Implementation and verification of a data structure using dependent types

Anton Danilkin

April 19, 2022

Abstract

1 Introduction

One of the most important data structures in functional programming are singly linked lists, which support prepending an element in the beginning, as well as destructing a non-empty list into its head (first element) and tail (the list containing the rest of elements). Conversely, arrays are frequently used in imperative programming; they support getting and setting an element by its index, as well as, in case of vectors, dynamic resizing.

The problem of lists is that the operation of retrieving or updating an element at a specific index has linear complexity. On the flip side, operations on arrays and vectors mutate the instance they work on, which not only makes reasoning about program behavior harder, but also means that older versions of the data structure become unaccessible for future use. One way to fix that would be making a copy before each operation, but the again would mean that now everything has linear complexity.

This way the following question arises: is it possible to have a data structure that combines the benefits of singly linked lists and vectors?

2 Similarities between numbers and lists

Here are some standard definitions of Peano natural numbers and lists, as well as operations on them in Coq:

```
Inductive nat :=
  | O : nat
  | S : nat → nat.
```

```
Inductive list A :=
  | nil : list A
  | cons : A → list A → list A.

Arguments nil {A}.
Arguments cons {A} x l.
```

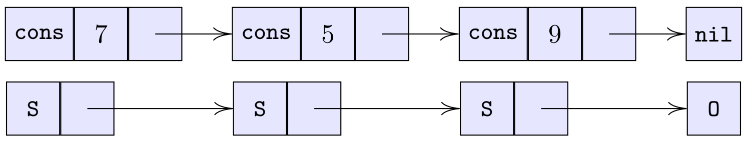
```
Definition pred (n : nat) : nat :=
  match n with
  | O ⇒ O
  | S n′ ⇒ n′
  end.
```

```
Definition tl {A} (l : list A) : list A :=
  match l with
  | nil ⇒ nil
  | cons _ l′ ⇒ l′
  end.
```

```
Fixpoint plus (n1 n2 : nat) : nat :=
  match n1 with
  | O ⇒ n2
  | S n1′ ⇒ S (plus n1′ n2)
  end.
```

```
Fixpoint app {A} (l1 l2 : list A) : list A :=
  match l1 with
  | nil ⇒ l2
  | cons x l1′ ⇒ cons x (app l1′ l2)
  end.
```

As remarked by Chris Okasaki [Oka98], there is a clear resemblance between them: the only real difference is that `list A` holds a datum of type `A`, where as `nat` does not. Here is how the list [7; 5; 9] and the number 3 (which is the length of the list) could be represented:



In fact, we can go from a list to the corresponding natural number by getting its length, and back from a natural number to one of the corresponding lists by repeating an element that many times:

```
Fixpoint length {A} (l : list A) : nat :=
  match l with
  | nil ⇒ O
  | cons _ l′ ⇒ S (length l′)
  end.
```

```
Fixpoint repeat {A} (x : A) (n : nat) : list A :=
  match n with
  | O ⇒ nil
  | S n′ ⇒ cons x (repeat x n′)
  end.
```

References

[Oka98] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998. DOI: 10.1017/CBO9780511530104.