Algoritmos y Estructuras de Datos

Guía Práctica
Especificación de problemas

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

En esta parte de la materia nos dedicamos a *especificar* problemas. Para eso planteamos *procedimientos* que reciben datos de entrada y modifican algunos de ellos o devuelven datos de salida. Describiremos con un lenguaje formal las propiedades que tienen que cumplir los datos de entrada para que el programa se comporte adecuadamente (los *requiere*) y las propiedades que cumplirán los datos de salida (los *asegura*). Este documento contiene el detalle del lenguaje que vamos a usar para esta tarea.

1. Especificación de procedimientos

La definición de un procedimiento tiene tres partes:

- la signatura, que incluye el nombre del procedimiento, la lista de parámetros y el tipo de datos del resultado (si lo hubiera)
- la precondición (los *requiere*)
- la postcondición (los *asegura*)

Veamos un ejemplo:

```
proc raizCuadrada(in x: ℝ): ℝ
    requiere {x > 0}
    asegura {res · res = x}
```

Esta especificación describe el comportamiento del procedimiento `raizCuadrada`, el cual recibe un dato de tipo real (`x`), que debe ser positivo ($x > 0$), y devuelve un valor de tipo real (res), que debe ser igual a la raíz cuadrada del valor de la entrada ($res \cdot res = x$).

1.1. Tipos de parámetros

Los parámetros de un procedimiento pueden ser de entrada (`in`), salida (`out`) o entrada/salida (`inout`).

Cuando el procedimiento devuelve un único valor, es posible, por conveniencia, escribir el resultado por fuera de la lista de parámetros, como resultado del procedimiento. En ese caso nos referiremos a dicho valor con la palabra reservada *res*. Ambas formas (parámetro de salida o resultado del procedimiento) son equivalentes.

```
proc doble(in a: ℤ): ℤ
    asegura {res = 2 · a}

proc doble(in a: ℤ, out res: ℤ)
    asegura {res = 2 · a}
```

Los parámetros de entrada (`in`) tienen un valor que puede leerse en cualquier momento y que no podrá ser modificado por el código del procedimiento, por lo que a la salida tendrá el mismo valor que a la entrada. Por lo tanto, es posible referirse al valor del mismo tanto en los requiere como en los asegura.

El parámetro de salida (`out`) es el resultado del procedimiento, el cual tendrá un valor válido a la salida respecto de la descripción del asegura. No tiene sentido referirse a su valor en los requiere.

```
proc divisionEntera(in numerador: ℤ, in denominador: ℤ, out divisor: ℤ, out resto: ℤ)
    requiere {denominador > 0}
    asegura {denominador · divisor + resto = numerador}
```

Por último, los parámetros de entrada/salida pueden ser modificados por el procedimiento, pudiendo tener un valor de salida distinto al recibido en la entrada. Para referirnos al valor inicial podemos utilizar *metavariables*, definir en el requiere el valor de una variable adicional (A_0) que preserva su valor y sobre la que podemos hacer referencia en los asegura.

`proc swap(inout a:` $\mathbb{Z}$`, inout b:` $\mathbb{Z}$`)`
`requiere` $\{a = A_0 \land b = B_0\}$
`asegura` $\{a = B_0 \land b = A_0\}$

En este ejemplo, las variables de entrada/salida a las que se les modifica su valor son a y b. Las metavariable A_0 y B_0 solo se usan para expresar los predicados lógicos.

Es posible escribir muchas expresiones *requiere* y *asegura*. Las mismas se considerarán unidas con el conector $\land_L$. En el siguiente ejemplo, ambas especificaciones son equivalentes:

`proc recortarRango(inout s: seq<`$\mathbb{Z}$`>, in desde:` $\mathbb{Z}$`, in hasta:` $\mathbb{Z}$`)`
`requiere` $\{0 \leq desde < |s|\}$
`requiere` $\{0 \leq hasta < |s|\}$
`requiere` $\{desde \leq hasta\}$
`asegura` $\{\cdots\}$

`proc recortarRango(inout s: seq<`$\mathbb{Z}$`>, in desde:` $\mathbb{Z}$`, in hasta:` $\mathbb{Z}$`)`
`requiere` $\{0 \leq desde < |s| \land_L 0 \leq hasta < |s| \land_L desde \leq hasta\}$
`asegura` $\{\cdots\}$

1.2. Predicados y funciones auxiliares

Para simplificar la escritura de predicados y facilitar su lectura y comprensión, es posible descomponerlos en funciones y predicados auxiliares. Veamos un ejemplo:

`proc distanciaEntrePuntos2D(in x1:` $\mathbb{Z}$`, in y1:` $\mathbb{Z}$`, in x2:` $\mathbb{Z}$`, in y2:` $\mathbb{Z}$`):` $\mathbb{Z}$
`requiere` $\{EsPositivo(x1) \land EsPositivo(y1) \land EsPositivo(x2) \land EsPositivo(y2)\}$
`asegura` $\{res = Dist(x1, y1, x2, y2)\}$

`pred EsPositivo(x:` $\mathbb{Z}$`)`
$\{x > 0\}$

`aux Dist(x1:` $\mathbb{Z}$`, y1:` $\mathbb{Z}$`, x2:` $\mathbb{Z}$`, y2:` $\mathbb{Z}$`):` $\mathbb{Z}$
$\{sqrt((x2 - x1)^2 + (y2 - y1)^2)\}$

Nótese que a diferencia de los procedimientos, los predicados y funciones auxiliares *no describen problemas*. Son simples herramientas sintácticas para descomponer predicados. El predicado anterior es *equivalente* a reemplazar el cuerpo en el predicado que lo referencia:

`proc distanciaEntrePuntos(in x1:` $\mathbb{Z}$`, in y1:` $\mathbb{Z}$`, in x2:` $\mathbb{Z}$`, in y2:` $\mathbb{Z}$`):` $\mathbb{Z}$
`requiere` $\{x1 \geq 0 \land y1 \geq 0 \land x2 \geq 0 \land y2 \geq 0\}$
`asegura` $\{res = sqrt((x2 - x1)^2 + (y2 - y1)^2)\}$

Es muy importante notar que *no se puede utilizar una referencia a un procedimiento desde los requiere o asegura de otro procedimiento*. Tampoco desde un predicado auxiliar. El siguiente ejemplo es incorrecto.

`proc máximo(in s: seq<`$\mathbb{Z}$`>):` $\mathbb{Z}$
`asegura` $\{\cdots\}$

`proc posiciónMaximo(in s: seq<`$\mathbb{Z}$`>):` $\mathbb{Z}$
`requiere` $\{|s| > 0\}$
`asegura` $\{s[res] = \text{máximo}(s)\}$ `// INCORRECTO`

1.3. Cuantificadores, secuencias y funciones especiales

Para escribir los predicados de las pre y postcondiciones (los requiere y asegura), usaremos lógica trivaluada de primer orden, tal cuál se vió en la teórica. Los **cuantificadores** que usaremos son los siguientes:

Operación	Sintaxis	Significado
cuantificador universal	$(\forall i : T)(P(i))$	Todo valor i de tipo T tiene que cumplir el predicado $P(i)$
cuantificador existencial	$(\exists i : T)(P(i))$	Existe al menos un valor i de tipo T que cumple el predicado $P(i)$

Algunos ejemplos:

- $(\forall n : \mathbb{Z})(n \cdot n \geq n)$
  Todo número entero cumple que su cuadrado es mayor o igual a sí mismo.
- $(\forall n : \mathbb{Z})(n \bmod 4 = 0 \to n \bmod 2 = 0)$
  Todo número entero cumple que si es divisible por 4, entonces es divisible por 2.
- $(\exists i : \mathbb{Z})(10 \bmod i = 0)$
  Existe un número entero que cumple que 10 es divisible por él.

Es posible cuantificar sobre múltiples variables y anidar cuantificadores.

- $(\forall n, m : \mathbb{Z})((n > 0 \land m > 0 \land n < m) \to (n^2 < m^2))$
  Para todos dos números positivos n y m que cumplen con que n es menor a m, entonces el cuadrado de n es menor que el cuadrado de m.
- $(\forall n : \mathbb{Z})((\exists m : \mathbb{Z})(m < n))$
  Para todo número entero n, siempre existe un número m que es menor.

Muy frecuentemente vamos a usar cuantificadores para describir el contenido de **secuencias**. Por ejemplo:

- $(\forall i : \mathbb{Z})(0 \leq i < |s| \to_L s[i] > 0)$
  Los elementos en todas las posiciiones de la secuencia s son mayores que cero.
- $(\exists i : \mathbb{Z})(0 \leq i < |s| \land i \bmod 2 = 0 \land_L s[i] \bmod 3 = 0)$
  Existe una posición par de la secuencia s que contiene un elemento divisible por 3.
- $(\forall i : \mathbb{Z})(0 \leq i < |s| \to_L ((\exists k : \mathbb{Z})(k.k = s[i])))$
  Todos los elementos de la secuencia s son cuadrados perfectos.

Una **función especial** que usaremos es la función *IfThenElse*, o `if` *cond* `then` val_1 `else` val_2. Esta función evalúa una condición y devuelve el primer valor si la condición es verdadera y el segundo si la condición es falsa. La condición puede ser cualquier predicado y los dos valores deben ser del mismo tipo. Nótese que el tipo de la expresión completa será el mismo que el de los valores.

Algunos ejemplos:

- $\text{if } x > 0 \text{ then } x \text{ else } -x$
  Devuelve el valor absoluto de x.
- $(\text{if } x1 > x2 \text{ then } x1 - x2 \text{ else } x2 - x1) + (\text{if } y1 > y2 \text{ then } y1 - y2 \text{ else } y2 - y1)$
  Devuelve la distancia de Manhatan (sobre una grilla) entre los puntos $(x1, y1)$ y $(x2, y2)$.

Nótese que esta función *no se utiliza* para describir causalidad (si pasa P entonces se cumple Q), ya que la evaluación de la función *if* devuelve un valor de algún tipo. La forma correcta de expresar causalidad es utilizando la implicación, de la forma $P \to Q$.

Por último, para operar con los elementos de secuencias, vamos a usar los siguientes **operadores especiales**:

Operación	Sintaxis	Significado
sumatoria	$(\sum_{i=j}^{n})(f(i))$	Equivalente a $f(j) + f(j+1) + \cdots + f(n)$
productoria	$(\prod_{i=j}^{n})(f(i))$	Equivalente a $f(j).f(j+1).\cdots.f(n)$

Ejemplos:

- $(\sum_{i=0}^{100})(i)$
  La suma de todos los enteros entre 0 y 100.

- $(\prod_{i=1}^{10})(2 \cdot i)$
  El producto de los primeros 10 números pares.

- $(\sum_{i=0}^{|s|-1})(s[i])$
  La suma de todos los elementos de la secuencia s.

Si se combinan estos operadores con el operador *if* se pueden agregar condiciones o incluso contar los elementos que cumplan una determinada condición:

- $(\sum_{i=0}^{|s|-1})(\text{if } s[i] \text{ mod } 2 = 0 \text{ then } 1 \text{ else } 0)$
  La cantidad de elementos pares en la secuencia s.

2. Tipos de especificación

Resumimos aquí los tipos de datos que podremos usar para especificar. Asimismo, indicamos sus operaciones y su notación en *Sintaxis*.

2.1. Tipos básicos

Las constantes devuelven un valor del tipo. Las operaciones operan con elementos de los tipos y retornan algun elemento de algun tipo. Las comparaciones generan fórmulas a partir de elementos de tipo.

`bool`: valor booleano.

Operación	Sintaxis
constantes	$True$, $False$
operaciones	$\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$
comparaciones	$=$, $\neq$

`int`: número entero.

Operación	Sintaxis
constantes	$1, 2, \cdots$
operaciones	$+, -, ., /$ (div. entera), % (módulo), $\cdots$
comparaciones	$<, >, \leq, \geq, =, \neq$

`real` o `float`: número real.

Operación	Sintaxis
constantes	$1, 2, \cdots$
operaciones	$+, -, ., /, \sqrt{x}, \sin(x), \cdots$
comparaciones	$<, >, \leq, \geq, =, \neq$

`char`: caracter.

Operación	Sintaxis
constantes	‘a’, ‘b’, ‘A’, ‘B’
operaciones	$ord(c)$, $char(c)$
comparaciones (a partir de ord)	$<, >, \leq, \geq, =, \neq$

2.2. Tipos complejos

`seq<T>`: secuencia de tipo T.

Operación	Sintaxis
crear	$\langle\rangle$, $\langle x, y, z\rangle$
tamaño	$\|s\|$, $length(s)$
pertenece	$i \in s$
ver posición	$s[i]$
cabeza	$head(s)$
cola	$tail(s)$
concatenar	$concat(s_1, s_2)$, $s_1 + s_2$
subsecuencia	$subseq(s, i, j)$, $s[i..j]$
setear posición	$setAt(s, i, val)$
suma	$\sum_{i=0}^{\|s\|} s[i]$
producto	$\prod_{i=0}^{\|s\|} s[i]$

`tupla<T1, ..., Tn>`: tupla de tipos $T_1, \ldots, T_n$

Operación	Sintaxis
crear	$\langle x, y, z\rangle$
campo	s_i

`struct<campo1: T1, ..., campon: Tn>`: tupla con nombres para los campos.

Operación	Sintaxis
crear	$\langle x : 20, y : 10\rangle$
campo	s_x, s_y

`string`: renombre de `seq<char>`.