Algoritmos y Estructura de Datos I

Primer cuatrimestre de 2011

Práctica 5– Especificación con tipos compuestos

1. Tipo Racional

Sea el tipo de datos compuesto Racional que representa los números racionales.

```
tipo Racional {
  observador numerador (t: Racional) : ℤ;
  observador denominador (t: Racional) : ℤ;
  invariante denominador(r) ≠ 0;
  invariante mcd(denominador(r), numerador(r)) == 1;
  aux mcd (a,b: ℤ) : ℤ = max[i|i ← ℤ, a mód i == 0, b mód i == 0];
}
```

Ejercicio 1 ¿Es correcta la siguiente especificación para el producto entre dos números racionales r_1 y r_2?

```
problema producto (r1,r2:Racional) = result : Racional {
  asegura denominador(res) = denominador(r1) * denominador(r2);
  asegura numerador(res) = numerador(r1) * numerador(r2);
}
```

Ejercicio 2 Especificar las siguientes operaciones:

a) `problema nuevoRac` (r_1,r_2:$\mathbb{Z}$) = `result` : Racional que dados dos enteros, obtiene el racional $\frac{a}{b}$.
b) `problema menor` (r_1,r_2:Racional) = `result` : Bool , que devuelve verdadero si y sólo si r_1 es menor que r_2.
c) `problema suma` (r_1,r_2:Racional) = `result` : Racional , que devuelve la suma de r_1 y r_2.
d) `problema representacionEntera` (r:Racional) = `result` : $\mathbb{Z}$, que devuelve, *en caso de que sea posible*, el número entero equivalente a r.

2. Tipo Vector

```
tipo Vector {
  observador abscisa (t: Vector) : ℝ;
  observador ordenada (t: Vector) : ℝ;
}
```

Ejercicio 3 Sobre este tipo, especificar las siguientes operaciones:

a) `problema igualX` (v_1,v_2: Vector) = `result` : Bool , que devuelve True sii v_1 y v_2 tienen la misma componente x.
b) `problema igualY` (v_1,v_2: Vector) = `result` : Bool , que devuelve True sii v_1 y v_2 tienen la misma componente y.
c) `problema colineales` (v_1,v_2: Vector) = `result` : Bool , que devuelve True sii v_1 y v_2 son linealmente dependientes.
d) `problema módulo` (v: Vector) = `result` : $\mathbb{R}$, que devuelve el módulo de v.
e) `problema productoEscalar` (v_1,v_2: Vector) = `result` : $\mathbb{R}$.
f) `problema todosColineales` (V: [Vector]) = `result` : Bool , que dice si todos los vectores de la lista son colineales tomados de a pares.
g) `problema resultante` (V: [Vector]) = `result` : Vector , que calcula la resultante entre todos los vectores de la lista.
h) `problema primerCuadrante` (V: [Vector]) = `result` : [Vector] , que devuelve una lista con todos los vectores de V que están en el primer cuadrante.
i) `problema equilibrio` (V: [Vector]) = `result` : Bool , que indica si la resultante entre todos los vectores de la lista es cero.

3. Trenes y Terminales

Se cuenta con el tipo enumerado `tipo` TipoVagon = Locomotora, Pasajero, Carga ;
y con el tipo compuesto Vagon:

```
tipo Vagon {
  observador tipo (v: Vagon) : TipoVagon ;
  observador peso (v: Vagon) : ℤ ;
  observador carga (v: Vagon) : ℤ ;
       requiere tipo(v) ≠ Locomotora ;
  observador potencia (v: Vagon) : ℤ ;
       requiere tipo(v) == Locomotora ;
  invariante peso(v) > 0 ;
  invariante carga(v) > 0 ;
  invariante potencia(v) > 0 ;
}
```

Un vagón puede ser una locomotora, un coche de pasajeros o un vagón de carga. Todos los vagones tienen peso (en Toneladas). Una locomotora tiene una determinada potencia que se refleja en la cantidad de Toneladas que puede arrastrar. La carga en un coche de pasajeros es la cantidad máxima de pasajeros que puede transportar, mientras que en uno de carga, representa el tonelaje máximo que puede cargar.

También tenemos el tipo Tren que es un sinónimo de [Vagon]. En un tren pueden aparecer más de una locomotora y no necesariamente encabezando el convoy. Por último, se tiene el tipo compuesto Terminal:

```
tipo Terminal {
  observador cantA (t: Terminal) : ℤ ;
       Devuelve la cantidad de andenes
  observador longA (t: Terminal) : ℤ ;
       Devuelve la cantidad máxima de vagones por andén
  observador iesimoTren (i: ℤ, t: Terminal) : Tren ;
       requiere 0 ≤ i < cantA(t) ;
       Devuelve el tren en el i-ésimo andén
  invariante cantA(t) > 0 ;
  invariante longA > 0 ;
  invariante VagonesNoDemasiadoLargos(t) ;
}
```

Notar que si en un andén hay un tren de longitud cero, significa que el andén está vacío.

Ejercicio 4 Sobre el tipo Terminal especificar el `invariante VagonesNoDemasiadoLargos(t)` , que indica que en la terminal no hay vagones más largos que la longitud del anden.

Ejercicio 5 Especificar los siguientes problemas:

a) `problema cabeEn` (t, u: Tren) = `result` : Bool donde t y u son trenes conformados por vagones del tipo Carga. Devuelve True en caso de que el tren t tenga menor carga que el tren u.
b) Dado un tren, decidir si el tren se puede mover. Para que un tren se mueva, la suma de las potencias de las locomotoras tiene que ser mayor o igual a la suma de los pesos de todos los vagones suponiendo que todas las personas pesan 80kg.

4. Ejercicios tipo parcial

Ejercicio 6 Dado el tipo compuesto JugadorDeFutbol:

```
tipo JugadorDeFutbol {
  observador nombre (t: JugadorDeFutbol) : [Char] ;
  observador goles (t: JugadorDeFutbol) : ℤ ;
  observador infracciones (t: JugadorDeFutbol) : ℤ ;
  observador partidos (t: JugadorDeFutbol) : ℤ ;
  invariante goles(t) ≥ 0 ;
  invariante infracciones(t) ≥ 0 ;
  invariante partidos(t) ≥ 0 ;
}
```

Se pide definir las siguientes operaciones:

a) `problema esBuenJugador` (j: JugadorDeFutbol) = `result` : Bool , que devuelve True sii j hizo al menos 10 goles y cometió menos de 5 infracciones.
b) `problema unaFigura` (J:Conjunto⟨JugadorDeFutbol⟩) = `result` : JugadorDeFutbol , que devuelve el jugador más goleador entre todos los "buenos jugadores" del conjunto J (o uno de ellos, si hay más de uno), asumiendo que tal jugador exista.
c) `problema juegoSucio` (J:Conjunto⟨JugadorDeFutbol⟩) = `result` : Bool , que devuelve True sii todos los jugadores del conjunto J que hicieron algún gol y jugaron más de 2 partidos cometieron alguna infracción.
d) `problema juegoSucio` (J:Conjunto⟨JugadorDeFutbol⟩) = `result` : Bool , que devuelve True sii algún jugador del conjunto tiene un promedio de más de un gol por partido.

Ejercicio 7

Dado el tipo enumerado Barrio, que asumimos conocido, y los tipos compuestos Parada y Colectivo, con los siguientes observadores e invariantes:

`tipo Parada` {
  `observador idParada` (p: Parada) : $\mathbb{Z}$;
    Identifica unívocamente cada parada.
  `observador barrio` (p: Parada) : Barrio ;
    Indica el barrio al que pertenece la parada.
}

`tipo Colectivo` {
  `observador nroLinea` (c: Colectivo) : $\mathbb{Z}$;
    Indica el número de línea del colectivo.
  `observador recorrido` (c: Colectivo) : [Parada] ;
    Indica todas las paradas por las que pasa el colectivo, en el orden de su recorrido.
  `invariante` $nroLinea(c) > 0$;
  `invariante` $|recorrido(c)| \geq 2$;
  `invariante` $(\forall i \leftarrow [0..|recorrido(c)|, j \leftarrow (i..|recorrido(c)|))recorrido(c)[i] \neq recorrido(c)[j]$;
}

Se pide especificar los siguientes algoritmos:

a) Dado un conjunto de colectivos, decidir si existe alguno en el conjunto que recorra sólo paradas de un mismo barrio.
b) Dados dos colectivos c_1 y c_2, decidir si c_1 y c_2 comparten la cabecera, comparten la terminal y no comparten ninguna otra parada a lo largo de sus respectivos recorridos. La cabecera y la terminal son respectivamente la primera y la última de las paradas del recorrido de un colectivo.
c) Dado un barrio y un conjunto de colectivos, decidir si todo colectivo del conjunto que empieza y termina en el barrio dado, pasa también por al menos una parada de otro barrio.

Ejercicio 8

Se cuenta con el tipo enumerado `tipo` Color = Rojo, Negro ;
y con los siguientes tipos compuestos, que modelan una noche en el casino:

`tipo Ruleta` {
  `observador sesion` (r: Ruleta) : [$\mathbb{Z}$] ;
    Representa la secuencia de números que salieron durante la noche, en orden cronológico.
  `invariante` $(\forall i \leftarrow [0..|sesion(R)|))0 \leq (sesion(R))_i \leq 36$;
}

`tipo Apuesta` {
  `observador monto` (a: Apuesta) : $\mathbb{Z}$;
    Indica el monto de dinero que se apuesta.
  `observador color` (a: Apuesta) : Color ;
    Devuelve el color por el que se está apostando.
  `invariante` $monto(a) > 0$;
}

`tipo Jugador` {
  `observador ahorros` (j: Jugador) : $\mathbb{Z}$;
    Indica el monto de dinero que tiene un jugador antes de empezar a jugar.

`observador ruleta` (j: Jugador) : Ruleta ;
Indica la ruleta en la que jugó el jugador.
`observador apuestas` (j: Jugador) : [Apuesta] ;
Devuelve la secuencia de apuestas que el jugador realizó, en orden cronológico.
`invariante` $ahorros(j) > 0$;
`invariante` $|apuestas(j)| \leq |sesion(ruleta(j))|$;
`invariante` $NoSeQuedaSinPlata(j)$;
`aux NoSeQuedaSinPlata` (j:Jugador) : Bool $= \ldots$;
}

El jugador sólo apuesta a un color (rojo o negro) y juega toda la noche en la misma ruleta. Se asume que comienza a jugar desde que la sesión de ruleta comienza y a partir de ahí apuesta continuamente hasta que se retira. Cada vez que el jugador gana, la banca le paga el doble de lo apostado (si apuesta 10, cobra 20), y cuenta inicialmente con sus ahorros para apostar.

a) Los números en la ruleta (salvo el cero) tienen asociado un color: rojo para los números 1, 3, 5, 7, 9, 12, 14, 16, 18, 19, 21, 23, 25, 27, 30, 32, 34 y 36 y negro para los demás. ¿Cómo se podría escribir la expresión auxiliar `aux color` (n:$\mathbb{Z}$) : Color, que para cada uno de estos números, dé su color?
b) Un jugador no puede apostar en ningún momento más dinero que el que tiene. Escribir la expresión auxiliar `aux NoSeQuedaSinPlata` (j:Jugador) : Bool (que define un invariante para el tipo) que exprese lo siguiente: "en todo momento el jugador tuvo suficiente plata (teniendo en cuenta los resultados de las apuestas anteriores) para realizar cada una de sus apuestas".
c) Especificar una operación que, dado un jugador, permita determinar cuánto fue el mayor monto de dinero que el jugador llegó a tener durante el transcurso de sus apuestas.
d) Especificar una operación que, dado un jugador, devuelva cuál fue la mayor racha negativa (cantidad de desaciertos consecutivos de sus apuestas) que el jugador debió afrontar durante sus apuestas.

Ejercicio 9

Sean los siguientes tipos compuestos y sus observadores:

`tipo Propietario` {
`observador nombre` (p: Propietario) : [Char] ;
Representa el nombre y apellido del propietario.
`observador tieneProblemasCon` (p: Propietario) : Conjunto⟨Propietario⟩ ;
El conjunto de propietarios con los que tiene problemas personales.
}

`tipo Edificio` {
`observador direccion` (e: Edificio) : [Char] ;
Representa el domicilio postal del edificio.
`observador cantPisos` (e: Edificio) : $\mathbb{Z}$;
`observador cantDeptos` (e: Edificio, p: $\mathbb{Z}$) : $\mathbb{Z}$;
`requiere` $1 \leq p \leq cantPisos(E)$;
La cantidad de departamentos en cierto piso del edificio.
`observador propietario` (e: Edificio, p: $\mathbb{Z}$, d: $\mathbb{Z}$) : Propietario ;
`requiere` $1 \leq p \leq cantPisos(E)$;
`requiere` $1 \leq d \leq cantDeptos(E, p)$;
Dado un piso y un departamento, devuelve su propietario.
`observador deudaExpensas` (e: Edificio, p: $\mathbb{Z}$, d: Char) : $\mathbb{R}$;
`requiere` $1 \leq p \leq cantPisos(E)$;
`requiere` $1 \leq d \leq cantDeptos(E, p)$;
El monto de expensas adeudado por el departamento indicado.
`invariante` $cantPisos(E) > 0$;
`invariante` $(\forall p \leftarrow [1..cantPisos(E)])cantDeptos(E, p) > 0$;
}

Notar que:

- Un propietario puede poseer más de un departamento, incluso en el mismo edificio.
- Aunque sea cierto que el propietario p_1 tiene problemas con p_2 (esto es, $p_2 \in tieneProblemasCon(p_1)$), no necesariamente p_2 tendrá problemas con p_1.

Se pide especificar las siguientes operaciones:

a) `problema propietarios` (e: Edificio) = `result` : Conjunto⟨Propietario⟩ , que dado un edificio devuelve el conjunto de todos los que sean propietarios dentro del edificio.
b) `problema cuantoDebePropietario` (E:Edificio, p:Propietario) = `result` : $\mathbb{Z}$, que dados un edificio y un propietario devuelve cuánto debe de expensas en el edificio. Si el propietario no tiene departamentos en el edificio debe devolver cero.
c) `problema elChanta` (E:Edificio) = `result` : Propietario , que devuelve al propietario que más debe en expensas dentro del edificio (o a alguno de ellos, si hay más de uno).
d) `problema propietariosConflictivos` (E:Edificio) = `result` : Conjunto⟨Propietario⟩ , que dado un edificio devuelve el conjunto de los propietarios que tienen problemas con sus vecinos. Dos propietarios son vecinos si sus departamentos están en el mismo piso del edificio y sus números de departamentos son consecutivos.
e) `problema esElInsoportable` (E:Edificio, p: Propietario) = `result` : Bool , que determina si el propietario dado es el propietario con la mayor cantidad de vecinos que tienen problemas con él (o alguno de ellos, si hay más de uno). Si p no es propietario en el edificio debe devolver False.

Ejercicio 10

Se cuenta con los siguientes tipos compuestos que modelan algunas características de los vuelos aéreos. Asumimos conocido el tipo enumerado Aeropuerto.

```
tipo FechaHora {
  observador minutosTotales (f: FechaHora) : ℤ;
      El monto de expensas adeudado por el departamento indicado. Devuelve la cantidad de minutos que han pasado desde
    el origen de los tiempos hasta la FechaHora indicada.
  invariante minutosTotales(FH) > 0;
}

tipo Vuelo {
  observador horaSalida (v: Vuelo) : FechaHora;
  observador horaLlegada (v: Vuelo) : FechaHora;
  observador precio (v: Vuelo) : ℝ;
  observador aeropOrigen (v: Vuelo) : Aeropuerto;
  observador aeropDestino (v: Vuelo) : Aeropuerto;
  invariante horaSalida(v) < horaLlegada(v);
  invariante precio(v) > 0;
}
```

Especificar los siguientes problemas:

a) Dados un conjunto de vuelos directos V, n aeropuertos $a_0 \dots a_n$ y una fecha/hora f, decidir si es posible tomar $n-1$ vuelos directos de V que comuniquen a_i con a_{i+1} $(0 \leq i < n-1)$ y llegar a a_n antes de la fecha/hora f.
b) Dado un conjunto de vuelos V, determinar si vale sobre él la propiedad "a mayor tiempo de viaje, mayor precio". Es decir, que no existan en V dos vuelos v_1 y v_2 tales que v_1 dure más que v_2 pero sea más barato.
c) Dados un conjunto de vuelos y dos aeropuertos origen y destino, obtener una lista de listas de vuelos correspondiente a todas las posibilidades de ir del origen al destino via un vuelo directo o un vuelo con un trasbordo (en este caso, el segundo avión debe salir al menos una hora después de que arriba el primero), ordenada por costo total (de más barato a más caro) y sin repeticiones. Los vuelos directos se codifican como listas de un elemento y los vuelos con un trasbordo se codifican como listas de dos elementos. Así, la lista $[[v_1], [v_2, v_3], [v_2, v_4]]$ representa que se puede utilizar el vuelo directo v_1 o el v_2 y luego el v_3, o el v_2 y luego el v_4.

El juego TEG consta de un mapa y fichas de colores, donde cada color representa a un participante. El mapa de juego está compuesto por paises que pueden o no limitar con otro país, y a su vez, cada país pertenece a un jugador que tiene apostado un ejército con una cantidad determinada de soldados. De esta manera, obtenemos la siguiente representación:

```
tipo Color = ℤ;
```

Color

```
tipo TEG {
  observador mapa (t : TEG) : Mapa;
  observador colores (t : TEG) : [Color];
  invariante ColoresÚnicos : sinRepetidos(colores(t));
}

tipo Pais {
```

```
  observador nombre (p : Pais) : String ;
  observador dueño (p : Pais) : Color ;
  observador ejercito (p : Pais) : ℤ ;
  invariante AlMenosUnSoldado : ejercito(p) > 0 ;
}

tipo Mapa {
  observador paises (m : Mapa) : [Pais] ;
  observador sonVecinos (m: Mapa, p1,p2: Pais) : Bool ;
       requiere p1 ∈ paises(m) ∧ p2 ∈ paises(m) ;
  invariante NingúnPaísEsVecinoDeSiMismo : (∀p ← paises(m))¬sonVecinos(m, p, p) ;
  invariante NoHayPaisesRepetidos : sinRepetidos(nombresDePaises(paises(m))) ;
}

aux sinRepetidos (l : [T]) : Bool = (∀i ← [0..|l|)) l_i ∉ l(i..|l|) ;
aux nombresDePaises (l : [Pais]) : [] = [ nombre(p) | p ← l ] ;
```

Ejercicio 11

a) Completar el tipo TEG, agregando `invariante MapaTodoOcupado` . Este invariante debe indicar que todos los paises del mapa pertenecen a un color del juego.
b) Completar el tipo Mapa, agregando `invariante TodoPaísTieneAlMenosUnPaísLimítrofe` . Este invariante debe indicar que cualquier país es vecino de al menos un país.
c) Completar el tipo Mapa, agregando `invariante sonVecinosEsSimetrico` . Este invariante debe indicar que si un país es vecino de otro, este otro también lo es del primero.

Ejercicio 12

a) Especificar el `aux esSuperPotencia` (m: Mapa,p: Pais) : Bool Que devuelve verdadero sólo si el ejército (cantidad de soldados) del país indicado es superior a la suma de los ejércitos (la suma de los soldados de cada ejercito) de sus vecinos.
b) Especificar el `problema paisesMasVulnerables` (t: TEG) = `result` : [Pais] Devuelve una lista con aquellos países que, siendo los que poseen la mayor cantidad de vecinos, están rodeados por la mayor cantidad de países enemigos con al menos tres soldados en cada uno de sus ejércitos. En otras palabras, primero se debe determinar cuales son los países con más vecinos. Luego, de estos últimos, cuáles están rodeados por la mayor cantidad de países de otro color con al menos tres soldados. Dos países son enemigos, si poseen distintos dueños.

Ejercicio 13 Especificar el `problema conquistarPaís` (t: TEG, paisAgresor: Pais, paisAgredido: Pais) Un país puede conquistar a otro, sólo si su ejército es más poderoso que el del otro **(Atención: dice más poderoso y NO más numeroso)**, y si sus respectivos dueños no son iguales. Ambos países deben pertenecer al juego y ser limítrofes entre sí.

En caso de que el país agresor tuviera un ejército más poderoso que el del país agredido, entonces dicho país pasa a pertenecer al dueño del país agresor.

Al conquistar un país, el país invasor debe pasar la mayor cantidad posible de soldados al país conquistado.

Por ejemplo, supongamos que China tiene un ejército de 6 soldados y Kamchatka de 2 soldados y además el ejército de China es más poderoso que el ejército de Kamchatka. Luego de un ataque, los países deberían quedar así: ambos con el mismo dueño (el de China), China con 1 soldado y Kamchatka con 5.

Se cuenta con el `aux esMasPoderoso` (e1: Ejercito, e2: Ejercito) : Bool que devuelve verdadero sólo si el ejército e1 es más poderoso que el ejército e2.

De Artículos, Promociones y Changos.

```
tipo Dinero = ℤ ;
```

Dinero

```
tipo Articulo {
  observador nombre (a : Articulo) : String ;
  observador costo (a : Articulo) : Dinero ;
  observador precio (a : Articulo) : Dinero ;
  invariante CostoPositivo : costo(a) > 0 ;
  invariante PrecioPositivo : precio(a) > 0 ;
  invariante LaCasaGana : precio(a) >= costo(a) ;
}

tipo Promocion {
  observador articulos (p : Promocion) : [Articulo] ;
  observador precio (p : Promocion) : Dinero ;
  invariante NoEstoySolo : |articulos(p)| > 0 ;
  invariante PrecioPositivo : precio(p) > 0 ;
}

tipo Chango {
  observador articulos (c : Chango) : [Articulo] ;
}
```

```
tipo Super {
  observador articulos (s : Super) : [Articulo];
  observador promociones (s : Super) : [Promocion];
  observador changos (s : Super) : [Chango];
  invariante NoEstoySolo : |articulos(s)| > 0;
  invariante promosOk : (∀p ← promociones(s))articulos(p) ∈
      articulos(s);
  invariante changosOk : (∀c ← changos(s))articulos(c) ∈
      articulos(s);
}
```

Ejercicio 14

a) Completar el tipo Promocion, agregando `invariante NoSePierdePlata.` Este invariante debe indicar que el precio de una promocion nunca puede ser inferior a la suma del costo de sus productos.
b) Completar el tipo Super, agregando `invariante NoHayArticulosRepetidos` . Este invariante debe indicar que no hay articulos repetidos. Dos articulos son iguales, sólo si poseen el mismo nombre.
c) Completar el tipo Super, agregando `invariante VariosEnPromos` . Este invariante debe garantizar que al menos el 50 por ciento de los productos esté en alguna promoción.

Ejercicio 15

a) Especificar el `aux promoQueNoConviene` (a: [Articulo],p: Promocion) : Bool Que devuelve verdadero sólo si el precio de la promo es superior a la suma de los precios de venta de los artículo pasados por parámetro. Es decir, tomar de la lista 'a'sólo los artículos correspondientes a la promoción y ver si la suma de sus precios de venta es menor o igual que el precio de la promoción.
b) Especificar el `aux promosAplicables` (c: Chango, p: [Promocion]) : [Promocion] Devuelve una lista con aquellas promociones que podrían aplicarse al changuito. Una promocion es aplicable a un chango, si sus articulos están en el chango. Nota: las promociones pueden superponerse.

Ejercicio 16

a) Especificar el `problema articulosSinVentas` (s: Super) = `result` : [Articulo] Un artículo no posee ventas, si no se encuentra en ningún chango.
b) Especificar el `problema promosMasUsadas` (s: Super) = `result` : [Promocion] Devuelve las promos más usadas por los changos del super (las mas aplicables). Hint: usar el **aux promosAplicables**
c) Especificar el `problema quitarPromosQueNoConvienen` (s: Super) Modifica el super, de manera tal que quede sin promociones que no convienen. Hint: usar el **aux promoQueNoConviene**