Lists

James Wilcox and Kevin Zatloukal

August 2024

> "Lists are the original data structure of functional programming, just as arrays are the original data structure of imperative programming." — Ravi Sethi

List Type

After the natural numbers, lists are the most important inductively defined type. They are defined as follows:

$$\textbf{type } \mathsf{List}\langle A\rangle := \mathsf{nil} \mid \mathsf{cons}(\mathsf{hd}\colon A,\ \mathsf{tl}\colon \mathsf{List}\langle A\rangle)$$

Note that List is a generic type. The parameter "A" indicates what sort of data is stored in the list.

Since lists are so commonly used, we often provide operators as shorthand for functions and constructors. In particular, we will use the binary operator "::" as shorthand for cons. For example,

$$1 :: 2 :: 3 :: \mathsf{nil} = \mathsf{cons}(1, \mathsf{cons}(2, \mathsf{cons}(3, \mathsf{nil})))$$

Both notations mean the same thing, but the former is shorter and easier to read.

We will also use the standard list notation $[1, 2, 3]$ at time as shorthand for $1 :: 2 :: 3 :: \mathsf{nil}$.

List Functions

As an inductive data type, lists come with a built in "=" operator. However, all other functions must be defined explicitly. Below, we will define some of the most important ones. For many, we will also define operators that act as shorthands for the same function.

The function $\mathsf{len} : \mathsf{List}\langle A\rangle \to \mathbb{N}$ returns the length of a list. It is defined by

$$\begin{aligned} \mathsf{len}(\mathsf{nil}) &:= 0 \\ \mathsf{len}(x :: L) &:= \mathsf{len}(L) + 1 \end{aligned}$$

The function $\mathsf{concat} : (\mathsf{List}\langle A\rangle, \mathsf{List}\langle A\rangle) \to \mathsf{List}\langle A\rangle$ takes two lists and returns a single list containing the elements of the first followed by those of the second. It is defined by

$$\begin{aligned} \mathsf{concat}(\mathsf{nil}, R) &:= R \\ \mathsf{concat}(x :: L, R) &:= x :: \mathsf{concat}(L, R) \end{aligned}$$

We will use the binary operator "+" as shorthand for concat.

Concatenation is arguably the most important operation on lists, which is why we have defined a special operator for it. It is important to note that this operation has the following mathematical properties:

Identity $L + \mathsf{nil} = L = \mathsf{nil} + L$ for any list L.

Associativity $(L + R) + S = L + (R + S)$ for any lists L, R, S.

Going forward, we will use these facts without any explanation. In particular, due to associativity, we can leave out (..) when concatenating multiple lists.

The function $\mathsf{rev} : \mathsf{List}\langle A\rangle \to \mathsf{List}\langle A\rangle$ returns the same numbers but in reverse order. It is defined by

$$\begin{aligned}\mathsf{rev}(\mathsf{nil}) &:= \mathsf{nil}\\ \mathsf{rev}(x :: L) &:= \mathsf{rev}(L) +\!\!+ (x :: \mathsf{nil})\end{aligned}$$

This function runs in $\Theta(n^2)$ time, but a tail-recursive equivalent exists that runs in linear time.

In a non-nil list, we have easy access to the first element (head) and the rest (tail). We can also define functions to find the last element and the initial part before the last. The functions $\mathsf{last} : \mathsf{List}\langle A\rangle \to A$ and $\mathsf{init} : \mathsf{List}\langle A\rangle \to \mathsf{List}\langle A\rangle$ do so and are defined by

$$\begin{aligned}\mathsf{last}(x :: \mathsf{nil}) &:= x\\ \mathsf{last}(x :: y :: L) &:= \mathsf{last}(y :: L)\end{aligned}$$

$$\begin{aligned}\mathsf{init}(x :: \mathsf{nil}) &:= \mathsf{nil}\\ \mathsf{int}(x :: y :: L) &:= x :: \mathsf{init}(y :: L)\end{aligned}$$

Note that both functions are undefined on nil.

We often want to know whether a list contains a particular element. The function $\mathsf{contains} : (\mathsf{List}\langle A\rangle, A) \to \mathbb{B}$ does this. It is defined by

$$\begin{aligned}\mathsf{contains}(\mathsf{nil}, y) &:= \mathsf{false}\\ \mathsf{contains}(x :: L,\, y) &:= \mathsf{true} && \text{if } x = y\\ \mathsf{contains}(x :: L,\, y) &:= \mathsf{contains}(L,\, y) && \text{if } x \neq y\end{aligned}$$

A closely related function $\mathsf{remove} : (\mathsf{List}\langle A\rangle, A) \to \mathsf{List}\langle A\rangle$ returns a list with all instances of a given value removed. It is defined by

$$\begin{aligned}\mathsf{remove}(\mathsf{nil}, y) &:= \mathsf{nil}\\ \mathsf{remove}(x :: L,\, y) &:= \mathsf{remove}(L,\, y) && \text{if } x = y\\ \mathsf{remove}(x :: L,\, y) &:= \mathsf{remove}(L,\, y) && \text{if } x \neq y\end{aligned}$$

Association Lists

A list of pairs $\mathsf{List}\langle K, V\rangle$ is called an *association list*. It is one way to mathematically define what is sometimes called a "map". The first part of each pair is a "key" and the second part is its associated "value".

The function $\mathsf{contains\text{-}key} : (\mathsf{List}\langle K, V\rangle, K) \to \mathbb{B}$ determines whether the given association list has a key with the given value. It is defined by

$$\begin{aligned}\mathsf{contains\text{-}key}(\mathsf{nil},\, y) &:= \mathsf{false}\\ \mathsf{contains\text{-}key}((x, v) :: L,\, y) &:= \mathsf{true} && \text{if } x = y\\ \mathsf{contains\text{-}key}((x, v) :: L,\, y) &:= \mathsf{contains\text{-}key}(L,\, y) && \text{if } x \neq y\end{aligned}$$

We will use the binary operator "in" as shorthand for $\mathsf{contains\text{-}key}$.

The function $\mathsf{get\text{-}value} : (\mathsf{List}\langle K, V\rangle, K) \to V$ returns the value associated with a given key, which must be in the association list. (Its value is undefined if not.) The function is defined by

$$\begin{aligned}\mathsf{get\text{-}value}((x, v) :: L,\, y) &:= v && \text{if } x = y\\ \mathsf{get\text{-}value}((x, v) :: L,\, y) &:= \mathsf{get\text{-}value}(L,\, y) && \text{if } x \neq y\end{aligned}$$

We will use the binary operator "[..]" as shorthand for $\mathsf{get\text{-}value}$ (e.g., $L[y] = \mathsf{get\text{-}value}(L, y)$).