Flat Lists, Induction and Recursion

Luke Hunsberger

February 10, 2009

1 Flat Lists

This section introduces *flat lists*—that is, lists that may contain any number of elements of any number of data types, but no subsidiary lists. Such lists in the syntax of Scheme would therefore include lists like `(1 a #t "hi!")`, but not lists like `(1 a (uh oh) 4)`, since the latter includes the subsidiary list, `(uh oh)`, as one of its elements. Lists that contain other lists will be addressed later on.

The presentation of flat lists has many parallels with the presentation of the natural numbers seen in a previous handout.

1.1 The Domain, $\mathcal{D}$, of List Contents

Often, it is useful to restrict the kinds of data that can appear as elements in a list. For example, we might want to restrict attention to lists of numbers or lists of symbols. In other cases, we might want to consider lists of numbers, symbols or strings. In this handout, we use $\mathcal{D}$ to denote the *set* of entities that can appear in our lists.[1] For example, if we want to consider lists whose elements are restricted to the integers, then $\mathcal{D}$ will be the set of integers. Similarly, if we want to consider lists whose elements are restricted to symbols, then $\mathcal{D}$ will be the set of symbols. Given the restriction that our lists are not allowed to contain subsidiary lists, we require that $\mathcal{D}$ *not* contain any lists. For convenience, we shall frequently refer to $\mathcal{D}$ as the *domain of list contents.*

1.2 The Definition of a Set of Lists

Given a domain of list contents, $\mathcal{D}$, as described above, we define, $\mathcal{L}_{\mathcal{D}}$, to be the smallest set that satisfies Axioms L_1–L_5, below. We call $\mathcal{L}_{\mathcal{D}}$ the set of lists with domain $\mathcal{D}$.

(L_1) The *empty list*, denoted by $\blacksquare$, is an an element of $\mathcal{L}_{\mathcal{D}}$.

(L_2) For each item $d \in \mathcal{D}$, and each list $\ell \in \mathcal{L}_{\mathcal{D}}$, there is a *constructed list* in $\mathcal{L}_{\mathcal{D}}$, denoted by $(d : \ell)$. We call d the *first* element of the constructed list, $(d : \ell)$; and we call ℓ the *rest* of the constructed list, $(d : \ell)$.

(L_3) No constructed list is equal to the empty list.

(L_4) For any items $d_1, d_2 \in \mathcal{D}$, and any lists $\ell_1, \ell_2 \in \mathcal{L}_{\mathcal{D}}$, if either $d_1 \neq d_2$ or $\ell_1 \neq \ell_2$, then $(d_1 : \ell_1) \neq (d_2 : \ell_2)$.

(L_5) Let P be any property for which the following conditions hold.

[1] Notice that $\mathcal{D}$ is an ordinary, mathematical set, not a list.

- $P(\blacksquare)$ holds.
- For any $d \in \mathcal{D}$ and any $\ell \in \mathcal{L}_\mathcal{D}$, if $P(\ell)$ holds, then $P(d : \ell)$ also holds.

Then $P(\ell)$ holds for every $\ell \in \mathcal{L}_\mathcal{D}$.

Axioms L_1–L_5 parallel Axioms P_1–P_5 in the handout on the natural numbers. Of course, there are some differences. Below, we consider each axiom, in turn.

First, recall that Axiom P_1 stipulated that the natural numbers includes an element called 0. Similarly, Axiom L_1 stipulates that the set of lists contains the null element, $\blacksquare$, called the empty list.

Second, recall that Axiom P_2 stipulated that every natural number has a *successor*. Thus, Axiom P_2 can be construed as a *constructor axiom*: it stipulates how additional natural numbers can be constructed from existing numbers. Similarly, Axiom L_2 stipulates the construction of additional lists from existing lists. In particular, given some list, ℓ, you can construct a new list by *prepending* an item $d \in \mathcal{D}$ onto the front of ℓ. The resulting list is denoted by $(d : \ell)$. The axiom also defines names for the constituents of this new list. In particular, d is called the *first* element of the new list, and ℓ is called the *rest* of the list. Notice that $d \in \mathcal{D}$, whereas $\ell \in \mathcal{L}_\mathcal{D}$. As we'll see shortly, the names *first* and *rest* can be thought of as *accessor* functions: they give access to the parts of a constructed list. Incidentally, given the successor, $Succ(n)$, of some natural number, n, we can think of the *predecessor* function, $Pred$, as an accessor function. In particular, $Pred(Succ(n)) = n$.

Third, recall that Axiom P_3 stipulated that zero is not the successor of any natural number. Similarly, Axiom L_3 stipulates that the empty list cannot be obtained by constructing a list of the form, $(d : \ell)$. In other words, the empty list is different from all of the constructed lists.

Fourth, recall that Axiom P_4 stipulated that different natural numbers have different successors. Similarly, Axiom L_4 stipulates that if the building blocks of two constructed lists differ in some way, then the resulting constructed lists are different. The building blocks can differ in two ways: d_1 might be different from d_2 or ℓ_1 might be different from ℓ_2. Of course, both pairs of building blocks might be different.

Fifth, recall that Axiom P_5 formed the basis for the proof procedure known as induction (over the natural numbers). Similarly, Axiom L_5 forms the basis for a proof procedure known as induction (over flat lists). The bulleted conditions in Axiom L_5 have the same general form as those in Axiom P_5. The main difference is that the second bulleted condition in Axiom L_5 includes the phrase, "for any $d \in \mathcal{D}$".

2 Proofs by Induction over Flat Lists

This section illustrates the process of proving theorems by induction over flat lists. The proof procedure is entirely analogous to that of induction over the natural numbers, except that we have to allow for the fact that the individual elements of a list can be arbitrary items in the domain $\mathcal{D}$.

Theorem 1. Every list in $\mathcal{L}_\mathcal{D}$ has one of two forms: either $\blacksquare$ or $(d : \ell)$ for some $d \in \mathcal{D}$ and some $\ell \in \mathcal{L}_\mathcal{D}$.

Proof. Let $P(\ell')$ be the property that ℓ' has the form $\blacksquare$ or $(d : \ell)$ for some $d \in \mathcal{D}$ and some $\ell \in \mathcal{L}_\mathcal{D}$. We want to show that $P(\ell')$ holds for all $\ell' \in \mathcal{L}_\mathcal{D}$.

- Consider $P(\blacksquare)$. It holds since $\blacksquare$ has the form $\blacksquare$.
- Suppose $P(\ell_1)$ holds for some list ℓ_1. Let d_1 be some arbitrary element of $\mathcal{D}$. We must show that $P(d_1 : \ell_1)$ also holds. But $(d_1 : \ell_1)$ certainly has the form $(d : \ell)$ for some $d \in \mathcal{D}$ and some $\ell \in \mathcal{L}_\mathcal{D}$.

Thus, by Axiom L_5, $P(\ell')$ holds for all $\ell' \in \mathcal{L}_\mathcal{D}$.

2.1 Recursive Functions over Flat Lists

Theorem 1 states that a list in $\mathcal{L}_{\mathcal{D}}$ has one of the following forms: either $\blacksquare$ or $(d : \ell)$. Thus, we can define a function, f, on *all* lists, as follows:

- Define $f(\blacksquare)$.
- Define $f(d : \ell)$ in terms of d and $f(\ell)$.

Defining $f(\blacksquare)$ is the *base case* of the definition. Defining $f(d : \ell)$ in terms of d and $f(\ell)$ is the *recursive case* of the definition. It is recursive because the definition of $f(d : \ell)$ depends on $f(\ell)$ being defined.

Theorem 2. If a function, f, is defined as described above, then it is defined for all lists in $\mathcal{L}_{\mathcal{D}}$.

Proof. Let $P(\ell)$ be the proposition that $f(\ell)$ is defined. The proof by induction is left as an exercise. (Use Theorem 2 in the previous handout as a guide.)

3 Some Recusively Defined Functions on Flat Lists

These section shows how a variety of recursive functions can be defined on flat lists. In each case, it gives a *math-like* definition and a Scheme-based definition. The Scheme definitions make use of the following *built-in* functions:

- The `cons` list-constructor function. This function takes an item $d \in \mathcal{D}$ and some list $\ell \in \mathcal{L}_{\mathcal{D}}$ as its inputs. It returns as its output the list $(d : \ell)$. Of course, the syntax of Scheme makes things look a little different, as demonstrated by the following drScheme session:

```
> (cons 3 '(a b c))
(3 a b c)
> (cons '+ '(1 2 3))
(+ 1 2 3)
> (cons 1 ())
(1)
```

- The `first` and `rest` accessor functions. These functions provide access to the parts of a constructed list, as demonstrated below.

```
> (first '(1 2 3))
1
> (first '(+ #t a))
+
> (rest '(1 2 3))
(2 3)
> (rest '(1))
()
> (rest '(+ #t a #f))
(#t a #f)
```

  It should be kept in mind that these functions will generate errors if they are applied to the empty list! They should only be applied to constructed lists!

  The following drScheme session illustrates that that `cons` is *sort of* the *dual* of `first` and `rest`.

```
> (define listy '(a b c d e))
> (cons (first listy) (rest listy))
(a b c d e)
> (first (cons 'z listy))
z
> (rest (cons 'z listy))
(a b c d e)
```

- The `null?` type-checker predicate. This predicate returns boolean *true* if its input is the empty list; otherwise, it returns boolean *false*, as illustrated below.

```
> (null? ())
#t
> (null? '(a b c d))
#f
> (null? (cons 1 '(a b c)))
#f
```

  Notice that an expression of the form, `(null? (cons ... ...))`, invariably evaluates to boolean *false*.

The Length of a List. The length of a list (i.e., the number of elements contained in the list) can be recursively defined, as follows:

- $f(\blacksquare) = 0$
- $f(d : \ell) = 1 + f(\ell)$

In Scheme, the definition looks like this:[2]

```
(define lengthy
  (lambda (listy)
    (if (null? listy)
      0
      (+ 1 (lengthy (rest listy))))))
```

It behaves as follows:

```
> (lengthy ())
0
> (lengthy '(a b c d e))
5
```

Summing the Elements in a List. The sum of the elements in a list can be recursively defined, as follows.

- $g(\blacksquare) = 0$
- $g(d : \ell) = d + g(\ell)$

In Scheme, it looks like this:

[2]Scheme includes a built-in function associated with the `length` symbol that computes the length of a list.

```
> (define summy
    (lambda (listy)
      (if (null? listy)
          0
          (+ (first listy)
             (summy (rest listy))))))
> (summy '(1 2 3))
6
> (summy ())
0
> (summy '(1 10 100 1000))
1111
```

Notice the use of indentation to make the Scheme definition easier to look at. For example, the expresssions, `(first listy)` and `(summy (rest listy))`, which are both inputs to the `+` operator, are indented the same amount.

Squaring the Elements in a List. In this case, we want a function that takes a list of numbers as its input. It should return a list of the same length as its output, where each entry in the output list should be the square of the corresponding entry from the input list, as demonstrated below:

```
> (square-all '(1 2 3 4 5))
(1 4 9 16 25)
> (square-all '(10 20 30 100))
(100 400 900 10000)
```

Such a function can be recursively defined as follows.

- $h(\blacksquare) = \blacksquare$
- $h(d : \ell) = (d * d : h(\ell))$

Notice that if this function is given the empty list as its input, it returns the empty list as its output. However, if it is given a constructed list as its input, it returns a constructed list as its output. In either case, the output is a list.

Here's the Scheme definition:

```
(define square-all
  (lambda (listy)
    (if (null? listy)
        ()
        (cons (* (first listy) (first listy))
              (square-all (rest listy))))))
```

Determining Whether an Item is a Member of a List. Given an item, x, and a list, ℓ, we desire a function, F, to return *true* if x is a member of ℓ; *false* otherwise. Notice that this function takes two inputs: an item and a list. However, we can still define it recursively:

- $F(x, \blacksquare) = false$
- $F(x, (d : \ell)) = true$ if $x = d$ or $F(x, \ell) = true$

Here's a Scheme definition:[3]

```
> (define memb
    (lambda (item listy)
      (if (null? listy)
          #f
          (or (= item (first listy))
              (memb item (rest listy))))))
> (memb 3 '(1 2 3 4 5))
#t
> (memb 3 '(1 2 4 5 6))
#f
```

Notice the use of the `or` special form instead of using a second `if` expression. This makes the code easier to read. We can take this one step further, as follows:

```
> (define memby
    (lambda (item listy)
      (and (not (null? listy))
           (or (= item (first listy))
               (memby item (rest listy))))))
> (memby 3 '(1 2 3 4 5))
#t
> (memby 3 '(1 2 4 5 6))
#f
```

You should walk through the code to convince yourself that it effectively implements the same function.

4 Proving that a Recursive Function has some Property

Below, the definitions of the *length* function, f, and the *square-all* function, h, are repeated.

(1) $f(\blacksquare) = 0$

(2) $f(d : \ell) = 1 + f(\ell)$

(3) $h(\blacksquare) = \blacksquare$

(4) $h(d : \ell) = (d * d : h(\ell))$

Notice that the clauses in these definitions have been numbered for easy reference.

Theorem 3. Let $\mathcal{D}$ be the set of natural numbers. For each $\ell \in \mathcal{L}_{\mathcal{D}}$, the length of ℓ is the same as the length of $h(\ell)$. Stated more concisely: $f(\ell) = f(h(\ell))$.

[3]There is a built-in Scheme function associated with the `member` symbol that does essentially the same thing as the `memb` function defined here.

Proof. Let $P(\ell)$ be the proposition that $f(\ell) = f(h(\ell))$. We want to show that $P(\ell)$ holds for all $\ell \in \mathcal{L}_{\mathcal{D}}$.

- $P(\blacksquare)$ is: $f(\blacksquare) = f(h(\blacksquare))$. Starting with the right-hand side, notice that $f(h(\blacksquare)) = f(\blacksquare)$ since $h(\blacksquare) = \blacksquare$, by clause (2). Thus, $P(\blacksquare)$ holds.

- Suppose d is some element of $\mathcal{D}$ and ℓ is some element of $\mathcal{L}_{\mathcal{D}}$ for which $P(\ell)$ holds (i.e., $f(\ell) = f(h(\ell))$). We need to show that $P(d : \ell)$ holds (i.e., that $f(d : \ell) = f(h(d : \ell))$ holds). Starting with the right-hand side:

  Right-hand side: $f(h(d : \ell))$
  $= f(d * d : h(\ell))$, by (4)
  $= 1 + f(h(\ell))$, by (2)
  $= 1 + f(\ell)$, since $P(\ell)$ holds
  $= f(d : \ell)$, by (2)
  $=$ Left-hand side!

Thus, by Axiom L_5, $P(\ell)$ holds for all $\ell \in \mathcal{L}_{\mathcal{D}}$.