“Insertion sort” – A √eriFun example for a functional program with lemmas

```
structure bool <=  true, false
```

(This data structure is predefined *)*

```
structure ℕ <=  0, ⁺(⁻ : ℕ)
```

(Data structure ℕ is predefined as well and represents the natural numbers with constructor function $^+$(…) for the successor and selector function $^-$(…) for the predecessor. *)*

```
function [infix*,20] ≤(n : ℕ, m : ℕ) : bool <=
if ?0(n)
  then true
  else if ?0(m) then false else ⁻(n) ≤ ⁻(m) end_if
end_if
```

(Defines the usual ≤-relation on natural numbers. [infix*,20] is an optional command for the language parser that '≤' will be used in infix notation with precedence 20. '*?0(n)*' stands for '*n = 0*'. *)*

```
lemma ≤ is total <= ∀ n, m : ℕ  if{n ≤ m, true, m ≤ n}
```

(This lemma is needed to prove lemma "insert keeps ordered" below. Only universal quantification is allowed for lemmas and case analyses (like in the procedure definitions) are used for writing connectives. E.g.,* if{a, true, b} *is the system’s notation for writing a disjunction. *)*

```
structure list[@ITEM] <=   ø,  [infixr,100] ::(hd : @ITEM, tl : list[@ITEM])
```

(Identifiers preceded by '@' denote type variables, and therefore polymorphic lists are defined here. Lists are built with the constructors 'ø' (for the empty list) and '::'. The functions 'hd' and 'tl' (for head and tail) are the selectors of '::' yielding the leftmost list element and the list with the leftmost list element removed respectively. For instance, a list <a, b, c> is formally represented by a :: b :: c :: ø and hd(a :: b :: c :: ø) = a as well as tl(a :: b :: c :: ø) = b :: c :: ø holds. *)*

```
function ordered(k : list[ℕ]) : bool <=
if ?ø(k)
  then true
  else if ?ø(tl(k))
            then true
            else if hd(k) ≤ hd(tl(k))
                     then ordered(tl(k))
                     else false
                   end_if
          end_if
end_if
```

(This procedure decides whether number list 'k' is ordered wrt. '≤'. *)*

```
function [infix*,50] #(i : @ITEM, k : list[@ITEM]) : ℕ <=
if ?ø(k)
  then 0
  else if i = hd(k)
          then ⁺(i # tl(k))
          else i # tl(k)
        end_if
end_if
```

(Procedure '#' counts the number of occurrences of 'i' in list 'k'. *)*

```
function [infixr,40] ⊕(n : ℕ, k : list[ℕ]) : list[ℕ] <=
if ?ø(k)
  then n :: ø
  else if n ≤ hd(k)
          then n :: k
          else hd(k) :: (n ⊕ tl(k))
        end_if
end_if
```

(Procedure ⊕ insert 'n' into number list 'k' such that the resulting list is ordered whenever 'k' is, cf. lemma "insert keeps ordered" below. *)*

```
function isort(k : list[ℕ]) : list[ℕ] <=  if ?ø(k) then ø else hd(k) ⊕ isort(tl(k)) end_if
```

(Defines a sorting algorithm, viz. "insertion sort", by an ordered insertion of list elements. *)*

```
lemma insert keeps ordered <= ∀ n : ℕ, k : list[ℕ]  if{ordered(k), ordered(n ⊕ k), true}
```

(This lemma claims that insertion of list elements by '⊕' keeps the ordered-property of the list.* if{a, b, true} *is the system's notation for writing an implication. This lemma is needed for proving lemma "isort sorts" below. *)*

```
lemma isort sorts <= ∀ k : list[ℕ] ordered(isort(k))
```

(This lemma claims that procedure 'isort' computes an ordered list. *)*

```
lemma occurs insert <= ∀ n, m : ℕ, k : list[ℕ]  n # (m ⊕ k) = if{n = m, ⁺(n # k), n # k}
```

(This lemma states how the number of element occurrences in a number list change when a further number is inserted with '⊕'. This lemma is needed to prove lemma "isort permutes" below. *)*

```
lemma isort permutes <= ∀ k : list[ℕ], n : ℕ  n # k = n # isort(k)
```

(This lemma claims that procedure 'isort' does not change the number of list element occurrences, i.e. the list computed by 'isort(k)' contains exactly the elements of 'k'. *)*