar	:	secu(α) × nat	$\longrightarrow$	secu(α)	

otras operaciones (no exportadas)

Tirar : secu(α) $\times$ nat $\longrightarrow$ secu(α)

axiomas

Agregar(s, i, a) $\equiv$ (Tomar(n, i) $\circ$ a) & Tirar(n, i)
Eliminar(s, i, a) $\equiv$ (Tomar(n, $i-1$) & Tirar(n, i)
Tomar(s, n) $\equiv$ **if** $n = 0 \vee$ vacia?(s) **then** <> **else** prim(s) $\bullet$ Tomar(fin(s), $n-1$) **fi**
Tirar(s, n) $\equiv$ **if** $n = 0 \vee$ vacia?(s) **then** s **else** Tirar(fin(s), $n-1$) **fi**

Fin TAD

Representaci?n

La idea de este m?dulo es tener una lista donde el i-?simo se puede obtener en tiempo $O(1)$. Para esto, necesitamos usar alg?n tipo de acceso aleatorio a los elementos, que se consigue utilizando un arreglo. Ademas, necesitamos que el agregado de elementos tome $O(1)$ copias cuando analizamos el tiempo amortizado, i.e., $O(f(n))$ copias. Para lograr esto, podemos duplicar el tama?o del arreglo cuando este se llena.

`vector`(α) **se representa con** `vec`

donde `vec` es `tupla`(*elementos*: `arreglo_dimensionable de puntero`(α), *longitud*: `nat`)

Rep : vec $\longrightarrow$ bool
Rep(v) $\equiv$ true $\iff$ ($\exists k$: nat)(tam(v.elementos) = $2^k \wedge v$.longitud $\leq$ tam(v.elementos) $\wedge$
($\forall i$: nat)($0 \leq i < v$.longitud $\Rightarrow_{\text{L}}$ def?(v.elementos, i)) $\wedge$
($\forall i, j$: nat)($0 \leq i < j < v$.longitud $\Rightarrow_{\text{L}}$ v.elementos[i] $\neq$ v.elementos[j]))

Abs : vec v $\longrightarrow$ secu(α) {Rep(v)}
Abs(v) $\equiv$ **if** v.longitud = 0 **then**
<>
else
Abs($\langle v$.elementos, v.longitud $- 1\rangle$) $\circ$ $*$(v.elementos[v.longitud $- 1$])
fi

Algoritmos

8. M?dulo Diccionario Lineal(κ, σ)

El m?dulo Diccionario Lineal provee un diccionario b?sico en el que se puede definir, borrar, y testear si una clave est? definida en tiempo lineal. Cuando ya se sabe que la clave a definir no esta definida en el diccionario, la definici?n se puede hacer en tiempo $O(1)$.

En cuanto al recorrido de los elementos, se provee un iterador bidireccional que permite recorrer y eliminar los elementos de d como si fuera una secuencia de pares κ, σ.

Para describir la complejidad de las operaciones, vamos a llamar $copy(k)$ al costo de copiar el elemento $k \in \kappa \cup \sigma$ y $equal(k_1, k_2)$ al costo de evaluar si dos elementos $k_1, k_2 \in \kappa$ son iguales (i.e., *copy* y *equal* son funciones de $\kappa \cup \sigma$ y $\kappa \times \kappa$ en $\mathbb{N}$, respectivamente).[3]

Interfaz

par?metros formales
g?neros κ, σ

funci?n $\bullet = \bullet$(**in** $k_1 : \kappa$, **in** $k_2 : \kappa$) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} (k_1 = k_2)$}
Complejidad: $\Theta(equal(k_1, k_2))$
Descripci?n: funci?n de igualdad de κ's

funci?n COPIAR(**in** $k : \kappa$) $\rightarrow$ res : κ
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} k$}
Complejidad: $\Theta(copy(k))$
Descripci?n: funci?n de copia de κ's

[3] N?tese que este es un abuso de notaci?n, ya que no estamos describiendo *copy* y *equal* en funci?n del tama?o de k. A la hora de usarlo, habr? que realizar la traducci?n.

funci?n COPIAR(**in** $s : \sigma$) $\to$ *res* $: \sigma$
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} s$}
Complejidad: $\Theta(copy(s))$
Descripci?n: funci?n de copia de σ's

se explica con: DICCIONARIO(κ, σ), ITERADOR BIDIRECCIONAL(TUPLA(κ, σ)).

g?neros: `dicc`(κ, σ), `itDicc`(κ, σ).

Operaciones b?sicas de diccionario

VAC?O() $\to$ *res* : `dicc`(κ, σ)
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ vacio}
Complejidad: $\Theta(1)$
Descripci?n: genera un diccionario vac?o.

DEFINIR(**in/out** d: `dicc`(κ, σ), **in** $k : \kappa$, **in** $s : \sigma$) $\to$ *res* : `itDicc`(κ, σ)
Pre $\equiv$ {$d =_{\text{obs}} d_0$}
Post $\equiv$ {$d =_{\text{obs}}$ definir(d, k, s) $\wedge$ haySiguiente(res) $\wedge_L$ Siguiente(res) $= \langle k, s \rangle \wedge$ alias(esPermutaci?n(SecuSuby(res), d))}
Complejidad: $\Theta\left(\sum_{k' \in K} equal(k, k') + copy(k) + copy(s)\right)$, donde $K =$ claves(d)
Descripci?n: define la clave k con el significado s en el diccionario. Retorna un iterador al elemento reci?n agregado.
Aliasing: los elementos k y s se definen por copia. El iterador se invalida si y s?lo si se elimina el elemento siguiente del iterador sin utilizar la funci?n ELIMINARSIGUIENTE. Adem?s, anteriores(res) y siguientes(res) podr?an cambiar completamente ante cualquier operaci?n que modifique el d sin utilizar las funciones del iterador.

DEFINIRRAPIDO(**in/out** d: `dicc`(κ, σ), **in** $k : \kappa$, **in** $s : \sigma$) $\to$ *res* : `itDicc`(κ, σ)
Pre $\equiv$ {$d =_{\text{obs}} d_0 \wedge \neg$definido?($d$, k)}
Post $\equiv$ {$d =_{\text{obs}}$ definir(d, k, s) $\wedge$ haySiguiente(res) $\wedge_L$ Siguiente(res) $= \langle k, s \rangle \wedge$ esPermutaci?n(SecuSuby(res), d)}
Complejidad: $\Theta(copy(k) + copy(s))$
Descripci?n: define la clave $k \notin$ claves(d) con el significado s en el diccionario. Retorna un iterador al elemento reci?n agregado.
Aliasing: los elementos k y s se definen por copia. El iterador se invalida si y s?lo si se elimina el elemento siguiente del iterador sin utilizar la funci?n ELIMINARSIGUIENTE. Adem?s, anteriores(res) y siguientes(res) podr?an cambiar completamente ante cualquier operaci?n que modifique el d sin utilizar las funciones del iterador.

DEFINIDO?(**in** d: `dicc`(κ, σ), **in** $k : \kappa$) $\to$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ def?(d, k)}
Complejidad: $\Theta(\sum_{k' \in K} equal(k, k'))$, donde $K =$ claves(d)
Descripci?n: devuelve `true` si y s?lo k est? definido en el diccionario.

SIGNIFICADO(**in** d: `dicc`(κ, σ), **in** $k : \kappa$) $\to$ *res* $: \sigma$
Pre $\equiv$ {def?(d, k)}
Post $\equiv$ {alias($res =_{\text{obs}}$ Significado(d, k))}
Complejidad: $\Theta(\sum_{k' \in K} equal(k, k'))$, donde $K =$ claves(d)
Descripci?n: devuelve el significado de la clave k en d.
Aliasing: res es modificable si y s?lo si d es modificable.

BORRAR(**in/out** d: `dicc`(κ, σ), **in** $k : \kappa$)
Pre $\equiv$ {$d = d_0 \wedge$ def?(d, k)}
Post $\equiv$ {$d =_{\text{obs}}$ borrar(d_0, k)}
Complejidad: $\Theta(\sum_{k' \in K} equal(k, k'))$, donde $K =$ claves(d)
Descripci?n: elimina la clave k y su significado de d.

#Claves(**in** d: `dicc(`κ, σ`)`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ #claves(d)}
Complejidad: $\Theta(1)$
Descripci?n: devuelve la cantidad de claves del diccionario.

Copiar(**in** d: `dicc(`κ, σ`)`) $\rightarrow$ res : `dicc(`κ, σ`)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} d$}
Complejidad: $\Theta\left(\sum_{k \in K} (copy(k) + copy(\text{significado}(k, d)))\right)$, donde $K = \text{claves}(d)$
Descripci?n: genera una copia nueva del diccionario.

$\bullet = \bullet$(**in** d_1 : `dicc(`κ, σ`)`, **in** d_2 : `dicc(`κ, σ`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} c_1 = c_2$}
Complejidad: $O\left(\sum_{\substack{k_1 \in K_1 \\ k_2 \in K_2}} equal(\langle k_1, s_1 \rangle, \langle k_2, s_2 \rangle)\right)$, donde $K_i = \text{claves}(d_i)$ y $s_i = \text{significado}(d_i, k_i)$, $i \in \{1, 2\}$.
Descripci?n: compara d_1 y d_2 por igualdad, cuando σ posee operaci?n de igualdad.
Requiere: $\bullet = \bullet$(**in** $s_1 : \sigma$, **in** $s_2 : \sigma$) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} (s_1 = s_2)$}
Complejidad: $\Theta(equal(s_1, s_2))$
Descripci?n: funci?n de igualdad de σ's

Operaciones del iterador

El iterador que presentamos permite modificar el diccionario recorrido, eliminando elementos. Sin embargo, cuando el diccionario es no modificable, no se pueden utilizar las funciones de eliminaci?n. Adem?s, las claves de los elementos iterados no pueden modificarse nunca, por cuestiones de implementaci?n. Cuando d es modificable, decimos que it es modificable.

Para simplificar la notaci?n, vamos a utilizar clave y significado en lugar de Π_1 y Π_2 cuando utilicemos una tupla(κ, σ).

CrearIt(**in** d: `dicc(`κ, σ`)`) $\rightarrow$ res : `itDicc(`κ, σ`)`
Pre $\equiv$ {true}
Post $\equiv$ {alias(esPermutaci?n(SecuSuby(res), d)) $\wedge$ vacia?(Anteriores(res))}
Complejidad: $\Theta(1)$
Descripci?n: crea un iterador bidireccional del diccionario, de forma tal que HayAnterior eval?e a `false` (i.e., que se pueda recorrer los elementos aplicando iterativamente Siguiente).
Aliasing: El iterador se invalida si y s?lo si se elimina el elemento siguiente del iterador sin utilizar la funci?n EliminarSiguiente. Adem?s, anteriores(res) y siguientes(res) podr?an cambiar completamente ante cualquier operaci?n que modifique d sin utilizar las funciones del iterador.

HaySiguiente(**in** it : `itDicc(`κ, σ`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ haySiguiente?(it)}
Complejidad: $\Theta(1)$
Descripci?n: devuelve `true` si y s?lo si en el iterador todav?a quedan elementos para avanzar.

HayAnterior(**in** it : `itDicc(`κ, σ`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ hayAnterior?(it)}
Complejidad: $\Theta(1)$
Descripci?n: devuelve `true` si y s?lo si en el iterador todav?a quedan elementos para retroceder.

Siguiente(**in** it : `itDicc(`κ, σ`)`) $\rightarrow$ res : `tupla(`κ, σ`)`

Pre $\equiv$ {HaySiguiente?(it)}
Post $\equiv$ {alias($res =_{\text{obs}}$ Siguiente(it))}
Complejidad: $\Theta(1)$
Descripci?n: devuelve el elemento siguiente del iterador.
Aliasing: res.significado es modificable si y s?lo si it es modificable. En cambio, res.clave no es modificable.

SiguienteClave(**in** it : `itDicc(`κ, σ`)`) $\rightarrow$ res : κ
Pre $\equiv$ {HaySiguiente?(it)}
Post $\equiv$ {alias($res =_{\text{obs}}$ Siguiente(it).clave)}
Complejidad: $\Theta(1)$
Descripci?n: devuelve la clave del elemento siguiente del iterador.
Aliasing: res no es modficable.

SiguienteSignificado(**in** it : `itDicc(`κ, σ`)`) $\rightarrow$ res : σ
Pre $\equiv$ {HaySiguiente?(it)}
Post $\equiv$ {alias($res =_{\text{obs}}$ Siguiente(it).significado)}
Complejidad: $\Theta(1)$
Descripci?n: devuelve el significado del elemento siguiente del iterador.
Aliasing: res es modificable si y s?lo si it es modficable.

Anterior(**in** it : `itDicc(`κ, σ`)`) $\rightarrow$ res : `tupla(clave:` κ`, significado:` σ`)`
Pre $\equiv$ {HayAnterior?(it)}
Post $\equiv$ {alias($res =_{\text{obs}}$ Anterior(it))}
Complejidad: $\Theta(1)$
Descripci?n: devuelve el elemento anterior del iterador.
Aliasing: res.`significado` es modificable si y s?lo si it es modificable. En cambio, res.`clave` no es modificable.

AnteriorClave(**in** it : `itDicc(`κ, σ`)`) $\rightarrow$ res : κ
Pre $\equiv$ {HayAnterior?(it)}
Post $\equiv$ {alias($res =_{\text{obs}}$ Anterior(it).clave)}
Complejidad: $\Theta(1)$
Descripci?n: devuelve la clave del elemento anterior del iterador.
Aliasing: res no es modficable.

AnteriorSignificado(**in** it : `itDicc(`κ, σ`)`) $\rightarrow$ res : σ
Pre $\equiv$ {HayAnterior?(it)}
Post $\equiv$ {alias($res =_{\text{obs}}$ Anterior(it).significado)}
Complejidad: $\Theta(1)$
Descripci?n: devuelve el significado del elemento anterior del iterador.
Aliasing: res es modificable si y s?lo si it es modficable.

Avanzar(**in/out** it : `itDicc(`κ, σ`)`)
Pre $\equiv$ {$it = it_0 \wedge$ HaySiguiente?(it)}
Post $\equiv$ {$it =_{\text{obs}}$ Avanzar(it_0)}
Complejidad: $\Theta(1)$
Descripci?n: avanza a la posici?n siguiente del iterador.

Retroceder(**in/out** it : `itDicc(`κ, σ`)`)
Pre $\equiv$ {$it = it_0 \wedge$ HayAnterior?(it)}
Post $\equiv$ {$it =_{\text{obs}}$ Retroceder(it_0)}
Complejidad: $\Theta(1)$
Descripci?n: retrocede a la posici?n anterior del iterador.

EliminarSiguiente(**in/out** it : `itDicc(`κ, σ`)`)
Pre $\equiv$ {$it = it_0 \wedge$ HaySiguiente?(it)}
Post $\equiv$ {$it =_{\text{obs}}$ EliminarSiguiente(it_0)}
Complejidad: $\Theta(1)$
Descripci?n: elimina del diccionario la clave del elemento que se encuentra en la posici?n siguiente.

EliminarAnterior(**in/out** it : `itDicc(`κ, σ`)`)
Pre $\equiv$ {$it = it_0 \wedge$ HayAnterior?(it)}

Post $\equiv \{it =_{\text{obs}} \text{EliminarAnterior}(it_0)\}$
Complejidad: $\Theta(1)$
Descripci?n: elimina del diccionario la clave del elemento que se encuentra en la posici?n anterior.

Especificaci?n de las operaciones auxiliares utilizadas en la interfaz

TAD Diccionario Extendido(κ, σ)

extiende DICCIONARIO(κ, σ)

otras operaciones (no exportadas)
esPermutacion? : secu(tupla(κ, σ)) × dicc(κ, σ) $\longrightarrow$ bool
secuADicc : secu(tupla(κ, σ)) $\longrightarrow$ dicc(κ, σ)

axiomas
esPermutacion?(s, d) $\equiv$ d = secuADicc(s) $\wedge$ #claves(d) = long(s)
secuADicc(s) $\equiv$ **if** vacia?(s) **then** vacio **else** definir(Π_1(prim(s)), Π_2(prim(s)), secuADict(fin(s))) **fi**

Fin TAD

Representaci?n

Representaci?n del diccionario

Hay dos opciones b?sicas para representar el diccionario lineal, con sus pros y sus contras. La que parece m?s natural, es representarlo como un conjunto de tuplas sobre secuencia (ver Seccion 9). La ventaja de esta representaci?n es que el invariante de representaci?n y la funci?n de abstracci?n resultan un poco m?s naturales. La desventaja es que, como en un conjunto no se pueden modificar los valores, no podr?amos modificar el significado de una clave dada. Esto es contrario a lo que queremos. Una opci?n alternativa por este camino, es definir el diccionario como un conjunto de claves y conjunto de significados, donde cada clave guarda un iterador o puntero a un significado. Esta opci?n puede resultar viable, pero es un poco molesta.

La representaci?n que optamos consiste en definir al diccionario como dos listas, una de claves y otra de significados. La lista de claves no puede tener repetidos, mientras que la de significados si puede. Ademas, la i-?sima clave de la lista se asocia al i-?simo significado. En cierto sentido, estamos definiendo al diccionario como un conjunto de claves y una secuencia de significados. Para no repetir la representaci?n y el codigo del diccionario en el conjunto, vamos a representar al conjunto como un diccionario (ver Secci?n 9). Si bien esto no parece ser una soluci?n natural, tampoco es tan rara, y nos permite resolver el problema reutilizando la mayor?a del codigo.

`dicc`(κ, σ) **se representa con** `dic`

donde `dic` es `tupla`(*claves*: `lista`(κ), *significados*: `lista`(σ))

Rep : dic $\longrightarrow$ bool
Rep(d) $\equiv$ true $\iff$ #claves(secuADicc(d.claves)) = long(d.claves) $\wedge$ long(d.claves) = long(d.significados)

Abs : dicc d $\longrightarrow$ dicc(κ, σ) {Rep(d)}
Abs(d) $\equiv$ **if** vac?a?(d.claves) **then** vac?o **else** definir(prim(d).claves, prim(d).significado, Abs(fin(d))) **fi**

Representaci?n del iterador

El iterador del diccionario es simplemente un par de iteradores a las listas correspondientes. Lo ?nico que hay que pedir es que se satisfaga el Rep de este par de listas.

`itDicc`(κ, σ) **se representa con** `itDic`

donde `itDic` es `tupla`(*claves*: `itLista`(κ), *significados*: `itLista`(σ))

Rep : itDic $\longrightarrow$ bool
Rep(it) $\equiv$ true $\iff$ Rep($\langle$SecuSuby(it.claves), SecuSuby(it.significados)$\rangle$)

Abs : itDic it $\longrightarrow$ itBi(tupla(κ, σ)) {Rep(it)}
Abs(it) $\equiv$ CrearItBi(Join(Anteriores(it.claves), Anteriores(it.significados)),
Join(Siguientes(it.claves), Siguientes(it.significados)))

Join : secu(α) a × secu(β) b $\longrightarrow$ secu(tupla(α, β)) {long(a) = long(b)}

Join(a, b) $\equiv$ **if** vacia?(a) **then** $<>$ **else** $\langle \text{prim}(a), \text{prim}(b) \rangle \bullet \text{Join}(\text{Fin}(a), \text{Fin}(b))$ **fi**

Algoritmos

9. M?dulo Conjunto Lineal(α)

El m?dulo Conjunto Lineal provee un conjunto b?sico en el que se puede insertar, eliminar, y testear pertenencia en tiempo lineal (de comparaciones y/o copias). Cuando ya se sabe que el elemento a insertar no pertenece al conjunto, la inserci?n se puede hacer con complejidad de $O(1)$ copias.

En cuanto al recorrido de los elementos, se provee un iterador bidireccional que permite eliminar los elementos iterados.

Para describir la complejidad de las operaciones, vamos a llamar *copy*(a) al costo de copiar el elemento $a \in \alpha$ y *equal*(a_1, a_2) al costo de evaluar si dos elementos $a_1, a_2 \in \alpha$ son iguales (i.e., *copy* y *equal* son funciones de α y $\alpha \times \alpha$ en $\mathbb{N}$, respectivamente).[4]

Interfaz

par?metros formales

g?neros α

funci?n $\bullet = \bullet$(**in** $a_1 : \alpha$, **in** $a_2 : \alpha$) $\to$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} (a_1 = a_2)$}
Complejidad: $\Theta(equal(a_1, a_2))$
Descripci?n: funci?n de igualdad de α's

funci?n COPIAR(**in** $a : \alpha$) $\to$ res : α
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} a$}
Complejidad: $\Theta(copy(a))$
Descripci?n: funci?n de copia de α's

se explica con: CONJ(α), ITERADOR BIDIRECCIONAL MODIFICABLE(α).

g?neros: `conj`(α), `itConj`(α).

Operaciones b?sicas de conjunto

VAC?O() $\to$ res : `conj`(α)
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} \emptyset$}
Complejidad: $\Theta(1)$
Descripci?n: genera un conjunto vac?o.

AGREGAR(**in/out** c: `conj`(α), **in** $a : \alpha$) $\to$ res : `itConj`(α)
Pre $\equiv$ {$c =_{\text{obs}} c_0$}
Post $\equiv$ {$c =_{\text{obs}} Ag(a, c_0) \wedge$ HaySiguiente(res) $\wedge_L$ Siguiente(res) $= a \wedge$ alias(esPermutacion?(SecuSuby(res), c))}
Complejidad: $\Theta\left(\sum_{a' \in c} equal(a, a')\right)$
Descripci?n: agrega el elemento a al conjunto. Para poder acceder al elemento a en $O(1)$, se devuelve un iterador a la posici?n de a dentro de c.
Aliasing: el elemento a se agrega por copia. El iterador se invalida si y s?lo si se elimina el elemento siguiente del iterador sin utilizar la funci?n ELIMINARSIGUIENTE. Adem?s, anteriores(res) y siguientes(res) podr?an cambiar completamente ante cualquier operaci?n que modifique c sin utilizar las funciones del iterador.

AGREGARRAPIDO(**in/out** c: `conj`(α), **in** $a : \alpha$) $\to$ res : `itConj`(α)
Pre $\equiv$ {$c =_{\text{obs}} c_0 \wedge a \notin c$}
Post $\equiv$ {$c =_{\text{obs}} Ag(a, c_0) \wedge$ HaySiguiente(res) $\wedge_L$ Siguiente(res) $= a \wedge$ alias(esPermutacion?(SecuSuby(res), c))}
Complejidad: $\Theta(copy(a))$
Descripci?n: agrega el elemento $a \notin c$ al conjunto. Para poder acceder al elemento a en $O(1)$, se devuelve un iterador a la posici?n de a dentro de c.
Aliasing: el elemento a se agrega por copia. El iterador se invalida si y s?lo si se elimina el elemento siguiente del iterador sin utilizar la funci?n ELIMINARSIGUIENTE. Adem?s, anteriores(res) y siguientes(res) podr?an cambiar

[4] N?tese que este es un abuso de notaci?n, ya que no estamos describiendo *copy* y *equal* en funci?n del tama?o de a. A la hora de usarlo, habr? que realizar la traducci?n.

completamente ante cualquier operaci?n que modifique c sin utilizar las funciones del iterador.

EsVac?o?(**in** c: `conj(`α`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} \emptyset?(c)$}
Complejidad: $\Theta(1)$
Descripci?n: devuelve `true` si y s?lo si c esta vac?o.

Pertenece?(**in** c: `conj(`α`)`, **in** a: α) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} a \in c$)}
Complejidad: $\Theta\left(\sum_{a' \in c} equal(a, a')\right)$
Descripci?n: devuelve `true` si y s?lo a pertenece al conjunto.

Eliminar(**in** c: `conj(`α`)`, **in** a: α)
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} c \setminus \{a\}$)}
Complejidad: $\Theta\left(\sum_{a' \in c} equal(a, a')\right)$
Descripci?n: elimina a de c, si es que estaba.

Cardinal(**in** c: `conj(`α`)`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} \#c$)}
Complejidad: $\Theta(1)$
Descripci?n: devuelve la cantidad de elementos del conjunto.

Copiar(**in** c: `conj(`α`)`) $\rightarrow$ res : `conj(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} c$}
Complejidad: $\Theta\left(\sum_{a \in c} copy(a)\right)$
Descripci?n: genera una copia nueva del conjunto.

$\bullet = \bullet$(**in** c_1 : `conj(`α`)`, **in** c_2 : `conj(`α`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} c_1 = c_2$}
Complejidad: $O\left(\sum_{a_1 \in c_1} \sum_{a_2 \in c_2} equal(a_1, a_2)\right)$.
Descripci?n: compara c_1 y c_2 por igualdad.

Operaciones del iterador

El iterador que presentamos permite modificar el conjunto recorrido, eliminando elementos. Sin embargo, cuando el conjunto es no modificable, no se pueden utilizar las funciones de eliminaci?n. Adem?s, los elementos iterados no pueden modificarse, por cuestiones de implementaci?n.

CrearIt(**in** c: `conj(`α`)`) $\rightarrow$ res : `itConj(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ {alias(esPermutacion?(SecuSuby(res), c)) $\wedge$ vacia?(Anteriores(res))}
Complejidad: $\Theta(1)$
Descripci?n: crea un iterador bidireccional del conjunto, de forma tal que HayAnterior eval?e a `false` (i.e., que se pueda recorrer los elementos aplicando iterativamente Siguiente).
Aliasing: El iterador se invalida si y s?lo si se elimina el elemento siguiente del iterador sin utilizar la funci?n EliminarSiguiente. Adem?s, anteriores(res) y siguientes(res) podr?an cambiar completamente ante cualquier operaci?n que modifique c sin utilizar las funciones del iterador.

HaySiguiente(**in** it: `itConj(`α`)`) $\rightarrow$ res : `bool`

Pre ≡ {true}
Post ≡ {$res =_{\text{obs}}$ haySiguiente?(it)}
Complejidad: Θ(1)
Descripci?n: devuelve `true` si y s?lo si en el iterador todav?a quedan elementos para avanzar.

HayAnterior(**in** it : `itConj(`α`)`) → res : `bool`
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}}$ hayAnterior?(it)}
Complejidad: Θ(1)
Descripci?n: devuelve `true` si y s?lo si en el iterador todav?a quedan elementos para retroceder.

Siguiente(**in** it : `itConj(`α`)`) → res : α
Pre ≡ {HaySiguiente?(it)}
Post ≡ {alias($res =_{\text{obs}}$ Siguiente(it))}
Complejidad: Θ(1)
Descripci?n: devuelve el elemento siguiente a la posici?n del iterador.
Aliasing: res no es modificable.

Anterior(**in** it : `itConj(`α`)`) → res : α
Pre ≡ {HayAnterior?(it)}
Post ≡ {alias($res =_{\text{obs}}$ Anterior(it))}
Complejidad: Θ(1)
Descripci?n: devuelve el elemento anterior a la posici?n del iterador.
Aliasing: res no es modificable.

Avanzar(**in/out** it : `itConj(`α`)`)
Pre ≡ {$it = it_0 \wedge$ HaySiguiente?(it)}
Post ≡ {$it =_{\text{obs}}$ Avanzar(it_0)}
Complejidad: Θ(1)
Descripci?n: Avanza a la posici?n siguiente del iterador.

Retroceder(**in/out** it : `itConj(`α`)`)
Pre ≡ {$it = it_0 \wedge$ HayAnterior?(it)}
Post ≡ {$it =_{\text{obs}}$ Retroceder(it_0)}
Complejidad: Θ(1)
Descripci?n: Retrocede a la posici?n anterior del iterador.

EliminarSiguiente(**in/out** it : `itConj(`α`)`)
Pre ≡ {$it = it_0 \wedge$ HaySiguiente?(it)}
Post ≡ {$it =_{\text{obs}}$ EliminarSiguiente(it_0)}
Complejidad: Θ(1)
Descripci?n: Elimina de la lista iterada el valor que se encuentra en la posici?n siguiente del iterador.

EliminarAnterior(**in/out** it : `itConj(`α`)`)
Pre ≡ {$it = it_0 \wedge$ HayAnterior?(it)}
Post ≡ {$it =_{\text{obs}}$ EliminarAnterior(it_0)}
Complejidad: Θ(1)
Descripci?n: Elimina de la lista iterada el valor que se encuentra en la posici?n anterior del iterador.

Especificaci?n de las operaciones auxiliares utilizadas en la interfaz

TAD Conjunto Extendido(α)

extiende Conjunto(α)

otras operaciones (no exportadas)
esPermutacion? : secu(α) × conj(α) ⟶ bool
secuAConj : secu(α) ⟶ conj(α)

axiomas
esPermutacion?(s, c) ≡ c = secuAConj(s) ∧ #c = long(s)

secuAConj(s) $\equiv$ **if** vacia?(s) **then** $\emptyset$ **else** Ag(prim(s), secuAConj(fin(s))) **fi**

Fin TAD

Representaci?n

Representaci?n del Conjunto

En este m?dulo vamos a utilizar un diccionario lineal para representar el conjunto. La idea es que el conjunto de claves del diccionario represente el conjunto lineal. Si bien esta representaci?n no es la m?s natural, permite resolver unas cuantas cuestiones sin duplicar codigo. La desventaja aparente es que gastamos memoria para guardar datos in?tiles. Sin embargo, los lenguajes de programaci?n actuales permiten resolver este problema de forma m?s o menos elegante. A nosotros no nos va a importar.

`conj`(α) **se representa con** `dicc(`α`, bool)`

Rep : dicc(α,bool) $\longrightarrow$ bool
Rep(d) $\equiv$ true

Abs : dicc(α,bool) d $\longrightarrow$ conj(α) {Rep(d)}
Abs(d) $\equiv$ claves(d)

Representaci?n del iterador

El iterador del conjunto es simplemente un iterador del diccionario representante.

`itConj`(α) **se representa con** `itDicc(`α`, bool)`

Rep : itDicc(α, bool) $\longrightarrow$ bool
Rep(it) $\equiv$ true

Abs : itDicc(α, bool) it $\longrightarrow$ itBi(α) {Rep(it)}
Abs(it) $=_{\text{obs}}$ b: itBi(α) | Anteriores(b) = Π_1(Anteriores(it)) $\wedge$ Siguientes(b) = Π_1(Siguientes(it))

Π_1 : secu(tupla(α, β)) $\longrightarrow$ secu(α)
$\Pi_1(s)$ $\equiv$ **if** vacia?(s) **then** <> **else** Π_1(prim(s)) $\bullet$ Π_1(Fin(s)) **fi**

Algoritmos

10. M?dulo Conjunto acotado de naturales

El m?dulo conjunto acotado de naturales provee un conjunto en el que se pueden insertar ?nicamente los elementos que se encuentran en un rango $[\ell, r]$ de naturales. La inserci?n, eliminaci?n y testeo de pertenencia de un elemento se pueden resolver en tiempo constante. El principal costo se paga cuando se crea la estructura, dado que cuesta tiempo lineal en $r - \ell$.

En cuanto al recorrido de los elementos, se provee un iterador bidireccional que tambi?n permite eliminar los elementos iterados.

Especificaci?n

TAD Conjunto acotado

géneros conjAcotado

igualdad observacional

$$(\forall c_1, c_2 : \text{conjAcotado}) \left(c_1 =_{\text{obs}} c_2 \iff \begin{pmatrix} \text{Infimo}(c_1) =_{\text{obs}} \text{Infimo}(c_2) \wedge \text{Supremo}(c_1) =_{\text{obs}} \text{Supremo}(c_2) \wedge \\ \text{ConjSuby}(c_1) =_{\text{obs}} \text{ConjSuby}(c_2) \end{pmatrix} \right)$$

observadores básicos

Infimo : conjAcotado $\longrightarrow$ nat
Supremo : conjAcotado $\longrightarrow$ nat
ConjSuby : conjAcotado $\longrightarrow$ conj(nat)

generadores

$\emptyset$: nat ℓ × nat r $\longrightarrow$ conjAcotado $\{\ell \leq r\}$
Ag : nat e × conjAcotado c $\longrightarrow$ conjAcotado $\{\text{Infimo}(c) \leq e \leq \text{Supremo}(c)\}$

otras operaciones
Rango : conjAcotado $\longrightarrow$ tupla(nat, nat)

axiomas
Infimo($\emptyset(\ell, r)$) $\equiv$ ℓ
Infimo(Ag(e, c)) $\equiv$ Infimo(c)
Supremo($\emptyset(\ell, r)$) $\equiv$ r
Supremo(Ag(e, c)) $\equiv$ Supremo(c)
ConjSuby($\emptyset(\ell, r)$) $\equiv$ $\emptyset$
ConjSuby(Ag(e, c)) $\equiv$ Ag(e, ConjSuby(c))
Rango(c) $\equiv$ $\langle$Infimo(c), Supremo(c)$\rangle$

Fin TAD

Interfaz

se explica con: Conjunto acotado, Iterador Bidireccional(nat).

g?neros: `conjAcotado`, `itConjAcotado`.

Operaciones b?sicas de conjunto

Vac?o(**in** ℓ : `nat`, **in** r : `nat`) $\rightarrow$ res : `conjAcotado`
Pre $\equiv$ $\{\ell \leq r\}$
Post $\equiv$ $\{res =_{\text{obs}} \emptyset(\ell, r)\}$
Complejidad: $\Theta(r - \ell)$
Descripci?n: genera un conjunto vac?o con el rango $[\ell, r.]$

Agregar(**in/out** c: `conjAcotado`, **in** e : `nat`)
Pre $\equiv$ $\{c =_{\text{obs}} c_0 \wedge \text{Infimo}(c) \leq e \leq \text{Supremo}(c)\}$
Post $\equiv$ $\{c =_{\text{obs}} Ag(e, c_0)\}$
Complejidad: $\Theta(1)$
Descripci?n: agrega el elemento e al conjunto.

Infimo(**in** c: `conjAcotado`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} \text{Infimo}(c)\}$
Complejidad: $\Theta(1)$
Descripci?n: devuelve el valor m?nimo que se puede agregar al conjunto.

Supremo(**in** c: `conjAcotado`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} \text{Supremo}(c)\}$
Complejidad: $\Theta(1)$
Descripci?n: devuelve el valor m?ximo que se puede agregar al conjunto.

EsVac?o?(**in** c: `conjAcotado`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} \emptyset?(\text{ConjSuby}(c))\}$
Complejidad: $\Theta(1)$
Descripci?n: devuelve `true` si y s?lo si c esta vac?o.

Pertenece?(**in** c: `conjAcotado`, **in** e: `nat`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} e \in \text{ConjSuby}(c)\}$
Complejidad: $\Theta(1)$
Descripci?n: devuelve `true` si y s?lo e pertenece al conjunto. Notar que no es requerido que e pertenezca al rango de c.

Eliminar(**in/out** c: `conjAcotado`, **in** e: `nat`)

Pre $\equiv \{c = c_0\}$
Post $\equiv \{\text{ConjSuby}(c) =_{\text{obs}} \text{ConjSuby}(c_0) \setminus \{e\} \wedge \text{Rango}(c) =_{\text{obs}} \text{Rango}(c_0)\}$
Complejidad: $\Theta(1)$
Descripci?n: Elimina a de c, si es que estaba. Observar que no es requerido que e pertenezca al rango de c.

CARDINAL(**in** c: `conjAcotado`) $\rightarrow$ res : `nat`
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} \#\text{ConjSuby}(c)\}$
Complejidad: $\Theta(1)$
Descripci?n: Devuelve la cantidad de elementos del conjunto.

COPIAR(**in** c: `conjAcotado`) $\rightarrow$ res : `conjAcotado`
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} c\}$
Complejidad: $\Theta(\text{Supremo}(c)$ - $\text{Infimo}(c))$
Descripci?n: genera una copia nueva del conjunto.

$\bullet = \bullet$(**in** c_1 : `conjAcotado`, **in** c_2 : `conjAcotado`) $\rightarrow$ res : `bool`
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} c_1 = c_2\}$
Complejidad: $\Theta(\text{mín}\{\#c_1, \#c_2\})$.
Descripci?n: compara c_1 y c_2 por igualdad.

Operaciones del iterador

El iterador que presentamos permite modificar el conjunto recorrido, eliminando elementos. Sin embargo, cuando el conjunto es no modificable, no se pueden utilizar las funciones de eliminaci?n. Todos los naturales del conjunto son iterados por copia.

CREARIT(**in** c: `conjAcotado`) $\rightarrow$ res : `itConjAcotado`
Pre $\equiv \{\text{true}\}$
Post $\equiv \{\text{alias}(\text{esPermutaci?n?}(\text{SecuSuby}(res), \text{ConjSuby}(c))) \wedge \text{vacia?}(\text{Anteriores}(res))\}$
Complejidad: $\Theta(1)$
Descripci?n: crea un iterador bidireccional del conjunto, de forma tal que HAYANTERIOR eval?e a `false` (i.e., que se pueda recorrer los elementos aplicando iterativamente SIGUIENTE).
Aliasing: El iterador se invalida si y s?lo si se elimina el elemento siguiente del iterador sin utilizar la funci?n ELIMINARSIGUIENTE. Adem?s, anteriores(res) y siguientes(res) podr?an cambiar completamente ante cualquier operaci?n que modifique c sin utilizar las funciones del iterador.

HAYSIGUIENTE(**in** it: `itConjAcotado`) $\rightarrow$ res : `bool`
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} \text{haySiguiente?}(it)\}$
Complejidad: $\Theta(1)$
Descripci?n: devuelve `true` si y s?lo si en el iterador todav?a quedan elementos para avanzar.

HAYANTERIOR(**in** it: `itConjAcotado`) $\rightarrow$ res : `bool`
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} \text{hayAnterior?}(it)\}$
Complejidad: $\Theta(1)$
Descripci?n: devuelve `true` si y s?lo si en el iterador todav?a quedan elementos para retroceder.

SIGUIENTE(**in** it: `itConjAcotado`) $\rightarrow$ res : `nat`
Pre $\equiv \{\text{HaySiguiente?}(it)\}$
Post $\equiv \{res =_{\text{obs}} \text{Siguiente}(it)\}$
Complejidad: $\Theta(1)$
Descripci?n: devuelve el elemento siguiente a la posici?n del iterador.
Aliasing: res se devuelve por copia.

ANTERIOR(**in** it: `itConjAcotado`) $\rightarrow$ res : `nat`
Pre $\equiv \{\text{HayAnterior?}(it)\}$
Post $\equiv \{res =_{\text{obs}} \text{Anterior}(it)\}$

Complejidad: $\Theta(1)$
Descripci?n: devuelve el elemento anterior a la posici?n del iterador.
Aliasing: *res* se devuelve por copia.

AVANZAR(**in/out** *it* : `itConjAcotado`)
Pre $\equiv \{it = it_0 \wedge \text{HaySiguiente?}(it)\}$
Post $\equiv \{it =_{\text{obs}} \text{Avanzar}(it_0)\}$
Complejidad: $\Theta(1)$
Descripci?n: Avanza a la posici?n siguiente del iterador.

RETROCEDER(**in/out** *it* : `itConjAcotado`)
Pre $\equiv \{it = it_0 \wedge \text{HayAnterior?}(it)\}$
Post $\equiv \{it =_{\text{obs}} \text{Retroceder}(it_0)\}$
Complejidad: $\Theta(1)$
Descripci?n: Retrocede a la posici?n anterior del iterador.

ELIMINARSIGUIENTE(**in/out** *it* : `itConjAcotado`)
Pre $\equiv \{it = it_0 \wedge \text{HaySiguiente?}(it)\}$
Post $\equiv \{it =_{\text{obs}} \text{EliminarSiguiente}(it_0)\}$
Complejidad: $\Theta(1)$
Descripci?n: Elimina del conjunto el elemento que se encuentra en la posici?n siguiente.

ELIMINARANTERIOR(**in/out** *it* : `itConjAcotado`)
Pre $\equiv \{it = it_0 \wedge \text{HayAnterior?}(it)\}$
Post $\equiv \{it =_{\text{obs}} \text{EliminarAnterior}(it_0)\}$
Complejidad: $\Theta(1)$
Descripci?n: Elimina del conjunto el elemento que se encuentra en la posici?n anterior.

Representaci?n

Representaci?n del Conjunto

La idea de este m?dulo es aprovechar que los elementos que se pueden llegar a agregar son naturales en un rango que se conoce desde el inicio, de forma tal de poder acceder a ellos en tiempo $O(1)$. Para esto, podemos tener un arreglo a de booleanos de tama?o $r - \ell + 1$ de forma tal que $\ell \leq e \leq r$ pertenezca al conjunto si y s?lo si $a[e - \ell] = \text{true}$. El inconveniente de esta representaci?n es que no permite iterar todos los elementos en tiempo lineal en la cantidad de elementos del conjunto. En efecto, si el conjunto tiene un ?nico elemento e, igual tenemos que recorrer todo el rango $r - \ell$ (que no es constante) para encontrar e. Para subsanar este inconveniente, vamos a guardar un conjunto lineal c con los elementos que pertenecen al conjunto acotado. Para poder eliminar el elemento e, debemos poner en false el valor de $a[e - \ell]$, a la vez que tenemos que eliminar a c del conjunto. Esto se puede hacer en tiempo $O(1)$ si podemos obtener eficientemente un "puntero" a e dentro de c. Este puntero podr?a ser un iterador. Luego, en a vamos a tener, ademas del booleano, un iterador al conjunto c que nos permita acceder en $O(1)$ a e dentro de c. Una mejora a esta estructura es eliminar el booleano de a, y considerar que e pertenece al conjunto acotado si y s?lo si el iterador de $a[e - \ell]$ tiene un elemento siguiente. Este elemento siguiente contiene a e en c.

`conjAcotado` **se representa con** `ca`

donde `ca` es `tupla`(*pertenencia*: `arreglo_dimensionable de iterConj(nat)`,
elementos: `conj(nat)`, *infimo*: `nat`)

$\text{Rep} : \text{ca} \longrightarrow \text{bool}$
$\text{Rep}(c) \equiv \text{true} \iff (\forall e\text{: nat})(e \in c.\text{elementos} \iff e \geq c.\text{infimo} \wedge e < c.\text{infimo} + \text{tam}(c.\text{pertenencia}) \wedge_{\text{L}}$
$\text{HaySiguiente?}(c.\text{pertenencia}[e - c.\text{infimo}])) \wedge_{\text{L}}$
$(\forall e\text{: nat})(e \in c.\text{elementos} \Rightarrow_{\text{L}} \text{Siguiente}(c.\text{pertenencia}[e - c.\text{infimo}]) = e)$

$\text{Abs} : \text{ca } e \longrightarrow \text{conjAcotado} \qquad \{\text{Rep}(e)\}$
$\text{Abs}(e) =_{\text{obs}} \text{c: conjAcotado} \mid \text{Infimo}(c) = e.\text{infimo} \wedge \text{Supremo}(c) = e.\text{infimo} + \text{tam}(e.\text{pertenencia}) - 1 \wedge$
$\text{ConjSuby}(c) = e.\text{elementos}$

Representaci?n del iterador

El iterador del conjunto acotado es simplemente un iterador del conjunto *elementos*, ya que con ?ste recorremos

todos los elementos, m?s un puntero a la estructura del conjunto, para poder borrar al eliminar el iterador.

`itConjAcotado` se representa con `itCA`

donde `itCA` es `tupla`(*iter*: `itConj(nat)`, *conj*: `puntero(ca)`)

Rep : itCA $\longrightarrow$ bool
Rep(it) $\equiv$ true $\iff$ Rep($*it$.conj) $\wedge$ EsPermutacion(SecuSuby(it.iter), it.conj$\rightarrow$elementos)

Abs : itCA it $\longrightarrow$ itBi(nat) {Rep(it)}
Abs(it) $\equiv$ it.elementos

Algoritmos