Lists and List Induction

James Caldwell

September 20, 2011

1 Lists

John McCarthy

John McCarthy (1927-),an American Computer Scientist and one of the fathers of Artificial Intelligence. John McCarty invented the LISP programming language after reading Alonzo Church's monograph on the lambda calculus. LISP stands for *List Processing* and lists are well supported as a the fundamental datatype in the language.

Lists may well be the most ubiquitous datatype in computer science. Functional programming languages like LISP, Scheme, ML and Haskell support lists in significant ways that make them a go-to data-structure. The can be used to model many collection classes (multisets or bags come to mind) as well as relations (as list of pairs) and finite functions.

We define lists here that may contain elements from some type T. These are the so-called *monomorphic* lists; they can only contain elements of type T. There are two constructors to create a list. Nil (written as "[]") is a constant symbol denoting the empty list and ":" is a symbol denoting the constructor that adds an element of the type T to a previously constructed list. This constructor is, for historical reasons, called "cons". Note that although "[]" and ":" are both written by sequences of two symbols, we consider them to be atomic symbols for the purposes of the syntax.

This is the first inductive definition where a parameter (in this case T) has been used.

Definition 1.1 (T List)

$$[T] ::= [\,] \mid a : L$$

where

T - is a type,
$[\,]$ - is a constant symbol denoting the *empty list*, which is called "nil",
$(:)$ - is the constructor cons and is of type $T \to [T] \to [T]$,
a - is an element of the type T, and
L - is an element of type $[T]$.

A list of the form $a{:}L$ is called a *cons*. The element a from T in $a{:}L$ is called the *head* and the list L in the cons $a{:}L$ is called the *tail*.

We assume that cons associates to the right, thus if x, y are of type a, then $x{:}y{:}[\,]$ denotes the list $x{:}(y{:}[\,])$ and is not $(x{:}y){:}[\,]$. The latter expression is not a list at all because $x{:}y$ is not well-types if both x and y are of type a.

Example 1.1. As an example, let $A = \{a, b\}$, then the set of terms in the class $[A]$ is the following:

$$\{[\,], a{:}[\,], b{:}[\,], a{:}a{:}[\,], a{:}b{:}[\,], b{:}a{:}[\,], b{:}b{:}[\,], a{:}a{:}a{:}[\,], a{:}a{:}b{:}[\,], \cdots\}$$

We call terms in the class $[A]$ *lists*. If there is at least one element in the type A then the collection of all lists in type $[A]$ is infinite, but also, like the representation of natural numbers, the representation of each individual list is finite[1] Finiteness follows from the fact that lists are constructed by consing some value from the set A onto a previously constructed $[A]$. To make reading lists easier we adopt the convention to separate the consed elements with commas and enclose them in square brackets "[" and "]", thus, we write $a{:}[\,]$ as $[a]$ and write $a{:}b{:}[\,]$ as $[a, b]$. Using this notation we can rewrite the set of lists in the class $[A]$ more succinctly as follows:

$$\{[\,], [a], [b], [a, a], [a, b], [b, a], [b, b], [a, a, a], [a, a, b], \cdots\}$$

Note that the type T need not be finite, for example, the class of $[\mathbb{N}]$ is perfectly sensible, in this case, there are an infinite number of lists containing only one element *e.g.*

$$\{[0], [1], [2], [3] \cdots\}$$

1.0.1 Abstract Syntax Trees for Lists

Note that the pretty linear notation for trees in only intended to make them more readable, the syntactic structure underlying the list $[a, b, a]$ is displayed by the abstract syntax tree: shown in Fig 1.

2 Definition by recursion

In the same way that we define functions by recursion on the structure of an argument of type $\mathbb{N}$, we can define functions of type $[A] \to B$ by recursion on the structure of list arguments.

For example, we can define the append function that glues two lists together (given inputs L and M where $L, M \in [T]$, $L +\!\!+ M$ is a list in $[T]$). The append function is defined by recursion on the structure of the first argument as follows:

[1]The infinite analog of lists are called streams. They have an interesting theory of their own where induction is replaced by a principle of co-induction. The Haskell programming language supports an elegant style of programming with streams by implementing an evaluation mechanism that is *lazy*; computations are only performed when the result is needed.

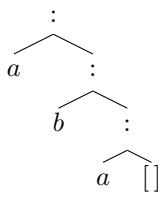

Figure 1: Syntax tree for the list $[a, b, a]$ constructed as $a{:}(b{:}(a{:}[\,]))$

Definition 2.1 (List append)

$$\begin{aligned} \mathit{append}([\,], M) &\stackrel{\text{def}}{=} M \\ \mathit{append}((a{:}L), M) &\stackrel{\text{def}}{=} a{:}(\mathit{append}(L, M)) \end{aligned}$$

The first equation of the definition says: if the first argument is the empty list $[\,]$, the result is just the second argument. The second equation of the definition says, if the first argument is a cons of the form $a{:}L$, then cons a on the *append* of L and M. Thus, there are two equations, one for each rule that could have been used to construct the first argument of the function. Note that since there are only two ways to construct a list, this definition covers all possible ways the first argument could be constructed.

Example 2.1. We give some example computations with the definition of *append*.

$$\begin{aligned} &\mathit{append}((a{:}b{:}[\,]), (c{:}[\,])) \\ &\quad = a{:}(\mathit{append}((b{:}[\,]), (c{:}[\,])) \\ &\quad = a{:}b{:}(\mathit{append}([\,], (c{:}[\,])) \\ &\quad = a{:}b{:}c{:}[\,] \end{aligned}$$

Using the more compact notation for lists, we have shown $\mathit{append}([a, b], [c]) = [a, b, c]$. Using the pretty notation for lists we can rewrite the derivation as follows:

$$\begin{aligned} &\mathit{append}([a, b], [c]) \\ &\quad = a{:}(\mathit{append}([b], [c])) \\ &\quad = a{:}b{:}(\mathit{append}([\,], [c])) \\ &\quad = a{:}b{:}[c] \\ &\quad = [a, b, c] \end{aligned}$$

We will use the more succinct notation for lists from now on, but do not forget that this is just a more readable display for the more cumbersome but precise notation which explicitly uses the cons constructor.

Append is such a common operation on lists that the ML and Haskell programming languages provide convenient infix notations for the list append operator. In Haskell, the symbol is "++", in the ML family of languages it is

"@". We will use "$+\!\!+$" here. Using the infix notation the definition appears as follows:

Definition 2.2 (List append (infix))

$$\begin{array}{rcl} [\,] +\!\!+ M & \stackrel{\text{def}}{=} & M \\ (a{:}L) +\!\!+ M & \stackrel{\text{def}}{=} & a{:}(L +\!\!+ M) \end{array}$$

Example 2.2. Here is an example of a computation using the infix operator for append and the more compact notation for lists.

$$\begin{array}{l} [a,b] +\!\!+ [c] \\ \quad = a{:}([b] +\!\!+ [c]) \\ \quad = a{:}b{:}([\,] +\!\!+ [c]) \\ \quad = a{:}b{:}[c] \\ \quad = [a,b,c] \end{array}$$

We would like to know that append really is a function of type $([A] \times [A]) \to [A]$. In Chapter **??** we presented a theorem justifying recursive definitions of a particular form (Thm. 2.1). That theorem and Corollary 2.1) guaranteed that definitions that followed a syntactic pattern were guaranteed to be functions. Similar results hold for lists and indeed there are similar theorems justifying definition by recursion for any inductively defined type.

Theorem 2.1 (Definition by recursion for list functions) Given sets A and B and an element $b \in B$ and a function $g \in (A \times B) \to B$, definitions having the following form:

$$\begin{array}{l} f([\,]) = b \\ f(x : xs) = g(x, f(xs)) \end{array}$$

result in well-defined functions, $f \in [A] \to B$.

The corollary for functions of two arguments is given as:

Corollary 2.1 (Definition by recursion (for binary functions)) Given sets A, B and C and a function $g \in (A \times C) \to C$, and a function $h \in A \to C$, and an element $b \in B$, definitions of the following form:

$$\begin{array}{l} f([\,], b) = h(b) \\ f((x{:}xs), b) = g(x, f(xs, b)) \end{array}$$

result in well-defined functions, $f \in ([A] \times B) \to C$.

Theorem 2.2 ($append \in ([A] \times [A]) \to [A]$**)** Recall the definition of append

$$\begin{array}{rcl} append([\,], M) & \stackrel{\text{def}}{=} & M \\ append((a{:}L), M) & \stackrel{\text{def}}{=} & a{:}(append(L, M)) \end{array}$$

Proof: We apply Corollary 2.1. Let A be an arbitrary set and let $B = [A]$ and $C = [A]$. Let h be the identity function on $[A] \to [A]$ and $g(x, m) = x{:}m$. Then $g \in (A \times [A]) \to [A]$. This fits the pattern and shows that append is a function of type $([A] \times [A]) \to [A]$.
□

In general, there is no need to apply the theorem or corollary, if a definition is given for an operator where the definition is presented by cases on a list argument and where the $[\,]$ case is not recursive. In that case, the operator can be shown to be a function. The idea is that at each recursive call, the length of the list argument is getting shorter and eventually will become $[\,]$, at that point the base case is invoked and there is no more recursion so it terminates.

Here are a few more functions defined by recursion on lists.

Definition 2.3 (List length)

$$\begin{array}{l} \mathit{length}([\,]) = 0 \\ \mathit{length}(x{:}xs) = 1 + \mathit{length}(xs) \end{array}$$

We will sometimes write $|L|$ instead of $\mathit{length}(L)$ in the following.

Definition 2.4 (List member)

$$\begin{array}{l} \mathit{mem}(y, [\,]) = \mathit{false} \\ \mathit{mem}(y, x{:}xs) = y = x \vee \mathit{mem}(y, xs) \end{array}$$

We will sometimes write $y \in M$ for $\mathit{mem}(y, M)$.

Definition 2.5 (List reverse)

$$\begin{array}{l} \mathit{rev}([\,]) = [\,] \\ \mathit{rev}(x{:}xs) = \mathit{rev}(xs) +\!\!+ [x] \end{array}$$

3 List Induction

The structural induction principle for lists is given as follows:

Definition 3.1 (List Induction) For a set A and a property P of $[A]$ we have the following axiom.

$$(P([\,]) \wedge \forall x :: A.\ \forall xs :: [A].\ P(xs) \Rightarrow P(x{:}xs)) \Rightarrow \forall ys :: [A].\ P(ys)$$

The corresponding proof rule is given as follows:

Proof Rule 3.1 (List Induction)

$$\frac{\Gamma \vdash P([\,]) \qquad \Gamma, x \in A, , xs \in [A], P(xs) \vdash P(x{:}xs)}{\Gamma \vdash \forall ys :: [A].\ P(ys)}\ ([A]\text{Ind}) \quad x, xs \text{ fresh.}$$

3.1 Some proofs by list induction

Def. 2.2 of the append function shows directly that $[\,]$ is a left identity for $+\!\!+$, is it a right identity as well? The following theorem establishes this fact.

Theorem 3.1 (Nil right identity for $+\!\!+$)

$$\forall ys :: [A].\ ys +\!\!+ [\,] = ys$$

Proof: By list induction on ys. The property P of ys is given as:

$$P(ys) \stackrel{\text{def}}{=} ys +\!\!+ [\,] = ys$$

Base Case: Show $P([\,])$, *i.e.* that $[\,] +\!\!+ [\,] = [\,]$. This follows immediately from the definition of append.

Induction Step: Assume $P(xs)$ (the induction hypothesis) and show $P(x{:}xs)$ for arbitrary $x \in A$ and arbitrary $xs \in [A]$. The induction hypothesis is:

$$xs +\!\!+ [\,] = xs$$

We must show $(x{:}xs) +\!\!+ [\,] = (x{:}xs)$. Starting with the left side of the equality we get the following:

$$(x{:}xs) +\!\!+ [\,] \stackrel{\langle\!\langle def.\,of\ +\!\!+ \rangle\!\rangle}{=} x{:}(xs +\!\!+ [\,]) \stackrel{\langle\!\langle ind.hyp. \rangle\!\rangle}{=} x{:}xs$$

So the induction step holds and the proof is complete.
□

Together, Def. 2.2 and Thm. 3.1 establish that $[\,]$ is a left and right identity with respect to append. Thus, with respect to $+\!\!+$, $[\,]$ behaves like zero does for ordinary addition.

The next theorem shows that append is associative.

Theorem 3.2 ($+\!\!+$ is associative)

$$\forall ys, zs, xs :: [A].\ xs +\!\!+ (ys +\!\!+ zs) = (xs +\!\!+ ys) +\!\!+ zs$$

Proof: Choose arbitrary $ys, zs \in [A]$. We continue by list induction on xs. The property P of xs is given as:

$$P(xs) \stackrel{\text{def}}{=} xs +\!\!+ (ys +\!\!+ zs) = (xs +\!\!+ ys) +\!\!+ zs$$

Base Case: Show $P([\,])$, *i.e.* that

$$[\,] +\!\!+ (ys +\!\!+ zs) = ([\,] +\!\!+ ys) +\!\!+ zs$$

On the left side:

$$[\,] +\!\!+ (ys +\!\!+ zs) \stackrel{\langle\!\langle def.\,of\ +\!\!+ \rangle\!\rangle}{=} (ys +\!\!+ zs)$$

On the right side:

$$([\,] +\!\!+\, ys) +\!\!+\, zs \overset{\langle\!\langle def.of +\!\!+ \rangle\!\rangle}{=} (ys +\!\!+\, zs)$$

So the base case holds.

Induction Step: For arbitrary $x \in A$ and $xs \in [A]$ we assume $P(xs)$ (the induction hypothesis) and show $P(x{:}xs)$.

$$xs +\!\!+\, (ys +\!\!+\, zs) = (xs +\!\!+\, ys) +\!\!+\, zs \qquad \text{Ind.Hyp.}$$

We must show

$$(x{:}xs) +\!\!+\, (ys +\!\!+\, zs) = ((x{:}xs) +\!\!+\, ys) +\!\!+\, zs$$

Starting on the left side:

$$(x{:}xs) +\!\!+\, (ys +\!\!+\, zs) \overset{\langle\!\langle def.of +\!\!+ \rangle\!\rangle}{=} x{:}(xs +\!\!+\, (ys +\!\!+\, zs)) \overset{\langle\!\langle Ind.Hyp. \rangle\!\rangle}{=} x{:}((xs +\!\!+\, ys) +\!\!+\, zs)$$

On the right side:

$$((x{:}xs) +\!\!+\, ys) +\!\!+\, zs \overset{\langle\!\langle def.of +\!\!+ \rangle\!\rangle}{=} (x{:}(xs +\!\!+\, ys)) +\!\!+\, zs \overset{\langle\!\langle def.of +\!\!+ \rangle\!\rangle}{=} x{:}((xs +\!\!+\, ys) +\!\!+\, zs)$$

So the left and right sides are equal and this completes the proof.

Here's an interesting theorem about reverse. We've seen this pattern before in Chapter **??** in Theorem **??** where we proved that the inverse of a composition is the composition of inverses.

Theorem 3.3 (Reverse of append)

$$\forall ys, xs :: [A].\ rev(xs +\!\!+\, ys) = rev(ys) +\!\!+\, rev(xs)$$

Proof: Choose an arbitrary $ys \in [A]$ and proceed by list induction on xs. Choose arbitrary $xs \in [A]$. The property P of xs is given as:

$$P(xs) \overset{\text{def}}{=} rev(xs +\!\!+\, ys) = rev(ys) +\!\!+\, rev(xs)$$

Base Case: Show $P([\,])$ *i.e.* that

$$rev([\,] +\!\!+\, ys) = rev(ys) +\!\!+\, rev([\,])$$

We start with the left side and reason as follows:

$$rev([\,] +\!\!+\, ys) \overset{\langle\!\langle def.of +\!\!+ \rangle\!\rangle}{=} rev(ys)$$

On the right side,

$$rev(ys) +\!\!+\, rev([\,]) \overset{\langle\!\langle def.of rev \rangle\!\rangle}{=} rev(ys) +\!\!+\, [\,] \overset{\langle\!\langle Thm\ 3.1 \rangle\!\rangle}{=} rev(ys)$$

So the left and right sides are equal and the base case holds.

Induction Step: For arbitrary $xs \in [A]$ assume $P(xs)$ and show $P(x{:}xs)$ for some arbitrary $x \in A$.

$$rev(xs +\!\!+ ys) = rev(ys) +\!\!+ rev(xs) \qquad \text{Ind.Hyp.}$$

We must show:

$$rev((x{:}xs) +\!\!+ ys) = rev(ys) +\!\!+ rev((x{:}xs))$$

We start with the left side.

$$\begin{array}{l} rev((x{:}xs) +\!\!+ ys) \\ \quad \overset{\langle\!\langle def.of +\!\!+ \rangle\!\rangle}{=} rev(x{:}(xs +\!\!+ ys)) \\ \quad \overset{\langle\!\langle def.of rev \rangle\!\rangle}{=} rev(xs +\!\!+ ys) +\!\!+ [x] \\ \quad \overset{\langle\!\langle Ind.Hyp. \rangle\!\rangle}{=} (rev(ys) +\!\!+ rev(xs)) +\!\!+ [x] \end{array}$$

On the right side:

$$\begin{array}{l} rev(ys) +\!\!+ rev((x{:}xs)) \\ \quad \overset{\langle\!\langle def.of rev \rangle\!\rangle}{=} rev(ys) +\!\!+ (rev(xs) +\!\!+ [x]) \\ \quad \overset{\langle\!\langle Thm\ 3.2 \rangle\!\rangle}{=} (rev(ys) +\!\!+ rev(xs)) +\!\!+ [x] \end{array}$$

□