Lab 13

Functional Programming (ITI0212)

2022-04-22

Indexing lists

During the lecture, we started encoding properties of lists as indexed types. We'll continue by defining a type that allows us to access the elements of a list by their index.

In imperative programming languages, a list of $n+1$ values is presented as an expression

$$\texttt{xs} = [\texttt{x}_0, \ldots, \texttt{x}_n]$$

and the value $\texttt{x}_k$ is called the *element at index* k. Importantly, only numbers $k \leqslant n$ are valid indices into `xs`, and we call them *in bounds*. The indices $n+1, n+2, \ldots$ are "out of bounds", since they do not refer to elements of the list. Lists are called "0-indexed" as the first element of a non-empty list has index 0.

In Idris, lists are defined inductively, so we do not immediately have access to the element at the kth index of a list. Instead, the standard library defines a function that traverses a list until it finds the correct element to return:

```
getAt : (k : Nat) -> List a -> Maybe a
getAt 0 (x :: xs) = Just x
getAt (S k) (x :: xs) = getAt k xs
getAt _ [] = Nothing
```

Note that this function returns `Just x` if `k` is in bounds, but `Nothing` if it is out of bounds.

In this exercise, well will define a function

```
at : (k : Nat) -> (xs : List a) -> IsInBounds k xs -> a
```

that returns the `k`th element of a list `xs`, given a *proof* that `k` is a valid index into `xs`. In comparison to `getAt`, this does return an element directly, no `Maybe a` necessary!

Note
Before writing any other code, make sure that all of your definitions are only accepted by the type checker if they are total. Put this *directive* at the top of your file:

```
%default total
```

In particular, this ensures that your proofs cover all the necessary cases!

Also, complete task 1 and 2 first; task 3 and 4 are independent of each other.

Task 1
Now, we encode the proposition "`k` is an in-bounds index into `xs`". For this, complete the following type definition:

```
data IsInBounds : (k : Nat) -> (xs : List a) -> Type where
  -- ...
```

Since this type depends on inductive types `Nat` and `List a`, try to come up with a similar inductive definition:

- Cover one or more obvious base cases. What are the indices of the empty list? What's the shape of lists that have index 0?
- Add case(s) to construct proofs for bigger indices or lists. Assuming you have an in-bounds index, can you prove its an index to another (bigger) list? Can you change the index so that it remains in-bounds?

Hint: Read this if you are stuck. Your type should have two constructors: One that says "0 is in-bounds for any *non-empty* list", and one that says "if `k` is in-bounds for `xs`, then `S k` is in-bounds for `x :: xs`".

Task 2
As a warm-up, define a *total* function that returns the *head* of a list, i.e. the element at index 0:

```
head : (xs : List a) -> (in_bounds : IsInBounds 0 xs) -> a
```

Hint: Case-splitting on `in_bounds` will make this a one-liner.

The function should do the obvious thing:

```
Lab13> head [1, 2, 3] ?proof1
1

Lab13> head ['a', 'z'] ?proof2
'a'
```

Of course you'll have to come up with the proofs `?proof1` and `?proof2` yourself, as they'll depend on your definition of `IsInBounds`.

Task 3
Write the function that returns elements of a list by index, given a proof that the index is in-bounds:

```
at : (k : Nat) -> (xs : List a) -> IsInBounds k xs -> a
```

Of course, `at 0` should behave like `head`, and for other indices it should do the expected thing:

```
Lab13> at 2 [1, 2, 3] ?proof1
3

Lab13> at 1 ['a', 'z'] ?proof2
'z'
```

Again, you'll have to provide proofs to call the function.

Task 4
Prove the following lemma concerning the predecessor of an index: *If `k + 1` is an index into `xs`, then `k` is also an index into `xs`.*

Hint: The type signature for this proposition is

```
isInBoundsPred : (k : Nat) -> (xs : List a)
              -> IsInBounds (S k) xs -> IsInBounds k xs
```

If you are stuck trying to prove this, case-split on the proof argument. That should result in two cases. The first is very easy. To prove the second case, recursively apply `isInBoundsPred` to a sub-term to obtain the induction hypothesis. Use that to prove the lemma in one step.

Elements of a list

In this exercise, we will focus on proving that certain terms are elements of a list. This task will be less guided than the previous task; use it to check whether you can translate natural language definitions into precise type definitions yourself. All tasks after 5 can be solved independently of each other.

Task 5
Given a value `x : a` and a list `xs : List`, define a type `IsElem x xs` that has a term if and only if `x` is an element of `xs`.

Check your definition by proving

```
example_elem : 5 `IsElem` [3, 5]
```

Hint: The type should again be an indexed type, this time with signature

```
data IsElem : (x : a) -> (xs : List a) -> Type where
  _ ...
```

Task 6
Prove that the singleton list `[x]` contains the element `x`.

Hint: Here's a type signature for this proposition:

```
isElemSingleton : (x : a) -> IsElem x [x]
```

Task 7
Given a proof `IsElem x xs`, write a function `indexOf` that computes the index at which `x` occurs in `xs`.

Optionally, combine this with `IsInBounds` and prove that the index of an element in a list is in-bounds for that list:

```
isInBoundsIndexOf : (is_elem : IsElem x xs)
  -> IsInBounds (indexOf is_elem) xs
```

Task 8
Prove that given any function `f : a -> b`, if `x` is an element of `xs`, then `f x` is an element of `map f xs`.

Hint: This is a two-line proof by induction on proofs of `IsElem x xs`.

Task 9
If you know that a list contains, you can remove precisely that element from the list. Write a function

```
dropElem : (xs : List a) -> IsElem x xs -> List a
```

that returns a the list with `x` removed from xs.