МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное автономное образовательное
учреждение высшего образования
ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

Институт математики, механики и компьютерных наук
имени И. И. Воровича

Направление подготовки
02.03.02 — Фундаментальная информатика
и информационные технологии

СЕРТИФИКАЦИЯ АЛГОРИТМОВ СОРТИРОВКИ СРЕДСТВАМИ COQ

Выпускная квалификационная работа
на степень бакалавра

Студента 4 курса
А. О. Орищенко

Научный руководитель:
Старший преподаватель кафедры ИВЭ В. Н. Брагилевский

Допущено к защите:
руководитель направления ФИИТ _____ В. С. Пилиди

Ростов-на-Дону
2017

Contents

Abstract

Одним из способов проверки корректности ПО является его верификация - доказательство с помощью формальных методов правильности программы (системы) в соответствии с ее спецификацией, описанной формальным языком.

В качестве программного средства для построения этого языка используется такой инструмент как COQ. Фактически этот инструмент является языком программирования с зависимыми типами, с помощью которого удобно записывать математические теоремы, модифицировать их и проверять на правильность.

В данной работе мы определим некоторый набор предикатов их их свойств, истинность которых докажем с помощью COQ, и построим на основе этой системы две сертифицированных программы сортировки (для сортировки пузырьком и вставками).

Для начала определим сортирующую функцию. Пусть нам дан список l с целочисленными элементами. Тогда сортирующая функция должна сопоставлять этому списку l список l', где

1. все элементы списка расположены в возрастающем порядке (то есть каждый следующий больше или равен предыдущему)

2. представлены все элементы списка l, учитывая количество их вхождений в l. Другими словами список l' является некоторой перестановкой элементов списка l.

Эти свойства могут быть формализованы и описаны с помощью двух предикатов, описанных ниже.

Свойство ’быть отсортированным’

Чтобы определить предикат ’отсортированный список’, следует рассмотреть три случая:

1. пустой список отсортирован

2. список из одного элемента также отсортирован

3. если некоторый список $l = (n :: l')$ (где n - голова списка, а l' - хвост списка) отсортирован, то если добавить любое число p, меньшее n, в начало списка, то результат $(p :: n :: l')$ также будет являться отсортированным списком.

С помощью Coq этот предикат может быть записан следующим образом.

```
Inductive sorted: list Z→Prop:=
  sorted0: sorted(nil)
  sorted1: ∀z : Z, sorted(z :: nil)
  sorted2: ∀z1, z2 : Z, ∀l : list Z, z1 ⩽ z2 ⇒ sorted(z2 :: l)
     ⇒ sorted(z1 :: z2 :: l)
```

Это определение имеет несколько свойств, истинность которых доказана в приложении к данной работе:

$$\frac{sorted\ (a :: l)}{sorted\ l} \qquad \frac{sorted\ (a :: b :: l)}{sorted\ (a :: l)}$$

$$\frac{x \leqslant a \;\; sorted\ (a :: l)}{sorted\ (x :: l)}$$

Свойство 'быть эквивалентным'

Следующим шагом сформируем бинарное отношение на двух списках, которое определяет, что два списка имеют одни и те же элементы, то есть являются перестановками друг друга. Для этого опишем функцию, которая возвращает число вхождений некоторого числа z в список l, иначе говоря сколько раз z встречается в списке l. Назовем эту функцию nb_occ, и ее тип будет $[Z \to list\ Z \to nat]$. Следующим шагом на основе этой функции определим бинарное отношение эквивалентности двух списков:

$$l \equiv l' \Leftrightarrow \forall z : Z, nb\_occ\ z\ l = nb\_occ\ z\ l'$$

Нетрудно заметить, что это отношение является отношением эквивалентности и удовлетворяет следующим свойствам:

$$\frac{}{l \equiv l} \qquad \frac{l \equiv l'}{l' \equiv l} \qquad \frac{l \equiv l' \;\; l' \equiv l''}{l \equiv l''}$$

$$\frac{l \equiv l'}{(z :: l) \equiv (z :: l')} \qquad \frac{l \equiv l'}{(n :: m :: l) \equiv (m :: n :: l')}$$

$$\frac{l \equiv [\,]}{l = [\,]} \qquad \frac{(a :: l) \equiv [b]}{a = b \wedge (a :: l) = [b]}$$

Вспомогательные функции

Далее будем рассматривать сортировку вставками. Этот алгоритм опирается на вспомогательную функцию, которая вставляет элементы уже в отсортированный список. Назовем эту функцию $insert$, и ее тип будет $[Z \to list\ Z \to list\ Z]$. Нетрудно определить эту функцию рекурсивно следующим образом:

```
if l is empty, then n :: [],
if l has the form p :: l then
    if n ⩽ p, then n :: p :: l
    if p > n, then p :: (insert n l).
```

Функция $insert$ удовлетворяет следующим свойствам, которые можно доказать с помощью индукции по длине списка l. Эти свойства будут впоследствии использоваться при построении сертифицированной программы.

$$\frac{}{insert\ n\ l \equiv (n :: l)}\ \text{insert-equiv}$$

$$\frac{sorted\ l}{sorted\ (insert\ n\ l)}\ \text{insert-sorted}$$

Аналогично сортировке вставками можно рекурсивно определить вспомогательную функцию для сортировки пузырьком:

```
if l is empty, then n :: [],
if l has the form p :: l then
    if n ⩽ p, then n :: (bubble p l)
    if p > n, then p :: (bubble n l).
```

Для сортировки также пузырьком требуется доказать два свойства, которым удовлетворяет определение функции $bubble$.

$$\frac{}{bubble\ n\ l\ \equiv\ (n :: l)}\ \text{bubble-equiv}$$

$$\frac{sorted\ l}{sorted\ (bubble\ n\ l)}\ \text{bubble-sorted}$$

Bubble-equiv доказывается индукцией по l. Но для доказательства леммы bubble-sorted следует определить новый предикат, который будет называться $minimum$. Он принимает на вход целое число n и список l и выполняется в том случае, если n меньше или равен любому элементу из l. Иначе говоря,

$$minimum\ m\ l \Leftrightarrow \forall z : Z, (nb\_occ\ z\ l > 0) :\ m \leqslant z$$

Для предиката $minimum$ можно также доказать некоторые свойства:

$$\frac{sorted\ (m :: l)}{minimum\ m\ l} \qquad \frac{l\ \equiv\ l' \quad minimum\ m\ l}{minimum\ m\ l'}$$

$$\frac{a \leqslant x \quad minimum\ a\ l}{minimum\ a\ (x :: l)} \qquad \frac{sorted\ l \quad minimum\ m\ l}{sorted\ (m :: l)}$$

Основная сортирующая функция

Осталось построить основную сертифицированную сортирующую функцию. Её основная задача заключается в том, чтобы сопоставить любому списку l список l': $sorted\ l'\ \wedge\ (l\ \equiv\ l')$. Определим ее с помощью индукции по длине списка l (вместо функции aux может быть использована любая из функций $insert$ или $bubble$).

- если l пустой список, тогда $sorted\ [\,]\ \wedge\ ([\,]\ \equiv\ [\,])$ – верно
- иначе l имеет вид $l = n :: l_1$
  - из шага индукции следует, что для l_1 можно построить такой l_1', что выполняется $sorted\ l_1' \wedge (l_1 \equiv l_1')$.

    Теперь, пусть l' имеет вид $aux\ n\ l_1'$
  - из леммы aux-sorted следует, что $sorted\ l'$ ($l_1'\ is\ sorted$)

– из леммы aux-equiv и свойств отношения эквивалентности следует, что $l = n :: l_1 \ \equiv \ n :: l'_1 \equiv aux\ n\ l'_1 = l'$.

Таким образом, мы в интерактивном режиме смогли построить l' из l и получить соответствующий терм Z_sort, то есть сертифицированную сортирующую функцию.

Теперь, применяя алгоритм извлечения ($Extraction$ command), получаем функциональную программу для сортировки списков целых чисел. Ниже представлен вывод команды Extraction на языке Haskell для сортировки пузырьком:

```
bubble :: Z -> (List Z) -> List Z
bubble z l =
  case l of {
   Nil -> Cons z Nil;
   Cons a l' ->
    case z_le_gt_dec z a of {
     Left -> Cons z (bubble a l');
     Right -> Cons a (bubble z l')}}

z_sort :: (List Z) -> (List Z)
z_sort l =
  list_rec Nil (\a _ iHl -> bubble a iHl) l
```

Таким образом, результат работы может быть кратко описан так:

1. построено два предиката (*sorted* и *equiv*) и доказаны их свойства
2. построены вспомогательные функции (*insert* и *bubble*), которые выполняют один шаг соответственного алгоритма сортировки
3. доказано, что при использовании функций ($insert$ и $bubble$) результат остается отсортированным и является эквивалентным обычной вставке в голову списка
4. функции ($insert$ и $bubble$) и их свойства были использованы при индуктивном построении сортирующей программы.

Introduction

An important part of software development is the guarantee of its correctness. Usually, the correctness of a program implies its compliance with some desired requirements. A task of checking the correctness of a program is often undertaken informally in the process of its implementation. But the complexity of software is constantly growing, so it becomes difficult to track all features of implementation and modification of a program, that is why errors occur. Some errors can go unnoticed for a long time and can be detected after the release of the finished product.

One of the ways to proof the correctness of the software is to verify it. It means to proof by formal methods the correctness or incorrectness of the program (system) in accordance with its specification described in the formal language.

As a software tool for building this language COQ system is used. In fact, this tool is a programming language with dependent types, which is convenient to write mathematical theorems, modify them and check for correctness.

This paper is organized as follows. The next section introduces some general facts about sorting algorithms, giving some information about Insertion and Bubble sort, their advantages and disadvantages. Section 2 defines a set of predicates, related lemmas and theorems that are used to build the certified sorting programs. We conclude with a discussion about results of the work and the methods used.

1. Sorting algorithm

In computer science, a sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful for canonicalizing data and for producing human-readable output.

Since the dawn of computing, the sorting problem has attracted a great deal of research due to the complexity of solving it efficiently despite its simple, familiar statement. The efficiency of sorting algorithms is very important because they have a great practical use. For example, they can be used for processing and storing large amounts of information. Some tasks of data search and selection are made easier if data are pre-order [1].

In this section we are going to consider two of the most popular sorting algorithms and describe them. They are: Bubble sort and Insertion sort.

1.1. Bubble sort

Bubble sort, also known as sinking sort, is a simple sorting algorithm that works by comparing each item in the list with the item next to it, and swapping them if required. The algorithm repeats this process until no swaps have occurred on the last pass, that indicates that the list is sorted [2].

Bubble sort algorithm is simple and less complex, so it is effective when data to be sorted is very small. Performance of bubble sort in best case is $O(n)$ when list is already-sorted and its worst case and average case complexity both are $O(n^2)$. Therefore, bubble sort is not a practical sorting algorithm when n is too large. The pseudocode of the algorithm can be represented in the following way [3]:

```
for i in 1 -> a.length - 1 do
  for j in 1 -> a.length - i - 1 do
    if a[j] < a[j+1]
      swap(a[j], a[j+1]);
```

1.2. Insertion Sort

Insertion sort is a simple sorting algorithm, which inserts each item into its proper place in the final list. It consumes one input at a time and growing a sorted list. It is highly efficient on small data and is very simple and easy to implement. One step of the algorithm process is shown in the figures 1 и 2.

Insertion sort is highly efficient on small lists. The worst case complexity of insertion sort is $O(n^2)$. When data is already sorted it is the best case of insertion sort and it takes $O(n)$ time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort provides several advantages [1]:

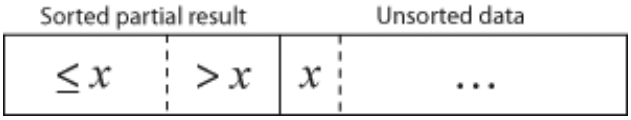

Figure 1 — Before insertion x

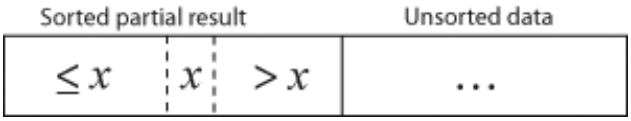

Figure 2 — After insertion x

- Simple implementation
- Efficient for (quite) small data sets
- Stable, i.e. does not change the relative order of elements with equal elements
- In-place, i.e. only requires a constant amount $O(1)$ of additional memory space
- Online, i.e. can sort a list as it receives it

2. Specification for Sorting Programs

In this section, we present an illustration the use of Coq in the development of a certified program.

We consider the type $list\ Z$ of the lists of elements of type Z (integer). We use the following notation: the empty list is written $[\ ]$, the list containing 1, 2, and -3 is written $1 :: 5 :: -3 :: [\,]$, and the result of adding n in front of the list l is written $(n :: l)$.

How can we specify a sorting program? Such program is a function that maps any list l of the type $list\ Z$ to a list l', in which

1. all the elements are placed in increasing order
2. all the elements of l are present, also respecting the number of times they occur. It means that all the elements are permutation or regrouping of the elements in l.

Such properties can be formalized and described with two different predicates. Their definitions are represented below.

2.1. Property of being sorted

First of all, let's define the predicate "to be a sorted list". In our case, we consider three clauses:

1. the empty list is sorted ($[\,]$ is sorted)
2. every list with only one element is sorted ($\forall x,\ [x]$ is sorted)
3. if a list $(n :: l)$ is sorted and if $p \leqslant n$ then the list $(p :: n :: l)$ is sorted.

This kind of definition is given as an inductive definition for a predicate sorted using three constructing rules. In COQ this predicate can be described by the following code [4]:

```
Inductive sorted: list Z→Prop:=
  sorted0: sorted(nil)
  sorted1: ∀z : Z, sorted(z :: nil)
  sorted2: ∀z1, z2 : Z, ∀l : list Z, z1 ⩽ z2 ⇒ sorted(z2 :: l)
     ⇒ sorted(z1 :: z2 :: l)
```

It is easy to prove that this definition satisfies some properties. In COQ system they can be described as lemmas or theorems. In this paper we describe they as rules which should be read: "if we have established the truth of the premises above the line, then we can come to conclusion under the line". Additionally, we assume that this property should be true for all variables. For example, the first lemma below should be read as: "$\forall a \in Z,\ \forall l \in list\ Z$, if list $(a :: l)$ is sorted, then l is sorted". So, we can proof the following properties:

$$\frac{sorted\ (a :: l)}{sorted\ l}\ \text{S1}$$

$$\frac{sorted\ (a :: b :: l)}{sorted\ (a :: l)}\ \text{S2}$$

$$\frac{x \leqslant a\ \ sorted\ (a :: l)}{sorted\ (x :: a :: l)}\ \text{S3}$$

$$\frac{x \leqslant a\ \ sorted\ (a :: l)}{sorted\ (x :: l)}\ \text{S4}$$

This properties will be used in sorting program certification.

2.2. Property of being equal

It is necessary to define a binary relation expressing that a list l is a permutation of another list l', it means that l' has the same elements.

A simple way is to define a function nb_occ of type $[Z \to list\ Z \to nat]$, which maps to any number z and list l the number of times that z occurs in l. This function can simply be defined as a recursive function.

In a second step we can define the following binary relation on lists of elements in Z [4]:

$$l \equiv l' \Leftrightarrow \forall z : Z, nb\_occ\ z\ l = nb\_occ\ z\ l'$$

It is easy to prove that the relation $\equiv$ is an equivalence relation and that it satisfies the following properties:

$$\frac{}{l \equiv l} \text{ E1}$$

$$\frac{l \equiv l'}{l' \equiv l} \text{ E2}$$

$$\frac{l \equiv l' \quad l' \equiv l''}{l \equiv l''} \text{ E3}$$

$$\frac{l \equiv l'}{(z :: l) \equiv (z :: l')} \text{ E4}$$

$$\frac{l \equiv l'}{(n :: m :: l) \equiv (m :: n :: l')} \text{ E5}$$

$$\frac{l \equiv [\,]}{l = [\,]} \text{ E6}$$

$$\frac{(a :: l) \equiv [b]}{a = b \wedge (a :: l) = [b]} \text{ E7}$$

Some of these lemmas will be used in the certification of a sorting program.

2.3. Auxiliary function (insert and bubble)

At first we consider insertion sort. This algorithm relies on an auxiliary function to insert an element in an already sorted list. At each step this function compares the item to insert and the current element from the list. If the first is less than or equal to the second, then we insert it into the head of the current list. Otherwise, the function compares this item with the next element of the sorted list. According to this algorithm, this function, named $insert$, has type [$Z\ Z \to list\ Z \to list\ Z$]. We define $insert\ n\ l$ in the following way where l varies [4]:

```
if l is empty, then n :: [],
if l has the form p :: l then
    if n ⩽ p, then n :: p :: l
    if p > n, then p :: (insert n l).
```

It is necessary to understand that the possibility to compare two numbers is a property of Z: the order $\leqslant$ is decidable. In other words, it is possible to program a function with two arguments n and p that returns a certain value when n $\leqslant$ p and a different value when n $>$ p. In the COQ system, this property is represented by a certified program given in the standard library and called $Z\_le\_gt\_dec$.

In other words, function $insert$ implement one step of the insertion sort algorithm (Figure 1 и 2). The purpose of function $insert$ is described in the following two lemmas, which are easily proved by induction on l:

$$\frac{}{insert\ n\ l\ \equiv\ (n :: l)}\ \text{insert-equiv}$$

$$\frac{sorted\ l}{sorted\ (insert\ n\ l)}\ \text{insert-sorted}$$

As an example we consider the proof of the first property. It means we will proof that $insert\ n\ l\ \equiv\ (n :: l)$. We can do this using induction on the list l:

- $l = [\,]$, then

  on the left side: $(n :: l) \Rightarrow (n :: [\,]) \Rightarrow [n]$

  on the right side: $insert\ n\ l \Rightarrow insert\ n\ [\,] \Rightarrow [n]$

  $[n] \equiv [n]$ - true

- $l = (a :: l')$. Then because of inductive step: $\forall n, (n :: l') \equiv insert\ n\ l'$.

  Let's proof that $(n :: a :: l') \equiv insert\ n\ (a :: l')$. We simplify our goal and split it into two cases:

  - $x \leqslant a$, then after applying $insert$ in our goal we get: $(n :: a :: l') \equiv (n :: a :: l')$ - true.
  - $a < x$, then after applying $insert$ in our goal we get: $(n :: a :: l') \equiv (a :: (insert\ n\ l'))$.
    - if $(n :: l') \equiv insert\ n\ l'$ then $(a :: n :: l') \equiv (a :: insert\ n\ l')$ (E4)
    - $(n :: a :: l') \equiv (a :: n :: l')$ (E5)
    - $(n :: a :: l') \equiv (a :: n :: l') \equiv (a :: (insert\ n\ l'))$ (E3)

  Q.E.D.

For now we consider the bubble sort. This algorithm relies on an auxiliary function, named bubble, that compares each element of list and the item to insert and swaps them if they are in the wrong order. This function is very similar to $insert$ and has type [$Z\ Z \to list\ Z \to list\ Z$]. We can define $bubble\ n\ l$ in following way:

```
if l is empty, then n :: [],
if l has the form p :: l then
    if n ⩽ p, then n :: (bubble p l)
    if p > n, then p :: (bubble n l).
```

To build a certified sorting program it is necessary to proof the following lemmas:

$$\frac{}{bubble\ n\ l\ \equiv\ (n :: l)}\ \text{bubble-equiv}$$

$$\frac{sorted\ l}{sorted\ (bubble\ n\ l)}\ \text{bubble-sorted}$$

The lemma bubble-equiv can be easily proved by induction on l(similar to insert-equiv). The proof of the lemma bubble-sorted is more complex. We consider it in detail.

To proof this statement we need to define a new predicate, which is named $minimum$. The predicate takes a number n and a list l and is true, when n is less then or equal to each elements of the list. Using the previously described function nb_occ we define the predicate $minimum$ as follows:

$$minimum\ m\ l \Leftrightarrow \forall z : Z, (nb\_occ\ z\ l > 0) :\ m \leqslant z$$

According to this definition we can proof some lemmas, which we will use below:

$$\frac{sorted\ (m :: l)}{minimum\ m\ l}\ \text{M1}$$

$$\frac{l\ \equiv\ l' \quad minimum\ m\ l}{minimum\ m\ l'}\ \text{M2}$$

$$\frac{a \leqslant x \quad minimum\ a\ l}{minimum\ a\ (x :: l)}\ \text{M3}$$

$$\frac{sorted\ l \quad minimum\ m\ l}{sorted\ (m :: l)}\ \text{M4}$$

The last lemma M4 can be simply described in this way: From the condition of the theorem: $sorted\ l$ and $\forall i\ in\ l,\ m \leqslant i$. We use induction on l:

- If l is empty, $[m]$ is sorted
- Otherwise l has the form $l = a :: l'$. Then

  $H1 :\ sorted\ (a :: l')$ (hypothesis)

  $H2 :\ minimumm\ (a :: l')$ (hypothesis)

  $H3\ :\ \forall n,\ sorted\ l\ \rightarrow\ minimum\ m\ l\ \rightarrow\ sorted\ m(a\ ::\ l)$ (the inductive step) We consider two cases:

  - if $m \leqslant a$, then we use S3 and get $(m :: a :: l)$ is sorted
  - $a\ <\ m$, it is not possible, because thanks $H2$ we can get that $m \leqslant a$.

  Q.E.D.

So, the lemma bubble-sorted can be proved using induction on the list l:

- If l is empty, then $bubble\ n\ l\ \Rightarrow\ bubble\ n\ [\ ]\ \Rightarrow\ [n]$. And $[n]$ is sorted according to the definition of $sorted$ predicate.
- Otherwise l has the form $l = a :: l'$. Then

  $H1 :\ sorted\ (a :: l')$ (hypothesis)

  $H2 : \forall n,\ sorted\ l \rightarrow sorted(bubble\ n\ l)$ (the inductive step)

  We should proof that $bubble\ n\ (a :: l')$ is sorted. For this we split it into two cases:

  - $n \leqslant a$, then after applying function $bubble$ the statement of the theorem is $sorted\ (n :: bubble\ a\ l')$. Then,
    - n is less then or equal to each element in the list $(a :: l')$ ($n \leqslant a$ and $minimum\ a\ l$)
    - the list $(a :: l')$ is equivalent to the list $bubble\ a\ l'$

* n is minimum in $bubble\ a\ l'$ and we can proof that $sorted\ (n :: bubble\ a\ l')$.

– $a < n$, then after applying function $bubble$ the statement of the theorem is $sorted\ (a :: bubble\ n\ l')$.

* a is less then or equal to each element in the list $(n :: l')$ ($a < n$ and $minimum\ a\ l$)

* the list $(n :: l')$ is equivalent to the list $bubble\ n\ l'$

* a is minimum in $bubble\ n\ l'$ and we can proof that $sorted\ (a :: bubble\ n\ l')$.

Q.E.D.

2.4. The main sorting function and extraction

It remains to build a certified sorting program. The goal is to map any list l to a list l' that satisfies $sorted\ l' \wedge (l \equiv l')$.

This program is defined using induction on the list l (we assume that instead of aux there can be any of the $insert$ or $bubble$ functions):

- If l is empty, then $l = [\,]$ is the right value.
- Otherwise l has the form $l = n :: l_1$
  - The induction hypothesis on l_1 expresses that we can take a list l'_1 satisfying $sorted\ l'_1 \wedge (l_1 \equiv l'_1)$.

    Now let l' be the list $aux\ n\ l'_1$
  - thanks to the lemma aux-sorted we know $sorted\ l'$ ($l'_1\ is\ sorted$)
  - thanks to the lemma aux-equiv and E4 we know $l = n :: l_1 \equiv n :: l'_1 \equiv aux\ n\ l'_1 = l'$.

This construction of l' from l, with its logical justifications, is developed in a dialogue with the Coq system. The outcome is a term of type Z_sort, in other words, a certified sorting program.

An extraction algorithm makes it possible to obtain an OCAML program that can be compiled and executed from a certified program. Such a program, obtained mechanically from the proof that a specification can be fulfilled, provides an optimal level of safety. The extraction algorithm works by removing all logical arguments to keep only the description of the computation to perform [4].

Using the extraction algorithm on this program, we obtain a functional program to sort lists of integers. Here is the output of the Extraction command to Haskell for bubble sort(2.1):

Figure 2.1. Bubble sort in Haskell

```
bubble :: Z -> (List Z) -> List Z
bubble z l =
  case l of {
   Nil -> Cons z Nil;
   Cons a l' ->
    case z_le_gt_dec z a of {
     Left -> Cons z (bubble a l');
     Right -> Cons a (bubble z l')}}

z_sort :: (List Z) -> (List Z)
z_sort l =
  list_rec Nil (\a _ iHl -> bubble a iHl) l
```

Conclusion

Finally, the result of the work can be briefly described as follows:

1. two predicates (*sorted* and *equiv*) were built and their properties were proved
2. auxiliary functions (*insert* and *bubble*), that perform one step of the corresponding sorting algorithm, were also built
3. it is proved that after using function ($insert$ and $bubble$) the resulted list remains sorted and is equivalent a normal insert in the head of the list
4. functions ($insert$ and $bubble$) and their properties were used for inductive building the certified sorting program.

The resulting system is quite adaptive and can be easily used for building certified sorting programs based on other sorting algorithms.

References

1. *Canaan C.*, *Garai M. S.*, *M D.* Popular sorting algorithms // WAP journal. — 2011. — URL: http://waprogramming.com/papers/50ae468b07bca6.46839978.pdf.
2. *Verma A. K.*, *Kumar P.* Sorting Algorithms: Review Paper. — 2015. — URL: https://arxiv.org/pdf/1310.7890.pdf.
3. Programming Algorithms. Bubble sort. — URL: http://www.programming-algorithms.net/article/39344/Bubble-sort ; (visited on 30 May 2017).
4. *Bertot Y.*, *Pierre C.* Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. — Springer, 2004.

Source Code

The source code can be found on github:
https://github.com/AlexandraOlegovna/certified_sorting