Lenguaje de especificación

Algoritmos y Estructuras de Datos

23 de agosto de 2023

¿Qué vamos a ver?

- Especificación
- Lenguaje de especificación
- Repaso de Secuencias
- Ejercicios

- ¿Por qué queremos especificar problemas formalmente?
  - Ayuda a entender mejor el problema.
  - El lenguaje natural es ambiguo.
  - Nos sirve para expresar formalmente QUÉ debe cumplir una posible solución de un problema dado.
  - No expresamos CÓMO solucionarlo (puede no haber solución o quizás no sabemos escribirla).

Un contrato


desmotivaciones.es
¿Que Es Un Contrato?
El diccionario lo define como
un acuerdo que no se puede romper...
...QUE NO SE PUEDE ROMPER...

```
proc nombre (parametros) {
  requiere {expresionBooleana1}
  asegura {expresionBooleana2}
}
```

```
proc nombre (parametros) {
  requiere {expresionBooleana1}
  asegura {expresionBooleana2}
  aux auxiliar1 (parametros) : tipoRes = expresion ;
  pred pred1 (parametros) {
          expresion
        }
}
```

Sentencias Precondición y Postcondición (requiere y asegura respectivamente)

- Ambas son condiciones booleanas.
- Tiene que haber una sola cláusula "requiere" y una sola cláusula "asegura" para cada problema.
- La cláusula "requiere" es una restricción que las variables de entrada deben respetar para garantizar una correcta solución al problema.
- La cláusula "asegura" es una condición que debe cumplir el resultado de un algoritmo para respetar la especificación.

Tipos de parámetros

- Los parámetros pueden ser de tres tipos
- **in**: parámetros de entrada, son el/los que debe recibir el programa que implemente la especificación para llegar al resultado.
- **out**: parámetros de salida, son el/los que debe retornar un programa que implemente la especificación.
- **inout**: son en simultáneo parámetros de entrada y de salida, se reciben como parámetros de entrada y se modifican para ser retornados como parámetros de salida.

- Subespecificar
  - Dar una precondición más restrictiva o bien una postcondición más débil que lo que se infiere del enunciado del problema
  - Una precondición más restrictiva deja afuera casos posibles de entrada
  - Una postcondición más débil admite soluciones no deseadas del problema
- Sobreespecificar
  - Dar una postcondición más restrictiva que lo que se necesita o bien dar una precondición más débil.
  - Una precondición más débil le exige al algoritmo considerar casos innecesarios
  - Una postcondición más restrictiva limita los posibles algoritmos que resuelven el problema porque impone más condiciones para la salida
- ¿Podrían pasar las dos cosas simultáneamente ? SI!

Queremos especificar una función tal que dados 2 parámetros a y b de tipo Int, me devuelva el cociente entre ambos.

`proc cociente` (in a : $\mathbb{Z}$, in b : $\mathbb{Z}$, out res: $\mathbb{Z}$) {
  `asegura` $\{res = a \ / \ b\}$
}

- ¿Está bien esto?
- ¿Qué pasa si b = 0?
- Aparentemente debería haber una precondición que lo restrinja
- Además, la precondición es obligatoria (si no hay precondición entonces es True).

A ver ahora...

```
proc cociente (in a : ℤ, in b : ℤ, out res: ℤ) {
  requiere {b ≠ 0}
  asegura {res = a / b}
}
```

Un repaso de la clase pasada:

- Secuencias $=$ listas
- Las siguientes expresiones, ¿son secuencias válidas?
  - $\langle 1, 2, 3+4, 5, 7-6\rangle = \langle 1, 2, 7, 5, 1\rangle$ **SI!**
  - $\langle\langle 1\rangle, 3, \langle\rangle, \langle 4, 3+4\rangle\rangle$ **NO!**
  - $\langle\langle 2, 3\rangle, \langle 5+1, 2\rangle, \langle\rangle\rangle$ **SI!**
  - $\langle 1, 2, 3,' a'\rangle$ **NO!**
  - $\langle '1',' 2',' 3',' a'\rangle$ **SI!**

- $\forall$: Para Todo
  - Notación: $(\forall x : T)P(x)$
  - Equivale a: "Para todo x de tipo T se cumple P de x"
  - Podemos pensarlo como una conjuncion (o sea un: "y") sobre todos los elementos del dominio: $P(x_1) \wedge P(x_2) \wedge \ldots$
- $\exists$: Existe
  - Notación: $(\exists x : T)P(x)$
  - Equivale a: "Hay algún elemento x del tipo T que cumple P de x"
  - Podemos pensarlo como una disyuncion (o sea un: "o") sobre todos los elementos del dominio: $P(x_1) \vee P(x_2) \vee \ldots$

Escribir un predicado que, dada una secuencia s, determine si hay algún elemento par en ella.

$\texttt{pred hayPares}$ (s: $seq\langle\mathbb{Z}\rangle$) {
$(\exists x : \mathbb{Z})(x \in s \land x \ mod \ 2 = 0)$
}

Escribir una función auxiliar que, dada una secuencia s de enteros, cuente la cantidad de números pares en s.

`aux cantidadDePares` (s: $seq\langle\mathbb{Z}\rangle$) :
$= \sum_{i=0}^{|s|-1}$ if s[i] mod 2 $=$ 0 then 1 else 0 fi ;

Dada una secuencia de números enteros s, devolver el menor elemento de la misma.
Tener en cuenta lo siguiente:

- Hay que poder darle entidad al elemento que queremos devolver, o sea: en la postcondición tenemos que poder expresar la propiedad "menor elemento" de la secuencia

`proc` `menorElemento` (in s: $seq\langle\mathbb{Z}\rangle$, out res: $\mathbb{Z}$) {
  `requiere` $\{|s| > 0\}$
  `asegura` $\{esElMenor(res, s)\}$
  `pred` `esElMenor` (n: $\mathbb{Z}$, s: $seq\langle\mathbb{Z}\rangle$) {
    $n \in s \wedge (\forall d : \mathbb{Z})(d \in s \rightarrow n \leq d)$
  }
}

Decidir si hay subespecificación/sobreespecificación.

Dado un número entero, devolver su inverso aditivo

proc inverso (in n: $\mathbb{Z}$, out res: $\mathbb{Z}$) {
  requiere $\{True\}$
  asegura $\{|n| = |res|\}$
}

Subespecificación porque la postcondición es más débil que lo que requiere el problema, ej.: admite el caso $res = n$

Decidir si hay subespecificación/sobreespecificación.

Dado un número natural, devolver su sucesor

proc sucesor (in n: $\mathbb{Z}$, out res: $\mathbb{Z}$) {
  requiere $\{True\}$
  asegura $\{n + 1 = res\}$
}

Sobreespecificación porque la precondición es más débil que lo que requiere el problema, ej.: admite los casos $n < 0$

Decidir si hay subespecificación/sobreespecificación.

Dada una secuencia de números enteros, devolver otra secuencia tal que en cada posición haya un valor mayor que en la posición correspondiente de la secuencia de entrada

proc mayores (in s: $seq\langle\mathbb{Z}\rangle$, out res: $seq\langle\mathbb{Z}\rangle$) {
  requiere $\{True\}$
  asegura
  $$\{|s| = |res| \wedge_L (\forall i : \mathbb{Z})(0 \leq i < |s| \rightarrow s[i] + 5 = res[i])\}$$
}

Sobreespecificación la postcondición es más restrictiva que lo que requiere el enunciado, excluye soluciones válidas.

Dado un número natural n (mayor que 0), obtener la lista de todos los números naturales que lo dividen.

`proc obtenerDivisores` (in n: $\mathbb{Z}$, out res: $seq\langle\mathbb{Z}\rangle$) {
  `requiere` $\{n > 0\}$
  `asegura` $\{(\forall d : \mathbb{Z})(d \in res \rightarrow (d > 0 \wedge_L n \ mod \ d = 0))\}$
}

- ¿Listo? Casi...
- Notar que si $n = 12$, $res = \langle 2, 6\rangle$ satisface la especificación
- ¿Este es un caso de subespecificación/sobreespecificación? SI! estamos subespecificando

- ¿Cómo lo arreglamos?
  - Todo lo que está, tiene que estar (no hay cosas de más)
  - Todo lo que tiene que estar, está (no hay cosas de menos)

`proc obtenerDivisores` (in n: $\mathbb{Z}$, out res: $seq\langle\mathbb{Z}\rangle$) {
  `requiere` $\{n > 0\}$
  `asegura` $\{(\forall d : \mathbb{Z})(d \in res \rightarrow (d > 0 \wedge_L n \ mod \ d = 0)) \wedge (\forall d : \mathbb{Z})((d > 0 \wedge_L n \ mod \ d = 0) \rightarrow d \in res)\}$
}

Ojo con los repetidos! Las secuencias no son conjuntos.

Ah...y no se olviden de usar predicados auxiliares!

```
proc obtenerDivisores (in n: ℤ, out res: seq⟨ℤ⟩)  {
  requiere {n > 0}
  asegura {todosSonDivisores(n, res) ∧
        noFaltanDivisores(n, res) ∧ noHayRepetidos(res)}

  pred todosSonDivisores (n: ℤ, s: seq⟨ℤ⟩) {
        (∀d : ℤ)(d ∈ s → (d > 0 ∧_L n mod d = 0))
       }
  pred noFaltanDivisores (n: ℤ, s: seq⟨ℤ⟩) {
        (∀d : ℤ)((d > 0 ∧_L n mod d = 0) → d ∈ s)
       }
  pred noHayRepetidos (s: seq⟨ℤ⟩) {
        (∀d : ℤ)((d ∈ s → #apariciones(s, d) = 1)
       }
}
```

Notar el antecedente y el consecuente de “todosSonDivisores” y de “noFaltanDivisores”. Podríamos usar “$\leftrightarrow$” ? **SI!**

`proc obtenerDivisores` (in n: $\mathbb{Z}$, out res: $seq\langle\mathbb{Z}\rangle$) {
 `requiere` $\{n > 0\}$
 `asegura` $\{todosSonYNoFaltan(n, res) \land noHayRepetidos(res)\}$

 `pred todosSonYNoFaltan` (n: $\mathbb{Z}$, s: $seq\langle\mathbb{Z}\rangle$) {
  $(\forall d : \mathbb{Z})(d \in s \leftrightarrow (d > 0 \land_L n \ mod \ d = 0))$
 }
 `pred noHayRepetidos` (s: $seq\langle\mathbb{Z}\rangle$) {
  $(\forall d : \mathbb{Z})((d \in s \to \#apariciones(s, d) = 1)$
 }
}

Ejercicio 4 - parámetros inout

Especificar el siguiente problema de modificación de secuencias:

proc intercambiarParesConImpares(inout l: $seq(char)$), que toma una secuencia de longitud par y la modifica de modo tal que todas las posiciones de la forma $2k$ quedan intercambiadas con las posiciones de la forma $2k+1$.

Por ejemplo, intercambiarPares("adinle") = "daniel".

```
proc intercambiarParesConImpares (inout l: seq⟨char⟩) {
  requiere {esPar(|l|) ∧ l = l_0}
  asegura {
  (∀i : ℤ)(0 ≤ i < |l| − 1 ∧ esPar(i) →_L l[i] = l_0[i + 1]) ∧
  (∀i : ℤ)(0 < i < |l| ∧ ¬esPar(i) →_L l[i] = l_0[i − 1])}
}
```

Ojo con la longitud de l! Como los parámetros *inout* se modifican, si no lo pedimos explícitamente, nada nos garantiza que l mantenga su longitud original, entonces la postcondición podría indefinirse.

```
proc intercambiarParesConImpares (inout l: seq⟨char⟩) {
  requiere {esPar(|l|) ∧ l = l_0}
  asegura {|l| = |l_0| ∧_L
  (∀i : ℤ)(0 ≤ i < |l| − 1 ∧ esPar(i) →_L l[i] = l_0[i + 1])∧
  (∀i : ℤ)(0 < i < |l| ∧ ¬esPar(i) →_L l[i] = l_0[i − 1])}
}
```

Ahora sí, garantizamos que l mantenga su longitud, y que sus posiciones pares e impares se intercambien.