Peter Dillinger

## Generalization

When proving theorems by induction, it turns out that how we state what we want to prove can significantly impact how the proof goes, including how difficult it is.

For an example, recall LEN,

```
(defun len (x)
  (if (endp x)
    0
    (+ 1 (len (cdr x)))))
```

and the accumulator version, LEN-AC:

```
(defun len-ac (x i)
  (if (endp x)
    (+ 0 i)  ; make sure it returns a number
    (len-ac (cdr x) (+ i 1))))
```

We can use this to define a drop-in replacement for LEN, LEN2:

```
(defun len2 (x)
  (len-ac x 0))
```

And if everything works as we think it should, then LEN2 should return the same thing as LEN. In other words, this should be a theorem:

```
(equal (len2 x)
       (len x))
```

How do we prove this? How about induction? We have just one variable, so we should probably induct based on (len x):

```
(implies (endp x)            ; The base case is easy to prove
         (equal (len2 x)
                (len x)))

(implies (and (consp x)      ; Induction step
              (equal (len2 (cdr x))
                     (len (cdr x))))
         (equal (len2 x)
                (len x)))
```

For the induction step, we start with

```
 (len2 x)
= { Def len2 }
 (len-ac x 0)
= { Def len-ac, (consp x) }
 (len-ac (cdr x) 1)
```

If we start from the other side,

```
 (len x)
= { Def len }
 (+ 1 (len (cdr x)))
= { I.H. }                (That is, "Induction Hypothesis")
 (+ 1 (len2 (cdr x)))
= { Def len2 }
 (+ 1 (len-ac (cdr x) 0))
```

So now we need to know that (len-ac (cdr x) 1) equals (+ 1 (len-ac (cdr x) 0)). But we can’t prove this without another inductive proof.

Conceptually, it seems we needed to induct based on len-ac for the preceding proof to go through in one induction, but the original proposition didn't even have more than one variable in it. (Inducting based on len-ac requires two variables for induction.)

Let us reconsider the original conjecture:

```
 (equal (len2 x)
        (len x))
= { Def len2 }
 (equal (len-ac x 0)
        (len x))
```

That directly mentions len-ac, the most complicated function in what we're proving, but we only have one variable. Can we induct based on (len-ac x 0)? That would mean we replace x with (cdr x) and 0 with (+ 0 1). As in instantiation, we are only allowed to replace variables in an induction hypothesis. This means if we want to induct based on len-ac, we need two variables to induct over, and we probably want to replace that 0 with a variable.

If I change the left hand side of the equality with (len-ac x i), how do I need to modify the rest to keep the statement true?

One way would be

```
  (implies (equal y 0)
           (equal (len-ac x y)
                  (len x)))
```

If we attempt to prove this by induction, the Induction Hypothesis Chaining step requires

```
  (implies (equal y 0)
           (equal (+ 1 y) 0))
```

which means we would never be able to use the induction hypothesis, making such a proof by induction useless.

Instead we want the conjecture to be more general, applying to any y. To make that work, we have to add y to (len x):

```
  (equal (len-ac x y)
         (+ y (len x)))
```

You may recall us proving this just a few lectures ago. After induction based on (len-ac x y), it is rather simple.

Let's now confirm that this is a generalization of the original:

```
  (equal (len-ac x y)
         (+ y (len x)))
=> { Instantiation with y = 0 }
  (equal (len-ac x 0)
         (+ 0 (len x)))
=> { Arithmetic (given LEN returns integers), Def len2 }
  (equal (len2 x)
         (len x))
```

A is a generalization of B if B is easy to prove from A but A is not easy to prove from B. (This is not a precise mathematical defintion, but it is good enough for us.) In essence, A will say the same thing as B but about more cases. Usually, there are infinitely many cases in which A says something interesting but B does not.

For example, the generalization above says something about all calls to
LEN-AC, while the original only applies to cases in which the second parameter
is 0.

Let’s consider another example:

```
  (equal (len (app x x))
         (* 2 (len x)))
```

If we attempt to prove this with an induction scheme based on one variable,
as in that based on (len x), the induction step causes trouble:

```
  (implies (and (consp x)
                (equal (len (app (cdr x) (cdr x)))
                       (* 2 (len (cdr x)))))
           (equal (len (app x x))
                  (* (len x))))
```

We would start with

```
 (len (app x x))
= { Def app }
 (len (cons (car x) (app (cdr x) x)))
= { Def len }
 (+ 1 (len (app (cdr x) x)))
```

So we have (len (app (cdr x) x)) but the induction hypothesis is about
(len (app (cdr x) (cdr x))).

This turns out to be another theorem that is easier to prove if we generalize
it first.  Rather than being restricted to the length of appending
something with itself, why not consider the length of appending any two things?
the generalization goes like this:

```
 (equal (len (app x x))
        (* 2 (len x)))
<= { Arith, given LEN returns integers }
 (equal (len (app x x))
        (+ (len x) (len x)))
<= { Instantiate y with x }
 (equal (len (app x y))
        (+ (len x) (len y)))
```

And that is a proposition we have proven before.  It is rather simple by
induction based on (len x) or (app x y), same scheme in either case.