Formal specification and verification of a sorting algorithm

Axel Suárez Polo[1], José de Jesús Lavalle Martínez[1]
[1]Facultad de Ciencias de la Computación - BUAP

Resumen

This is the abstract in English.

1 Introduction

TODO

2 Specification

TODO

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

TODO

```
n1 : ℕ
n1 = suc (suc zero)

n1' : ℕ
n1' = 2

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

TODO

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

TODO

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

le1 : 0 ≤ 1
le1 = z≤n

le2 : 1 ≤ 2
le2 = s≤s z≤n
```

TODO

```
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)
```

TODO
TODO

```
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
```

TODO

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

TODO
TODO

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

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

TODO

```
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''
```

TODO
TODO

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

TODO

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

TODO

3 Verification

TODO

```
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)
```

TODO
TODO

```
≤*-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)
```

TODO
TODO

```
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
```

TODO

```
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
```

TODO

TODO

```
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
```

TODO

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

TODO

```
~-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'')
```

TODO

```
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))
```

TODO

```
~-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)
```

TODO

```
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
```

TODO
TODO

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

4 Conclusions

TODO

5 Acknowledgements

TODO

Referencias

1. What is Agda? (2021). https://agda.readthedocs.io/en/v2.6.2/getting-started/what-is-agda.html
2. Stump, Aaron. (2016). Verified Functional Programming in Agda. 10.1145/2841316.2841321.
3. T. Altenkirch, C. McBride, and J. McKinna. Why dependent types matter. Disponible en línea, Abril 2005.
4. Mimram, S. (2020). Program = Proof. Independently Published.