Iteradores

Algoritmos y Estructuras de Datos II

Segundo cuatrimestre - 2010

1. Motivación

La necesidad de los iteradores surge con la implementación de un algoritmo que utilice alguna estructura contenedora puesto que es de esperarse que en algún momento sea necesario recorrer los elementos que esta contiene.

Para ser más concretos, supóngase que se cuenta con una secuencia de números naturales y se desea calcular su suma. Utilizando la interfaz de secuencia definida en el Apéndice A.1, la versión más simple de este algoritmo sería:

```
sumaSecu(in/out s: secu(nat)) → result: nat
result ← 0
t: secu(nat) ← copiarSecu(s) // Costo de innecesario copiado, O(n)
while tam(t) > 0 do // Costo O(n)
   result ← result + prim(t)
   fin(t)
end while
```

Dado que la **única** forma para iterar los elementos de la secuencia pasada por parámetro es destruyéndola, es necesario realizar una copia para no destruir a la secuencia original dado que fue pasada por referencia. Queda en claro que sería posible pasarla por copia. En ese caso estaríamos parados ante el mismo problema, pero disfrazado en nuestro lenguaje de diseño. El hecho de que sea necesario copiar el contenedor para poder realizar una operación tan sencilla como sumar sus elementos hace notar un **problema** en la interfaz, puesto que no es posible utilizar el contenedor sin copiarlo.

Este mismo problema ocurre si consideramos otras estructuras, como conjuntos o colas. Por ejemplo, utilizando la interfaz para conjunto definida en el apendice A.2, esta misma función puede definirse de la siguiente forma:

```
sumaConj(in/out c: conj(nat)) → result: nat
result ← 0
d: conj(nat) ← copiarConj(c) // Costo de innecesario copiado, O(n)
while tam(d) > 0 do // Costo: O(n)
   result ← result + dameUno(d)
   sinUno(d)
end while
```

Notar que la función es la misma, lo único que cambió fue el nombre de las operaciónes. Es decir, si se contara con una forma genérica para iterar estas estructuras, no sólo se podría evitar su copia, sino que se podría utilizar un mismo algoritmo para sumar elementos de una secuencia y de un conjunto.

Tener en cuenta que la interfaz presentada para el tipo Conjunto no es implementable para cualquier tipo α. Esto es porque la implementación de `dameUno` debe devolver **siempre** el mismo elemento ante conjuntos observacionalmente iguales. Esto hace que sea impracticable si α es un tipo complejo o cuya relación de orden no sea clara, como un árbol.

2. Iteradores

Un iterador, en su forma más simple, es una estructura de datos que permite recorrer de forma eficiente otra estructura. Como se vio en la sección anterior, no tener iteradores trae problemas en la utilización de estructuras contenedoras. Estos problemas pueden resumirse en dos tipos:

- **Copiado innecesario**. Dado que la única forma de iterar una estructura es a través de operaciones que la destruyen, es necesario realizar una copia previa a iterarla
- **Especificidad de los algoritmos**. Dado que el mecanismo de iteración es el diferente para cada estructura, es necesario reimplementar algoritmos básicos para cada una.

Dado que se pretende que la implementación de un iterador no realice una copia del contenedor que va a iterar, es de esperarse que conozca su estructura interna. Es por esto que para cada contenedor, habrá que diseñar su iterador. Supóngase que se cuenta ahora un módulo, que llamaremos *itsecu*, que permite iterar una secuencia. Supóngase ahora que este módulo exporta las operaciones `crearIt`, que lo construye a partir de una secuencia, `hayMas?` que informa si todavía restan elementos por recorrer, `actual` que devuelve el elemento actual y `avanzar` que avanza hacia el próximo elemento. Utilizando este módulo sería posible recorrer una secuencia y calcular, por ejemplo, `sumaSecu`:

```
sumaSecu(in/out s: secu(nat)) → result: nat
result ← 0
it: itsecu(nat) ← crearIt(s) // Costo O(1)
while hayMas?(it) do
    result ← result + actual(it)
    avanzar(it)
end while
```

Lo mismo se podría hacer para conjuntos, o se podría ir más allá y diseñar un algoritmo que reciba un iterador y realice la suma del contenido que es iterado, no importa si itera un conjunto, una secuencia o una cola de prioridad:

```
sumaIt(in/out it: iterador(nat)) → result: nat
result ← 0
while hayMas?(it) do
    result ← result + actual(it)
    avanzar(it)
end while
```

Obsérvese que el tipo de la entrada es ***iterador(nat)***. Esto es un abuso de notación, puesto que esta función no podría nunca recibir una entrada de tipo `itsecu(nat)` ni `itconj(nat)`. Puesto que no es el objetivo de este apunte mostrar como es posible construir una interfaz genérica de este tipo, se continuará utilizando este abuso de notación. Más adelante de detallará de que forma implementarlo.

A partir de estos ejemplos se puede ver la utilidad de los iteradores, y la elegancia con la que resuelven los problemas planteados. Ahora, para poder llevar a cabo diseños que utilicen iteradores será necesario modelarlo con un TAD, para luego especificar la interfaz para los ejemplos planteados.

3. TAD Iterador$<\alpha>$

Como se expuso en la sección anterior, los iteradores se explicarán con un TAD genérico. A continuación se dá la especificación del TAD ITERADOR$<\alpha>$

TAD ITERADOR$<\alpha>$

géneros it$<\alpha>$

exporta it$<\alpha>$, generadores, observadores

usa BOOL, NAT, SECUENCIA$<\alpha>$

igualdad observacional

$$(\forall it1, it2 : \text{it}{<}\alpha{>}) \left(it1 =_{\text{obs}} it2 \iff \begin{pmatrix} (hayMas?(it1) =_{\text{obs}} hayMas?(it2)) \land_{\text{L}} \\ (hayMas?(it1) \Rightarrow_{\text{L}} \\ (actual(it1) =_{\text{obs}} actual(it2) \land \\ proximo(it1) =_{\text{obs}} proximo(it2))) \end{pmatrix} \right)$$

observadores básicos

hayMas?	: it(α)	$\longrightarrow$	bool	
actual	: it(α)i	$\longrightarrow$	α	{hayMas?(i)}
avanzar	: it(α)i	$\longrightarrow$	it(α)	{hayMas?(i)}

generadores

crearIt : secu(α) $\longrightarrow$ it(α)

axiomas $\forall it$: it(α)

hayMas(crearIt(s)) $\equiv$ $\neg$ vacia?(s)
actual(crearIt(s)) $\equiv$ prim(s)
avanzar(crearIt(s)) $\equiv$ crearIt(fin(s))

Fin TAD

Como se puede observar, este iterador tiene la funcionalidad mínima, permitiendo solamente iterar la secuencia con la que se lo construye. Es importante notar que este es sólo un modelo, es decir, se explicará el diseño utilizando este TAD, pero no se creará explicitamente la secuencia que este recibe en el generador *crearIt*, puesto que es eso una de las cosas que se desea evitar en el diseño.

Nótese que el tipo que recibe el constructor del TAD Iterador es una secuencia. Esto quiere decir que a la hora de especificar el comportamiento de un iterador para conjuntos será necesario construir una secuencia. Este problema puede ser atacado de dos formas: O bien extender el TAD Conjunto agregando una operación que construya la secuencia de elementos a iterar, o bien extendiendo el TAD Secuencia para que construya el conjunto que corresponde a la secuencia.

Si se siguiera la primera aproximación, estaríamos ante un problema puesto que sería necesario construir siempre la misma secuencia ante dos conjuntos que son observacionalmente iguales.

Esto se podría resolver construyendo una secuencia ordenada sin repetidos, sin embargo, éste también sería un problema. Por un lado, esto sería costoso en el diseño, y por otro lado, no necesariamente los elementos a iterar cuentan con una relación de orden.

Es por esto que se tomará la segunda alternativa, y se construirá una función en el TAD Secuencia que construya el conjunto que esta representa. Queda para el lector el ejercicio de analizar este mismo problema para TAD Diccionario.

De esta forma, se extenderá el TAD Secuencia con dos operaciones que luego se utilizarán para especificar la interfaz de los iteradores para conjuntos y diccionarios.

secuAConj : secu(α) $\longrightarrow$ conj(α)
secuAConj(s) $\equiv$ **if** vacía?(s) **then** $\emptyset$ **else** Ag(prim(s), secuAConj(fin(s))) **fi**

secuADict : secu(tupla($\alpha \times \beta$)) $\longrightarrow$ dict(α, β)
secuAConj(s) $\equiv$ **if** vacía?(s) **then** vacío **else** definir(π_1(prim(s)), π_2(prim(s)), secuADict(fin(s))) **fi**

Estas operaciones permitirán especificar la interfaz del iterador para conjuntos y diccionarios.

4. Interfaces

Utilizando las funciones definidas en la sección anterior, se especificarán las interfaces de los iteradores para los TADs Secuencia, Conjunto y Diccionario.

4.1. Iterador para secuencias

Como se expuso en la sección anterior, este es el iterador más sencillo para especificar.

CREARIT(**in** s : secu(α)) $\rightarrow$ `res : itsecu(`α`)`
Pre: {`true`}
Post: {res $=_{\text{obs}}$ crearIt(s)}

HAYMAS?(**in** it : itsecu(α)) $\to$ `res : bool`
Pre: {`true`}
Post: {res $=_{\text{obs}}$ hayMas?(it)}

ACTUAL(**in** it : itsecu(α)) $\to$ `res :` α
Pre: {`true`}
Post: {res $=_{\text{obs}}$ actual(it)}

AVANZAR(**in/out** it : itsecu(α)) $\to$ `res : itsecu(`α`)`
Pre: {hayMas?(it) $\land$ it= it_0}
Post: {res $=_{\text{obs}}$ avanzar(it_0)}

4.2. Iterador para conjuntos

CREARIT(**in** c : conj(α)) $\to$ `res : itconj(`α`)`
Pre: {`true`}
Post: {($\exists$ s : secu(α))(long(s) $=_{\text{obs}}$ #c $\land$ secuAConj(s) $=_{\text{obs}}$ c $\land$ res $=_{\text{obs}}$ crearIt(s))}

HAYMAS?(**in** it : itconj(α)) $\to$ `res : bool`
Pre: {`true`}
Post: {res $=_{\text{obs}}$ hayMas?(it)}

ACTUAL(**in** it : itconj(α)) $\to$ `res :` α
Pre: {`true`}
Post: {res $=_{\text{obs}}$ actual(it)}

AVANZAR(**in/out** it : itconj(α)) $\to$ `res : itconj(`α`)`
Pre: {hayMas?(it) $\land$ it= it_0}
Post: {res $=_{\text{obs}}$ avanzar(it_0)}

Obsérvese que en la postcondición de la operación crearIt, se requiere, no solo que la secuencia represente al conjunto, sino que no tenga elementos repetidos.

4.3. Iterador para diccionarios

CREARIT(**in** d : dicc(α, β)) $\to$ `res : itdicc(`α`, `β`)`
Pre: {`true`}
Post: {($\exists$ s : secu(tupla(α, β)))(long(s) $=_{\text{obs}}$ #claves(d) $\land$ secuADict(s) $=_{\text{obs}}$ d $\land$ res $=_{\text{obs}}$ crearIt(s))}

HAYMAS?(**in** it : itdicc(α, β)) $\to$ `res : bool`
Pre: {`true`}
Post: {res $=_{\text{obs}}$ hayMas?(it)}

ACTUAL(**in** it : itdicc(α, β)) $\to$ `res :` α
Pre: {`true`}
Post: {res $=_{\text{obs}}$ actual(it)}

AVANZAR(**in/out** it : itdicc(α, β)) $\to$ `res : itdicc(`α`, `β`)`
Pre: {hayMas?(it) $\land$ it= it_0}
Post: {res $=_{\text{obs}}$ avanzar(it_0)}

5. Representación y algoritmos

5.1. Secuencia

Una vez especificado el comportamiento del iterador, es necesario dar su representación y su implementación. Para ello se asumirá una representación sencilla para $secu(\alpha)$:

`secu(`α`)` **se representa con** `puntero(nodo)`
donde `nodo` es `tupla<dato:`α`, prox:puntero(nodo)` >

Utilizando esta representación para el tipo $secu(\alpha)$ representaremos al tipo $itsecu(\alpha)$ de la siguiente forma:

`itsecu(`α`)` **se representa con** `puntero(nodo)`

Ambas estructuras tendrán el mismo invariante de representación que garantiza que la secuencia no tiene ciclos:

$\mathtt{Rep} : \mathtt{puntero(nodo)} \to \mathtt{bool}$
$\mathtt{Rep(p)} \equiv \mathtt{true} \Leftrightarrow (\exists n : \mathtt{nat})\mathtt{Finaliza(p,\ n)}$
donde $\mathtt{Finaliza(p,\ n)} \equiv \mathtt{n} > 0 \wedge_{\mathrm{L}} (\mathtt{p} = \mathtt{nil} \vee_{\mathrm{L}} \mathtt{Finaliza(p} \to \mathtt{prox}, \mathtt{n} - 1))$

Utilizando este invariante de representación se definirá la función de abstracción para ambos tipos:

$\mathtt{Abs\_secu}\text{: } \mathtt{puntero(nodo)\ p} \to \mathtt{secu}(\alpha) \qquad \{\mathtt{Rep(p)}\}$
$\mathtt{Abs\_secu(p)} \equiv \textbf{if } \mathtt{p = nil} \textbf{ then } <> \textbf{ else } \mathtt{p}\to \mathtt{dato} \bullet \mathtt{Abs\_secu(p} \to \mathtt{prox)} \textbf{ fi}$

$\mathtt{Abs\_it}\text{: } \mathtt{puntero(nodo)\ p} \to \mathtt{itsecu}(\alpha) \qquad \{\mathtt{Rep(p)}\}$
$\mathtt{Abs\_it(p)} \equiv \mathtt{CrearIt(Abs\_it(p))}$

Teniendo el invariante de representación y la función de abstracción, a continuación se exhibe la implementación del iterador.

---

iCrearIt(**in** p: puntero(nodo)) → res: puntero(nodo)
1: $res \leftarrow p$

---

iHayMas?(**in** p: puntero(nodo)) → res: bool
1: $res \leftarrow p \neq nil$

---

iActual(**in** p: puntero(nodo)) → res: α
1: $res \leftarrow p \to dato$

---

iAvanzar(**in/out** p: puntero(nodo))
1: $p \leftarrow p \to prox$

Utilizando la especificación dada por el Iterador<α>, la interfaz definida en la sección 4.1, las funciones de representación y abstracción, junto a los algoritmos definidos en esta sección, el ejemplo, la función `sumaSecu` de la sección 2 esta totalmente especificada.

5.2. Conjunto

Nuevamente, será necesario recorrer los pasos anteriores para finalizar la especificación del iterador para conjuntos. Estos pasos constan en definir la representación para el conjunto, invariantes, funciones de abstracción y algoritmos.

Dado que el objetivo no es diseñar un conjunto eficiente, utilizaremos la siguiente representación para *conj*(α):

`conj(`α`)` **se representa con** `secu(`α`)`

Utilizando esta representación para el tipo *conj*(α) representaremos al tipo *itconj*(α) de la siguiente forma:

`itconj(`α`)` **se representa con** `itsecu(`α`)`

Queda claro que existen varios invariantes posibles para la representación elegida para `conj(`α`)`. La que se utilizará en este apunte es
`Rep : secu(`α`)` $\to$ `bool`
`Rep(s)` $\equiv$ `true` $\Leftrightarrow$ `sinRepetidos(s)`
donde `sinRepetidos(s)` $\equiv$ `vacia?(s)` $\vee_{\mathrm{L}}$ ($\neg$`esta?(prim(s), fin(s))` $\wedge$ `sinRepetidos(fin(s))`)

Dado que el invariante de representación del iterador para conjuntos no pide ninguna restricción, se definirá la función de abstracción para ambos tipos:

`Abs_conj: secu(`α`) s` $\to$ `conj(`α`)` $\{$`Rep(s)`$\}$
`Abs_conj(s)` $\equiv$ **if** `vacia?(s)` **then** $\emptyset$ **else** `Ag(prim(s), Abs_conj(fin(s))` **fi**

`Abs_it: itsecu(`α`) its` $\to$ `itconj(`α`)` $\{$`true`$\}$
`Abs_it(its)` $\equiv$ `CrearIt(itASecu(its))`
donde `itASecu(its)` $\equiv$ **if** `hayMas?(its)` **then** `actual(its)` $\bullet$ `itASecu(avanzar(its))` **else** `<>` **fi**

Teniendo el invariante de representación y la función de abstracción, a continuación se exhibe la implementación del iterador.

---

iCrearIt(**in** s: secu(α)) $\to$ res: itsecu(α)
1: $res \leftarrow crearIt(s)$

---

iHayMas?(**in** its: itsecu(α)) $\to$ res: bool
1: $res \leftarrow hayMas?(its)$

---

iActual(**in** its: itsecu(α)) $\to$ res: α
1: $res \leftarrow actual(its)$

---

iAvanzar(**in/out** its: itsecu(α))
1: $avanzar(its)$

Obsérvese que el hecho de definir una estructura sobre otra que ya cuenta con un iterador simplifica mucho la tarea de diseñar el nuevo iterador. Tener en cuenta que esto es así en gran parte por el invariante de representación, puesto que no admite repetidos en la secuencia que representa al conjunto. Queda como ejercicio para el lector el caso donde la secuencia admite repetidos y si es posible construir un iterador para el conjunto en tiempo lineal.

5.3. Diccionario

Por último, resta finalizar la especificación del diccionario.
Como es de esperarse, se representará el diccionario sobre conjuntos:
`dicc(`α,β`)` **se representa con** `conj(tupla(`α,β`))`

Utilizando esta representación para el tipo *dicc*(α, β) representaremos al tipo *itdicc*(α, β) de la siguiente forma:

`itdicc`(α, β) **se representa con** `itconj(tupla`(α, β)`)`

Cuyo invariante será:
`Rep : conj`$(\alpha, \beta) \rightarrow$ `bool`
`Rep(c)` $\equiv$ `true` $\Leftrightarrow (\forall t_1, t_2 : \texttt{tupla}(\alpha, \beta))(t_1 \in c \wedge t_2 \in c \wedge t_1 \neq t_2 \Rightarrow \pi_1(t_1) \neq \pi_1(t_2))$

De la misma forma que en el conjunto sobre secuencia, el invariante del diccionario sobre conjuntos tiene un invariante que es siempre verdadero. De esta forma, se definirá la función de abstracción para ambos tipos:

`Abs_dicc: conj(tupla`(α, β)`) c` $\rightarrow$ `dicc`(α, β) $\{$`Rep(c)`$\}$
`Abs_dicc(c)` $\equiv$ **if** $\emptyset$`?(c)` **then**
dicc()
else
`definir(`π_1`(dameUno(c)),` π_2`(dameUno(c)), Abs_dicc(sinUno(c)))`
fi

`Abs_it: itconj(tupla`(α, β)`) itc` $\rightarrow$ `itdicc`(α, β) $\{$`true`$\}$
`Abs_it(itc)` $\equiv$ `CrearIt(itASecu(itc))`

Teniendo el invariante de representación y la función de abstracción, a continuación se exhibe la implementación del iterador.

---

iCrearIt(**in** c: conj(tupla(α, β))) $\rightarrow$ res: itconj(tupla(α, β))
1: $res \leftarrow crearIt(c)$

---

iHayMas?(**in** itc: itconj(tupla(α, β))) $\rightarrow$ res: bool
1: $res \leftarrow hayMas?(itc)$

---

iActual(**in** itc: itconj(tupla(α, β))) $\rightarrow$ res: α, β
1: $res \leftarrow actual(itc)$

---

iAvanzar(**in/out** itc: itconj(tupla(α, β)))
1: $avanzar(itc)$

A. Interfaces de TADs utilizados

A lo largo del apunte se utilizaron los TADs SECUENCIA, CONJUNTO y DICCIONARIO. A continuación se especificarán sus interfaces.

A.1. Interfaz para Secuencia

SECU() $\rightarrow$ `res : secu(`α`)`
Pre: {`true`}
Post: { res $=_{\text{obs}}$ <>}

PRIM(**in** s : secu(α)) $\rightarrow$ `res :` α
Pre: {`true`}
Post: {res $=_{\text{obs}}$ prim(s)}

FIN(**in/out** s : secu(α))
Pre: $\{s = s_0 \wedge \neg vacia(s)\}$
Post: {s $=_{\text{obs}}$ fin(s_0)}

VACIA(**in** s : secu(α)) $\rightarrow$ `res : bool`
Pre: {`true`}
Post: {res $=_{\text{obs}}$ prim(s)}

A.2. Interfaz para Conjunto

CONJ() $\rightarrow$ `res : conj(`α`)`
Pre: {`true`}
Post: {res $=_{\text{obs}}$ $\emptyset$}

DAMEUNO(**in** c : conj(α)) $\rightarrow$ `res :` α
Pre: {`true`}
Post: {res $=_{\text{obs}}$ dameUno(c)}

SINUNO(**in/out** c : conj(α))
Pre: $\{c = c_0 \wedge \neg\emptyset?(c)\}$
Post: {c $=_{\text{obs}}$ sinUno(c_0)}

VACIO(**in** c : conj(α)) $\rightarrow$ `res : bool`
Pre: {`true`}
Post: {res $=_{\text{obs}}$ dameUno(c)}

A.3. Interfaz para Diccionario

DICC() $\rightarrow$ `res : dicc(`α`,` β`)`
Pre: {`true`}
Post: {res $=_{\text{obs}}$ vacio}

OBTENER(**in** d : dicc(α, β), **in** clave : α) $\rightarrow$ `res :` β
Pre: {def?(clave, d)}
Post: {res $=_{\text{obs}}$ obtener(clave, d)}