Guía 1 — Soluciones a los ejercicios seleccionados

Algoritmos y Estructuras de Datos II, DC, UBA.

Primer cuatrimestre de 2020

1. Axiomatización

Ejercicio 1:

Se pide extender el tipo SECUENCIA(α) definiendo nuevas operaciones.

DUPLICAR: dada una secuencia la devuelve duplicada elemento a elemento.
Solución:
En este caso brindamos una solución previamente (en la guía) donde se axiomatizaba partiendo de generadores. Si bien esa solución es correcta, lo mejor para definir operaciones es utilizar los observadores básicos. El motivo principal, es porque queremos evitar *romper congruencia*, es decir, queremos evitar que otras operaciones *observen* cosas que los observadores básicos no. Si tienen dudas con este tema, consulten en el foro o a través del zoom. ¿Cómo sería entonces una solución utilizando observadores básicos?

Duplicar : secu(α) s $\longrightarrow$ secu(α)
Duplicar?(s) $\equiv$ **if** vacía?(s) **then** <> **else** prim(s) $\bullet$ prim(s) $\bullet$ Duplicar(fin(s)) **fi**

REVERSO: dada una secuencia devuelve su reverso (la secuencia dada vuelta).
Solución:
En este caso estamos seguros que esta operación no rompe congruencia, podríamos utilizar tanto el esquema de resolución anterior a través de los observadores como a través de los generadores. Igual recuerden: ante la duda, siempre mejor axiomatizar utilizando observadores básicos. Utilizaremos la operación $\circ$, se recomienda tener en mano el apunte de TADs básicos del campus.

Reverso : secu(α) $\longrightarrow$ secu(α)
Reverso(<>) $\equiv$ <>
Reverso($e \bullet s$) $\equiv$ Reverso(s) $\circ$ e

ESPREFIJO?: chequea si una secuencia es prefijo de otra.
Solución:
Vamos a chequear si a es prefijo de b.

EsPrefijo? : secu(α) a $\times$ secu(α) b $\longrightarrow$ bool
esPrefijo?(<>, b) $\equiv$ true
esPrefijo?($\underbrace{e \bullet s}_{a}$, b) $\equiv$ **if** vacía?(b) **then** false **else** e = prim(b) $\wedge$ esPrefijo?(s, fin(b)) **fi**

Noten que hacemos recursión sobre la primera lista. ¿Podríamos hacerlo sobre la segunda? Sí, pueden pensarlo como ejercicio.
Tengan en cuenta que si decidimos axiomatizar sobre los generadores en ambas secuencias hay que tener el cuidado de contemplar todos los casos y que sean disjuntos, es decir, para un par de listas a y b nunca valen dos axiomas a la vez. Por ejemplo:

esPrefijo?(<>, <>) $\equiv$ ...
esPrefijo?(e $\bullet$ s, <>) $\equiv$ ...
esPrefijo?(<>, e $\bullet$ s) $\equiv$ ...
esPrefijo?($e_1 \bullet s_1$, $e_2 \bullet s_2$) $\equiv$ ...
En el siguiente ejemplo, para esPrefijo?(<>,<>) valdrían tanto el primero como el último renglón. Esto **no debe pasar**:

esPrefijo?(<>, <>) $\equiv$ ...
esPrefijo?(e $\bullet$ s, <>) $\equiv$ ...
esPrefijo?(<>, s) $\equiv$ ...

BUSCAR: busca una secuencia dentro de otra. Si la secuencia buscada está una o más veces, la función devuelve la posición de la primera aparición; si no está, la función se indefine.
Solución:
¿Qué sería que la operación se indefina para algunos casos? Simplemente eso, que no está definida. Es decir, no debemos agregar ninguna restricción particular.

Buscar : secu(α) $\times$ secu(α) $\longrightarrow$ nat
Buscar(s, $e \bullet t$) $\equiv$ **if** EsPrefijo?(s, $e \bullet t$) **then** 0 **else** 1 + Buscar(s, t) **fi**

ESTÁORDENADA?: verifica si una secuencia está ordenada de menor a mayor.
Solución:

estáOrdenada? : secu(α) $\longrightarrow$ bool
estáOrdenada?(<>) $\equiv$ true
estáOrdenada?(e $\bullet$ s) $\equiv$ **if** vacía?(s) **then** true **else** e $\leq$ prim(s) $\wedge$ estáOrdenada?(s) **fi**

INSERTARORDENADA: dados una secuencia *so* (que debe estar ordenada) y un elemento e (de género α) inserta e en *so* de manera ordenada.
Solución:
Acá tenemos nuevamente una restricción, y es que la secuencia s debe estar ordenada. Utilizamos la operación definida previamente.

insertarOrdenada : secu(α) s $\times$ α $\longrightarrow$ secu(α) {estáOrdenada?(s)}
insertarOrdenada(<>, e) $\equiv$ $e \bullet$ <>
insertarOrdenada(a $\bullet$ s, e) $\equiv$ **if** a $\geq$ e **then** e $\bullet$ (a $\bullet$ s) **else** a $\bullet$ insertarOrdenada(s, e) **fi**

CANTIDADAPARICIONES: dados una secuencia y un α devuelve la cantidad de apariciones del elemento en la secuencia.
Solución:

cantidadApariciones : secu(α) $\times$ α $\longrightarrow$ nat
cantidadApariciones(<>,e) $\equiv$ 0
cantidadApariciones(a $\bullet$ s, e) $\equiv$ cantidadApariciones(s, e) + **if** a = e **then** 1 **else** 0 **fi**

ESPERMUTACIÓN?: chequea si dos secuencias dadas son permutación una de otra.
Pista: utilizar *CantidadApariciones*.
Solución:
Como el enunciado sugiere usar la operación anterior, intentemos con eso. Probemos con la siguiente axiomatización.
esPermutación?(<>, b) $\equiv$ vacía?(b)
esPermutación?(e $\bullet$ s, b) $\equiv$ cantidadApariciones(b, e) = cantidadApariciones(e $\bullet$ s, e) $\wedge$ ...
Y acá tenemos un problema. En los puntos suspensivos me gustaría poner una expresión que recursivamente me ayude a saber si las secuencias son permutación. El problema que tengo es que un llamado recursivo requiere que alguna de las listas (en este caso la primera) sea más pequeña para poder efectivamente llegar al caso base. Si achico cualquiera de las dos, pierdo la información de los elementos que saqué. Veamos un ejemplo:

$$a = [2, 4, 2, 2] \text{ y } b = [4, 2, 2, 2]$$

Como la lista a no es vacía usamos el segundo axioma. Vemos que cantidadApariciones(a, 2) = cantidadApariciones(b, 2). Supongamos que ahora miramos esPermutacion?(fin(a), b). Tenemos que ver si $[4, 2, 2]$ es permutación de $[4, 2, 2, 2]$ y no lo es. Por reducir a perdí ese 2.
Una idea para solucionar esto (puede haber más formas) es usar una operación auxiliar. Podría, por ejemplo, tener como parámetros tres secuencias: dos, que son las que quiero saber si son permutación (a y b) y una tercera l que sea la secuencia de elementos que tengo que chequear que aparezcan igual cantidad de veces en a y b. Esta l me va a permitir reducir mi problema, que ahora es "chequear si todos los elementos de l aparecen una igual cantidad de veces en a y en b". Nuestra operación se vería así:

igualCantApariciones : secu(α) $\times$ secu(α) $\times$ secu(α) $\longrightarrow$ bool
igualCantApariciones(a, b, <>) $\equiv$ true
igualCantApariciones(a, b, e $\bullet$ s) $\equiv$ cantidadApariciones(a, e) = cantidadApariciones(b, e)
$\wedge$ igualCantApariciones(a, b, s)

Nunca me deshago de las secuencias originales, entonces no pierdo información. Finalmente, para saber si son permutación:
esPermutación?(a, b) $\equiv$ |a| = |b| $\wedge$ igualCantApariciones(a, b, a)

Ejercicio 2:

#Hojas cuenta la cantidad de hojas del árbol.
Solución:
Una hoja es un nodo que no tiene hijos. Es decir, el árbol *nil* tiene 0 hojas, mientras que un árbol bin(i, r, d) tendrá las hojas del lado derecho más las hojas del lado izquierdo. En el caso de que ambos lados sean nil, entonces estamos en presencia de una hoja y en ese caso devolvemos 1.

#Hojas : ab(α) $\longrightarrow$ nat
#Hojas(nil) $\equiv$ 0
#Hojas(bin(i,r,d)) $\equiv$ **if** nil?(i) $\wedge$ nil?(d) nil **then** 1 **else** #Hojas(i) + #Hojas(d) **fi**

DegeneradoAIzquierda chequea si todo nodo interno (no hoja) tiene sólo subárbol izquierdo.
Solución:
En este caso un árbol será degenerado a izquierda si cada subárbol interno también es degenerado a izquierda.

DegeneradoAIzquierda : ab(α) $\longrightarrow$ bool
DegeneradoAIzquerda(nil) $\equiv$ true
DegeneradoAIzquerda(bin(i,r,d)) $\equiv$ **if** nil?(d) **then** DegeneradoAIzquerda(i) **else** false **fi**

ZigZag chequea si todo el árbol es degenerado con direcciones alternadas.
Solución:
Lo primero que podemos pensar al leer esta clase de ejercicios es qué forma tienen los arboles de la familia de arboles binarios "ZigZag". En este caso hay 4 tipos:

- El arbol vacío: nil.
- El arbol de un único nodo: una hoja.
- Un arbol zigzageante que empieza por izquierda.
- Un arbol zigzageante que empieza por derecha.

Habiendo hecho esta caracterización, que en particular es excluyente, podemos escribir la axiomatización de la operación como la disyuntiva exclusiva entre estas cuatro clases:

ZigZag : ab(α) a $\longrightarrow$ bool
ZigZag(a) $\equiv$ nil?(a) $\vee$ esHoja(a) $\vee_L$ (ZigZagDireccionado(a,izq) $\oplus^1$ZigZagDireccionado(a,der))

Coloquialmente esto nos dice que para que el arbol sea ZigZag tiene que ser de alguno de estos cuatro tipos.
La información de "para que lado empieza" la podemos codificar en el segundo parametro de la función ZigZag-Direccionado. Para eso podemos usar un renombre Dir del tipo ya existente Bool y renombrar los generadores de este como true→izq, false→der.

Supongamos que el arbol no es "nil" ni es una hoja. Sin perdida de generalidad pensemos el caso en el cual queremos fijarnos si un arbol empieza por izquierda (el caso derecha es análogo). Para empezar es necesario fijarse que a no tenga un hijo derecho. Si es así, como ya sabemos que no es hoja, entonces sabemos que tiene un hijo izquierdo. Teniendo eso ¿qué es lo que resta que valga? seguro es necesario que el hijo izquierdo de a solo tenga un hijo derecho y a su vez este debe tener un hijo izquierdo únicamente y así para hasta llegar a una hoja. Esencialmente el hijo izquierdo de a debe ser zigzageante pero empezando por *derecha*. Veamos que la siguiente axiomatización cumple con esta idea:

ZigZagDireccionado : ab(α) a × Dir d $\longrightarrow$ bool {$\neg$nil(a)}
ZigZagDireccionado(a,d) $\equiv$ **if** esHoja?(a) **then**
true
else
if ($d \wedge \neg$nil?(der(a)) $\vee$ ($\neg d \wedge \neg$nil?(izq(a))) **then**
false
else
ZigZagDireccionado(hijoNoNil(a),$\neg d$)
fi
fi

hijoNoNil : ab(α) a $\longrightarrow$ ab(α) {$\neg$nil?(a) $\wedge$ $\neg$esHoja?(a) $\wedge$ ($\neg$nil?(izq(a)) $\oplus$ $\neg$nil?(der(a)))}
hijoNoNil(a) $\equiv$ **if** $\neg$nil?(izq(a)) **then** izq(a) **else** der(a) **fi**

En principio el caso base es igual a la situación que ya tuvimos en cuenta para la función ZigZag, pero es necesario ponerlo por los mismos llamados recursivos de la función. En el segundo "If" nos fijamos que si me pregunto por la dirección d entonces el arbol no tenga un hijo en la dirección contraria. Finalmente si se llega al tercer "If" nos encontramos con que el arbol no es ni una hoja y no tiene un hijo contrario a d (ni es nil), entonces necesariamente tiene un hijo en la dirección d. Por lo tanto es necesario que este hijo sea zigzageante empezando por la dirección contraria a d: $\neg d$.
Como función auxiliar utilizamos hijoNoNil cuya restricción es precisamente el estado en el que la llamamos y nos devuelve el (único) hijo no nil de a. En particular esta función nos asegura que se cumpla la restricción en el llamado recursivo de ZigZagDireccionado.

ÚltimoNivelCompleto devuelve el número del último nivel que está completo (es decir, aquél que tiene todos los nodos posibles).
Solución:
Acá la clave es no confundirse haciendo malabares de todos los casos posibles. Pensemos estando posicionados en cualquier árbol, ¿cuál es el último nivel completo? pues el mínimo entre los máximos niveles completos de sus hijos sumado el nivel actual. Para diferenciar el hecho de tener un árbol vacío y un árbol que sea hoja, diremos que el último nivel completo de un árbol vacío es 0 mientras que en el caso de una hoja es 1.

ÚltimoNivelCompleto : ab(α) $\longrightarrow$ nat
ÚltimoNivelCompleto(ab) $\equiv$ **if** nil?(ab) **then**
0
else
1 + mín{ÚltimoNivelCompleto(izq(ab), ÚltimoNivelCompleto(der(ab))}
fi

Ejercicio 3:

Altura nos pide obtener la altura de un rosetree.

altura : rt(α) r $\longrightarrow$ Nat
altura(r) $\equiv$ **if** vacía?(hijos(r)) **then** 0 **else** 1+maxAlturaHijos(hijos(r)) **fi**

maxAlturaHijos : secu(rt(α)) sr $\longrightarrow$ Nat {$\neg$vacía?(sr)}
maxAlturaHijos(sr) $\equiv$ **if** long(sr)=1 **then**
altura(prim(sr))
else
Max(altura(prim(sr)),maxAlturaHijos(fin(sr)))
fi

[1]Donde $\oplus$ es el operador binario de la disjuncion exclusiva (*xor*). Tarea: extender el TAD Bool con la operación $\oplus$

CantHojas nos pide la cantidad de hojas.

cantHojas : rt(α) r $\longrightarrow$ Nat
cantHojas(r) $\equiv$ **if** vacía?(hijos(r)) **then** 1 **else** cantHojasHijos(hijos(r)) **fi**

cantHojasHijos : secu(rt(α)) sr $\longrightarrow$ Nat
maxAlturaHijos(sr) $\equiv$ **if** vacía?(sr) **then** 0 **else** cantHojas(prim(sr))+cantHojasHijos(fin(sr))) **fi**

Podar elimina todas las hojas.
Como no existe el rosetree vacío y un rosetree con solo la raiz es una hoja, restringimos a que no puede ser el rosetree de un solo nodo.

podar : rt(α) r $\longrightarrow$ rt(α) {¬vacía?(hijos(r))}
podar(r) $\equiv$ rose(raiz(r), podarHijos(r)

podarHijos : secu(rt(α)) sr $\longrightarrow$ secu(rt(α))
podarHijos(sr) $\equiv$ **if** vacía?(sr) **then**
<>
else
if vacía?(hijos(prim(sr))) **then**
podarHijos(fin(sr))
else
podar(prim(sr)) • podarHijos(fin(sr))
fi
fi

Ramas. Se nos pide obtener todas las ramas de un rosetree cuya longitud sea menor o igual a un valor n dado. Una rama es una secuencia de nodos del árbol que va desde la raíz hasta una de sus hojas.

Podemos notar que estos ejercicios tienen una estructura muy parecida, todos operan sobre una secuencia de rosetrees para resolver los casos recursivos.
Por otro lado, los ejercicios que vienen a continuación se resuelven en 2 etapas, una consiste en obtener más información que la solicitada y la siguiente en filtrar por la condición requerida.

En este caso, supongamos que conociésemos todas las ramas, entonces para resolver este ejercicio tendríamos que filtrar por longitud. Por lo tanto axiomatizemos primero una función que nos devuelva todas las ramas y luego axiomaticemos el filtrado por longitud.

ramas : rt(α) $\longrightarrow$ secu(secu(α))
ramas(r) $\equiv$ **if** vacía?(hijos(r)) **then** $\langle$ $\langle$raiz(r)$\rangle$ $\rangle$ **else** prefijarEnTodos(raiz(r), ramasHijos(hijos(r)) **fi**

ramasHijos : secu(rt(α)) $\longrightarrow$ secu(secu(α))
ramasHijos(sr) $\equiv$ **if** vacía(sr) **then** <> **else** ramas(prim(sr)) & ramasHijos(fin(sr)) **fi**

Donde *prefijarEnTodos* toma el elemento raíz de tipo α y lo agrega como cabecera a cada una de las secuencias obtenidas por los hijos. Lo dejamos de tarea. Luego nos queda solo el filtrado:

ramasN : rt(α) r $\times$ Nat n $\longrightarrow$ secu(secu(α))
ramasN(r, n) $\equiv$ soloMenoresA(n,ramas(r))

soloMenoresA : Nat n $\times$ secu(secu(α)) $\longrightarrow$ secu(secu(α))
soloMenoresA(n, sr) $\equiv$ **if** vacía?(sr) **then**
<>
else
if long(prim(sr)) $\leq$ n **then**
prim(sr) • soloMenoresA(n,fin(sr))
else
soloMenoresA(n,fin(sr))
fi
fi

NivelN. Dado un número natural n, devolver una secuencia con los elementos del nivel n enumerados de izquierda a derecha. Los elementos de nivel n son aquellos nodos que se encuentran a distancia n de la raíz del rosetree.

nivelN : rt(α) r × Nat n $\longrightarrow$ secu(α)
nivelN(r, n) $\equiv$ **if** $n = 0$ **then** raiz(r) • <> **else** nivelNrec(hijos(r), $n - 1$) **fi**

nivelNrec : secu(rt(α)) sr × Nat n $\longrightarrow$ secu(α)
nivelNrec(sr, n) $\equiv$ **if** vacía?(sr) **then** <> **else** nivelN(prim(sr),n) & nivelNrec(fin(sr),n) **fi**

MásLargasConRepetidos consiste en obtener el conjunto de las ramas más largas de un rosetree que contienen al menos un elemento repetido.

Notemos algo importante. Para poder resolver el ejercicio utilizamos funciones más pequeñas y fáciles de definir, una sería obtener la máxima longitud y otra quedarte con las ramas que además de tener esa longitud tienen repetidos. Dividir el problema más grande en subproblemas es lo que permite resolver todo más fácilmente.

máximasYrepiten : rt(α) r $\longrightarrow$ conj(secu(α))
máximasYrepiten(r) $\equiv$ filtrar(ramas(r), maxLong(ramas(r))

filtrar : secu(secu(α)) × Nat $\longrightarrow$ conj(secu(α))
filtrar(s, M) $\equiv$ **if** vacía?(s) **then**
∅
else
filtrar(fin(s), M) ∪ **if** long(prim(s))= M ∧ tieneRepetidos(prim(s)) **then** {prim(s)} **else** ∅ **fi**
fi

Las operaciones maxLong y tieneRepetidos quedan de tarea.

2. Especificación

Ejercicio 1:

A. Fila normal

TAD FILA

géneros fila

igualdad observacional

$$(\forall f_1, f_2 : \text{fila}) \left(f_1 =_{\text{obs}} f_2 \iff \begin{pmatrix} (\forall p : persona)\ \text{Esperando}(p,\ f_1) =_{\text{obs}} \text{Esperando}(p,\ f_2) \\ \wedge_{\text{L}}\ (\text{Esperando}(p, f_1) \Rightarrow_{\text{L}} \text{Posición}(p, f_1) =_{\text{obs}} \text{Posición}(p, f_2)) \end{pmatrix} \right)$$

observadores básicos

Esperando	: persona $\times$ fila	$\longrightarrow$ bool	
Posición	: persona p $\times$ fila f	$\longrightarrow$ nat	$\{\text{Esperando}(p, f)\}$

generadores

AbrirVentanilla	:	$\longrightarrow$ fila	
Llegar	: persona p $\times$ fila f	$\longrightarrow$ fila	$\{\neg\ \text{Esperando}(p, f)\}$
Atender	: fila f	$\longrightarrow$ fila	$\{\neg\ \text{Vacía}(f)\}$

otras operaciones

Longitud	: fila	$\longrightarrow$ nat	
Vacía	: fila	$\longrightarrow$ bool	

(...)

Fin TAD

¿Los observadores están bien elegidos? Veamos...

Recordemos que lo que estamos especificando es una fila de clientes, con lo cual nuestros observadores deberían ser capaces de observarla. Con *Esperando*, podemos conocer qué personas se encuentran en la fila; y con *Posición*, en qué posición se encuentra cada una de esas personas. El enunciado no parece requerir nada más, con lo cual este par de observadores serían suficientes.

Un hecho que puede ser soprendente es que *Longitud* no es parte de los observadores. El motivo es que entonces nuestro conjunto de observadores ya no sería minimal, pues *Longitud* es deducible a partir de *Esperando*: si yo conozco qué personas están en mi fila, las puedo contar y obtener la longitud de la misma. Sin embargo, aunque esto suena razonable, hay algo curioso que ocurre al axiomatizar, lo que estará comentado después. *Vacía* también es deducible a partir de *Esperando*: una fila está vacia cuando ninguna personá está esperando.

Con respecto a los generadores...

Notar como *AbrirVentanilla* y *Llegar* nos permiten armar cualquier fila que querramos. ¿Y qué pasa con *Atender* entonces? Si tuviésemos una fila con varias personas, podríamos axiomatizar *Atender* armando una nueva fila sin el cliente de adelante, utilizando una vez *AbrirVentanilla* y varias veces *Llegar*. ¡Esto implica que agregar *Atender* hace que los generadores no sean minimales, pues ya podemos construir todas las instancias de nuestro TAD con las otras dos operaciones! ¿Por qué lo incluimos entonces? Por qué es más cómodo de axiomatizar: minimalidad en generadores es deseable, pero este es uno de los casos en qué añadir redundancia nos hace la vida más fácil.

Esperando(p, AbrirVentanilla)	$\equiv$	*// Si la fila acaba de abrir, seguro que p no está esperando* $false$
Esperando(p, Llegar(pl, f))	$\equiv$	*// Si alguien acaba de llegar a la fila, p esta esperando si es el que acaba* *// de llegar o ya estaba desde antes* $p = pl \vee \text{Esperando}(p, f)$

Esperando(p, Atender(f))	$\equiv$	*// Si alguien acaba de ser atendido, p esta esperando si ya estaba // desde antes y no fue el atendido* Esperando(p, f) $\wedge_{\text{L}}$ Posicion(p, f) $\neq 1$

Notar que estamos utilizando *Longitud* que es otra operación para axiomatizar un observador. ¿Eso se podía? ¡Sí! Lo único importante es que en el lado derecho del $\equiv$ estemos aplicando operaciones sobre instancias más pequeñas que del lado izquierdo. Mientras respetemos esto, podemos tranquilamente axiomatizar una operación a utilizando una operación b, e incluso a su vez b con a, sin importar si sean otras operaciones ú observadores (mejor explicado en el apunte de especificación).
Lo que acabo de escribir *es un tema distinto* a decidir si axiomatizar en base a generadores u observadores.

Posicion(p, Llegar(pl, f))	$\equiv$	**if** $p = pl$ **then** Longitud(f) + 1 **else** Posicion(p, f) **fi**
Posicion(p, Atender(f))	$\equiv$	Posicion(p, f) - 1

Notar como no axiomatizamos sobre *AbrirVentanilla*. Estaría mal hacerlo, pues la restricción de *Posición* es que p esté esperando en la fila, y ninguna persona puede estar esperando en una fila que acaba de abrir. ¿El caso base entonces donde está? Es el momento en que p entró a la fila. Además, en la axiomatización sobre *Atender*, no chequeamos si p fue la persona atendida: seguro que no, pues de lo contrario p habría salido de la fila y la restricción de *Posición* se invalidaría.
Ambos de estos casos ilustran un concepto importante: Antes de escribir una axioma, ¡tener presente que la restricción vale! Y actuar en consecuencia... (esta es una de las razones por las que es tan importante **escribir las retricciones ni bien escribimos las signaturas**).

Longitud(AbrirVentanilla)	$\equiv$	0
Longitud(Llegar(pl, f))	$\equiv$	Longitud(f) + 1
Longitud(Atender(f))	$\equiv$	Longitud(f) $-$ 1
Vacia(f)	$\equiv$	Longitud(f) $= 0$

Las otras operaciones deberían poder axiomatizarse a partir de los observadores. La razón es que los observadores, por definición, son todo lo que nos interesa de las instancias de nuestro TAD. Pero en este caso tenemos un problema. Dijimos que *Longitud* era deducible a partir de *Esperando* porque bastaba con contar las personas que estaban esperando. ¿Pero cómo hariamos esto en nuestro lenguaje? Tendríamos que iterar sobre TODAS las personas y quedarnos con aquellas tales que *Esperando* de *true*. ¿Pero cómo generamos el conjunto de todas las personas? ¿No sería infinito si persona es renombre de *Nat* por ejemplo? Por este tipo de cuestiones, en este caso en particular se nos complica axiomatizar en base a los observadores, razón por la cual recurrimos a axiomatizar en base a los generadores. Notar que podríamos evitar este problema si tuviesemos un observador de *PersonasEsperando* que nos devuelva un conjunto de las personas esperando: la información sería la misma que en *Esperando*, pero podríamos iterarlo en nuestro lenguaje más facilmente.

B. Fila dinámica

TAD Fila

géneros fila

igualdad observacional

$$(\forall f_1, f_2 : \text{fila}) \left(f_1 =_{\text{obs}} f_2 \iff \begin{pmatrix} (\forall p : \textit{persona})\ \text{Esperando}(p,\ f_1) =_{\text{obs}} \text{Esperando}(p,\ f_2) \\ \wedge_{\text{L}}\ (\text{Esperando}(p,\ f_1) \Rightarrow_{\text{L}} (\text{Posición}(p,\ f_1) =_{\text{obs}} \text{Posición}(p,\ f_2) \wedge \text{SeColó?}(p,\ f_1) =_{\text{obs}} \text{SeColó?}(p,\ f_2))) \end{pmatrix} \right)$$

observadores básicos

Esperando	: persona $\times$ fila	$\longrightarrow$ bool	
Posición	: persona p $\times$ fila f	$\longrightarrow$ nat	{Esperando(p, f)}

SeColó?	: persona p × fila f ⟶ bool		{Esperando(p, f)}
generadores			
AbrirVentanilla	: ⟶ fila		
Llegar	: persona p × fila f ⟶ fila		{¬ Esperando(p, f)}
ColarseAdelanteDe	: persona p × persona q × fila f ⟶ fila		{¬Esperando(p, f) ∧ Esperando(q, f)}
Atender	: fila f ⟶ fila		{¬ Vacía(f)}
Retirarse	: persona p × fila f ⟶ fila		{Esperando(p, f)}
otras operaciones			
Longitud	: fila ⟶ nat		
Vacía	: fila ⟶ bool		

(...)

Fin TAD

- De igual forma que en la anterior sección, *Retirarse* y *Atender* son generadores para facilitar la axiomatización. Podrían ser tranquilamente otras operaciones.

Esperando(p, AbrirVentanilla) ≡ $false$
Esperando(p, Llegar(pl, f)) ≡ $p = pl \vee$ Esperando(p, f)
Esperando(p, Atender(f)) ≡ Esperando(p, f) $\wedge_{\text{L}}$ Posicion(p, f) $\neq 1$
Esperando(p, ColarseAdelanteDe(pc, pf, f)) ≡ $p = pc \vee$ Esperando(p, f)
Esperando(p, Retirarse(pr, f)) ≡ Esperando(p, f) $\wedge\ p \neq pr$

Posicion(p, Llegar(pl, f)) ≡ **if** $p = pl$ **then**
    Longitud(f) + 1
  else
    Posicion(p, f)
  fi
Posicion(p, Atender(f)) ≡ Posicion(p, f) - 1
Posicion(p, ColarseAdelanteDe(pc, pf, f)) ≡ **if** $p = pc$ **then**
    Posicion(pf, f)
  else
    // Sumamos uno si pc se coló más adelante de p
    Posicion(p, f) + β(Posicion(p, f) ≥ Posicion(pf, f))
  fi
Posicion(p, Retirarse(pr, f)) ≡ *// Restamos uno si pr se retiró más adelante de p*
  Posicion(p, f) - β(Posicion(p, f) > Posicion(pr, f))

SeColó?(p, Llegar(pl, f)) ≡ **if** $p = pl$ **then** $false$ **else** SeColó?(p, f) **fi**
SeColó?(p, Atender(f)) ≡ SeColó?(p, f)
SeColó?(p, ColarseAdelanteDe(pc, pf, f)) ≡ **if** $p = pc$ **then** $true$ **else** SeColó?(p, f) **fi**
SeColó?(p, Retirarse(pr, f)) ≡ SeColó?(p, f)

Longitud(AbrirVentanilla) ≡ 0
Longitud(Llegar(pl, f)) ≡ Longitud(f) + 1
Longitud(Atender(f)) ≡ Longitud(f) − 1
Longitud(ColarseAdelanteDe(pc, pf, f)) ≡ Longitud(f) + 1
Longitud(Retirarse(pr, f)) ≡ Longitud(f) − 1

Vacia(f) ≡ Longitud(f) = 0

- De nuevo terminamos axiomatizando *Longitud* en base a los generadores porque no es posible a partir de *Esperando*.

C. Fila con info histórica

TAD FILA

géneros fila

igualdad observacional

$$(\forall f_1, f_2 : \text{fila}) \left(f_1 =_{\text{obs}} f_2 \iff \begin{pmatrix} (\forall p : \mathit{persona})\ \text{Esperando}(p,\ f_1) =_{\text{obs}} \text{Esperando}(p,\ f_2) \\ \wedge_{\text{L}}\ (\text{Esperando}(p,\ f_1) \Rightarrow_{\text{L}} (\text{Posición}(p,\ f_1) =_{\text{obs}} \text{Posición}(p,\ f_2) \wedge \text{SeColó?}(p,\ f_1) =_{\text{obs}} \text{SeColó?}(p,\ f_2))) \wedge \\ \text{Entro?}(p,\ f_1) =_{\text{obs}} \text{Entro?}(p,\ f_2) \wedge \\ \text{FueAtendido?}(p,\ f_1) =_{\text{obs}} \text{FueAtendido?}(p,\ f_2) \end{pmatrix} \right)$$

observadores básicos

Esperando	: persona × fila $\longrightarrow$ bool	
Posición	: persona p × fila f $\longrightarrow$ nat	{Esperando(p, f)}
SeColó?	: persona p × fila f $\longrightarrow$ bool	{Esperando(p, f)}
Entro?	: persona × fila $\longrightarrow$ bool	
FueAtendido?	: persona × fila $\longrightarrow$ bool	

generadores

AbrirVentanilla	: $\longrightarrow$ fila	
Llegar	: persona p × fila f $\longrightarrow$ fila	{¬ Esperando(p, f)}
ColarseAdelanteDe	: persona p × persona q × fila f $\longrightarrow$ fila	{¬Esperando(p, f) ∧ Esperando(q, f)}
Atender	: fila f $\longrightarrow$ fila	{¬ Vacía(f)}
Retirarse	: persona p × fila f $\longrightarrow$ fila	{Esperando(p, f)}

otras operaciones

Longitud	: fila $\longrightarrow$ nat
Vacía	: fila $\longrightarrow$ bool

(...)

Fin TAD

- Notar que hace falta agregar como observadores a *Entro?* y *FueAtendido?*. Los observadores que teníamos antes se 'olvidaban' de los clientes una vez que salían de la fila, así que seguro que no nos servían.
- ¡Ahora *Atender* sí **debe ser un generador**! Sin ella, no podemos saber si un cliente que alguna vez entró a la fila fue atendido, o se retiró antes.
- *Retirarse* **también es necesario como generador**, aunque esto es más dificil de ver. Si fuera otra operación y no un generador, axiomatizaríamos *Retirarse* armando una fila idéntica a la de antes, pero sin el *Llegar* (o el *Colarse*) del cliente que se acaba de retirar. Pero entonces... Nos quedaría una fila en la que dicho cliente nunca entró a la fila... Y si un cliente *Entro* alguna vez a la fila es justamente algo que queremos observar.

// Esperando, Posicion, SeColo?, Longitud y Vacia se axiomatizan igual que antes
(...)

Entro?(p, AbrirVentanilla)	$\equiv\ false$
Entro?(p, Llegar(pl, f))	$\equiv\ p = pl \vee$ Entro?(p, f)
Entro?(p, Atender(f))	$\equiv$ Entro?(p, f)
Entro?(p, ColarseAdelanteDe(pc, pf, f))	$\equiv\ p = pc \vee$ Entro?(p, f)
Entro?(p, Retirarse(pr, f))	$\equiv$ Entro?(p, f)
FueAtendido?(p, AbrirVentanilla)	$\equiv\ false$
FueAtendido?(p, Llegar(pl, f))	$\equiv$ FueAtendido?(p, f)
FueAtendido?(p, Atender(f))	$\equiv$ (Esperando?(p, f) $\wedge_{\text{L}}$ Posicion(p) = 1) $\vee$ FueAtendido?(p, f)
FueAtendido?(p, ColarseAdelanteDe(pc, pf, f))	$\equiv$ FueAtendido?(p, f)
FueAtendido?(p, Retirarse(pr, f))	$\equiv$ FueAtendido?(p, f)

Ejercicio 2:

Algunas consideraciones:

- Pendientes es una secuencia porque importa el orden de llegada en caso de empate de distancia.
- Solicitar recibe una dirección, y no importa si hay o no un técnico atendiendo ahí, de hecho puede haber más de uno a la vez.
- Finalizar recibe un técnico, este no *decide* a donde va a estar una vez que termine su trabajo, eso lo decide la especificación brindada (comportamiento esperado del sistema) y se puede observar a través de los observadores.
- Sería un error tenér MaxVisitas en vez de un contador de visitas por dirección, tarea: pensar porque.

TAD TECNICOS A DOMICILIO

géneros TaD

igualdad observacional

$$(\forall t, p : \text{TaD}) \left(t =_{\text{obs}} p \iff \begin{pmatrix} \text{disponibles(t)} =_{\text{obs}} \text{disponibles(p)} & \wedge \\ \text{ocupados(t)} =_{\text{obs}} \text{ocupados(p)} & \wedge \\ \text{pendientes(t)} =_{\text{obs}} \text{pendientes(p)} \wedge_{\text{L}} & \\ (\forall j\text{: Tec})(\text{j} \in \text{disponibles(t)} \cup \text{ocupados(t)} \Rightarrow_{\text{L}} & \\ (\text{ubicacion(t,j)} =_{\text{obs}} \text{ubicacion(p,j)}) \wedge (\#\text{visitas(t,j)} =_{\text{obs}} & \\ \#\text{visitas(p,j)})) & \end{pmatrix} \right)$$

observadores básicos

disponibles : TaD $\longrightarrow$ conj(Tec)
ocupados : TaD $\longrightarrow$ conj(Tec)
pendientes : TaD $\longrightarrow$ secu(Dir)
ubicación : TaD t × Tec p $\longrightarrow$ Dir {p∈ocupados(t)}
#visitas : TaD t × Tec p × Dir d $\longrightarrow$ Nat {p∈disponibles(t)∪ocupados(t)}

generadores

crear : conj(Tec) c $\longrightarrow$ TaD {¬∅?(c)}
solicitar : TaD × Dir $\longrightarrow$ TaD
finalizar : TaD r × Tec t $\longrightarrow$ TaD {t∈Ocupados(r)}

otras operaciones

TecMásVisitaron : TaD t × Tec p × Dir d $\longrightarrow$ Conj(Tec)

axiomas $\forall t$: TaD, $\forall p, p'$: Tec, $\forall d, d'$: Dir

disponibles(crear(c)) ≡ c
disponibles(solicitar(t,d)) ≡ **if** ∅?(disponibles(t) **then** ∅ **else** sinUno(disponibles(t) **fi**
disponibles(finalizar(t,p)) ≡ disponibles(t) ∪ **if** vacía?(pendientes(t)) **then** {p} **else** ∅ **fi**
ocupados(crear(c)) ≡ ∅
ocupados(solicitar(t,d)) ≡ ocupados(t) ∪ **if** ∅?(disponibles(t) **then** ∅ **else** dameUno(disponibles(t) **fi**
ocupados(finalizar(t,p)) ≡ ocupados(t) \ **if** vacía?(pendientes(t)) **then** {p} **else** ∅ **fi**
pendientes(crear(c)) ≡ <>
pendientes(solicitar(t,d)) ≡ **if** ∅?(disponibles(t) **then** pendientes(t) ∘ d **else** <> **fi**
pendientes(finalizar(t,p)) ≡ **if** vacía?(pendientes(t)) **then**
<>
else
sacar(másCercano(pendientes(t),ubicación(t,p)), pendientes(t)
fi
ubicación(solicitar(t,d),p) ≡ **if** $p \in$ ocupados(t) **then** ubicación(t,p) **else** d **fi**
ubicación(finalizar(t,p),p') ≡ **if** $p \neq p'$ **then**
ubicación(t, p')
else
másCercano(pendientes(t, ubicación(t, p)
fi
#visitas(crear(c),p,d) ≡ 0

#visitas(solicitar(t,d'), p, d) $\equiv$ #visitas(t, p, d) + **if** $d = d'$ $\wedge_{\text{L}} \neg\emptyset?$(disponibles($t$) $\wedge_{\text{L}} p$ =dameUno(disponibles(t)) **then**
1
else
0
fi

#visitas(finalizar(t,p'), p, d) $\equiv$ #visitas(t, p, d) + **if** $p = p' \wedge$ ¬vacía?(pendientes(t)) $\wedge_{\text{L}}$d=másCercano(pendientes(t),ubicación(t, p)) **then**
1
else
0
fi

Donde utilizamos las funciones auxiliares:

másCercano : secu(Dir) $s \times$ Dir $d \longrightarrow$ Dir {¬vacía?(s)}

másCercano(s, d) $\equiv$ **if** long(s)=1 **then**
prim(s)
else
if dist(d,prim(s)) $\leq$ dist(d,másCercano(fin(s),d)) **then**
prim(s)
else
másCercano(fin(s),d)
fi
fi

sacar : secu(α) $s \times \alpha$ $e \longrightarrow$ secu(α)

sacar(s, e) $\equiv$ **if** vacía?(s) **then**
<>
else
if prim(s)=e **then** fin(s) **else** prim(s) • sacar(fin(s),e) **fi**
fi

Fin TAD