Especificación y verificación formal de un algoritmo de ordenamiento

Axel Suárez Polo

BUAP
axel.suarez@alumno.buap.mx

Abstract. Este es el resumen en español.

1 Introducción

Crear software correcto es una de las tareas más complejas que se tiene en el desarrollo de software debido a que requiere un entendimiento completo tanto del problema que se desea resolver, como de la solución que se propone.

Primeramente, se deben establecer los criterios para determinar que el software desarrollado es correcto. La expresión de estos criterios debe realizarse de forma no ambigua y completa; es decir, no debe estar sujeta a la interpretación y debe abarcar todas las características deseadas del software en cuestión. Esto se conoce como la **especificación** del software.

Una vez que se ha hecho la especificación, el siguiente problema es asegurar que la solución desarrollada satisface los criterios que se establecieron. Nuevamente, no es una tarea fácil ya que los mecanismos utilizados para asegurarse de la correspondencia con la especificación deben ser correctos por si mismos y además debe revisarse que se hayan aplicado correctamente. Esto es la **verificación** del software.

En este documento se utiliza el lenguaje de programación **Agda**, que al contar con *tipos dependientes*, permite realizar la especificación, implementación y verificación en el mismo lenguaje. Este lenguaje pertenece a la clase de métodos formales conocida como *demostración automática de teoremas*.

2 Especificación de un algoritmo de ordenamiento

Para empezar, tenemos que determinar a qué nos referimos cuando decimos que un algoritmo es un **algoritmo de ordenamiento**.

Para los fines de este documento, nos enfocaremos en el ordenamiento de listas de números naturales, por lo que tenemos que importar las definiciones correspondientes de la librería estándar de Agda.

```
open import Data.Nat using (ℕ; suc; zero) public
open import Data.List using (List; _∷_; []) public
```

Esto nos permite construir valores de cada uno de los tipos correspondientes, utilizando sus constructores.

```
n1 : ℕ
n1 = suc (suc zero)

n1' : ℕ
n1' = 2
```

```
list1 : List ℕ
list1 = 1 ∷ 2 ∷ 3 ∷ []
```

Para determinar el orden entre los elementos, utilizamos la relación menor o igual de los naturales que nos proporciona la librería estándar:

```
open import Data.Nat using (_≤_) public
```

Esta relación, nos permite dar evidencia de que un natural es menor que otro, construyendo términos de este tipo:

```
open import Data.Nat using (z≤n; s≤s)

le1 : 0 ≤ 1
le1 = z≤n

le2 : 1 ≤ 2
le2 = s≤s z≤n
```

Ahora podemos definir una relación auxiliar $\leq^\star$, en la que siendo x un natural y l una lista de naturales; $x \leq^\star l$ significa que x es menor que todos los elementos de l; en otras palabras, x acota inferiormente a l. Esta relación se puede definir inductivamente de la siguiente forma en Agda:

```
open import Data.Unit using (⊤; tt) public
open import Data.Product using (_×_; _,_) public


_≤*_ : (x : ℕ) → (l : List ℕ) → Set
x ≤* [] = ⊤
x ≤* (y ∷ l) = (x ≤ y) × (x ≤* l)
```

Lo que la relación está codificando es lo siguiente:

- Cualquier natural acota inferiormente una lista vacía.
- Un natural x acota inferiormente una lista que inicia con y, si $x \leq y$ y x acota inferiormente al resto de la lista.

Nótese además que se está haciendo uso del tipo producto $\times$ y unit $\top$ que nos ofrece Agda en la librería estándar para expresar la noción de tautología y de conjunción, correspondientemente.

Esta relación se puede utilizar para dar evidencia de que un número acota inferiormente a una lista:

```
ac1 : 1 ≤* (2 ∷ 3 ∷ [])
ac1 = s≤s z≤n , s≤s z≤n , tt

-- El tipo de ac1 normalizado
ac1' : 1 ≤ 2 × 1 ≤ 3 × ⊤
ac1' = s≤s z≤n , s≤s z≤n , tt
```

Con la anterior relación, podemos definir un predicado para verificar que una lista se encuentra ordenada, también de forma inductiva en Agda:

```
sorted : (l : List ℕ) → Set
sorted [] = ⊤
sorted (x ∷ l) = x ≤* l × sorted l
```

Nuevamente, esta definición codifica lo siguiente:

– Un lista vacía está ordenada.
– Una lista $x :: l$ está ordenada si x es menor que todos los elementos de l y además l está ordenada.

Podría parecer que esta es la única definición que necesitamos para especificar que un algoritmo de ordenamiento es correcto, sin embargo, esto permite que funciones como la siguiente, sean consideradas algoritmos de ordenamiento, ya que al devolver la lista vacía, esta devolviendo efectivamente una lista ordenada, pero no la lista ordenada que queremos:

```
no-sort : List ℕ → List ℕ
no-sort l = []

no-sort-sorts : ∀ (l : List ℕ) → sorted (no-sort l)
no-sort-sorts l = tt
```

Por lo tanto, es necesario refinar la especificación. La otra condición que necesitamos de un algoritmo de ordenamiento es que devuelva una lista con los mismos elementos que la lista de entrada, aunque ordenados. Es decir, necesitamos que el algoritmo de ordenamiento no borre, agregue o duplique elementos arbitrarios de la lista de entrada.

En otras palabras, lo que necesitamos del algoritmo de ordenamiento es que devuelva una *permutación* de la lista original, y que además esta permutación se encuentre ordenada.

Para esto, podemos definir la relación de permutación $\sim$ entre dos listas como sigue en Agda:

```
data _~_ {A : Set} : List A → List A → Set where
  ~-nil   :                          [] ~ []
  ~-drop  : (x : A) { l l' : List A} → l ~ l' → (x :: l) ~ (x :: l')
  ~-swap  : (x y : A) (l : List A)   → (x :: y :: l) ~ (y :: x :: l)
  ~-trans : {l l' l'' : List A}      → l ~ l' → l' ~ l'' → l ~ l''
```

Esta definición codifica lo siguiente:

– Una lista vacía es permutación de si misma.
– Si una lista l' es permutación de otra lista l, agregar un elemento cualquiera x al inicio de ambas, conserva la relación de permutación.
– Si a una lista l se le agregan al inicio dos elementos x y y, esta nueva lista es permutación de la lista l con los elementos x y y agregados en orden inverso.
– La relación de permutación es transitiva, es decir, dadas tres listas l, l' y l'', si $l \sim l'$ y $l' \sim l''$, entonces $l \sim l''$.

Lo que nos permite dar evidencia de que una lista es permutación de otra:

```
perm1 : (1 :: 2 :: 3 :: []) ~ (3 :: 1 :: 2 :: [])
perm1 =
  let p1 = ~-swap 1 3 (2 :: [])
      p2 = ~-drop 1 (~-swap 2 3 [])
    in ~-trans p2 p1
```

Con estas definiciones, finalmente podemos especificar de forma no ambigua y completa lo que consideramos como algoritmo de ordenamiento:

```
Correct-Sorting-Algorithm : (f : List ℕ → List ℕ) → Set
Correct-Sorting-Algorithm f = ∀ (l : List ℕ) → sorted (f l) × l ~ f l
```

Esta predicado define un algoritmo de ordenamiento como una función que recibe una lista de naturales y devuelve una lista de naturales; tal que para todas las listas de naturales, aplicar esta función devuelve una lista ordenada y además la lista que devuelve es permutación de la lista de entrada.

3 Verificación

Para llevar a cabo la verificación, primero necesitamos definir nuestra función de ordenamiento. En este caso, verificaremos la siguiente implementación del algoritmo de ordenamiento por inserción, utilizando su definición recursiva:

```
open import Data.Sum using (inj₁; inj₂)
open import Data.Nat.Properties using (≤-total)

insert : (x : ℕ) → (l : List ℕ) → List ℕ
insert x [] = x ∷ []
insert x (y ∷ l) with ≤-total x y
... | inj₁ x≤y = x ∷ y ∷ l
... | inj₂ y≤x = y ∷ insert x l

insertion-sort : List ℕ → List ℕ
insertion-sort [] = []
insertion-sort (x ∷ l) = insert x (insertion-sort l)
```

Esta definición hace uso de la función $\leq$`-total` que decide si un natural es menor o igual que otro, o viceversa; devolviendo `inj`$_1$ en el caso de que sea menor o igual e `inj`$_2$ en caso contrario. Estos constructores pertenecen al *tipo suma* definido en la librería estándar de Agda.

Para poder verificar que la función de `insertion-sort` cumple con la especificación, requerimos probar primero propiedades e invariantes que siguen las funciones definidas con anterioridad. Por ejemplo, una invariante que es relevante es la siguiente:

```
≤*-insert : ∀ (x y : ℕ) (l : List ℕ) → x ≤ y → x ≤* l → x ≤* insert y l
≤*-insert x y [] x≤y x≤*l = x≤y , tt
≤*-insert x y (z ∷ l) x≤y (x≤z , z≤*l) with ≤-total y z
... | inj₁ y≤z = x≤y , x≤z , z≤*l
... | inj₂ z≤y = x≤z , (≤*-insert x y l x≤y z≤*l)
```

Lo que esta invariante nos dice es que si $x \leq y$ y además x acota inferiormente a l, entonces x seguirá acotando inferiormente a la lista que resulte de insertar y en l.

Para hacer la demostración, se procede por inducción sobre la lista l, que en Agda se traduce en realizar un análisis de casos sobre el parámetro que corresponde a la lista. Además, en el caso en el que l sea una lista con al menos un elemento z, se utiliza una cláusula `with`, para permitir a Agda continuar la normalización de la expresión `insert y l`.

Agda realiza normalización tan pronto como tiene un termino reducible, como puede observarse en el caso de que la lista sea de la forma $z :: l$ y al aparecer este valor en los tipos de los parámetros, puede proceder con la reducción de $x \leq^\star l$, por lo que el argumento que corresponde a la prueba de que $x \leq^\star l$, se normaliza a una pareja, tal y como lo indica su definición; lo que nos permite realizar el análisis de casos sobre ese argumento.

Una propiedad importante de la relación $\leq^\star$, es que es transitiva con respecto a la relación $\leq$. Esto lo podemos demostrar como sigue, por inducción sobre la lista acotada:

```
open import Data.Nat.Properties using (≤-trans)

≤*-trans : {x y : ℕ} {l : List ℕ} → x ≤ y → y ≤* l → x ≤* l
```

```
≤*-trans {l = []} x≤y y≤*l = tt
≤*-trans {l = z :: l} x≤y (x≤z , y≤*l) =
  ≤-trans x≤y x≤z , ≤*-trans x≤y y≤*l
```

Con esta propiedad, podemos realizar la prueba de la invariante más relevante:

```
insert-preserves-sorted : ∀ (x : ℕ) (l : List ℕ)
                        → sorted l
                        → sorted (insert x l)
insert-preserves-sorted x [] sl = tt , tt
insert-preserves-sorted x (y :: l) (y≤*l , sl) with ≤-total x y
... | inj₁ x≤y = (x≤y , ≤*-trans x≤y y≤*l) , y≤*l , sl
... | inj₂ y≤x =
       ≤*-insert y x l y≤x y≤*l , insert-preserves-sorted x l sl
```

Esta demostración muestra que teniendo una lista ordenada l, al realizar `insert x l` para cualquier natural x, se preserva la propiedad de que la lista está ordenada. Nuevamente, se procede por inducción sobre la lista de entrada y se replica parcialmente la estructura de `insert`, utilizando la cláusula `with` para permitir a Agda continuar con la normalización. Además se hace uso de los lemas que se demostraron con anterioridad.

Finalmente, podemos probar que `insertion-sort` devuelve una lista ordenada, por inducción sobre la lista y utilizando el lema anterior, ya que la definición de `insertion-sort` utiliza repetidamente la función `insert`.

```
insertion-sort-sorts : ∀ (l : List ℕ) → sorted (insertion-sort l)
insertion-sort-sorts [] = tt
insertion-sort-sorts (x :: l) =
  let h-ind = insertion-sort-sorts l
   in insert-preserves-sorted x (insertion-sort l) h-ind
```

Para probar que la lista devuelta por `insertion-sort` es una permutación de la lista original, tenemos que probar algunos lemas adicionales. Por ejemplo, que la relación de permutación es reflexiva para todas las listas, realizando la prueba por inducción:

```
~-refl : {A : Set} {l : List A} → l ~ l
~-refl {l = []}    = ~-nil
~-refl {l = x :: l} = ~-drop x ~-refl
```

Además, podemos probar que la relación de permutación es simétrica, por inducción sobre el constructor de la permutación:

```
~-sym : {A : Set} {l l' : List A} → l ~ l' → l' ~ l
~-sym ~-nil                 = ~-nil
~-sym (~-drop x l~l')       = ~-drop x (~-sym l~l')
~-sym (~-swap x y l)        = ~-swap y x l
~-sym (~-trans l~l'' l''~l) = ~-trans (~-sym l''~l) (~-sym l~l'')
```

Con esto, podemos probar que `insert x l` devuelve una permutación de la lista $x :: l$, con lo que garantizamos que insert no remueve o agrega elementos a la lista salvo x:

```
insert-~ : (x : ℕ) (l : List ℕ) → (x :: l) ~ (insert x l)
insert-~ x [] = ~-drop x ~-nil
insert-~ x (y :: l) with ≤-total x y
```

```
... | inj₁ x≤y = ~-refl
... | inj₂ y≤x = ~-trans (~-swap x y l) (~-drop y (insert-~ x l))
```

Lo que nos permite a su vez probar que insertar el mismo elemento en dos listas, preserva la propiedad de permutación, haciendo uso de este lema y la transitividad de $\sim$:

```
~-insert : (x : ℕ) {l l' : List ℕ} → l ~ l' → insert x l ~ insert x l'
~-insert x {l} {l'} l~l' =
  let p1 = ~-sym (insert-~ x l)
      p2 = insert-~ x l'
      mid = ~-drop x l~l'
    in ~-trans p1 (~-trans mid p2)
```

Finalmente, podemos probar con estos dos lemas sobre la relación $\sim$ e `insert` que `insertion-sort` devuelve efectivamente una permutación de la lista original:

```
insertion-sort-~ : (l : List ℕ) → l ~ (insertion-sort l)
insertion-sort-~ [] = ~-nil
insertion-sort-~ (x :: l) =
  let h-ind = insertion-sort-~ l
      p1 = insert-~ x l
      p2 = ~-insert x h-ind
    in ~-trans p1 p2
```

Básicamente se realiza inducción sobre la lista y se utilizan directamente ambos lemas sobre la lista original y la hipótesis de inducción.

Así tenemos los lemas necesarios para asegurar que `insertion-sort` cumple con la propiedad que establecimos en la especificación y el sistema de tipos de Agda se encargará de la verificación de que nuestro razonamiento es correcto, implicando que hemos verificado el algoritmo de ordenamiento.

```
insertion-sort-correct : Correct-Sorting-Algorithm insertion-sort
insertion-sort-correct l =
  insertion-sort-sorts l , insertion-sort-~ l
```