

1 Review

Last time we proved various theorems about `rev`.

For example, recall:

```
(defun true-listp (x)
  (if (endp x)
      (equal x nil)
      (true-listp (cdr x))))

(defun app (x y)
  (if (endp x)
      y
      (cons (car x) (app (cdr x) y))))

(defun rev (x)
  (if (endp x)
      nil
      (app (rev (cdr x)) (list (car x)))))

(implies (true-listp x) (equal (rev (rev x)) x))

(equal (rev (rev (rev x))) (rev x))

(true-listp (rev x))
```

2 Today: Induction Principle Unplugged

From a practical point of proof, `rev` is unsatisfactory. Try evaluating it on long lists and you will notice that the running time is worse than you might naively expect. Why? Well, let's try analyzing the running time. Since `rev` is a recursive function, we have a recurrence relation. If we look at the recursive call, we see that when `rev` is given a list of length n , it calls `rev` with a list of length $n - 1$ and it takes $O(n)$ work to finish the recursive case because of the call to `append`. So, we have:

$$T(n) = n + T(n - 1)$$

so $T(n) = O(n^2)$.

Can't we do better? Yes we can. Here is the idea, shown with an example. Imagine reversing the list

(1 2 3 4)

where the only basic operations we have are `car` and `cdr`. Then, we can decompose the list into

1, (2 3 4)

Now, one more step gives us three things:

1, 2, (3 4)

What can we do with them? Well, what about consing 2 onto (1)? That gives us:

(2 1), (3 4)

One more time:

(3 2 1), (4)

And finally:

(4 3 2 1), ()

So, only four steps, which is linear. But, this required remembering what elements of the list we've already seen, and `rev` doesn't have that information. So, this indicates that we need an accumulator, to remember this. Using the design recipe template, we get:

```
(defun revt (x)
  (rev-tail x nil))

(defun rev-tail (x a)
  (if (endp x)
      ...
      (rev-tail (cdr x) ... (car x) ... a ...)))
```

Now, let's look at the recursive case first. Remember that the accumulator needs to remember all the elements we have seen so far, so let's cons the new element `((car x))` onto `a`. That gives us

```
(defun rev-tail (x a)
  (if (endp x)
```

```

...
(rev-tail (cdr x) ... (cons (car x) a) ...))

```

Now, as we've seen that already is the (partial) reverse of the list. That is, the accumulator is the reverse of the part of the list processed so far. That is our accumulator invariant. So, what's the base case? We wind up with:

```

(defun rev-tail (x a)
  (if (endp x)
      a
      (rev-tail (cdr x) (cons (car x) a))))

```

Now, let's prove the following:

φ : (equal (rev1 x nil) (rev x))

Is it true?

Can we solve this with equational reasoning? No. Why not?

What does the induction principle give us?

(endp x) $\Rightarrow \varphi$

and

(consp x) $\wedge \varphi((x (cdr x))) \Rightarrow \varphi$

Let's try to prove this.

Proof?

The base case is simple. Here is an attempt at proving the induction step.

My context is:

1. (consp x) and 2. (equal (rev1 (cdr x) nil) (rev (cdr x)))

Proof

```

(rev1 x nil)
= { By 1 and the definition of rev1 }
  (rev1 (cdr x) (cons (car x) nil))

```

But, now what? Our induction hypothesis tells us something about (rev1 (cdr x) nil), but we really need to know something about (rev1 (cdr x) (cons (car x) nil)), so we're stuck. The point is that when we expand (rev1 x nil), we get an expression that is of the form (rev1 ... (cons ...)). The point is that the second argument is not nil, but any way of instantiating the theorem we want to prove (as we do to get the induction hypothesis) will give us a nil in the second argument.

We need a handle on that second argument, but we're not going to get it if the theorem we want to prove has a nil there; we need a variable. We call this a generalization step because we are now going to prove a theorem about (rev1 x y), whereas before we were proving a theorem about a special case of the above, namely about (rev1 x nil). But, now we have to figure out what the new theorem we want to prove is.

φ : (equal (rev1 x y) ???)

What is ????. Think about the role of `y` in the definition of `rev1`. The accumulator corresponds to a partial result: it should be the reverse of the elements of the original argument to `rev1` that we have seen already, so we wind up with:

φ : (equal (rev1 x y) (append (rev x) y))

The base case is simple. Here is a proof of the induction step. First, our context is:

1. (consp x)
2. $\varphi((x \text{ cdr } x))$

Here is the proof. It starts the same way as before.

Proof

(rev1 x y)
 = { By 1 and the definition of rev1 }
 (rev1 (cdr x) (cons (car x) y))

But, now we have a problem. The induction hypothesis doesn't match the above, since, as stated it is (rev1 (cdr x) y) = (append (rev (cdr x)) y) and we don't want a `y` as the second argument of `rev1`; we want (cons (car x) y). But, the induction principle only constrains the substitution we use for the induction hypothesis to map `x` to (cdr x). It can do anything else with any other variables. So, our context can be:

1. (consp x)
2. $\varphi((x \text{ cdr } x) (y \text{ cons } (car \ x) \ y))$

So now, we can prove the theorem as follows.

Proof

(rev1 x y)
 = { By 1 and the definition of rev1 }
 (rev1 (cdr x) (cons (car x) y))
 = { By IH }
 (append (rev (cdr x)) (cons (car x) y))
 = { Def of append }
 (append (rev (cdr x)) (append (list (car x)) y))
 = { Associativity of append }
 (append (append (rev (cdr x)) (list (car x))) y)
 = { Def of rev, hyp 1 }
 (append (rev x) y) \square